

UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO DE JOINVILLE
CURSO DE ENGENHARIA MECATRÔNICA

ABNER GABRIEL DE SOUZA

TESTE DE SOFTWARE CONCORRENTE ORIENTADO POR MODELO FORMAL

Joinville
2026

ABNER GABRIEL DE SOUZA

TESTE DE SOFTWARE CONCORRENTE ORIENTADO POR MODELO FORMAL

Trabalho apresentado como requisito parcial para obtenção do título de bacharel em Engenharia Mecatrônica, no Curso de Engenharia Mecatrônica, do Centro Tecnológico de Joinville, da Universidade Federal de Santa Catarina.

Orientador: Dr. Gian Ricardo Berkenbrock

Joinville
2026

Este trabalho é dedicado aos meus colegas de classe e aos meus queridos pais e
irmã.

“ A natureza é um enorme jogo de xadrez disputado por deuses, e que temos o privilégio de observar. As regras do jogo são o que chamamos de física fundamental, e compreender essas regras é a nossa meta. ”

Richard Phillips Feynman

RESUMO

Sistemas multitarefa são sistemas que executam diversas tarefas concorrentemente, e estão presentes em sistemas operacionais, computação de alto desempenho, servidores e sistemas embarcados. Pelo fato de as tarefas serem concorrentes, defeitos como condição de corrida, deadlock e inanição podem ocorrer, e dependem da ordem específica em que o escalonador intercala as tarefas em cada execução. Tais defeitos escapam aos testes convencionais, uma vez que uma implementação incorreta pode produzir, em uma execução isolada, a mesma saída de uma implementação correta, o que torna insuficiente a simples comparação entre saída obtida e saída esperada. Este trabalho desenvolve um sistema de testes para software multitarefa, utilizando um modelo formal em UPPAAL como oráculo de verificação. O desenvolvimento partiu do levantamento de requisitos e da modelagem do estudo de caso em UPPAAL, com a verificação das propriedades temporais relevantes em lógica temporal CTL. Em seguida, foi implementado em Python um framework de caixa-preta que compila e executa a implementação sob teste — ou consome diretamente os registros de execução que ela produz — captura esses registros e os confronta com o espaço de estados derivado do modelo. A consistência do carregador foi validada por comparação cruzada com os rastros oficiais gerados pelo verificador do UPPAAL. O jantar dos filósofos foi adotado como estudo de caso por reunir os três fenômenos de interesse em um modelo de tamanho tratável.

Palavra-chave: Teste de software multitarefa; Verificação baseada em modelo UPPAAL Programação concorrente Análise de rastros

ABSTRACT

Multitasking systems execute several tasks concurrently and are present in operating systems, high-performance computing, servers, and embedded systems. Because these tasks run concurrently, defects such as race conditions, deadlock, and starvation may arise, and they depend on the specific order in which the scheduler interleaves the tasks in each execution. Such defects escape conventional testing, since an incorrect implementation may produce, in an isolated execution, the same output as a correct one, which makes a simple comparison between the obtained output and the expected output insufficient. This work develops a testing system for multitasking software, using a formal UPPAAL model as a verification oracle. The development began with requirements elicitation and the modeling of the case study in UPPAAL, followed by the verification of the relevant temporal properties expressed in CTL (Computation Tree Logic). Next, a framework was implemented in Python that compiles the implementation under test, runs the binary (one or more times, as configured), captures the execution traces, and confronts them with the state space derived from the model. The consistency of the loader was validated through cross-comparison with the official traces generated by the UPPAAL verifier. The dining philosophers problem was adopted as the case study because it brings together the three phenomena of interest in a model of tractable size.

Keywords: Multitasking software testing; Model-based verification UPPAAL
Concurrent programming Trace analysis

LISTA DE FIGURAS

Figura 1 – Concorrência <i>versus</i> paralelismo na execução de tarefas.	14
Figura 2 – Fluxo da verificação de modelo (<i>model checking</i>).	16
Figura 3 – Exemplo de autômato temporizado: a lâmpada (<i>Lamp</i>) e o usuário (<i>User</i>), sincronizados pelo canal <i>press</i>	18
Figura 4 – Os operadores $A[]$ e $E<>$ sobre a árvore de computação.	18
Figura 5 – Rastro de execução: sequência ordenada de estados de uma execução.	19
Figura 6 – Teste de caixa-preta: o sistema sob teste é avaliado apenas pelas saídas observáveis, sem acesso à sua estrutura interna.	20
Figura 7 – Exemplo de máquina de estados finita: estados, transição inicial e estado final (duplo círculo).	21
Figura 8 – Processo de pesquisa adotado no desenvolvimento do trabalho. . .	23
Figura 9 – Jantar dos filósofos.	25
Figura 10 – Estrutura do repositório.	28
Figura 11 – Visão geral do pipeline de teste por verificação de registros de execução.	30
Figura 12 – Autômato <i>Filosofo</i> no editor do UPPAAL.	31
Figura 13 – Autômato <i>Garfo</i> no editor do UPPAAL.	31
Figura 14 – Resultado da verificação das propriedades no UPPAAL.	33
Figura 15 – Abstração de caso de estudo: a descrição do caso parametriza o orquestrador e a pipeline.	34
Figura 16 – Carregamento do modelo: o XML do UPPAAL é convertido no espaço de estados.	35
Figura 17 – Projeção: cada estado formal é mapeado para uma linha observável no formato CSV.	36
Figura 18 – Compilação e execução: da implementação em C++ ao registro de execução.	37
Figura 19 – Verificação por propriedades: o registro de execução e o espaço de estados produzem o veredito por propriedade.	37
Figura 20 – Geração de relatórios e visualização comparativa a partir dos vereditos.	38
Figura 21 – Validação cruzada: os rastros do UPPAAL confirmam o espaço de estados construído pelo carregador.	39
Figura 22 – Correspondência entre as categorias de propriedades CTL do modelo (Quadro 6) e as <i>checks</i> do framework (Quadro 8).	43

Figura 23 – Cobertura do espaço de estados por implementação. As barras de 17852 e 17857 estão vazias porque nenhuma das execuções produziu registro de execução reconhecido pelo executor.	44
Figura 24 – Legenda de cores dos grafos de comparação entre registro de execução e modelo.	46
Figura 25 – Visão geral do registro de execução da referência: progressão dos estados seguintes.	47
Figura 26 – Detalhe (zoom) do registro de execução da referência, com os próximos estados possíveis segundo o modelo em cinza tracejado. . .	47
Figura 27 – Contagem por filósofo no registro de execução da implementação de referência.	48
Figura 28 – Visão geral do registro de execução da submissão 17853: parte de estados válidos (azul) e, a partir do passo 6, atinge estados fora do modelo (vermelho).	49
Figura 29 – Detalhe (zoom) da submissão 17853 no ponto da divergência: o modelo admite os estados em cinza (próximos possíveis), mas a execução vai a um estado fora do espaço alcançável (vermelho). . .	50
Figura 30 – Saída real do sistema ao avaliar a submissão 17853: o veredito, os estados inválidos detectados e a pontuação são produzidos automaticamente a partir do registro de execução.	50
Figura 31 – Detalhe (zoom) da submissão 17864 no ponto da divergência: o modelo admite os estados em cinza (próximos possíveis), mas a execução vai a um estado fora do espaço alcançável (vermelho), em que um filósofo permanece comendo sem segurar garfos. . . .	51
Figura 32 – Visão geral do registro de execução da submissão 17856: progressão dos estados seguintes.	52
Figura 33 – Detalhe (zoom) do início do registro de execução da submissão 17856, com os próximos estados possíveis segundo o modelo em cinza tracejado.	52
Figura 34 – Contagem por filósofo no registro de execução da submissão 17856.	53

LISTA DE QUADROS

Quadro 1 – Saída de duas execuções do programa.	15
Quadro 2 – Entrelaçamento que provoca a condição de corrida.	16
Quadro 3 – Ferramentas utilizadas no desenvolvimento.	24
Quadro 4 – Requisitos funcionais do sistema de testes e seções que os atendem.	27
Quadro 5 – Variáveis globais declaradas no modelo UPPAAL.	32
Quadro 6 – Propriedades CTL verificadas no modelo UPPAAL.	32
Quadro 7 – Alfabeto dos registros de execução.	33
Quadro 8 – Propriedades verificadas no estudo de caso.	42
Quadro 9 – Resultados por implementação. A indica propriedade aprovada; R, reprovada.	48

SUMÁRIO

1	INTRODUÇÃO	11
1.1	OBJETIVOS	12
1.1.1	Objetivo geral	12
1.1.2	Objetivos Específicos	12
2	FUNDAMENTAÇÃO TEÓRICA	13
2.1	SISTEMAS MULTITAREFA	13
2.1.1	Problemas de sincronização	13
2.2	VERIFICAÇÃO FORMAL	15
2.2.1	Autômatos temporizados	16
2.2.2	Lógica temporal computacional	17
2.2.3	Rastros de execução	19
2.3	TESTES DE SOFTWARE	19
2.3.1	Teste de caixa-preta	20
2.3.2	Máquinas de estado finito	20
2.4	TESTE BASEADO EM MODELO	21
2.5	TRABALHOS RELACIONADOS	22
3	MATERIAIS E MÉTODO	23
3.1	ETAPAS DO DESENVOLVIMENTO	23
3.2	FERRAMENTAS UTILIZADAS	24
3.3	ESTUDO DE CASO ADOTADO	24
3.4	ESTRATÉGIA DE VALIDAÇÃO	25
3.5	REQUISITOS DO SISTEMA DE TESTES	26
3.6	VISÃO GERAL DO SISTEMA	26
3.7	ARQUITETURA DO SISTEMA	27
3.8	MODELO FORMAL	29
3.9	GERAÇÃO E FORMATO DOS REGISTROS DE EXECUÇÃO	32
3.10	FRAMEWORK DE VERIFICAÇÃO	34
3.10.1	Abstração de caso de estudo	34
3.10.2	Carregamento do modelo formal	35
3.10.3	Projeção do estado formal	35
3.10.4	Compilação e execução do programa sob teste	36
3.10.5	Verificação por propriedades	37
3.10.6	Geração de relatórios	38
3.10.7	Validação cruzada do carregador	38
3.11	EXTENSIBILIDADE	39
4	RESULTADOS E DISCUSSÃO	41

4.1	ESTUDO DE CASO: JANTAR DOS FILÓSOFOS	41
4.2	IMPLEMENTAÇÕES AVALIADAS	43
4.3	COBERTURA DO ESPAÇO DE ESTADOS	44
4.4	DETECÇÃO DE PROPRIEDADES VIOLADAS	45
4.5	LIMITAÇÕES OBSERVADAS	53
5	CONCLUSÃO	55
5.1	ESCLARECIMENTOS	57
	REFERÊNCIAS	58

1 INTRODUÇÃO

Sistemas computacionais modernos executam, em regra, múltiplas tarefas concorrentemente, sejam elas processos de um sistema operacional, requisições de um servidor web ou malhas de controle de um sistema embarcado. A capacidade de coordenar essas tarefas para maximizar o uso dos recursos disponíveis é o que caracteriza um software multitarefa, e o projeto é um dos temas centrais da engenharia de software (Buttazzo, 2011).

A concorrência, no entanto, introduz uma classe de defeitos que não ocorre em programas estritamente sequenciais. Os principais são condições de corrida, impasses (*deadlocks*) e inanição (*starvation*). Esses defeitos são reconhecidamente difíceis de detectar por testes convencionais, porque dependem da ordem específica em que o escalonador do sistema operacional intercala as tarefas, ordem essa que pode variar a cada execução (Lamport, 2009).

A consequência prática do não-determinismo associado à concorrência é que um programa pode passar por dezenas de execuções aparentemente corretas e falhar na enésima, em um cenário que não se reproduz facilmente. Em ambientes industriais, esse comportamento traduz em defeitos contínuos que escapam às baterias de teste tradicionais e só se manifestam em produção, frequentemente sob condições de carga difíceis de reproduzir em laboratório. Por conta disso, técnicas que verificam além da execução repetida em casos de teste vêm sendo investigadas há décadas. Entre elas, destaca-se a verificação formal, em que o comportamento esperado é descrito matematicamente e propriedades de interesse podem ser verificadas sobre todas as execuções possíveis (Baier; Katoen, 2008). A integração entre teste de software e verificação formal abre o caminho para uma classe de ferramentas de teste em que a verdade sobre uma execução deixa de depender de oráculos construídos manualmente caso a caso.

Observar uma saída de execução não basta para distinguir uma implementação concorrente correta de uma implementação bem sucedida por acidente. Programas com falhas de sincronização podem produzir, em uma execução, o mesmo resultado de programas corretos, ainda que sujeito a deadlock ou inanição sob outro escalonamento. As ferramentas de teste que se baseiam apenas na comparação de saídas com um valor esperado são, portanto, insuficientes para programas concorrentes.

Automatizar testes de programas concorrentes exige um oráculo independente da execução observada. A verificação formal oferece esse oráculo: um modelo matemático do problema descreve quais comportamentos são aceitáveis, e qualquer registro de execução pode ser confrontado com esse modelo. Construir um sistema

de testes que torne essa ideia prática — capaz de produzir um veredito para qualquer implementação que respeite uma interface de saída — contribui para a área de testes de software multitarefa e ajuda a garantir que software concorrente, do controle de qualidade industrial ao apoio ao ensino, se comporte conforme o esperado.

1.1 OBJETIVOS

Para abordar a problemática apresentada, este trabalho define o seguinte objetivo geral e os objetivos específicos correspondentes.

1.1.1 Objetivo geral

Desenvolver um sistema de testes para software multitarefa, capaz de verificar softwares de forma caixa-preta, a partir de um modelo formal e da análise dos registros de execução por eles produzidos.

1.1.2 Objetivos Específicos

- Levantar os requisitos funcionais do sistema de testes.
- Modelar formalmente um problema clássico de sincronização adotado como estudo de caso.
- Implementar a infraestrutura de compilação, execução e captura de registros de execução do programa sob teste.
- Implementar a comparação dos registros de execução contra o espaço de estados derivado do modelo formal.
- Validar o sistema com implementações de referência, verificando se o veredito atribuído reflete a presença ou ausência das propriedades esperadas.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo reúne os conceitos necessários para compreender as decisões de projeto adotadas no desenvolvimento. A seção 2.1 apresenta sistemas multitarefa e os problemas de sincronização. A seção 2.2 introduz verificação formal, autômatos temporizados e lógica temporal. A seção 2.3 discute teste de software, com foco no teste de caixa-preta. A seção 2.4 contextualiza o trabalho na área de teste baseado em modelo.

2.1 SISTEMAS MULTITAREFA

Sistemas multitarefa são definidos como sistemas capazes de executar diversas tarefas de forma concorrente, alternando controladamente entre elas.

Os recursos do sistema são compartilhados entre as tarefas para dessa forma ampliar o desempenho e simular a execução paralela, o que dá a impressão de um sistema mais ágil (Buttazzo, 2011). Esse paradigma é predominante em sistemas operacionais, servidores, computação de alto desempenho e sistemas embarcados. A Figura 1 contrasta os dois casos: na concorrência, as tarefas se intercalam em um único núcleo; no paralelismo, executam simultaneamente em núcleos distintos.

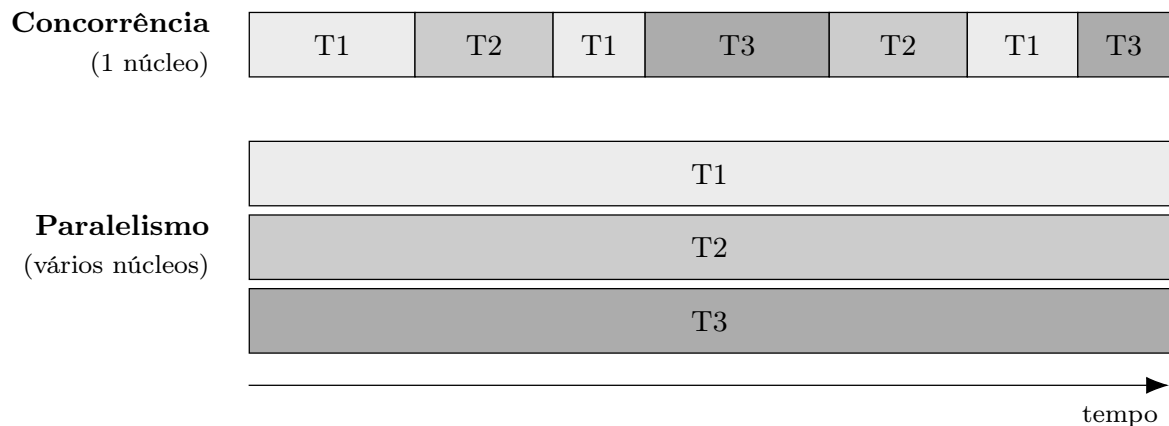
Segundo Harrow (2000, p. 331), aplicações multitarefas são notoriamente difíceis de projetar e construir sem defeitos. A literatura distingue dois tipos principais de sistemas multitarefa: os cooperativos, em que cada tarefa controla por quanto tempo retém o processador e libera voluntariamente; e os preemptivos, em que o escalonador do sistema operacional pode interromper uma tarefa a qualquer momento para conceder o processador a outra (Buttazzo, 2011).

Os sistemas preemptivos têm uma abordagem mais dominante em sistemas operacionais modernos porque garantem responsividade e evitam que uma tarefa mal desenvolvida monopolize o processador. O preço dessa flexibilidade é que o conjunto de intercalações possíveis entre as tarefas cresce de forma combinada, e qualquer defeito de sincronização passa a ser dependente do escalonamento específico observado em cada execução (Henard; et al., 2016).

2.1.1 Problemas de sincronização

Os principais problemas de sincronização em sistemas concorrentes são a condição de corrida, o deadlock e a inanição. A condição de corrida ocorre quando duas ou mais tarefas acessam uma variável compartilhada simultaneamente, e o resultado depende da ordem em que os acessos foram intercalados. O deadlock ocorre quando duas ou mais tarefas aguardam indefinidamente uma pela outra, e nenhuma

Figura 1 – Concorrência *versus* paralelismo na execução de tarefas.



Fonte: Autor (2026).

consegue prosseguir (Coffman *et al.*, 1971). A inanição ocorre quando uma tarefa, embora apta a executar, nunca recebe acesso aos recursos de que precisa porque outras tarefas continuamente os obtêm primeiro (Tanenbaum; Bos, 2016).

Um exemplo clássico que reúne os três fenômenos é o problema do jantar dos filósofos, proposto por Edsger Dijkstra (Dijkstra, 1971). Cinco filósofos sentam ao redor de uma mesa redonda; entre cada par de filósofos há um único garfo compartilhado. Para se alimentar, um filósofo precisa adquirir os dois garfos adjacentes; caso contrário, permanece com fome. Quando todos os filósofos pegam o garfo da esquerda simultaneamente, instala-se um impasse clássico, em que nenhum consegue adquirir o segundo garfo (Buttazzo, 2011).

A condição de corrida ocorre quando duas ou mais tarefas acessam uma variável compartilhada e o resultado depende da ordem em que os acessos são intercalados. O Código 2.1 exemplifica o problema: duas threads incrementam a mesma variável `a` sem qualquer sincronização. O incremento `++counter` não é atômico, pois se decompõe em ler o valor, somar um e escrever de volta; quando o escalonador troca de tarefa entre a leitura e a escrita, um dos incrementos é sobrescrito e se perde.

Sem perdas, o valor final deveria ser 20001: a variável parte de 1 e recebe 2000 incrementos por iteração ao longo de dez iterações. O Quadro 1 compara duas execuções do programa: em uma o contador chega aos 20001 esperados, enquanto na outra termina em 19732 — 269 incrementos se perdem, e a divergência surge na quinta iteração e se mantém até o fim. Esse não-determinismo é a marca da condição de corrida: o defeito não se manifesta em toda execução, mas apenas quando a troca de contexto cai entre a leitura e a escrita de um incremento.

O Quadro 2 detalha, para um único incremento, o entrelaçamento que provoca a perda. As duas tarefas leem o mesmo valor antes que qualquer uma escreva; por

Listagem 2.1 – Programa com condição de corrida.

```

1 #include <iostream>
2 #include <thread>
3
4 void increment(int& counter, int numIncrements) {
5     for (int i = 0; i < numIncrements; ++i) {
6         ++counter; // incremento nao sincronizado
7     }
8 }
9
10 int main() {
11     int a = 1;
12     const int qntIncrements = 1000;
13     const int iterations = 10;
14
15     for (int iteration = 0; iteration < iterations; ++iteration) {
16         std::thread first(increment, std::ref(a), qntIncrements);
17         std::thread second(increment, std::ref(a), qntIncrements);
18
19         first.join();
20         second.join();
21
22         std::cout << "Valor do contador: " << a << std::endl;
23     }
24     return 0;
25 }

```

Quadro 1 – Saída de duas execuções do programa.

Iteração	Sem corrida	Com corrida
1	2001	2001
2	4001	4001
3	6001	6001
4	8001	8001
5	10001	9732
6	12001	11732
7	14001	13732
8	16001	15732
9	18001	17732
10	20001	19732

Fonte: Autor (2026).

isso a última escrita sobrescreve a anterior e um incremento desaparece.

Repetido ao longo de milhares de incrementos concorrentes, esse mesmo entrelaçamento explica os incrementos perdidos na execução do Código 2.1.

2.2 VERIFICAÇÃO FORMAL

A verificação formal é o conjunto de técnicas que permite provar, com base em uma especificação matemática do sistema, que determinadas propriedades são

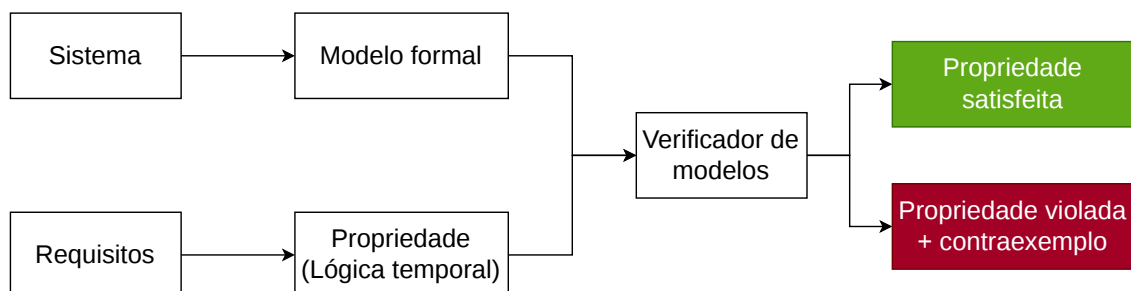
Quadro 2 – Entrelaçamento que provoca a condição de corrida.

Tarefa	Ação	a
A	lê a em um registrador (leu 5)	5
→ <i>troca de contexto: A é interrompida antes de escrever</i>		5
B	lê a em um registrador (leu 5)	5
B	soma e escreve $a = 6$	6
→ <i>troca de contexto: B termina, A retoma</i>		6
A	escreve $a = 6$ (valor lido antes da troca)	6

Fonte: Autor (2026).

satisfeitas em todas as execuções possíveis. Diferentemente dos testes convencionais, que examinam um número finito de execuções, a verificação formal examina o conjunto completo dos comportamentos descritos pela especificação (Baier; Katoen, 2008).

Dentro das técnicas de verificação formal, a verificação de modelo (*model checking*) é destaque pela automação. Dado um modelo finito do sistema e uma propriedade temporal, um verificador de modelo decide se a propriedade é satisfeita e, em caso negativo, produz um contraexemplo concreto que pode ser inspecionado (Hartonas-Garmhausen *et al.*, 1996). A Figura 2 resume esse fluxo, adequado a sistemas concorrentes, em que a explicação textual de uma falha raramente é suficiente.

Figura 2 – Fluxo da verificação de modelo (*model checking*).

Fonte: Adaptado de Baier e Katoen (2008).

2.2.1 Autômatos temporizados

Um autômato temporizado é um autômato finito estendido com um conjunto de *relógios*: variáveis de valor real que avançam continuamente e à mesma taxa, medindo a passagem do tempo (Alur; Dill, 1994). O autômato é formado por um conjunto finito de *localizações*, uma delas inicial, e por *transições* entre elas. Cada transição pode ser condicionada por uma *guarda* — uma restrição sobre os relógios e sobre variáveis discretas — e pode *reiniciar* relógios ao ser disparada. A cada

localização pode ainda ser associada uma *invariante*, que limita por quanto tempo o sistema pode nela permanecer e, assim, força o progresso (Baier; Katoen, 2008).

O estado de um autômato temporizado é o par formado pela localização atual e os valores dos relógios. A partir dele, o sistema evolui de dois modos: por uma transição de *atraso*, em que apenas o tempo passa e todos os relógios aumentam igualmente, respeitando a invariante da localização; ou por uma transição *discreta*, em que uma aresta habilitada por sua guarda é percorrida, possivelmente reiniciando relógios (Alur; Dill, 1994).

Sistemas concorrentes são modelados como uma *rede* de autômatos temporizados que executam em paralelo e se sincronizam por meio de *canais*: quando um autômato emite por um canal e outro o recebe, as duas transições disparam em conjunto. Variáveis globais compartilhadas completam o modelo, de modo que o estado global do sistema é dado pela combinação das localizações de todos os autômatos e dos valores dessas variáveis.

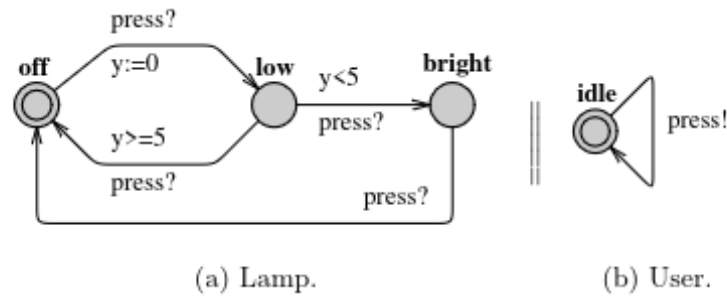
Neste trabalho, esse formalismo é instrumentalizado pela ferramenta UPPAAL, que implementa redes de autômatos temporizados e acrescenta recursos úteis à modelagem: canais *broadcast* (um emissor sincroniza com todos os receptores prontos), localizações *committed* (que devem ser deixadas imediatamente, garantindo a atomicidade de uma sequência de transições) e variáveis inteiras limitadas (Behrmann *et al.*, 2004; UPPAAL, 2024). O UPPAAL foi adotado por oferecer editor gráfico, verificador automático de propriedades em CTL e exportação direta dos rastros de execução, artefatos consumidos diretamente pelo restante do sistema. O estudo de caso aqui tratado emprega o fragmento discreto desse formalismo: a coordenação entre filósofos e garfos é expressa com localizações, canais e variáveis booleanas, sem restrições temporais sobre relógios.

A Figura 3 ilustra esses elementos em um exemplo clássico do próprio tutorial do UPPAAL: uma lâmpada cuja intensidade depende da rapidez dos toques. O relógio y mede o tempo desde o último toque; um novo toque com $y < 5$ leva a lâmpada de `low` a `bright`, ao passo que $y \geq 5$ a desliga, e cada toque reinicia o relógio com $y := 0$. A lâmpada e o usuário são dois autômatos que disparam suas transições em conjunto pela sincronização no canal `press` — a mesma ideia de sincronização por canais empregada no modelo deste trabalho.

2.2.2 Lógica temporal computacional

A Lógica Temporal Computacional (CTL) é a linguagem em que as propriedades verificadas pelo UPPAAL são expressas. Ela combina operadores temporais (X (Next), F (Future), G (Global), U (Until)) com quantificadores sobre caminhos (A para "em todos os caminhos" e E para "existe um caminho"). Duas combinações são particularmente relevantes para este trabalho: $A[] \varphi$, lida como "em todos os estados de

Figura 3 – Exemplo de autômato temporizado: a lâmpada (*Lamp*) e o usuário (*User*), sincronizados pelo canal *press*.



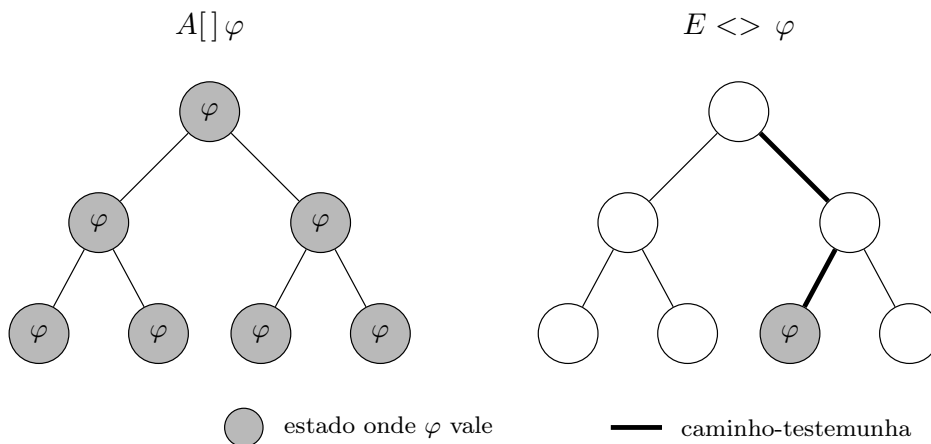
Fonte: Behrmann *et al.* (2004).

todos os caminhos vale φ ", e $E<>\varphi$, lida como "existe um caminho que eventualmente leva a um estado em que vale φ " (Clarke *et al.*, 1986; Baier; Katoen, 2008).

Propriedades de segurança, como ausência de deadlock, são tipicamente expressas com $A[]$; propriedades de vivacidade, como a possibilidade de cada filósofo eventualmente comer, são expressas com $E<>$. A relevância prática dessas expressões para o trabalho é que cada propriedade do modelo UPPAAL admite uma tradução direta para uma função de verificação sobre registros de execução, conforme detalhado na seção 3.6.

A Figura 4 ilustra os dois operadores sobre a árvore de computação: em $A[]\varphi$, φ vale em todos os estados de todos os caminhos (propriedade de segurança); em $E<>\varphi$, basta existir um caminho que eventualmente atinja um estado onde φ vale (alcançabilidade).

Figura 4 – Os operadores $A[]$ e $E<>$ sobre a árvore de computação.



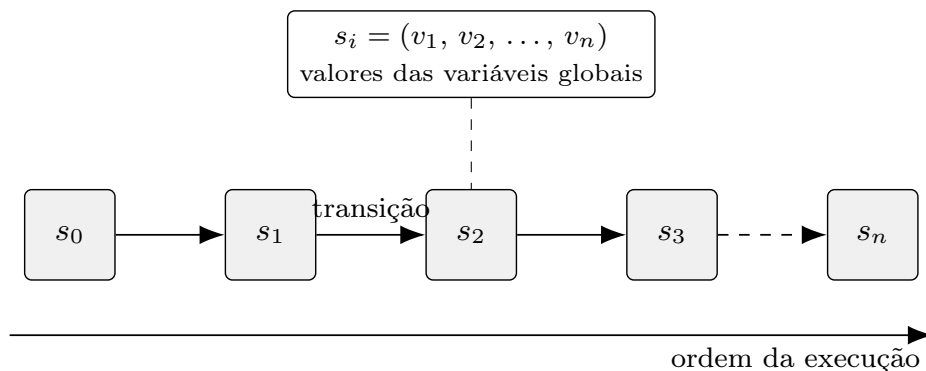
Fonte: Adaptado de Baier e Katoen (2008).

2.2.3 Rastros de execução

Rastros de execução, também chamados de *traces*, são sequências ordenadas de estados ou eventos observados durante a execução de um sistema concorrente. Cada trace é uma evidência observável de uma das execuções possíveis do sistema. A coleção de todos os traces descreve o comportamento de um sistema; já a verificação de propriedades pode ser reformulada como a verificação de propriedades sobre rastros (Leucker; Schallhart, 2009).

A Figura 5 ilustra a ideia: um rastro é a sequência ordenada de estados $s_0, s_1, \dots, s_n$ observada em uma execução, em que cada estado é uma captura das variáveis globais e cada par consecutivo corresponde a uma transição.

Figura 5 – Rastro de execução: sequência ordenada de estados de uma execução.



Fonte: Autor (2026).

Quando um trace é gerado a partir de um modelo formal, ele tem a garantia de pertencer ao espaço de estados especificado. Caso o trace seja capturado da execução de um programa real, essa garantia precisa ser verificada: cada par consecutivo de estados deve corresponder a uma transição existente no modelo, e o estado final deve admitir um caminho para que os estados desejados pelas propriedades de vivacidade sejam alcançados. Essa correspondência é o mecanismo do sistema proposto.

2.3 TESTES DE SOFTWARE

Testes de software constituem o conjunto de atividades destinadas a verificar se um programa atende aos seus requisitos. Eles podem ser classificados, quanto ao grau de conhecimento que o testador tem do código em avaliação, em três grandes categorias: caixa-branca, caixa-cinza e caixa-preta (Desikan; Ramesh, 2006; Delamaro *et al.*, 2007).

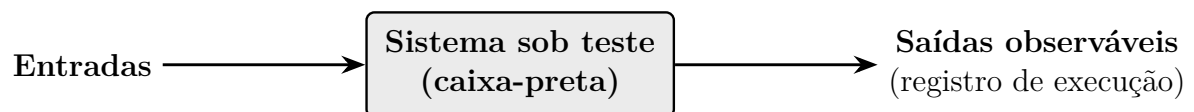
Em testes de caixa-branca, o sistema testador tem acesso completo ao código-fonte e pode projetar casos de testes baseados na estrutura interna do programa. Em

testes de caixa-cinza, o acesso parcial, o sistema conhece interfaces internas, mas não a implementação completa. Em teste de caixa-preta, conhecem-se apenas a entrada aceita pelo programa e as saídas esperadas, e com isso julga-se o programa exclusivamente pelo comportamento observável (Melo; et al., 2019).

2.3.1 Teste de caixa-preta

O teste de caixa-preta é particularmente adequado a cenários em que o sistema não pode ou não deve depender de detalhes internos da implementação. Esse é o caso de sistemas de teste genéricos, desenvolvidos para avaliar qualquer implementação que respeite uma interface bem definida: como diferentes implementações resolvem o mesmo problema de formas estruturalmente distintas, qualquer técnica de avaliação que se apoie na estrutura interna teria de ser refeita para cada nova implementação (Heimdahl; et al., 2004). A Figura 6 ilustra essa ideia: o sistema é tratado como uma caixa-preta, julgada apenas por suas saídas observáveis.

Figura 6 – Teste de caixa-preta: o sistema sob teste é avaliado apenas pelas saídas observáveis, sem acesso à sua estrutura interna.



Fonte: Autor (2026).

A abordagem adotada neste trabalho é de caixa-preta. O programa sob teste é tratado como uma função opaca que produz um registro de execução; a fonte de verdade é construída exclusivamente sobre esse registro de execução, sem inspeção do código-fonte. Essa decisão tem dois efeitos práticos: torna o sistema agnóstico ao estilo da implementação avaliada e permite que ele seja estendido a qualquer outro problema de sincronização cuja especificação possa ser expressa como uma máquina de estados.

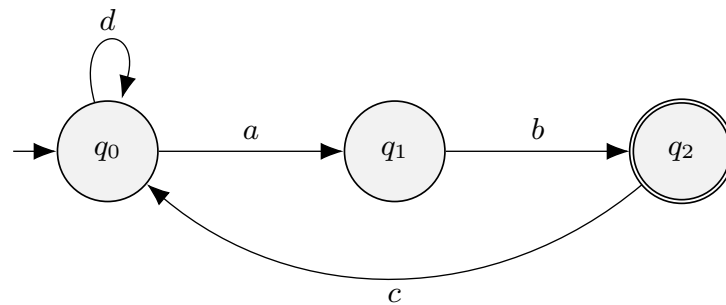
2.3.2 Máquinas de estado finito

Uma máquina de estados finita (MEF) é um modelo formal composto por um conjunto finito de estados, um estado inicial, um alfabeto de eventos e uma relação de transição entre estados. A cada instante, a máquina ocupa exatamente um dos estados; um evento de entrada provoca uma transição controlada para outro estado, possivelmente acompanhada de uma ação de saída (Lee; Yannakakis, 1996).

As MEFs são adequadas para a representação de sistemas concorrentes porque permitem enumerar explicitamente o espaço de comportamentos possíveis

(Chow, 1978). Quando o número de estados é finito e tratável, um verificador de modelos consegue percorrer o espaço completo e decidir se as propriedades de interesse são satisfeitas. O modelo UPPAAL do estudo de caso adotado neste trabalho é, em essência, uma MEF estendida com variáveis globais e canais de sincronização. A Figura 7 ilustra uma MEF: a cada evento (a , b , c , d) a máquina transita entre os estados q_0 , q_1 e q_2 .

Figura 7 – Exemplo de máquina de estados finita: estados, transição inicial e estado final (duplo círculo).



Fonte: Autor (2026).

2.4 TESTE BASEADO EM MODELO

Teste baseado em modelo (*Model-Based Testing*, MBT) é uma abordagem em que casos de teste, oráculos ou ambos são derivados automaticamente de um modelo formal do comportamento esperado do sistema (Utting; Legeard, 2007; Neto *et al.*, 2007). A motivação do MBT é eliminar a etapa manual de construção de oráculos para cada caso de teste, etapa essa que costuma ser complexa e suscetível a erros.

No contexto de softwares concorrentes, o MBT tem um desafio adicional: o modelo precisa descrever não somente as saídas esperadas para cada entrada, mas também o conjunto de comportamentos possíveis ao longo do tempo. Modelos baseados em autômatos, como os autômatos do UPPAAL, são adequados a esse propósito porque o espaço de estados pode ser percorrido por um verificador automático, e os traces obtidos podem servir tanto de referência como de base para comparação com os registros de execução emitidos pelo programa em testes (Utting *et al.*, 2012).

Este trabalho está na interseção entre teste baseado em modelo e o teste de caixa-preta sobre sistemas concorrentes. A contribuição é a construção de um sistema de testes genérico e de **caixa-preta**: dado um modelo formal em UPPAAL como oráculo, ele avalia o programa sob teste exclusivamente pelos **registros de execução** que este produz, sem qualquer acesso ao seu código-fonte. Esses registros de execução podem ser obtidos de duas formas: compilando e executando uma implementação (em C++, no estudo de caso) ou recebendo diretamente os registros de

execução já gerados por ela — inclusive produzidos em outra máquina ou ferramenta. A saída é um veredito sobre a conformidade da execução com o modelo.

2.5 TRABALHOS RELACIONADOS

O teste de programas concorrentes reúne abordagens diversas, acumuladas em décadas de pesquisa. Arora *et al.* (2016) revisam sistematicamente a área e a organizam em oito categorias — entre elas o teste baseado em modelo e os métodos formais — levantando técnicas, ferramentas e lacunas das abordagens existentes. Este trabalho situa-se justamente na interseção dessas duas categorias: deriva um oráculo de um modelo formal e o aplica, como teste baseado em modelo, sobre os registros de execução produzidos pelo programa sob teste.

Dentro do teste baseado em modelo a partir de autômatos temporizados, aproxima-se deste trabalho a ferramenta DTRON (Anier *et al.*, 2017), que realiza teste baseado em modelo distribuído de aplicações de tempo crítico a partir de especificações em UPPAAL. A diferença central está no modo de uso do modelo: o DTRON gera e executa estímulos de teste em tempo de execução, sincronizando componentes distribuídos, ao passo que o sistema aqui proposto não gera casos de teste — ele consome o registro de execução emitido por uma implementação qualquer e o confronta, *dos efeitos às causas*, com o espaço de estados derivado do modelo, atuando como oráculo de conformidade em caixa-preta.

Quanto à detecção de defeitos específicos de concorrência, Cai *et al.* (2018) propõem o projeto de suítes de teste voltadas a faltas de controle de concorrência em sistemas de transações distribuídas. Embora compartilhe o objetivo de revelar defeitos de sincronização, aquela abordagem é dirigida a um domínio específico e baseada na geração de casos de teste; o presente trabalho, por outro lado, é genérico — parametrizável por estudo de caso — e fundamenta o veredito na comparação de registros de execução completos contra o modelo, sem inspeção do código-fonte.

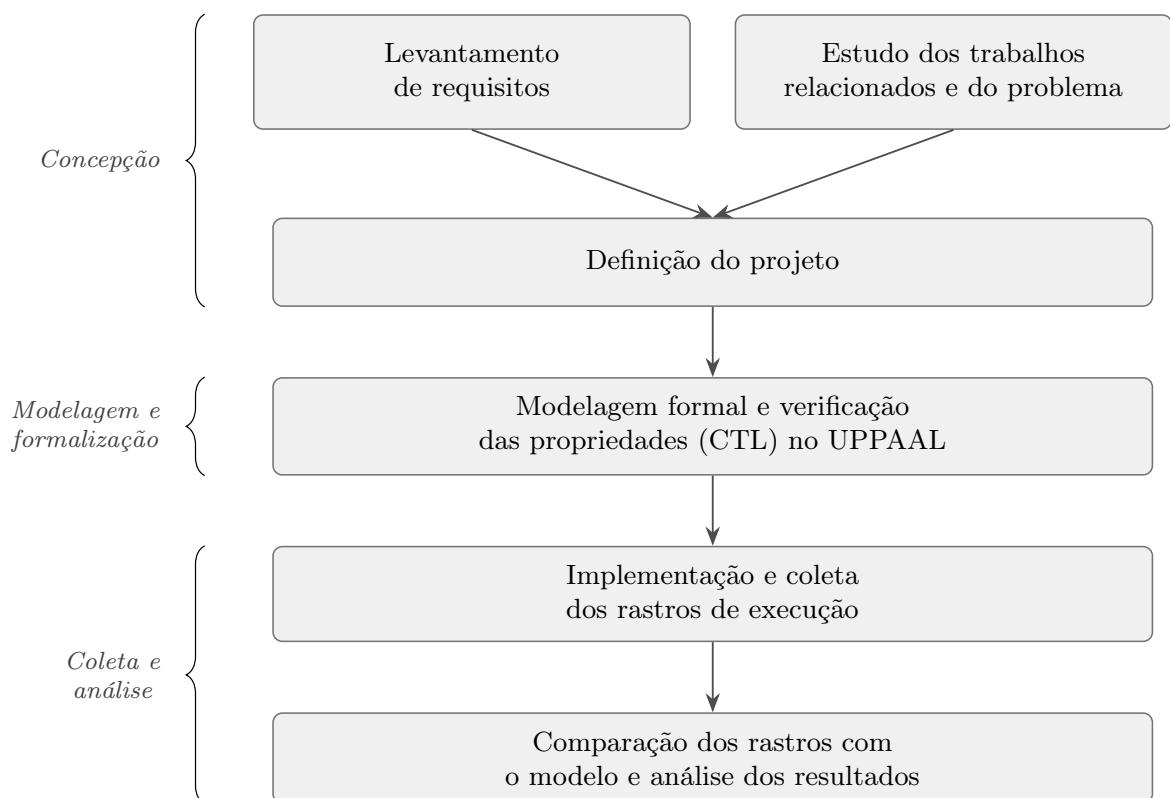
Em síntese, o trabalho posiciona-se na interseção entre teste baseado em modelo e teste de caixa-preta de software concorrente: em vez de gerar testes ou monitorar propriedades pontuais, propõe um oráculo genérico que valida o registro de execução completo contra o espaço de estados de um modelo em UPPAAL, produzindo um veredito ponderado por propriedade.

3 MATERIAIS E MÉTODO

Esta pesquisa é classificada como aplicada, uma vez que visa à construção de um artefato que resolve um problema prático. Quanto ao método, foi adotada a estratégia de estudo de caso, em que um único problema clássico de sincronização, o Jantar dos filósofos, é modelado, implementado e utilizado para validar o sistema de testes. Quanto à abordagem, a pesquisa combina elementos qualitativos, na construção de um modelo e na análise de falhas detectadas, com elementos quantitativos, na agregação de vereditos produzidos pelo sistema.

3.1 ETAPAS DO DESENVOLVIMENTO

Figura 8 – Processo de pesquisa adotado no desenvolvimento do trabalho.



Fonte: Autor (2026).

A Figura 8 resume o processo de pesquisa adotado, da concepção do projeto à coleta e análise dos dados; as etapas são detalhadas a seguir. A concepção do trabalho combinou duas frentes conduzidas em paralelo: o levantamento de requisitos, que delimitou o escopo do sistema de testes e cujo resultado está consolidado na Seção 3.5, e o estudo dos trabalhos relacionados e do problema, apoiado em uma

revisão da literatura sobre teste de software multitarefa e teste baseado em modelo. Dessas duas frentes resultou a definição do projeto, que orientou as etapas seguintes.

A primeira delas foi a modelagem formal do estudo de caso em UPPAAL, incluindo a verificação das propriedades temporais de interesse (CTL) com o verificador automático. A segunda foi a implementação e coleta dos rastros de execução, que compreendeu o programa sob teste em C++, empregando `std::thread` e `std::mutex`, a definição do formato de registro de execução emitido pelo binário em sua saída padrão, e o framework de verificação em Python, organizado em módulos de compilação, execução, carregamento do modelo, verificação por propriedades e geração de relatórios. A terceira foi a comparação dos rastros com o modelo e a análise dos resultados, incluindo a validação do sistema com implementações de referência, descrita na Seção 3.4.

3.2 FERRAMENTAS UTILIZADAS

O Quadro 3 sintetiza as ferramentas e tecnologias empregadas no desenvolvimento, agrupadas por etapa.

Quadro 3 – Ferramentas utilizadas no desenvolvimento.

Etapas	Ferramenta
Modelagem formal	UPPAAL 4.1
Verificação de propriedades	<i>verifyta</i> (CLI UPPAAL)
Programa sob teste	C++20, g++, <code>std::thread</code>
Sistema de testes	Python 3.x, <i>dataclasses</i>
Visualização	Graphviz
Versionamento	Git

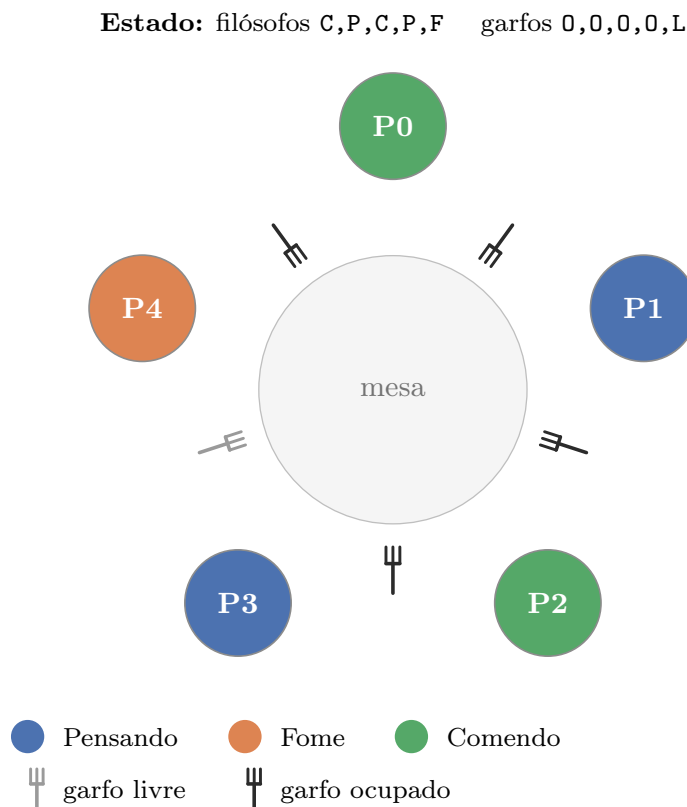
Fonte: Autor (2026).

3.3 ESTUDO DE CASO ADOTADO

O problema do jantar dos filósofos foi adotado como estudo de caso por três razões. A primeira é que ele reúne, em um problema compacto, os três fenômenos de interesse: condição de corrida sobre o estado dos garfos, deadlock no caso de os filósofos pegarem os garfos de forma simétrica e vivacidade no caso de políticas de acesso injustas. A segunda razão é a especificação formal caber em um modelo do UPPAAL de tamanho moderado, com cinco filósofos e cinco garfos, é suficiente para gerar um espaço de estados rico mas tratável. A terceira é que o problema é amplamente conhecido na literatura de sistemas concorrentes, dessa forma facilitando a interpretação de resultados e comparação com trabalhos relacionados.

A Figura 9 ilustra o problema: cinco filósofos sentam ao redor de uma mesa e, entre cada par de vizinhos, há um garfo compartilhado. Para comer, um filósofo precisa adquirir os dois garfos adjacentes; caso contrário, permanece com fome. Cada estado do sistema é codificado por dez tokens: os cinco primeiros descrevem os filósofos e os cinco últimos os garfos. No estado mostrado na figura (C,P,C,P,F e 0,0,0,0,L), os filósofos P0 e P2, não-adjacentes, comem em paralelo, enquanto P4 permanece com fome porque um de seus garfos já está ocupado.

Figura 9 – Jantar dos filósofos.



Fonte: Autor (2026).

3.4 ESTRATÉGIA DE VALIDAÇÃO

Para validar o sistema, foram conduzidos experimentos com implementações de referência, projetadas para testar cada propriedade verificada. Uma implementação correta, que adota a política de aquisição ordenada dos garfos, deve ter um veredito de conformidade integral. Implementações com problemas introduzidos intencionalmente, por exemplo, aquisição simultânea que induz deadlock, ou inversão de prioridade que induz inanição, devem ter o veredito reduzido exatamente pelas propriedades violadas pelo erro. O Capítulo 4 apresenta os resultados desses experimentos.

3.5 REQUISITOS DO SISTEMA DE TESTES

O levantamento de requisitos foi a primeira etapa do desenvolvimento (Seção 3.1) e delimitou o escopo do sistema a partir de duas fontes. A primeira é a lacuna identificada na revisão da literatura (Capítulo 2): a necessidade de um oráculo que não dependa da execução observada nem da estrutura interna da implementação, mas que seja derivado de um modelo formal do comportamento esperado, na linha do teste baseado em modelo (Seção 2.4) e do teste de caixa-preta (Seção 2.3.1). A segunda são as aplicações pretendidas para o sistema — a avaliação em lote de múltiplas implementações de um mesmo problema de sincronização. Desses dois eixos derivam os requisitos funcionais consolidados no Quadro 4, cuja última coluna indica a seção em que cada requisito é atendido, estabelecendo a rastreabilidade entre o que o sistema deve fazer e como foi construído.

Além dos requisitos funcionais, três restrições não funcionais guiaram o projeto. A primeira é a **modularidade**: cada etapa da pipeline é um módulo independente, de responsabilidade única (Seção 3.10), o que viabiliza o atendimento do RF11. A segunda é a **reprodutibilidade**: a execução ocorre em diretórios isolados e sob versionamento, e os artefatos de cada caso são resolvidos por convenção de diretórios. A terceira é a **legibilidade do carregador**, que motivou restringir deliberadamente o subconjunto da linguagem UPPAAL suportado (Seção 3.11). Esses requisitos orientam a arquitetura descrita nas seções seguintes.

3.6 VISÃO GERAL DO SISTEMA

A construção foi organizada em três camadas independentes: o modelo formal em UPPAAL, que define o comportamento esperado; o programa sob teste em C++, que materializa a implementação a ser avaliada; e um framework em Python, que orquestra compilação, execução, captura de registros de execução e comparação contra o modelo. As próximas seções detalham cada uma dessas camadas e o estudo de caso utilizado para validação.

A organização do repositório separa os artefatos por tipo no nível mais externo (`models/`, `traces/`, `cpp/`, `python/`) e, dentro de cada um, por caso de estudo (Figura 10).

Essa separação foi adotada para que a inclusão de um novo caso não exija modificações no orquestrador, bastando depositar os artefatos nas pastas correspondentes e declarar um caso de estudo em código, conforme descrito na Seção 3.10.

Quadro 4 – Requisitos funcionais do sistema de testes e seções que os atendem.

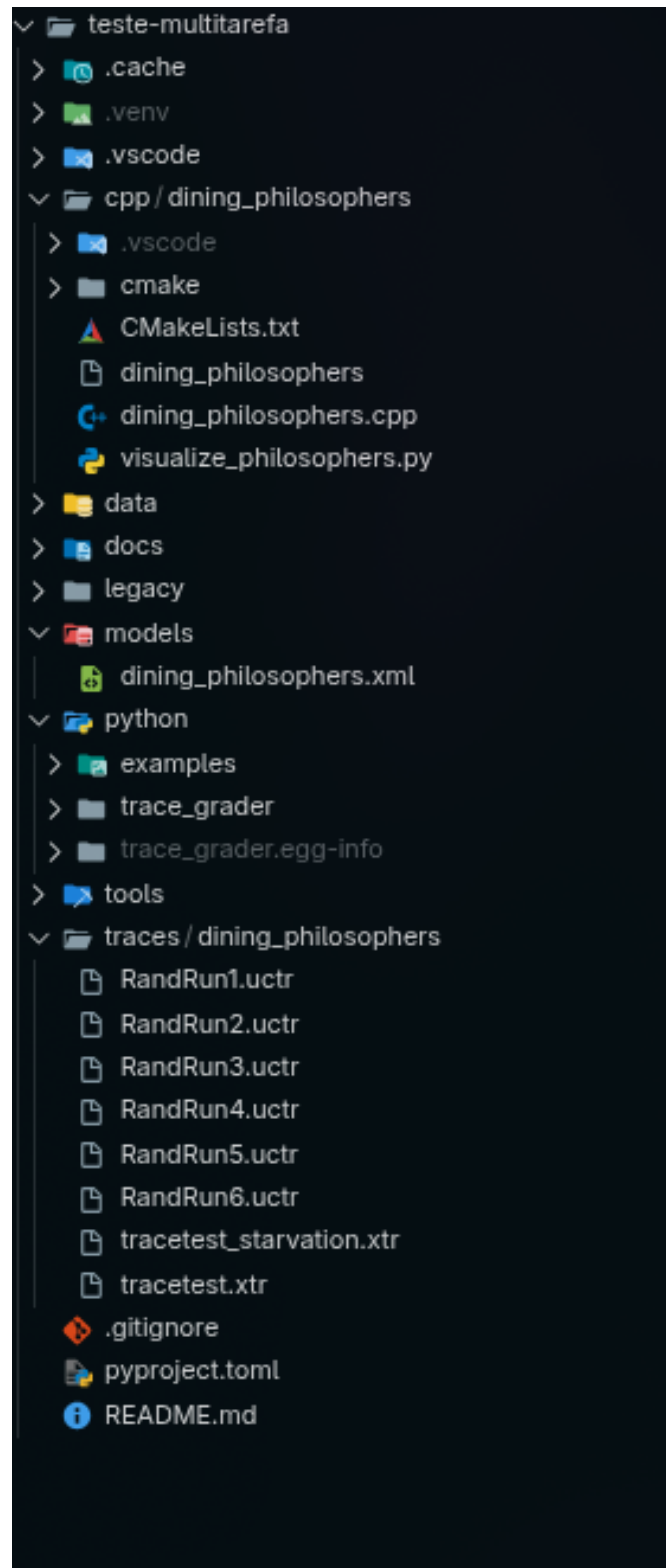
ID	Requisito funcional	Seção
RF01	Avaliar implementações em qualquer linguagem, desde que produzam o registro de execução no formato canônico esperado: no modo direto o sistema consome esse registro sem reexecutar o programa, independentemente da linguagem ou ferramenta que o gerou.	3.7
RF02	Avaliar o programa sob teste de forma estritamente caixa-preta, sem inspeção do código-fonte.	2.3.1
RF03	Quando recebe código-fonte, compilar e executar a implementação automaticamente (g++/C++, no estudo de caso), tratando o sucesso da compilação como pré-condição da avaliação.	3.10.4
RF04	Executar a implementação em diretório isolado e sob limite de tempo, capturando o registro de execução produzido.	3.10.4
RF05	Normalizar registros de execução de origens e formatos distintos (arquivo em disco ou saída padrão) para um formato canônico único.	3.9
RF06	Derivar o oráculo do modelo formal, carregando o XML do UPPAAL e construindo o espaço de estados observável (estados e transições).	3.10.2
RF07	Confrontar o registro de execução com o espaço de estados e as propriedades derivadas do modelo, emitindo um veredito por propriedade e a lista de erros.	3.10.5
RF08	Calcular uma pontuação agregada ponderada de conformidade a partir dos vereditos por propriedade.	3.10.6
RF09	Gerar relatórios por implementação e um consolidado, em formato legível e em formato estruturado (Markdown e JSON).	3.10.6
RF10	Produzir uma visualização comparativa entre o espaço de estados do modelo e o subconjunto efetivamente visitado pelo registro de execução.	3.10.6
RF11	Ser genérico: isolar a lógica específica do problema em uma descrição de caso de estudo (modelo, projeção, formato do registro e propriedades), com orquestrador agnóstico, e permitir incluir novos casos sem alterar o núcleo.	3.10.1, 3.11
RF12	Validar a fidelidade do oráculo, confrontando o espaço de estados construído pelo carregador com os rastros oficiais gerados pelo UPPAAL.	3.10.7

Fonte: Autor (2026).

3.7 ARQUITETURA DO SISTEMA

O sistema avalia o programa sob teste de forma caixa-preta: a entrada pode ser uma implementação em código-fonte (em C++, no estudo de caso), a ser compilada e executada, ou, diretamente, os registros de execução já produzidos por qual-

Figura 10 – Estrutura do repositório.



Fonte: Autor (2026).

quer programa — em qualquer linguagem ou ferramenta — que respeite o formato esperado. No primeiro caso, o binário é compilado, executado em ambiente isolado e

o seu registro de execução é capturado. Em ambos os casos, os registros de execução são confrontados com as propriedades derivadas do modelo formal, sem inspeção do código-fonte. A saída é um relatório por implementação e um consolidado, os dois contendo o veredito por propriedade e os erros encontrados.

A pipeline foi implementada de forma genérica, com toda a lógica específica do estudo de caso isolada na descrição do caso. O orquestrador não conhece o caso avaliado: ele consome essa descrição, que declara o caminho do modelo do UPPAAL, a função de projeção do estado formal para a forma observável, o formato esperado do registro de execução e a lista de propriedades a verificar. Com essa abstração, viabiliza-se a reutilização da infraestrutura para qualquer problema de sincronização que possa ser modelado como uma máquina de estados finita.

A Figura 11 ilustra a sequência de etapas. Cada bloco corresponde a um módulo Python independente, com responsabilidade única. As próximas seções descrevem o conteúdo desses blocos em ordem de execução.

3.8 MODELO FORMAL

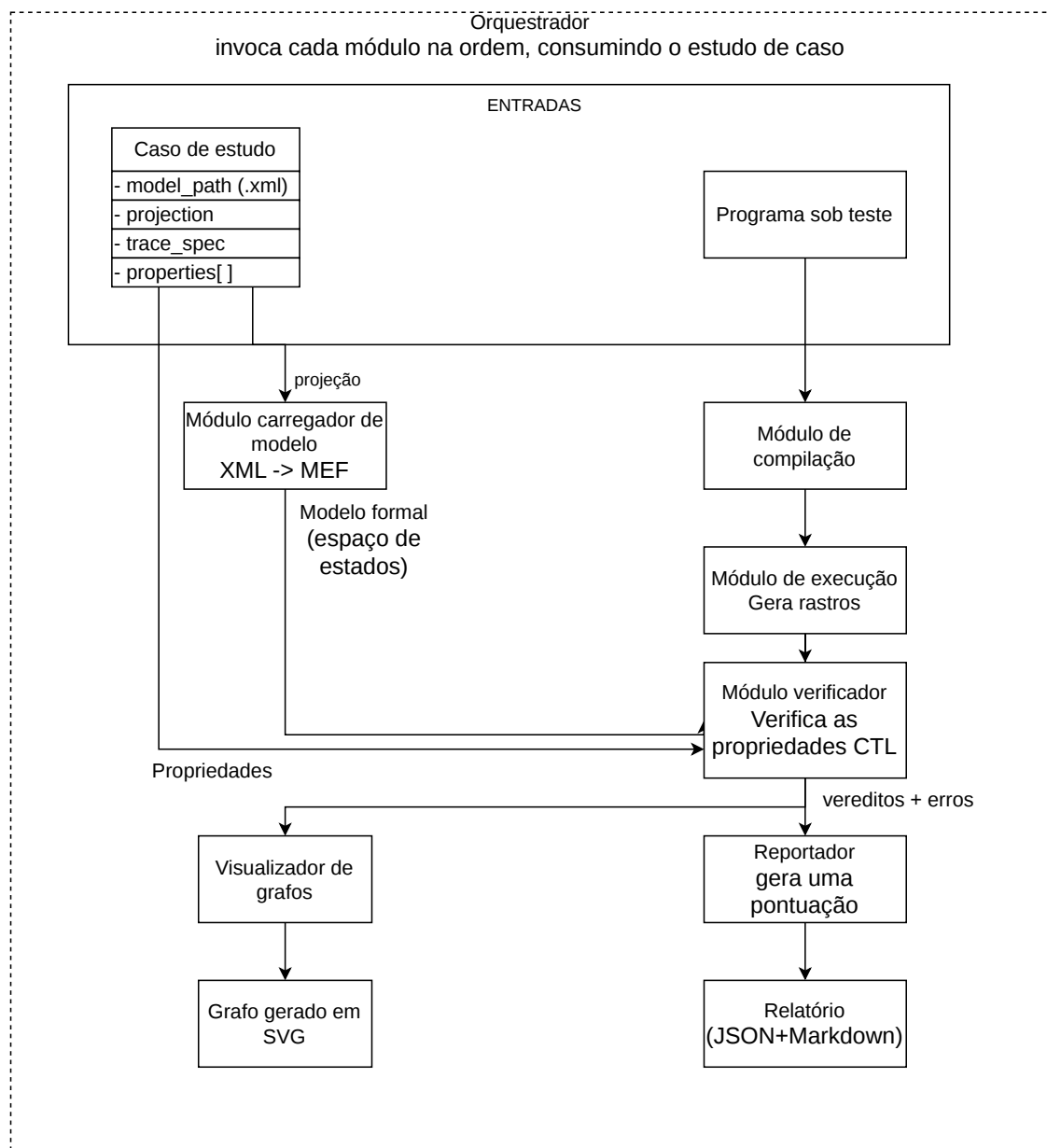
O modelo formal foi construído na ferramenta UPPAAL, que oferece um editor gráfico para autômatos temporizados e um verificador (*verifyta*) capaz de checar propriedades em CTL e gerar contraexemplos como rastros concretos. A escolha do UPPAAL deu-se pela facilidade de exportar rastros em formato textual e pelo ferramenta gráfico, que reduz a barreira de entrada para quem não tem familiaridade com lógica temporal.

O modelo do jantar dos filósofos, armazenado em `models/dining_philosophers.xml`, é composto por dois autômatos parametrizados pelo identificador do processo: *Filosofo* (Figura 12) e *Garfo* (Figura 13). O autômato *Filosofo* possui três localizações observáveis — *Pensando*, *Fome* e *Comendo* — e localizações intermediárias *committed* (*c1*, *c2*, *c3*) que garantem a aquisição atômica dos dois garfos; suas transições são disparadas por canais *broadcast* indexados. O autômato *Garfo* é sincronizado por esses mesmos canais.

A fonte da verdade para o estado dos garfos é o array booleano global `garfo[]`, manipulado pelas funções globais `pegar_garfo` e `liberar_garfo`. Todas as fórmulas CTL verificadas pelo *verifyta* referenciam esse array, e a função de projeção do framework Python também lê o array. O Quadro 5 resume as variáveis globais relevantes para a verificação.

Sobre esse modelo, foram declaradas manualmente no editor do UPPAAL e verificadas com o *verifyta* propriedades em quatro categorias, conforme o Quadro 6. As propriedades de segurança (*A[]*) garantem invariantes globais: ausência de deadlock, capacidade máxima de dois filósofos comendo simultaneamente, exclusão mútua entre vizinhos e coerência entre o estado do filósofo e o estado dos garfos

Figura 11 – Visão geral do pipeline de teste por verificação de registros de execução.

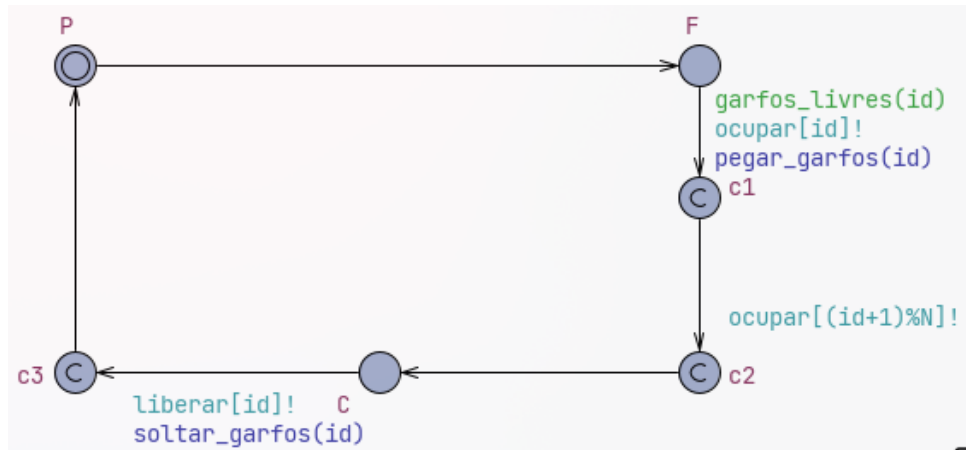


Fonte: Autor (2026).

adjacentes. As propriedades de alcançabilidade positiva ($E<>$) confirmam que comportamentos desejáveis são possíveis: cada filósofo eventualmente come, e pares não-adjacentes podem comer em paralelo. Um terceiro grupo, denominado contraexemplos, declara propriedades que *devem falhar*: a ferramenta confirma a corretude do modelo precisamente quando essas queries são refutadas (por exemplo, é impossível alcançar um estado em que dois vizinhos comam juntos).

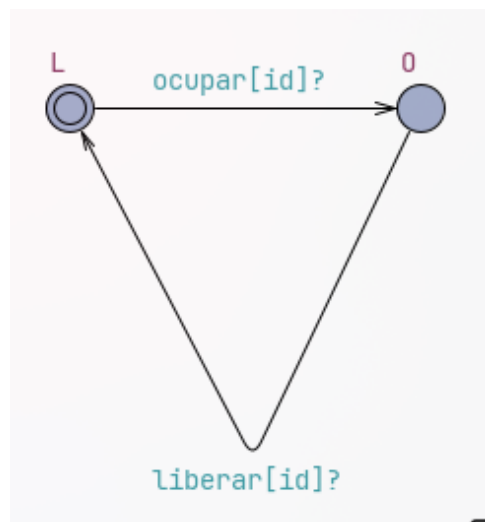
A quarta categoria é destacada pela conexão direta com o framework Python.

Figura 12 – Autômato Filósofo no editor do UPPAAL.



Fonte: Autor (2026).

Figura 13 – Autômato Garfo no editor do UPPAAL.



Fonte: Autor (2026).

A query $E[] \text{ not } P0$.Comendo passa, e essa passagem é desejada: ela formaliza que, na ausência de uma política de escalonamento justo, existe uma execução infinita em que o filósofo $P0$ nunca come — ou seja, inanição é alcançável neste modelo. A propriedade complementar $E<> P0$.Comendo também passa: existe uma execução em que ele eventualmente come. Essas duas queries existenciais correspondem, respectivamente, às verificações de ausência de inanição e de vivacidade, aplicadas ao estado final do registro de execução do programa sob teste. A escolha de operadores existenciais no modelo decorre de uma limitação prática do UPPAAL, que não aceita aninhamento direto de $A[] E<> \varphi$; a semântica desejada é recuperada no framework Python por meio de uma busca em largura a partir do último estado observado.

A Figura 14 apresenta o resultado da verificação de cada propriedade pelo

Quadro 5 – Variáveis globais declaradas no modelo UPPAAL.

Identificador	Tipo	Significado
N	const int	número de filósofos (5)
garfo[N]	bool []	true = livre; false = em uso
pegar[N]	chan	canal broadcast de aquisição
liberar[N]	chan	canal broadcast de liberação

Fonte: Autor (2026).

Quadro 6 – Propriedades CTL verificadas no modelo UPPAAL.

Categoria	Propriedade
Segurança	$A[] \text{ not deadlock}$ (ausência de deadlock global)
	$A[] \sum P_i.\text{Comendo} \leq 2$ (capacidade máxima)
	$A[] \text{ not } (P_i.\text{Comendo} \text{ and } P_{(i+1)}.\text{Comendo})$ para $i \in \{0..4\}$ (exclusão mútua entre vizinhos)
	$A[] (P_i.\text{Comendo} \text{ imply } (\text{garfos adjacentes ocupados}))$ para $i \in \{0..4\}$ (coerência filósofo–garfos)
Alcance positivo	$E<> P_i.\text{Comendo}$ para cada i (cada filósofo eventualmente come)
	$E<> (P_i.\text{Comendo} \text{ and } P_j.\text{Comendo})$ para $ i - j > 1$ (não-adjacentes em paralelo)
Contraexemplos	$E<> (P_i.\text{Comendo} \text{ and } P_{(i+1)}.\text{Comendo})$ (devem falhar: vizinhos juntos é impossível)
	$E<> \sum P_i.\text{Comendo} \geq 3$ (deve falhar: 3+ comendo é impossível)
Vivacidade/Inanição existencial	$E<> P_0.\text{Comendo}$ (vivacidade é alcançável)
	$E[] \text{ not } P_0.\text{Comendo}$ (inanição é alcançável sem fairness)

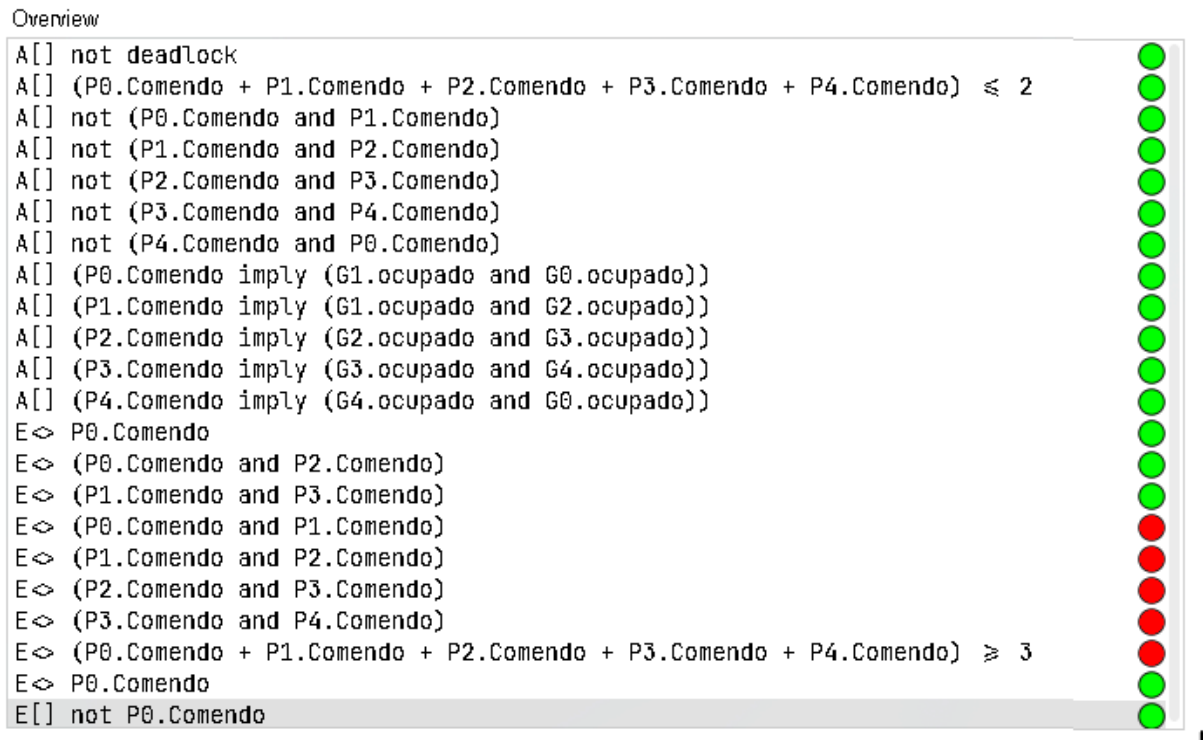
Fonte: Autor (2026).

verifityta. As propriedades de segurança e de alcançabilidade positiva foram confirmadas (marcadas como satisfeitas), enquanto os contraexemplos foram refutados, conforme esperado. Esse resultado valida o modelo formal antes de ele ser utilizado como oráculo pelo framework Python.

3.9 GERAÇÃO E FORMATO DOS REGISTROS DE EXECUÇÃO

Os rastros e registros de execução têm três papéis distintos no sistema. O primeiro são os rastros gerados pelo próprio UPPAAL durante a verificação e a simulação, nos formatos `.xtr` (rastros simbólicos) e `.uctr` (rastros concretos em JSON). O segundo papel desses rastros é servir como fonte da verdade para validar o carregador do modelo formal: o validador cruzado traduz cada arquivo e confirma que todo estado observável visitado pelo UPPAAL pertence ao espaço de estados construído pelo carregador a partir do XML. Esse cruzamento garante que a interpretação do XML é consistente com a semântica oficial do UPPAAL, e não introduz estados errados nem omite transições reais. Os rastros podem ainda substituir o XML como fonte

Figura 14 – Resultado da verificação das propriedades no UPPAAL.



Fonte: Autor (2026).

primária do espaço de estados, construído diretamente a partir do corpus.

O terceiro papel são os registros de execução emitidos pelo programa sob teste durante a execução. Toda implementação que pretenda ser avaliada pelo sistema deve escrever, na sua saída padrão, uma sequência de linhas em que cada linha representa uma captura do estado global do sistema. No estudo de caso do jantar dos filósofos, cada linha contém dez tokens separados por vírgula: os cinco primeiros descrevem o estado de cada filósofo (P para Pensando, F para Fome, C para Comendo) e os cinco últimos descrevem o estado de cada garfo (L para livre, O para ocupado).

O estado inicial canônico, comum a todos os registros de execução válidos, corresponde a todos os filósofos pensando e todos os garfos livres (P,P,P,P,P para os filósofos e L,L,L,L,L para os garfos). O Quadro 7 resume o alfabeto adotado.

Quadro 7 – Alfabeto dos registros de execução.

Token	Entidade	Significado
P	Filósofo	Pensando
F	Filósofo	Fome
C	Filósofo	Comendo
L	Garfo	Livre
O	Garfo	Ocupado

Fonte: Autor (2026).

Listagem 3.1 – Trecho real de um registro de execução do jantar dos filósofos no formato CSV canônico.

```

1 P,P,P,P,P,L,L,L,L,L estado inicial: todos pensando, garfos livres
2 F,P,P,P,P,L,L,L,L,L P0 fica com fome
3 C,P,P,P,P,0,0,L,L,L P0 pega os garfos 0 e 1 e passa a comer
4 C,P,P,P,F,0,0,L,L,L P4 fica com fome
5 C,P,F,P,F,0,0,L,L,L P2 fica com fome
6 C,P,C,P,F,0,0,0,0,L P2 pega os garfos 2 e 3 e come (em paralelo com P0)

```

A Listagem 3.1 mostra as primeiras linhas de um registro de execução real: os cinco primeiros campos descrevem os filósofos e os cinco últimos os garfos (Quadro 7). Cada filósofo passa por Pensando, Fome e Comendo, e P0 e P2, não-adjacentes, chegam a comer em paralelo.

3.10 FRAMEWORK DE VERIFICAÇÃO

O núcleo do sistema é um framework de verificação escrito em Python, organizado em módulos descritos a seguir. Um módulo orquestrador coordena a pipeline, invocando os demais na ordem correta e propagando entre eles a descrição do caso de estudo. A organização modular segue o princípio de responsabilidade única: cada módulo cuida de uma etapa da pipeline e expõe uma interface para o orquestrador.

3.10.1 Abstração de caso de estudo

A abstração central é a descrição do caso de estudo, que reúne o nome do caso, a referência ao arquivo XML do modelo, a função de projeção, a especificação do formato do registro de execução e a lista de propriedades a verificar. A resolução de caminhos é feita por convenção de diretórios, a partir do nome do caso, sem necessidade de configuração explícita. A Figura 15 resume essa abstração.

Figura 15 – Abstração de caso de estudo: a descrição do caso parametriza o orquestrador e a pipeline.



Fonte: Autor (2026).

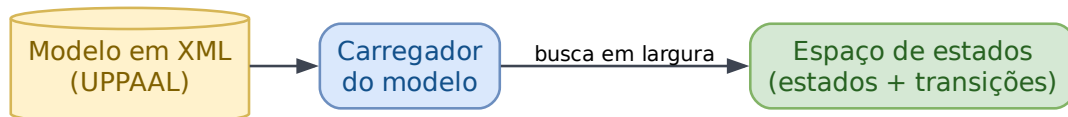
Cada propriedade declara uma chave única, uma descrição legível, um peso relativo e uma função de verificação. Essa função recebe o registro de execução nor-

malizado, o modelo formal carregado e o próprio caso, e devolve o veredito booleano e a lista de erros encontrados. Essa assinatura uniforme permite que o verificador trate todas as propriedades da mesma forma, sem conhecer o que cada uma olha.

3.10.2 Carregamento do modelo formal

O carregador do modelo converte o XML do UPPAAL em uma estrutura de dados que representa o modelo formal. A conversão percorre as declarações globais para extrair variáveis e canais, lê cada template para mapear localizações e transições, e instancia os processos conforme a declaração do sistema. O resultado é o conjunto completo de estados alcançáveis e a relação de transição entre eles. A Figura 16 resume essa etapa.

Figura 16 – Carregamento do modelo: o XML do UPPAAL é convertido no espaço de estados.



Fonte: Autor (2026).

Como o XML do UPPAAL contém expressões em C em guardas, atribuições e corpos de função, o carregador faz a expansão das funções globais e traduz as expressões por meio de um mini-conversor restrito aos operadores lógicos utilizados pelo modelo. A avaliação dessas expressões foi restrita ao XML do modelo, fornecido pelo caso de teste, e nunca ao código do programa sob teste.

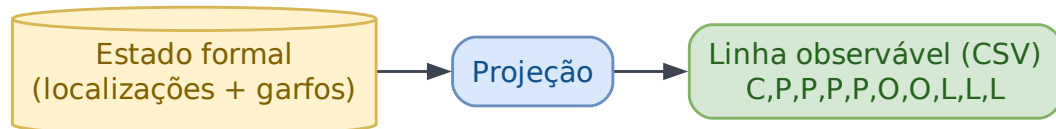
O conjunto bruto de estados alcançáveis costuma ser maior do que o conjunto de estados observáveis, porque o UPPAAL inclui transições que passam por localizações committed (que precisam ser executadas atomicamente antes de qualquer outra) ou por configurações intermediárias não significativas para a avaliação. A função de projeção descarta esses estados, e o carregador os filtra durante a construção do espaço observável.

3.10.3 Projeção do estado formal

A projeção é o elo entre o modelo formal e o registro de execução emitido pelo programa sob teste. Para o jantar dos filósofos, ela percorre as instâncias do autômato do filósofo e mapeia cada localização para o token correspondente, lê o estado dos garfos para emitir os tokens correspondentes, e descarta estados que estejam em

uma localização committed. O resultado é uma linha no mesmo formato CSV que o programa sob teste deve emitir, o que viabiliza a comparação direta. A Figura 17 ilustra a projeção.

Figura 17 – Projeção: cada estado formal é mapeado para uma linha observável no formato CSV.



Fonte: Autor (2026).

3.10.4 Compilação e execução do programa sob teste

O compilador localiza o arquivo principal da implementação varrendo o diretório de entrada em busca de um arquivo-fonte em C++ que contenha uma função principal, compila-o com o g++ usando o padrão C++20, com avisos habilitados e suporte a threads, e registra o resultado em um log. Um limite de trinta segundos é imposto à compilação, e o seu sucesso é uma pré-condição para a verificação: se a compilação falha, todas as propriedades são marcadas como falsas e o veredito final reflete essa condição.

O executor roda o binário compilado um número configurável de vezes (com valor padrão um) em diretórios isolados, cada execução sujeita a um limite de trezentos segundos para evitar que implementações com deadlock real travem a pipeline. A captura do registro de execução segue uma política em duas etapas: o executor procura primeiro por um arquivo de saída em formatos comuns (CSV, texto ou log) que o programa sob teste possa ter gravado em disco, preferindo nomes associados a rastro ou saída; se nenhum arquivo válido é encontrado, o conteúdo da saída padrão é interpretado como alternativa. Em ambos os casos, o registro é normalizado para o formato CSV canônico definido pelo caso de estudo antes de seguir para a verificação. O escalonador de threads introduz não-determinismo: uma mesma implementação pode produzir registros de execução distintos em execuções consecutivas. Justamente porque uma implementação incorreta pode, ocasionalmente, produzir um registro válido por mero acaso, o veredito é formado a partir de uma única execução, sem selecionar o melhor de várias (Seção 3.10.5). A Figura 18 resume essa etapa.

Figura 18 – Compilação e execução: da implementação em C++ ao registro de execução.



Fonte: Autor (2026).

Quando a entrada já é um registro de execução — um diretório que contém arquivos `.csv`, mas nenhum fonte compilável — o orquestrador omite as etapas de compilação e execução e segue diretamente para a verificação. Esse modo desacopla a origem do registro de execução do restante da pipeline: o programa sob teste pode ter sido executado em outra máquina, ou o registro de execução pode ter sido produzido por outra ferramenta, sem alteração alguma nas etapas seguintes.

3.10.5 Verificação por propriedades

O verificador lê o registro normalizado e itera sobre as propriedades declaradas no caso, chamando a respectiva função de verificação. Cada propriedade é avaliada de forma independente, e o resultado agregado contém o veredito de cada uma, a lista de erros acumulados e um indicador global de aprovação. Esse resultado é serializado em JSON para inspeção posterior. A Figura 19 resume essa etapa.

Figura 19 – Verificação por propriedades: o registro de execução e o espaço de estados produzem o veredito por propriedade.



Fonte: Autor (2026).

Cada implementação é avaliada a partir de uma única execução, confrontada contra o espaço de estados completo do modelo. Optou-se deliberadamente por não

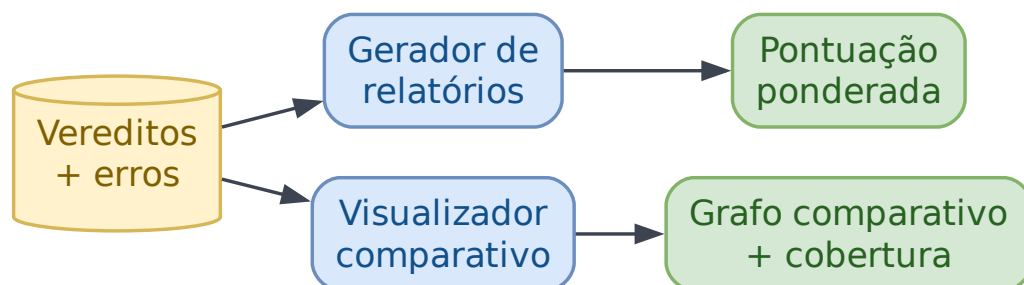
selecionar a melhor entre várias execuções: como uma implementação incorreta pode, por acaso, produzir um registro de execução válido em uma execução isolada, escolher o melhor resultado mascararia o defeito. O veredito reflete, assim, o que a execução de fato produziu, e não o seu melhor caso.

3.10.6 Geração de relatórios

O gerador de relatórios consome os resultados da avaliação de cada implementação e calcula uma pontuação agregada como soma ponderada: cada propriedade satisfeita contribui com o seu peso para um total de cem pontos. Essa pontuação serve como métrica quantitativa de conformidade, e é especialmente útil quando o sistema é aplicado em lote a várias implementações da mesma especificação.

O visualizador comparativo gera, em paralelo, um grafo que confronta o espaço de estados do modelo com o subconjunto efetivamente visitado pelo registro de execução do programa sob teste. Esse grafo é exportado em formato vetorial e serve para inspeção humana quando o veredito automático não esclarece a causa raiz da falha. A Figura 20 resume essa etapa.

Figura 20 – Geração de relatórios e visualização comparativa a partir dos vereditos.

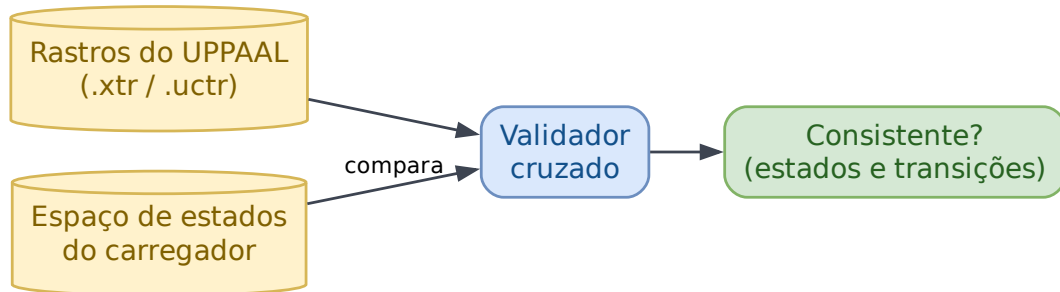


Fonte: Autor (2026).

3.10.7 Validação cruzada do carregador

O carregador descrito na Seção 3.10.2 converte uma especificação em XML para um espaço de estados, e essa conversão pode, em princípio, divergir da semântica oficial do UPPAAL. Para detectar tal divergência, o framework inclui um validador cruzado, que decodifica os rastros gerados pelo UPPAAL — nos formatos `.xtr` e `.uctr` — e confronta cada estado observável que aparece nesses rastros com o espaço de estados construído pelo carregador. A Figura 21 resume essa validação.

Figura 21 – Validação cruzada: os rastros do UPPAAL confirmam o espaço de estados construído pelo carregador.



Fonte: Autor (2026).

A validação consiste em duas etapas. A primeira é a decodificação do rastro simbólico (.xtr), que requer interpretar o formato interno de localizações, variáveis globais e relógios do UPPAAL; o formato concreto (.uctr), mais recente, dispensa essa etapa por codificar o rastro diretamente em JSON. A segunda é a aplicação da função de projeção do caso, que produz, para cada estado decodificado, a mesma representação observável em CSV utilizada no restante do framework. Estados em localizações committed são descartados, em coerência com a regra de observabilidade adotada na Seção 3.10.3.

Se algum estado visitado pelo UPPAAL não pertence ao espaço de estados construído pelo carregador, ou se alguma transição observada não consta da relação de transição, identifica-se um erro de interpretação do XML. A passagem dessa validação sobre o corpus completo de rastros oficiais permite concluir que o carregador é consistente com a saída do verificador de modelos, condição necessária para que o framework seja confiável como ferramenta de teste. Como benefício adicional, o espaço de estados pode ser construído diretamente a partir do corpus de rastros, contornando o XML quando a especificação contiver elementos fora do subconjunto suportado pelo carregador.

3.11 EXTENSIBILIDADE

A inclusão de um novo caso de estudo no sistema é assistida por um subcomando de inicialização, que cria a estrutura mínima de diretórios e arquivos para o novo caso — um esboço da descrição do caso, uma pasta para a implementação de referência e uma entrada para o XML do UPPAAL — sem alterar o orquestrador.

O subconjunto da linguagem UPPAAL suportado pelo carregador atual abrange canais broadcast com índice, variáveis booleanas e inteiras (escalares ou em array),

localizações committed e expressões em C compostas pelos operadores lógicos básicos. Casos que envolvam relógios contínuos ou estruturas compostas exigem entender o mini-conversor de C para Python e o iterador do espaço de estados. Essa limitação é aceitável para os problemas clássicos de sincronização tratados na literatura, e foi deliberadamente adotada para manter o carregador legível.

4 RESULTADOS E DISCUSSÃO

O estudo de caso que dá base a este capítulo é o jantar dos filósofos, problema clássico de sincronização introduzido na Seção 2.1.1 e adotado como vitrine dos três fenômenos de interesse deste trabalho. Cinco filósofos compartilham, ao redor de uma mesa, cinco garfos — um entre cada par de vizinhos — e cada filósofo alterna entre pensar, ter fome e comer; para comer, precisa adquirir simultaneamente os dois garfos adjacentes (Dijkstra, 1971). Essa regra simples concentra, em um espaço de estados pequeno, os três defeitos que o sistema de testes precisa distinguir: a *condição de corrida* sobre o estado compartilhado dos garfos, o *deadlock* quando todos pegam o garfo de um mesmo lado ao mesmo tempo, e a *inanição* quando uma política de acesso injusta mantém um filósofo indefinidamente com fome (Coffman *et al.*, 1971; Tanenbaum; Bos, 2016).

Com o enunciado definido, as treze implementações avaliadas resolvem o mesmo problema por estratégias de sincronização distintas, e é isso que torna os resultados comparáveis: todas são confrontadas com o mesmo espaço de estados derivado do modelo (Seção 3.8) e com as mesmas cinco propriedades (Quadro 8). Cada estado observável do registro de execução corresponde a uma configuração dos cinco filósofos e dos cinco garfos, no alfabeto definido na Seção 3.9, de modo que cobertura, transições válidas e os vereditos por propriedade têm o mesmo significado para qualquer das implementações. É sob essa ótica que as seções seguintes leem os registros de execução coletados.

Este capítulo apresenta os resultados obtidos com a aplicação do sistema a treze programas: a implementação de referência descrita na seção 3.6 e doze implementações independentes do mesmo problema, produzidas por terceiros. A Seção 4.2 descreve o conjunto avaliado. A Seção 4.3 analisa a cobertura do espaço de estados. A Seção 4.4 discute a detecção das propriedades violadas. A Seção 4.5 aponta limitações observadas.

4.1 ESTUDO DE CASO: JANTAR DOS FILÓSOFOS

A implementação de referência em C++, disponível em `cpp/dining_philosophers/dining_philosophers.cpp`. Utiliza `std::thread` para representar cada filósofo e `std::mutex` para cada garfo. A política de aquisição evita deadlock por meio da ordenação dos garfos: o filósofo de índice mais alto adquire primeiro o garfo da direita, enquanto os demais adquirem o da esquerda (Williams, 2019). Cinco propriedades foram declaradas no caso, totalizando os cem pontos da pontuação agregada, conforme o Quadro 8.

Quadro 8 – Propriedades verificadas no estudo de caso.

Chave	Descrição	Peso
valid_transitions	Cada par de estados consecutivos do registro de execução corresponde a uma transição do modelo	25
valid_forks	Todos os estados do registro de execução pertencem ao espaço de estados alcançável	20
no_deadlock	O registro de execução não termina com todos os filósofos em Fome simultaneamente	20
no_starvation	Para todo filósofo em Fome no estado final, existe execução futura que o leva a Comendo	15
liveness	Todo filósofo come no registro de execução ou existe execução futura em que ele coma	20

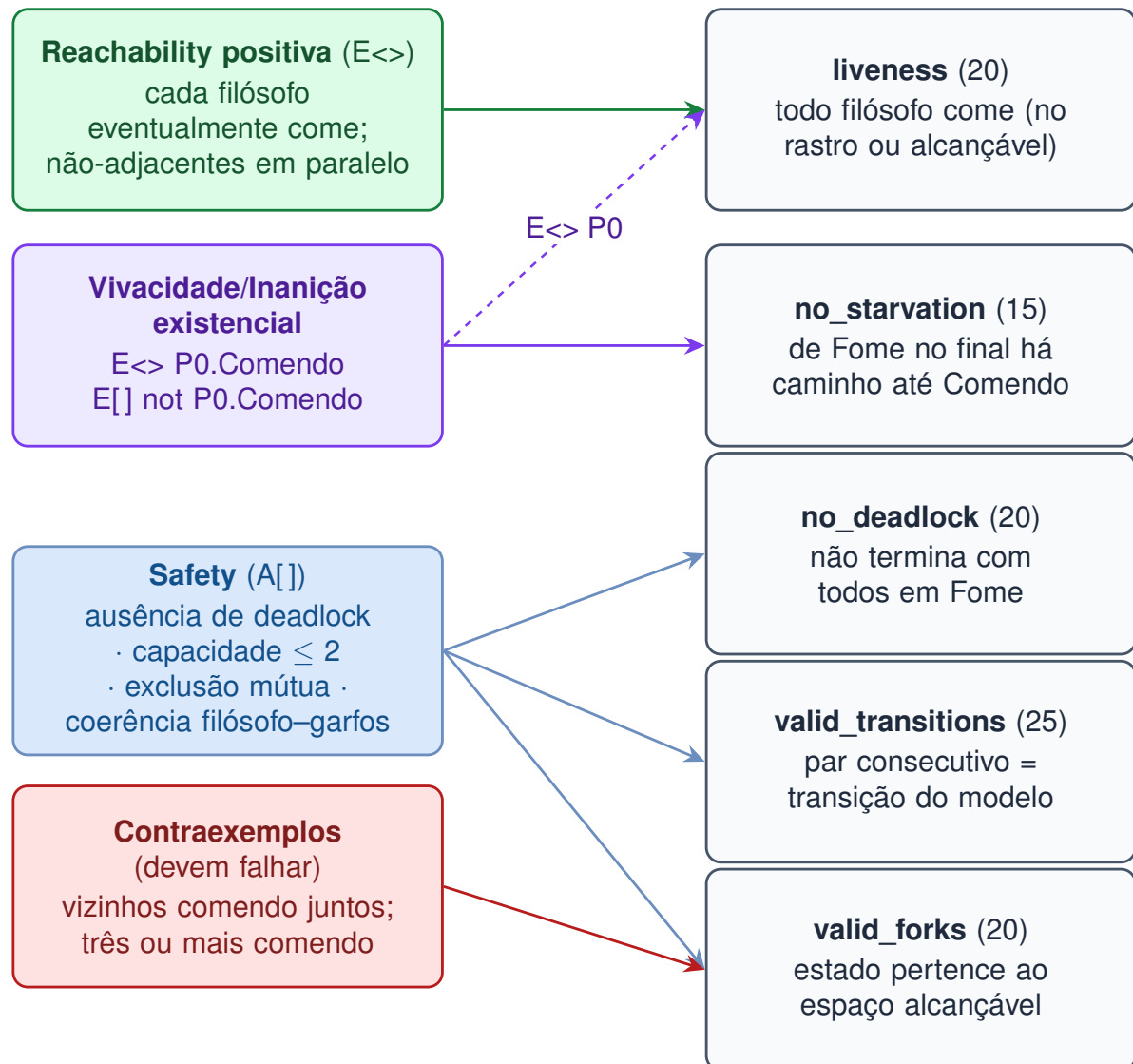
Fonte: Autor (2026).

A verificação de transições compara cada par consecutivo de estados do registro de execução com a relação de transição do modelo, e exige adicionalmente que o primeiro estado do registro de execução corresponda ao estado inicial canônico. A verificação de consistência dos garfos exige que todo estado pertença ao espaço alcançável. As propriedades de vivacidade e inanição são implementadas por uma busca em largura sobre o modelo formal, a partir do estado final do registro de execução, em busca de um estado em que o filósofo em questão esteja na localização Comendo. Essa busca corresponde diretamente à semântica do operador $E \langle \rangle$ do CTL, aplicada ao último estado observável do registro de execução.

As cinco *checks* verificadas no estudo de caso (Quadro 8) não são independentes das propriedades CTL declaradas no modelo (Quadro 6): cada uma instancia, sobre o registro de execução do programa sob teste, uma ou mais daquelas propriedades. A Figura 22 resume essa correspondência. As *checks* *valid_transitions* e *valid_forks* materializam a categoria *Safety* — a primeira exige que cada par consecutivo de estados seja uma transição legal do modelo, e a segunda que todo estado pertença ao espaço alcançável, o que preserva, por construção, as invariantes $A[]$ de capacidade máxima, exclusão mútua entre vizinhos e coerência filósofo–garfos. A *valid_forks* também cobre os *contraexemplos*: um estado em que vizinhos comam juntos, ou em que três ou mais filósofos comam ao mesmo tempo, não existe no espaço alcançável e seria sinalizado como inválido. A *no_deadlock* corresponde diretamente a $A[] \text{ not deadlock}$. Por fim, *liveness* e *no_starvation* traduzem as propriedades existenciais $E \langle \rangle P_i \text{.Comendo}$, ambas implementadas pela busca em largura a partir do último estado do registro de execução: *liveness* considera todos os filósofos, ao passo que *no_starvation* restringe-se àqueles que terminam o registro de execução em Fome.

Figura 22 – Correspondência entre as categorias de propriedades CTL do modelo (Quadro 6) e as *checks* do framework (Quadro 8).

Das categorias CTL do modelo (Quadro 4)
para as checks do framework (Quadro 6)



Fonte: Autor (2026).

4.2 IMPLEMENTAÇÕES AVALIADAS

O conjunto avaliado reúne dois grupos. O primeiro é a implementação de referência (REF), desenvolvida em conjunto com o modelo formal e usada como controle positivo: dela se espera a pontuação máxima, e qualquer reprovação indicaria defeito no próprio sistema de testes, não no programa sob teste. O segundo grupo reúne doze implementações independentes do mesmo enunciado, produzidas por estudantes em uma disciplina de graduação sobre sistemas de tempo real ao longo de vários semes-

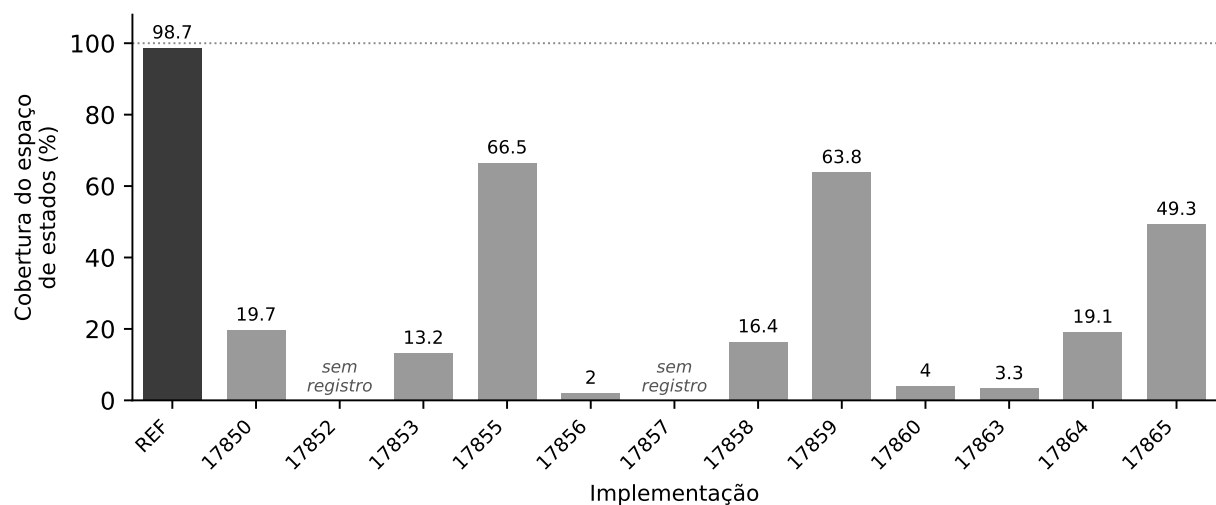
tres e cedidas para uso neste trabalho sem qualquer dado de identificação. Cada uma é identificada, ao longo deste capítulo, por um código numérico de cinco dígitos (por exemplo, 17853). Nenhuma delas foi modificada antes da avaliação; em particular, cada uma conserva a sua própria estratégia de instrumentação, isto é, o momento e a frequência com que registra o estado global.

Na rodada relatada neste capítulo, cada implementação coletada foi compilada e executada uma única vez, com limite de trezentos segundos de execução — programas que não terminam por conta própria são encerrados nesse ponto, e o registro de execução produzido até ali é aproveitado. A referência entrou pelo modo de verificação direta descrito na Seção 3.10.4: um registro de quinze minutos de execução contínua, com 76.093 estados, capturado separadamente e dedicado a exercitar comportamentos raros que registros de execução curtos dificilmente alcançam. Os resultados desse registro de execução aparecem nas seções seguintes identificados como REF.

4.3 COBERTURA DO ESPAÇO DE ESTADOS

O carregador do modelo identificou 152 estados observáveis para o cenário com cinco filósofos, com 620 transições entre eles. O registro de execução da única execução de cada implementação coletada foi confrontado com esse espaço de estados. A Figura 23 apresenta a fração do espaço visitada por cada implementação, junto com a cobertura do registro longo da referência.

Figura 23 – Cobertura do espaço de estados por implementação. As barras de 17852 e 17857 estão vazias porque nenhuma das execuções produziu registro de execução reconhecido pelo executor.



Fonte: Autor (2026).

A referência cobre 98,7% do espaço (150 de 152 estados) no registro de quinze minutos. Os dois estados que restam de fora são configurações em que três

ou mais filósofos estão famintos com todos os garfos livres (F,F,F,F,F,L,L,L,L,L e P,P,F,F,F,L,L,L,L,L) — janelas que a política de aquisição da referência atravessa rápido demais para que o registro as capture. Esses dois casos exigem vários filósofos com fome ao mesmo tempo e nenhum garfo livre; pela forma como a referência foi implementada, assim que um filósofo entra no estado de fome, ele tenta imediatamente pegar os garfos.

A Figura 24 apresenta a legenda de cores usada nos grafos de comparação entre registro de execução e modelo. As Figuras 25 e 26 mostram, respectivamente, a progressão dos estados seguintes e um detalhe com os próximos estados possíveis (em cinza) do registro de execução da implementação de referência: como todos os estados visitados pertencem ao espaço alcançável, nenhum aparece em vermelho. A Figura 27 mostra a contagem por filósofo nesse registro de execução — todos os cinco filósofos comem ao longo da execução, confirmando a propriedade de *vivacidade*.

As implementações coletadas ficam entre 2,0% e 66,5%, e a cobertura baixa tem duas causas distintas. Uma é o registro de execução curto ou de granularidade grossa: 17860, por exemplo, registra apenas onze estados em cinco minutos de execução. A outra é o defeito de sincronização que confina a execução a uma região estreita do espaço: 17856 visita somente três estados válidos enquanto quatro dos cinco filósofos esperam indefinidamente.

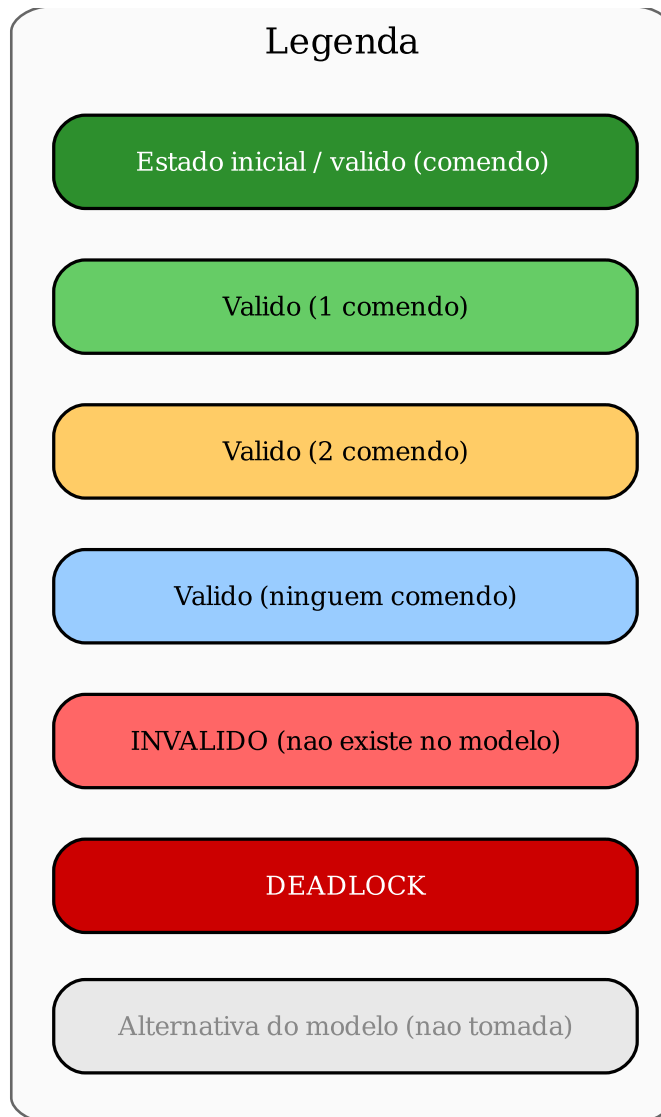
Cabe a ressalva de que cobertura não mede correção: 17855 cobre 66% do espaço — a maior cobertura entre as coletadas — e ainda assim reprova em duas propriedades, enquanto 17859, aprovada em todas, cobre 64%. A métrica descreve o quanto do comportamento modelado foi exercitado — e, por consequência, o quanto do veredito se apoia em evidência — e não a qualidade do programa sob teste.

4.4 DETECÇÃO DE PROPRIEDADES VIOLADAS

Cada implementação foi avaliada contra as cinco propriedades declaradas a partir do registro da única execução. O Quadro 9 apresenta o veredito por propriedade — aprovada (A) ou reprovada (R) — e a pontuação final.

A referência obteve a pontuação máxima, o que sustenta o seu papel de controle positivo. Entre as implementações coletadas, as reprovações encontram correspondência nos registros de execução. Em 17853, 17858, 17864 e 17865, os estados fora do modelo incluem configurações em que filósofos vizinhos comem ao mesmo tempo, violação direta da exclusão mútua sobre os garfos — as três primeiras acumulam centenas de estados desse tipo (299, 219 e 210, respectivamente). Em 17863, o registro de execução de apenas catorze linhas começa em um estado inválido, com quatro filósofos já famintos, e dois deles jamais comem. As Figuras 28 e 29 mostram essa violação: a primeira acompanha a progressão dos estados seguintes — a execução parte de estados válidos e, a partir do passo 6, atinge configurações inexis-

Figura 24 – Legenda de cores dos grafos de comparação entre registro de execução e modelo.



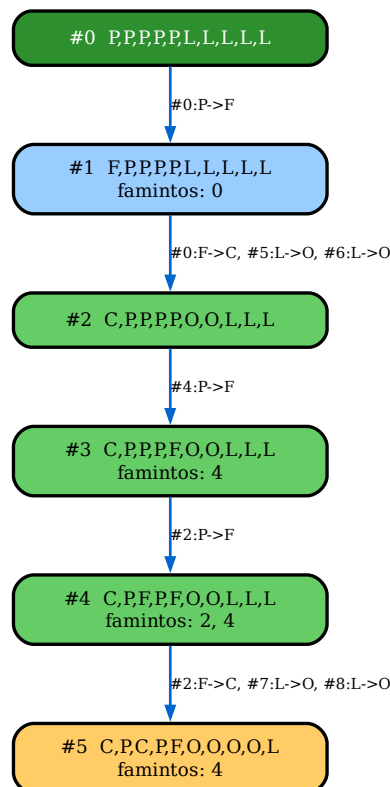
Fonte: Autor (2026).

tentes no modelo; a segunda foca o ponto da divergência, em que a execução vai a um estado fora dos próximos possíveis admitidos pelo modelo (em cinza).

A Figura 30 mostra a saída real do sistema ao avaliar essa submissão: o veredito de invalidez, os estados fora do modelo e a pontuação final são produzidos automaticamente, sem inspeção do código-fonte.

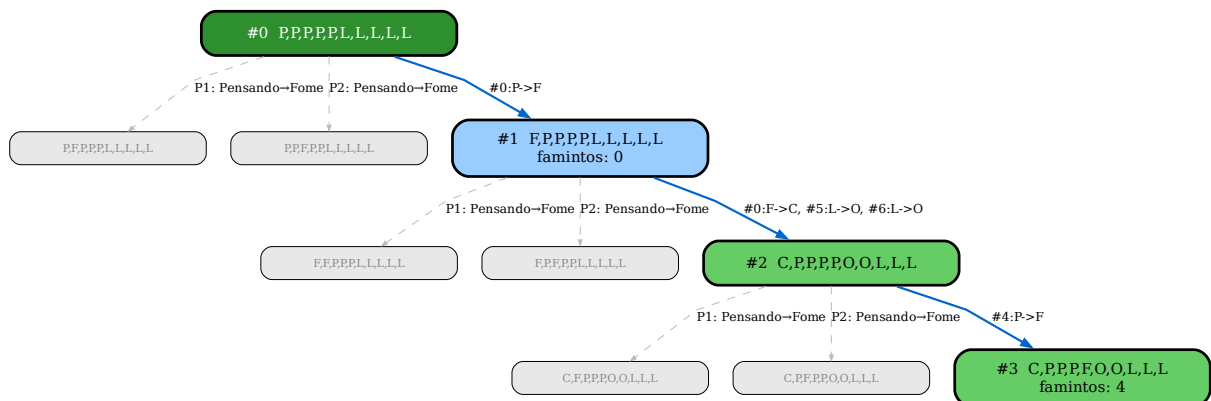
Um contraexemplo de natureza distinta aparece na submissão 17864. A Figura 31 mostra o ponto em que sua execução deixa o modelo: no passo 6 o Filósofo 0 está Comendo segurando os dois garfos adjacentes (estado válido, em verde), e o modelo admite apenas que os demais filósofos passem a Fome (cinza tracejado); a execução, porém, avança para um estado em que o Filósofo 0 permanece Comendo enquanto todos os garfos aparecem livres (vermelho) — configuração impossível, que

Figura 25 – Visão geral do registro de execução da referência: progressão dos estados seguintes.



Fonte: Autor (2026).

Figura 26 – Detalhe (zoom) do registro de execução da referência, com os próximos estados possíveis segundo o modelo em cinza tracejado.

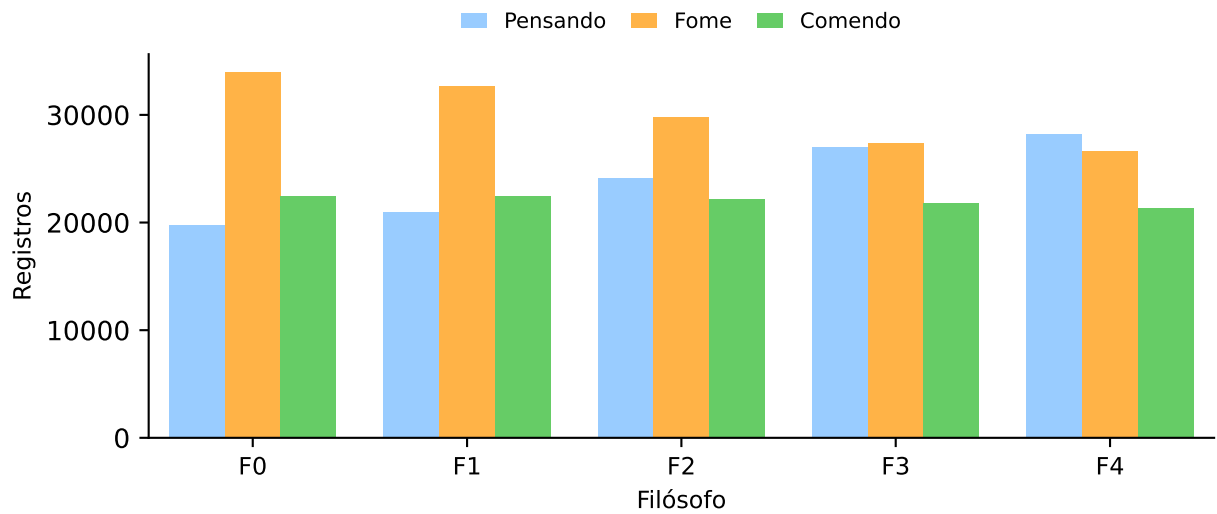


Fonte: Autor (2026).

viola a consistência dos garfos. A partir desse ponto a execução acumula estados fora do modelo, incluindo as configurações em que filósofos vizinhos comem ao mesmo tempo.

O caso de 17856 merece atenção. O sistema lhe atribui a segundo melhor pontuação entre as coletadas (75), ainda que quatro dos cinco filósofos jamais alcan-

Figura 27 – Contagem por filósofo no registro de execução da implementação de referência.



Fonte: Autor (2026).

Quadro 9 – Resultados por implementação. A indica propriedade aprovada; R, reprovada.

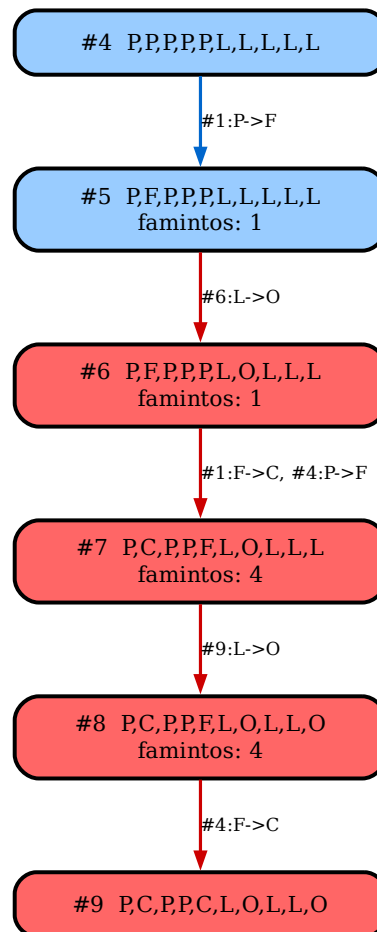
Implementação	trans	forks	deadlock	starv	liveness	pontuação
REF	A	A	A	A	A	100
17850	R	A	A	A	A	75
17852	R	R	R	R	R	0
17853	R	R	A	A	A	55
17855	R	R	A	A	A	55
17856	R	A	A	A	A	75
17857	R	R	R	R	R	0
17858	R	R	A	A	A	55
17859	A	A	A	A	A	100
17860	R	A	A	A	A	75
17863	R	R	R	R	R	0
17864	R	R	A	A	A	55
17865	R	R	A	A	A	55

Fonte: Autor (2026).

cem Comendo em seiscentos registros. A checagem formal de inanição passa porque a busca $E<>$ parte do último estado válido do registro de execução, e desse estado ainda existe caminho até Comendo no modelo; a inanição efetivamente observada aparece apenas na análise diagnóstica, que não pontua. O par de vereditos ilustra a complementaridade entre a checagem formal e a análise empírica — uma responde sobre possibilidade, a outra sobre ocorrência — e sugere que a análise empírica mereceria peso próprio na pontuação.

As Figuras 32 e 33 mostram a progressão dos estados seguintes e um detalhe

Figura 28 – Visão geral do registro de execução da submissão 17853: parte de estados válidos (azul) e, a partir do passo 6, atinge estados fora do modelo (vermelho).



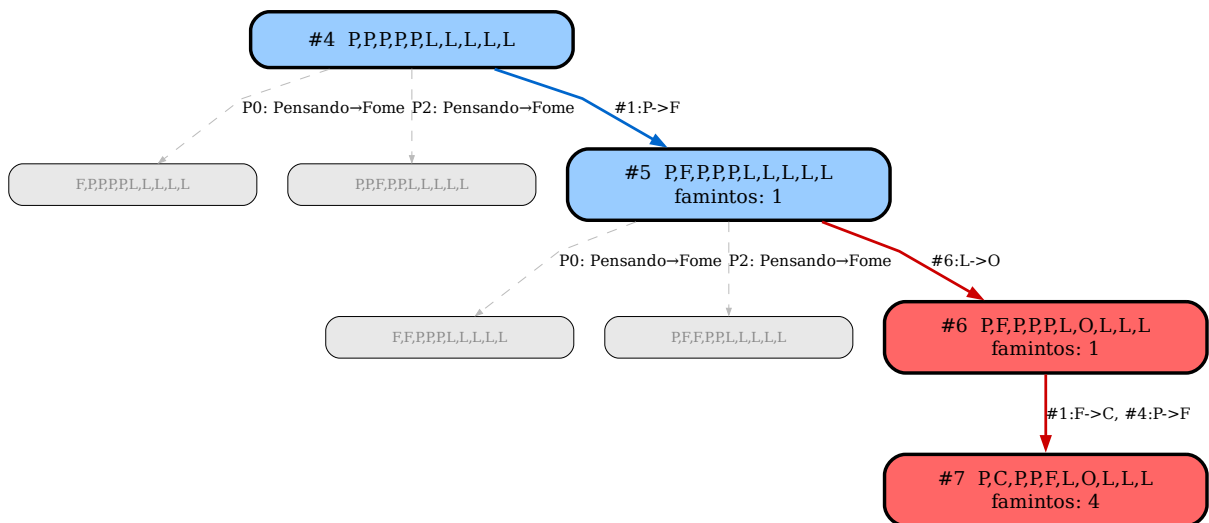
Fonte: Autor (2026).

com os próximos possíveis do registro de execução da submissão 17856: todos os estados visitados são válidos, mas apenas um filósofo alterna para Comendo (cores conforme a legenda da Figura 24). A Figura 34 apresenta a contagem por filósofo no registro de execução completo: apenas o Filósofo 1 come, enquanto os outros quatro permanecem com Comendo = 0, caracterizando a inanição (*starvation*) observada empiricamente.

As pontuações nulas de 17852 e 17857 têm outra natureza: ambas compilam e terminam sem erro, mas a saída que produzem não é reconhecida pelo executor como registro de execução. O zero, nesses casos, registra ausência de evidência, e não defeito comprovado — distinção retomada na Seção 4.5.

A checagem de transições reprovou onze das doze implementações coletadas, e a leitura dos erros aponta uma causa dominante: o registro do estado global em intervalos, em vez de a cada evento de sincronização. Entre dois registros consecutivos pode ocorrer mais de um evento, e o par de estados observado deixa de cor-

Figura 29 – Detalhe (zoom) da submissão 17853 no ponto da divergência: o modelo admite os estados em cinza (próximos possíveis), mas a execução vai a um estado fora do espaço alcançável (vermelho).



Fonte: Autor (2026).

Figura 30 – Saída real do sistema ao avaliar a submissão 17853: o veredito, os estados inválidos detectados e a pontuação são produzidos automaticamente a partir do registro de execução.

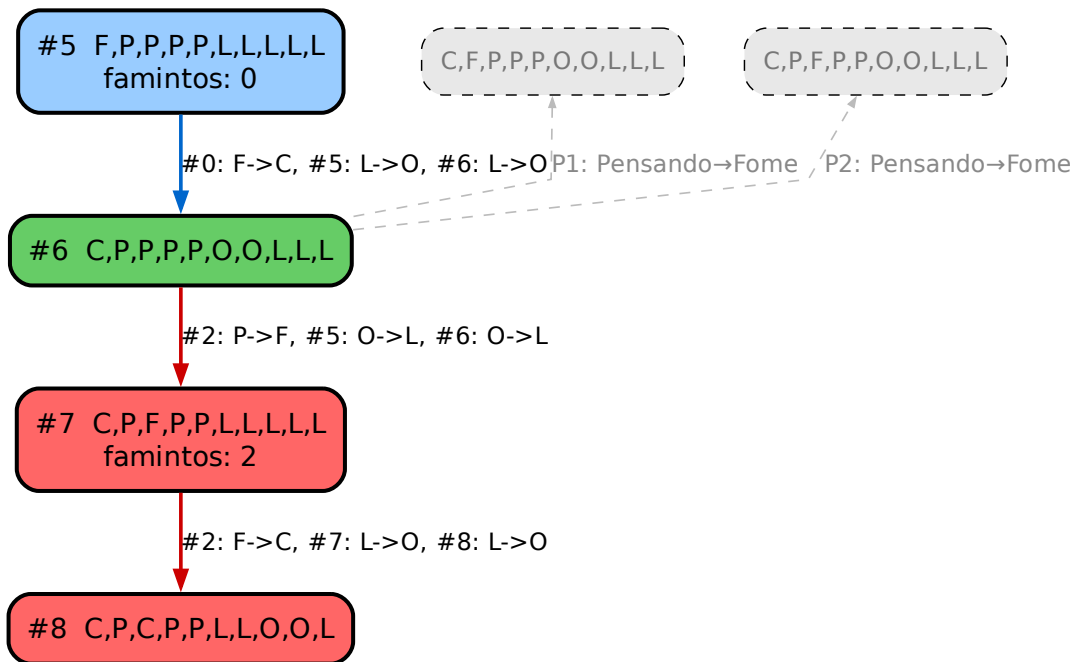
```
$ python -m trace_grader --submissions-dir data/submissions/
=====
Sistema de Correção Automática - caso: dining_philosophers
=====
Carregando modelo formal (BFS do XML): dining_philosophers.xml
Estados observáveis: 152 | transições: 620 | queries: 22

Submissões encontradas: 1
=====
Avaliando: 17853
=====
Verificação direta: 1 registro fornecido (sem compilação)
Verificação run_1: INVÁLIDO
Gerando grafo comparativo... OK
Cobertura: 24/152 estados (15,79%)
ALERTA: 335 estados inválidos no registro de execução!
- C,C,C,C,C,... (todos comendo - impossível no modelo)
Nota final: 55
```

Fonte: Autor (2026).

responder a uma transição do modelo mesmo quando cada evento individual é legal –17856, por exemplo, reprova por um único registro composto em seiscentos. Apenas

Figura 31 – Detalhe (zoom) da submissão 17864 no ponto da divergência: o modelo admite os estados em cinza (próximos possíveis), mas a execução vai a um estado fora do espaço alcançável (vermelho), em que um filósofo permanece comendo sem segurar garfos.



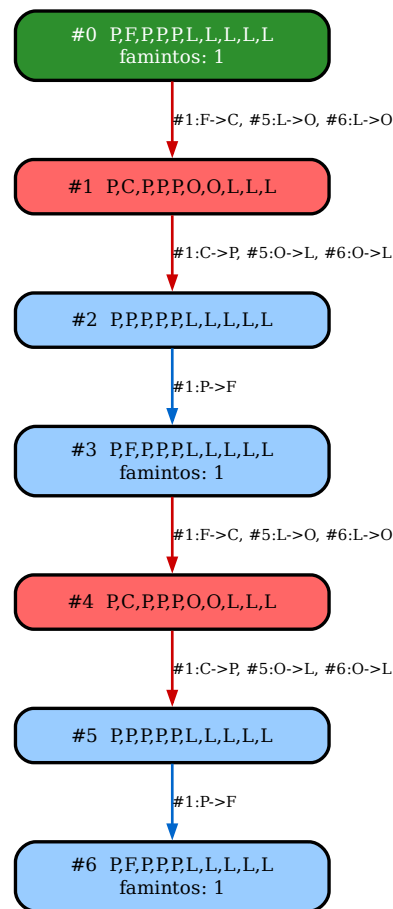
Fonte: Autor (2026).

17859, além da referência, registra o estado a cada evento atômico. Na prática, portanto, a propriedade mede também a granularidade da instrumentação do programa sob teste, e não apenas a validade da sua sincronização.

Vale registrar também o efeito do não-determinismo entre execuções. Uma execução preliminar do mesmo conjunto, com registros de execução mais curtos e três execuções por implementação, produziu vereditos diferentes em quatro das doze coletadas — consistência dos garfos e inanição que passam em uma execução e reprovam em outra, conforme o entrelaçamento sorteado pelo escalonador.

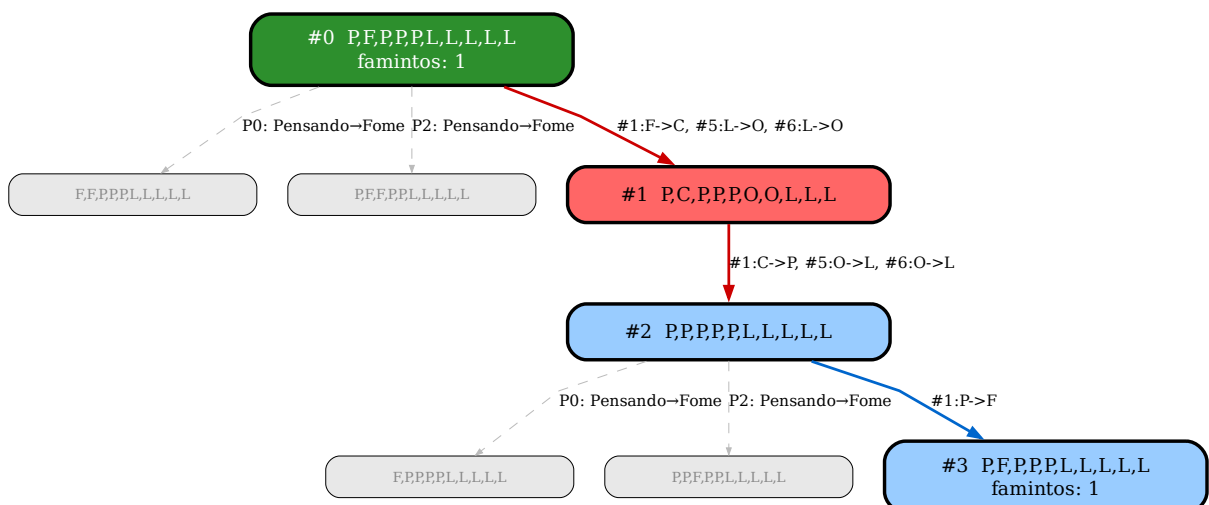
Dois resultados adicionais merecem registro. O primeiro envolve a própria implementação de referência. Na sua primeira versão, o registro de quinze minutos continha dois estados inválidos em 76 mil registros: um filósofo em *Fome* segurando os dois garfos adjacentes, configuração que o modelo não admite. A causa era uma corrida na instrumentação — a marcação dos garfos como ocupados e a transição para *Comendo* ocorriam em seções críticas distintas, e o registro de estado disparado por outra thread podia capturar a janela entre as duas. O defeito não altera a lógica de sincronização, apenas a observação do estado, e nunca se manifestou em execuções

Figura 32 – Visão geral do registro de execução da submissão 17856: progressão dos estados seguintes.



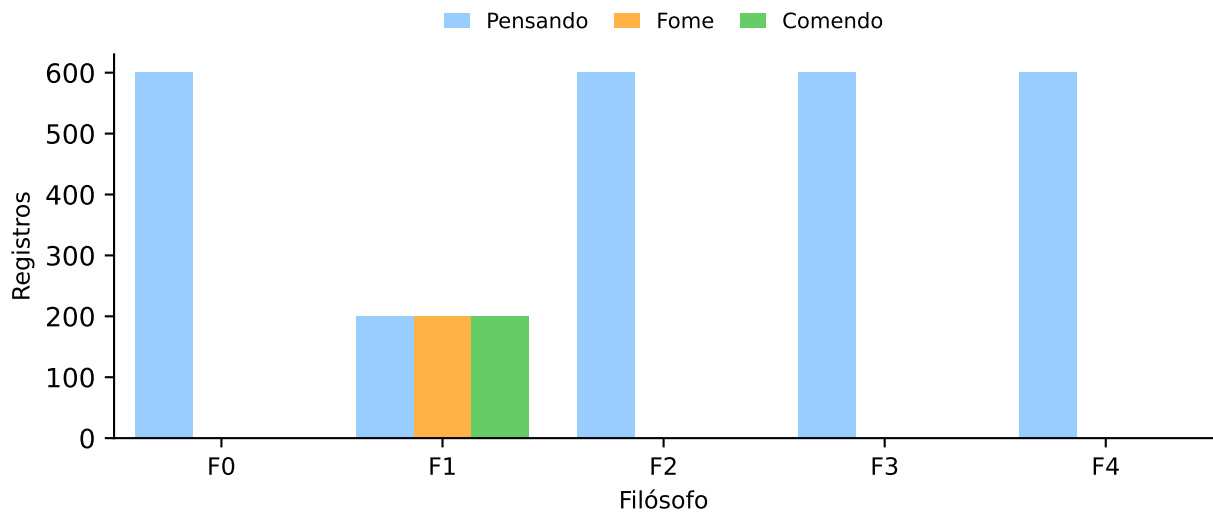
Fonte: Autor (2026).

Figura 33 – Detalhe (zoom) do início do registro de execução da submissão 17856, com os próximos estados possíveis segundo o modelo em cinza tracejado.



Fonte: Autor (2026).

Figura 34 – Contagem por filósofo no registro de execução da submissão 17856.



Fonte: Autor (2026).

curtas, da ordem de dez segundos: com dois registros espúrios em 76 mil, a probabilidade de um registro de execução desse tamanho contê-los fica em torno de 2%. A correção unificou a aquisição dos garfos e a mudança de estado em uma única seção crítica.

O segundo resultado compara as fontes de oráculo. Verificado contra o oráculo derivado do corpus de rastros do simulador UPPAAL, o registro longo da referência reprova na checagem de transições, com pontuação 75: o corpus cobre os 152 estados do modelo, mas registra 619 das 620 transições, e justamente a transição ausente é exercitada quatro vezes pelo registro de execução. Trata-se de um falso positivo por viés de amostragem — o corpus reúne simulações capturadas manualmente no simulador, e nada garante que toda transição legal tenha sido sorteada. Contra o oráculo analítico, completo por construção, o mesmo registro de execução obtém pontuação 100. O episódio fornece justificativa empírica para a escolha da enumeração exaustiva como fonte primária do oráculo, reservando o corpus ao papel de validação cruzada do carregador.

4.5 LIMITAÇÕES OBSERVADAS

Quatro limitações ficaram evidentes durante a avaliação. A primeira é a sensibilidade do sistema ao tempo de execução do programa sob teste, para a qual a Seção 4.4 oferece uma medida concreta: o defeito de instrumentação da referência, presente em dois de 76 mil registros, tinha cerca de 2% de chance de aparecer em um registro de execução de dez segundos e só foi exposto pelo registro de quinze minutos. Registros de execução curtos não exercitam defeitos raros; registros de execução longos elevam o custo da pipeline. A duração adequada depende do defeito que se

procura, e o sistema não oferece critério automático para escolhê-la.

A segunda é a granularidade da instrumentação. A checagem de transições pressupõe registro a cada evento de sincronização e reprova implementações que registram o estado em intervalos maiores, ainda que a sincronização em si esteja correta. Distinguir as duas situações exigiria aceitar, entre estados consecutivos do registro de execução, caminhos de comprimento maior que um no modelo — extensão possível do verificador, ao custo de enfraquecer o critério.

A terceira envolve as pontuações nulas por registro de execução não reconhecido: na pontuação agregada, o programa cuja saída não pôde ser interpretada é indistinguível do programa que viola todas as propriedades. O relatório individual registra a diferença, mas a pontuação a descarta.

A quarta é o detector de deadlock, que se baseia na configuração do último estado observável, e não em uma análise de ausência de progresso ao longo do registro de execução; implementações que atravessam um impasse parcial seguido de recuperação escapam à detecção. Permanece, por fim, a restrição do subconjunto de UPPAAL suportado pelo carregador, conforme descrito na Seção 3.11.

5 CONCLUSÃO

Este trabalho teve como objetivo geral o desenvolvimento de um sistema genérico de testes para software multitarefa, capaz de verificar software a partir de um modelo formal em UPPAAL e da análise dos registros de execução por elas produzidos. O objetivo foi atingido por meio da construção de uma pipeline de caixa-preta que avalia o programa sob teste pelos seus registros de execução — obtidos compilando e executando a implementação, ou recebidos diretamente — e os confronta com o espaço de estados derivado do modelo formal.

Cada um dos objetivos específicos foi cumprido conforme descrito a seguir. Os requisitos funcionais do sistema de testes foram levantados a partir da revisão da literatura (Capítulo 2) e das aplicações pretendidas, e consolidados no Quadro 4 (Seção 3.5). O estudo de caso do jantar dos filósofos foi modelado em UPPAAL e as propriedades temporais de interesse foram verificadas com o *verifyta*, conforme apresentado na Seção 3.8. A infraestrutura de compilação, execução e captura de registros de execução foi implementada no framework de verificação, conforme descrito na Seção 3.10. A comparação dos registros de execução contra o espaço de estados foi implementada no verificador e nas funções de checagem por propriedade, conforme a Seção 3.10.5. Por fim, o sistema foi validado com uma implementação de referência e doze implementações independentes do mesmo problema, conforme apresentado no Capítulo 4.

O conjunto dos resultados permite concluir que a abordagem proposta é viável como sistema de testes para programas concorrentes. A separação rigorosa entre lógica genérica do orquestrador e lógica específica do caso de estudo possibilita a adição de novos problemas de sincronização sem modificar o núcleo do sistema, desde que esses problemas caibam no subconjunto de UPPAAL suportado pelo carregador.

O sistema desenvolvido atingiu o objetivo proposto, mas deixa em aberto extensões naturais, organizadas aqui em duas frentes: o fechamento do ciclo de teste baseado em modelo, por meio da geração automática de casos de teste, e o enriquecimento do veredito com métricas de justiça.

Na sua forma atual, o sistema é um verificador passivo: ele recebe registros de execução — compilando e executando o programa sob teste ou consumindo registros de execução já prontos — e os confronta com o espaço de estados do modelo, sem empregar esse modelo para *produzir* os estímulos que conduzem a execução. Uma evolução natural é fechar o ciclo de teste baseado em modelo (Utting; Legeard, 2007; Utting *et al.*, 2012), derivando casos de teste do próprio espaço de estados: a cobertura, hoje calculada apenas como métrica de saída, passaria a ser o objetivo de uma busca que seleciona sequências de eventos capazes de levar o programa a regiões

ainda não exercitadas, à semelhança de abordagens que projetam conjuntos de teste voltados à detecção de faltas de concorrência (Cai *et al.*, 2018). Os contraexemplos concretos que o *verifyta* já gera durante a verificação são candidatos imediatos a casos de teste dirigidos — estratégia já explorada na geração de sequências de teste a partir de verificadores de modelo (Heimdahl; et al., 2004) — e as técnicas clássicas de derivação de casos a partir de máquinas de estado (Chow, 1978; Lee; Yannakakis, 1996) oferecem o arcabouço teórico para essa geração.

O principal obstáculo é o não-determinismo do escalonador, que torna incerta a reprodução de uma intercalação-alvo: gerar a sequência desejada não garante que o sistema operacional a execute. Superar essa barreira aproxima o trabalho das técnicas de exploração sistemática de intercalações, em que o próprio escalonamento é controlado para varrer as ordens de execução relevantes em vez de depender do acaso (Arora *et al.*, 2016), e das ferramentas de teste baseado em modelo que dirigem a execução do sistema sob teste a partir do modelo (Anier *et al.*, 2017). Na mesma direção, a campanha aqui relatada apoiou-se em uma única execução por implementação; uma extensão de menor custo é complementar essa execução única com uma caracterização estatística do veredito sobre muitas execuções — distribuição de cobertura e intervalo de confiança por propriedade — capaz de quantificar a confiança do resultado e de tornar comparáveis implementações hoje separadas por um único registro de execução afortunado.

A segunda frente parte da constatação de que o oráculo formal responde se um comportamento desejável é *possível*, e não se ele de fato *ocorreu*. A avaliação expôs essa distinção: uma das implementações satisfaz a propriedade de ausência de inanição — existe, a partir do estado final, uma execução futura em que cada filósofo come — e, ainda assim, quatro dos cinco filósofos jamais comem ao longo do registro de execução observado. Essa inanição empírica é hoje apenas diagnosticada, sem repercussão na pontuação. Um trabalho futuro é incorporar métricas quantitativas de justiça (*fairness*) — por exemplo, a distribuição das oportunidades de progresso entre as tarefas ao longo do registro de execução — e torná-las pontuáveis, de modo que o veredito combine o que o modelo prova ser possível com uma medida empírica do que a execução realizou, reforçando a tese de que modelo e execução são fontes de evidência complementares.

Por fim, a estrutura do sistema — avaliação em lote de submissões, pontuação agregada e relatório fundamentado por submissão — aproxima-o de uma ferramenta de correção automática. A isso soma-se a geração de grafos ao final da execução, que torna visível o comportamento observado do programa e auxilia o usuário na interpretação do resultado, em vez de restringir o retorno a um valor numérico.

5.1 ESCLARECIMENTOS

Durante a elaboração deste trabalho, foram utilizadas ferramentas de inteligência artificial generativa (especificamente o modelo Claude Opus 4.8, com janela de contexto de 1 milhão de tokens e configuração de esforço elevado), baseadas em grandes modelos de linguagem, como apoio às seguintes atividades: revisão linguística (ortografia, gramática, pontuação e estilo); sugestões de organização e de redação de trechos do texto; geração de figuras esquemáticas a partir de descrições do autor; auxílio na organização e na verificação de referências bibliográficas; e apoio à implementação, na forma de sugestões de código, depuração e refatoração de trechos.

A concepção do trabalho, a definição do problema, a modelagem formal, a arquitetura do sistema e as decisões de projeto, a condução dos experimentos e a análise dos resultados e conclusões são de autoria própria. O apoio à implementação restringiu-se a sugestões pontuais, depuração e refatoração, que foram revisadas, testadas e validadas pelo autor. Todo o conteúdo produzido com apoio dessas ferramentas foi conferido pelo autor, que assume integral responsabilidade pelo conteúdo final do trabalho.

REFERÊNCIAS

- ALUR, R.; DILL, D. L. A theory of timed automata. **Theoretical Computer Science**, v. 126, n. 2, p. 183–235, 1994.
- ANIER, A.; VAIN, J.; TSIPOULOS, L. Dtron: a tool for distributed model-based testing of time critical applications. **Proceedings of the Estonian Academy of Sciences**, v. 66, n. 1, p. 88, 2017. Disponível em: <https://doi.org/10.3176/proc.2017.1.08>. Acesso em: 22 jun. 2026.
- ARORA, V.; BHATIA, R.; SINGH, M. A systematic review of approaches for testing concurrent programs. **Concurrency and Computation: Practice and Experience**, v. 28, n. 5, p. 1572–1611, 2016. Disponível em: <https://doi.org/10.1002/cpe.3711>. Acesso em: 22 jun. 2026.
- BAIER, C.; KATOEN, J.-P. **Principles of model checking**. Cambridge, MA, EUA: MIT Press, 2008.
- BEHRMANN, G.; DAVID, A.; LARSEN, K. G. A tutorial on uppaal. *In: Formal Methods for the Design of Real-Time Systems* (SFM-RT 2004), LNCS 3185. Berlin, Heidelberg: Springer, 2004. p. 200–236.
- BUTTAZZO, G. C. **Hard real-time computing systems: predictable scheduling algorithms and applications**. 3. ed. Nova York: Springer, 2011.
- CAI, S.; GALLINA, B.; NYSTRÖM, D.; SECELEANU, C. Effective test suite design for detecting concurrency control faults in distributed transaction systems. *In: Leveraging Applications of Formal Methods, Verification and Validation* (ISoLA 2018), LNCS 11247. Cham: Springer, 2018. p. 355–374. Disponível em: https://doi.org/10.1007/978-3-030-03424-5_24. Acesso em: 22 jun. 2026.
- CHOW, T. S. Testing software design modeled by finite-state machines. **IEEE Transactions on Software Engineering**, SE-4, n. 3, p. 178–187, 1978.
- CLARKE, E. M.; EMERSON, E. A.; SISTLA, A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. **ACM Transactions on Programming Languages and Systems**, v. 8, n. 2, p. 244–263, 1986.
- COFFMAN, E. G.; ELPHICK, M. J.; SHOSHANI, A. System deadlocks. **ACM Computing Surveys**, v. 3, n. 2, p. 67–78, 1971.
- DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. **Introdução ao teste de software**. Rio de Janeiro: Elsevier, 2007.
- DESIKAN, S.; RAMESH, G. **Software testing: principles and practices**. 1. ed. Upper Saddle River, NJ, EUA: Pearson Education, 2006. ISBN 978-0-321-28629-9.
- DIJKSTRA, E. W. Hierarchical ordering of sequential processes. **Acta Informatica**, v. 1, n. 2, p. 115–138, 1971.

HARROW, J. J. Runtime checking of multithreaded applications with visual threads. *In: Proceedings of the 7th International SPIN Workshop on Model Checking of Software (SPIN 2000)*, LNCS 1885. Stanford, CA, EUA: Springer, 2000. p. 331–342. Disponível em: <https://api.semanticscholar.org/CorpusID:8773627>. Acesso em: 5 jul. 2024.

HARTONAS-GARMHAUSEN, V.; CLARKE, E. M.; CAMPOS, S. V. A. Deadlock prevention in flexible manufacturing systems using symbolic model checking. *In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 1996)*. Minneapolis, MN, EUA: IEEE, 1996. v. 1, p. 527–532. Disponível em: <https://api.semanticscholar.org/CorpusID:27108098>. Acesso em: 5 jul. 2024.

HEIMDAHL, M. P. E.; et al. Auto-generating test sequences using model checkers: a case study. *In: PETRENKO, A.; ULRICH, A. (Ed.). Formal Approaches to Software Testing (FATES 2003)*, LNCS 2931. Berlin, Heidelberg: Springer, 2004. p. 42–59.

HENARD, C.; et al. Comparing white-box and black-box test prioritization. *In: Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. Austin, TX, EUA: ACM, 2016. p. 523–534. Disponível em: <https://doi.org/10.1145/2884781.2884791>. Acesso em: 5 jul. 2024.

LAMPORT, L. Teaching concurrency. **ACM SIGACT News**, v. 40, n. 1, p. 58–62, 2009. Disponível em: <https://doi.org/10.1145/1515698.1515713>. Acesso em: 5 jul. 2024.

LEE, D.; YANNAKAKIS, M. Principles and methods of testing finite state machines: a survey. **Proceedings of the IEEE**, v. 84, n. 8, p. 1090–1123, 1996.

LEUCKER, M.; SCHALLHART, C. A brief account of runtime verification. **The Journal of Logic and Algebraic Programming**, v. 78, n. 5, p. 293–303, 2009.

MELO, S. M.; et al. Empirical research on concurrent software testing: a systematic mapping study. **Information and Software Technology**, v. 105, p. 226–251, 2019. Disponível em: <https://doi.org/10.1016/j.infsof.2018.08.017>. Acesso em: 5 jul. 2024.

NETO, A. C. D.; SUBRAMANYAN, R.; VIEIRA, M.; TRAVASSOS, G. H. A survey on model-based testing approaches: a systematic review. *In: Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies (WEASELTech '07)*. Atlanta, GA, EUA: ACM, 2007. p. 31–36. Disponível em: <https://doi.org/10.1145/1353673.1353681>. Acesso em: 11 jun. 2026.

TANENBAUM, A. S.; BOS, H. **Sistemas operacionais modernos**. 4. ed. São Paulo: Pearson Education do Brasil, 2016.

UPPAAL. **UPPAAL: integrated tool environment for modeling, validation and verification of real-time systems**. 2024. Ferramenta desenvolvida pela Aalborg University e pela Uppsala University. Disponível em: <https://uppaal.org/>. Acesso em: 11 jun. 2026.

UTTING, M.; LEGEARD, B. **Practical model-based testing: a tools approach**. San Francisco, CA, EUA: Morgan Kaufmann, 2007.

UTTING, M.; PRETSCHNER, A.; LEGEARD, B. A taxonomy of model-based testing approaches. **Software Testing, Verification and Reliability**, v. 22, n. 5, p. 297–312, 2012.

WILLIAMS, A. **C++ concurrency in action**. 2. ed. Shelter Island, NY, EUA: Manning, 2019.