UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FELIPE AUGUSTO DA SILVA MENDONÇA

Investigando a corretude de um sistema robótico via RoboChart: Um sistema de distribuição de medicamentos do mundo real

Recife

2025

FELIPE AUGUSTO DA SILVA MENDONÇA

Investigando a corretude de um sistema robótico via RoboChart: Um sistema de distribuição de medicamentos do mundo real

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Engenharia de Software e Linguagens de Programação

Orientador (a): Alexandre Cabral Mota

Coorientador (a): Madiel Conserva Filho

Recife

2025

.Catalogação de Publicação na Fonte. UFPE - Biblioteca Central

Mendonça, Felipe Augusto da Silva.
   Investigando a corretude de um sistema robótico via RoboChart: Um sistema de distribuição de medicamentos do mundo real / Felipe Augusto da Silva Mendonça. - Recife, 2025.
   80f.: il.

   Dissertação (Mestrado)- Universidade Federal de Pernambuco, Centro de Informática, Programa de Pós-Graduação em Ciência da Computação, 2025.
   Orientação: Alexandre Cabral Mota.
   Coorientação: Madiel Conserva Filho.

   1. RoboChart; 2. CSP; 3. Formalização de requisitos; 4. Verificação de conformidade; 5. Refinamento de Traces. I. Mota, Alexandre Cabral. II. Conserva Filho, Madiel. III. Título.

UFPE-Biblioteca Central

Felipe Augusto da Silva Mendonça

“Investigando a corretude de um sistema robótico via RoboChart: Um sistema de distribuição de medicamentos do mundo real”

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação. Área de Concentração: Engenharia de Software e Linguagens de Programação.

Aprovado em: 22/08/2025.

BANCA EXAMINADORA

_____

Prof. Dr. Gustavo Henrique Porto de Carvalho
Centro de Informática / UFPE

_____

Prof. Dr. Sidney de Carvalho Nogueira
Departamento de Computação / UFRPE

_____

Prof. Dr. Alexandre Cabral Mota
Centro de Informática/UFPE
(**orientador**)

Dedico esta dissertação à minha mãe, cuja força, esforço, sabedoria e amor incondicional foram meu alicerce em cada etapa desta jornada.

AGRADECIMENTOS

A realização desta dissertação só foi possível graças ao apoio e à presença constante de pessoas fundamentais ao longo dessa trajetória. Primeiramente, agradeço profundamente à minha mãe, cuja força, dedicação, sabedoria e amor incondicional sempre foram meu maior alicerce. Sua presença firme e generosa me sustentou em cada passo deste percurso acadêmico e pessoal.

Agradeço também à minha família pelo apoio contínuo, pela paciência nos momentos de ausência e pelos inúmeros sacrifícios feitos para que eu pudesse seguir em busca dos meus sonhos. À minha namorada, expresso minha gratidão pelo carinho, pela compreensão e pelo incentivo incansável nos momentos mais difíceis.

Agradeço profundamente ao meu orientador, Professor Alexandre Mota, pela orientação precisa, pelos ensinamentos valiosos e pela confiança depositada ao longo desta jornada. Sua experiência e dedicação foram fundamentais para o amadurecimento deste trabalho. Estendo meus agradecimentos a Madiel Conserva, cuja ajuda generosa, disponibilidade constante e apoio técnico foram indispensáveis em diversos momentos do desenvolvimento da dissertação. Sou igualmente grato a Sidney Nogueira, Gustavo Carvalho e Juliano Iyoda pelas considerações pertinentes e pelos comentários valiosos feitos durante discussões em fases anteriores da pesquisa, que contribuíram para o amadurecimento das ideias aqui desenvolvidas.

Registro ainda meu sincero agradecimento ao projeto CRIAR, do Centro de Informática da Universidade Federal de Pernambuco, pela oportunidade e suporte oferecidos ao longo da minha formação. Sou especialmente grato a todos os funcionários e colaboradores do projeto, cuja dedicação, competência e acolhimento contribuíram significativamente para meu desenvolvimento acadêmico e profissional.

RESUMO

A crescente complexidade dos sistemas de controle em robôs autônomos exige métodos rigorosos para assegurar a conformidade entre especificações e implementações, especialmente em contextos críticos, como a dispensação de medicamentos. Este trabalho apresenta uma abordagem baseada na formalização de requisitos usando RoboChart (uma notação gráfica para a modelagem de sistemas robóticos), que possui uma semântica formal definida na álgebra de processos CSP. A metodologia proposta inclui a obtenção e abstração do LTS proveniente da semântica CSP de um modelo RoboChart e a verificação de conformidade por meio de um algoritmo próprio simplificado de verificação de refinamento de traces, permitindo identificar inconsistências entre especificações formais e implementações práticas desenvolvidas em Python. A abordagem foi aplicada em um sistema robótico de dispensação de medicamentos do Hospital das Clínicas da UFPE (HC-UFPE), desenvolvido no âmbito do projeto CRIAR — Centro de Robótica e Inteligência Artificial Responsável. O sistema integra controle de braço robótico e visão computacional. Os resultados indicam que a abordagem facilita a detecção de erros e promove um desenvolvimento mais robusto. Como contribuições principais, destacam-se: uma sistemática de formalização de requisitos informais utilizando RoboChart; o desenvolvimento de um algoritmo próprio para verificação de refinamento de traces e a aplicação da metodologia em dois estudos de caso.

Palavras-chaves: RoboChart; CSP; Formalização de requisitos; Verificação de conformidade; Refinamento de Traces.

ABSTRACT

The increasing complexity of control systems in autonomous robots demands rigorous methods to ensure conformance between specifications and implementations, especially in critical contexts such as medication dispensing. This paper presents an approach based on formalizing requirements using RoboChart (a graphical notation for modeling robotic systems), which has formal semantics defined in the CSP process algebra. The proposed methodology includes obtaining and abstracting the LTS from the CSP semantics of a RoboChart model and verifying compliance using a proprietary simplified trace refinement checking algorithm, allowing for the identification of inconsistencies between formal specifications and practical implementations developed in Python. The approach was applied to a robotic medication dispensing system at the *Hospital das Clínicas* of the (HC-UFPE), developed within the scope of the CRIAR project — Center for Responsible Robotics and Artificial Intelligence. The system integrates robotic arm control and computer vision. The results indicate that the approach facilitates error detection and promotes more robust development. The main contributions include: a systematic formalization of informal requirements using RoboChart; the development of a proprietary algorithm for verifying trace refinement, and the application of the methodology in two case studies.

Keywords: RoboChart; CSP; Requirements formalization; Conformance verification; Trace refinement.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

A crescente complexidade dos sistemas de controle em robôs autônomos exige o desenvolvimento de abordagens rigorosas para garantir a conformidade das implementações com os requisitos especificados. À medida que a robótica avança, esses sistemas tornam-se cada vez mais sofisticados, integrando sensores, atuadores, sistemas de percepção e algoritmos avançados de tomada de decisão que operam de forma autônoma em ambientes dinâmicos, desestruturados e, muitas vezes, imprevisíveis (THRUN, 2002). Além disso, o aumento da autonomia exige que os sistemas consigam lidar com situações não previstas durante o desenvolvimento, tomando decisões seguras em tempo real (ALUR, 2015). A precisão e a confiabilidade desses sistemas não são somente desejáveis, mas essenciais, sobretudo quando se considera sua aplicação em cenários críticos, onde qualquer falha pode gerar não somente prejuízos financeiros, mas também riscos à integridade física de pessoas e à continuidade de processos essenciais (LEVESON, 2016).

Sistemas robóticos, frequentemente modelados por máquinas de estado, desempenham papéis centrais em setores onde a segurança, a previsibilidade e a precisão são absolutamente indispensáveis (CASSANDRAS; LAFORTUNE, 2008). Hospitais, linhas de produção industrial, instalações de geração de energia e centros de pesquisa científica são exemplos de ambientes onde tais sistemas operam em condições rigorosas (LYNCH; PARK, 2017). Nessas aplicações, qualquer falha, por menor que seja, pode ter consequências severas, tanto do ponto de vista operacional — com paralisações, perdas de produtividade e danos materiais — quanto em termos de segurança, colocando em risco operadores, pacientes e o próprio ambiente (LEVESON, 2016). Por exemplo, robôs hospitalares que realizam tarefas sensíveis, como a dispensação automática de medicamentos ou a esterilização de ambientes, devem operar com total exatidão e rastreabilidade (MURPHY, 2019). Qualquer erro nesse contexto não somente compromete o funcionamento do sistema, interrompendo fluxos hospitalares, como também pode colocar vidas humanas em risco diretamente, seja pela administração incorreta de um fármaco, seja pela falha em protocolos de biossegurança (TENNER, 2015).

Esses cenários ilustram de forma clara e contundente a importância de garantir que o comportamento de um sistema robótico esteja em total conformidade com suas especificações, desde as fases iniciais de projeto até a implementação final. Essa necessidade vai além de uma boa prática de engenharia; ela se torna um imperativo técnico, ético e regulatório em

muitos setores. Nesse contexto, torna-se evidente que o desenvolvimento de robôs confiáveis e seguros representa, atualmente, um dos maiores desafios contemporâneos na engenharia de software aplicada a sistemas robóticos (CAVALCANTI et al., 2021). A crescente dependência desses sistemas para executar tarefas críticas, anteriormente realizadas exclusivamente por seres humanos, somente intensifica a urgência por soluções que assegurem sua corretude formal, sua robustez operacional e sua capacidade de se adaptar seguramente a ambientes complexos e dinâmicos (SCHNEIDER; SHABOLT; TAYLOR, 2004).

Este trabalho foi desenvolvido no contexto do Centro de Robótica e Inteligência Artificial Responsável (CRIAR) do Centro de Informática da UFPE, iniciativa voltada à pesquisa aplicada e ao desenvolvimento de soluções tecnológicas para problemas reais enfrentados por instituições públicas e privadas do Brasil. Um dos projetos conduzidos nesse Centro é a automação do processo de dispensação de medicamentos no Hospital das Clínicas da UFPE, onde sistemas robóticos desempenham papéis críticos na organização, separação e entrega de fármacos. A segurança e a confiabilidade dessas aplicações são essenciais, não somente para evitar falhas operacionais, mas também para proteger a integridade de pacientes e profissionais de saúde. A motivação deste trabalho surgiu diretamente da necessidade de garantir que os controladores robóticos desenvolvidos para esse ambiente estejam em conformidade com os requisitos estabelecidos — muitos dos quais definidos de forma informal e passíveis de interpretação ambígua. A Figura 1 ilustra o arranjo físico do sistema real utilizado, que serviu como base para um dos estudos de caso desenvolvido neste trabalho.

Apesar da criticidade envolvida em aplicações robóticas como a dispensação automatizada de medicamentos, muitos projetos de sistemas robóticos ainda são desenvolvidos a partir de especificações informais, frequentemente pouco estruturadas e suscetíveis a interpretações divergentes. Na prática, é comum que essas especificações estejam registradas em documentos textuais genéricos, em diagramas que carecem de rigor semântico ou até mesmo sejam transmitidas por meio de comunicações verbais e informais entre membros da equipe de desenvolvimento (WIEGERS; BEATTY, 2013). Esse tipo de abordagem, embora bastante difundido na indústria devido à sua aparente simplicidade e flexibilidade, introduz um grau significativo de ambiguidade no processo de desenvolvimento (LAMSWEERDE, 2009). Essa ambiguidade, por sua vez, abre margem para que diferentes desenvolvedores interpretem os requisitos de maneiras distintas, levando a implementações que, apesar de aparentemente corretas, podem divergir dos comportamentos originalmente desejados (MALL, 2018).

Esse cenário torna-se ainda mais crítico ao considerar a natureza sensível de muitos sis-

Figura 1 – Arranjo físico do sistema de dispensação de medicamentos

Fonte: Elaborada pelo autor (2025)

temas robóticos, nos quais qualquer desvio em relação às especificações pode resultar em consequências graves, tanto do ponto de vista operacional quanto da segurança (LEVESON, 2016). A ausência de uma formalização precisa dos requisitos compromete não somente a implementação correta, mas também a capacidade do sistema de ser auditado, validado e certificado por órgãos regulatórios (HINCHEY; BOWEN, 2012). Além disso, os sistemas robóticos modernos tendem a ser cada vez mais distribuídos, heterogêneos e interativos, com múltiplos componentes operando concorrentemente e se comunicando frequentemente de maneira assíncrona. Esse tipo de arquitetura introduz um conjunto complexo de interdependências e abre espaço para o surgimento de propriedades emergentes — comportamentos que não podem ser previstos a partir da análise isolada dos componentes, mas resultantes das interações entre eles. Essas propriedades são particularmente difíceis de prever e controlar com as abordagens tradicionais de engenharia de software, reforçando a necessidade do uso de métodos formais para lidar com tais desafios de forma sistemática e verificável (MITCHELL, 2006; LEVESON, 2016).

Diante desses desafios, os métodos formais emergem como uma abordagem não somente robusta, mas cada vez mais indispensável para garantir que sistemas robóticos operem em estrita conformidade com seus requisitos funcionais e não funcionais. Esses métodos oferecem uma base matemática rigorosa para a descrição precisa do comportamento esperado dos sistemas e, consequentemente, para a realização de verificações sistemáticas de sua correção. Entre as abordagens mais consolidadas, destaca-se Communicating Sequential Processes (CSP) (HOARE et al., 1985; ROSCOE, 1998), que fornece uma estrutura teórica extremamente poderosa para modelagem de sistemas concorrentes, permitindo representar formalmente tanto os processos individuais quanto os padrões de comunicação e sincronização entre eles. CSP permite descrever, com precisão, não somente o comportamento interno de cada componente, mas também as interações complexas que ocorrem em sistemas distribuídos e concorrentes, oferecendo, assim, um modelo coerente e matematicamente validável.

O uso de CSP, aliado a ferramentas como FDR (Failures-Divergence Refinement), permite não somente descrever formalmente os comportamentos esperados de sistemas concorrentes, mas também aplicar técnicas rigorosas de verificação, como model checking e análise de refinamento. O model checking possibilita a exploração automática de todos os estados possíveis do sistema para verificar se certas propriedades são satisfeitas, enquanto o refinamento permite demonstrar, formalmente, que uma implementação está em conformidade com uma especificação abstrata, garantindo que os comportamentos desejados sejam preservados ao longo do desenvolvimento. Essas ferramentas também oferecem mecanismos de geração de contraexemplos, facilitando a identificação e correção de falhas no modelo.

Tais técnicas são essenciais para detectar, ainda nas fases iniciais do projeto, problemas como deadlocks, livelocks, não determinismo, violação de propriedades de segurança, inconsistências nos protocolos de comunicação e falhas no cumprimento de requisitos temporais. Ao permitir a identificação precoce desses erros, os métodos formais evitam que problemas custosos avancem para etapas posteriores do desenvolvimento, promovendo maior robustez, previsibilidade e segurança no sistema final. Nesse contexto, os métodos formais deixam de ser ferramentas meramente acadêmicas para se consolidarem como elementos indispensáveis no ciclo de vida de sistemas críticos — especialmente na robótica, onde a confiabilidade operacional não é somente desejável, mas essencial.

Domain-Specific Languages (DSLs) também desempenham um papel fundamental no desenvolvimento de sistemas robóticos ao oferecerem abstrações especializadas que reduzem a complexidade da modelagem e implementação. Diversas DSLs foram propostas com diferentes

enfoques: RobotML (DHOUIB et al., 2012) abstrai detalhes de baixo nível por meio de uma arquitetura baseada em componentes; $G^{en}oM$ (FLEURY; HERRB; CHATILA, 1997) e seu sucessor $G^{en}{}_oM3$ (MALLET et al., 2010) possibilitam a geração automática de código e a definição de propriedades temporais; ORCCAD (BORRELLY et al., 1998) oferece suporte a controle em tempo real com verificação formal; e o RoboFlow (ALEXANDROVA; TATLOCK; CAKMAK, 2015) prioriza a acessibilidade por meio de uma interface gráfica baseada em fluxos. Cada uma dessas linguagens apresenta vantagens e limitações que as tornam mais ou menos adequadas conforme o contexto de aplicação. Enquanto RobotML e RoboFlow favorecem rapidez e simplicidade no desenvolvimento, $G^{en}oM$ e ORCCAD oferecem maior controle e rigor em ambientes com restrições temporais e requisitos críticos de sincronização.

Nesse cenário, RoboChart (MIYAZAWA et al., 2016) destaca-se como uma DSL voltada à modelagem de sistemas robóticos que alia uma notação gráfica intuitiva a uma fundamentação formal rigorosa baseada na teoria de processos concorrentes CSP. Ela permite especificar formalmente sistemas de controle, realizar análises com ferramentas como FDR e gerar código automaticamente a partir dos modelos. Com essa integração entre clareza visual e verificação matemática, RoboChart proporciona uma abordagem robusta para o desenvolvimento de sistemas robóticos críticos. Sua capacidade de aproximar requisitos informais de uma modelagem precisa permite detectar inconsistências antecipadamente. No entanto, sua aplicação em sistemas maiores revelou limitações de escalabilidade, com aumento no custo computacional das verificações, o que pode dificultar a análise contínua e a validação eficiente do comportamento frente à implementação real.

Para mitigar essas limitações, a estratégia adotada, apresentada no Capítulo 3, foi restringir o escopo da modelagem, concentrando-se nas Máquinas de Estados de RoboChart, que representam o núcleo do comportamento sequencial dos sistemas. Além disso, a análise foi aplicada somente a um subconjunto bem definido do sistema, suficiente para capturar os requisitos críticos. Essa abordagem tornou a verificação mais viável, preservando as propriedades comportamentais essenciais e mantendo a rastreabilidade entre os modelos e a implementação. Complementarmente, o desenvolvimento de um algoritmo de refinamento de trace contribuiu para garantir que o comportamento observado na prática estivesse alinhado com a especificação formal.

Embora estudos anteriores (MIYAZAWA et al., 2016; MIYAZAWA et al., 2017; LI et al., 2024; DAROLTI, 2019; MURRAY et al., 2022) demonstrem a eficácia de RoboChart na verificação de controladores em sistemas críticos, observa-se que muitos deles acabam não abordando de

forma sistemática uma etapa fundamental: a tradução dos requisitos informais em modelos formais. Esse processo, frequentemente negligenciado, é crucial para assegurar que o modelo represente fielmente as necessidades operacionais do sistema, evitando que modelos formalmente corretos se afastem dos objetivos práticos do projeto.

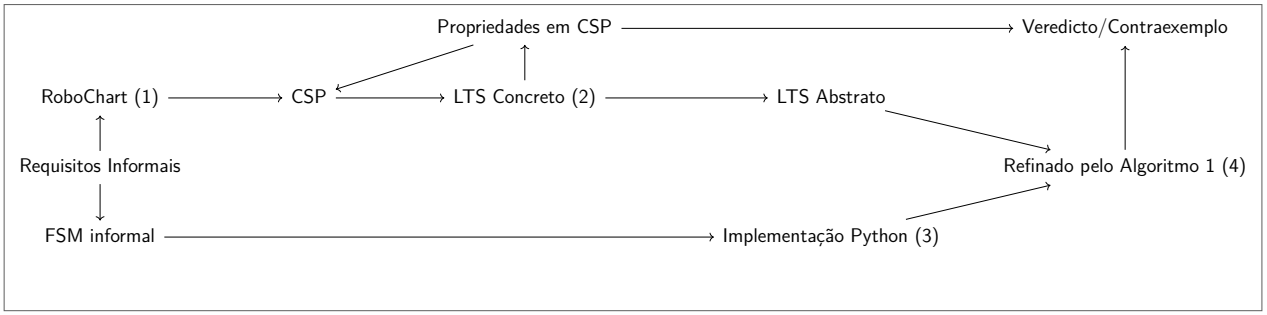
Diante desse cenário, a aplicação de RoboChart neste trabalho não se restringiu à verificação formal isolada, mas também buscou enfrentar desafios práticos relacionados à formalização de requisitos e à escalabilidade dos modelos. A experiência adquirida durante o desenvolvimento reforça que, com estratégias adequadas de delimitação de escopo e suporte de ferramentas complementares, é possível aplicar métodos formais de maneira eficiente no desenvolvimento de sistemas robóticos complexos. Esse contexto motivou a construção de uma abordagem específica para este trabalho, cuja concepção, estrutura e resultados serão detalhados a seguir na seção de contribuições.

1.1 CONTRIBUIÇÕES DESTE TRABALHO

Este trabalho combina modelagem formal e verificação automatizada para assegurar a conformidade e aprimorar a correção de sistemas de controle robótico, permitindo a identificação e resolução precoce de inconsistências entre especificação e implementação. A metodologia desenvolvida apoia-se em quatro pilares principais: (1) a formalização de requisitos informais por meio de RoboChart, que gera especificações formais em CSP; (2) a utilização do verificador de refinamentos do FDR para derivar um Sistema de Transição Rotulado (Labelled Transition Systems - LTS) a partir da especificação CSP; (3) o comportamento implementado utilizando máquinas de estados finitos em Python, por meio da biblioteca pytransitions, cuja execução é interpretada como um LTS; e (4) a aplicação do algoritmo proposto de refinamento de traces para comparar o LTS gerado por RoboChart com as máquinas de estados da implementação, verificando a conformidade formal entre eles, seguindo a definição de refinamento de traces (ROSCOE, 1998; ROSCOE, 2010). Essa abordagem permite reduzir ambiguidades típicas de requisitos informais, fortalecendo o processo de desenvolvimento por meio de verificações formais automatizadas que garantem validação sistemática e iterativa, alinhando a implementação com os requisitos e aumentando a confiabilidade do sistema. A abordagem proposta está resumida na Figura 2.

A relevância da metodologia proposta reside em sua aplicabilidade prática e no avanço que oferece em relação às abordagens existentes. Diferentemente de trabalhos que utilizam

Figura 2 – Fluxo de verificação e refinamento do modelo.



Fonte: Elaborada pelo autor (2025)

métodos formais para análise de modelos de sistemas robóticos, este estudo se concentra na verificação diretamente da implementação, em relação aos requisitos esperados. Além disso, o uso de RoboChart (e sua ferramenta de apoio RoboTool[1]) como base para a modelagem formal do comportamento do software, combinada com um algoritmo próprio de análise de refinamento inspirado na definição clássica de refinamento de traces em CSP, oferece uma solução robusta e inovadora para a verificação de conformidade em sistemas de controle distribuídos. Essa abordagem personalizada permite lidar com as particularidades das implementações baseadas em máquinas de estado em Python, ampliando a confiabilidade e a precisão da verificação em contextos reais. As principais contribuições deste trabalho incluem:

- Formalização de requisitos informais de dois estudos de caso usando RoboChart;
- Uma metodologia que mostra como garantir que uma implementação baseada em pytransitions esteja em conformidade com um sistema robótico descrito por requisitos informais usando a linguagem formal RoboChart;
- Um algoritmo de verificação de modelo para realizar a verificação de refinamento de traces CSP do LTS abstrato de um modelo RoboChart em relação à sua implementação, construída usando a biblioteca pytransitions;
- Aplicação do algoritmo proposto em dois estudos de caso, evidenciando discrepâncias em alguns casos entre os resultados esperados e os resultados obtidos.

A abordagem proposta melhora o desenvolvimento dos sistemas ao permitir que inconsistências sejam identificadas e corrigidas no estágio inicial do desenvolvimento, economizando recursos durante a fase de integração. Ela também promove maior segurança ao garantir que

[1] https://robostar.cs.york.ac.uk/robotool/

comportamentos críticos sejam verificados formalmente, reduzindo o risco de falhas catastróficas. Por fim, a eficiência é aprimorada por meio da automação no processo de verificação e refinamento, permitindo iterações mais rápidas e confiáveis no desenvolvimento de sistemas robóticos. Os resultados dessa pesquisa foram submetidos para publicação no periódico especializado *The Journal of Systems & Software* (MENDONÇA; CONSERVA; MOTA, 2025).

1.2 ORGANIZAÇÃO DA DISSERTAÇÃO

O restante deste trabalho está organizado da seguinte forma: o Capítulo 2 apresenta a fundamentação teórica necessária, incluindo conceitos de CSP e RoboChart; o Capítulo 3 descreve a metodologia adotada, abordando a formalização de requisitos, a abstração de LTS e o algoritmo de refinamento de traces; o Capítulo 4 detalha os estudos de caso aplicados, evidenciando as validações realizadas, os resultados obtidos, a discussão crítica desses resultados, além das ameaças à validade e das medidas para mitigá-las; o Capítulo 5 explora os trabalhos relacionados e como eles se conectam a este estudo; e, por fim, o Capítulo 6 apresenta as considerações finais, ressaltando as principais contribuições e sugerindo direções para pesquisas futuras.

2 FUNDAMENTAÇÃO TEÓRICA

O desenvolvimento de sistemas críticos, como os utilizados em robótica, requer o uso de ferramentas e métodos que garantam precisão e confiabilidade desde as etapas iniciais do projeto. Para isso, linguagens formais e técnicas de verificação desempenham um papel essencial, permitindo modelar, especificar e validar rigorosamente os comportamentos do sistema. Nesse contexto, destaca-se RoboChart, uma notação gráfica projetada para descrever controladores robóticos de forma estruturada, integrando elementos de estados, eventos e temporização típicos da robótica. Uma das suas principais forças reside na associação com CSP, uma linguagem formal voltada à modelagem de sistemas concorrentes. RoboChart utiliza CSP como base semântica formal, o que permite que modelos desenvolvidos visualmente possam ser traduzidos para uma representação matemática precisa, possibilitando a verificação automática de propriedades. Este capítulo começa introduzindo CSP, abordando sua semântica denotacional, conceitos de refinamento e ferramentas de suporte como FDR, destacando seu papel na modelagem e verificação de sistemas concorrentes. Em seguida, apresenta RoboChart, explorando sua estrutura, capacidades de modelagem e como sua semântica, baseada em CSP, permite a especificação rigorosa e a análise formal de sistemas robóticos.

Por fim, este capítulo também aborda a biblioteca *pytransitions*, uma ferramenta amplamente utilizada para a implementação de Máquinas de Estados Finitos em Python. Embora não constitua uma linguagem formal, a *pytransitions* exerce um papel fundamental na etapa de implementação, ao permitir que os modelos formalmente especificados sejam convertidos em sistemas funcionais e robustos.

2.1 COMMUNICATING SEQUENTIAL PROCESSES (CSP)

Communicating Sequential Processes (CSP) (HOARE et al., 1985; ROSCOE, 1998) é uma linguagem formal utilizada para modelar e analisar sistemas concorrentes. Desenvolvida inicialmente por Tony Hoare e posteriormente modificada por Roscoe, CSP permite a especificação de processos que interagem por meio de eventos atômicos, sincronizando-se somente quando todos os participantes estão prontos. Essa abordagem é especialmente útil para sistemas distribuídos e concorrentes, como linhas de montagem e sistemas robóticos, ao possibilitar a definição rigorosa de propriedades críticas e a verificação automática da correção de suas

implementações.

CSP utiliza os chamados processos para representar componentes do sistema que realizam interações observáveis. Essas interações são modeladas por canais simples (ou eventos) ou canais que carregam dados (ROSCOE, 2010). Por exemplo, uma declaração como `channel a` representa um evento, uma declaração como `channel b: Bool`, permite comunicar os valores True e False por meio do canal b, dando origem aos eventos complexos *b.True* ou *b.False*. Essa flexibilidade permite modelar trocas de informações síncronas entre processos, o que é essencial para capturar a complexidade de sistemas que requerem comunicação estruturada.

Tabela 1 – Operadores CSP.

Operador	**Sintaxe**	**Descrição**
Terminação com sucesso	SKIP	O processo que termina imediatamente
Deadlock	STOP	O processo que não aceita eventos e, portanto, gera deadlocks
Prefixação Simples	a $\rightarrow$ P	Comunica o evento a e age como o processo P
Composição Sequencial	P ; Q	Executa os processos P e depois Q em sequência
Escolha Externa	P $\Box$ Q	Oferece uma escolha entre dois processos P e Q
Ecolha Interna	P $\sqcap$ Q	A escolha é arbitrária, sem influência do ambiente.
Composição Paralela	P $\|_A$ Q	Executa P e Q simultaneamente, sincronizando o evento compartilhado a.
Ocultação	P $\backslash A$	Executa o processo P, mas eventos do conjunto A não aparecem no trace.

Fonte: Elaborada pelo autor (2025)

Os operadores de CSP definem as interações e o comportamento dos processos, possibilitando modelar sistemas com diferentes níveis de complexidade. Entre os operadores principais expostos na Tabela 1 estão o de prefixação ($a \rightarrow P$), que descreve que um processo realiza o evento a antes de continuar como P, incluindo casos como *a?x!y*, onde a comunicação ocorre por meio de canais, permitindo que a receba um valor x e envie y para outro processo. O operador de composição sequencial ($P;Q$), onde P é termina com sucesso antes de iniciar Q. A escolha externa ($P \Box Q$) e interna ($P \sqcap Q$) definem comportamentos alternativos, sendo que no primeiro caso o ambiente escolhe, e no segundo, a escolha ocorre internamente ao sistema. O operador de composição paralela $P \|_A Q$ permite que os processos P e Q sejam executados em paralelo, sincronizando-se em todos os eventos do conjunto A. Por fim, o operador de ocultação ($P \setminus A$) tornam internos os eventos do conjunto A, escondendo-os do ambiente externo, facilitando o gerenciamento de complexidade em sistemas distribuídos (ROSCOE, 1998).

A especificação CSP a seguir (Figura 3) representa um modelo de controle para um robô móvel capaz de se mover de forma autônoma, detectando obstáculos e ajustando sua trajetória de acordo.

Figura 3 – Especificação CSP para controle de robôs móveis.

$$
\begin{aligned}
&\texttt{SMovement} = \texttt{Moving} \\
&\texttt{Moving} = \texttt{moveCall.lv.0} \rightarrow \texttt{Obstacle} \\
&\texttt{Obstacle} = \texttt{obstacle.in} \rightarrow \texttt{stopCall} \rightarrow \texttt{Turning} \\
&\texttt{Turning} = \texttt{moveCall.0.av} \rightarrow \texttt{Moving}
\end{aligned}
$$

Fonte: Elaborada pelo autor (2025)

Essa especificação utiliza a prefixação ($a \rightarrow P$) para definir a ordem das ações do robô, garantindo que ele execute um evento antes de transicionar para outro estado, que nesse caso é representado por um processo. A estrutura sequencial do modelo assegura que o robô primeiro recebe o comando de movimento (*moveCall.lv.0*) antes de transitar para a detecção de obstáculo (*Obstacle*), e, ao detectar um obstáculo, executa o comando de parada (*stopCall*) antes de iniciar a manobra de reorientação (*Turning*). Neste código, lv e av denotam a velocidade linear e angular do robô, respectivamente. A repetição do ciclo entre os estados *Moving* e *Turning* representa a continuidade do comportamento do robô, modelando um fluxo dinâmico e adaptativo.

2.1.1 Semântica Denotacional e Refinamento

A semântica denotacional de CSP estabelece um vínculo rigoroso entre a sintaxe dos processos e seu comportamento observável, por meio de uma interpretação matemática abstrata que associa a cada processo um conjunto de comportamentos possíveis. Ao contrário da semântica operacional, que descreve como os estados de um sistema evoluem ao longo do tempo, a semântica denotacional concentra-se na caracterização do "significado" de um processo de forma composicional, ou seja, a semântica de uma construção composta pode ser deduzida diretamente a partir das semânticas de suas partes. Essa abordagem proporciona uma base sólida para análise formal, permitindo a verificação de propriedades como segurança, ausência de deadlocks e correção funcional por meio de refinamento.

No contexto de CSP, diversos modelos denotacionais foram desenvolvidos, sendo os mais comuns os baseados em *traces*, *failures* e *divergences*. Cada um desses modelos captura diferentes aspectos do comportamento de um processo. O modelo de *traces* foca exclusivamente na sequência de eventos visíveis que o processo pode realizar, abstraindo falhas e comportamentos internos. O modelo de *failures* amplia essa visão incluindo os conjuntos de eventos

que um processo pode recusar após determinado trace, capturando informações sobre disponibilidade e sincronização com o ambiente. Por fim, o modelo de *failures-divergences* considera também o fenômeno de divergência, ou seja, a possibilidade de o processo entrar em um ciclo infinito de ações internas não observáveis, representando perda de controle ou bloqueio interno.

O modelo de traces é o mais simples e serve como ponto de partida para a compreensão dos modelos mais complexos. Nesse contexto, o comportamento de um processo é descrito pelo conjunto de sequências finitas de eventos observáveis que ele pode executar, representadas por palavras sobre o alfabeto de eventos Σ. A função *traces*, definida de forma indutiva sobre a estrutura dos processos, associa a cada processo CSP o conjunto de suas possíveis execuções observáveis. Para o operador de prefixo, por exemplo, a definição formal da função é apresentada na Figura 4.

Figura 4 – Definição formal da função *traces* para o operador de prefixo no CSP.

$$\mathit{traces}(a \rightarrow P) = \{\langle\rangle\} \cup \{\langle a\rangle \frown tr \mid tr \in \mathit{traces}(P)\}$$

Fonte: Elaborada pelo autor (2025)

A equação exposta na Figura 4 expressa que o processo $a \rightarrow P$ pode inicialmente não realizar nenhuma ação (daí o trace vazio), ou pode realizar o evento a e então continuar se comportando como o processo P, concatenando os traces de P após a ocorrência de a. Com base nessa definição, é possível derivar os traces dos processos básicos. Por exemplo, o processo $STOP$, que não realiza nenhuma ação, possui somente o trace vazio, ou seja, $traces(STOP) = \{\langle\rangle\}$. Já o processo $a \rightarrow STOP$ possui dois traces: o trace vazio, e o trace $\langle a\rangle$, que corresponde à execução do evento a seguido da terminação imediata. Portanto, na Figura 5 temos:

Figura 5 – Exemplo da função *traces* aplicada ao processo $a \rightarrow STOP$.

$$traces(a \rightarrow STOP) = \{\langle\rangle, \langle a\rangle\}$$

Fonte: Elaborada pelo autor (2025)

Esses exemplos ilustram como a definição formal da função *traces* permite construir, de forma composicional, o comportamento de processos arbitrários. Em termos gerais, o modelo de *traces* oferece uma perspectiva essencialmente sequencial do sistema, desconsiderando as-

pectos como recusas ou divergências, mas já possibilitando a análise de propriedades (safety), como a ordem correta dos eventos e o alcance de determinadas ações. Para um tratamento mais aprofundado das definições formais e do desenvolvimento do modelo de *traces*, recomenda-se a leitura das obras de Roscoe, em especial *The Theory and Practice of Concurrency* e *Understanding Concurrent Systems*, que apresentam uma abordagem abrangente com definições rigorosas e exemplos ilustrativos (ROSCOE, 1998; ROSCOE, 2010).

A noção de refinamento emerge naturalmente a partir dessa semântica. Refinar um processo significa restringir seus comportamentos, garantindo que seu comportamento esteja restrito ao comportamento da especificação. No modelo de *traces*, um processo Q é considerado uma implementação válida de uma especificação P se todo comportamento de Q também é permitido por P, ou seja, $traces(Q) \subseteq traces(P)$. Em notação formal, diz-se que Q refina P no modelo de traces, representado por $P \sqsubseteq_T Q$. Tal relação expressa que Q é, do ponto de vista do comportamento observável, mais determinístico ou mais restrito que P, preservando todas as permissividades da especificação original.

A aplicação do refinamento denotacional é especialmente útil em contextos nos quais a ordem e a ocorrência de eventos são determinantes para a correção funcional do sistema. Considere, por exemplo, o modelo de controle de um robô móvel autônomo. Nesse sistema, eventos como detecção de obstáculo, parada e rotação devem ocorrer em uma ordem específica para garantir uma navegação segura. O conjunto de traces associado à especificação descreve os ciclos completos de movimentação, onde o robô inicia o deslocamento, detecta um obstáculo, realiza uma parada e executa uma manobra de desvio antes de retomar o movimento. A Figura 6 ilustra esse conjunto de comportamentos esperados.

Figura 6 – Traces de uma execução cíclica do sistema.

$$\{\langle\rangle, \langle moveCall.lv.0\rangle, \langle moveCall.lv.0, obstacle.in\rangle,$$
$$\langle moveCall.lv.0, obstacle.in, stopCall\rangle,$$
$$\langle moveCall.lv.0, obstacle.in, stopCall, moveCall.0.av\rangle, ...\}$$

Fonte: Elaborada pelo autor (2025)

Se uma implementação Q executa, por exemplo, os eventos *moveCall.lv.0*, *obstacle.in* e, em seguida, *moveCall.0.av*, omitindo o evento *stopCall*, essa sequência não pertence ao conjunto de traces da especificação P, violando a ordem definida e, portanto, invalidando o refinamento. Tal violação evidencia que $P \not\sqsubseteq_T Q$, indicando que a implementação falha em seguir a lógica

de controle prescrita. Em sistemas críticos como esse, o refinamento de traces atua como um critério formal para assegurar que a implementação não introduza comportamentos inesperados ou inseguros.

Embora o modelo de *traces* seja intuitivo e útil para capturar sequências válidas de eventos, ele não é suficiente para caracterizar aspectos como a capacidade do processo de se recusar a cooperar com o ambiente (por exemplo, quando não oferece certas ações) ou de entrar em ciclos internos infinitos. Por esse motivo, modelos mais ricos, como o de *failures-divergences*, são geralmente preferidos na verificação formal. No entanto, o refinamento no modelo de *traces* já oferece garantias importantes para sistemas determinísticos ou com foco no controle da sequência de eventos.

Além disso, a semântica denotacional tem como característica fundamental a composicionalidade. Isso significa que a semântica de operadores como escolha, paralelismo, ocultação ou recursão pode ser definida em termos das semânticas de seus operandos, permitindo construir a semântica de sistemas complexos incrementalmente. Essa propriedade é essencial tanto para a análise manual quanto para a automação da verificação, sendo explorada por ferramentas como FDR, que analisam a estrutura dos processos e realizam verificações de refinamento de maneira eficiente.

Em síntese, a semântica denotacional de CSP fornece uma base matemática sólida para a especificação e análise de sistemas concorrentes. O conceito de refinamento, central nesse paradigma, permite comparar precisamente implementações e especificações, assegurando que os comportamentos da implementação estejam contidos nos comportamentos esperados. Ao considerar diferentes níveis de abstração comportamental por meio dos modelos de *traces*, *failures-divergences*, essa abordagem oferece uma estrutura robusta para a verificação formal de propriedades fundamentais de sistemas reativos, concorrentes e críticos.

2.1.2 Semântica Operacional

A semântica operacional de CSP fornece um mecanismo formal para descrever o comportamento dinâmico dos processos por meio da evolução dos seus estados ao longo do tempo, modelando explicitamente as transições possíveis entre estados em resposta à ocorrência de eventos. Essa semântica baseia-se nos *Labelled Transition Systems* (LTS), sendo estruturas matemáticas fundamentais para a representação precisa de sistemas concorrentes e distribuídos. Formalmente, um LTS é definido como uma quádrupla $(S, L, \rightarrow, s_0)$, em que S representa

um conjunto finito ou enumerável de estados, L corresponde ao conjunto de rótulos (eventos) que descrevem as ações do sistema, $\rightarrow \subseteq S \times L \times S$ é a relação de transição que define o comportamento do sistema e s_0 é o estado inicial a partir do qual a execução inicia.

Os rótulos de transição pertencentes ao conjunto L incluem eventos visíveis, sendo aqueles definidos na especificação (elementos do alfabeto Σ), e eventos especiais, como a ação interna τ e a terminação bem-sucedida $\checkmark$ (tick). A ação τ representa um passo de execução interno ao sistema, que ocorre sem cooperação do ambiente e, portanto, não é observável externamente. Por outro lado, o evento $\checkmark$ indica a conclusão de um processo, sinalizando que ele atingiu um estado final com sucesso. A distinção entre ações visíveis e invisíveis é crucial para a modelagem de sistemas reais, ao permitir capturar tanto o comportamento externo quanto as decisões internas e automáticas do sistema.

Do ponto de vista operacional, a semântica de CSP é expressa por regras de inferência que definem como as transições entre estados ocorrem com base na estrutura sintática dos processos. Cada operador da linguagem (como prefixo, escolha externa, paralelismo, ocultação, recursão, entre outros) possui um conjunto específico de regras que determinam os eventos iniciais possíveis e os estados resultantes após sua ocorrência. Essas regras, apresentadas no estilo de sistemas dedutivos, formam um mecanismo sistemático para derivar, passo a passo, todas as possíveis execuções de um processo, construindo assim seu espaço de estados de maneira estruturada. Por exemplo, para o processo $a \rightarrow P$, há uma única transição rotulada por a levando ao estado P, enquanto para a composição sequencial $P; Q$, o processo Q só se torna ativo após a terminação de P.

O evento $\checkmark$ possui um papel especial dentro dos LTSs. Ele é tratado como uma ação visível que indica a terminação do processo, mas, diferentemente das demais ações externas, não requer cooperação do ambiente. Isso significa que, uma vez habilitado, o evento de terminação não pode ser impedido externamente, sendo inevitável. Essa interpretação intermediária — entre um evento visível comum e uma ação interna — é necessária para preservar leis semânticas desejáveis, como a identidade à direita da composição sequencial, expressa por $P; \mathit{SKIP} = P$, que só é válida quando o comportamento de $\checkmark$ é tratado com esse cuidado.

Ao modelar sistemas robóticos, LTSs permitem descrever de maneira precisa o fluxo de controle entre sensores, atuadores e componentes de decisão, incorporando tanto as interações observáveis com o ambiente quanto os comportamentos internos. Essa capacidade de representar estados e transições com granularidade adequada é essencial para garantir segurança e confiabilidade em sistemas críticos, onde a análise formal dos comportamentos possíveis é

indispensável. Por exemplo, a ausência de deadlocks, a capacidade de terminação e o respeito à ordem das operações são propriedades verificáveis diretamente sobre o LTS gerado. Essa representação por LTSs não apenas favorece o entendimento do comportamento do sistema, como também serve de base para a aplicação de técnicas automatizadas de verificação. É nesse contexto que ferramentas como a FDR se destacam, permitindo analisar formalmente propriedades críticas a partir dos modelos especificados.

Por fim, a semântica operacional e a semântica denotacional são profundamente interligadas. Embora a primeira modele explicitamente a dinâmica de execução, e a segunda se concentre em abstrações comportamentais, ambas podem ser utilizadas de maneira complementar. A partir de um LTS derivado de um processo, é possível extrair seu conjunto de *traces*, *failures* e *divergences*, aproximando a análise operacional da denotacional. Essa correspondência é particularmente importante na verificação formal, ao permitir aplicar métodos rigorosos de análise a partir da estrutura sintática dos processos, assegurando que suas propriedades estejam conforme os requisitos estabelecidos pela especificação formal.

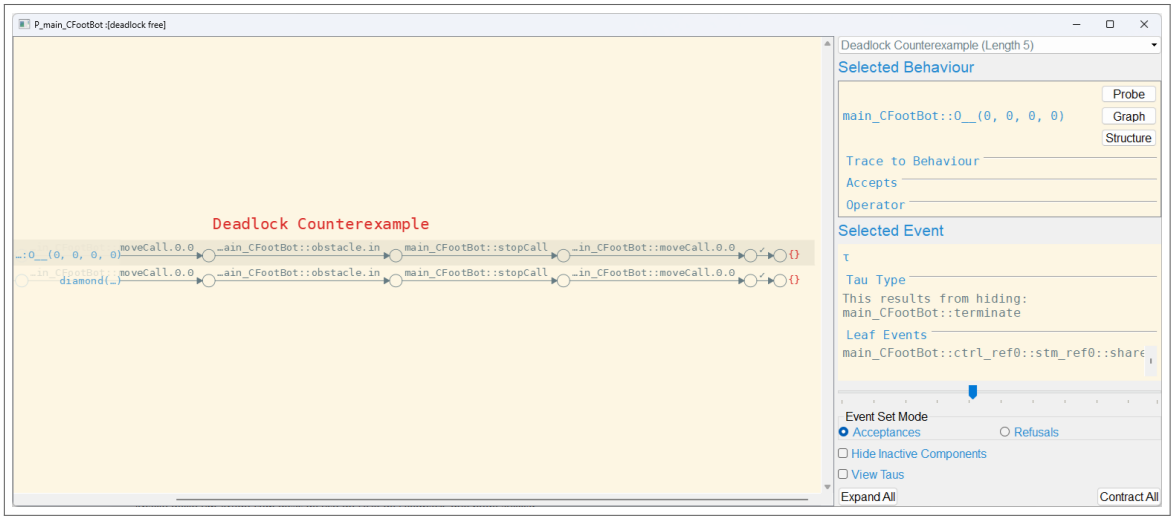
2.1.3 FDR

Failures-Divergence Refinement (FDR) (ROSCOE, 1998; ROSCOE, 2010) é uma ferramenta de verificação formal projetada para analisar modelos especificados em CSP. Desenvolvida originalmente pela Formal Systems (Europa) Ltd., FDR permite verificar automaticamente propriedades críticas, como ausência de deadlocks, segurança e refinamento de processos, sendo amplamente utilizado em sistemas críticos, como automação industrial, sistemas distribuídos e controle robótico, onde a confiabilidade é indispensável. Sua aplicação garante que erros de implementação possam ser detectados e corrigidos antes de comprometerem a funcionalidade do sistema.

Um dos recursos fundamentais de FDR é sua capacidade de transformar especificações CSP em LTS, representando graficamente estados e transições de um processo. Esse mapeamento facilita a análise de comportamentos, a detecção de inconsistências e a comparação direta entre especificação e implementação. Utilizando CSP_M, uma extensão mecanizável de CSP, FDR viabiliza análises automatizadas que garantem a conformidade das implementações com requisitos especificados, permitindo a validação de propriedades de segurança e correção em sistemas críticos. Além disso, essa abordagem robusta minimiza erros operacionais e aumenta a confiabilidade do sistema (ROSCOE, 1998; BAIER; KATOEN, 2008; GIBSON-ROBINSON et al.,

2014).

Figura 7 – Resultado da verificação de deadlock no FDR com exibição de contraexemplo.



Fonte: Elaborada pelo autor (2025)

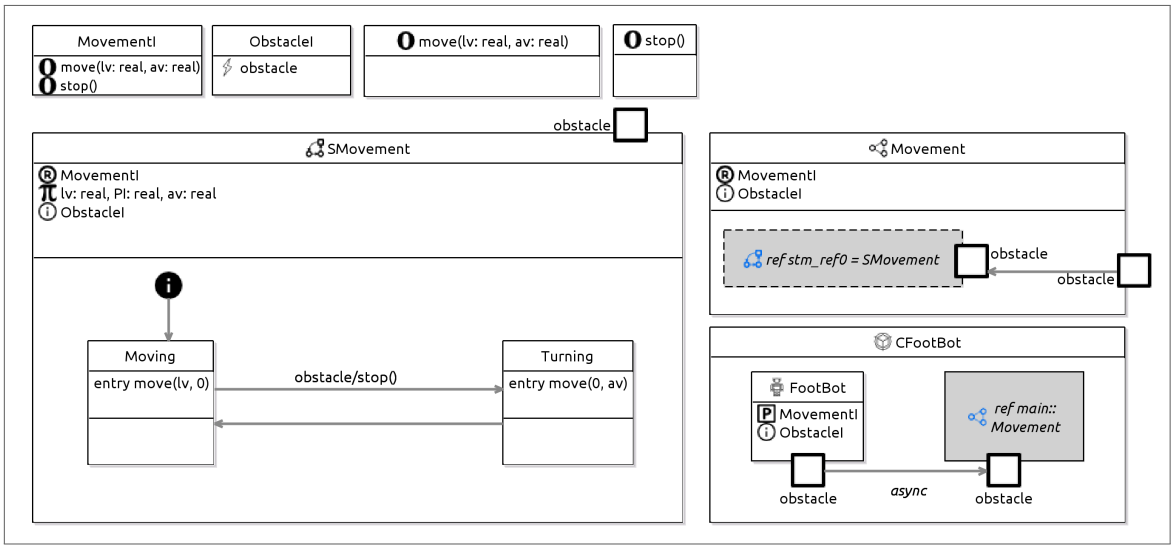
Na prática, FDR fornece uma interface gráfica que permite ao usuário carregar arquivos *.csp*, configurar as propriedades a serem verificadas e visualizar os resultados de maneira interativa. Quando uma propriedade como ausência de deadlocks não é satisfeita, FDR gera automaticamente um contraexemplo, exibido graficamente, que ilustra o trace exato de execução onde o erro ocorre. Esse trace inclui os eventos executados até o ponto de falha, auxiliando o engenheiro a localizar com precisão a origem do comportamento incorreto. A Figura 7 mostra uma tela do FDR evidenciando uma violação de deadlock no modelo de um robô detector de obstáculos. Nesse caso, após a ocorrência de determinados eventos relacionados à detecção de obstáculos, o processo atinge um estado no qual nenhum evento adicional pode ser executado, impossibilitando qualquer progresso adicional do sistema. Em outras palavras, o robô permanece bloqueado sem opções de transição, caracterizando um deadlock.

2.2 ROBOCHART

RoboChart é uma linguagem diagramática de domínio específico para projetar o comportamento de software de controle para sistemas robóticos. É um perfil UML com semântica formal definida em CSP que pode ser calculada automaticamente usando sua ferramenta associada, RoboTool. Ela foi projetada para capturar comportamentos de sistemas robóticos, interações e restrições rigorosas, incluindo aspectos temporais e probabilísticos, fundamentais para sistemas críticos. RoboChart permite modelar sistemas complexos, integrando controla-

dores, promovendo segurança e confiabilidade em aplicações como robôs industriais, sistemas autônomos e plataformas de pesquisa (MIYAZAWA et al., 2016; MIYAZAWA et al., 2019).

Figura 8 – CFootBot: modelo completo.



Fonte: Elaborada pelo autor (2025)

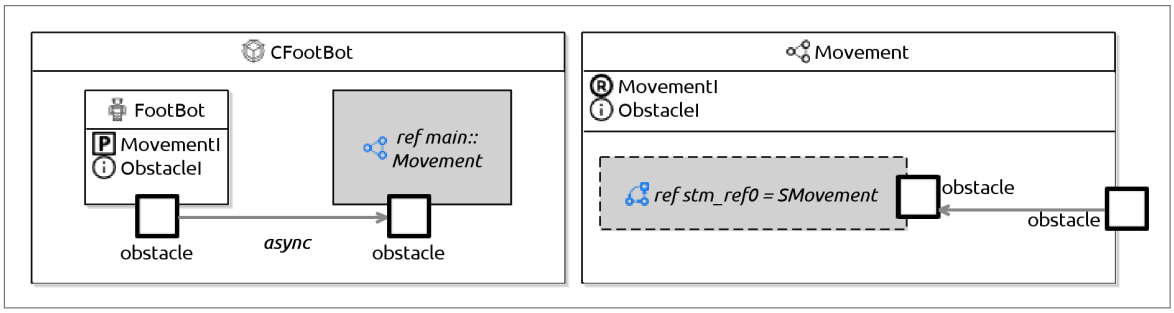
Como já mencionado, a característica central de RoboChart é que sua semântica formal é definida em CSP e, portanto, herda toda a expressividade e potencial de verificação disponível para CSP por meio da ferramenta de verificação de refinamento FDR. Essa integração possibilita verificar propriedades críticas, como ausência de deadlocks e alcançabilidade de estado, garantindo que todas as transições e comportamentos sejam analisados com segurança e precisão (MIYAZAWA et al., 2016; MIYAZAWA et al., 2019; ROSCOE, 1998).

O principal elemento de um modelo RoboChart é o Módulo, uma estrutura que registra as premissas feitas sobre o hardware do robô, descreve seu software de controle e estabelece a ligação entre ambos. Ele pode conter ou referenciar uma Plataforma e um ou mais Controladores, além de definir as conexões entre eles. Usaremos o modelo de um robô móvel capaz de se deslocar de forma autônoma, detectando e reagindo a obstáculos no ambiente para exemplificar a utilização de RoboChart na modelagem. O modelo *CFootBot* representa esse robô móvel e é mostrado na Figura 8.

O modelo *CFootBot* é descrito em RoboChart como um módulo de mesmo nome que encapsula uma plataforma e um controlador, a Figura 9 representa esse módulo. O módulo atua como um contêiner que organiza esses componentes e define como eles interagem. A plataforma denominada *FootBot* representa as capacidades físicas do robô, incluindo sensores e atuadores, e expõe um conjunto de operações, e um conjuntos de eventos, utilizados

para sinalizar mudanças no ambiente e acionar comportamentos específicos. O controlador *Movement*, por sua vez, gerencia a lógica do robô e sua interação com a plataforma robótica.

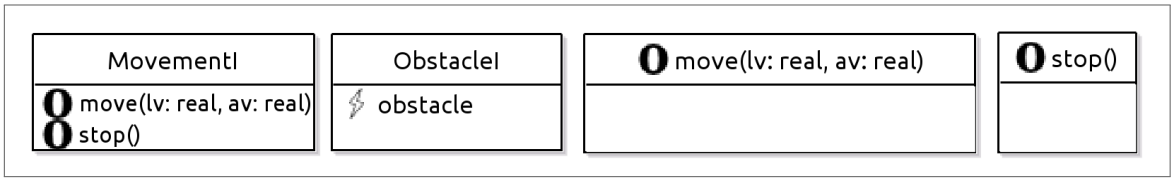
Figura 9 – CFootBot: módulo e controlador.



Fonte: Elaborada pelo autor (2025)

Como dito anteriormente, plataformas representam recursos internos do hardware. Isso inclui variáveis e constantes tipadas, operações que o robô pode executar e eventos. A plataforma do *FootBot* especifica as operações *move(lv, av)*, representando o deslocamento do robô com velocidade linear *lv* e velocidade angular *av*, e a operação *stop()*, que interrompe o movimento do robô. Neste modelo, a operação *move* representa a ação do robô de iniciar um deslocamento sem especificar detalhes sobre a rota. A decisão sobre a direção é tomada pelo controlador com base nos eventos sensoriais. Assim, o controle de planejamento de trajetória é tratado de forma implícita pelo comportamento de uma máquina de estados, fazendo com que o robô ande reto se não detectar obstaculo e vire caso exista obstáculo. A Figura 10 mostra as interfaces, operações e eventos possíveis.

Figura 10 – CFootBot: interfaces, eventos e operações.



Fonte: Elaborada pelo autor (2025)

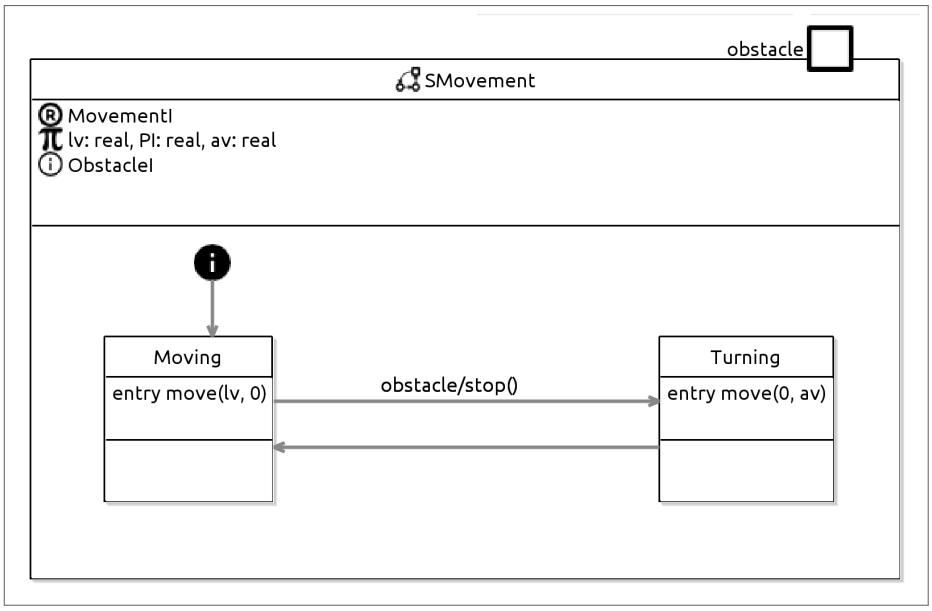
Todas as operações citadas anteriormente estão contidas em uma interface fornecida chamada *MovementI*. As interfaces encapsulam eventos, variáveis e operações. Elas podem ser de três tipos: *Fornecidas*, *Definidas* e *Necessárias*. O primeiro tipo descreve variáveis e operações que uma plataforma robótica fornece. Interfaces definidas declaram eventos e variáveis usadas em um elemento. Interfaces necessárias descrevem operações e variáveis que um controlador ou máquina de estados assume como fornecidas pela plataforma e outros controladores,

permitindo que o comportamento seja definido independentemente da plataforma específica (MIYAZAWA et al., 2016; MIYAZAWA et al., 2017).

Eventos representam uma comunicação atômica. No caso do *CFootBot*, a plataforma define o evento *obstacle*, através da interface *ObstacleI*, que indica a detecção de um obstáculo. Eventos existem tanto no nível da plataforma quanto no nível do Controlador e da Máquina de Estados. Por exemplo, o controlador do *CFootBot* tem uma interface necessária de operações que ele espera da plataforma, conectando-se à interface fornecida pela mesma. O controlador também define eventos internos como *obstacle*, que indica quando um obstáculo é detectado.

Tendo definido a plataforma e o controlador e os vinculado no módulo, definimos o comportamento dentro do controlador. Isso é feito com uma máquina de estados, *SMovement*. Ela tem a mesma interface necessária que o controlador, variáveis internas para controle de estado e eventos como *obstacle*, especificados em uma interface definida chamada *MovementI* e vinculados aos eventos do controlador conforme o esperado.

Figura 11 – CFootBot: máquina de estados.



Fonte: Elaborada pelo autor (2025)

Toda máquina de estados é composta de estados, junções e transições. Os estados podem ter ações a serem executadas na entrada do estado, durante o estágio ativo do estado ou na saída. As junções podem não ter essas operações e devem ter pelo menos uma transição de saída. As junções agem como estados temporários instáveis pelos quais o robô deve passar e sair imediatamente. Cada Máquina de Estados deve ter uma junção inicial. A máquina *SMovement* imediatamente transita para um estado chamado Moving, onde o robô inicia seu

deslocamento até que um obstáculo seja detectado. A Figura 11 apresenta uma representação esquemática dessa estrutura, ilustrando a relação entre estados, junções e transições dentro da máquina de estados.

Uma vez que o evento *obstacle* é recebido, o robô interrompe seu movimento, realiza a operação *stop()* e transita para o estado Turning. Na entrada do estado Turning, uma ação de reorientação é executada através da operação *move(0, av)*, iniciando uma rotação. O robô realiza uma operação de rotação e após isso, a transição para Moving é acionada, retomando o deslocamento normal.

Para resumir, o *CFootBot* usa tipos de dados definidos na plataforma, e suas operações e eventos externos estão encapsulados na interface fornecida. Seu comportamento é especificado por uma máquina de estados, que é um componente do controlador. O nível superior da especificação é o módulo, que conecta a plataforma e o controlador. Além disso, a semântica de RoboChart, que será abordada a seguir, é fundamental para entender como os modelos de sistemas dinâmicos, como o *CFootBot*, podem ser formalizados e analisados, oferecendo uma base sólida para a implementação e validação de comportamentos complexos em sistemas robóticos.

2.2.1 Semântica de RoboChart

Como dito anteriormente, a semântica formal de RoboChart é baseada em CSP, permitindo que modelos especificados na notação diagramática sejam traduzidos para uma forma textual adequada para verificação formal. Essa abordagem possibilita o uso de ferramentas como FDR para análise de propriedades como refinamento, deadlock e determinismo. CSP oferece uma base sólida para a modelagem do comportamento concorrente e reativo dos sistemas robóticos, garantindo que as interações entre componentes sejam expressas de forma rigorosa e verificável.

Uma das particularidades de RoboChart é a definição de uma semântica própria utilizando CSP, incorporando eventos específicos que representam interações entre componentes do sistema. Em particular, eventos como *entered*, *in*, *out*, *exit*, *during* e *terminate* são utilizados para capturar mudanças de estado e comunicações entre processos. O evento *entered* é acionado sempre que um estado é alcançado, permitindo registrar explicitamente a entrada em um novo estado dentro da máquina de estados de RoboChart. O evento *exit* representa a saída de um estado, garantindo que a transição seja capturada formalmente. Já os eventos *in* e *out*

são usados para comunicação síncrona entre processos, onde *in* representa a recepção de um evento e *out* a sua emissão. O evento *during* é utilizado para modelar ações contínuas em um estado, representando ciclos de execução que ocorrem enquanto o estado está ativo. O evento *.terminate* é fundamental para indicar a finalização de um processo ou sistema, garantindo que uma terminação explícita seja modelada e tratada corretamente (MIYAZAWA et al., 2016; MIYAZAWA et al., 2017).

A semântica de mapeamento de RoboChart para CSP traduz cada estado em um processo cujo ciclo de vida é gerenciado por eventos específicos. A entrada em um estado é sinalizada pelo evento *.entered*, e quaisquer ações de entrada associadas são modeladas como eventos que ocorrem imediatamente em sequência. Ações contínuas, executadas enquanto o estado está ativo, são representadas pelo evento *.during*. A saída, por sua vez, é modelada pelo evento *.exit*, que também encapsula as ações de finalização do estado. As transições entre estados utilizam eventos *.in* e *.out* para sincronização, enquanto variáveis e operações são abstraídas como canais de comunicação. Finalmente, o encerramento explícito de um processo é garantido pelo evento *.terminate*, permitindo modelar a finalização completa de um componente.

Além disso, a semântica de RoboChart em CSP inclui o uso de eventos de tempo para modelar restrições temporais. Em RoboChart, o tempo pode ser expresso por clocks, que impõem restrições sobre quando eventos podem ocorrer. No CSP, essas restrições temporais são representadas utilizando operadores como *wait* e *tock*, permitindo a especificação e verificação de propriedades temporais de modelos robóticos. Embora haja a semântica temporizada, este trabalho usará somente a semântica sem considerar tempo.

Outro aspecto importante é como RoboChart lida com a composição de processos. Em CSP, a composição paralela é usada para modelar a execução concorrente de diferentes componentes do sistema. No contexto de RoboChart, essa composição permite descrever interações entre múltiplos controladores e entre controladores e a plataforma. A sincronização entre esses processos é feita por meio de eventos compartilhados, garantindo que a execução ocorra de maneira coordenada.

Com a formalização em CSP, é possível realizar verificações rigorosas de modelos RoboChart. A análise de refinamento permite comparar a implementação de um sistema com sua especificação formal, garantindo que todos os comportamentos permitidos pela implementação sejam compatíveis com a especificação. Além disso, a verificação de deadlocks, não determinismos e divergências assegura que o sistema não entre em estados indesejados ou apresente comportamentos não determinísticos.

Portanto, a semântica CSP de RoboChart oferece uma base formal sólida para a modelagem e verificação de sistemas robóticos. A inclusão de eventos como *.entered*, *.exit*, *.during*, *.in*, *.out* e *.terminate* permite um controle preciso das interações entre os componentes, enquanto os mecanismos de sincronização e composição asseguram a consistência do comportamento concorrente. Assim, RoboChart se posiciona como uma ferramenta eficiente para o desenvolvimento de sistemas robóticos, garantindo confiabilidade e verificabilidade formal.

2.3 PYTRANSITIONS

A biblioteca *pytransitions* é uma implementação orientada a objetos de FSM (*Finite State Machine*), ou Máquina de Estados Finitos, em Python, amplamente reconhecida por sua flexibilidade e simplicidade na modelagem de sistemas baseados em estados. Criada com o propósito de facilitar o gerenciamento de estados e transições em sistemas complexos, pytransitions é frequentemente empregada em áreas como sistemas robóticos e simulação de comportamentos em software. Esta seção detalha as funcionalidades e componentes principais da biblioteca, bem como sua aplicação em sistemas críticos.

A pytransitions fornece um conjunto abrangente de ferramentas para modelar FSMs, com suporte para estados, transições, eventos e triggers. Além disso, ela permite a personalização de máquinas por meio de ações (callbacks) associadas a transições e estados, viabilizando a execução de comportamentos específicos em diferentes etapas do ciclo de vida do sistema. Uma de suas características marcantes é a capacidade de suportar máquinas de estados hierárquicas e paralelas, tornando a biblioteca adequada para sistemas altamente complexos.

Os componentes centrais do pytransitions incluem:

- Estados: representam as condições ou modos de operação de um sistema. Os estados podem ser definidos de maneira simples (por strings) ou como objetos mais complexos, permitindo maior flexibilidade na modelagem.

- Transições: definem como o sistema se move de um estado para outro. Cada transição é ativada por um evento (trigger) e pode incluir condições específicas que precisam ser satisfeitas para que a mudança de estado ocorra.

- Triggers (Eventos): são os estímulos que iniciam as transições. Podem ser definidos como métodos que simulam o recebimento de comandos externos ou internos.

- Máquina: representa a FSM toda, gerenciando os estados, transições, eventos e o estado atual do sistema.
- Modelo: refere-se ao objeto que mantém o estado atual e pode ser enriquecido com dados adicionais para representar as propriedades de um sistema.

Os estados são os blocos fundamentais de qualquer máquina de estados modelada com pytransitions. Cada estado representa uma condição específica ou um modo de operação do sistema. Por exemplo, em um robô industrial, os estados podem incluir "Idle", "Movendo para a Posição", "Inspecionando Componente" e "Retornando à Base". A biblioteca permite definir estados de maneira simples, utilizando strings, ou como objetos mais complexos, possibilitando a adição de propriedades específicas e métodos associados a cada estado.

Além disso, pytransitions suporta estados aninhados, conhecidos como estados hierárquicos ou compostos, onde um estado pode conter subestados. Isso é útil para modelar sistemas com comportamentos relacionados, como um robô que, no estado "Operacional", pode alternar entre subestados como "Navegando" e "Carregando Objeto". Essa organização hierárquica melhora a clareza do modelo, reduz a redundância e facilita a gestão de sistemas com múltiplos níveis de complexidade.

As transições são os elementos que conectam os estados e definem como o sistema muda de uma condição para outra. No pytransitions, cada transição é ativada por um evento (trigger) e pode incluir condições que devem ser satisfeitas para a mudança ocorrer. Por exemplo, uma transição do estado "Movendo para a Posição" para "Inspecionando Componente" pode ser ativada pelo evento *reached_position* e condicionada à verificação de que a posição alvo foi alcançada com precisão.

Além disso, as transições podem ter ações associadas por meio de callbacks, permitindo que tarefas sejam executadas durante a mudança de estado. Por exemplo, ao transitar para o estado "Retornando à Base", a transição pode incluir um comando para desativar os atuadores ou salvar dados coletados no sistema. Essa funcionalidade garante que o modelo não somente descreva os estados e mudanças, mas também integre logicamente as ações associadas a cada etapa.

Existem três tipos principais de callbacks: ao entrar em um estado (*on_enter*), executados assim que o sistema transita para um estado específico; ao sair de um estado (*on_exit*), ativados antes que o sistema deixe um estado; e callbacks associados a transições (*before/after*), que permitem verificar condições ou executar ações específicas antes, ou depois de uma mu-

dança de estado. Esses recursos garantem maior flexibilidade e adaptabilidade, especialmente em sistemas complexos.

A biblioteca *pytransitions* também se destaca pela possibilidade de trabalhar de maneira orientada a eventos, permitindo que as transições entre estados sejam ativadas por estímulos externos ou internos, conhecidos como triggers. Essa abordagem é particularmente útil em sistemas reativos, onde as mudanças de estado dependem de eventos dinâmicos, como a detecção de um sensor ou a recepção de um comando. Essa flexibilidade faz da biblioteca uma ferramenta boa para modelar sistemas em tempo real, como aplicações robóticas e sistemas de automação, onde a responsividade a eventos é crucial para garantir um comportamento adequado.

Com suporte para modelagem orientada a eventos, controle detalhado do ciclo de vida, e recursos avançados para gerenciar estados e transições, pytransitions é uma biblioteca poderosa para sistemas baseados em estados. Sua flexibilidade permite modelar sistemas simples e altamente complexos, atendendo às necessidades de diversas aplicações, desde robótica até automação industrial.

A seguir, o Código Fonte 1 é um exemplo básico de uma FSM para o robô *CFootBot*, conforme o modelo especificado na Seção 2.2. O robô alterna entre dois estados principais: *Moving* (movendo-se para frente) e *Turning* (girando para desviar de um obstáculo). As transições entre os estados são ativadas por eventos representando a detecção de obstáculos e a conclusão da manobra de desvio. Este exemplo ilustra de forma simplificada como representar o comportamento reativo do robô utilizando a biblioteca *pytransitions*.

Código Fonte 1 – Implementação do CFootBot usando pytransitions.

```
from transitions import Machine

class CFootBot:
    def move_forward(self):
        print("Movendo para frente.")

    def turn(self):
        print("Girando para desviar.")

robot = CFootBot()

states = ['Initial', 'Moving', 'Turning']
transitions = [
    {'trigger': 'start', 'source': 'Initial', 'dest': 'Moving',
     'after': 'move_forward'},
```

```
    {'trigger': 'obstacle_detected', 'source': 'Moving', 'dest': 'Turning',
17   'after': 'turn'},
    {'trigger': 'turn_completed', 'source': 'Turning', 'dest': 'Moving',
19   'after': 'move_forward'}
]
21

machine = Machine(model=robot, states=states, transitions=transitions, initial='
    Initial')
23

robot.start()
25 robot.obstacle_detected()
robot.turn_completed()
```

Fonte: Elaborada pelo autor (2025)

Durante a execução do código, o robô *CFootBot* segue um ciclo de vida bem definido, estruturado pela máquina de estados. Inicialmente, o sistema encontra-se no estado *Initial*, representando o ponto de partida do controle comportamental. A primeira transição ocorre com a invocação do evento `start()`, que leva o robô ao estado *Moving*. Como parte dessa transição, é executada a ação `mov_forward()`, que simboliza o início do deslocamento em linha reta. Enquanto o robô permanece neste estado, presume-se que ele esteja operando normalmente até que um evento externo, como a detecção de um obstáculo, dispare a transição para o estado *Turning* por meio do evento `obstacle_detected()`. Ao entrar neste novo estado, a função `turn()` é acionada, simulando a execução de uma manobra de desvio. Após o término dessa ação, o evento `turn_completed()` promove o retorno ao estado *Moving*, reiniciando o ciclo. Esse comportamento cíclico demonstra a dinâmica de um sistema reativo, no qual os estados e as transições definem claramente as possíveis trajetórias de execução, enquanto os eventos e métodos associados controlam o fluxo de ações do robô ao longo de sua operação.

3 METODOLOGIA

Este capítulo apresenta a metodologia usada para verificar a conformidade de uma implementação de sistema de controle robótico em Python, usando a biblioteca pytransitions, com seu modelo de design em RoboChart (e sua especificação formal). Para a apresentação da metodologia, considere um exemplo de uma linha de montagem que utiliza um sistema robótico para executar operações como inspecionar e processar componentes, descartar itens defeituosos e montar produtos. Dadas as preocupações com escalabilidade, optamos por reduzir o escopo de nossa análise focando somente em máquinas de estado e avaliando-as separadamente. Essa abordagem permitiu um processo de verificação mais estruturado e computacionalmente viável. Neste cenário, espera-se que o sistema robótico passe por vários estados, como inspecionar componentes, identificar defeitos e executar tarefas de montagem com base na condição de cada item. O sistema deve conseguir lidar com diferentes cenários, como encontrar componentes defeituosos, o que levaria ao descarte desses itens e à continuação do processo.

A metodologia de análise consiste em cinco etapas principais: formalização de requisitos informais usando RoboChart, obtenção do LTS da semântica formal de RoboChart em CSP, abstração do LTS, verificação de conformidade entre o LTS e o FSM da implementação por meio do Algoritmo 1 de verificação de refinamento de traces e uma etapa final de análise de resultados.

3.1 FORMALIZAÇÃO DOS REQUISITOS

A formalização de requisitos é um passo essencial no desenvolvimento de sistemas robóticos, especialmente em contextos reativos, críticos e concorrentes. Em aplicações como linhas de montagem industriais, dispensação automatizada de medicamentos ou robôs móveis autônomos, qualquer ambiguidade nos requisitos pode resultar em falhas operacionais graves. Especificações informais, geralmente escritas em linguagem natural, tendem a ser incompletas ou ambíguas, falhando em representar detalhes como condições de sincronização, tratamento de exceções e comportamentos emergentes. Nesse cenário, o uso de modelos formais permite explicitar decisões, estruturar comportamentos condicionais e estabelecer limites operacionais com precisão matemática, promovendo confiabilidade, segurança e rastreabilidade.

Com base nessa motivação, o primeiro passo metodológico deste trabalho consiste na utili-

zação de RoboChart para traduzir requisitos informais — geralmente expressos em linguagem natural — em modelos formais. Esses requisitos descrevem as operações esperadas do sistema e podem ser classificados em duas categorias principais: funcionais e não funcionais. Os requisitos funcionais especificam as ações que o sistema deve realizar — por exemplo, "pegar componente" ou "embalar produto" — enquanto os não funcionais tratam de restrições relacionadas a desempenho, segurança, tempo de resposta, entre outros aspectos. Embora este trabalho concentre-se na formalização dos requisitos funcionais por meio de RoboChart, é importante destacar que o formalismo também permite capturar requisitos não funcionais relacionados a aspectos probabilísticos e temporais, os quais estão fora do escopo do presente trabalho. Estruturas como pré-condições, guardas e eventos podem representar, por exemplo, restrições como "só inspecionar o componente se ele estiver posicionado corretamente", oferecendo uma base formal para aspectos críticos de operação. Como exemplo, consideramos o seguinte requisito informal para uma linha de montagem automatizada:

> "O braço robótico, auxiliar da linha de montagem, deve pegar um componente e inspecioná-lo. Caso o componente não seja defeituoso, o braço leva o componente para a montagem, verifica a montagem, embala o produto e finaliza a operação".

Para tornar mais clara a correspondência entre os elementos do requisito informal e os componentes formais da máquina de estados, é útil detalhar como cada ação e decisão foi mapeada no modelo. A sistemática adotada para a tradução dos requisitos segue um padrão claro: o verbo principal da ação no requisito origina o nome do evento (trigger) da transição, enquanto o estado resultante é nomeado a partir do particípio do verbo, invertendo a ordem com o objeto da ação. Por exemplo, a ação "pegar um componente" gera o evento *pick_component* para a transição, e o estado associado passa a ser *ComponentPicked*, representando que o componente foi capturado com sucesso. De forma análoga, "inspecionar componente" dá origem à transição *inspect_component* e ao estado *ComponentInspected*.

O braço robótico, elemento central do requisito, dá origem ao módulo responsável pelas operações principais. A decisão condicional "caso o componente não seja defeituoso" é traduzida para uma transição com guarda lógica, permitindo o desvio para diferentes caminhos: se o componente estiver em boas condições, segue para a montagem por meio da transição *place_component [defective_component* $==$ *false]*, caso contrário, o fluxo segue para o descarte. Essas transições condicionais incorporam diretamente a lógica de decisão do domínio, mantendo a rastreabilidade entre a linguagem natural e os artefatos formais. A Figura 12

torna visível essa estruturação, evidenciando como estados, eventos e guardas representam os comportamentos esperados e promovem uma modelagem rigorosa e interpretável do sistema.

Para garantir a fidelidade entre o comportamento modelado e as expectativas do domínio, algumas heurísticas de modelagem foram adotadas. A primeira delas foi o uso de estados intermediários para representar ações compostas, o que permite rastrear cada etapa do processo com precisão. Também se optou por manter os nomes de eventos e transições próximos à linguagem original dos requisitos, facilitando a validação com especialistas do domínio. Adicionalmente, priorizou-se uma granularidade que favorecesse modularidade, evitando que estados acumulassem múltiplas funções, o que dificultaria a verificação formal posterior.

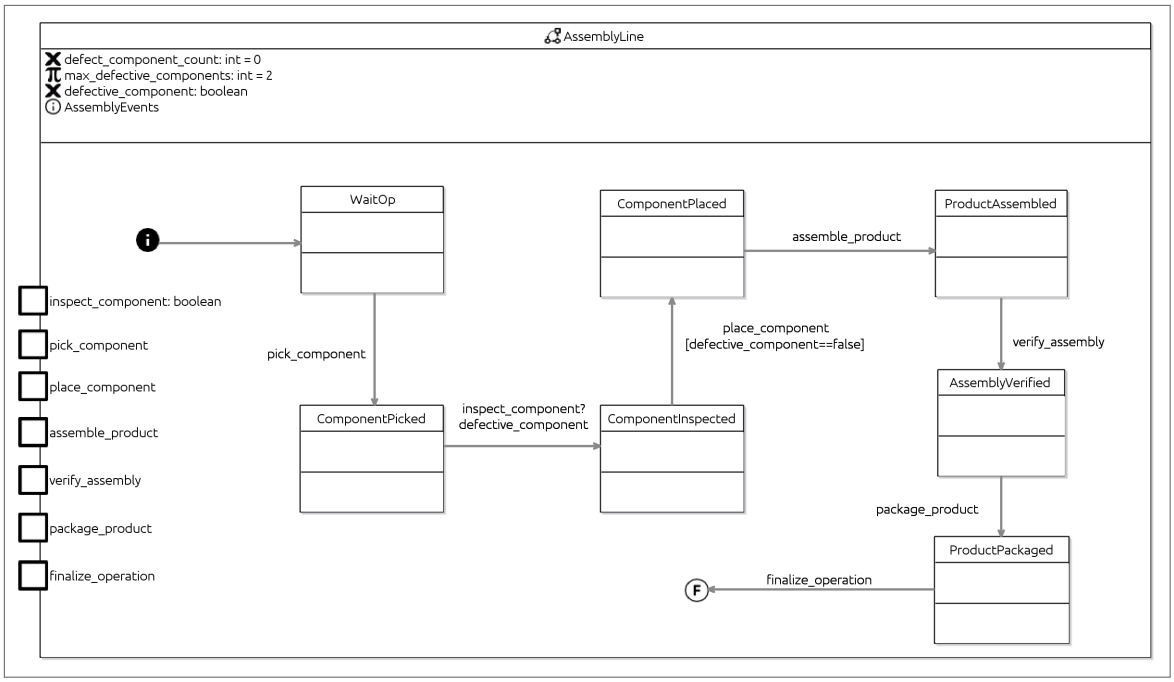
Conforme destacado em (GIESE; HELDAL, 2004; CARVALHO et al., 2015; SANTOS; CARVALHO; SAMPAIO, 2018), traduzir requisitos informais em modelos formais é uma etapa crucial no desenvolvimento de sistemas reativos, especialmente em domínios onde a previsibilidade e a segurança são fundamentais. Essa transição tem o papel de eliminar ambiguidade, assegurar precisão semântica e possibilitar a aplicação de técnicas de verificação automática. Requisitos escritos em linguagem natural tendem a ser interpretados subjetivamente por diferentes membros da equipe de desenvolvimento, o que pode gerar inconsistências entre o comportamento implementado e o desejado. A formalização, por sua vez, torna explícitos os comportamentos esperados, os fluxos condicionais e os cenários de exceção, favorecendo a rastreabilidade entre as etapas do ciclo de desenvolvimento.

Embora ferramentas automatizadas, como a *NAT2TEST* (CARVALHO et al., 2015), ofereçam suporte parcial para converter sentenças em linguagem natural para modelos formais de teste, ainda existem limitações consideráveis quanto à expressividade, controle semântico e generalização para domínios complexos como a robótica. Por essa razão, neste trabalho, a modelagem da máquina de estados em RoboChart foi realizada manualmente, para representar com maior precisão os aspectos operacionais, restrições contextuais e particularidades do domínio. Apesar de permitir um mapeamento mais cuidadoso entre requisitos e modelo, essa tradução manual não assegura, por si só, que o comportamento especificado esteja completamente correto ou livre de ambiguidades. Por isso, a verificação de propriedades sobre a especificação formal é uma etapa essencial para validar se o modelo realmente satisfaz os requisitos pretendidos. No entanto, uma direção futura promissora seria a automação desse processo, integrando técnicas de aprendizado de máquina e linguagens naturais controladas, para agilizar e sistematizar a transição de requisitos informais para modelos formais verificáveis.

Embora nenhuma ferramenta automatizada tenha sido utilizada para gerar o modelo Ro-

boChart a partir dos requisitos informais, essa tradução foi conduzida de forma sistemática, por meio de um mapeamento estruturado, conforme demonstrado no exemplo fornecido. No texto do requisito, verbos e ações-chave como “pegar”, “inspecionar”, “levar”, “verificar” e “embalar” foram traduzidos cuidadosamente em transições na máquina de estados RoboChart, representando mudanças entre diferentes etapas do processo. Os estados, por sua vez, foram definidos para capturar condições ou configurações específicas do sistema, como a posição do braço robótico ou o status de uma tarefa, compondo uma estrutura que reflete as fases operacionais descritas no requisito. Para apoiar a validação do comportamento modelado, foram realizadas animações baseadas no código CSP gerado automaticamente por RoboTool, permitindo observar a dinâmica de execução do sistema e verificar se ela condizia com os cenários esperados. Embora essa abordagem não substitua a prova formal de propriedades — sendo o método ideal para garantir a correção do modelo —, ela contribui como uma etapa complementar de validação após a formalização, reforçando a confiança na adequação do comportamento especificado.

Figura 12 – Assembly line parcial em RoboChart.



Fonte: Elaborada pelo autor (2025)

O uso do componente *máquinas de estados* (conforme descrito na Seção 2.2) em RoboChart se alinha naturalmente com a implementação, que também é baseada em conceitos de máquina de estados. Essa consistência facilita o desenvolvimento da metodologia e reduz a complexidade e os desafios de escalabilidade da análise. Ao definir explicitamente estados e

transições, o modelo se torna mais modular e fácil de entender, permitindo a decomposição sistemática do comportamento do sistema. Além disso, essa abordagem aprimora a manutenibilidade e a verificação, pois cada estado encapsula funcionalidades específicas, permitindo uma validação mais direta de requisitos e condições. Como já mencionado, a máquina de estados resultante do requisito informal antes apresentado é ilustrada na Figura 12.

Por exemplo, na Figura 12, o estado *WaitOp* representa o braço robótico esperando pelo próximo comando. A transição *pick_component* modela a ação de mover o braço para a posição desejada e pegar o componente, enquanto o estado *ComponentPicked* resultante representa o status atualizado do sistema. Da mesma forma, a transição para o estado *ComponentInspected* é acionada pelo evento *inspect_component*, indicando que a inspeção do componente foi concluída. Nesse processo, as transições também podem envolver troca de dados e avaliações condicionais. Por exemplo, a transição *inspect_component?defective_component* não somente sinaliza o evento de inspeção, mas também recebe informações sobre se o componente está com defeito. Esses dados são então usados em transições subsequentes, como *place_component [defective_component==false]*, o que garante que somente componentes não defeituosos prossigam para a montagem. Ao estruturar sistematicamente esse processo de tradução, o modelo RoboChart organiza efetivamente o comportamento do sistema e valida os requisitos, garantindo que os estados e transições reflitam com precisão os objetivos operacionais descritos na especificação informal.

Usar RoboChart fornece uma abordagem estruturada e rigorosa para modelagem, e a capacidade de gerar código CSP_M aprimora ainda mais o processo de verificação ao permitir análise formal e verificação de refinamento, garantindo que todas as possibilidades sejam rigorosamente verificadas. Essa formalização permite uma transição natural para análise automatizada, garantindo que os comportamentos esperados sejam implementados com precisão. Além disso, o uso de variáveis e tipos em RoboChart fornece flexibilidade para lidar com diferentes níveis de abstração, como contadores de peças ou estados booleanos para condições de falha, adicionando robustez ao modelo. Finalmente, destaca-se que essa formalização também prepara o sistema para etapas posteriores do ciclo de verificação e validação, como testes formais, geração automática de testes e simulação. Ao estruturar os requisitos desde o início em uma notação formal compatível com verificação automática, abrem-se possibilidades para integração com testes baseados em modelos, verificação de tempo real e até síntese de controladores. Esses desdobramentos podem ser explorados em trabalhos futuros.

Antes de avançar para a etapa de obtenção do LTS, é fundamental assegurar que o sistema

especificado em CSP esteja livre de propriedades indesejadas, como deadlocks, não determinismo e livelocks. A presença dessas propriedades compromete tanto a análise subsequente quanto a confiabilidade geral do modelo. Por exemplo, deadlocks indicam estados de bloqueio onde o sistema não pode mais avançar; o não determinismo pode causar comportamentos ambíguos e imprevisíveis; enquanto livelocks representam ciclos infinitos de execução que impedem o progresso efetivo do sistema. A verificação rigorosa dessas propriedades garante que o modelo formal gerado por RoboChart seja consistente e apropriado para as fases seguintes de análise e refinamento. Somente após a confirmação da ausência desses problemas é que se prossegue para a geração do LTS.

3.2 OBTENÇÃO DO LTS

Uma vez que a especificação CSP de um modelo RoboChart esteja disponível, o próximo passo em nossa metodologia é derivar seu LTS. Este processo é realizado usando a API de FDR4[1] (FDR4, 2016), que fornece uma interface programática para carregar, analisar e gerar o LTS correspondente ao modelo CSP. A API permite acessar os estados gerados, identificação de transições habilitadas e inspeção do comportamento dinâmico do sistema. Embora esteja disponível para Java e C++, usamos a versão Python para manter a consistência com nossa linguagem de implementação, garantindo integração perfeita em nossa metodologia e automatizando a geração de LTS e a verificação de transições válidas.

O processo de obtenção do LTS começa com a geração automatizada do código CSP a partir do modelo desenvolvido em RoboChart, preservando toda a estrutura da máquina de estados, incluindo estados, transições, condições de guarda e ações associadas. Após essa geração, a especificação CSP é carregada na API de FDR4, que executa a análise e gera o LTS. Este LTS, por sua vez, descreve todos os estados possíveis do sistema e as transições entre eles, fornecendo uma representação precisa e completa do comportamento do modelo, essencial para a validação formal e a análise do sistema. Porém, esse LTS obtido não está pronto para ser usado na nossa abordagem e demanda um certo nível de abstração.

O Código Fonte 2 mostra a função Python utilizada para carregar a especificação CSP e extrair o LTS e o primeiro nó do LTS, por meio da API de FDR4.

Código Fonte 2 – Função utilizada para obter o LTS a partir da especificação.

[1] https://cocotec.io/fdr/manual/

```python
def obtain_lts(fdr_instance, csp_path):
    fdr_instance.library_init()

    session = fdr_instance.Session()
    session.load_file(csp_path)

    lts_target_process = session.evaluate_process('target_process', fdr_instance.
        SemanticModel_Traces, None).result()
    first_node = lts_target_process.root_node()

    return lts_target_process, first_node
```

Fonte: Elaborada pelo autor (2025)

3.3 ABSTRAÇÃO DO LTS

O LTS gerado a partir do modelo CSP inclui construções semânticas específicas de RoboChart, muitas das quais não são diretamente relevantes para a metodologia proposta neste trabalho. Para focar no comportamento essencial do sistema, utilizamos o processo *VS_O___* da especificação CSP,que representa uma máquina de estados com estados visuais, ou seja, estados como eventos, fornecendo uma visão de alto nível da máquina de estados. Esse processo expõe eventos-chave — como *s.entered* (onde *s* é o nome completo do estado) e eventos de comunicação marcados com o sufixo *.in* — enquanto já exclui transições internas τ. Essas características tornam o *VS_O___* particularmente adequado para extrair comportamentos significativos.

Em nossa abordagem, os eventos são processados incrementalmente à medida que são encontrados, de forma dinâmica. Cada evento é avaliado de acordo com sua relevância: apenas entradas de estado e eventos de comunicação entre componentes são retidos, com nomes de estado simplificados e elementos estruturais desconsiderados. Embora fosse possível realizar essa renomeação e filtragem diretamente no modelo CSP, optamos por aplicar essas transformações no nível da implementação Python. Essa escolha proporcionou maior flexibilidade e integração com as etapas de análise subsequentes. O resultado dessa abstração por eventos é um LTS abstrato que reflete o comportamento observável do sistema. Um exemplo de trace desse LTS abstrato, derivado do processo *VS_O___* da máquina de estados *AssemblyLine*, é apresentado na Figura 13.

Essa estratégia garante a compatibilidade com o nível de abstração da implementação

Figura 13 – Exemplo de trace após a abstração.

$$\langle WaitOp, receive\_command.in, ComponentPicked, ..., Finish, terminate \rangle$$

Fonte: Elaborada pelo autor (2025)

Python, que naturalmente omite eventos internos e sobrecarga estrutural. Além disso, ao interpretar eventos diretamente do processo *VS_O___*, preservamos a semântica definida em RoboChart e mantemos o alinhamento com a geração automatizada fornecida por RoboTool (MIYAZAWA et al., 2016; MIYAZAWA et al., 2017). Isso permite que a metodologia permaneça sólida e consistente com o modelo formal subjacente. Em particular, essa abordagem evita transformações desnecessárias da especificação original, garantindo que o comportamento usado para validação seja derivado fielmente da mesma semântica que rege a análise formal do modelo.

3.4 ALGORITMO DE VERIFICAÇÃO DE REFINAMENTO DE TRACES

A metodologia para verificar a conformidade entre a especificação formal e a implementação é baseada no conceito de refinamento de traces, formalmente representado como $P \sqsubseteq Q$, cuja definição é dada por *traces*(Q) $\subseteq$ *traces*(P), veja a Seção 2.1.1; P representa a especificação e Q a implementação. Para executar essa verificação, um algoritmo de *Verificação de Refinamento de Traces* é proposto[2]. Esse algoritmo permite a exploração sistemática dos estados e transições definidos no LTS abstrato da especificação CSP, conforme apresentado na seção anterior, e os compara com aqueles do FSM da implementação. Importante destacar que o algoritmo desenvolvido é uma versão simplificada do método utilizado pelo FDR (ROSCOE, 2010), especialmente no que diz respeito à manipulação da supermáquina e às normalizações empregadas na verificação completa.

O algoritmo central da metodologia, apresentado no Algoritmo 1, utiliza uma abordagem recursiva para percorrer o LTS abstraído a partir de CSP, verificando, em cada estado, se os eventos e transições observáveis na FSM implementada com a biblioteca *pytransitions* estão contidos no conjunto de comportamentos permitidos pela especificação. A função principal, *explore_transitions*, itera sobre os nós (estados) do LTS abstraído da especificação, gerando

[2] Uma abordagem alternativa seria derivar uma especificação CSP da implementação e usar FDR para refinamento, mas o algoritmo descrito aqui foi escolhido.

as transições a partir de um nó atual e comparando-as com as transições disponíveis na FSM da implementação. Durante o processo, informações detalhadas sobre as transições e estados visitados são registradas para análise posterior.

Algoritmo 1: Algoritmo de verificação de refinamento de traces.

```
1  Function
   is_traces_refined_by(spec_lts, implementation_machine, current_lts_node):
2      foreach spec_transition ∈ spec_lts do
3          dest_node = spec_transition.destination() // Target state associated with
               the transition
4          event_lts = spec_transition.event()
5          impl_transitions = implementation_machine.get_transitions()
6          if {event_lts} ∩ {entered, terminate} then
7              spec_transitions = specification_lts.transitions(current_lts_node)
8              available_impl_transitions =
                 implementation_machine.get_available_transitions(impl_transitions)
9              if available_impl_transitions ⊈ spec_transitions then
10                 return false ;                          // Refinement failed
11             end
               // Executes the transition specified by the model in the
                  implementation's machine
12             new_impl_machine = implementation_machine.trigger(spec_transition)
13         end
14         if event_lts =′ terminate′ then
15             return true ;                          // End of refinement process
16         end
17         is_traces_refined_by(spec_lts, new_impl_machine, dest_node)
18     end
19 return true
```

Fonte: Elaborada pelo autor (2025)

A verificação de conformidade é realizada comparando o FSM derivado da implementação com o LTS abstrato gerado a partir do modelo formal em CSP. O algoritmo garante que a implementação execute as mesmas transições especificadas no modelo, garantindo que o comportamento observado seja consistente com o definido. Se a implementação não suportar as mesmas transições ou apresentar comportamentos divergentes, o processo é interrompido, registrando o ponto de erro e o trace correspondente até a falha. Esse registro detalhado permite a identificação precisa de inconsistências e fornece uma base para corrigir a implementação.

Uma das principais dificuldades encontradas nesse processo foi garantir que a máquina de estados implementada em *pytransitions* seguisse os mesmos caminhos e alfabeto (nomes de eventos) especificados pelo LTS do modelo. Para superar esse desafio, um dispositivo foi usado para cadenciar a execução da implementação conforme a especificação. Cada evento na especificação comunica, por meio de canais, o estado de variáveis booleanas associadas à exe-

cução dos estados de implementação. A manipulação dessas variáveis garante que os estados internos da máquina de implementação estejam alinhados com nós do LTS da especificação, permitindo que a implementação faça a transição entre estados de maneira idêntica ao modelo. Essa sincronização garante consistência entre o FSM da implementação e o modelo formal.

Antes de iniciar a exploração dos caminhos no LTS, o algoritmo realiza uma verificação preliminar para assegurar o alinhamento dos eventos entre a especificação formal e a máquina de estados da implementação. Na prática, essa etapa consiste em verificar se o alfabeto de eventos — isto é, os nomes dos eventos que ambos os modelos reconhecem — está devidamente compatível. Caso sejam detectadas discrepâncias entre esses eventos, a execução é interrompida com a geração de um erro específico, indicando que não é possível prosseguir devido à incompatibilidade. Essa checagem inicial é fundamental para garantir que a análise posterior compare efetivamente comportamentos correspondentes nos dois modelos.

O corpo principal do algoritmo implementa uma busca em profundidade sobre o LTS da especificação, representada pela função recursiva *is_traces_refined_by*. Para cada transição disponível no estado atual da especificação (obtida no laço **foreach** *spec_transition in spec_lts*), o algoritmo identifica o estado de destino associado (*dest_node*) e o evento que dispara essa transição (*event_lts*). Em seguida, são recuperadas todas as transições possíveis no estado corrente da implementação (*impl_transitions* = *implementation_machine.get_transitions()*). Essas transições passam por um filtro, que considera somente aquelas habilitadas pelas condições atuais do sistema (*available_impl_transitions*). Se algum evento disponível na implementação não estiver previsto na especificação para aquele estado, o algoritmo reconhece a falha de refinamento e retorna *false*, sinalizando a divergência. Caso contrário, a transição da especificação é "executada" na implementação por meio do disparo da transição correspondente (*new_impl_machine* = *implementation_machine.trigger(spec_transition)*), atualizando o estado da máquina da implementação.

Se o evento da transição da especificação for o evento especial `terminate`, o algoritmo considera que o caminho foi explorado com sucesso e retorna `true`, encerrando a busca naquele ramo. Caso contrário, a função é chamada recursivamente com o novo estado da especificação (`dest_node`) e o estado atualizado da implementação (`new_impl_machine`), permitindo a continuação da exploração em profundidade. Assim, o algoritmo percorre sistematicamente todos os caminhos possíveis no LTS da especificação, verificando em cada passo se o comportamento da implementação acompanha corretamente o modelo formal, assegurando a propriedade de refinamento de traces. Os eventos `entered` são tratados de forma especial, pois

indicam a entrada em um novo estado, e são sempre considerados válidos para transições na implementação, desde que o estado atual da implementação corresponda ao estado esperado na especificação.

Uma das principais vantagens do algoritmo proposto é sua capacidade de alinhamento direto com máquinas de estados implementadas em Python. Diferentemente das abordagens tradicionais que verificam propriedades no nível abstrato da especificação formal, nosso método interage diretamente com a FSM da implementação, garantindo que estados e transições ocorram de maneira consistente com o modelo CSP. Esse alinhamento permite a detecção precisa de desvios entre a especificação e a implementação, proporcionando uma verificação mais prática e aplicável para sistemas reais. Além disso, essa metodologia é generalizável para outras implementações, desde que os sistemas modelados mantenham uma estrutura de estados e transições comparáveis e as especificações sejam verificáveis com FDR. Ao combinar CSP com ferramentas de verificação automatizadas, ela oferece uma abordagem rigorosa e repetível para sistemas que exigem alta confiabilidade e estrita adesão às especificações formais.

3.5 ANÁLISE DOS RESULTADOS

Ao aplicar o Algoritmo 1, pode ocorrer uma incompatibilidade entre a especificação e a implementação. Nesses casos, o algoritmo acusa um erro e fornece um contraexemplo — um trace completo até o ponto da falha — que detalha o momento exato em que a execução da implementação diverge da especificação formal, bem como todo o caminho até esse ponto. Esse contraexemplo evidencia não somente o evento que causou a falha, mas também o caminho percorrido, oferecendo uma visão precisa do comportamento incorreto.

A análise dessas informações permite identificar e compreender as causas do erro, que podem incluir eventos inesperados na implementação, condições de guarda mal definidas, ordens incorretas de transições ou interpretações equivocadas de eventos. A seguir, detalhamos os principais tipos de inconsistências que o algoritmo pode detectar. O algoritmo de verificação consegue identificar duas classes principais de inconsistências entre a implementação e a especificação: desalinhamento de eventos e divergência de comportamento. Essa distinção é fundamental para orientar a análise e a posterior correção.

3.5.1 Alfabetos incompatíveis

Antes de verificar os traces, o algoritmo compara os alfabetos de eventos da implementação e da especificação. É essencial que esses conjuntos sejam idênticos. Caso existam eventos definidos em uma das partes e ausentes na outra, o algoritmo detecta essa discrepância e apresenta um retorno específico, sinalizando o desalinhamento entre os modelos. Essa verificação preliminar previne análises incorretas e garante que a comparação ocorra em um contexto semanticamente compatível. Na prática, esse tipo de erro é comum em situações em que eventos são modelados incompletamente, mal nomeados ou simplesmente omitidos em uma das representações.

Um exemplo de alfabeto incompatível pode ocorrer quando a especificação formal define um evento chamado `inspect_component`, mas a implementação utiliza `check_component`. Apesar de ambos os eventos representarem a mesma ação conceitual, a diferença nominal impede o alinhamento necessário para a verificação de refinamento. Nesse caso, o algoritmo interrompe a análise e retorna um erro indicando que os alfabetos não são compatíveis, destacando os eventos discrepantes. Esse tipo de problema geralmente surge de falhas na comunicação entre equipes de desenvolvimento e modelagem ou de mudanças não sincronizadas entre o modelo formal e a implementação.

3.5.2 Divergência de Comportamento

Mesmo com os alfabetos alinhados e compatíveis, a implementação pode executar sequências de eventos que não são admitidas pela especificação formal. Nesses casos, o algoritmo detecta a primeira transição que viola o conjunto de traces definidos pelo modelo abstrato. O contraexemplo gerado apresenta a sequência completa de eventos percorrida por ambos os modelos até o ponto de divergência, facilitando a identificação exata da causa do problema. Essa classe de erro está geralmente associada a falhas de lógica, como guardas incorretas, transições ausentes ou estados mal conectados.

A abordagem baseada em verificação de modelos orienta, assim, o processo de correção, permitindo que desenvolvedores e analistas revisem o comportamento do sistema de forma direcionada. Quando uma transição inesperada ou um comportamento ausente é detectado, a implementação pode ser ajustada por meio da reformulação de guardas, reorganização da lógica

ou refinamento da sequência de ações. Alternativamente, se o erro estiver na especificação formal, é possível revisar o modelo RoboChart e gerar novamente sua semântica CSP para refletir com maior fidelidade os requisitos esperados.

Esse processo dá origem a um ciclo iterativo de refinamento e validação, em que cada execução do algoritmo contribui para alinhar mais precisamente a implementação ao comportamento especificado. Além disso, os logs gerados durante a execução registram os caminhos analisados e os estados visitados, possibilitando uma inspeção detalhada do sistema e fornecendo insumos valiosos para a geração de testes automatizados. Assim, além de assegurar a conformidade, a metodologia favorece a rastreabilidade e a manutenção do sistema à medida que ele evolui. A integração entre verificação formal, contraexemplos e iteração contínua reforça a robustez do processo de desenvolvimento, promovendo a construção de sistemas confiáveis mesmo em cenários de elevada complexidade.

4 ESTUDOS DE CASO

Este capítulo apresenta dois estudos de caso que demonstram a aplicação prática da metodologia proposta neste trabalho. O primeiro estudo de caso aborda uma linha de montagem simples, onde um braço robótico atua como auxiliar, realizando tarefas como inspeção de componentes e movimentação de peças para diferentes estágios do processo. O segundo estudo se refere ao sistema de controle de um robô dispensador de medicamentos, com foco no módulo responsável por localizar e inspecionar os itens (*Locate Medicine*). Por se tratar de um sistema inserido em ambiente hospitalar, ele exige alto grau de precisão e confiabilidade para garantir a dispensação correta dos medicamentos. Ambos os estudos evidenciam a eficácia da metodologia nas etapas de formalização dos requisitos, geração do LTS, verificação de conformidade entre modelo e implementação, e análise dos resultados, destacando seus benefícios e os desafios enfrentados em cenários reais e industriais.

Todos os arquivos utilizados nesses estudos de caso — incluindo os modelos RoboChart, as especificações formais em CSP, o código Python da implementação e demais artefatos gerados — estão disponíveis publicamente nos repositórios do GitHub: pharmacy-artifacts e assembly-line-artifacts. Esses repositórios fornecem os elementos necessários para reprodução, extensão ou validação independente dos experimentos descritos neste capítulo.

4.1 ASSEMBLY LINE

Este estudo de caso apresenta um sistema de controle para uma linha de montagem automatizada, onde um braço robótico auxilia em diferentes etapas do processo produtivo, desde a inspeção até a finalização de um produto. Esse sistema representa uma abstração comum em ambientes industriais modernos, nos quais robôs desempenham tarefas repetitivas e críticas com precisão e autonomia. O estudo foi inicialmente introduzido na Seção 2.2 e modelado formalmente na Figura 14, destacando a aplicabilidade da metodologia baseada em especificações formais na validação de comportamentos esperados em sistemas industriais. Trata-se de um sistema hipotético, proposto visando demonstrar a viabilidade da abordagem em um cenário representativo, porém simplificado, que permite isolar e avaliar com clareza os aspectos essenciais do processo de controle e verificação formal.

O sistema é responsável por executar operações como a coleta de componentes, inspeção

de qualidade, montagem, verificação final, embalagem e encerramento da operação. O modelo incorpora ainda mecanismos de detecção e tratamento de falhas, garantindo que componentes defeituosos sejam identificados e descartados corretamente, e que o sistema adote estratégias de recuperação ou finalização segura quando condições anormais forem detectadas.

Dentre os requisitos essenciais, destacam-se:

- Inspecionar cada componente antes da montagem, descartando aqueles considerados defeituosos.
- Executar a montagem somente após a validação do componente.
- Verificar a montagem, embalar o produto e concluir a operação se não houver falhas.
- Retornar à posição Home e iniciar uma calibração caso o componente esteja defeituoso.
- Encerrar a operação após duas falhas consecutivas, evitando ciclos infinitos de tentativa e erro.

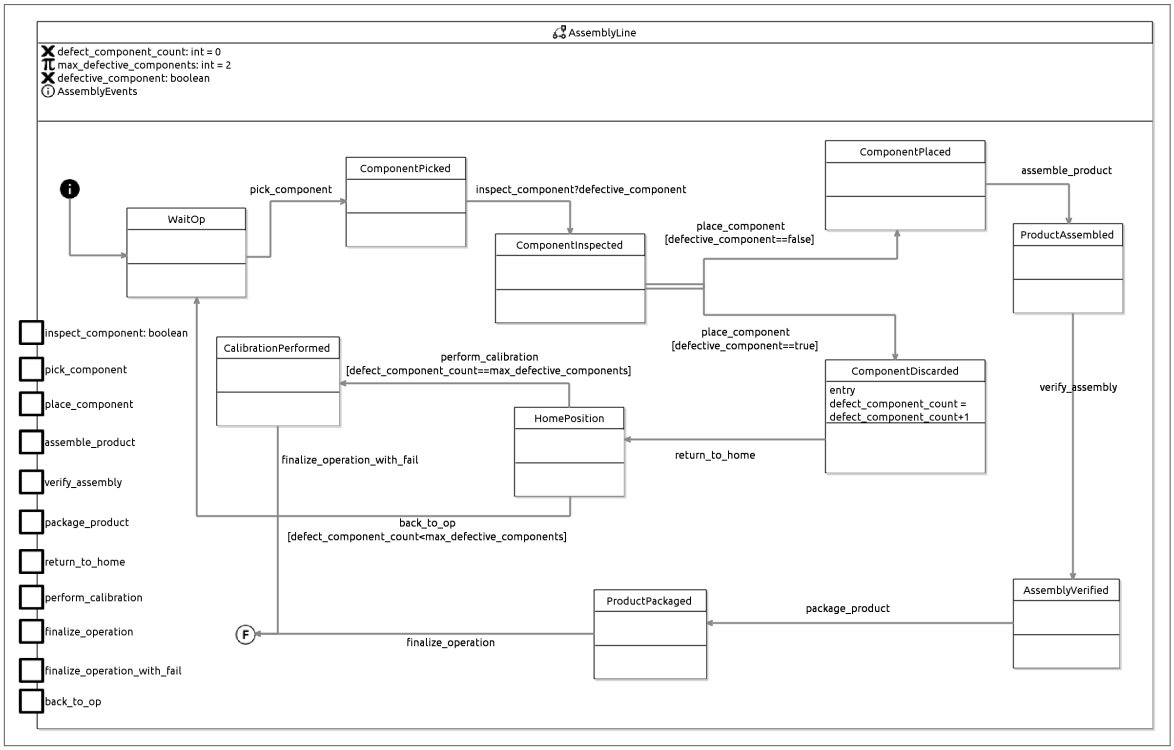
O modelo completo da máquina de estados responsável por controlar o braço robótico nesse cenário é composto por doze estados, incluindo estados iniciais, intermediários e finais, organizando o fluxo da operação de forma sequencial e robusta. Essa estrutura foi implementada com base nas boas práticas de modelagem formal, utilizando a ferramenta RoboChart, e posteriormente convertida para um modelo formal em CSP. O uso dessa abordagem permite representar de maneira clara os comportamentos esperados do sistema, facilitando tanto a verificação automática quanto a identificação de inconsistências entre a especificação e a implementação.

4.1.1 Formalização dos Requisitos e Obtenção do LTS

A descrição informal dos requisitos funcionais do sistema da linha de montagem é expressa como segue:

> "O braço robótico auxiliar da linha de montagem deve pegar um componente, inspecionar o componente e, caso ele não seja defeituoso, levá-lo para a montagem, verificar a montagem, embalar o produto e finalizar a operação. Caso o componente seja defeituoso, o braço deve retornar para a posição Home e iniciar o

Figura 14 – Assembly Line formalizado em RoboChart.



Fonte: Elaborada pelo autor (2025)

processo de calibração. Se ocorrerem duas falhas consecutivas durante o processo, o sistema deve retornar à posição Home e encerrar a operação."

Para alinhar a especificação formal ao comportamento descrito, foi construída uma máquina de estados em RoboChart que reflete essa lógica. Cada etapa da operação foi modelada como um estado distinto, incluindo transições explícitas para o tratamento de falhas. Essa estrutura modela precisamente a sequência de ações e permite que propriedades como ausência de deadlock, correção de fluxo e conformidade com os requisitos sejam avaliadas formalmente.

O modelo resultante da formalização dos requisitos pode ser observado na Figura 14. Ele detalha o fluxo completo do processo, desde o início da operação até o término, incluindo os caminhos alternativos para casos de erro. A estrutura da máquina de estados inclui eventos como *pick_component*, *inspect_component*, *assemble_product*, *verify_assembly*, *package_product*, *finalize_operation* e *return_to_home*, entre outros.

Após a construção do modelo, a ferramenta RoboTool foi utilizada para gerar automaticamente o código correspondente em CSP, o qual confere a semântica dos comportamentos modelados. Em seguida, esse código foi usado como entrada para a geração do grafo de transições LTS, que representa todos os estados possíveis do sistema e suas transições associadas.

Esse grafo é fundamental para a aplicação posterior do algoritmo de verificação de refinamento de traces.

4.1.2 Análise dos Resultados

Esse estudo de caso desempenhou um papel fundamental no desenvolvimento e validação inicial da metodologia proposta, oferecendo um cenário completo para testar as etapas de formalização, geração do LTS e aplicação do algoritmo. Utilizando uma asserção de ausência de deadlock, analisamos o processo CSP *AssemblyLine::VS__*[1], que define o comportamento da máquina de estados RoboChart. Com 15 estados e 29 transições, o modelo apresenta uma complexidade suficiente para representar comportamentos relevantes do sistema, incluindo fluxos normais e situações de falha. A execução do algoritmo gerou logs úteis para análise e possibilitou ajustes na lógica de verificação, contribuindo diretamente para o refinamento da abordagem e demonstrando seu potencial de aplicação em contextos industriais.

No início do desenvolvimento, o algoritmo identificou algumas divergências entre a especificação formal e a implementação, como transições ausentes ou eventos incorretos. Esses problemas foram sendo corrigidos durante o desenvolvimento e testes, garantindo que a implementação seguisse os caminhos definidos no modelo formal. Após os ajustes necessários, o algoritmo rodou sem qualquer falha, confirmando que a metodologia era eficaz em assegurar a conformidade total entre a especificação e a implementação.

No fim, a implementação seguiu fielmente os caminhos especificados no modelo formal em cenários normais e situações de falhas na operação. Por exemplo, ao identificar um componente defeituoso, o sistema transitou corretamente para o estado de calibração, conforme modelado. A limitação de duas falhas consecutivas foi respeitada, com o sistema alcançando o estado final após a terceira tentativa fracassada, atendendo aos requisitos. Os logs detalhados gerados pelo algoritmo não somente confirmaram a conformidade da implementação, mas também forneceram um recurso útil para testes futuros, garantindo que o sistema mantivesse o comportamento esperado mesmo em condições simuladas.

[1] Os arquivos CSP podem ser encontrados em: <https://github.com/felipeadsm/assembly-line-artifacts>

4.2 LOCATE MEDICINE

O estudo de caso analisado, consiste em um sistema de controle que utiliza um braço robótico para dispensar medicamentos em uma farmácia do Hospital das Clínicas da Universidade Federal de Pernambuco. Esse sistema é responsável por executar tarefas críticas, como selecionar medicamentos prescritos para pacientes, garantindo precisão, segurança e eficiência durante a operação. O controle do braço robótico é implementado por meio de máquinas de estados finitos (FSM), que gerenciam as transições entre diferentes estados do sistema, conforme os eventos recebidos e condições definidas.

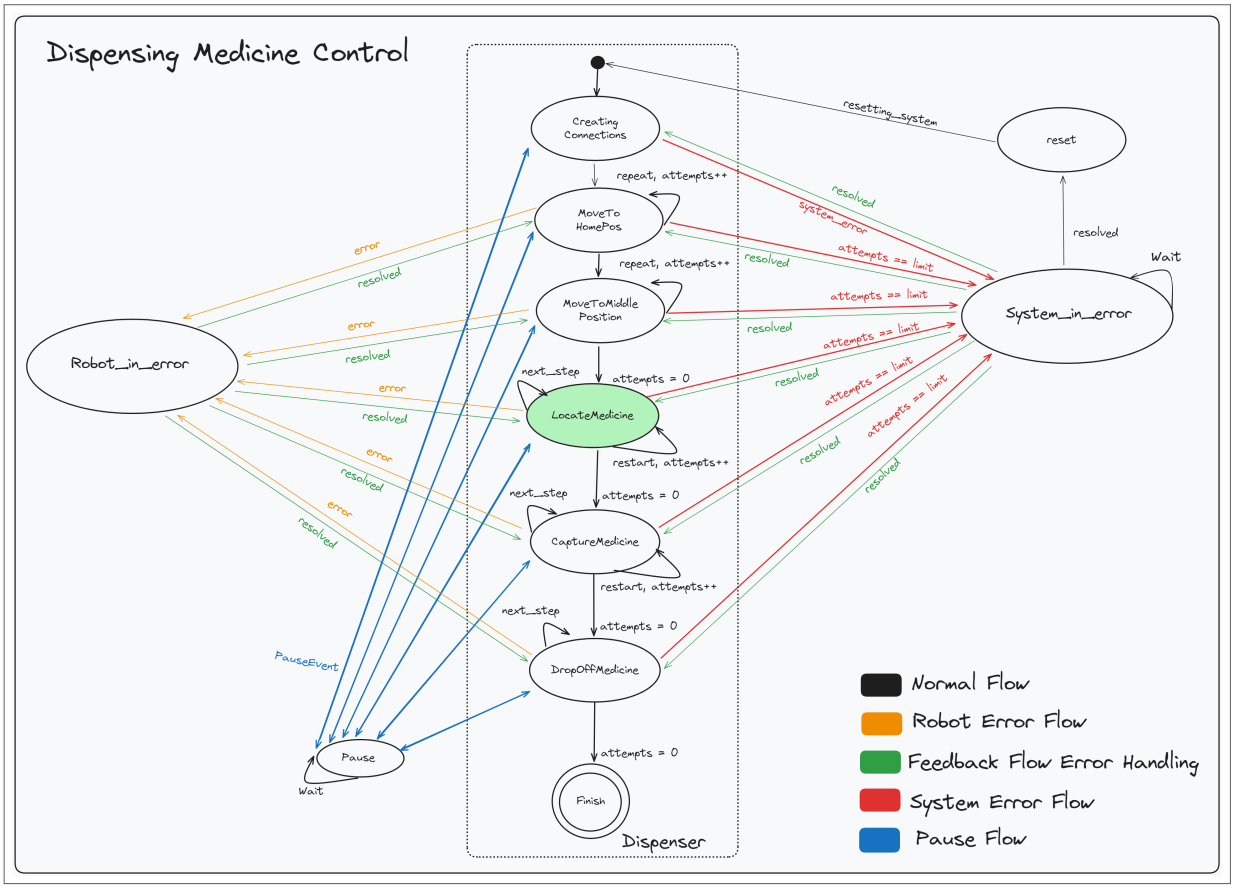
Alguns dos requisitos centrais do sistema especificam que o braço robótico deve ser capaz de:

- Dispensar medicamentos prescritos com base nas informações recebidas.
- Localizar os medicamentos na estante de armazenamento utilizando visão computacional, com estimação de pose baseada em marcadores fiduciais.
- Garantir que o movimento do braço robótico não interfira ou colida com a estante, promovendo segurança durante a operação.
- Permitir pausas e retomadas a critério do operador humano, bem como interromper imediatamente a operação quando solicitado.
- O sistema só pode tentar no máximo duas vezes realizar uma tarefa, caso não consiga, na terceira vez o sistema deve ir para erro, recomendando uma resolução do erro se possível.
- Identificar e tratar erros toleráveis e continuar a operação, enquanto solicita suporte humano em caso de erros críticos.

O modelo geral (informal) do sistema de controle do braço robótico é mostrado na Figura 15. Ele é composto por módulos interconectados que gerenciam funcionalidades essenciais, como controle de movimento do braço, estimativa de pose por visão computacional e gerenciamento de interação com o operador humano. No centro do sistema está a máquina de estados DispensingMachine, que orquestra as operações do braço robótico, desde a identificação do medicamento até a dispensação segura. A máquina de estados DispensingMachine

é composta por três estados, como CreatingConnections e MoveToHomePos, bem como três máquinas de estados, cada uma responsável por uma funcionalidade específica: LocateMedicine, CaptureMedicine e DropOffMedicine. Juntas, elas formam a estrutura hierárquica da DispensingMachine. A adoção dessa estratégia torna o sistema modular, permitindo trabalhar com cada máquina de estados separadamente, além de simplificar significativamente os testes e a validação. Essas máquinas de estados foram implementadas usando a biblioteca *pytransitions*.

Figura 15 – Esboço do sistema: controle de dispensação de medicamentos.



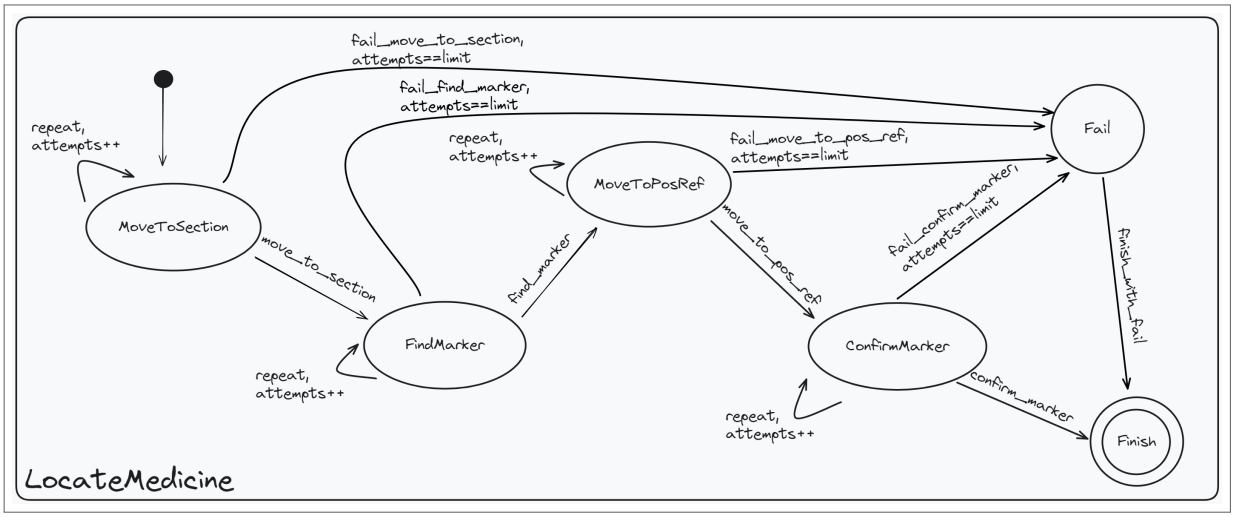
Fonte: Elaborada pelo autor (2025)

Por questões de escalabilidade, restringimos nossa análise a uma parte específica do sistema, em vez de avaliá-lo na totalidade. Essa abordagem seletiva nos permitiu focar na verificação de propriedades comportamentais essenciais, mantendo a viabilidade computacional. Nesse contexto, nosso estudo concentra-se no módulo LocateMedicine, destacado na Figura 15 como parte da estrutura geral do sistema. Este módulo, detalhado na Figura 16, desempenha um papel crucial na identificação da posição dos medicamentos no ambiente de trabalho. Ele integra múltiplos componentes, incluindo controle de braço robótico, aquisição de dados de

câmera e processamento de visão computacional. O módulo gerencia transições entre estados críticos, como detecção de marcadores fiduciais, estimativa de pose e posicionamento preciso, garantindo a precisão e a confiabilidade do processo de dispensação de medicamentos.

A Figura 16 ilustra a máquina de estados LocateMedicine analisada neste estudo, desenhada com a ferramenta *Excalidraw* (EXCALIDRAW, 2020). Tanto a Figura 15, que descreve o sistema como um todo, quanto a Figura 16, que detalha a funcionalidade de localização de medicamentos, serviram como base para a implementação. No diagrama de estados ilustrado na Figura 16, em termos gerais, o braço robótico se move para uma região específica do arranjo físico, procura o marcador associado ao medicamento e, se o marcador for encontrado, move-se para uma posição intermediária próxima ao marcador. O sistema então confirma se o marcador detectado corresponde ao medicamento a ser dispensado.

Figura 16 – Esboço do sistema: Locate Medicine.



Fonte: Elaborada pelo autor (2025)

A metodologia será aplicada ao estudo de caso iniciando com a formalização dos requisitos utilizando RoboChart, seguida da obtenção do LTS da especificação, abstração do LTS da especificação e então a aplicação do algoritmo verificação de refinamento de traces.

4.2.1 Formalização dos Requisitos e Obtenção do LTS

A tarefa de localizar medicamentos envolve uma sequência de operações interdependentes, que devem ser executadas de maneira ordenada e confiável para garantir a segurança e a eficácia do sistema. Essa tarefa é descrita informalmente como:

> O braço robótico deve se mover para uma posição de leitura, identificar o marcador associado ao medicamento, aproximar-se do local estimado, confirmar a identificação do marcador e, se todas as etapas forem bem-sucedidas, concluir o processo. No entanto, se ocorrerem duas falhas de execução, o sistema transita para um estado de falha.

Como este estudo de caso se baseia em um sistema existente, a modelagem de RoboChart se alinhou à estrutura utilizada nos requisitos informais, onde os verbos eram representados como estados. Essa abordagem difere da utilizada na Seção 3.1. Essa adaptação é essencial para alinhar o modelo RoboChart (e, consequentemente, sua semântica formal em CSP) com o comportamento do sistema já implementado, garantindo consistência e precisão na análise.

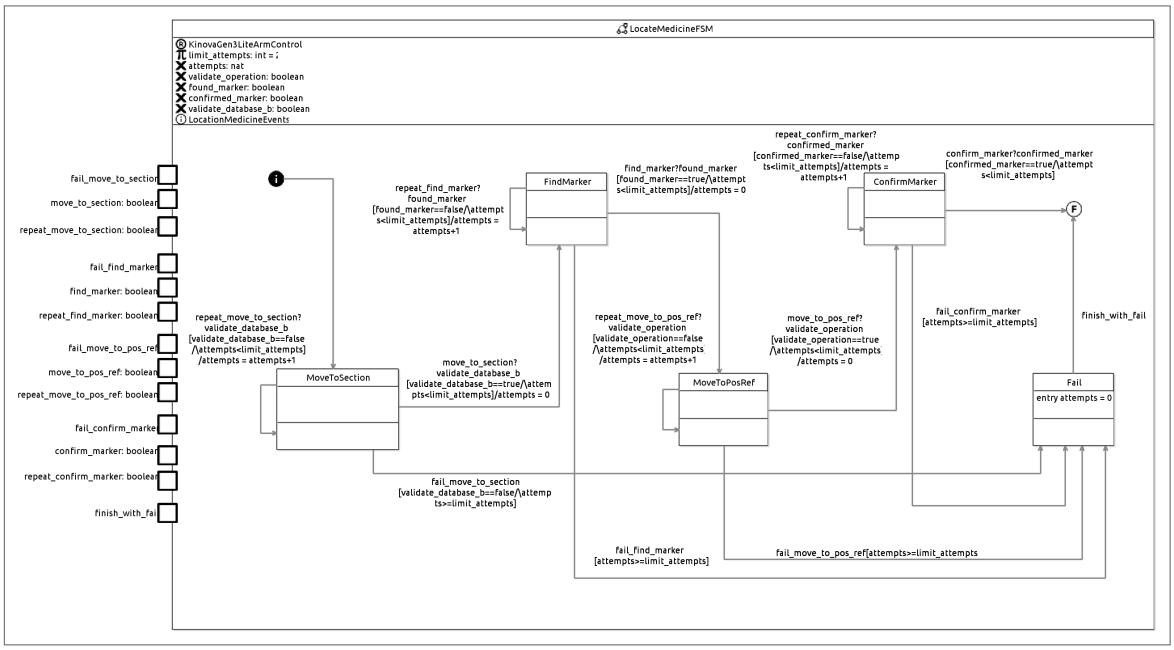
A etapa de formalização de requisitos desempenha um papel crucial na redução de ambiguidades e na captura de detalhes críticos. Nesse contexto, a formalização do requisito de localização do medicamento em RoboChart resultou na criação de uma máquina de estados, mostrada na Figura 17, que encapsula os principais comportamentos e transições do sistema. Cada etapa do processo foi modelada como um estado específico, representando as diferentes fases da operação do braço robótico. Esses estados foram definidos para capturar tanto as principais ações quanto os momentos de verificação necessários para garantir a precisão do sistema. O modelo inclui os seguintes estados:

- MoveToSection: O estado em que o robô inicia a busca pelo marcador fiducial.
- FindMarker: O sistema identifica o marcador associado ao medicamento.
- MoveToPoseRef: O robô se move para a posição estimada próxima ao medicamento.
- ConfirmMarker: O sistema verifica se o marcador identificado corresponde ao medicamento.
- Fail: O processo é interrompido devido a uma tentativa malsucedida ou a um erro.
- Finish: O processo é concluído com sucesso.

A Figura 17 apresenta a máquina de estados de RoboChart resultante da formalização de requisitos informais, representando o comportamento do sistema de controle do braço robótico. Neste modelo, comportamentos como movimentação para a posição de leitura, identificação do marcador associado ao medicamento, aproximação da posição estimada e confirmação do

marcador são representados de forma clara e precisa. Este projeto em RoboChart serve como ponto de partida para a obtenção automática de sua semântica formal em CSP, garantindo que os comportamentos especificados possam ser rigorosamente verificados nas etapas subsequentes.

Figura 17 – Locate Medicine formalizado em RoboChart.



Fonte: Elaborada pelo autor (2025)

A validação do modelo RoboChart poderia ser incluída, mas estenderia o escopo além do objetivo principal do estudo de verificar a conformidade do sistema por meio do LTS. Isso pode ser explorado em trabalhos futuros. Após a formalização dos requisitos como uma máquina de estados RoboChart, as próximas etapas envolvem a obtenção e abstração automáticas do LTS correspondente a partir do código CSP gerado por RoboTool, conforme detalhado nas Seções 3.2 e 3.3. Uma verificação básica de deadlocks, livelocks e determinismo é realizada antes da geração e abstração do LTS, conforme descrito na Seção 3.2. Os processos de geração e abstração do LTS preservam a semântica CSP de todos os construtores em RoboChart. Essa representação em LTS nos permite explorar todas as configurações possíveis do sistema e suas transições, facilitando a aplicação do algoritmo de verificação de refinamento de traces.

4.2.2 Análise dos Resultados

Esta seção apresenta os resultados do processo de verificação para este estudo de caso, destacando os casos em que a implementação foi consistente com os requisitos e os cenários

em que inconsistências foram detectadas. A complexidade adicionada pelo manuseio de diversos fármacos não é considerada por esta máquina de estados, que se concentra somente na lógica de controle do robô. A camada de software Python, que escolhe o fármaco específico a ser administrado, contém essa variabilidade. Consequentemente, o modelo comportamental do robô é propositalmente mantido básico, gerenciando uma única prescrição por vez. Inicialmente, aplicamos a metodologia a um modelo menor e mais gerenciável devido a preocupações preliminares com as possíveis restrições de escalabilidade de FDR4. Apesar da simplicidade intencional do modelo CSP gerado automaticamente por RoboTool, sua validade foi confirmada por meio da ferramenta FDR4, utilizando uma asserção de ausência de deadlock. Analisamos o processo CSP *LocateMedicineFSM::VS__*[2], que define o comportamento da máquina de estados RoboChart mostrada na Figura 17. Como citado na introdução, a modelagem foi propositalmente simplificada para focar no núcleo sequencial do sistema, restringindo o número de estados e transições a fim de garantir a viabilidade da verificação formal. Ele contém 46 estados e 57 transições, incluindo eventos internos (τ). O tempo total de processamento para compilar o modelo e realizar a verificação foi de 0,11 segundos, utilizando um computador com sistema operacional Linux, processador AMD Ryzen 5 5600G a 4400 MHz e 32 GB de RAM.

Como já havia uma implementação inicial construída utilizando os requisitos informais, o primeiro passo foi formalizar os requisitos e obter o LTS. Com base nos requisitos informais e na implementação já construída por uma equipe de desenvolvimento, o sistema apresentado na Seção 4.2 foi formalizado utilizando RoboChart, dando origem ao modelo apresentado na Figura 17.

A aplicação do Algoritmo 1 revelou cenários de consistências e inconsistências entre a especificação formal e o modelo implementado inicialmente. O primeiro cenário de inconsistência foi identificado logo no início da aplicação do Algoritmo 1, no qual o algoritmo relatou divergências nos nomes de estados e transições. Se os nomes não fossem exatamente os mesmos na especificação e na implementação, o algoritmo sinalizaria imediatamente a discrepância, fornecendo o trace exato até o ponto do erro. Inicialmente, a correção envolveu apenas a renomeação de elementos para garantir correspondências exatas entre a especificação e a implementação. Essas alterações foram aplicadas diretamente na implementação para manter a consistência. Esse feedback imediato se mostrou essencial para corrigir erros de nomenclatura e garantir uma correspondência precisa entre o modelo formal e sua implementação prática.

[2] Os arquivos CSP podem ser encontrados em: <https://github.com/felipeadsm/pharmacy-artifacts>

O segundo cenário de inconsistência a destacar é a ausência de uma transição para falha no estado MoveToPosRef na implementação. Nesse estado, não houve resultado para uma falha consecutiva, ou seja, a partir desse estado, o sistema nunca mais apresentou falha, permanecendo em um ciclo sem tratamento. Como resultado, o sistema poderia falhar várias vezes consecutivas sem nenhuma ação corretiva. Ao aplicar a estratégia desenvolvida, o rastro obtido da especificação foi o seguinte:

$$\langle move\_to\_section, find\_marker, repeat\_move\_to\_pos\_ref,$$
$$repeat\_move\_to\_pos\_ref, \mathbf{fail\_move\_to\_pos\_ref}\rangle$$

O trace da implementação após a aplicação da estratégia foi esse:

$$\langle move\_to\_section, find\_marker, repeat\_move\_to\_pos\_ref,$$
$$repeat\_move\_to\_pos\_ref, \mathbf{repeat\_move\_to\_pos\_ref}\rangle$$

Note que o último evento dos dois traces são eventos diferentes e por isso o algoritmo de verificação e refinamento apontou uma divergência. Enquanto a especificação disparou o evento *fail_move_to_pos_ref*, a implementação disparou o evento *repeat_move_to_pos_ref* erradamente, pois, como dito anteriormente, o estado MoveToPosRef da implementação não tinha uma transição que lidaria com falhas consecutivas.

Outro cenário de inconsistência observado foi que, em alguns estados, o limite de execuções com falha excedeu o valor especificado no requisito informal (descrito na Seção 4.2.1), que foi definido como padrão em todo o sistema. Como resultado, a implementação não refina a especificação, pois os traces da implementação não seriam um subconjunto dos traces da especificação, levando a um comportamento diferente do esperado. Em casos de inconsistência como esse, o algoritmo emite o trace da especificação e o trace da implementação, juntamente com informações sobre quais eventos são aceitos pela especificação e pela implementação.

Após a correção dessas inconsistências iniciais, a implementação seguiu de perto os comportamentos esperados definidos no modelo formal. Por exemplo, em situações em que o braço robótico precisava identificar o marcador associado ao medicamento e se mover para a posição correta, a FSM da implementação replicou com precisão o fluxo de transição modelado. Conforme declarado na Seção 3.4, a aplicação do algoritmo gera um relatório contendo todos os caminhos exercitados durante a execução. Ao final de cada execução, esses caminhos podem ser revisados para identificar quaisquer divergências.

4.3 DISCUSSÕES

A formalização de requisitos no desenvolvimento de sistemas robóticos críticos é essencial para garantir que o comportamento esperado seja rigorosamente especificado e validado. Neste trabalho, o uso de RoboChart e sua semântica CSP fornece uma estrutura sólida para descrever sistemas de controle de braços robóticos, permitindo a verificação de propriedades essenciais, como ausência de deadlocks e conformidade funcional. Essa abordagem não somente melhora a confiabilidade do sistema, mas também simplifica a detecção precoce de inconsistências, reduzindo os custos associados a correções em estágios posteriores de desenvolvimento. Vale ressaltar que o método foi aplicado em um projeto real, relacionado à automação da dispensação de medicamentos no Hospital das Clínicas, demonstrando sua aplicabilidade em cenários práticos e relevantes.

Um dos principais impactos dessa metodologia é a capacidade de estabelecer uma ligação clara entre requisitos informais e sua implementação. A modelagem de sistemas robóticos usando RoboChart e CSP facilitou a transição de conceitos abstratos para especificações concretas, que foram posteriormente validadas por meio do método desenvolvido. No contexto do projeto em andamento, isso garantiu que as operações críticas do braço robótico fossem descritas e analisadas com precisão, garantindo o atendimento de requisitos essenciais, como navegação segura no ambiente de trabalho e identificação correta dos medicamentos. Essa integração aumentou a rastreabilidade dos requisitos, um fator crucial para a segurança e a precisão exigidas em sistemas de dispensação de medicamentos hospitalares.

A aplicação do método em um sistema real em desenvolvimento trouxe resultados tangíveis, demonstrando a viabilidade da abordagem para projetos práticos e complexos. A formalização e a validação contínuas contribuíram para um desenvolvimento mais estruturado e para a identificação de melhorias no sistema de controle do robô. Além disso, o uso de ferramentas como RoboTool e FDR para validar a especificação em relação à implementação destaca a capacidade do método de integrar práticas acadêmicas e industriais. Essa conexão entre teoria e prática fortalece o impacto do trabalho e sua relevância para sistemas críticos reais.

Apesar de suas vantagens, este trabalho enfrentou desafios significativos, particularmente no controle da execução da implementação em Python e na extração do LTS da especificação usando a API FDR. Um dos principais desafios foi garantir que a máquina de estados implementada seguisse exatamente o comportamento especificado no modelo CSP, respeitando a cadência dos eventos. Para superar esse problema, foi necessário implementar uma estratégia

baseada em variáveis de controle sincronizadas entre a especificação e a implementação, o que exigiu instrumentação e esforços de verificação adicionais. Embora eficaz, essa solução introduz complexidade ao processo, especialmente quando aplicada a sistemas maiores.

Uma possível solução para lidar com os desafios de complexidade e escalabilidade seria adotar a verificação modular e o raciocínio composicional. Isso envolve a decomposição de sistemas grandes em submódulos menores e a análise de cada parte independentemente, facilitando a validação de sistemas maiores. Na Seção 4.2, o subsistema de 46 estados analisado demonstrou a viabilidade da aplicação dessas abordagens, fornecendo uma base para o escalonamento da verificação para sistemas mais complexos no futuro. O foco no módulo *LocateMedicine* foi escolhido devido à sua viabilidade computacional, permitindo que a metodologia seja efetivamente validada em limites práticos. Embora o estudo de caso seja específico e limitado a um módulo de controle mais restrito, ele serve como uma base sólida para a aplicação da metodologia em sistemas maiores e mais complexos.

Outro desafio relevante foi extrair o LTS da especificação CSP. A API FDR usada para essa tarefa é limitada à versão 2.7 do Python, apresentando problemas de compatibilidade e restringe o suporte a bibliotecas mais modernas. Além disso, o tratamento da semântica CSP (especificação formal) de RoboChart, gerado automaticamente por RoboTool, apresentou desafios devido à sua complexidade e interpretabilidade limitada, dificultando a adoção de abordagens e ferramentas de análise alternativas.

Nossa metodologia, embora aplicada especificamente aos dois estudos de caso antes apresentados, demonstra potencial significativo para generalização em outros domínios que exigem alta confiabilidade e segurança. A combinação de RoboChart, CSP e FDR fornece uma estrutura versátil e rigorosa para a modelagem formal e verificação de sistemas robóticos, garantindo confiabilidade e segurança em sua operação. O uso de ferramentas automatizadas, como FDR, simplifica a análise e validação de especificações formais, aumentando a eficiência do processo de verificação. Além disso, embora a implementação apresentada utilize a biblioteca *pytransitions*, a abordagem proposta não se limita a uma tecnologia específica. Sistemas que utilizam outras bibliotecas de máquinas de estados ou arquiteturas de controle podem ser facilmente integrados, uma vez que o foco da metodologia está no alinhamento entre o comportamento formalizado e a implementação prática, independentemente da ferramenta empregada. Essa flexibilidade amplia o escopo do método, permitindo sua aplicação em sistemas com diferentes escalas e complexidades. Por exemplo, no controle de tráfego ferroviário, onde os estados representam seções da via e as transições modelam as permissões de movimento, ou em dro-

nes autônomos, onde os estados incluem fases de voo e as transições envolvem comandos e sensores, a formalização e a validação propostas neste trabalho podem ser diretamente adaptadas. Essa capacidade de generalização reforça o valor da abordagem, destacando-a como uma solução robusta e replicável em diferentes contextos.

4.4 AMEAÇAS À VALIDADE

Uma ameaça potencial à validade é a ausência de uma fase de normalização durante a verificação de refinamento, particularmente no contexto do LTS da especificação. A normalização, conforme descrita em (ROSCOE, 2010), é uma etapa fundamental no FDR. Esse processo garante que cada trace finito seja mapeado para um único nó, simplificando as comparações ao reduzir a complexidade estrutural do LTS. No entanto, em nossa metodologia, optamos por avaliar diretamente o LTS bruto, sem normalização, porque nosso sistema não apresenta paralelismo ou propriedades altamente não determinísticas. Assim, o modelo de trace gerado permanece funcionalmente equivalente a uma representação normalizada, tornando desnecessário o processamento adicional.

Outro fator motivador foi a eliminação de eventos redundantes. Na semântica CSP de uma máquina de estados RoboChart, vários eventos são ocultos (e representados por τs no LTS), simplificando o modelo sem comprometer a precisão na avaliação dos comportamentos esperados. Essa abordagem é particularmente eficaz nos sistemas abordados, onde os fluxos de trabalho são lineares e não exigem resolução de conflitos entre múltiplos caminhos possíveis. Assim, a ausência de normalização não compromete a análise, mas permite um foco mais direto na comparação de Traces entre a implementação e a especificação. Apesar dessa justificativa, é importante reconhecer que a ausência de normalização pode limitar a aplicabilidade de nossa abordagem a sistemas mais complexos, especialmente aqueles que envolvem paralelismo ou alto grau de não determinismo. Nesses casos, a ausência de um processo de simplificação estrutural pode dificultar a análise de conformidade, tornando a extensão e a generalização da metodologia mais desafiadoras. Portanto, embora os fluxos de trabalho lineares no estudo de caso tenham minimizado o não determinismo, trabalhos futuros devem explorar como integrar técnicas de normalização, particularmente para sistemas paralelos, onde a normalização se torna uma etapa mais crítica para garantir a simplificação estrutural do sistema.

O processo de formalização de requisitos informais em modelos RoboChart apresenta outra ameaça potencial. Interpretações errôneas ou omissões durante esse processo podem resultar

em uma especificação formal que não consegue capturar o comportamento pretendido do sistema. Por exemplo, requisitos incompletos ou ambíguos podem introduzir erros no modelo RoboChart. Além disso, a natureza manual desse processo introduz o risco de viés, onde julgamentos subjetivos ou suposições incorretas podem influenciar a formalização do sistema. Abordar essa questão requer ferramentas e metodologias aprimoradas para elicitação e formalização de requisitos, como as abordagens propostas no Capítulo 5, incluindo a pesquisa de (SANTOS; CARVALHO; SAMPAIO, 2018), que podem servir de base para avanços futuros nessa área. Para mitigar essa ameaça, esforços são feitos para revisar e validar requisitos durante a fase de modelagem, incluindo revisões iterativas por pares. No entanto, erros humanos nesse processo não podem ser completamente eliminados.

O processo iterativo de verificação e refinamento também apresenta desafios de validade. Embora o ciclo de correção e revalidação permita ajustes iterativos nas especificações e implementações, existe o risco de viés, onde as correções podem inadvertidamente alinhar a implementação a uma especificação incorreta. Para atenuar isso, práticas rigorosas de revisão foram implementadas em cada iteração, garantindo que os ajustes fossem justificados por evidências claras de falha ou inconsistência.

5 TRABALHOS RELACIONADOS

Métodos formais desempenham um papel crucial na garantia da confiabilidade e segurança em sistemas robóticos críticos. Esses métodos permitem a especificação, verificação e validação precisas de propriedades essenciais, como a ausência de deadlocks, correção funcional e segurança operacional. Nos últimos anos, diversas abordagens têm explorado a aplicação de técnicas formais a sistemas autônomos e ciberfísicos, destacando a crescente relevância desse campo para a robótica. Nosso trabalho se concentra na integração de RoboChart e de CSP para formalizar requisitos e validar implementações, com ênfase em sistemas robóticos que lidam com tarefas críticas, como a dispensação de medicamentos.

As pesquisas de Luckcuck et al. (LUCKCUCK et al., 2019) e Schlegel et al. (SILVA et al., 2021) fornecem uma visão geral abrangente da aplicação de métodos formais em sistemas robóticos. As primeiras destacam os desafios e avanços na especificação e verificação de sistemas autônomos, identificando lacunas em metodologias focadas em confiabilidade e segurança. Este último complementa essa perspectiva revisando abordagens baseadas em modelos, enfatizando sua relevância no desenvolvimento de sistemas robóticos adaptativos. Juntos, esses trabalhos fornecem uma base sólida para a compreensão do contexto e da evolução de ferramentas e metodologias na área.

RoboTool tem sido amplamente utilizado para modelar sistemas robóticos. (LI et al., 2024) apresentaram uma estrutura que combina RoboChart e sua ferramenta associada, RoboTool, para o projeto e a verificação formal de controladores robóticos. Por meio de um estudo de caso envolvendo robôs exploratórios, eles demonstraram como a geração automática de código e modelos matemáticos pode facilitar a validação e a simulação em plataformas robóticas reais e independentes de hardware. Essa abordagem não somente verificou as propriedades de segurança, mas também abriu caminho para uma implementação mais confiável de sistemas robóticos.

Complementando e expandindo as capacidades de ferramentas anteriores, RoboWorld foi introduzido para oferecer um suporte mais completo ao ciclo de vida de sistemas robóticos, em particular, ao considerar premissas sobre o ambiente (BAXTER et al., 2023). Um editor textual permite documentar de forma precisa restrições e propriedades do ambiente robótico. A semântica formal da documentação é automaticamente gerada por RoboTool, permitindo sua integração com a semântica de modelos RoboChart. Desta forma, tem-se uma abordagem

integrada que permite analisar e testar modelos de sistemas robóticos em conjunto com os seus respectivos ambientes operacionais.

Darolți (DAROLȚI, 2019) utiliza RoboChart para modelar e verificar o comportamento de um robô de limpeza autônomo projetado para operar em painéis solares, com foco na validação de propriedades-chave, como a retomada da operação após a recarga e a cobertura total dos painéis, por meio de verificações baseadas em CSP. O estudo também apresenta uma nova modelagem para controladores PID, explorando os limites da expressividade de RoboChart. Trabalhos como o de Murray et al. (MURRAY et al., 2022) destacam a aplicação de RoboChart em softwares de modelagem para sistemas industriais críticos, como o controle eletrostático de alta tensão (HVC). O estudo demonstra os desafios das abstrações de baixo nível e o equilíbrio entre precisão e complexidade computacional, enquanto Simulink foi utilizado para modelar o hardware. Além disso, Santos et al. (SANTOS; FILHO; SAMPAIO, 2023) exploraram o uso de RoboChart em competições robóticas, como Veículos Aéreos Não Tripulados (VANTs), aplicando a ferramenta para modelar e verificar sistemas de navegação e detecção de objetos, evidenciando sua capacidade de gerenciar a complexidade em sistemas que exigem alta precisão e confiabilidade.

Yan et al. (YAN; FOSTER; HABLI, 2023) propõem uma técnica de verificação composicional automatizada para modelos de máquinas de estados RoboChart usando Isabelle/HOL. Este método utiliza Z-Machines como notação intermediária para transformar modelos RoboChart em uma representação semântica compatível com Isabelle. A técnica permite a verificação de invariantes estruturais e propriedades críticas, como a ausência de deadlocks. Para demonstrar a abordagem, os autores a aplicaram a um estudo de caso envolvendo um veículo subaquático autônomo, validando modos de operação e transições em cenários de alto risco, como colisões com obstáculos. Este trabalho destaca a escalabilidade e a eficiência de métodos formais baseados em provas para garantir a confiabilidade em sistemas robóticos complexos.

O uso de CSP e FSMs provou ser eficaz na formalização de requisitos e na garantia da conformidade do sistema em cenários críticos. Trabalhos recentes exploraram diferentes aspectos dessas ferramentas, destacando suas aplicações em projetos reais. O trabalho de CSP2Turtle (MACCONVILLE et al., 2023), que combina CSP e Python para verificar a navegação de robôs e a prevenção de obstáculos em um mundo de grade 2D, é um exemplo prático desse tipo de abordagem. Da mesma forma, o estudo de Isobe et al. (ISOBE et al., 2021) apresenta uma metodologia que utiliza FSMs concorrentes para modelar e verificar sistemas robóticos cooperativos baseados em eventos. Neste trabalho, a formalização em CSP e a

verificação usando a ferramenta FDR foram aplicadas em um sistema de transporte cooperativo envolvendo robôs autônomos. Essa abordagem destacou o uso de FSMs para representar modos de controle e a validação da comunicação baseada em eventos, demonstrando a eficácia da formalização na detecção de erros de projeto antes da implementação. O uso de CSP com linguagens de programação como Python é um ponto comum em nossa pesquisa.

O desenvolvimento de sistemas robóticos críticos requer métodos formais que vão além da especificação e verificação, integrando diferentes ferramentas e técnicas para lidar com a crescente complexidade desses sistemas. Diversas abordagens têm explorado alternativas a RoboChart e a CSP, aplicando diferentes metodologias para verificar e validar sistemas robóticos em diversos domínios. Uma linha de pesquisa notável é a integração de métodos formais distintos, conforme apresentado por Bourbouh et al. (BOURBOUH et al., 2021). Seu trabalho combina múltiplas ferramentas, incluindo *FRET* (GIANNAKOPOULOU et al., 2020), *COCOSIM* (CARDOSO et al., 2020) e *Event-B* (ABRIAL et al., 2010), guiados pelo framework *AdvoCATE* (DENNEY; PAI, 2018), para verificar propriedades de um sistema de inspeção autônomo. A abordagem destaca os benefícios da integração de artefatos formais de forma coerente ao longo do ciclo de desenvolvimento, garantindo fortes vínculos entre os processos de especificação e validação.

Outras abordagens focam em modelos orientados a tarefas, por exemplo, em (ASKARPOUR et al., 2021) é proposta uma cadeia de ferramentas baseada em perfis UML para facilitar o projeto de sistemas colaborativos. Este trabalho utiliza Papyrus UML (LANUSSE et al., 2009) para modelar tarefas colaborativas de HRC (*Human-Robot Collaboration*) e Zot para verificar formalmente os modelos traduzidos para a lógica TRIO. Além da verificação formal ou da identificação de cenários inseguros, a metodologia inclui ferramentas para automatizar o desenvolvimento e as atualizações de tarefas. Essa abordagem é semelhante à ferramenta desenvolvida neste trabalho, compartilhando as etapas de modelagem usando uma ferramenta baseada em UML e a etapa de verificação formal.

Em cenários onde fluxos de trabalho robóticos exigem avaliação, o trabalho em (RATH-MAIR et al., 2021) explora a verificação formal em camadas, aceitando modelos de entrada em Business Process Model and Notation (BPMN). Em (ROSING et al., 2015), a abordagem permite uma análise abrangente de propriedades em grandes espaços de estados, destacando a importância do refinamento e da abstração em aplicações industriais. A verificação em tempo real foi abordada por (CHANDLER et al., 2023), que apresentou um método de verificação para a criação de planos multietapas em robôs com rodas diferenciais, destacando a geração

de planos em tempo real para lidar com variações ambientais imediatas. A combinação de algoritmos personalizados e discretização de dados LiDAR foi aplicada em cenários complexos de navegação.

Sistemas ciberfísicos distribuídos e críticos foram o foco de (SIRJANI et al., 2021), que apresentaram uma metodologia iterativa utilizando a linguagem Rebeca (REYNISSON et al., 2014) para garantir a verificação de propriedades críticas desde os estágios iniciais de desenvolvimento. Outra contribuição significativa é apresentada em (FOUGHALI; ZUEPKE, 2022), onde os autores combinaram métodos formais e análise de escalonabilidade em uma abordagem interdisciplinar para verificar robôs autônomos em tempo real. O trabalho incluiu a criação de um mecanismo de controle de acesso multirrecursos, demonstrando melhorias no comportamento em tempo real de drones.

O estudo (WEBSTER et al., 2020) abordou a interação humano-robô (HRI) apresentando uma abordagem colaborativa de verificação e validação (V&V). Ele combinou técnicas como verificação de modelos, testes baseados em simulação e experimentos com robôs reais para validar requisitos de segurança e tarefas complexas de manufatura cooperativa. Artigos como (HORVÁTH et al., 2023) e (HOSSEINI; SAUTER; KASTNER, 2023) exploraram abordagens formais aplicadas a contextos industriais. O primeiro focou em modelos SysML simplificados para garantir verificações práticas em escala industrial, enquanto o último utilizou a plataforma AVATAR (*Automated Verification of Real Time Software*) (PEDROZA; APVRILLE; KNORRECK, 2011) para verificar propriedades de segurança em sistemas da Indústria 4.0. Ambos os artigos destacaram a importância de metodologias formais para enfrentar os desafios de sistemas complexos em ambientes industriais.

Para reforçar os diferenciais da abordagem proposta, a Tabela 2 apresenta uma visão comparativa entre os principais trabalhos relacionados a esta dissertação. Os critérios selecionados incluem aspectos fundamentais como a modelagem formal, a geração de código, a verificação de propriedades e a validação prática. Essa comparação destaca objetivamente as lacunas preenchidas por este trabalho em relação às abordagens existentes.

A Tabela 2 evidencia que, embora vários trabalhos explorem o uso de RoboChart, CSP e métodos formais para modelagem e verificação de sistemas robóticos, poucos integram essas etapas com validação prática e foco na rastreabilidade entre especificações e implementação. O diferencial desta dissertação reside justamente na abordagem holística, que cobre desde a formalização manual dos requisitos até a comparação estruturada entre o modelo verificado e o código executável, promovendo maior confiabilidade no desenvolvimento de sistemas robóticos

Tabela 2 – Comparação entre trabalhos relacionados a esta dissertação.

Critério	**(LI et al., 2024)**	**(DAROLTI, 2019)**	**(SANTOS; FILHO; SAMPAIO, 2023)**	**(YAN; FOSTER; HABLI, 2023)**	**Este trabalho**
Modelagem com RoboChart	✓	✓	✓	✓	✓
Verificação formal	✓	✓	✓	✓	✓
Validação com sistema real	×	✓	✓	✓	✓
Comparação formal x implementação	×	×	×	×	✓
Cobertura do ciclo completo	×	×	×	×	✓

críticos.

Embora todos os trabalhos apresentados demonstrem abordagens inovadoras e robustas para a verificação e validação de sistemas robóticos, este artigo se destaca por seu foco em uma questão específica: a comparação sistemática entre especificações formais e implementações práticas. Nossa abordagem combina a precisão de ferramentas como RoboChart e CSP com o refinamento de traces, aplicando essas técnicas diretamente a um sistema real em desenvolvimento. Ao contrário de muitos dos trabalhos citados, que se concentram em etapas específicas do ciclo de desenvolvimento, como modelagem ou geração de código, nosso trabalho integra todo o fluxo de validação, desde a especificação inicial até a análise detalhada da conformidade da implementação com o modelo formal. Essa perspectiva prática e orientada à aplicação reforça a utilidade da metodologia proposta, particularmente em sistemas críticos, onde a confiabilidade da implementação é tão importante quanto a robustez da especificação.

6 CONCLUSÃO

Este trabalho apresentou uma abordagem para verificar a conformidade de sistemas robóticos com suas especificações formais, integrando a linguagem RoboChart, sua semântica baseada em CSP e a implementação prática em Python com a biblioteca pytransitions. A principal contribuição foi a proposta de um fluxo sistemático de desenvolvimento e verificação que inicia na formalização de requisitos, passa pela extração e abstração de LTSs e culmina na aplicação de um algoritmo próprio de refinamento de traces para avaliar se a implementação está em conformidade com o modelo formal.

Na prática, a metodologia proposta foi aplicada a dois estudos de caso com diferentes níveis de complexidade e origem. O primeiro, Assembly Line, foi desenvolvido de forma controlada e inspirada em sistemas industriais, permitindo explorar situações de verificação em um ambiente mais flexível. O segundo estudo de caso, Locate Medicine, representa uma funcionalidade real integrada a um sistema de dispensação automatizada de medicamentos em operação no HC-UFPE. Nele, foi modelada formalmente somente uma parte do processo — a etapa de localização do medicamento — a fim de verificar a viabilidade da abordagem em um cenário prático e crítico. Em ambos os casos, a abordagem possibilitou identificar inconsistências entre os comportamentos esperados e os observados, demonstrando sua utilidade tanto para aplicações reais quanto como ferramenta experimental de apoio ao desenvolvimento de sistemas robóticos. As análises indicam que, mesmo com escopo limitado, é possível aplicar métodos formais de forma eficaz, mantendo a rastreabilidade entre os requisitos e a implementação executável.

Além disso, o algoritmo de verificação desenvolvido neste trabalho evidenciou-se capaz de identificar falhas de conformidade de forma automatizada e precisa, reforçando o papel da verificação formal desde as fases iniciais do ciclo de vida do sistema. Essa verificação antecipada é especialmente valiosa porque permite identificar erros ainda na fase de desenvolvimento, evitando que falhas críticas avancem para etapas mais custosas, como integração, testes finais ou operação em campo. Com isso, o processo de desenvolvimento se torna não somente mais seguro e confiável, mas também mais eficiente, uma vez que reduz retrabalho e favorece a entrega de sistemas com maior aderência aos requisitos. Em domínios regulados ou sensíveis — como saúde, automação laboratorial e processos industriais —, essa abordagem contribui diretamente para o cumprimento de exigências normativas, além de reforçar a rastreabilidade

e a transparência do processo de validação de software embarcado.

Outro ponto de destaque foi a abordagem proposta para traduzir requisitos informais em modelos formais de maneira sistemática, adotando heurísticas de nomeação e modularização que facilitaram a comunicação entre especialistas da área de domínio e engenheiros de software. Essa estratégia buscou reduzir a distância entre a linguagem natural, frequentemente utilizada na especificação de requisitos, e a precisão exigida por métodos formais, promovendo uma modelagem mais acessível e alinhada ao contexto real do sistema. Ao padronizar a estrutura dos modelos e organizar os comportamentos em blocos lógicos e reutilizáveis, foi possível não somente melhorar a clareza dos modelos gerados, mas também favorecer sua manutenção e evolução ao longo do tempo. Essa contribuição é particularmente relevante, dado que a formalização dos requisitos ainda representa uma lacuna significativa em muitos trabalhos que utilizam RoboChart, como apontado na revisão da literatura. A proposta aqui apresentada oferece, portanto, um caminho viável e replicável para incorporar métodos formais desde as primeiras fases do desenvolvimento, aumentando a rastreabilidade, contribuindo para a redução de ambiguidades e fortalecendo a consistência entre os modelos formais e os sistemas que deles derivam.

6.1 TRABALHOS FUTUROS

Como desdobramento natural deste trabalho, várias oportunidades se abrem para avanços metodológicos e técnicos. Uma direção complementar essencial para o avanço deste trabalho consiste na validação da metodologia em ambientes industriais de maior escala, envolvendo projetos e sistemas significativamente mais complexos do que aqueles inicialmente abordados. Embora a metodologia tenha sido aplicada com sucesso em um cenário real, sua aplicação restringiu-se a um exemplo relativamente limitado, reforçando a necessidade de ampliar a avaliação para casos de maior porte e diversidade, característicos do mercado industrial. Exemplos industriais com um espaço de estados muito maior e maior complexidade estrutural poderiam evidenciar desafios práticos adicionais, ampliando a robustez da validação. Esse processo deve incluir o acompanhamento próximo das equipes de desenvolvimento durante todo o ciclo de vida do sistema, possibilitando a coleta sistemática de métricas quantitativas e qualitativas, como o tempo médio para detecção e correção de falhas, o esforço dedicado à modelagem formal, a curva de aprendizado dos profissionais envolvidos e o impacto efetivo na qualidade e confiabilidade do software entregue.

Paralelamente, para garantir a escalabilidade e viabilidade da metodologia em sistemas complexos, é necessário explorar melhorias no processo de verificação, especialmente para lidar com o aumento exponencial do espaço de estados, típico desses sistemas. Investigar técnicas alternativas de abstração de estados, otimizações no algoritmo de refinamento de traces e outras abordagens para reduzir a complexidade computacional pode aumentar consideravelmente a eficiência do método. Essa dupla abordagem, que alia uma validação mais abrangente a avanços técnicos na escalabilidade, permitirá consolidar de forma sólida o valor prático da metodologia, tornando-a mais adequada para adoção em contextos industriais críticos, onde a segurança, a confiabilidade e o desempenho são requisitos fundamentais.

Outra possibilidade relevante e promissora para o avanço desta metodologia é a sua extensão para suportar modelos com restrições temporais explícitas, especialmente aquelas relacionadas a sistemas com requisitos de tempo real. Em aplicações robóticas críticas à segurança, como controle de robôs autônomos e robótica colaborativa, o tempo de resposta é um fator determinante para garantir a integridade e a operação correta do sistema. Modelar essas restrições temporais permite capturar aspectos essenciais como atrasos na percepção, variações no tempo de atuação e limites máximos aceitáveis para respostas seguras, elementos que hoje são suportados somente parcialmente por RoboChart. Incorporar essas propriedades temporais amplia significativamente a expressividade dos modelos e possibilita a realização de verificações temporais específicas, fundamentais para garantir o comportamento esperado em cenários dinâmicos e críticos.

Além disso, a inclusão de comportamentos probabilísticos e estocásticos nos modelos é um passo natural e necessário para lidar com as incertezas inerentes a ambientes reais e sistemas robóticos complexos. Falhas intermitentes, ruídos nos sensores, variações imprevisíveis no ambiente e incertezas na execução são desafios constantes que exigem abordagens capazes de representar e analisar eventos com características probabilísticas. A incorporação dessas técnicas permite a aplicação de model checking probabilístico e análise estocástica, fortalecendo a robustez das garantias fornecidas pela metodologia e ampliando seu escopo para sistemas que operam em condições incertas ou parcialmente observáveis. Dessa forma, a extensão para restrições temporais e comportamentos probabilísticos não só eleva a precisão da modelagem, mas também amplia o impacto prático da metodologia, tornando-a aplicável a uma gama mais ampla de domínios robóticos, como navegação autônoma, sistemas colaborativos e outras aplicações onde a segurança e a confiabilidade dependem do tratamento rigoroso do tempo e da incerteza.

Por fim, um dos caminhos mais promissores para dar continuidade a este trabalho é a automação da tradução de requisitos informais para modelos RoboChart. Embora o processo atual tenha sido manual e baseado em heurísticas bem definidas, ele demonstrou ser viável e eficaz como prova de conceito. No entanto, sua replicação em contextos maiores ou com equipes multidisciplinares exige um suporte automatizado mais robusto. O desenvolvimento de uma ferramenta capaz de interpretar requisitos escritos em linguagem natural — ainda que restrita a domínios específicos — e sugerir automaticamente estruturas formais iniciais em RoboChart poderia não somente acelerar o processo de modelagem, mas também reduzir erros de interpretação e ampliar a acessibilidade dos métodos formais a desenvolvedores não especialistas. Além disso, essa ferramenta funcionaria como uma ponte essencial entre especialistas da área de domínio e engenheiros de software, promovendo uma formalização mais colaborativa, iterativa e rastreável, o que se mostra particularmente valioso em ambientes regulados ou que exigem elevada garantia de conformidade.

REFERÊNCIAS

ABRIAL, J.-R.; BUTLER, M.; HALLERSTEDE, S.; HOANG, T. S.; MEHTA, F.; VOISIN, L. Rodin: an open toolset for modelling and reasoning in Event-B. *International journal on software tools for technology transfer*, Springer, v. 12, p. 447–466, 2010.

ALEXANDROVA, S.; TATLOCK, Z.; CAKMAK, M. RoboFlow: A flow-based visual programming language for mobile manipulation tasks. In: IEEE. *2015 IEEE international conference on robotics and automation (ICRA)*. [S.l.], 2015. p. 5537–5544.

ALUR, R. *Principles of cyber-physical systems*. [S.l.]: MIT press, 2015.

ASKARPOUR, M.; LESTINGI, L.; LONGONI, S.; IANNACCI, N.; ROSSI, M.; VICENTINI, F. Formally-based model-driven development of collaborative robotic applications. *Journal of Intelligent & Robotic Systems*, Springer, v. 102, n. 3, p. 59, 2021.

BAIER, C.; KATOEN, J.-P. *Principles of model checking*. a: MIT press, 2008.

BAXTER, J.; CARVALHO, G.; CAVALCANTI, A.; JÚNIOR, F. R. Roboworld: verification of robotic systems with environment in the loop. *Formal Aspects of Computing*, ACM New York, NY, v. 35, n. 4, p. 1–46, 2023.

BORRELLY, J.-J.; COSTE-MANIERE, É.; ESPIAU, B.; KAPELLOS, K.; PISSARD-GIBOLLET, R.; SIMON, D.; TURRO, N. The ORCCAD architecture. *The International Journal of Robotics Research*, Sage Publications Sage CA: Thousand Oaks, CA, v. 17, n. 4, p. 338–359, 1998.

BOURBOUH, H.; FARRELL, M.; MAVRIDOU, A.; SLJIVO, I.; BRAT, G.; DENNIS, L. A.; FISHER, M. Integrating formal verification and assurance: an inspection rover case study. In: SPRINGER. *NASA Formal Methods Symposium*. a: a, 2021. p. 53–71.

CARDOSO, R. C.; DENNIS, L. A.; FARRELL, M.; FISHER, M.; LUCKCUCK, M. Towards compositional verification for modular robotic systems. *arXiv preprint arXiv:2012.01648*, v. 0, p. 435–499, 2020.

CARVALHO, G.; BARROS, F.; CARVALHO, A.; CAVALCANTI, A.; MOTA, A.; SAMPAIO, A. NAT2TEST tool: From natural language requirements to test cases based on CSP. In: SPRINGER. *Software Engineering and Formal Methods: 13th International Conference, SEFM 2015, York, UK, September 7-11, 2015. Proceedings*. a: a, 2015. p. 283–290.

CASSANDRAS, C. G.; LAFORTUNE, S. *Introduction to discrete event systems*. [S.l.]: Springer, 2008.

CAVALCANTI, A.; DONGOL, B.; HIERONS, R.; TIMMIS, J.; WOODCOCK, J. *Software engineering for robotics*. [S.l.]: Springer Nature, 2021.

CHANDLER, C.; PORR, B.; MILLER, A.; LAFRATTA, G. Model checking for closed-loop robot reactive planning. *arXiv preprint arXiv:2311.09780*, v. 0, n. 1, p. 15, 2023.

DAROLȚI, B. *Software engineering for robotics: an autonomous robotic vacuum cleaner for solar panels*. Dissertação (Mestrado) — Master's thesis, University of York, 2019.

DENNEY, E.; PAI, G. Tool support for assurance case development. *Automated Software Engineering*, Springer, v. 25, n. 3, p. 435–499, 2018.

DHOUIB, S.; KCHIR, S.; STINCKWICH, S.; ZIADI, T.; ZIANE, M. RobotML, a domain-specific language to design, simulate and deploy robotic applications. In: SPRINGER. *Simulation, Modeling, and Programming for Autonomous Robots: Third International Conference, SIMPAR 2012, Tsukuba, Japan, November 5-8, 2012. Proceedings 3*. [S.l.], 2012. p. 149–160.

EXCALIDRAW. *Excalidraw: Virtual whiteboard for sketching hand-drawn like diagrams*. 2020. <https://excalidraw.com>. Accessed: December 29, 2024.

FDR4. *FDR4 API Documentation*. 2016. <https://cocotec.io/fdr/manual/>. Accessed: 2025-01-08.

FLEURY, S.; HERRB, M.; CHATILA, R. GenoM: a tool for the specification and the implementation of operating modules in a distributed robot architecture. In: IEEE. *Proceedings of the 1997 IEEE/RSJ International Conference on Intelligent Robot and Systems. Innovative Robotics for Real-World Applications. IROS'97*. [S.l.], 1997. v. 2, p. 842–849.

FOUGHALI, M.; ZUEPKE, A. Formal verification of real-time autonomous robots: An interdisciplinary approach. *Frontiers in Robotics and AI*, Frontiers Media SA, v. 9, p. 791757, 2022.

GIANNAKOPOULOU, D.; MAVRIDOU, A.; RHEIN, J.; PRESSBURGER, T.; SCHUMANN, J.; SHI, N. Formal requirements elicitation with FRET. In: *International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ-2020)*. a: a, 2020. p. 10.

GIBSON-ROBINSON, T.; ARMSTRONG, P.; BOULGAKOV, A.; ROSCOE, A. W. FDR3—a modern refinement checker for CSP. In: SPRINGER. *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings 20*. a: a, 2014. p. 187–201.

GIESE, M.; HELDAL, R. From informal to formal specifications in UML. In: SPRINGER. *International Conference on the Unified Modeling Language*. a: a, 2004. p. 197–211.

HINCHEY, M. G.; BOWEN, J. P. *Industrial-strength formal methods in practice*. [S.l.]: Springer Science & Business Media, 2012.

HOARE, C. A. R. et al. *Communicating sequential processes*. a: Prentice-hall Englewood Cliffs, 1985. v. 178.

HORVÁTH, B.; MOLNÁR, V.; GRAICS, B.; HAJDU, Á.; RÁTH, I.; HORVÁTH, Á.; KARBAN, R.; TRANCHO, G.; MICSKEI, Z. Pragmatic verification and validation of industrial executable SysML models. *Systems Engineering*, Wiley Online Library, v. 26, n. 6, p. 693–714, 2023.

HOSSEINI, A. M.; SAUTER, T.; KASTNER, W. Formal verification of safety and security properties in industry 4.0 applications. In: IEEE. *2023 IEEE 28th International Conference on Emerging Technologies and Factory Automation (ETFA)*. a: a, 2023. p. 1–8.

ISOBE, Y.; MIYAMOTO, N.; ANDO, N.; OIWA, Y. Formal modeling and verification of concurrent FSMs: Case study on event-based cooperative transport robots. *IEICE TRANSACTIONS on Information and Systems*, The Institute of Electronics, Information and Communication Engineers, v. 104, n. 10, p. 1515–1532, 2021.

LAMSWEERDE, A. v. *Requirements engineering: from system goals to UML models to software specifications*. [S.l.]: John Wiley & Sons, Ltd, 2009.

LANUSSE, A.; TANGUY, Y.; ESPINOZA, H.; MRAIDHA, C.; GERARD, S.; TESSIER, P.; SCHNEKENBURGER, R.; DUBOIS, H.; TERRIER, F. Papyrus UML: an open source toolset for MDA. In: CITESEER. *Proc. of the Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009)*. a: a, 2009. p. 1–4.

LEVESON, N. G. *Engineering a safer world: Systems thinking applied to safety*. [S.l.]: The MIT Press, 2016.

LI, W.; RIBEIRO, P.; MIYAZAWA, A.; REDPATH, R.; CAVALCANTI, A.; ALDEN, K.; WOODCOCK, J.; TIMMIS, J. Formal design, verification and implementation of robotic controller software via RoboChart and RoboTool. *Autonomous Robots*, Springer, v. 48, n. 6, p. 14, 2024.

LUCKCUCK, M.; FARRELL, M.; DENNIS, L. A.; DIXON, C.; FISHER, M. Formal specification and verification of autonomous robotic systems: A survey. *ACM Computing Surveys (CSUR)*, ACM New York, NY, USA, v. 52, n. 5, p. 1–41, 2019.

LYNCH, K. M.; PARK, F. C. *Modern robotics*. [S.l.]: Cambridge University Press, 2017.

MACCONVILLE, D.; FARRELL, M.; LUCKCUCK, M.; MONAHAN, R. CSP2Turtle: Verified turtle robot plans. *Robotics*, MDPI, v. 12, n. 2, p. 62, 2023.

MALL, R. *Fundamentals of software engineering*. [S.l.]: PHI Learning Pvt. Ltd., 2018.

MALLET, A.; PASTEUR, C.; HERRB, M.; LEMAIGNAN, S.; INGRAND, F. GenoM3: Building middleware-independent robotic components. In: IEEE. *2010 IEEE International Conference on Robotics and Automation*. [S.l.], 2010. p. 4627–4632.

MENDONÇA, F.; CONSERVA, M.; MOTA, A. Investigating the correctness of a robotic system via RoboChart: A real-world Medication Dispensing System. Artigo submetido para publicação. 2025.

MITCHELL, M. Complex systems: Network thinking. *Artificial intelligence*, Elsevier, v. 170, n. 18, p. 1194–1212, 2006.

MIYAZAWA, A.; CAVALCANTI, A.; RIBEIRO, P.; LI, W.; WOODCOCK, J.; TIMMIS, J. *RoboChart reference manual*. [S.l.], 2017.

MIYAZAWA, A.; RIBEIRO, P.; LI, W.; CAVALCANTI, A.; TIMMIS, J.; WOODCOCK, J. Robochart: a state-machine notation for modelling and verification of mobile and autonomous robots. *University of York, Department of Computer Science, York, UK, Tech. Rep*, v. 18, p. 3097–3149, 2016.

MIYAZAWA, A.; RIBEIRO, P.; LI, W.; CAVALCANTI, A.; TIMMIS, J.; WOODCOCK, J. RoboChart: modelling and verification of the functional behaviour of robotic applications. *Software & Systems Modeling*, Springer, v. 18, p. 3097–3149, 2019.

MURPHY, R. R. *Introduction to AI robotics*. [S.l.]: MIT press, 2019.

MURRAY, Y.; SIREVÅG, M.; RIBEIRO, P.; ANISI, D. A.; MOSSIGE, M. Safety assurance of an industrial robotic control system using hardware/software co-verification. *Science of Computer Programming*, Elsevier, v. 216, p. 102766, 2022.

PEDROZA, G.; APVRILLE, L.; KNORRECK, D. AVATAR: A SysML environment for the formal verification of safety and security properties. In: IEEE. *2011 11th Annual International Conference on New Technologies of Distributed Systems*. a: a, 2011. p. 1–10.

RATHMAIR, M.; HASPL, T.; KOMENDA, T.; REITERER, B.; HOFBAUR, M. A formal verification approach for robotic workflows. In: IEEE. *2021 20th International Conference on Advanced Robotics (ICAR)*. a: a, 2021. p. 670–675.

REYNISSON, A. H.; SIRJANI, M.; ACETO, L.; CIMINI, M.; JAFARI, A.; INGÓLFSDÓTTIR, A.; SIGURDARSON, S. H. Modelling and simulation of asynchronous real-time systems using Timed Rebeca. *Science of Computer Programming*, Elsevier, v. 89, p. 41–68, 2014.

ROSCOE, A. *The theory and practice of concurrency*. a: Prentice Hall, 1998.

ROSCOE, A. W. *Understanding concurrent systems*. a: Springer Science & Business Media, 2010.

ROSING, M. V.; WHITE, S.; CUMMINS, F.; MAN, H. D. *Business Process Model and Notation-BPMN.* 2015.

SANTOS, M.; FILHO, M. C.; SAMPAIO, A. A model-based approach to the development and verification of robotic systems for competitions. In: IEEE. *2023 Latin American Robotics Symposium (LARS), 2023 Brazilian Symposium on Robotics (SBR), and 2023 Workshop on Robotics in Education (WRE)*. a: a, 2023. p. 236–241.

SANTOS, T.; CARVALHO, G.; SAMPAIO, A. Formal modelling of environment restrictions from natural-language requirements. In: SPRINGER. *Formal Methods: Foundations and Applications: 21st Brazilian Symposium, SBMF 2018, Salvador, Brazil, November 26–30, 2018, Proceedings 21*. a: a, 2018. p. 252–270.

SCHNEIDER, K.; SHABOLT, J.; TAYLOR, J. G. *Verification of reactive systems: formal methods and algorithms*. [S.l.]: Springer, 2004.

SILVA, E. de A.; VALENTIN, E.; CARVALHO, J. R. H.; BARRETO, R. da S. A survey of Model Driven Engineering in robotics. *Journal of Computer Languages*, Elsevier, v. 62, p. 101021, 2021.

SIRJANI, M.; PROVENZANO, L.; ASADOLLAH, S. A.; MOGHADAM, M. H.; SAADATMAND, M. Towards a verification-driven iterative development of software for safety-critical cyber-physical systems. *Journal of Internet Services and Applications*, Springer, v. 12, n. 1, p. 2, 2021.

TENNER, E. The design of everyday things by Donald Norman. *Technology and Culture*, Johns Hopkins University Press, v. 56, n. 3, p. 785–787, 2015.

THRUN, S. Probabilistic robotics. *Communications of the ACM*, ACM New York, NY, USA, v. 45, n. 3, p. 52–57, 2002.

WEBSTER, M.; WESTERN, D.; ARAIZA-ILLAN, D.; DIXON, C.; EDER, K.; FISHER, M.; PIPE, A. G. A corroborative approach to verification and validation of human-robot teams. *The International Journal of Robotics Research*, SAGE Publications Sage UK: London, England, v. 39, n. 1, p. 73–99, 2020.

WIEGERS, K.; BEATTY, J. *Software requirements*. [S.l.]: Pearson Education, 2013.

YAN, F.; FOSTER, S.; HABLI, I. Automated compositional verification for robotic state machines using Isabelle/HOL. In: IEEE. *2023 27th International Conference on Engineering of Complex Computer Systems (ICECCS)*. a: a, 2023. p. 167–176.