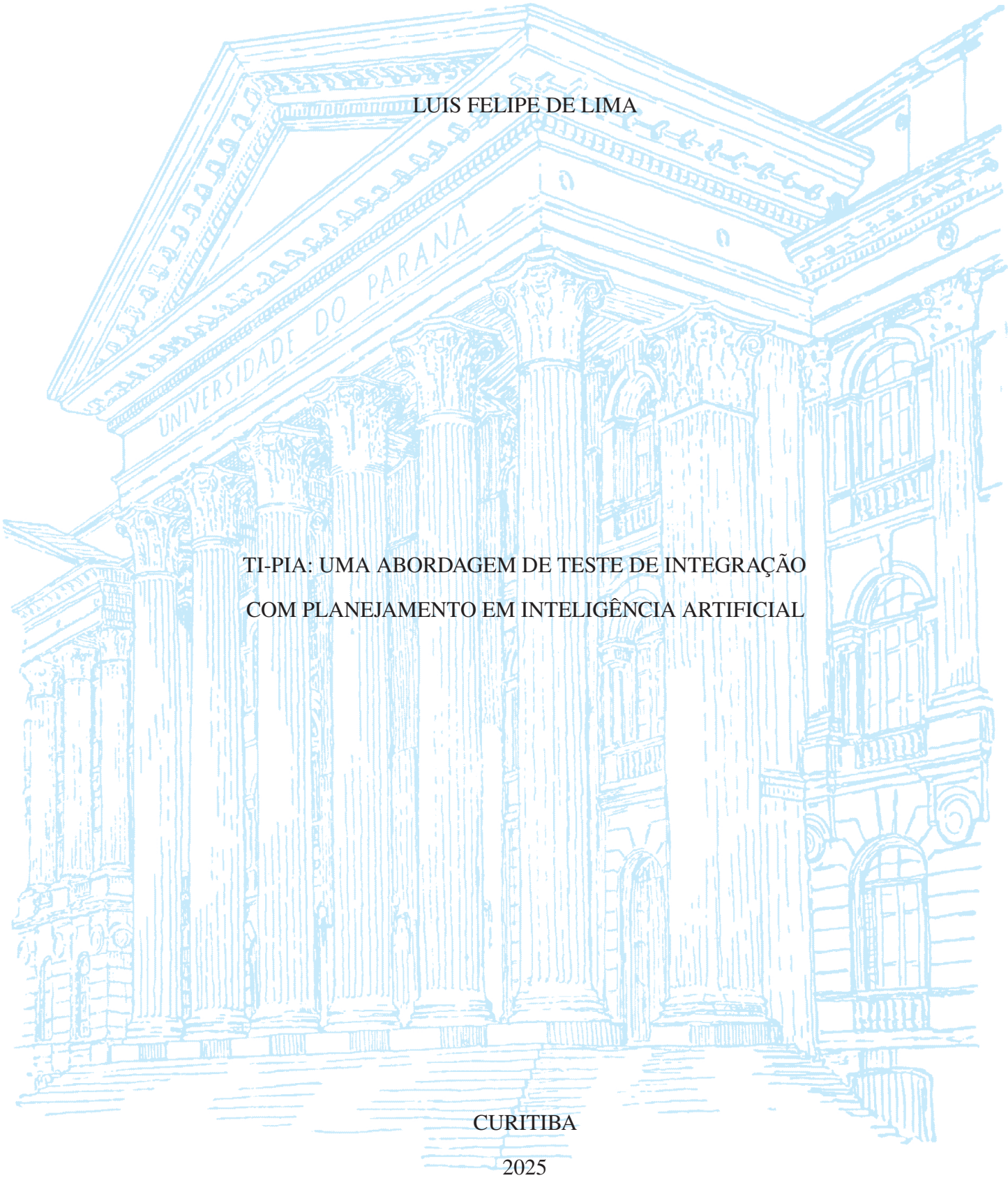
UNIVERSIDADE FEDERAL DO PARANÁ

LUIS FELIPE DE LIMA



TI-PIA: UMA ABORDAGEM DE TESTE DE INTEGRAÇÃO COM PLANEJAMENTO EM INTELIGÊNCIA ARTIFICIAL

CURITIBA

2025

LUIS FELIPE DE LIMA

TI-PIA: UMA ABORDAGEM DE TESTE DE INTEGRAÇÃO COM PLANEJAMENTO EM INTELIGÊNCIA ARTIFICIAL

Tese apresentada como requisito parcial à obtenção do grau de Doutor em Ciência da Computação no Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Computação*.

Orientador: Profª. Drª. Leticia Mara Peres.

CURITIBA

2025

DADOS INTERNACIONAIS DE CATALOGAÇÃO NA PUBLICAÇÃO (CIP)
UNIVERSIDADE FEDERAL DO PARANÁ
SISTEMA DE BIBLIOTECAS – BIBLIOTECA DE CIÊNCIA E TECNOLOGIA

Lima, Luis Felipe de
TI-PIA: uma abordagem de teste de integração com planejamento em inteligência artificial / Luis Felipe de Lima. – Curitiba, 2025.
1 recurso on-line : PDF.

Tese (Doutorado) - Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática.

Orientador: Leticia Mara Peres

1. Software – Testes. 2. Integração de dados (Computação). 3. Inteligência artificial. I. Universidade Federal do Paraná. II. Programa de Pós-Graduação em Informática. III. Peres, Leticia Mara. IV. Título.

Bibliotecário: Elias Barbosa da Silva CRB-9/1894

TERMO DE APROVAÇÃO

Os membros da Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação INFORMÁTICA da Universidade Federal do Paraná foram convocados para realizar a arguição da tese de Doutorado de **LUIS FELIPE DE LIMA**, intitulada: **TI-PIA: Uma Abordagem de Teste de Integração Com Planejamento em Inteligência Artificial**, sob orientação da Profa. Dra. LETICIA MARA PERES, que após terem inquirido o aluno e realizada a avaliação do trabalho, são de parecer pela sua APROVAÇÃO no rito de defesa.

A outorga do título de doutor está sujeita à homologação pelo colegiado, ao atendimento de todas as indicações e correções solicitadas pela banca e ao pleno atendimento das demandas regimentais do Programa de Pós-Graduação.

CURITIBA, 10 de Abril de 2025.

Assinatura Eletrônica
11/04/2025 14:36:48.0
LETICIA MARA PERES
Presidente da Banca Examinadora

Assinatura Eletrônica
11/04/2025 14:52:17.0
CLEBER GIMENEZ CORREA
Avaliador Externo (UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ)

Assinatura Eletrônica
14/04/2025 15:38:29.0
MARCOS ALEXANDRE CASTILHO
Avaliador Interno (UNIVERSIDADE FEDERAL DO PARANÁ)

Assinatura Eletrônica
11/04/2025 10:57:38.0
ALEX MATEUS PORN
Avaliador Externo (INSTITUTO FEDERAL DO PARANÁ CAMPUS UNIÃO DA VITÓRIA)

Assinatura Eletrônica
13/04/2025 20:39:03.0
MARIA CLAUDIA FIGUEIREDO PEREIRA EMER
Avaliador Externo (UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ)

Aos meus pais, Cleide e Valdecir.

“Isso de querer
ser exatamente aquilo
que a gente é
ainda vai
nos levar além.”

Paulo Leminski

AGRADECIMENTOS

Que jornada. Entre graduação, mestrado e doutorado, são quase doze anos ininterruptos na Universidade Federal do Paraná (UFPR). Sou profundamente grato pelo acolhimento recebido e por todas as oportunidades proporcionadas por essa instituição ao longo desses anos. O resultado desse período vai muito além de um trabalho de conclusão, uma dissertação ou uma tese... O que realmente permanece é o amadurecimento pessoal adquirido em todas essas etapas.

Esta tese foi iniciada em um momento pandêmico, no qual pesquisadores e a ciência foram fortemente atacados e desvalorizados. Além de todas as adversidades inerentes à elaboração de um trabalho da magnitude de um doutorado, foi necessária uma dose extra de resiliência diante dessa realidade. Assim, gosto de pensar que finalizar este trabalho, nesse contexto específico, é mais do que uma vitória pessoal: é uma vitória de todos aqueles que acreditam no papel transformador da ciência em nossa sociedade. Certamente, este trabalho não seria possível sem o apoio daqueles a quem expresso meus agradecimentos a seguir.

Aos meus pais, Cleide e Valdecir. Me faltam palavras para descrever o quão importantes vocês foram nesse processo. Agradeço por todo o suporte, a compreensão e o incentivo durante minha vida acadêmica. As frases que mais ouvia de vocês a caminho de cada dia de trabalho eram “Boa aula” e “Tudo vai dar certo”. De certa forma, as aulas sempre foram boas e tudo deu certo. Esta conquista também é de vocês.

Aos meus familiares, que estiveram presentes em todos os momentos. Em especial, deixo meus agradecimentos à minha avó Lourdes (*in memoriam*), meu maior exemplo de afeto, generosidade, serenidade, paciência e conhecimento. Ela sempre dizia com orgulho que, um dia, seu neto seria doutor. Sei que hoje ela está feliz com esta realização.

À minha orientadora Leticia, por todo o aprendizado e pela colaboração durante a condução desta pesquisa. Aos professores do Departamento de Informática (DInf - UFPR) e da Universidade Tecnológica Federal do Paraná (UTFPR - Campus Cornélio Procópio), L’Erario, Fabiano, Roberto, Spinosa e Vignatti, pelas sugestões em reuniões, auxílios em publicações, conversas de corredor e caronas para eventos. Aos professores da banca avaliadora, Alex, Castilho, Cléber e Maria Cláudia, pela disponibilidade de analisar o trabalho e pelas considerações para a melhoria do mesmo. Às minhas professoras do ensino básico, Débora, Reni, Julia e Nadir, por pavimentarem minha trajetória pela educação.

Aos servidores do DInf e do Programa de Pós-Graduação em Informática (PPGInf - UFPR), Jonas, Raquel, Otávio e Lucas, por todas as (muitas) ajudas ao longo desses anos.

Às colegas do Laboratório Fundamentos e Aplicações em Engenharia de Software (Lab FAES - UFPR): Flávia Meireles, pela parceria nas disciplinas remotas e na escrita do projeto e dos artigos; Flávia Haddad e Eduarda, pela companhia e contribuição em nossos seminários de pesquisa; Maria e Cris, pelos momentos de convívio (e café!) no laboratório.

Às entidades que sempre me acompanharam, me guiaram e abriram meus caminhos.

Este trabalho foi realizado com o apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) e do CAPES - Programa de Excelência Acadêmica (PROEX).

RESUMO

O teste de integração ocorre em um cenário complexo influenciado por diversos fatores, como o ambiente, o próprio sistema de software sob teste (SUT), os paradigmas de programação e os tipos de integração utilizados. Para lidar com essa complexidade, é comum que esse teste seja apoiado por artefato de gerenciamento chamado plano de integração. Esta tese propõe o uso do planejamento em inteligência artificial (IA) para a geração desse plano de integração considerando atributos de teste ainda não cobertos na literatura. Para isso, foi definida a abordagem de teste de integração com planejamento em IA (TI-PIA). Essa abordagem contém uma estrutura de geração de planos de integração dividida em módulos associados a representações com uma linguagem de planejamento em IA. Essas representações geram planos de integração para o teste de SUTs desenvolvidos com os paradigmas de programação procedimental e orientado a objetos. Foram conduzidos três estudos avaliativos que investigaram: elementos das representações na linguagem de planejamento em IA adotada; a viabilidade de sua instanciação em projetos de desenvolvimento; e a viabilidade de sua aplicação em contextos da manutenção de software. Os resultados permitem concluir que a geração de planos foi factível para diferentes cenários de teste de integração a partir das representações definidas. Esses achados indicam que a abordagem TI-PIA pode contribuir para a definição do planejamento do teste e, consequentemente, viabilizar uma execução do teste de integração mais criteriosa e bem estruturada.

Palavras-chave: Teste de software. Teste de integração. Planejamento do teste. Planejamento em IA. Planejamento automático.

ABSTRACT

Integration testing occurs in a complex scenario influenced by several factors, such as the environment, the software system under test (SUT) itself, the programming paradigms, and the types of integration. To deal with this complexity, it is common for this testing to be supported by a management artifact called an integration plan. This thesis proposes using artificial intelligence (AI) planning to generate this integration plan, considering testing attributes not yet covered in the literature. For this purpose, we defined the AI planning integration testing (TI-PIA) approach. This approach contains an integration plan generation structure divided into modules associated with representations with an AI planning language. These representations generate integration plans for testing SUTs developed with procedural and object-oriented programming paradigms. We conducted three evaluation studies to investigate: elements of the representations of the adopted AI planning language; the feasibility of its instantiation in development projects; and the feasibility of its application in software maintenance contexts. The results conclude that generating plans was feasible for different integration testing scenarios based on the defined representations. These findings indicate that the TI-PIA approach can contribute to the definition of testing planning and, consequently, enable a more careful and well-structured execution of the integration testing.

Keywords: Software testing. Integration testing. Testing planning. AI planning. Automated planning.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ACRÔNIMOS

ADL	*Action Description Language*
AIPS	*Artificial Intelligence Planning Systems*
ASAP	*Automated Specifier And Planner*
BCC	Bacharelado em Ciência da Computação
CAPES	Coordenação de Aperfeiçoamento de Pessoal de Nível Superior
DInf	Departamento de Informática
DSC	Design Socialmente Consciente
ECP	*European Conference on Planning*
ES	Engenharia de software
FF	*FastFoward*
Lab FAES	Laboratório Fundamentos e Aplicações em Engenharia de Software
GFC	Grafo de fluxo de controle
GPL	Grafo de planejamento
GQM	*Goal-Question-Metric*
IA	Inteligência artificial
ICAPS	*International Conference on Autonomous Planning and Scheduling*
IPC	*International Planning Competition*
IPP	*Interference Progression Planner*
OO	Orientado(s) a objetos
PDDL	*Planning Domain Definition Language*
PPGInf	Programa de Pós-Graduação em Informática
PROEX	Programa de Excelência Acadêmica
QP	Questão de pesquisa
SUT	Sistema sob teste (*system under test*)
STRIPS	*Stanford Research Institute Problem Solver*
TAM	*Technology Acceptance Model*
UFPR	Universidade Federal do Paraná
UML	*Unified Modeling Language*
UTFPR	Universidade Tecnológica Federal do Paraná

LISTA DE SÍMBOLOS

a, A	Ação, conjunto de ações
Am_{oo}	Ambiente de teste de um sistema orientado a objetos
Am_p	Ambiente de teste de um sistema procedimental
ar, Ar	Arco, conjunto de arcos
arg, Arg	Argumento, conjunto de argumentos
$argt, Argt$	Tipo de argumento, conjunto de tipos de argumento
at, At	Tipo de atributo, conjunto de tipos de atributo
atr, Atr	Atributo, conjunto de atributos
c, C	Classe, conjunto de classes
cap	Capacidade de teste
Co	Conjunto de chamadas de operação
$Ct(d, e)$	Caso de teste com um dado de teste d e uma saída esperada e
d	Dado de teste
$D(Prog)$	Domínio do programa $Prog$
e	Saída esperada de um teste
$E(Prog)$	Especificação do programa $Prog$
f, F	Função, conjunto de funções
g	Proposições que estabelecem o estado final
G_{fc}	Grafo de fluxo de controle
G_{pl}	Grafo de planejamento
l, L	Nível, conjunto de níveis
m, M	Método, conjunto de métodos
Me	Conjunto de trocas de mensagem
$nome(a)$	Nome da ação a
o, O	Objeto, conjunto de objetos
p, P	Predicado, conjunto de predicados
pa, PA	Parâmetro, conjunto de parâmetros
P_{iac}	Problema de planejamento em IA clássico
P_{ianc}	Problema de planejamento em IA não-clássico
$pos(a)$	Efeito da ação a
pr, Pr	Prioridade de integração, conjunto de prioridades de integração
pr_c, Pr_c	Prioridade de integração de classe, conjunto de prioridades de integração de classe
pr_m, Pr_m	Prioridade de integração de método, conjunto de prioridades de integração de método

$pre(a)$	Pré-condição da ação a
$proc, Proc$	Procedimento, conjunto de procedimentos
$Prog$	Programa
$Prog(d)$	Execução do programa $Prog$ com o dado de teste d
pt, Pt	Tipo de parâmetro de entrada, conjunto de tipos de parâmetros de entrada
rt, Rt	Tipo de retorno, conjunto de tipos de retorno
rt_m, Rt_m	Tipo de retorno de método, conjunto de tipos de retorno de método
s_0	Estado inicial
s_g	Estado final
SUT_{oo}	Sistema orientado a objetos sob teste
SUT_p	Sistema procedimental sob teste
$size$	Tamanho de uma unidade, um método ou uma classe
t, T	Testador, conjunto de testadores
TS_{max}	Capacidade máxima de teste em um *sprint*
u, U	Unidade, conjunto de unidades
v, V	Vértice, conjunto de vértices
v_0	Vértice de entrada
w	Peso de uma prioridade de integração
Σ	Sistema de estado-transição
γ	Função de transição de estados
π	Plano de IA

SUMÁRIO

1 INTRODUÇÃO

O desenvolvimento de um software envolve um processo complexo, mesmo quando amparado por metodologias de engenharia de software (ES) (Delamaro et al., 2016). Diversos elementos exercem influência no processo de desenvolvimento, tais como as dimensões do software e as habilidades, competências e interpretações dos indivíduos responsáveis por sua criação. Consequentemente, é comum o surgimento de erros, muitas vezes provenientes de equívocos humanos, que podem ocasionar um estado inconsistente no software.

Os testes de software são conduzidos para auxiliar na revelação de falhas no software provenientes de erros (Myers, 1979). No decorrer de um teste, o testador executa o software utilizando dados de entrada específicos e avalia o resultado obtido. Caso o resultado de uma dessas execuções seja diferente do esperado, isso indica a revelação de uma falha no software (Myers, 1979). Uma vez que um estado inconsistente é encontrado no software, medidas podem ser tomadas para corrigi-lo e evitar que uma falha se manifeste no produto final.

Dessa forma, o propósito dos testes é assegurar que o software em desenvolvimento esteja em conformidade com suas especificações. Ainda, a detecção precoce de falhas viabiliza a construção de softwares confiáveis e de qualidade, ao mesmo tempo em que contribui para a redução dos custos de desenvolvimento e manutenção (Garousi et al., 2020). Para alcançar esses benefícios, os testes são realizados em fases ao longo do ciclo de desenvolvimento e podem representar até metade do esforço total investido na produção do software (Beizer, 1990).

As fases de teste se iniciam com avaliações individuais das unidades, ou módulos de programação, do software. Na fase seguinte, essas unidades são progressivamente integradas para formar a arquitetura do software estipulada na especificação. Nessa fase, conhecida como teste de integração, os testes são realizados a medida que as unidades são combinadas, com o objetivo de revelar falhas resultantes das interações entre elas (Lewis, 2004).

Uma vez que o software esteja completamente integrado, são conduzidas fases para verificar se as funcionalidades foram implementadas corretamente e para validar o software em relação aos requisitos do usuário (Lewis, 2004). A divisão do teste em diferentes fases proporciona a flexibilidade necessária para ajustar o teste de acordo com as características do software e as categorias de falhas envolvidas (Delamaro et al., 2016).

Cada fase de teste possui um propósito específico que se alinha com o estágio correspondente do desenvolvimento do software em que é planejada e conduzida. Esse processo estruturado de teste pode ser aplicado a uma ampla variedade de softwares, independentemente de seu domínio de aplicação, e pode ser adaptado de acordo com o paradigma de programação e o modelo de desenvolvimento de software. Portanto, torna-se evidente que a atividade de planejamento do teste é fundamental na condução eficaz dos testes (Burnstein, 2006).

Durante o planejamento do teste é elaborado um artefato conhecido como plano de teste. Esse artefato estabelece os processos e os recursos necessários para alcançar os objetivos pretendidos (Myers, 1979). Do ponto de vista do gerenciamento, o plano de teste é considerado o artefato mais importante (Lewis, 2004). O plano de teste contém a descrição de um plano de integração que indica os processos necessários para a realização do teste de integração. Um plano de teste abrangente e cuidadosamente concebido facilita a realização do teste (Lewis, 2004).

Planejamento pode ser definido como um processo deliberado e abstrato de selecionar e estruturar ações que atingem um objetivo. Assim, o planejamento proporciona antecipar resultados e alcançar um objetivo previamente definido de maneira mais eficaz. Na área da

inteligência artificial (IA), que aborda esse processo de forma computacional, o planejamento é conhecido como planejamento automático[1] (Ghallab et al., 2004).

O planejamento em IA envolve a geração de um conjunto de ações sequenciais, chamado de plano[2], para resolver um problema com base na representação do conhecimento sobre esse problema (Russell e Norvig, 2022). As ações estabelecidas em um plano de IA transformam o estado atual do problema e alcançam o objetivo em um outro estado desejado. Uma vez que o planejamento em IA pode demandar recursos significativos, seu objetivo principal é gerar planos de IA que sejam factíveis em vez de necessariamente ótimos (Ghallab et al., 2004).

Os primeiros estudos relacionados ao planejamento em IA datam da década de 60 a partir de investigações sobre busca no espaço de estados, prova de teoremas e teoria de controle (Russell e Norvig, 2022). Na década de 70, Fikes e Nilsson (1971) propuseram STRIPS como um dos primeiros formalismos de planejamento em IA. STRIPS agregava um algoritmo de resolução e uma linguagem de representação de estados, operadores e transições entre estados por meio de ações. Nos anos seguintes, STRIPS inspirou a criação de formalismos mais expressivos.

Nos anos 80, a *Action Description Language* (ADL) (Pednault, 1989) expandiu a linguagem STRIPS ao permitir pré-condições mais complexas, incluindo disjunções e quantificadores, e efeitos condicionais, em que a aplicação de uma ação depende do estado atual. Na década de 90, com o aumento da complexidade dos problemas abordados, surgiu a *Planning Domain Definition Language* (PDDL) (McDermott et al., 1998). A PDDL incorporou extensões para lidar com características como tempo, recursos e ações condicionais.

Com base nesse contexto, as próximas subseções apresentam os seguintes tópicos: o problema abordado nesta tese (Subseção 1.1), a motivação que direciona esta tese (Subseção 1.2), os objetivos propostos (Subseção 1.3), o método de pesquisa utilizado (Subseção 1.4), as principais contribuições desta tese (Subseção 1.5) e, por fim, a organização dos demais capítulos deste documento (Subseção 1.6).

1.1 PROBLEMA

O teste de software representa a maior parte do esforço técnico no processo de desenvolvimento de software (Pressman e Maxim, 2021). Durante esse processo, a atividade de planejamento do teste deve considerar os aspectos do sistema de software sob teste (SUT, do inglês *system under test*), do ambiente externo, do ambiente de desenvolvimento e das demais atividades de teste, como design de casos de teste e execução do teste.

Um SUT é inerentemente complexo devido à sua composição multifacetada que engloba componentes de software, componentes de hardware e interfaces. Esses elementos interagem entre si e com o ambiente que abrange, por sua vez, os usuários do SUT, outros atores associados e sistemas de software externos. Essas interações acrescentam uma camada significativa de complexidade ao SUT (Binder, 2018).

O teste de integração representa um desafio adicional, pois envolve também as relações entre as unidades do SUT (Binder, 2018). Mesmo após assegurar que as unidades funcionam individualmente, a combinação de unidades pode ocasionar diferentes problemas. Por exemplo, dados podem ser perdidos, uma unidade pode gerar um efeito inesperado ou adverso em outra unidade e funções quando combinadas podem não produzir a funcionalidade principal desejada (Pressman e Maxim, 2021).

Essas características do teste de integração afetam na elaboração dos casos de teste, na escolha de ferramentas adequadas, na identificação e correção de erros (Myers, 1979). Além

[1] Nesta tese, o planejamento automático é chamado de *planejamento em IA*.

[2] Nesta tese, um plano para um problema de planejamento em IA é chamado de *plano de IA*.

disso, as especificidades do teste de integração acarretam na necessidade de atividades específicas, como a definição da sequência em que as unidades são integradas.

Mudanças na abordagem de desenvolvimento de software têm implicações significativas na estratégia de integração a ser adotada. Para sistemas procedimentais, o foco reside na revelação de falhas decorrentes da interação entre procedimentos. Em contrapartida, no contexto de sistemas orientados a objetos (OO), o teste de integração pode se concentrar na revelação de falhas ocasionadas pela interação entre métodos ou classes (Delamaro et al., 2016).

Diante do exposto, percebe-se que o sucesso da execução de um teste está intrinsecamente ligado à forma como o teste foi planejado. Aspectos essenciais do planejamento do teste de integração, como a determinação da ordem de integração, podem ter um impacto significativo na complexidade do processo e contribuir para um aumento substancial do esforço necessário para o teste (Ding et al., 2022). Assim, é evidente a necessidade de suporte técnico e a formulação de estratégias para o planejamento eficaz do teste de integração (Delamaro et al., 2016).

Embora os benefícios do uso de técnicas de planejamento em IA no contexto de teste de software sejam reconhecidos na literatura (Bozic e Wotawa, 2019; Lima et al., 2020d), sua aplicação foi pouco explorada no âmbito do teste de integração. Portanto, o problema de pesquisa central que norteia esta tese de doutorado se concentra em como o planejamento em IA pode oferecer suporte ao processo de teste de integração. Tendo isso em vista, esta tese busca responder a seguinte questão de pesquisa: *"Como o planejamento em IA pode auxiliar nas atividades de planejamento do teste de integração?"*.

1.2 MOTIVAÇÃO

O uso do planejamento em IA é fundamentado por motivações práticas e teóricas (Ghallab et al., 2004). A motivação prática envolve a possibilidade de planejar tarefas complexas de forma acessível, cuidadosa e eficiente. A motivação teórica está relacionada com a contribuição para o entendimento da representação do conhecimento e comportamentos racionais. Desse modo, a aplicação do planejamento em IA no teste de software pode abranger esses benefícios diante da complexidade e da grande quantidade de informação envolvidas nesse processo.

O planejamento em IA é indicado para a tomada de decisão de problemas complexos que ultrapassam a capacidade cognitiva humana de encontrar soluções eficazes. Esses problemas envolvem grande número de variáveis e parâmetros, vários objetivos, múltiplas restrições, necessidade de otimizar recursos, ambientes com alto risco, custos elevados, atividades conjuntas com outras pessoas e atividades sincronizadas (Ghallab et al., 2004). Entende-se que o teste de integração possui elementos que caracterizam um problema complexo.

A principal motivação para aplicar o planejamento em IA no teste de software está na capacidade de abordar problemas complexos com foco em objetivos. A natureza do problema envolvido na execução de um teste, que inclui uma situação inicial bem definida e a busca por objetivos por meio de ações sequenciais, pode ser diretamente associada a um problema de planejamento em IA (Bozic e Wotawa, 2019).

As representações do teste de software com planejamento em IA podem descrever o ambiente associado ao teste e os seus resultados desejados. Ferramentas automáticas, chamadas planejadores, geram planos de IA para o problema modelado. Assim, o planejamento em IA contribui para indicar uma sequência apropriada de ações que atingem os objetivos do teste a partir dos elementos representados.

O planejamento em IA possibilita a parametrização de comandos e a definição de condições sobre eles (Bozic e Wotawa, 2019). Esses aspectos permitem que atividades e serviços associados ao teste de software sejam orquestrados e que as especifidades do software em teste

sejam expressadas nas representações. Essa capacidade de expressão resulta em representações intuitivas, explícitas e legíveis aos interessados.

A flexibilidade do planejamento em IA proporciona o reuso de conhecimento e possibilita maior configuração e adaptação para atender variações do ambiente de teste. Nesse sentido, as linguagens de planejamento em IA formalizam problemas e permitem a representação de critérios. Conforme apontam Delamaro et al. (2016), associar critérios ao teste atribui confiabilidade a esse processo, ou seja, indica uma tendência de que o software apresenta comportamento correto para grande parte do seu domínio de entrada.

O uso de planejadores contribui para a automatização, a corretude e o desempenho das soluções de teste. Além disso, o acesso a ferramentas de código aberto e a não necessidade do uso de bases experimentais favorece a aplicação do planejamento em IA na prática de teste. Ainda, a disponibilidade do código promove a colaboração e o compartilhamento de conhecimento entre os pesquisadores e profissionais das áreas de teste de software e IA.

O Laboratório Fundamentos e Aplicações em Engenharia de Software (Lab FAES, PPGInf-UFPR), grupo de pesquisa em que esta tese foi desenvolvida, vem estudando a aplicação do planejamento em IA na engenharia de software (ES). Foram elaborados estudos sobre o planejamento em IA no teste de segurança de aplicações Web (Lima, 2020) e na alocação de requisitos de software (Pereira et al., 2022). Tais estudos exploram as capacidades das técnicas de planejamento em IA em problemas significativos na ES.

1.3 OBJETIVOS

Considerando a possibilidade do uso do planejamento em IA para a realização de testes de software de forma criteriosa e com embasamento técnico, os elementos envolvidos no planejamento do teste de integração e as especificidades inerentes a SUTs desenvolvidos com os paradigmas de programação procedimental e orientado a objetos, são objetivos desta tese:

- **Objetivo geral**
  - Desenvolver uma abordagem de teste de integração baseada em planejamento em IA que gera planos de integração para sistemas procedimentais e sistemas orientados a objetos.
- **Objetivos específicos**
  a) Delinear um panorama da utilização do planejamento em IA no teste de software;
  b) Caracterizar soluções que aplicam o planejamento em IA no teste de integração;
  c) Propor uma estrutura de geração de planos de integração;
  d) Representar o teste de integração em linguagem de planejamento em IA;
  e) Avaliar elementos da abordagem em estudos de viabilidade.

1.4 MÉTODO DE PESQUISA

O método de pesquisa é composto de nove etapas descritas a seguir com base nos elementos do *Design Science* apresentados na Figura 1.1. O *Design Science* é um paradigma proposto por Engström et al. (2020) para apresentar soluções de pesquisas aplicadas, além de sumarizar, justificar, avaliar e comunicar suas contribuições.

O *Design Science* descreve a pesquisa em três partes: a primeira parte apresenta a proposta em termos de efeito que se deseja alcançar, do contexto em que a proposta se aplica e da solução ou intervenção utilizada; a segunda parte destaca a contribuição empírica da pesquisa, a partir da descrição de uma ou mais instâncias de um par problema-solução e os correspondentes ciclos de entendimento do problema, design de solução e validação; e a terceira parte indica o que se espera de relevância prática, de rigor e de inovação do conhecimento produzido pela pesquisa.

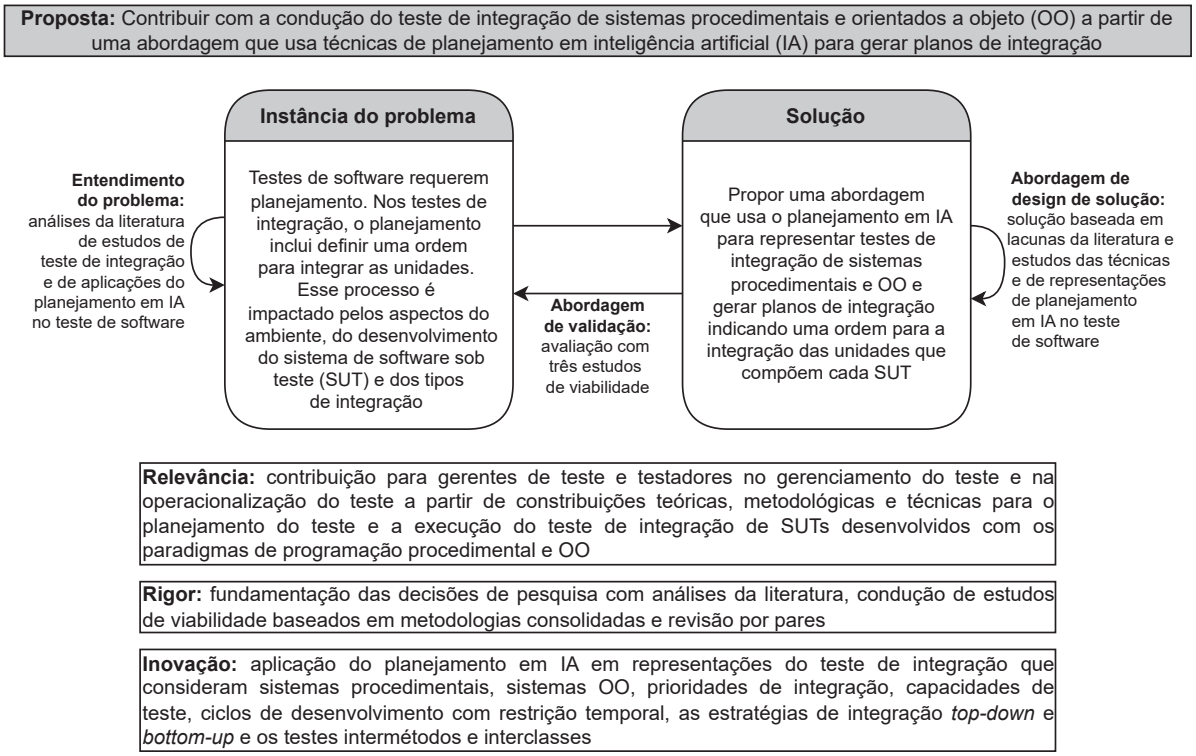


Figura 1.1: Método de pesquisa estruturado com o *Design Science*. Fonte: Adaptado de Engström et al. (2020).

Conforme indicado na Figura 1.1, a *proposta* é referente ao objetivo geral indicado na Subseção 1.3. O *entendimento do problema* abrange as análises da literatura que aprofundam o embasamento teórico da pesquisa. A *instância do problema* são aspectos do planejamento do teste de integração que motivaram o desenvolvimento da solução. A *abordagem de design de solução* inclui estudos que contribuíram para a definição da abordagem.

Diante desses elementos, foi proposta como *solução* uma abordagem composta de uma estrutura e representações do teste de integração com uma linguagem de planejamento em IA. A *abordagem de validação* destaca os estudos que avaliam a abordagem. O aspecto de *relevância* engloba as contribuições esperadas da pesquisa. Na sequência, é indicado o *rigor* metodológico empregado na pesquisa. A *inovação* destaca o diferencial da pesquisa em relação aos estudos correlatos identificados na literatura.

A partir desse delineamento, o método de pesquisa se baseia nas seguintes etapas:

1. **Revisão bibliográfica da literatura**: esta etapa consiste na realização de uma revisão bibliográfica da literatura para identificar estudos que aplicam o planejamento em IA no teste de software, obter uma visão geral das propostas que associam esses conceitos e posicionar a tese perante os estudos da literatura;

2. **Identificação de estratégias de teste de integração**: esta etapa busca identificar estudos que aplicam o planejamento em IA no teste de integração na amostra encontrada na

etapa 1. Os estudos são analisados e as estratégias decorrentes de suas propostas são agrupadas por similaridade de conteúdo e categorizadas;

3. **Análise de trabalhos relacionados**: esta etapa consiste um uma análise dos estudos identificados na etapa 2 para encontrar lacunas de pesquisa que direcionam esta tese;

4. **Definição de módulos de uma estrutura para o teste de integração**: esta etapa busca definir módulos que compõem uma estrutura que aplica o planejamento em IA no teste de integração com base nas lacunas identificadas na etapa 3;

5. **Definição de modelagem para SUTs procedimentais**: esta etapa define uma modelagem das características de SUTs procedimentais e das estratégias de teste de integração procedimental. Os elementos dessa modelagem são formalizados, representados em PDDL e são contidos na estrutura definida na etapa 4;

6. **Definição de modelagem para SUTs OO**: esta etapa consiste em uma modelagem dos elementos de SUTs OO e dos testes de integração OO. A modelagem é formalizada e posteriormente representada em PDDL. Essa modelagem é um dos elementos da estrutura definida na etapa 4;

7. **Condução do estudo de viabilidade 1**: esta etapa consiste na condução de um estudo que avalia a representação em PDDL proposta na etapa 5. O objetivo do estudo é avaliar elementos como atribuição de unidades, as capacidades de teste dos testadores, as capacidade de teste nos ciclos de desenvolvimento, o balanceamento da atribuição das unidades entre testadores e as chamadas de operações entre unidades;

8. **Condução do estudo de viabilidade 2**: esta etapa consiste em um estudo que avalia a viabilidade de instanciação das representações em PDDL propostas na etapas 5 e 6 usando projetos de desenvolvimento. Esse estudo objetiva investigar as representações instanciadas com informações de um SUT procedimental e de um SUT OO e discutir a inteligibilidade dos planos de integração gerados;

9. **Condução do estudo de viabilidade 3**: esta etapa consiste em um estudo que avalia a viabilidade da aplicação da representação em PDDL proposta na etapa 5 na manutenção de software. Esse estudo visa avaliar cenários caracterizados por alteração, inclusão e remoção de unidades. Além disso, esse estudo investiga a aplicação da abordagem em processos diversos da ES.

1.5 CONTRIBUIÇÕES

São contribuições desta tese:

- **Contribuições teóricas**:
  - revisão bibliográfica que fornece um panorama de 35 estudos que aplicam o planejamento em IA no teste de software;
  - delineamento de 4 estruturas que associam o teste de integração com planejamento em IA baseadas nos estudos da revisão bibliográfica;
  - definição de uma estrutura com planejamento em IA que abrange interessados, artefatos, atividades, ferramentas e repositórios envolvidos no teste de integração;

  - extensão de elementos de um plano de integração que considera sistemas procedimentais, sistemas OO, múltiplos testadores, capacidades de teste, prioridades de integração, tipos diversos de integração e *sprints*.

- **Contribuição metodológica**:

  - definição das etapas necessárias para planejar o teste de integração a partir da estrutura proposta.

- **Contribuição técnica**:

  - disponibilização dos artefatos gerados durante a pesquisa em repositórios abertos.

- **Publicações científicas**:

  - *A Tool for Software Requirement Allocation Using Artificial Intelligence Planning* (Pereira et al., 2022);
  - Teste de Integração com Planejamento em Inteligência Artificial (Lima e Peres, 2023);
  - Uma Abordagem de Teste de Integração com Planejamento em Inteligência Artificial (Lima, 2023);
  - *Artificial Intelligence Planning and Software Testing: An Extended Bibliographic Review* (em preparação);
  - *A Survey AI Planning-based Integration Testing Architectures* (em preparação);
  - *Software Requirement Allocation: An Automatic Tool Based On Artificial Intelligence Planning* (em preparação);
  - *Artificial Intelligence Planning Models for Generation of Integration Testing Plans* (em preparação).

- Demais publicações científicas e produções técnicas desenvolvidas durante esta tese são indicadas no Apêndice F.

1.6 ORGANIZAÇÃO DA TESE

Além deste capítulo introdutório, esta tese está organizada em outros cinco capítulos. O Capítulo 2 apresenta os conceitos teóricos que fundamentam a pesquisa. O Capítulo 3 descreve uma revisão bibliográfica sobre o uso do planejamento em IA no teste de software e no teste de integração, destacando trabalhos relacionados à tese. O Capítulo 4 introduz a abordagem, incluindo sua estrutura e as modelagens dos testes de integração procedimental e OO. O Capítulo 5 detalha e discute três estudos que avaliam a abordagem. As considerações finais da tese e os trabalhos futuros decorrentes desta tese são apresentados no Capítulo 6.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta a fundamentação teórica desta tese que consiste nos conceitos de teste de software (Seção 2.1), teste de sistema procedimentais (Seção 2.2), teste de sistemas orientados a objetos (Seção 2.3) e planejamento em inteligência artificial (IA) (Seção 2.4).

2.1 TESTE DE SOFTWARE

Com base na definição de Myers (1979), esta tese entende **teste de software** como o processo de executar um programa $Prog$ em um ambiente controlado, em uma determinada arquitetura e de maneira limitada com a intenção de revelar falhas em $Prog$. Essa conceituação se complementa com o entendimento dos seguintes fundamentos (Delamaro et al., 2016):

- **engano** (do inglês, *mistake*): é uma ação humana que produz um resultado incorreto;
- **defeito** (do inglês, *fault*): é uma linha de código, passo, processo ou definição de dados incorretos;
- **erro** (do inglês, *error*): é um estado inesperado ou inconsistente entre o valor obtido e o valor teoricamente correto conforme a especificação;
- **falha** (do inglês, *failure*): é uma inabilidade do programa de realizar sua função requerida ocasionada pela produção de uma saída incorreta com relação à saída esperada.

Diante desses fundamentos, considere $x = a + b$ como um comando de atribuição em $Prog$. Um possível **engano** pode ser a modificação desse comando para $x = a - b$. Esse engano introduz um **defeito** em $Prog$. Caso o comando seja executado com $b = 0$, nenhum erro é produzido em $Prog$. Para $b \neq 0$, esse defeito produz um **erro** na variável x. O estado inconsistente proveniente desse erro, caso propagado até a saída, causará uma **falha** em $Prog$.

2.1.1 Atividades de teste

Nesta tese, os testes de software são divididos em atividades de planejamento do teste, design de casos de teste, codificação do teste, execução do teste e análise do teste (Souza et al., 2017). A Figura 2.1 ilustra a organização das atividades de teste.

Conforme indicado na Figura 2.1, um sistema de software sob teste[1] interage com elementos do ambiente. O ambiente inclui os usuários do SUT, sistemas externos, os usuários dos sistemas externos, além de outros interessados (do inglês, *stakeholders*) que podem influenciar o uso do SUT. As características do ambiente influenciam a definição das atividades de teste. Além disso, cada atividade envolve interessados com papeis complementares e que atuam colaborativamente para validar o software.

A atividade de **planejamento do teste** define as funções, os responsáveis e os métodos adequados os testes (Humphrey, 1989). Nessa atividade, o gerente de teste define um plano de teste com base nas especificações do SUT geradas e disponíveis em cada momento do desenvolvimento do SUT. O **plano de teste** é um artefato de gerenciamento que auxilia os

[1] Um sistema é um conjunto de partes que interagem para alcançar um objetivo comum (Gouveia e Ranito, 2004). Um sistema de software $S \subset Prog_i$ é um conjunto de i programas $Prog$, onde $i \geq 1$. Nesta tese, S é chamado de sistema de software sob teste (SUT, do inglês *system under test*).

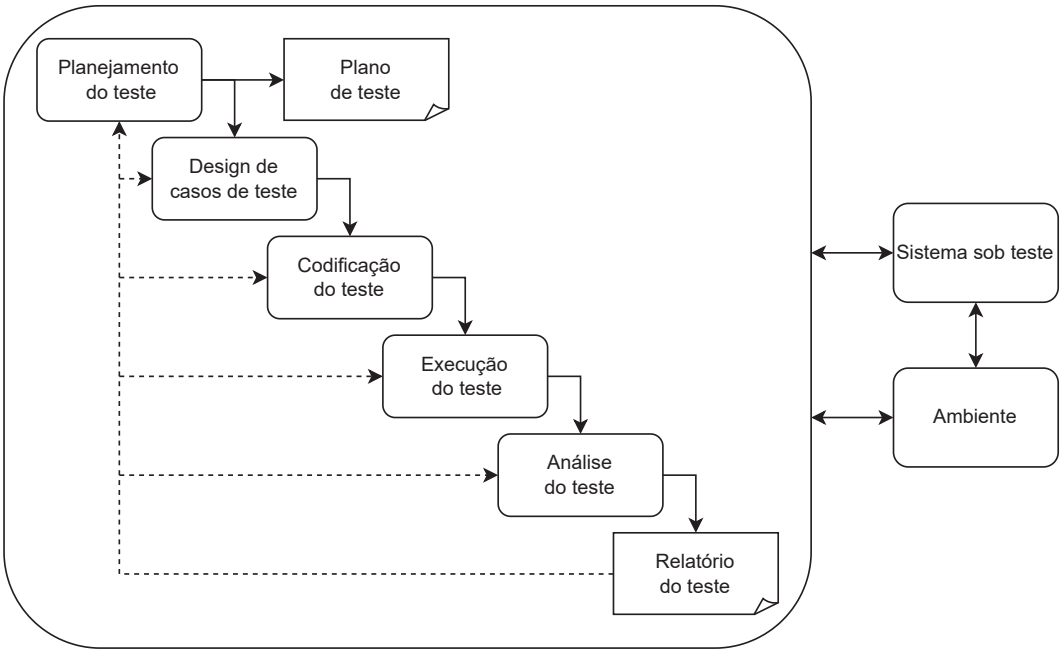


Figura 2.1: Organização das atividades de teste de software. Fonte: Adaptado de Binder (2018).

envolvidos a planejar e a executar as demais atividades de teste de maneira organizada (Lewis, 2004). Myers (1979) destaca os principais elementos contidos em um plano de teste:

- **Objetivos**: indicação do objetivo do teste;
- **Critérios de conclusão**: indicação dos critérios usados para considerar um teste finalizado;
- **Cronograma**: indicação da estimativa do tempo necessário para a condução do teste;
- **Responsabilidades**: indicação dos responsáveis pelo projeto, pela codificação, pela execução dos casos de teste, pela análise dos resultados do teste e pelo posterior reparo dos erros identificados no SUT;
- **Bibliotecas de casos de teste e padrões**: indicação de métodos sistemáticos de identificação, escrita e armazenamento de casos de teste, geralmente requeridos em projetos de larga escala;
- **Ferramentas**: indicação das ferramentas de teste necessárias e de como elas serão implementadas, adquiridas e utilizadas;
- **Tempo de computação**: indicação da estimativa do tempo de computação esperado em cada teste, considerando características de hardware e software como servidores de compilação, *desktops* de instalação e servidores de rede;
- **Configuração de hardware**: indicação de configurações específicas de hardware e dispositivos se necessárias;
- **Plano de integração**: indicação da estratégia de integração, da ordem de integração das unidades, das funcionalidades do SUT testadas a cada integração, dos responsáveis pela execução do teste e pela codificação de implementações auxiliares.

- **Processos de rastreamento**: indicação de aspectos para rastrear o progresso do teste, como estimativas relacionadas ao cronograma, recursos e critérios de conclusão;
- **Processos de depuração**: indicação de mecanismos, cronograma, responsáveis, ferramentas e recursos necessários para relatar falhas reveladas pelo teste e acompanhar as correções adicionadas no SUT; e
- **Teste de regressão**: indicação dos responsáveis e dos processos para a execução dos testes de regressão que visam determinar se alguma mudança (melhoria funcional ou reparo no SUT) regrediu outros aspectos do SUT.

Os casos de teste são projetados na atividade de **design de casos de teste** por um designer de casos de teste. Na sequência, os casos de teste são codificados e executados pelo testador durante as atividades de **codificação do teste** e **execução do teste**. A Subseção 2.1.2 detalha um cenário típico de execução do teste. Voltando à Figura 2.1, a última atividade é a **análise do teste**, em que o testador elabora um relatório do teste que documenta os resultados obtidos. Com base nesse relatório é possível ajustar e reexecutar as atividades conforme necessário.

2.1.2 Um cenário típico de execução do teste

Com base em Delamaro et al. (2016), a Figura 2.2 ilustra um cenário de execução do teste. Os rótulos das setas indicam informações usadas ou geradas durante o teste. Cada programa $Prog$ possui uma especificação $E(Prog)$ que descreve suas funcionalidades. A partir dessa especificação é possível determinar um domínio $D(Prog)$ composto por todos os dados de entrada que podem ser usados em execuções de $Prog$. Cada dado de entrada $d \in D(Prog)$ é chamado de **dado de teste**.

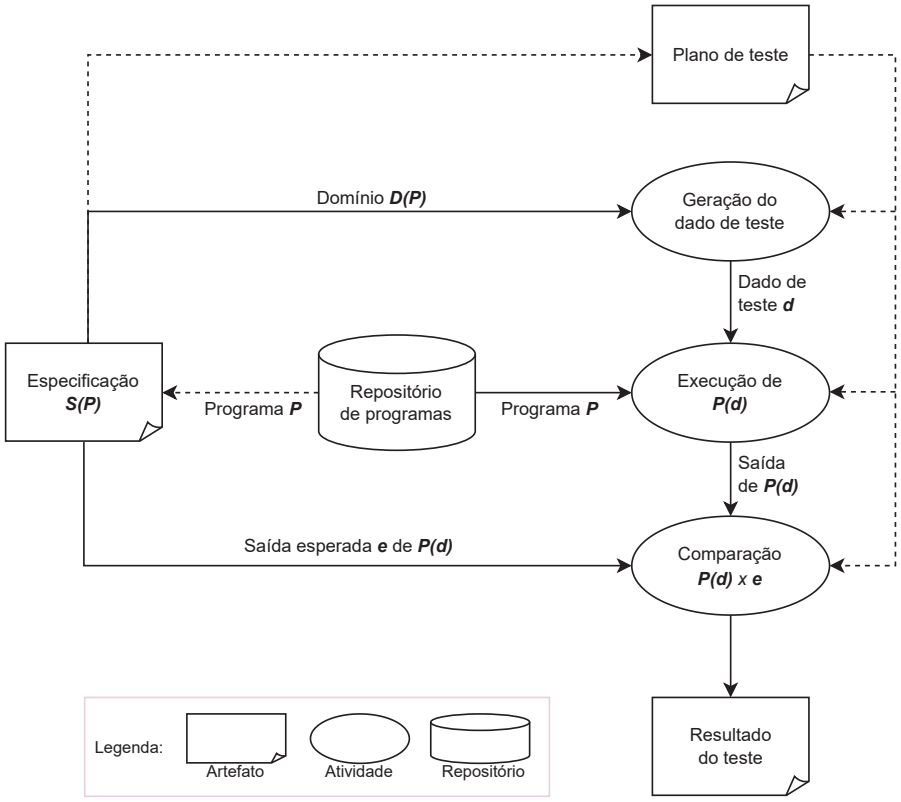


Figura 2.2: Cenário típico de execução de teste de software. Fonte: Adaptado de Delamaro et al. (2016).

Um teste busca revelar falhas com a execução de $Prog$ usando casos de teste $Ct(d, e)$. Um **caso de teste** $Ct(d, e)$ é um par ordenado onde d é um dado de teste e e é a saída esperada na execução de $Prog$ com d. A execução de um teste indica que uma falha foi revelada em $Prog$ caso a saída resultante da execução de algum d em $Prog$, denotada por $Prog(d)$, seja diferente da saída esperada e. Um caso de teste bem-sucedido revela uma falha em $Prog$ não descoberta até aquele momento do teste (Myers, 1979).

Em contrapartida, caso a saída obtida em $Prog(d)$ coincida com a saída esperada e, nenhuma falha foi revelada com $Prog(d)$. Vale ressaltar que a execução de um teste pode mostrar a presença, mas não pode assegurar a ausência de defeitos em $Prog$ (Myers, 1979). Nesta tese, o testador é quem realiza a comparação entre $Prog(d)$ e e. Com base nessa comparação, o testador indica se o teste revelou ou não revelou falhas.

2.1.3 Técnicas de teste

Na prática, a cardinalidade do domínio de entrada de um SUT torna infactível a execução exaustiva de testes com todos os elementos desse domínio (Delamaro et al., 2016). Assim, é necessário selecionar um subconjunto reduzido de dados de entrada para a execução dos testes. A identificação de quais dados devem compor esse subconjunto ocorre a partir de regras estabelecidas por critérios de teste, como critérios de geração de dados de teste e critérios de seleção de dados de teste.

Um **critério de teste** é um predicado que define quais propriedades de um SUT devem ser executadas visando obter um teste sistemático. Os critérios de teste são definidos a partir de técnicas de teste que estabelecem o tipo de informação usada para formar os subdomínios (Myers, 1979; Delamaro et al., 2016). As principais técnicas de teste são baseadas em teste funcional, teste estrutural e teste baseado em erros. Essas técnicas são usualmente utilizadas complementarmente para ampliar a cobertura do teste.

A técnica de **teste funcional**, ou teste de caixa-preta, define os critérios de teste com base na especificação do SUT (Myers, 1979). Dessa forma, há pouca preocupação com a implementação e a estrutura lógica interna do SUT para a definição desses critérios. Logo, a aplicação dessa técnica requer uma documentação detalhada das funcionalidades esperadas do SUT e dos requisitos do usuário. Particionamento em classes de equivalências e análise do valor limite são exemplos de critérios de teste funcional (Delamaro et al., 2016).

A técnica de **teste estrutural**, ou teste de caixa-branca, usa o conhecimento do código-fonte do SUT para a definição dos critérios de teste (Myers, 1979). Essa definição é geralmente baseada em uma representação do SUT como um grafo de fluxo de controle (GFC)[2] (Delamaro et al., 2016). O GFC é construído com as informações do fluxo de controle do SUT, como desvios condicionais, e do fluxo de dados do SUT, como declarações e referências de variáveis. Os critérios de teste estrutural podem considerar a cobertura de vértices, arcos e caminhos do GFC.

A técnica de **teste baseado em erros** estabelece os critérios de teste a partir dos erros mais frequentes ocasionados durante o desenvolvimento de um software (Delamaro et al., 2016). Logo, essa técnica objetiva determinar maneiras de detectar a ocorrência dos erros que o programador possa ocasionar durante o desenvolvimento. Análise de mutantes é um dos critérios de teste baseado em erros comumente utilizado (Delamaro et al., 2016).

[2]Formalmente, o GFC é definido como um grafo orientado $G_{fc} = (V, Ar, v_0)$, onde V é um conjunto de vértices, Ar é um conjunto de arcos e $v_0 \in V$ é o vértice de entrada. Cada vértice $v \in V$ representa um bloco de comandos do SUT executados sequencialmente. Cada arco $ar \in Ar$ representa um possível desvio entre os blocos de V. O GFC contém um único vértice de entrada $v_0 \in V$ e um único vértice de saída. Na representação do GFC, um caminho é uma sequência finita de vértices $(v_1, v_2, ..., v_k), k \geq 2$, tal que existe um arco de v_i para $v_{i+1}, i = 1, 2, ..., k - 1$.

2.1.4 Fases de teste

Esta tese considera os testes de software divididos em fases de teste de unidade, teste de integração, teste de sistema e teste de aceitação (Lewis, 2004). A Figura 2.3 ilustra essas fases organizadas no Modelo V (Spillner e Bremenn, 2002). Esse modelo associa o planejamento e a execução do teste às fases de desenvolvimento do SUT. Assim, o Modelo V enfatiza que o teste é importante durante todo o desenvolvimento e não apenas no final desse processo. A Figura 2.3 destaca em cinza o planejamento do teste explorado nesta tese.

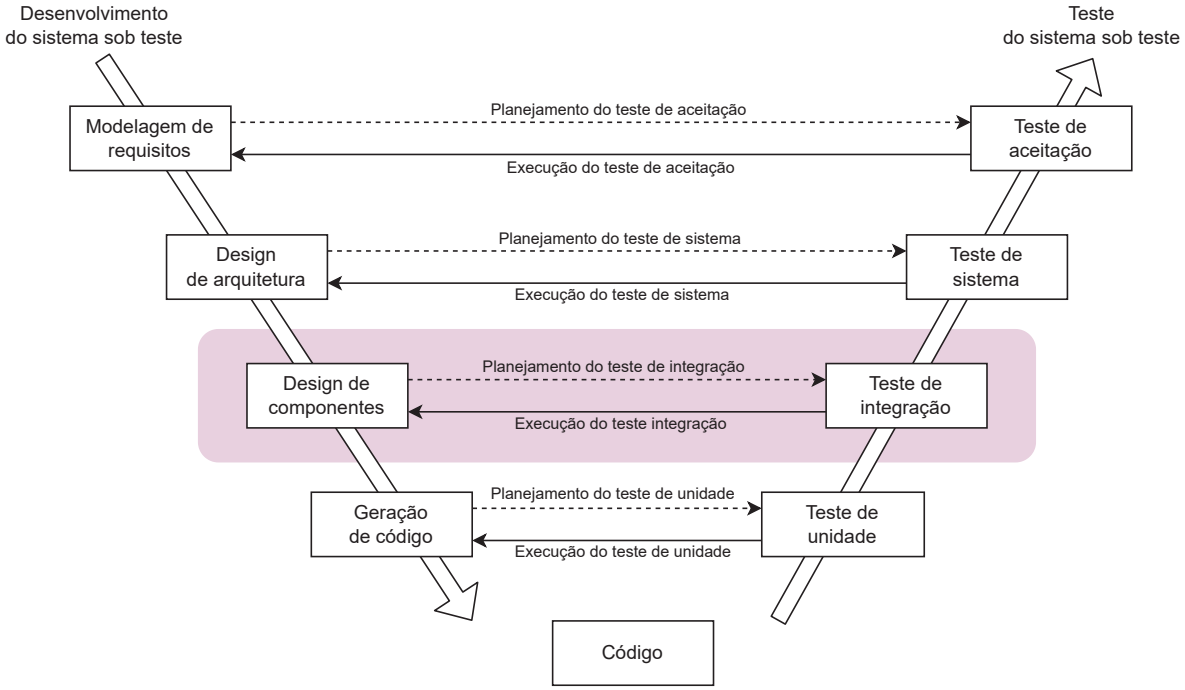


Figura 2.3: Fases de teste no desenvolvimento de software. Fonte: Adaptado de Spillner e Bremenn (2002).

Como indica a seta descendente na Figura 2.3, o desenvolvimento de um SUT é dividido em fases de modelagem de requisitos, design de arquitetura, design de componentes e geração de código (Wazlawick, 2013; Pressman e Maxim, 2021). Inicialmente, na fase de **modelagem de requisitos**, os usuários e a equipe liderada por um engenheiro de software (ou analista) elicitam e documentam os requisitos do SUT (Wazlawick, 2013; Pressman e Maxim, 2021).

Na sequência, na fase de **design de arquitetura**, um arquiteto de software organiza os requisitos em unidades funcionais coesas e define como as partes arquiteturais do sistema de software se conectam e colaboram entre si (Wazlawick, 2013). Nesse contexto, a arquitetura de um sistema é a estrutura que abrange os componentes (unidades) de software, as propriedades externamente visíveis desses componentes e as relações entre eles (Pressman e Maxim, 2021).

Em seguida, na fase de **design de componentes**, um engenheiro de software aprofunda o detalhamento de implementação dos componentes. Cada componente é uma porção reutilizável de um SUT. Finalmente, na fase de **geração de código**, os desenvolvedores implementam os componentes conforme a especificação (Wazlawick, 2013). Os componentes são formados por uma ou mais unidades. As **unidades** são módulos de programação de tamanho pequeno e que não devem ser subdivididos, como procedimentos, funções, subrotinas, métodos e classes.

A seta ascendente da Figura 2.3 mostra a sequência das fases de teste. O **teste de unidade** tem como foco o teste das unidades de um SUT. Cada unidade é testada individualmente com o objetivo de identificar possíveis problemas de lógica e de implementação em algoritmos e

estruturas de dados (Delamaro et al., 2016). O teste de unidade é planejado durante a fase de geração de código, sendo frequentemente baseado em teste estrutural (Delamaro et al., 2016). Os próprios desenvolvedores executam o teste de unidades no decorrer da codificação.

No **teste de integração**, os testes são executados a medida que as unidades do SUT são gradualmente integradas umas às outras (Lewis, 2004). O objetivo desse teste é revelar falhas ocasionadas pela interação entre as unidades. Os testes de integração são geralmente realizados por testadores apoiados pela equipe de desenvolvimento e baseados em teste funcional (Lewis, 2004). O teste de integração é planejado durante a fase de design de componentes.

O **teste de sistema** verifica se as funcionalidades da versão totalmente integrada do SUT estão implementadas conforme a especificação (Lewis, 2004). Logo, o objetivo desse teste é verificar os aspectos de correção, coerência, completude e de requisitos não-funcionais, como segurança e desempenho. O teste de sistema pode ser baseado em técnicas de teste funcional e estrutural e ser conduzido por equipes de teste independentes ao desenvolvimento do SUT. O teste de sistema é planejado durante a fase de design de arquitetura.

O **teste de aceitação** ocorre com a interface da versão final do SUT para verificar se o sistema atende aos requisitos do usuário (Lewis, 2004). Nesses testes, os usuários utilizam o SUT livremente durante um período determinado e exploram suas funcionalidades. Após isso, os usuários realizam uma avaliação seguindo um *checklist* que indica o que é esperado do SUT e enviam suas percepções à equipe de desenvolvimento. Assim, esse teste é geralmente associado ao teste funcional. O teste de aceitação é planejado durante a fase de modelagem de requisitos.

Após a finalização dos testes, o SUT é instalado, configurado e disponibilizado aos usuários em uma fase denominada **implantação**. Uma equipe de suporte pode realizar processos de manutenção necessários após modificações no sistema para melhorá-lo, corrigi-lo ou adaptá-lo às mudanças no ambiente e nos requisitos (Wazlawick, 2013). Durante a manutenção, unidades podem ser alteradas, incluídas ou excluídas. Essas modificações envolvem a realização de todas as fases indicadas no Modelo V.

Nesta tese, as fases de desenvolvimento e as fases de teste do Modelo V são realizadas em entregas iterativas e incrementais com restrição temporal. Cada entrega resulta em um conjunto de funcionalidades implementadas e testadas. Neste texto, essas entregas são chamadas de ***sprints***.

2.2 TESTE DE SISTEMAS PROCEDIMENTAIS

Segundo Baranauskas (1993), considerando o contexto de linguagens de programação, a palavra paradigma pode ser usada para indicar os diferentes modelos utilizados para representar um determinado problema a ser resolvido por um sistema de software. O paradigma de programação impacta nas linguagens, estilos e formas de codificação e teste de um sistema. Esta tese envolve os paradigmas de programação procedimental e orientado a objetos (OO).

O paradigma de programação procedimental (ou paradigma procedimental) baseia-se na declaração de procedimentos em blocos estruturados executados pelo sistema de software em determinada sequência a partir da passagem de dados entre esses procedimentos (Delamaro et al., 2016). Nesta tese, um sistema de software desenvolvido com o paradigma procedimental é chamado de **sistema procedimental**. Nesta tese, o teste de um sistema procedimental é chamado de **teste procedimental**.

Os procedimentos podem ser funções, rotinas ou subrotinas e são entendidos como as unidades do sistema procedimental. A Figura 2.4 ilustra a estrutura de um sistema procedimental. Como é possível observar, um programa principal pode enviar dados aos procedimentos. Além disso, os dados gerados por um procedimento podem ser reutilizados pelo mesmo procedimento

ou encaminhados para outros procedimentos do sistema. Cada procedimento possui um conjunto de parâmetros tipificados e retorna um valor com sua execução.

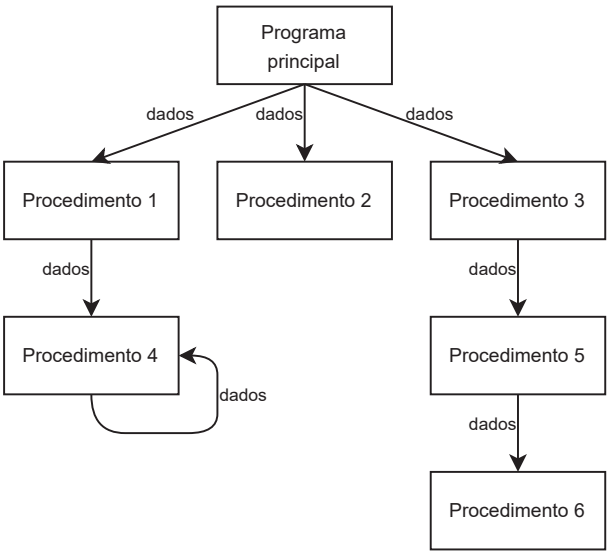


Figura 2.4: Estrutura de um sistema procedimental.

Seguindo as fases de teste do Modelo V indicadas na Seção 2.1.4, o teste procedimental se inicia com o teste de unidade. Nesse contexto, o teste de unidade é realizado sobre os procedimentos do SUT. O teste de unidade é também chamado de teste intraprocedimental. O teste de integração, também conhecido como teste interprocedimental, ocorre a partir da integração dos procedimentos. Por fim, o teste de sistema e o teste de aceitação são realizados com o SUT totalmente integrado.

O teste de integração procedimental pode ser realizado com abordagens não-incremental e incremental. Na integração não-incremental, ou *big-bang*, todos os procedimentos do SUT são combinados e o sistema é testado como um todo (Myers, 1979). Na **integração incremental**, os testes do SUT são realizados em incrementos. Nesse cenário, a integração de cada procedimento gera um subsistema SUT' caracterizado por uma configuração funcional parcial do SUT. Logo, o teste busca revelar falhas em cada SUT'.

A integração incremental de sistemas procedimentais segue uma representação da estrutura da hierarquia de controle do SUT dividida em níveis. As estratégias de integração incremental geralmente utilizadas são a *top-down* e a *botom-up*.

Na estratégia de **integração *top-down***, ou integração descendente, a integração ocorre a partir da unidade principal seguida da integração das unidades subordinadas (Myers, 1979). Desse modo, a estratégia *top-down* permite que as funcionalidades mais importantes para o funcionamento do SUT sejam testadas primeiro. No entanto, pode depender da criação de *stubs*[3] para sua execução e da qualidade dessas implementações para o sucesso do teste.

Na estratégia de **integração *bottom-up***, ou integração ascendente, a integração se inicia com as unidades subordinadas do nível mais baixo que serão integradas com unidades de nível imediatamente mais alto (Myers, 1979). Assim, o teste de uma unidade em um determinado nível só acontece após a integração de todas as suas unidades subordinadas. A estratégia *bottom-up* pode requerer *drivers*[4] e considerar unidades agrupadas (*clusters*) para o teste.

[3] *Stub* é uma implementação que emula e simplifica a funcionalidade de uma unidade subordinada (Myers, 1979).
[4] *Driver* é uma implementação que emula chamadas de operações de uma unidade em teste (Myers, 1979).

2.3 TESTE DE SISTEMAS ORIENTADOS A OBJETOS

O paradigma de programação OO (ou paradigma OO) foi consolidado durante os anos 80 como uma alternativa ao paradigma procedimental. Geralmente, as dependências entre os dados nos procedimentos dificultam a estruturação e a alteração de problemas grandes. Para suprir essa dificuldade, o paradigma OO fornece mecanismos para isolar os dados da forma como são manipulados (Delamaro et al., 2016). Nesta tese, um sistema de software desenvolvido com o paradigma OO é chamado de **sistema OO** e o teste de um sistema OO é chamado de **teste OO**.

Os objetos são os principais elementos envolvidos na programação OO. Cada objeto é uma estrutura contendo um conjunto de informações e operações que disponibilizam uma funcionalidade ao usuário. As informações declaradas em um objeto são chamadas de atributos e as operações que manipulam os atributos são conhecidas como métodos (Masiero et al., 2006). Cada atributo possui um tipo, como inteiro e *booleano*, e a execução de cada método retorna um valor com determinado tipo a partir de operações em um conjunto de argumentos.

Os objetos comunicam-se por mensagens e são considerados instâncias de classes que formam uma estrutura hierárquica. As classes são entidades estáticas que podem ser entendidas como uma abstração de objetos com propriedades iguais ou similares (Siegel, 1996). A Figura 2.5 ilustra a estrutura de um sistema OO em um diagrama de classes simplificado. Como indicado, os dados de um método podem ser usados na mesma classe do método ou em outras classes. Assim, tanto os métodos quanto as classes podem ser entendidos como as unidades de um sistema OO.

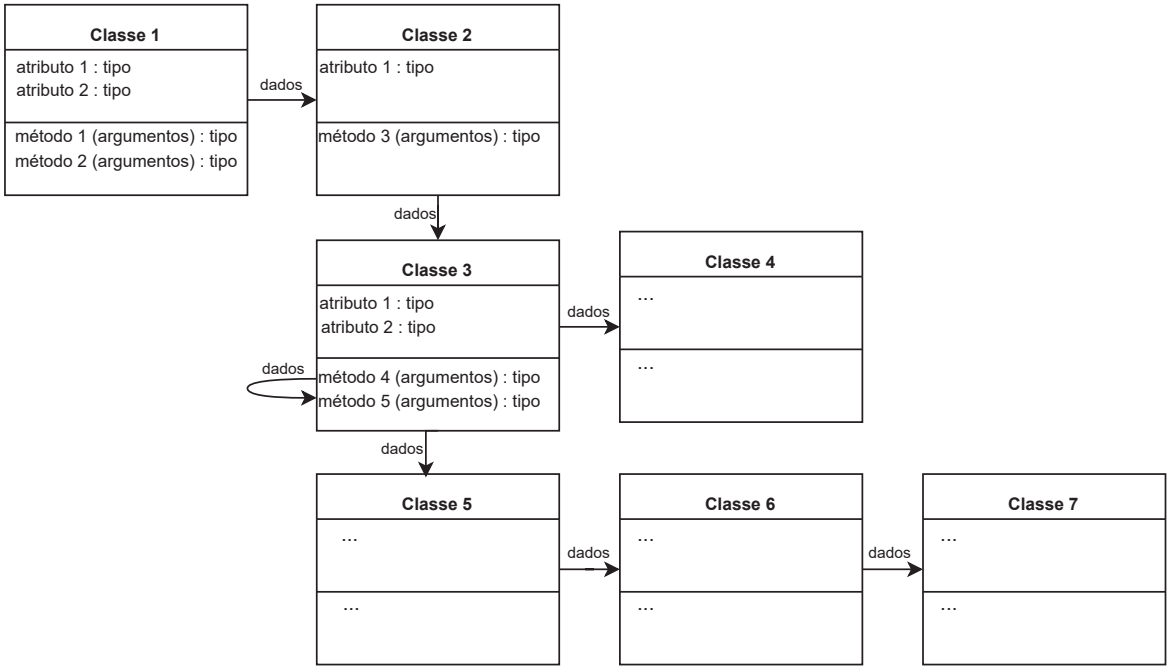


Figura 2.5: Estrutura de um sistema orientado a objetos.

As características e construções do paradigma OO favorecem a manutenção do software, o reuso de software e o desenvolvimento de componentes (Delamaro et al., 2016). Apesar dessas vantagens, tais características da OO acarretam novas fontes de falhas, de modo que o teste de software continue fundamental nesse contexto de desenvolvimento. Ainda, essas possibilidades inerentes aos sistemas OO refletem em mudanças nas definições das fases de teste em comparação ao teste procedimental.

Na literatura, não há um consenso sobre o que deve ser considerado como a menor unidade para o teste de integração OO. Alguns autores indicam os métodos como as menores

unidades (Harrold e Rothermel, 1994). Em contrapartida, também existe o entendimento de que as classes são as unidades em teste (Binder, 2000). Essas possibilidades acarretam em diferenças na organização das fases de teste de unidade e integração (Delamaro et al., 2016):

- **Método é a unidade em teste**:
  - **Teste de unidade**: fase para teste individual de um método, também chamada de teste intramétodo;
  - **Teste de integração**: fase para teste da interação entre métodos em uma mesma classe (**teste intermétodos**), entre métodos públicos em uma mesma classe (teste intraclasse) e entre métodos públicos de classes distintas (**teste interclasses**).
- **Classe é a unidade em teste**:
  - **Teste de unidade**: fase para teste individual de uma classe, englobando os testes que ocorrem no escopo de uma única classe (testes intramétodo, intermétodos e intraclasse);
  - **Teste de integração**: fase para teste da interação entre classes distintas (teste interclasses).

A realização do teste de sistema e do teste de aceitação para o contexto OO não possui alterações em relação aos testes de sistemas procedimentais. Assim, essas fases finais de teste são executadas em uma versão completa do SUT. A partir dos conceitos das Seções 2.1.4 e 2.2 e da categorização apresentada nesta seção, a Figura 2.6 apresenta uma comparação entre as fases de teste procedimental e de teste OO considerando métodos como as unidades.

2.4 PLANEJAMENTO EM IA

O planejamento em IA pode ser definido informalmente como a escolha e a organização de ações sequenciais que alteram o estado atual e atingem os objetivos do problema em outro estado desejado. Assim, um modelo conceitual do planejamento em IA requer a representação de um modelo de **sistema de estado-transição** formalmente definido como uma tripla $\Sigma = (S, A, \gamma)$, onde (Ghallab et al., 2004):

- $S = \{s_1, s_2, s_3, ...\}$ é um conjunto finito de **estados**;
- $A = \{a_1, a_2, a_3, ...\}$ é um conjunto finito de **ações** que mudam o estado;
- $\gamma : S \times A \rightarrow S$ é uma **função de transição de estados** que indica qual estado deve ser produzido quando uma ação a é executada em um estado s.

O **planejamento em IA clássico** considera as seguintes suposições sobre um sistema de estado-transição Σ (Ghallab et al., 2004):

- **Finito**: Σ contém um número finito de estados;
- **Totalmente observável**: é disponível o conhecimento completo sobre qualquer estado do Σ;
- **Determinístico**: cada ação aplicada a um estado leva a um único estado, ou seja, para cada estado s e cada ação a do Σ, $|\gamma(s, a)| \leq 1$.

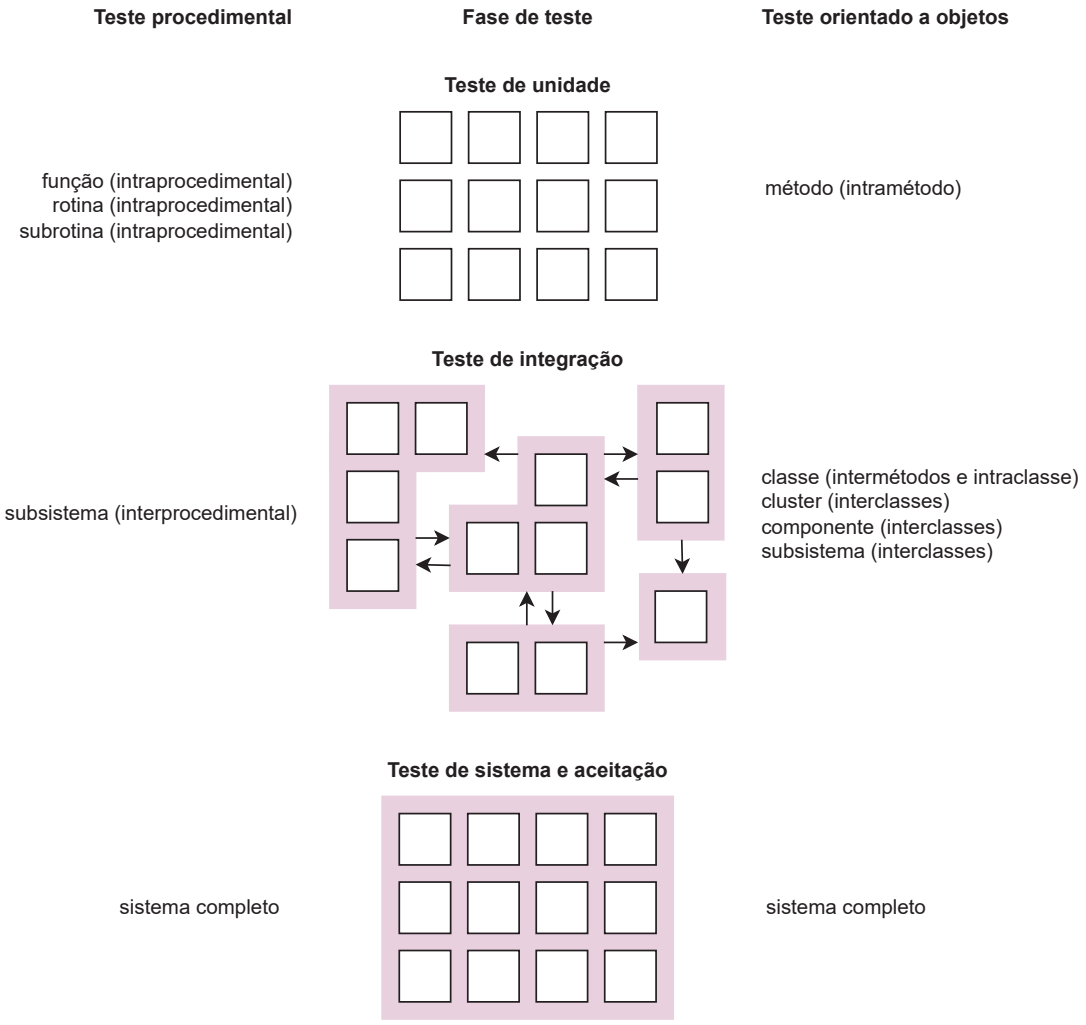


Figura 2.6: Relação entre fases de teste procedimental e OO. Fonte: Adaptado de Vincenzi (2004).

- **Estático**: cada estado do Σ permanece inalterado até que uma ação seja aplicada no estado.
- **Objetivos restritos**: o planejador gera soluções somente a partir do que estiver especificado e explícito no estado objetivo;
- **Planos sequenciais**: um plano de IA para um problema de planejamento é uma sequência finita e linearmente ordenada de ações;
- **Tempo implícito**: as ações ocorrem instantaneamente e sem tempo de duração; e
- **Planejamento *offline***: o planejador não considera mudanças que podem ocorrer no Σ durante a geração da solução.

Semanticamente, um **problema de planejamento em IA clássico** é uma tripla $P_{iac} = (\Sigma, s_0, g)$, onde (Ghallab et al., 2004):

- $\Sigma = (S, A, \gamma)$ é um **sistema de estado-transição**;
- $s_0 \in S$ é o **estado inicial** do problema; e

- g é um conjunto de proposições que fornecem os requisitos para que um estado seja o estado final (objetivo) do problema. Logo, o **estado final** é $s_g = \{s_g \in S \mid g \subseteq s_g\}$.

A partir de uma especificação formal de Σ, s_0 e s_g, um **planejador** é um algoritmo ou sistema de software que gera uma sequência de ações que atingem o estado final a partir do estado inicial. Essa sequência de ações que soluciona o problema especificado é denominado **plano de IA**. Considerando um sistema de estado-transição determinístico, um plano de IA é uma sequência finita de ações $\pi = (a_1, ..., a_k) \in A, k \geq 0$, associada a um conjunto de estados $(s_1, s_2, ..., s_n) \in S$, tal que $s_1 = \gamma(s_0, a_1)$, $s_2 = \gamma(s_1, a_2)$, ..., $s_n = \gamma(s_{n-1}, a_n)$ e $s_n \in g$.

As primeiras formalizações e resoluções computacionais de problemas de planejamento em IA clássico ocorreram com a linguagem e o planejador *Stanford Research Institute Problem Solver* (STRIPS) de Fikes e Nilsson (1971). A partir da década de 80, surgem novas linguagens como extensões da STRIPS contendo características que aumentam a expressividade para representar problemas mais complexos. Essas extensões, que introduzem elementos adicionais às especificações, são caracterizadas como planejamento em IA não-clássico.

A Tabela 2.1 apresenta uma visão geral da evolução das principais extensões do planejamento em IA. Além da década de surgimento e uma breve descrição das características, são indicados exemplos de linguagens e planejadores associados a cada tipo de extensão.

Tabela 2.1: Visão geral das extensões do planejamento em IA. Adaptado de Ghallab et al. (2004) e Russell e Norvig (2022).

Extensão	Década	Descrição	Linguagens	Planejadores
Planejamento clássico	1970	Caracterizado por estados finitos e ações determinísticas	**STRIPS** (Fikes e Nilsson, 1971), **Linguagem de planejamento adaptativo** (Anderson e Fickas, 1989), **PDDL1.2** (McDermott et al., 1998)	**STRIPS** (Fikes e Nilsson, 1971), **Planejador hierárquico** (Anderson e Fickas, 1989) ***Fast-Foward*** **(FF)** (Hoffmann e Nebel, 2001)
Planejamento com quantificadores e expressividade lógica	1980	Caracterizado pelo uso de quantificadores como "para todos" (*forall*) e "existe pelo menos um" (*exists*) e por ações mais expressivas com disjunções e negações	**ADL** (Pednault, 1989), **PDDL1.2** (McDermott et al., 1998)	**UCPOP** (Penberthy e Weld, 1993), ***Interference Progression Planner*** **(IPP)** (Koehler et al., 1997), **FF** (Hoffmann e Nebel, 2001), **LAMA** (Richter e Westphal, 2008), **MF-IPP** (Li et al., 2009), **JavaGP** (Meneguzzi, 2010)
Planejamento com ações condicionais	1980	Caracterizado por ações com efeitos que dependem do estado atual e pela representação de escolhas ou alternativas no fluxo de ações	**ADL** (Pednault, 1989), **PDDL1.2** (McDermott et al., 1998), **PDDL2.1** (Fox e Long, 2003), **PDDL2.2** (Edelkamp e Hoffmann, 2004), **PDDL3.0** (Gerevini e Long, 2005), **PDDL3.1** (Helmert, 2008)	**UCPOP** (Penberthy e Weld, 1993), **IPP** (Koehler et al., 1997), **FF** (Hoffmann e Nebel, 2001), **Metric-FF** (Hoffmann, 2003b), **LAMA** (Richter e Westphal, 2008), **MF-IPP** (Li et al., 2009)
Planejamento com recursos	1990	Caracterizado pela representação de valorações numéricas como recursos limitados (como tempo, energia e materiais) consumidos ou produzidos por ações	**PDDL2.1** (Fox e Long, 2003)	**Metric-FF** (Hoffmann, 2003b), **SGPlan** (Hsu et al., 2006)
Planejamento com tempo e restrições temporais	1990	Caracterizado por ações durativas (ações com início, duração e fim) e por restrições temporais entre ações	**PDDL2.1** (Fox e Long, 2003)	**SAPA** (Do e Kambhampati, 2003), **CRIKEY** (Halsey et al., 2004)
Planejamento não determinístico	2000	Caracterizado por ações com resultados probabilísticos e por lidar com incertezas no ambiente	**extNADL** (Jensen, 2003), **Probabilistic PDDL** (Younes e Littman, 2004)	**Bifrost** (Jensen, 2003)

2.4.1 Planejamento em IA não-clássico

O planejamento em IA não-clássico surgiu para cobrir limitações do planejamento em IA clássico, como a falta de tipificação em elementos da representação e a impossibilidade de

representar valorações numéricas. Considerando as extensões da Tabela 2.1, as representações desta tese utilizam quantificadores, expressividade lógica, ações condicionais e recursos.

Com base nas definições de planejamento clássico de Ghallab et al. (2004), nas definições de planejamento em IA de Russell e Norvig (2022) e em linguagens que suportam a representação dessas extensões (McDermott et al., 1998), a representação formal de um **problema de planejamento em IA não-clássico** delimitada para esta tese consiste em uma tupla $\mathrm{P}_{\mathrm{ianc}} = (\Sigma, s_0, s_g, O, P, F)$, onde:

- $\Sigma = (S, A, \gamma)$ é um **sistema de estado-transição**;
- $s_0 \in S$ é o **estado inicial** do problema; e
- s_g é o **estado final** do problema;
- $O = \{o_1, o_2, o_3...\}$ é um conjunto finito de **objetos** contantes que representam as entidades do problema;
- $P = \{p_1, p_2, p_3...\}$ é um conjunto finito de **predicados** usados na representação das propriedades e características das entidades do problema;
- $F = \{f_1, f_2, f_3, ...\}$ é um conjunto finito de **funções** que representam recursos como variáveis numéricas.

Os predicados de P são representados em notação de lógica de primeira ordem e são instanciados com os objetos de O. Os estados de S são conjuntos de predicados instanciados e funções de F que representam os fatos de uma determinada configuração do ambiente modelado. Os predicados instanciados nos estados são considerados verdadeiros. Uma ação de A é uma tripla $a = (nome(a), pre(a), pos(a))$, onde:

- $nome(a)$ é uma expressão sintática que descreve o nome único da ação a;
- $pre(a)$ é uma expressão lógica de predicados que definem as pré-condições de a; e
- $pos(a)$ é uma expressão lógica de predicados que definem os efeitos de a;

Se os predicados contidos em um estado $s \in S$ tornam $pre(a)$ verdadeira, então a é executada em s. A execução de uma ação a ocasiona a transição para um outro estado s', conforme indicado por γ. Denota-se a transição entre dois estados s, s', a partir da execução de uma ação a, por $s \xrightarrow{a} s'$. Os efeitos $pos(a)$ da ação a são conjuntos de predicados positivos ou negativos (negados). O estado s', alcançado com a execução de a, é obtido com a inclusão de efeitos positivos e a remoção dos efeitos negados de s.

Além das representações *booleanas* (verdadeiras e falsas) dos predicados, representações numéricas podem ser incluídas nas ações e nos estados utilizando as funções de F e operadores aritméticos e relacionais básicos. Um plano de IA π para um problema de planejamento em IA não-clássico pode ser denotado por $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \xrightarrow{a_3} \ldots \xrightarrow{a_k} s_g$.

A potencialidade de representação de conhecimento fornecida pelas definições do planejamento em IA clássico é formalizada em modelos descritos em linguagens de planejamento em IA. A formalização com essas linguagens resulta em representações que podem ser usadas por planejadores para a construção automática de planos de IA. A Subseção 2.4.2 descreve a formalização estabelecida pela linguagem de planejamento usada nos modelos desta tese. O funcionamento dos planejadores é descrito na Subseção 2.4.3.

2.4.2 *Planning Domain Definition Language* (PDDL)

Como contextualizado no Capítulo 1, os primeiros estudos relacionados ao planejamento em IA datam das décadas de 70 e 80, com o desenvolvimento das linguagens *Stanford Research Institute Problem Solver* (STRIPS) (Fikes e Nilsson, 1971) e *Action Description Language* (ADL) (Pednault, 1989). A *Planning Domain Definition Language* (PDDL) é uma linguagem consolidada de planejamento em IA não-clássico que deriva de formalismos de STRIPS e ADL (McDermott et al., 1998).

A sintaxe da PDDL foi desenvolvida inicialmente para competições[5] de planejamento em IA realizadas na principal conferência[6] da área. Desde então, a PDDL é usada como linguagem-padrão nessas competições principalmente em comparações de desempenho entre planejadores (Russell e Norvig, 2022; McDermott et al., 1998). A partir disso, PDDL vem sendo aprimorada em versões com novas características que permitem a representação de uma maior variedade de problemas. As versões existentes da PDDL são:

- **PDDL 1.2**: essa versão foi elaborada por McDermott et al. (1998) para a competição IPC-1 realizada na conferência AIPS-98. As características dessa versão incluem *ações baseadas em STRIPS*, *efeitos condicionais*, *quantificadores universais* e *especificação de restrições*.

- **PDDL 2.1**: versão desenvolvida por Fox e Long (2003) para a IPC-3 da AIPS-02. Essa versão introduziu características que permitem especificar problemas que envolvem *tempo* e *números*. O tempo é modelado em ações durativas, entendidas como ações que possuem duração. Essas ações contém as mesmas características das ações da versão 1.2, mas incluem um parâmetro que modela o tempo de duração de uma ação.

  Os números são modelados em variáveis numéricas que podem ser usadas em condições e efeitos. Esta versão também introduziu a noção de métrica em PDDL, permitindo minimizar ou maximizar o valor de uma variável numérica.

- **PDDL 2.2**: versão elaborada por Edelkamp e Hoffmann (2004) para a IPC-4 que ocorreu durante a ICAPS-04. Esta versão estende a sintaxe da versão 2.1 com a inclusão de *predicados derivados* e *literais iniciais temporizados*. Predicados derivados são predicados estabelecidos a partir de regras de outros predicados modelados.

  Literais iniciais temporizados podem ser entendidos como eventos exógenos e determinísticos. Esses literais tornam um predicado verdadeiro ou falso em pontos de tempo conhecidos pelo planejador com antecedência.

- **PDDL 3.0**: versão elaborada por Gerevini e Long (2005) no contexto da IPC-5 da ICAPS-06. Essa versão inclui características que permitem expressar *restrições na trajetória dos planos*. Essas restrições atuam em possíveis ações e estados intermediários alcançados pelo plano de IA.

  Nessa versão, as restrições e os objetivos podem ser *fortes* ou *fracos*. Restrições e objetivos fortes devem ser respeitados pelo planejador na geração de qualquer plano de IA, enquanto os fracos não precisam ser necessariamente alcançados.

[5] *International Planning Competition* (IPC). Informações sobre as edições da IPC estão disponíveis em: `https://www.icaps-conference.org/competitions`.

[6] As competições de planejamento em IA acontecem na *International Conference on Autonomous Planning and Scheduling* (ICAPS). A ICAPS é uma junção das conferências *Artificial Intelligence Planning Systems* (AIPS) e *European Conference on Planning* (ECP). Informações sobre a ICAPS estão disponíveis em `https://www.icaps-conference.org`.

- **PDDL 3.1**: essa versão não foi publicada em artigo acadêmico e está descrita apenas na página Web da IPC-6 da conferência ICAPS-08 (Helmert, 2008). As versões anteriores permitem a declaração de variáveis booleanas e numérias. A versão 3.1 introduz *variáveis de objetos* que permitem representar outros objetos. Essa versão também adiciona um requisito que permite a definição de um *custo da ação*. A definição desse custo desabilita o uso de outras valorações numéricas na mesma representação.

A criação de linguagens de representação mais expressivas requer o desenvolvimento de novas ferramentas de resolução. Com o aumento da expressividade da linguagem, se torna mais difícil desenvolver uma ferramenta adequada às novas características e geralmente o tempo de resposta do algoritmo resultante aumenta proporcionalmente. Esta tese utiliza a versão PDDL 2.1[7], cuja expressividade é suficiente para as representações propostas. Além disso, os planejadores existentes para esta versão apresentam desempenho[8] satisfatório.

As representações de problemas com planejamento em IA em PDDL são especificadas por um modelo composto de um domínio e um problema. O **domínio** contém o conhecimento de um determinado problema a ser resolvido e o **problema** compreende uma instância com definições válidas para um problema específico (McDermott et al., 1998).

As representações em PDDL são feitas em arquivos distintos (*dominio.pddl* e *problema.pddl*), permitindo a associação de diferentes problemas a um mesmo domínio. O conhecimento é descrito no arquivo do domínio pelos conjuntos A, P e F. Além disso, o domínio contém elementos da PDDL, como o nome do domínio, requisitos da linguagem, tipos e a declaração opcional de constantes. Os requisitos são características contidas na PDDL para a representação do domínio e tipos são usados para tipificar objetos e contantes.

Cada ação do domínio é parametrizada por objetos de O. As funções de F podem ser associadas às ações representando restrições ou efeitos numéricos. O uso de uma função como restrição numérica ocorre em $pre(a)$ a partir de comparações entre valores de variáveis e constantes. O uso de uma função como efeito numérico possibilita operações aritméticas entre variáveis e constantes em $pos(a)$.

A representação da instância do problema no arquivo *problema.pddl* é caracterizada pelos estados s_0 e s_g. Além da declaração dos estados inicial e final, este arquivo contém o nome do problema, o nome do domínio ao qual o problema está associado e o conjunto de objetos O tipificados com os tipos declarados no domínio. Opcionalmente, as funções declaradas no domínio podem ser utilizadas como métricas em s_g para a geração do plano de IA.

Os Códigos 2.1 e 2.2 exemplificam, respectivamente, representações de um domínio e de um problema em PDDL. Essas representações geram planos de IA que indicam uma sequência para a integração e o teste de componentes de um SUT seguindo uma ordem pré-estabelecida.

O domínio contém os requisitos `adl`[9] e `fluents` (linha 3, Código 2.1). O requisito `(adl` permite a definição de tipos, checagem de igualdade, declaração de efeitos condicionais e declaração de pré-condições com quantificadores e disjunções. O requisito `fluents` permite a representação de valorações numéricas. Esse domínio introduz o tipo `component` (linha 5, Código 2.1) que representa os elementos individuais do SUT a serem integrados e testados.

O domínio contém a declaração de dois predicados utilizados para monitorar o estado de integração e teste de cada componente. O predicado `integrated ?C`[10] (linha 7, Código

[7] No decorrer deste texto, o termo PDDL se refere à versão PDDL 2.1.

[8] Boddy et al. (2005) e Pereira et al. (2022) mencionam o desempenho de tempo de execução do plano de IA com planejadores para representações em PDDL.

[9] Os termos iniciados por "**:**" são rótulos reservados da linguagem PDDL.

[10] Os termos iniciados por "**?**" são variáveis.

2.1) indica se o componente *C* já foi integrado, enquanto o predicado `tested ?C` (linha 8, Código 2.1) indica se o componente componente *C* já foi testado.

Código 2.1: Representação em PDDL do domínio do teste de integração. Fonte: Retirado de Silva e Lemos (2011).

```
1  (define (domain IntegrationTestingDomain)
2
3    (:requirements :adl :fluents)
4
5    (:types component)
6
7    (:predicates (integrated ?C - component)
8                 (tested ?C - component))
9
10   (:functions (testing_step)
11               (integration_step)
12               (integration_order ?C - component))
13
14   (:action IntegrateComponent
15     :parameters (?C - component)
16
17     :precondition (and
18                     (not (integrated ?C))
19                     (not (tested ?C))
20                     (= (integration_step) (integration_order ?C))
21                     (= (integration_step) (testing_step)))
22
23     :effect (and
24               (integrated ?C)
25               (increase (integration_step) 1))
26   )
27
28   (:action TestComponent
29     :parameters (?C - component)
30
31     :precondition (and
32                     (not (tested ?C))
33                     (integrated ?C)
34                     (= (testing_step) (integration_order ?C)))
35
36     :effect (and
37               (tested ?C)
38               (increase (testing_step) 1))
39   )
40 )
```

As funções são definidas no domínio para acompanhar o progresso da integração. As funções `testing_step` e `integration_step` (linhas 10 e 11, Código 2.1) representam os passos atuais do teste e da integração, respectivamente. A função `integration_order ?C` (linha 12 Código 2.1) especifica a ordem de integração de cada componente.

O domínio contém duas ações que refletem os processos de integração e teste. A ação `IntegrateComponent` (linhas 14 a 26, Código 2.1) indica a realização da integração de um componente. A ação `TestComponent` (linhas 28 a 39, Código 2.1) indica a execução do teste de um componente após sua integração.

O Código 2.2 apresenta a representação do problema PDDL para uma instância do problema do teste de integração de Silva e Lemos (2011). Essa instância contém cinco componentes representados por objetos (linha 5). O estado inicial (`init`, linhas 7 a 15)

estabelece o contexto inicial em que os passos atuais de integração e de teste são configurados como 1. Além disso, a ordem de integração de cada componente é especificada. O estado final (`goal`, linhas 17 a 25) é alcançado após o teste de todos os componentes.

Código 2.2: Representação em PDDL do problema do teste de integração. Fonte: Retirado de Silva e Lemos (2011).

```
1  (define (problem IntegrationTestingProblem)
2
3    (:domain IntegrationTestingDomain)
4
5    (:objects C1 C2 C3 C4 C5 - component)
6
7    (:init
8      (= (integration_step) 1)
9      (= (testing_step) 1)
10     (= (integration_order C1) 5)
11     (= (integration_order C2) 4)
12     (= (integration_order C3) 2)
13     (= (integration_order C4) 3)
14     (= (integration_order C5) 1)
15   )
16
17   (:goal (and
18            (tested C1)
19            (tested C2)
20            (tested C3)
21            (tested C4)
22            (tested C5))
23   )
24 )
```

2.4.3 Planejadores

Os planejadores são algoritmos ou ferramentas que encontram planos de IA automaticamente. A entrada do planejador é uma especificação formal de um problema em alguma linguagem de planejamento em IA. A partir dessa entrada, o planejador constroi um grafo de planejamento (GPL)[11]. Na sequência, o planejador usa uma estratégia de busca, como busca heurística, busca em grafo ou métodos baseados em satisfatibilidade lógica, no GPL[12] para encontrar um dos possíveis planos de IA para o problema.

A Figura 2.7 ilustra a arquitetura envolvida na geração de um plano de IA a partir da representação de um problema em PDDL. Os planejadores são desenvolvidos para atender especificidades de uma determinada linguagem de planejamento em IA. O Metric-FF (Hoffmann, 2003b) é um exemplo de planejador para problemas modelados em PDDL. O Metric-FF é um planejador determinístico, gerando um único plano de IA em cada execução.

O Metric-FF gera planos de IA de acordo com a heurística de busca indicada nos parâmetros de execução. Além da heurística, o Metric-FF pode utilizar as variáveis numéricas como métricas para encontrar uma solução. Assim, os planos de IA encontrados pelo Metric-FF

[11] O conjunto de estados S também é chamado de espaço de estados. Um GPL é uma possível representação de um espaço de estados.

[12] O GPL é um grafo orientado $G_{pl} = (V, Ar)$ onde cada vértice $v \in V$ representa um estado do problema e cada arco $ar \in Ar$, rotulado com uma ação de A, representa uma transição entre os estados. Caminho é uma sequência finita de vértices $(v_1, v_2, ..., v_k), k \geq 2$, tal que existe um arco de v_i para v_{i+1}, $i = 1, 2, ..., k - 1$. representa um plano de IA (Ghallab et al., 2004).

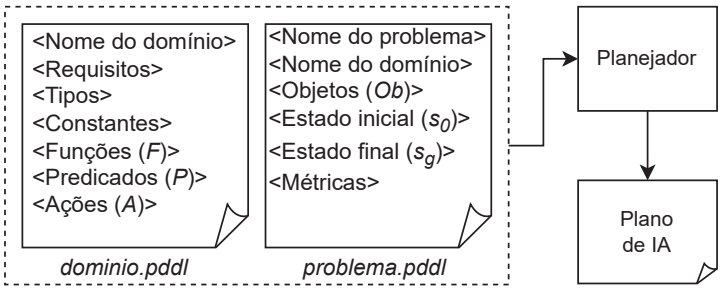


Figura 2.7: Geração de planos de IA a partir de representações em PDDL.

atendem aos critérios definidos pelo usuário na execução do planejador e na representação do problema e não são necessariamente os menores possíveis.

Em sua configuração padrão, o Metric-FF usa uma variação do algoritmo Subida de Encosta (do inglês, *Hill-Climbing*) para as buscas no grafo de planejamento e o algoritmo de planejamento *Graphplan* como heurística de avaliação[13]. Estudos indicam que o Metric-FF apresenta melhor desempenho em termos de tempo de execução quando comparado a outros planejadores para problemas em PDDL (Boddy et al., 2005; Pereira et al., 2022).

O Código 2.3 mostra o plano de IA encontrado pelo Metric-FF para a instância do problema do teste de integração apresentado na Subseção 2.4.2. O plano de IA é composto por 10 ações que indicam a sequência de integração e teste dos componentes. Além do plano de IA, a execução do planejador também indica o tempo gasto, em segundos, para a instanciação do problema, a criação do grafo de planejamento e as buscas pela solução.

Código 2.3: Plano de IA para o problema do teste de integração de Silva e Lemos (2011).

```
0: INTEGRATECOMPONENT C5
1: TESTCOMPONENT C5
2: INTEGRATECOMPONENT C3
3: TESTCOMPONENT C3
4: INTEGRATECOMPONENT C4
5: TESTCOMPONENT C4
6: INTEGRATECOMPONENT C2
7: TESTCOMPONENT C2
8: INTEGRATECOMPONENT C1
9: TESTCOMPONENT C1
```

2.5 CONSIDERAÇÕES DO CAPÍTULO

Conforme apresentado na Seção 2.1, a conceitualização de teste de software que orienta esta tese baseia-se nas definições clássicas de Myers (1979) e considera que:

- **o teste de software é conduzido em um ambiente controlado**: os processos para a execução do teste são previamente planejados de acordo com as limitações e especificidades do SUT e do ambiente. Tais processos são posteriormente detalhados em um plano de teste, conforme indicado na Subseção 2.1.1;

- **o teste de software é executado em uma determinada arquitetura**: o teste é apoiado por uma infraestrutura de software e hardware e conduzido em atividades que utilizam e

[13]Detalhes sobre os princípios algorítmicos e o código-fonte do Metric-FF estão disponíveis em `https://fai.cs.uni-saarland.de/hoffmann/metric-ff.html`.

geram artefatos variados. Esses aspectos são estabelecidos conforme as características do SUT, do tipo de teste e da escala do projeto; e

- **o teste de software é realizado de maneira limitada**: é infactível executar exaustivamente um teste com todo seu o domínio de dados de entrada. Assim, o teste é executado com um conjunto de dados de teste selecionado a partir do critério de teste escolhido.

Visando estabelecer os conceitos de teste considerados nesta tese, a Seção 2.1 abordou as atividades de teste, um cenário típico de execução do teste, as técnicas de teste e as fases de teste. O Capítulo 4 apresenta a abordagem que considera o artefato de plano de teste gerado na atividade de planejamento do teste e explora o uso desse artefato na atividade de execução do teste. Essa abordagem considera a técnica de teste funcional e a fase de teste de integração.

A Seção 2.1 estruturou as fases de desenvolvimento e as fases de teste no Modelo V. Nesse contexto, a integração é entendida como um processo intermediário que é realizado em cada *sprint*. Para contextualizar a integração no gerenciamento de software, esta tese posiciona o plano de integração como parte da estrutura contida na abordagem do Capítulo 4.

A integração de unidades é orientada pelo entendimento do conceito de unidade estabelecido em cada paradigma de programação. Nesse contexto, as Seções 2.2 e 2.3 descreveram a organização das fases do teste procedimental e do teste OO, respectivamente. A abordagem do Capítulo 4 considera procedimentos e métodos como unidades, usando teste interprocedimental, intermétodos e interclasses.

O uso do planejamento em IA, descrito na Seção 2.4, visa atender os aspectos mencionados na conceitualização de teste estabelecida nesta tese. Com relação ao ambiente, o planejamento em IA pode gerar soluções baseadas na representação do conhecimento do SUT e do ambiente de teste. Quanto à arquitetura, o planejamento em IA oferece ferramentas que podem integrar a infraestrutura do teste. No que diz respeito à realização do teste, o planejamento em IA pode contribuir na indicação criteriosa e automática de casos de teste.

A resolução de problemas representados com o planejamento em IA não-clássico, apresentado na Subseção 2.4.1, possui maior complexidade computacional em comparação ao planejamento em IA clássico. No entanto, entende-se que o uso das valorações numéricas é necessário para uma representação mais expressiva do teste. Assim, a abordagem do Capítulo 4 é associada a uma linguagem e a um planejador de planejamento em IA não-clássico. Ainda, optou-se por um planejador determinístimo para gerar um único plano para o SUT e não aumentar a complexidade do processo.

A Subseção 2.4.2 exemplificou uma representação utilizando a linguagem PDDL a partir de uma aplicação do planejamento em IA na ES encontrada na literatura. O exemplo foi definido para explorar a sintaxe e algumas características da PDDL, como a declaração de valorações numéricas e seu uso em pré-condições e efeitos. O exemplo reforçou a ideia de que as características do planejamento em IA não-clássico são fundamentais para a completude da representação de cenários de teste.

A Subseção 2.4.3 detalhou o funcionamento dos planejadores e destacou o planejador Metric-FF, utilizado para gerar um plano de IA para o problema apresentado na Subseção 2.4.2. As execuções do Metric-FF permitiram identificar a configuração necessária para a compilação e utilização do planejador, além de proporcionar aprendizado sobre sua parametrização. Embora o exemplo seja uma instância pequena, observou-se que o planejador apresentou bom desempenho na geração do plano de IA, o que motivou o uso desse planejador nesta tese.

Os conceitos deste capítulo destacaram os artefatos mencionados nesta tese. O **plano de teste** é um artefato construído pela equipe de teste para auxiliar o gerenciamento do teste. O plano de teste especifica os precedimentos para o teste de integração em um **plano de integração**.

O **plano de IA** é uma solução para um problema de planejamento em IA gerada pela execução de um planejador. O plano de IA gerado pela abordagem do Capítulo 4 tem como objetivo complementar o plano de integração e pode ser entendido como um **plano de IA de integração**.

Para investigar o estado da arte do uso do planejamento em IA no teste de software, posicionar esta tese em relação às soluções existentes, identificar trabalhos relacionados e encontrar lacunas de pesquisa, foi conduzida uma revisão bibliográfica da literatura. O próximo capítulo apresenta os principais resultados dessa revisão.

3 REVISÃO BIBLIOGRÁFICA

Visando obter um panorama sobre o uso do planejamento em IA no teste software, foi realizada uma revisão bibliográfica que investigou 35 estudos. A revisão foi conduzida durante o primeiro semestre de 2022 e posteriormente revisada no último trimestre de 2024. O Apêndice A detalha a metodologia usada na revisão e os dados extraídos nos estudos. O Apêndice B resume as propostas dos 35 estudos identificados. Este capítulo apresenta os principais resultados da revisão (Seção 3.1), descreve estudos identificados no contexto de teste de integração (Seção 3.2) e analisa esses estudos entendidos como trabalhos relacionados à tese (Seção 3.3).

3.1 TESTE DE SOFTWARE E PLANEJAMENTO EM IA

A revisão bibliográfica da literatura foi conduzida para responder a questão de pesquisa "*Como o planejamento em IA tem apoiado o teste de software?*". Para isso, os 35 estudos encontrados na revisão foram analisados a partir de oito aspectos: sistemas sob teste, artefatos, atividades de teste, fases de teste, técnicas de teste, linguagens de planejamento, planejadores e planos de IA. Os dados apresentados nesta seção se referem à análise desses aspectos nos 35 estudos identificados. A Seção A.2 detalha os dados quantitativos da revisão.

Anderson e Fickas (1989) foi o primeiro estudo selecionado na revisão que associa o planejamento em IA e o teste de software. Desde então, com o aprimoramento das linguagens e dos planejadores, o uso do planejamento em IA foi ampliado para vários contextos de teste. Com os resultados da revisão, foi possível notar um interesse constante de pesquisa nessa área ao longo dos últimos 35 anos. A Figura 3.1 indica o período de publicação desses estudos.

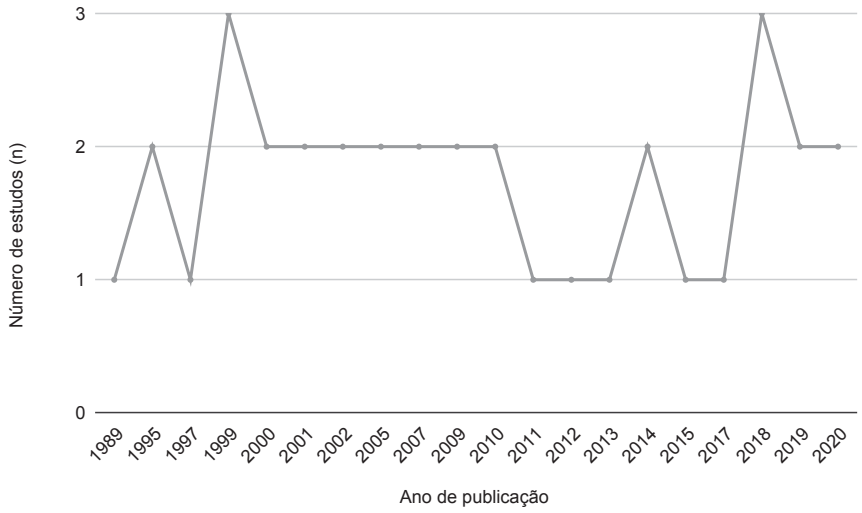


Figura 3.1: Período de publicação dos estudos analisados

A diversidade de características e funcionalidades observada nos sistemas sob teste usados pelos estudos ressalta a habilidade do planejamento em IA de representar o conhecimento de diferentes domínios. Em relação ao paradigma de programação, apenas dois estudos mencionaram explicitamente testes de sistemas OO. Isso sugere que as soluções estão direcionadas para testes de sistemas procedimentais. Por fim, a incidência significativa de SUTs em plataformas Web pode ser atribuída à popularidade dessa plataforma.

Os artefatos que apoiam a geração de planos de IA envolvem uma variedade de descrições textuais e gráficas dos SUTs. Essas descrições possuem semelhanças, como o detalhamento das associações existentes entre os elementos que serão testados. Essas associações são geralmente mapeadas como pré-condições e efeitos das ações. Como ilustra a Figura 3.2, os artefatos utilizados são provenientes principalmente das fases iniciais de desenvolvimento de software, como design de arquitetura e modelagem de requisitos.

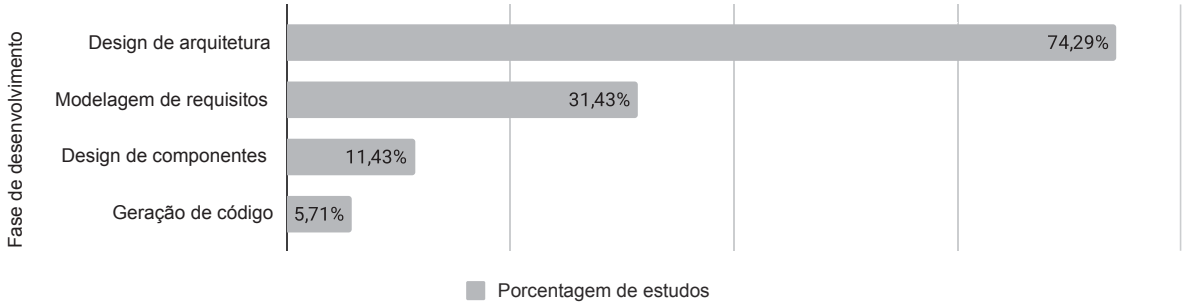


Figura 3.2: Fases de desenvolvimento de geração dos artefatos usados em estudos de teste de software com planejamento em IA.

A Figura 3.3 indica que o design de casos de teste é a principal atividade de teste apoiada pelo planejamento em IA. Entende-se que essa fase envolve a escolha de um conjunto abrangente e representativo de casos de teste e o uso do planejamento em IA pode oferecer uma forma de automatizar essa atividade. Ainda, ao explorar uma ampla variedade de cenários possíveis e identificar combinações de entradas que seriam difíceis de descobrir manualmente, o planejamento em IA pode contribuir para uma cobertura de teste mais abrangente.

Compreende-se que a aplicação do planejamento em IA na atividade de planejamento do teste visa representar as informações dos processos necessários para o teste para construir soluções com prioridades, restrições e métricas específicas. Dessa forma, é possível aproveitar essas informações e criar planos de IA otimizados que atendam aos requisitos e objetivos do teste. Por fim, a incidência na execução do teste pode estar relacionada com a possibilidade de concretizar as ações dos planos de IA em informações usadas para executar o teste.

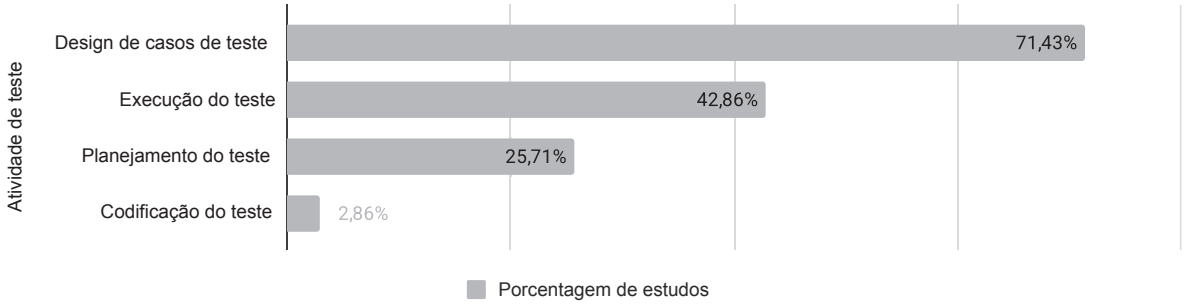


Figura 3.3: Atividades de teste nos estudos de teste de software com planejamento em IA

O destaque da fase de teste de sistema, ilustrado na Figura 3.4, pode ser justificado pela variedade de cenários e interações envolvidos nessa fase. Logo, o planejamento em IA pode contribuir para a tomada de decisão nesses ambientes complexos. Já o destaque no teste de integração pode ser associado com a adaptabilidade do planejamento em IA para ajustar as representações conforme necessário. As adaptações e ajustes podem tornar os testes mais flexíveis e capazes de responder às mudanças que ocorrem durante a integração.

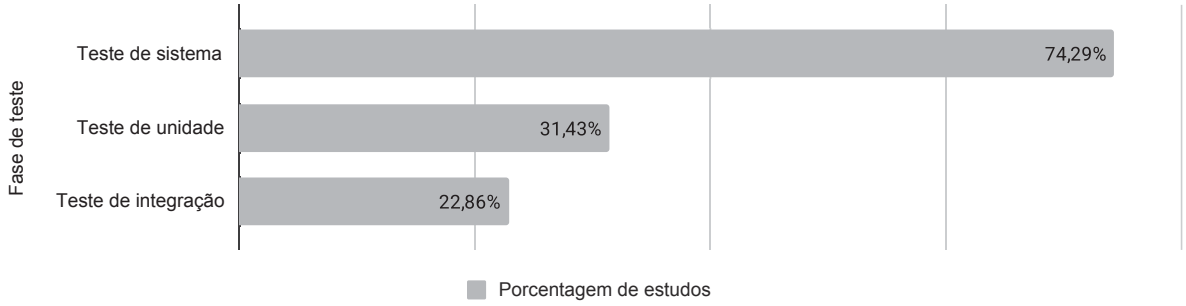


Figura 3.4: Fases de teste nos estudos de teste de software com planejamento em IA

Os resultados sobre as técnicas de teste apresentadas na Figura 3.5 destacam o teste funcional apoiado pelo planejamento em IA. Esse destaque é observado principalmente no contexto de teste de segurança. Compreende-se que o planejamento em IA pode contribuir na detecção de falhas e vulnerabilidades que seriam difíceis de identificar manualmente.

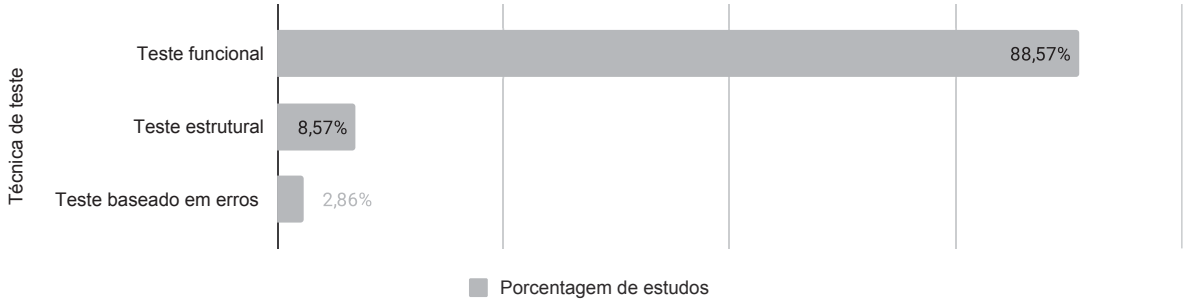


Figura 3.5: Técnicas de teste nos estudos de teste de software com planejamento em IA

O amplo uso da ADL e da PDDL indicado na Figura 3.6 mostra que as funcionalidades dessas linguagens são necessárias para representar todos os elementos associados aos testes. A versão mais utilizada da PDDL entre os estudos analisados é a PDDL2.1. Essa versão introduziu a capacidade de representar e realizar operações com valorações numéricas. Desse modo, essa característica pode ser entendida como fundamental para lidar com especificidades dos testes, como os critérios, as restrições e as métricas.

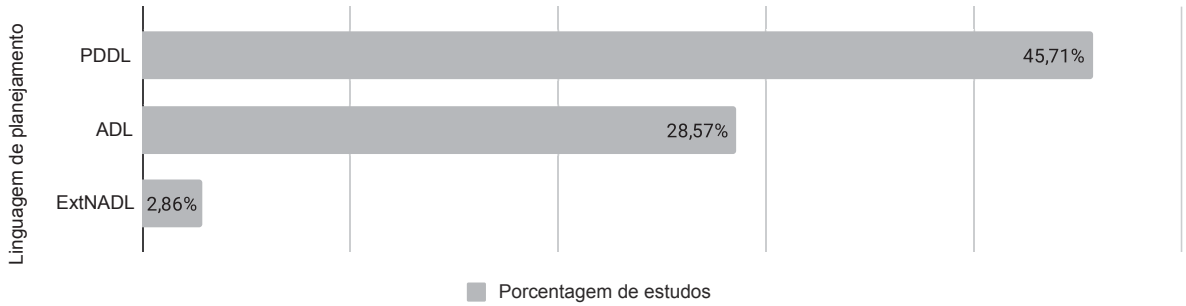


Figura 3.6: Linguagens de planejamento nos estudos de teste de software com planejamento em IA

Como apresentado na Figura 3.7, os planejadores UCPOP e o Metric-FF são amplamente usados pois suportam as principais linguagens de planejamento utilizadas nos estudos. Ainda, o código-fonte desses planejadores é disponibilizado integralmente nas páginas dos seus

desenvolvedores (Penberthy e Weld, 1993; Hoffmann, 2003a). Compreende-se que disponibilizar o código-fonte possibilita o uso das ferramentas por outros usuários e contribui para o estudo adicional e possível modificação do software em outras pesquisas.

Desempenho é outro fator relevante para a escolha do Metric-FF como o planejador preferível para problemas em PDDL. Estudos, como Boddy et al. (2005), indicam que o Metric-FF geralmente apresenta desempenho superior em termos de tempo de execução quando comparado a outros planejadores para PDDL. Considerando as extensões do planejamento em IA da Tabela 2.1, os resultados das Subseções A.2.6 e A.2.7 mostraram que as soluções usam principalmente quantificadores, operações lógicas, ações condicionais e recursos (valorações numéricas).

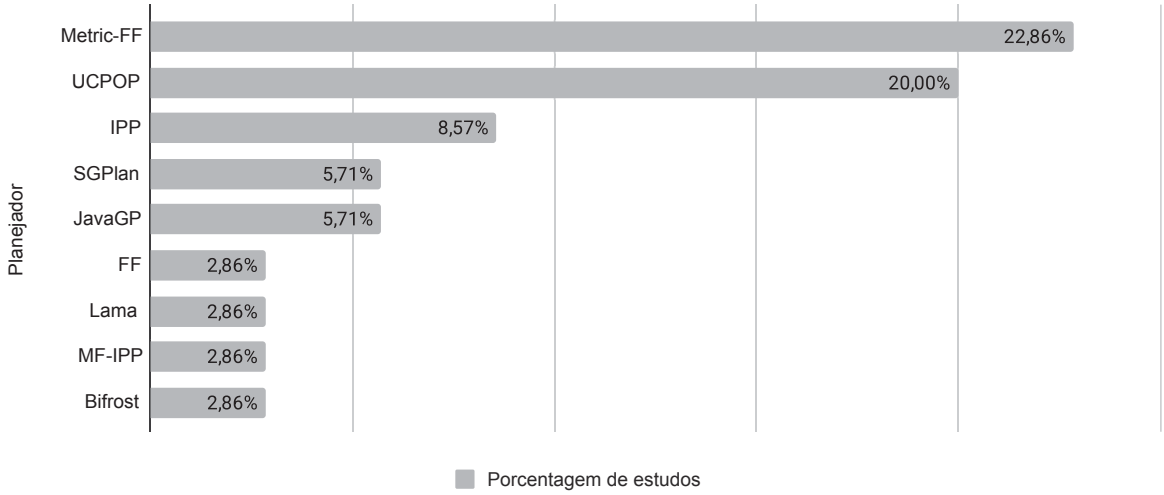


Figura 3.7: Planejadores nos estudos de teste de software com planejamento em IA

Os estudos analisados geralmente associam os planos de IA a comandos para o teste de uma funcionalidade do SUT. Assim, seguindo as definições da Seção 2.1, as ações podem ser mapeadas como dados de teste e não necessariamente ao par ordenado que constitui um caso de teste, como mencionado por alguns estudos. Acredita-se que os estudos simplificaram esse conceito para tornar as soluções mais abrangentes e genéricas.

Considerando as extensões de planejamento em IA da Tabela 2.1, as linguagens e os planejadores usados nos estudos indicam que as soluções de teste de software são apoiadas principalmente por quantificadores, expressividade lógica, ações condicionais e recursos (valorações numéricas). O estudo de Leitner e Bloem (2005) utiliza a extensão de planejamento não-determinístico.

3.1.1 Motivações, vantagens e desvantagens do uso do planejamento em IA

No decorrer da análise dos 35 estudos, foi realizado um levantamento de motivações, vantagens e desvantagens do uso do planejamento em IA no teste de software. Os relatos identificados nos estudos sobre esses aspectos foram codificados e agrupados por similaridade de conteúdo em três categorias (motivação, vantagem e desvantagem). A codificação e a categorização foram baseadas respectivamente nos processos de codificação aberta e codificação axial do método *Grounded Theory* (Corbin e Strauss, 2014).

A Tabela 3.1 apresenta a catalogação de motivações, vantagens e desvantagens da utilização do planejamento em IA no teste de software. Nota-se que a principal motivação

relatada é a possibilidade de *associação*[1] entre o problema de executar um teste de software e um problema de planejamento em IA. Essa associação ocorre pela similaridade da natureza desses problemas, sendo ambos com *ênfase em objetivo*.

Dessa forma, é possível utilizar uma *especificação* do problema referente a execução e ao objetivo do teste em termos de *operações sequenciais*. A definição dessa especificação viabiliza realizar a *representação* e a *modelagem* dos problemas em linguagens de planejamento. Somado a isso, as linguagens de planejamento em IA possibilitam a *formalização* de elementos definidos na especificação.

Tabela 3.1: Motivações, vantagens e desvantagens do planejamento em IA no teste de software.

	Código	**Referências**
Motivação	Formalização	Schnelte (2009)
	Modelagem	Razavi et al. (2014)
	Operações sequenciais	von Mayrhauser et al. (1999), Andrews et al. (2002)
	Representação	Schnelte (2009), Bozic et al. (2019)
	Especificação	Memon et al. (1999), Memon et al. (2000), Bozic e Wotawa (2018b)
	Ênfase em objetivo	Mraz et al. (1995), Scheetz et al. (1999), von Mayrhauser et al. (1999), Andrews et al. (2002)
	Associação	Anderson e Fickas (1989), Mraz et al. (1995), Howe et al. (1995), Howe et al. (1995), Bozic e Wotawa (2018a), Bozic et al. (2019), Lima (2020)
Vantagem	Abstração	Memon et al. (2001)
	Adaptação	Wotawa e Bozic (2014)
	Agregação	Scheetz et al. (1999)
	Expressividade	Razavi et al. (2014)
	Configurabilidade	Bozic et al. (2017), Bozic e Wotawa (2018b)
	Reuso de conhecimento	Wotawa e Bozic (2014), Bozic et al. (2017)
	Legibilidade	Boddy et al. (2005), Schnelte (2009)
	Uso de critérios/heurísticas	Schnelte (2009), Lima (2020)
	Extensibilidade	Bozic et al. (2017), Bozic e Wotawa (2018b), Lima (2020)
	Completude da descrição	Mraz et al. (1995), Howe et al. (1995), Howe et al. (1995)
	Corretude	Mraz et al. (1995), Howe et al. (1995), Howe et al. (1995)
	Disponibilidade	von Mayrhauser et al. (2000), Boddy et al. (2005), Lima (2020)
	Performance	Memon et al. (1999), Boddy et al. (2005), Obes et al. (2013), Bozic e Wotawa (2018b)
	Flexibilidade	Mraz et al. (1995), Howe et al. (1995), Howe et al. (1995), Scheetz et al. (1999), Andrews et al. (2002)
	Automatização	Mraz et al. (1995), Howe et al. (1995), Howe et al. (1995), von Mayrhauser et al. (1999), Bozic e Wotawa (2015), Bozic et al. (2017)
	Facilidade	Mraz et al. (1995), Howe et al. (1995), Howe et al. (1995), von Mayrhauser et al. (2000), Andrews et al. (2002), Wotawa e Bozic (2014), Bozic e Wotawa (2018b), Lima (2020)
	Representação	Mraz et al. (1995), Howe et al. (1995), Howe et al. (1995), Scheetz et al. (1999), Memon et al. (1999), von Mayrhauser et al. (2000), Memon et al. (2001), Boddy et al. (2005), Schnelte (2009), Li et al. (2009), Bozic e Wotawa (2018b), Bozic e Wotawa (2020), Lima (2020)
Desvantagem	Completude da descrição	Howe et al. (1995)
	Escalabilidade	Howe et al. (1995)
	Nível de abstração	Boddy et al. (2005)
	Tempo de execução	Howe et al. (1995)
	Concretização	Memon et al. (2001), Bozic et al. (2017)

Como destacado na Tabela 3.1, a principal vantagem da utilização do planejamento em IA no teste de software é a possibilidade de *representação* de problemas. Os estudos enfatizam que, de maneira geral, a sintaxe das linguagens de planejamento em IA e a usabilidade dos planejadores proporcionam *facilidade* para a obtenção de representações e planos de IA legíveis (*legibilidade*) para usuários gerais.

[1]Esta seção destaca os termos contidos na Tabela 3.1.

O planejamento em IA possibilita a definição de diferentes níveis de *abstração* para a representação. Isso permite que o usuário defina o nível de *expressividade* conforme a *completude da descrição* do problema. Diante disso, o planejamento em IA também proporciona representações flexíveis (*flexibilidade*) e que podem ser adaptadas (*adaptação*), configuradas (*configurabilidade*) e estendidas (*extensibilidade*), conforme a necessidade.

Ainda, as representações com planejamento em IA podem ser construídas com a *agregação* de informação existente e o conhecimento resultante pode ser reutilizado (*reuso de conhecimento*) em outros contextos. Os estudos reforçam que a *disponibilidade* de ferramentas de código aberto contribuem para o uso do planejamento em IA na prática. Ainda sobre as ferramentas, a *automatização* da execução e o *uso de critérios e heurísticas* favorecem a *performance* de construção e a *corretude* do plano de IA gerado.

Como mostra a Tabela 3.1, a principal desvantagem da aplicação do planejamento em IA no teste de software é a dificuldade para a *concretização* e uso prático das informações dos planos de IA. Ainda, de acordo com as características de cada problema, a definição do *nível de abstração* e da *completude da descrição* requeridos para a representação pode ser dificultada. Além disso, os problemas são facilmente escaláveis (*escalabilidade*), o que pode impactar no *tempo de execução* das ferramentas e na performe das soluções.

3.2 TESTE DE INTEGRAÇÃO E PLANEJAMENTO EM IA

A revisão bibliográfica identificou oito estudos que aplicam o planejamento em IA no teste de integração. Esses estudos abordam explicitamente o teste de integração ou propõem abordagens que podem ser direcionadas para esse teste. Esta seção apresenta as propostas, detalha as representações com planejamento em IA e descreve os planos de IA desses estudos. Os estudos foram categorizados em quatro grupos: domínio, *Unified Modeling Language* (UML), erros e componentes. As próximas subseções são organizadas conforme essa categorização.

3.2.1 Estratégia baseada em domínio

Mraz et al. (1995), Howe et al. (1995) e Howe et al. (1997) se baseiam no domínio do SUT para elaborar uma representação com planejamento em IA. Nesses estudos, um domínio é um conjunto de funcionalidades de um SUT descritas como comandos funcionais. Esses comandos estabelecem subdomínios quando parametrizados. Cada subdomínio pode ser associado a um subsistema gerado a partir da integração de unidades. Dessa forma, esses estudos contribuem para o teste de integração ao permitir a representação de um ou mais comandos agregados.

Os estudos ilustram a representação utilizando uma aplicação de *biblioteca*[2] (sistema sob teste). A partir de uma *especificação de comandos* (artefato), são obtidos elementos para representar o comportamento do SUT. Cada comando ou conjunto de comandos com funcionalidade similar é representado como uma ação. Regras e parâmetros indicam sequências significativas de comandos e são modelados como pós-condições e efeitos, respectivamente.

Comandos de sintaxe são representados como ações e as pré-condições incluem condições que satisfazem a execução de um comando. Condições que se tornam verdadeiras após a execução de um comando complementam as pós-condições das ações. Os estados inicial e final são conjuntos de parâmetros que denotam relações e operações entre eles. O Código 3.1 descreve a modelagem em *ADL* (linguagem) da ação `movefour`[3] que representa o comando de mover um volume `?vid` do módulo de origem `?slsm` para o módulo de destino `?dlsm`.

[2]Esta seção destaca os elementos analisados na revisão bibliográfica indicados na Tabela A.1.

[3]Os códigos foram mantidos no idioma original adotado pelos estudos analisados.

Código 3.1: Ação para a estratégia baseada em domínio. Fonte: Mraz et al. (1995).

```
(define (operator movefour)

  :parameters ((loc ?slsm) ?sp ?sc ?sr
               (loc ?dlsm) ?dp ?dc ?dr
               (tape ?vid))

  :precondition (and
                   (full slev)
                   (on ?slsm)
                   (on ?dlsm)
                   (eq ?slsm ?dlsm)
                   (neq ?sp ?dp)
                   (in ?vid ?slsm ?sp ?sr ?sc)
                   (eq ?dc unknown)
                   (eq ?dr unknown)
                 )

  :effect (and
             (from ?vid ?slsm ?sp ?sr ?sc ?dlsm ?dp ?dr ?dc)
             (in ?vid ?dlsm ?dp ?dr ?dc)
             (not (in ?vid ?slsm ?sp ?sr ?sc))
           )
)
```

Planos de IA indicam *sequências de comandos* (plano de IA) que compreendem o teste da funcionalidade especificada no estado objetivo. O Código 3.2 apresenta o plano de IA gerado pelo *UCPOP* (planejador) para uma instância específica do comando `movefour`. Após o tratamento, o plano de IA destaca as operações relevantes, conforme indicado no Código 3.3. Um caso de teste é selecionado para cada operação. Quando executados, esses casos de teste estabelecem o teste completo da funcionalidade representada.

Código 3.2: Plano de IA para a estratégia baseada em domínio. Fonte: Mraz et al. (1995).

```
Step 1: (MODIFYTOON 0) Created 3
  0 -> (FULL SLEV)
  0 -> (OFF 0)
Step 2: (MODIFYTOON 1) Created 2
  0 -> (FULL SLEV)
  0 -> (OFF 1)
Step 3: (MOVEONE EVT280 1 0 UNKNOWN UNKNOWN UNKNOWN UNKNOWN
                            UNKNOWN UNKNOWN) Created 1
  3 -> (ON 0)
  2 -> (ON 1)
  0 -> (FULL SLEV)
  0 -> (CONNECT 0 1)
  0 -> (IN EVT280 0 UNKNOWN UNKNOWN UNKNOWN)
  0 -> (TAPE EVT280)
  0 -> (Loc 1)
  0 -> (Loc 0)
```

As características dos planos de IA sugerem que os estudos estão relacionados ao *design de casos de teste* (atividade de teste). Diante das possibilidades de parametrização, a representação pode ser associada aos testes de *unidade*, *integração* ou *sistema* (fase de teste), baseados em *teste funcional* (técnica de teste).

Código 3.3: Plano de IA simplificado para a estratégia baseada em domínio. Fonte: Mraz et al. (1995).

```
MODIFY 0 0 0 ONLINE
SRVLEV FULL
DRAIN 000
ENTER 000
MOVE VOLUME(EVT280)
Tlsm (001)
```

3.2.2 Estratégia baseada em UML

Scheetz et al. (1999), von Mayrhauser et al. (1999) e Andrews et al. (2002) propõem uma representação com planejamento em IA baseada em diagramas de classes da UML. Esses diagramas representam a arquitetura conceitual de sistemas OO, incluindo as classes do SUT e as interações entre as classes. A partir do diagrama UML, é possível definir objetivos de teste que abrangem uma única classe ou grupos de classes agregadas. Dessa forma, a representação pode ser ajustada para o teste de integração.

No contexto desses estudos, o diagrama de classes é derivado do domínio do SUT. Assim, a estrutura geral da representação segue a descrição da Subseção 3.2.1. No entanto, as informações utilizadas para instanciar a representação são extraídas dos *diagramas de classes* (artefato). Identificadores e atributos de classes são usados para instanciar predicados, enquanto métodos e transições entre classes são modelados como predicados.

Cada diagrama pode ser dividido em subdiagramas que contêm objetivos de teste e condições específicas do SUT. O estado inicial da representação é parametrizado de acordo com as condições definidas e as relações entre classes indicadas no diagrama. Cada objetivo de teste corresponde a um estado final delimitado na representação. O uso da representação é ilustrado com uma aplicação de *biblioteca* (sistema sob teste).

A modelagem é feita em *ADL* (linguagem) e os planos de IA são gerados com o *UCPOP* (planejador). Um possível objetivo de teste no SUT representado é verificar a operação "uma porta de acesso segura uma fita". Esse objetivo foi delineado a partir do subdiagrama que representa a interação entre a classe porta (`cap`) e a classe fita (`tape`). Em termos de planejamento em IA, o objetivo foi descrito no estado final da representação apresentado no Código 3.4. O Código 3.5 apresenta o excerto do plano de IA para esse objetivo.

Código 3.4: Estado final para a estratégia baseada em UML. Fonte: Scheetz et al. (1999).

```
(:exists (cap ?c)
(:exists (tape ?t)
(holds cap ?c tape ?t)))
```

Código 3.5: Plano de IA para a estratégia baseada em UML. Fonte: Scheetz et al. (1999).

```
Eject EVT185 CAP000
```

A operação indicada no plano de IA é mapeada para um caso de teste que deve ser executado para alcançar o objetivo pretendido. Dessa maneira, as ações dos planos de IA indicam uma *sequência de comandos* (plano de IA) que pode ser mapeada em casos de teste executáveis. Ao permitir esse mapeamento, a representação pode ser associada ao *design de casos de teste* (atividade de teste). Os subdiagramas associam a representação aos testes de *unidade*, *integração* ou *sistema* (fase de teste) baseados em *teste funcional* (técnica de teste).

3.2.3 Estratégia baseada em erros

A proposta de von Mayrhauser et al. (2000) usa planejamento em IA para representar possíveis erros do SUT. O objetivo é definir uma representação errônea do SUT e gerar planos de IA que auxiliam em testes para a recuperação de erros. Essa representação é uma variação da estratégia baseada em domínio. Assim, a estratégia baseada em erros se associa ao teste de integração pelas mesmas características mencionadas na Subseção 3.2.1.

A representação do SUT inclui operadores de mutação derivados do domínio do SUT. Dessa forma, a representação é associada ao *teste baseado em erros* (técnica de teste). Os operadores de mutação definidos envolvem a substituição ou remoção de parâmetros, objetos e nome de ações, alteração de valores de parâmetros e objetos, negação ou remoção de predicados e troca de conectivos lógicos. Esses operadores podem ser locais (aplicados no escopo de uma única ação) ou globais (aplicados à representação completa).

A representação é instanciada a partir do *documento de operadores de mutação* (artefato), dos objetivos do teste e das condições iniciais do SUT. A representação é exemplificada em uma aplicação de *biblioteca* (sistema sob teste), porém seu foco principal de aplicação são SUTs críticos, como dispositivos médicos. O Código 3.6 apresenta um excerto da pré-condição da ação `move` em *ADL* (linguagem). Essa ação descreve a movimentação de uma fita de um módulo de origem `?lsm_s` para um módulo de destino `?lsm_d`.

Código 3.6: Ação para a estratégia baseada em erros. Fonte: von Mayrhauser et al. (2000)

```
1 (:and
2   (on ?lsm_s)
3   (on ?lsm_d)
4   (full)
5   ...
6   (inPos Slot ?tape_id ?slot_s ?p_s ?lsm_s ?acs_s)
7   (isOpen ?slot_d ?p_d ?lsm_d ?acs_d)
8 )
```

O Código 3.7 apresenta a mesma ação com a inclusão de um operador de mutação de substituição de parâmetros. Conforme indicado nas linhas 2 e 3 do Código 3.7, os objetos `?lsm_s` e `?lsm_d` foram substituídos por `?acs_a`. O Código 3.8 apresenta o excerto do plano de IA gerado pelo *UCPOP* (planejador) que indica o parâmetro ilegal como erro esperado.

Código 3.7: Operador de mutação aplicado na ação para a estratégia baseada em erros. Fonte: von Mayrhauser et al. (2000)

```
1 (:and
2   (on ?acs_a)
3   (on ?acs_a)
4   (full)
5   ...
6   (inPos Slot ?tape_id ?slot_s ?p_s ?lsm_s ?acs_s)
7   (isOpen ?slot_d ?p_d ?lsm_d ?acs_d)
8 )
```

Código 3.8: Plano de IA simplificado para a estratégia baseada em erros. Fonte: von Mayrhauser et al. (2000)

```
1 MODIFY ACSO ON
2 MOVE LSMOO PO0 S O O O O LSMOO PO1
```

Seguindo essa abordagem, o plano de IA gerado a partir dessa representação é uma *sequência de comandos* (plano de IA). As ações do plano de IA podem ser mapeadas em casos de teste, associando a representação ao *design de casos de teste* (atividade de teste). A

parametrização de predicados e ações vincula a representação ao teste de *unidade*, *integração* ou *sistema* (fase de teste).

3.2.4 Estratégia baseada em componentes

Silva e Lemos (2011) elaboraram uma representação com planejamento em IA baseada na configuração arquitetural de componentes. No contexto desse estudo, cada componente é uma porção reutilizável do SUT composta por interfaces bem definidas. O objetivo da solução é gerar planos de IA que indicam uma ordem de integração e teste dos componentes conforme a organização estabelecida na arquitetura.

A representação é aplicada em *sistemas autoadaptativos* (sistema sob teste). Esses sistemas incorporam componentes em sua estrutura durante adaptações, resultando em uma reconfiguração arquitetônica. Consequentemente, surge a necessidade de realizar novos testes em cada configuração resultante. Esse estudo considera a existência de mecanismos auxiliares que estabelecem a ordem de integração dos componentes, geram os *stubs* para o teste e determinam os casos de teste necessários.

A representação é instanciada a partir de informações contidas em uma *especificação de configuração dos componentes* (artefato) do SUT. Essa representação inclui ações que indicam a integração do componente, o teste do componente, além de duas ações responsáveis por restaurar o teste e o SUT a uma determinada configuração. O Código 3.9 descreve a representação da ação `TestComponent` em *PDDL* (linguagem).

Essa representação utiliza variáveis numéricas para indicar a etapa da integração, a etapa do teste e uma ordem de integração associada a cada componente. As pré-condições das ações envolvem principalmente a comparação entre essas variáveis (linha 7, Código 3.9). Nos efeitos da ações, essas variáveis são incrementadas (linha 11, Código 3.9). O estado inicial inicializa as variáveis e indica a ordem de integração de cada componente.

Código 3.9: Ação para a estratégia baseada em componentes. Fonte: Silva e Lemos (2011).

```
1  (:action TestComponent
2   :parameters (?C - component)
3
4   :precondition (and
5                   (not (tested ?C))
6                   (integrated ?C)
7                   (= (testing_step) (integration_order ?C)))
8
9   :effect (and
10            (tested ?C)
11            (increase (testing_step) 1))
12 )
```

O Código 3.10 apresenta o plano de IA gerado pelo *SGPlan* (planejador) para um sistema composto por cinco componentes. Como é possível observar, as ações do plano de IA indicam a *ordem de integração e teste* (plano de IA) dos componentes. Essas características do plano de IA associam a representação às atividades de *planejamento do teste* e *execução do teste* (atividade de teste).

Os testes são realizados seguindo a ordem estabelecida pelo plano de IA, usando os casos de teste e *stubs* previamente definidos. Diante desses aspectos, esse estudo aborda explicitamente o *teste de integração* (fase de teste) apoiado por *teste funcional* (técnica de teste).

Código 3.10: Plano de IA para a estratégia baseada em componentes. Fonte: Silva e Lemos (2011).

```
1: INTEGRATECOMPONENT C5
2: TESTCOMPONENT C5
3: INTEGRATECOMPONENT C3
4: TESTCOMPONENT C3
5: INTEGRATECOMPONENT C4
6: TESTCOMPONENT C4
7: INTEGRATECOMPONENT C2
8: TESTCOMPONENT C2
9: INTEGRATECOMPONENT C1
10: TESTCOMPONENT C1
```

3.3 ANÁLISE DOS TRABALHOS RELACIONADOS

Esta seção apresenta uma análise dos estudos descritos na Seção 3.2. Ao associar os conceitos de planejamento em IA e teste de integração, esses estudos são entendidos como trabalhos relacionados à esta tese. A análise busca discutir as principais características das estruturas de teste envolvidas em cada estratégia, argumentar decisões de implementação e destacar lacunas nas propostas definidas. Essa análise se complementa com o aprofundamento desses estudos apresentado no Apêndice C.

Com relação aos estudos baseadas em domínio da Subseção 3.2.1, observa-se o uso de especificações de comandos que podem ser derivadas de documentos que indicam os requisitos funcionais do SUT. Isso pode ser interpretado como um fator favorável à aplicação dessa estratégia, já que esses documentos são artefatos convencionalmente utilizados no processo de desenvolvimento de software.

Sobre os estudos baseados em UML descritos na Subseção 3.2.2, percebe-se que sua utilização é favorecida pois os diagramas UML são convencionais no desenvolvimento de software. Nesse sentido, essa estratégia pode ser associada à atividade de planejamento do teste, pois os diagramas UML fornecem detalhes da estrutura do SUT que são utilizados no início do processo do teste. Essas informações proporcionam representações que podem estabelecer uma maneira sistemática e automatizada de gerar casos de teste a partir de descrições UML.

A estratégia baseada em erros, apresentada na Subseção 3.2.3, é direcionada para aplicações críticas. Compreende-se que a detecção precoce e a correção de erros são fundamentais para garantir o funcionamento seguro e adequado dessas aplicações. Assim, o planejamento em IA pode contribuir para a obtenção de SUTs confiáveis. Além disso, o contexto baseado em erros prioriza o planejamento do teste. Acredita-se que isso ocorre pois os erros mais comuns do SUT, que derivam os operadores de mutação, devem ser identificados nos processos iniciais do teste.

A estratégia baseada em componentes descrita na Subseção 3.2.4 oferece uma maneira sistemática e automatizada de lidar com a complexidade da integração de sistemas autoadaptativos. Nesse contexto, a utilização de planejamento em IA proporciona uma estratégia adaptável para cenários de integração variados. Dessa forma, a aplicação dessa estratégia é uma maneira de otimizar a validação de SUTs em diferentes configurações arquiteturais.

Diferentemente das demais estratégias mencionadas, a estratégia baseada em componentes se concentra principalmente na atividade de execução do teste. Esse foco pode estar relacionado à delimitação no contexto de teste de integração do estudo usado como base para a definição dessa estratégia. Dessa forma, elementos adicionais requeridos para a execução do teste de integração, como *stubs*, estão explícitos na abordagem.

As diferentes estratégias abordam uma variedade de tipos de unidades em suas propostas. Nas abordagens baseadas em domínio e erros, os comandos funcionais são considerados

como unidades. Por outro lado, o contexto baseado em UML utiliza as classes descritas nos diagramas como suas unidades centrais, enquanto a estratégia baseada em componentes adota os componentes como unidades.

Essa diversidade de abordagens ressalta a importância da ênfase no planejamento do teste de integração, o qual deve levar em conta as características específicas de cada contexto. Apesar dessa variedade destacada nos SUTs, somente no contexto da UML é possível inferir o uso de um paradigma de programação específico e detalhes da estrutura hierárquica dos sistemas.

Dentre os estudos analisados, apenas Silva e Lemos (2011) abordam explicitamente a aplicação da solução no teste de integração. No entanto, aspectos específicos desses testes, como a ordenação das unidades e a codificação e utilização de *stubs*, são alcançados com mecanismos externos à abordagem. Observa-se também que a solução não se destina a uma estratégia de integração específica. Contudo, a utilização de *stubs* implica um possível direcionamento da proposta para uma estratégia *top-down*.

Os outros estudos foram relacionados ao teste de integração devido à possibilidade de adaptação de suas características a esse contexto. No entanto, vale ressaltar a existência de uma lacuna em relação ao detalhamento de aspectos intrínsecos aos testes de integração. Entre esses aspectos, destacam-se a falta de definição de critérios para a ordenação das unidades e a ausência de consideração sobre a utilização de *drivers*. Outra lacuna identificada em todos os estudos é a falta de indicação de interessados e o modelo de desenvolvimento de software previstos.

Diante dessas limitações, não foi identificado nas estratégias um direcionamento explícito para uma estratégia de integração específica. Além disso, no que diz respeito aos artefatos, nenhuma das estruturas estabelece especificações detalhadas quanto ao formato dos arquivos de entrada e saída, tampouco oferece orientações sobre a utilização dos planos de teste e de integração.

3.4 CONSIDERAÇÕES DO CAPÍTULO

A revisão bibliográfica sumarizada na Seção 3.1 proporcionou obter um panorama da aplicação do planejamento em IA no teste de software nos últimos 35 anos. A partir dos elementos analisados na revisão, esta tese se posiciona perante as soluções da literatura da seguinte forma:

- **Sistema sob teste**: esta tese considera sistemas procedimentais e sistemas OO como os sistemas sob teste. Dessa maneira, a abordagem proposta é adaptada para cenários em que procedimentos, métodos e classes são as unidades em teste durante a integração;
- **Artefato**: esta tese utiliza especificações variadas do SUT para obter informações usadas para instanciar as representações com planejamento em IA e gerar planos de IA. As especificações contêm descrições textuais ou gráficas estabelecidas durante as fases de design de arquitetura e design de componentes;
- **Fase de teste**: esta tese é inserida na fase de teste de integração procedimental e OO. Assim, a abordagem proposta abrange as estratégias de integração incremental (*top-down* e *bottom-up*), para o contexto procedimental, e os testes intermétodos e interclasses, no contexto OO.
- **Atividade de teste**: esta tese abrange principalmente a atividade de planejamento, com possibilidade de extensão para a execução do teste. Dessa forma, a abordagem envolve os elementos representados na Figura 2.2 apoiados pelo artefato de plano de teste, considerando gerentes de teste e testadores como os usuários pretendidos;

- **Técnica de teste**: esta tese é baseada em técnicas de teste funcional;
- **Linguagem de planejamento em IA**: esta tese utiliza a linguagem PDDL. Conforme discutido na Seção 3.1, a escolha da PDDL é corroborada pela literatura, sendo motivada principalmente pela possibilidade de representar elementos associados ao teste com valorações numéricas;
- **Planejador**: esta tese utiliza o Metric-FF por ser um planejador usualmente considerado para a resolução de problemas em PDDL. O uso desse planejador também é motivado por questões de desempenho e disponibilidade de código, como abordado na Seção 3.1;
- **Plano de IA**: esta tese gera planos de IA que indicam uma ordem para integrar e testar os SUTs conforme os aspectos do cenário de teste representado. Dessa forma, esses planos de IA são entendidos como **planos de IA de integração**, pois podem complementar o plano de integração contido no plano de teste, conforme detalhado na Subseção 2.1.1.

A Seção 3.2 apresentou estudos que usam o planejamento em IA no teste de integração. Esses estudos se baseiam no domínio das funcionalidades de um SUT, em diagramas UML que apresentam a estrutura de um SUT, nos principais erros que um SUT pode conter e nos componentes que compõem um SUT. As estratégias desses estudos consideram o teste de tipos variados de unidades, como conjuntos de comandos agregados, classes e componentes.

A variedade de contextos de integração identificados reforça a ênfase requerida no planejamento do teste de integração. Os contextos de cada estudo foram modelados com duas linguagens de planejamento em IA. As estratégias referentes a domínio, UML e erros foram representadas em ADL. Essas representações utilizam principalmente a representação de restrições proporcionada pela ADL. A estratégia de componentes foi modelada em PDDL, usando variáveis numéricas dessa linguagem como o diferencial da representação proposta.

Com a análise da Seção 3.3, o planejamento em IA pode ser observado principalmente como uma técnica de apoio ao planejamento do teste. A expressividade das linguagens permite delimitar representações que auxiliam a planejar as particularidades de cada contexto. No entanto, pode-se observar que os estudos são baseados em contextos que não apresentam os aspectos que motivam o uso do planejador indicados no Capítulo 1. Diante do exposto, as principais lacunas encontradas durante as análises dos estudos envolvem:

- **Aspectos gerais do teste de software**:

  la_1) os estudos não englobam o artefato de plano de teste;

  la_2) os estudos não consideram interessados específicos para diferentes atividades de teste.

- **Aspectos gerais do teste de integração**:

  la_3) os estudos não utilizam o artefato de plano de integração;

  la_4) os estudos não direcionam explicitamente a aplicação de uma estratégia de integração específica;

  la_5) os estudos não englobam a estratégia de integração *bottom-up*;

  la_6) os estudos não englobam a integração OO (intermétodos e interclasses);

  la_7) os estudos não indicam o uso de *drivers*.

- **Aspectos gerais dos sistemas sob teste**:

la_8) os estudos não incluem explicitamente nas representações elementos dos SUTs, como parâmetros de entrada, tipos de dados e informações da estrutura hierárquica;

la_9) os estudos não consideram especificidades de SUTs de paradigmas de programação distintos.

- **Aspectos gerais da estrutura**:

la_{10}) os estudos não delimitam formatos para os arquivos de entrada e saída;

la_{11}) os estudos não indicam formas para instanciar os arquivos de entrada e tratar os arquivos de saída do planejador.

- **Aspectos gerais do desenvolvimento de software**:

la_{12}) os estudos não indicam um modelo de desenvolvimento de software específico.

Com base nas motivações abordadas no Capítulo 1, nos conceitos descritos no Capítulo 2 e nas lacunas de pesquisa encontradas nos estudos analisados neste capítulo, o próximo capítulo apresenta uma abordagem de teste de integração com planejamento em IA.

4 ABORDAGEM TI-PIA

Conforme discutido no capítulo anterior, foram identificadas lacunas nas propostas da literatura que aplicam o planejamento em IA no teste de integração. Essas lacunas estão relacionadas a aspectos do teste de software, do teste de integração, dos SUTs, das estruturas de teste e do modelo de desenvolvimento. Para cobrir essas deficiências e com base nas motivações e conceitos previamente apresentados, este capítulo introduz uma abordagem de teste de integração com planejamento em inteligência artificial denominada TI-PIA.

A abordagem TI-PIA inclui uma estrutura de geração de planos de integração organizada em módulos e representações em PDDL baseadas em modelagens do teste de integração em diferentes contextos. A abordagem gera planos de IA usados como planos de integração que indicam uma ordem para a atribuição de unidades aos testadores. Este capítulo apresenta o contexto da abordagem (Seção 4.1), uma visão geral da abordagem (Seção 4.2), a modelagem do teste procedimental (Seção 4.3) e a modelagem do teste OO (Seção 4.4).

4.1 CONTEXTO

A Figura 4.1 ilustra o contexto geral da abordagem TI-PIA e destaca em cinza seu contexto específico. Como indicado, a abordagem considera o teste de integração executado em *sprints* que envolvem as fases do Modelo V. Cada fase é realizada em um ambiente: a modelagem de requisitos, as fases de design, a codificação e o teste de unidade ocorrem em um ambiente de desenvolvimento; os testes de integração, sistema e aceitação são conduzidos em um ambiente de homologação; e a implantação do SUT é realizada em um ambiente de produção.

Na abordagem, o teste possui restrições relacionadas às características dos SUTs, aos tipos de integração e às limitações de recursos. Com base nessas restrições, o gerente de teste define um plano de integração que apoia a realização do teste. A abordagem assume que uma ou mais unidades associadas implementam um requisito funcional do SUT. Nesse contexto, cada unidade do SUT codificada possui um tamanho[1] (*size*) variável. Após a codificação, cada unidade é testada individualmente pela equipe de desenvolvimento.

Na sequência, as unidades são integradas seguindo prioridades[2] de integração para sistemas procedimentais e OO. Para SUTs procedimentais, a abordagem abrange a integração seguindo as estratégias *top-down* e *bottom-up*. Para SUTs OO, são considerados os testes intermétodos e interclasses. Em todos esses casos, *stubs* ou *drivers* podem substituir unidades indisponíveis ou que não podem ser acessadas. Além disso, unidades de reuso podem ser incorporadas durante a integração para compor a estrutura do SUT.

Na abordagem, o teste de integração é entendido como um processo intermediário em um *sprint*. Assim, o teste de unidade é um pré-requisito para o teste de integração, que, por sua vez, antecede os testes de sistema e de aceitação. A abordagem considera o trabalho paralelo de um ou mais testadores que possuem uma capacidade[3] de teste limitada por *sprint*. Ao

[1] Tamanho pode ser entendido como uma métrica de complexidade ou de tamanho de código, como: linhas de código; número de atributos; complexidade ciclomática; e quantidade de unidades.

[2] Prioridade pode ser entendida como uma métrica de priorização de atividades (maior número realiza a atividade primeiro), como: número de chamadas; número de atributos; quantidade de unidades; e indicação numérica de alteração, remoção ou inclusão.

[3] Capacidade pode ser entendida como uma métrica da quantidade máxima de capacidade de teste por um testador, como: tamanho das unidades (complexidade ou linhas de código); quantidade de unidades; e tempo (duração do *sprint*).

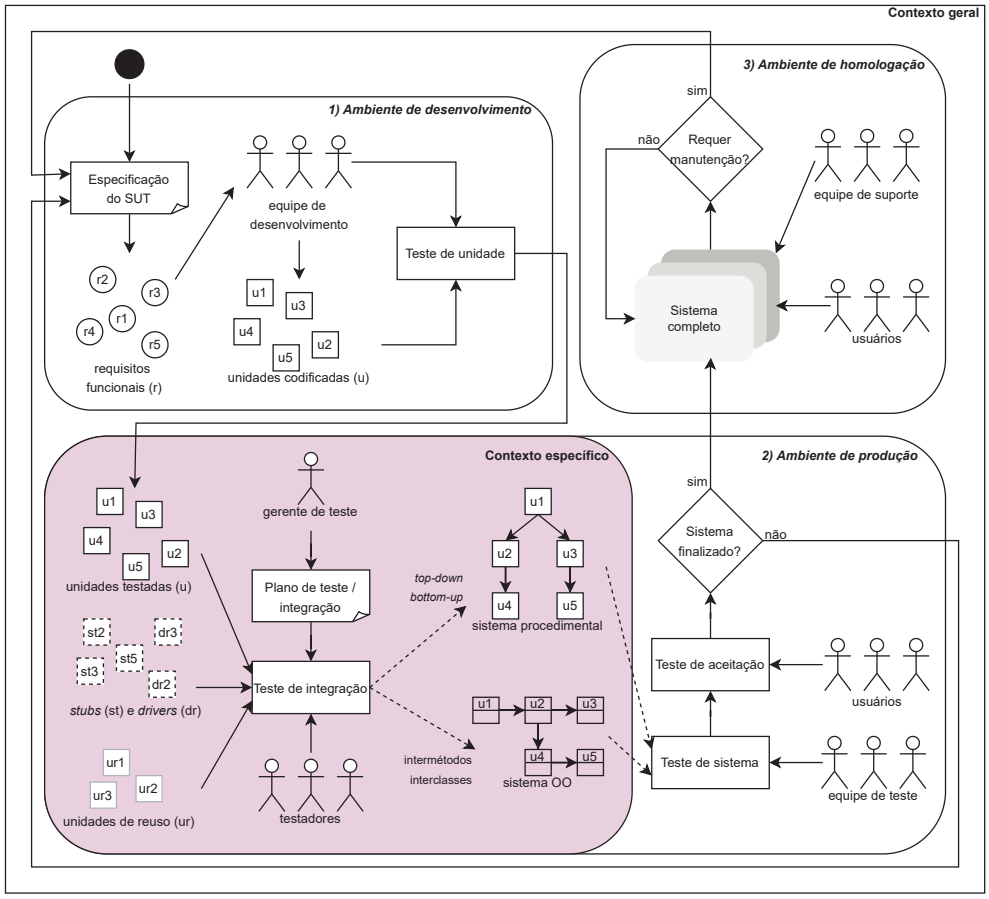


Figura 4.1: Contextos geral e específico da abordagem TI-PIA.

considerar as atividades de planejamento do teste e de execução do teste, a abordagem distribui responsabilidades entre gerentes de teste e testadores e situa o plano de teste como um elemento central da realização do teste.

4.2 VISÃO GERAL

A partir do contexto apresentado, a Figura 4.2 ilustra a abordagem TI-PIA. Essa figura destaca em cinza os elementos da abordagem investigados nesta tese. A abordagem TI-PIA é estruturada em uma estrutura dividida em módulos. Nesta tese, entende-se estrutura como um conjunto de elementos associados, como atividades, artefatos, software e hardware, que possibilitam o planejamento e a execução do teste de integração.

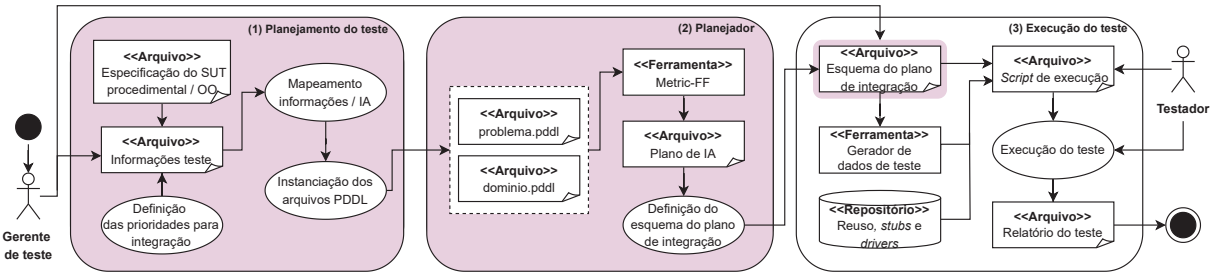


Figura 4.2: Visão geral da estrutura da abordagem TI-PIA.

A representação visual da estrutura incorpora elementos das Figuras 2.2 e 2.1 e é baseada na estrutura definida por Andrews et al. (2002). A estrutura é dividida em módulos de (1) Planejamento do teste, (2) Planejador e (3) Execução do teste. A Figura 4.3 apresenta a organização dos módulos e demais elementos definidos para a estrutura. O objetivo de cada módulo é apresentado adiante. A Subseção 4.2.1 detalha os módulos da abordagem TI-PIA.

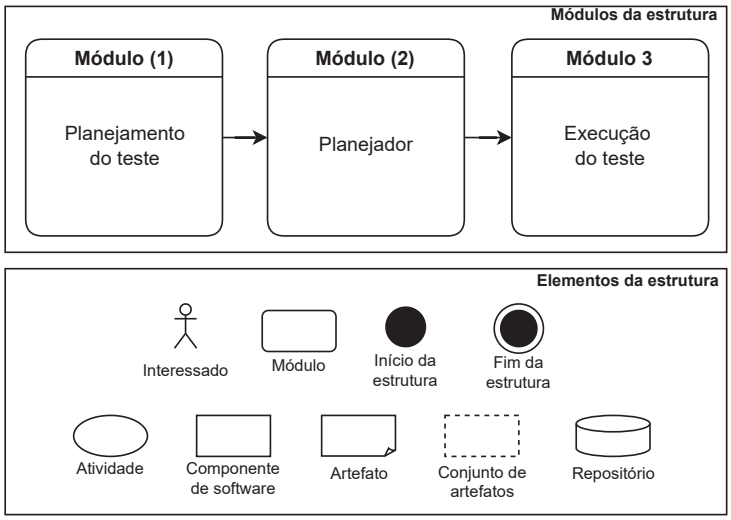


Figura 4.3: Módulos e elementos da estrutura da abordagem TI-PIA.

- **(1) Módulo do planejamento do teste**: esse módulo contém as especificações dos SUTs e as demais documentações e etapas necessárias para a definição do problema de planejamento em IA. Assim, esse módulo pode ser entendido como parte da atividade de planejamento do teste, descrita na Subseção 2.1.1;

- **(2) Módulo do planejador**: esse módulo compreende as ferramentas de planejamento em IA e os mecanismos que definem a entrada do planejador. Essa entrada é gerada por um mapeamento entre os elementos da especificação do SUT e as características do ambiente como um problema de planejamento em IA.

  O artefato resultante desse módulo é o plano de IA cujas informações podem ser entendidas como parte do artefato de plano de teste. Mais especificamente, o conteúdo do plano de IA pode complementar o plano de integração, mencionado na Seção 2.1.1. Logo, esse módulo também é associado com a atividade de planejamento do teste; e

- **(3) Módulo da execução do teste**: esse módulo indica os processos necessários para executar o teste de integração nos cenários delimitados, considerando ferramentas de apoio que geram dados de teste e repositórios que disponibilizam unidades de reuso, *stubs* e *drivers*.

  Assim, esse módulo associa-se com a atividade de execução de teste ao usar o plano de integração contido no plano de teste, complementado pelas informações do plano de IA, para a realização do teste. A saída esperada dessa atividade é um relatório de resultados do teste. Nesse contexto, a execução do teste equivale às atividades *Execução* $Prog(d)$ e *Comparação* $Prog(d)$ x e, indicadas na Figura 2.2.

4.2.1 Módulos da estrutura

Esta seção detalha os módulos ilustrados na Figura 4.2 e legendados na Figura 4.3.

• **(1) Módulo do planejamento do teste**: nesse módulo, o gerente de teste (*Gerente*[4]) obtém informações para o teste de integração em documentos com especificações textuais ou gráficas do SUT procedimental ou OO (≪Arquivo≫ *Especificação do SUT*). Esses documentos devem fornecer informações gerais das unidades, como parâmetros de entrada, tipos de dados, chamadas de operações e trocas de mensagens.

A estrutura considera que as prioridades atribuídas às unidades para determinar a ordem de integração (*Definição da ordem de integração*) são estabelecidas por mecanismos externos. O gerente de teste gera um arquivo com informações da especificação do SUT, com as características do ambiente e com as prioridades de integração estabelecidas (≪Arquivo≫ *Informações teste*). As características do ambiente incluem a quantidade de testadores disponíveis e a capacidade de teste suportada nos *sprints*.

Visando a automatização da abordagem e a flexibilidade para a representação de informações em texto, sugere-se o formato XML (W3C, 2008) para o arquivo *Informações teste*. As informações desse arquivo são mapeadas em elementos de um problema de planejamento em IA não-clássico estabelecido para a abordagem (*Mapeamento informações / IA*). Os elementos mapeados são usados na instanciação de arquivos PDDL contendo a representação do problema de planejamento (*Instanciação dos arquivos PDDL*).

A estrutura considera cenários do teste de integração representados em PDDL em arquivos de problema e domínio (≪Arquivo≫ *problema.pddl* e ≪Arquivo≫ *dominio.pddl*). As Seções 4.3 e 4.4 detalham, respectivamente, as representações PDDL do teste procedimental e do teste OO definidas para a abordagem.

• **(2) Módulo do planejador**: nesse módulo, os arquivos PDDL instanciados são encaminhados para o planejador (≪Ferramenta≫ *Planejador*), que gera automaticamente um plano de IA (≪Arquivo≫ *Plano de IA*). Esse plano indica uma ordem para integrar as unidades e testar o subsistema, conforme ilustra a estrutura da Figura 4.4. O planejador utilizado é determinístico e sua execução resulta em um único plano associado à integração em largura.

As unidades são distribuídas entre os *m* testadores disponíveis. As unidades são atribuídas alternadamente aos testadores conforme as prioridades de integração estabelecidas. Nesse contexto, uma atribuição representa a integração da unidade e o teste do subsistema gerado após essa integração. Após cada atribuição, a prioridade é incrementada. Um novo *sprint* é iniciado quando a capacidade máxima de teste dos testadores é atingida. Esse processo é repetido em *k sprints* até que as *n* unidades sejam atribuídas aos testadores.

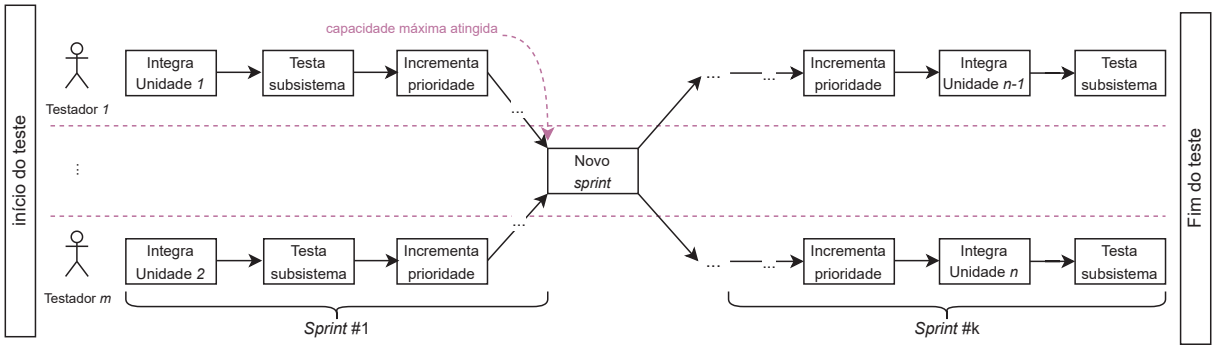


Figura 4.4: Estrutura dos planos de IA da abordagem TI-PIA.

[4]Esta seção destaca os elementos da Figura 4.2.

As ações usadas como *log* no plano de IA são descartadas e as informações restantes são usadas na definição de um esquema gráfico do plano de integração (*Definição do esquema do plano de integração*). O artefato resultante dessa definição ilustra a atribuição das unidades entre os testadores disponíveis ao longo dos *sprints* (≪Arquivo≫ *Esquema do plano de integração*). O esquema gráfico gerado integra o plano de teste.

• **(3) Módulo da execução do teste**: esse módulo visa conduzir os testes de integração com base nas informações do plano de integração. Para isso, esse módulo contém a infraestrutura necessária para a condução dos testes. Assim, os interessados previstos nesse módulo são testadores (*Testador*). As etapas dos testes consideradas no módulo de execução são baseadas na teoria de teste de software de Delamaro et al. (2016) introduzida na Seção 2.1.

Para iniciar a execução dos testes, as informações do plano de teste são usadas em um *script* (≪Arquivo≫ *Script de execução*). O testador utiliza esse *script* para executar os testes (*Execução do teste*) no SUT com base na ordem indicada no plano de teste. O *script* é executado até que cada subsistema gerado com a integração de unidades seja testado pelo menos uma vez.

O módulo de execução assume que uma ferramenta de software (≪Ferramenta≫ *Gerador de dados de teste*) gera e encaminha os dados de teste apropriados para cada execução do teste. Esse módulo também agrega um repositório (≪Repositório≫ Reuso, *stubs* e *drivers*) que fornece as unidades de reuso disponíveis, os *stubs* ou os *drivers* necessários de acordo com a estratégia de integração delimitada na representação.

Os resultados de cada execução de teste são direcionados para um arquivo texto (≪Arquivo≫ *Relatório do teste*) que o testador deve analisar. Durante a análise, o testador compara os resultados obtidos com cada dado de teste e os resultados esperados. O conjunto de casos de testes, formado pelos dados de teste e os respectivos resultados esperados com sua execução, são obtidos pela descrição dos requisitos contidos na especificação do SUT com base em uma técnica de teste definida pela equipe de teste.

4.3 MODELAGEM 1: TESTE DE INTEGRAÇÃO PROCEDIMENTAL

Nesta tese, uma modelagem consiste em três etapas: a) uma formalização dos elementos do SUT e do ambiente seguindo o contexto da Seção 4.1; b) um mapeamento desses elementos em um problema de planejamento em IA não-clássico P_{ianc} apresentado na Subseção 2.4.1; e c) uma representação desse P_{ianc} em um domínio e um problema PDDL, conforme apresentado na Subseção 2.4.1, usada no módulo (2) Planejador da estrutura da abordagem TI-PIA (Figura 4.2). Esta seção descreve a modelagem do teste de integração procedimental.

4.3.1 Formalização

Nesta modelagem, um SUT procedimental é formalizado como uma tupla $SUT_p = (Proc, Pa, Pt, Rt, Co, L)$, onde:

- $Proc = \{proc_1(size_1, Pa_1, pr_1, rt_1), proc2(size_2, Pa_2, pr_1, rt_2), ...\}$ é um conjunto finito de **procedimentos** do sistema, sendo cada $proc_i$ caracterizado por seu **tamanho** $size_i$, seus **parâmetros de entrada** Pa_i, sua **prioridade de integração** pr_i e seu **tipo de retorno** rt_i, sendo $pr_i \in Pr$ e $rt_i \in Rt$;
- $Pa = \{pa_1(pt_1), pa_2(pt_2), ...\}$ é um conjunto finito de **parâmetros de entrada**, onde cada pa_i possui um **tipo** pt_i, sendo $pt_i \in Pt$;
- $Pt = \{pt_1, pt_2, ...\}$ é um conjunto finito de **tipos de parâmetros de entrada**;

- $Rt = \{rt_1, rt_2, ...\}$ é um conjunto finito de **tipos de retorno**;
- $Co = \{(proc_1, proc_2), (proc_1, proc_3), ...\}$ é um conjunto finito de **chamadas de operações**, onde cada par $(proc_i, proc_j)$ representa que o procedimento $proc_i$ realiza uma chamada para o procedimento $proc_j$, com $proc_j \in Proc - \{proc_i\}$;
- $L = \{l_1, l_2, ...\}$ é um conjunto finito de **níveis** de uma árvore que representa a estrutura hierárquica do SUT. O nível l_i indica a profundidade de um nó na árvore, sendo k a altura da árvore (a maior distância entre um nó folha e a raiz da árvore).

Um sistema SUT_p está inserido em um ambiente de teste $Am_p = (T, Pr, TS_{max})$, onde:

- $T = \{t_1(cap_1), t_2(cap_2), ...\}$ é um conjunto finito de **testadores**, onde cada t_i possui uma **capacidade de teste** cap_i;
- $Pr = \{pr_1(w_1), pr_2(w_2), ...\}$ é um conjunto finito de **prioridades de integração**, onde cada pr_i possui um **peso** w_i;
- TS_{max} é a **capacidade máxima de teste** em um *sprint*.

A integração de procedimentos $proc \in Proc$ ocorre após a checagem de seus parâmetros. Cada procedimento é associado a um tamanho $size$ estabelecido a partir do esforço requerido para o seu teste. Cada procedimento possui um conjunto de parâmetros de entrada Pa tipificados e um valor de retorno com um tipo rt baseado na especificação do SUT. Os procedimentos são associados a uma prioridade $pr \in P_r$. Prioridades pr mais altas têm precedência na integração e possuem um peso w associado proporcionalmente.

As chamadas de operação de Co estabelecem um estrutura hierárquica de SUT_p dividida em níveis $l \in L$. Os níveis são usados para definir o início da integração conforme a estratégia desejada: l_1 para a *top-down* e a l_k, sendo k a altura da árvore, para a *bottom-up*. Ainda, cada procedimento $proc_i$ está associado a um subconjunto de procedimentos, denotado por $Proc'_i \subseteq Proc - proc_i$, que inclui os procedimentos chamados por $proc_i$. A cada integração, um elemento de $Proc'_i$ é testado.

Os testes são divididos em *sprints* contendo uma capacidade máxima de teste TS_{max}. Os procedimentos são atribuídos aos testadores $t \in T$ seguindo as prioridades de P_r. Cada testador t possui um capacidade de teste cap limitada por *sprint*. As atribuições de procedimentos são feitas alternadamente entre os testadores $t \in T$ disponíveis no ambiente. Um testador t_i realiza a integração de um procedimento $proc$ caso $size(proc) \leq cap(t_i)$ naquele momento. As capacidades dos testadores são reiniciadas a cada novo *sprint*.

4.3.2 Mapeamento

A Tabela 4.1 apresenta o mapeamento entre os elementos formalizados na Subseção 4.3.1 e os elementos de um problema de planejamento em IA não-clássico P_{ianc} em PDDL. As próximas subseções detalham a representação em PDDL dos elementos da coluna "Planejamento em IA" e descrevem a estrutura do plano de IA resultante dessa representação. A Tabela 4.1 destaca os elementos da representação ajustáveis para cenários com mais de um testador e para cenários com as estratégias de integração *top-down* e *bottom-up*.

Tabela 4.1: Mapeamento entre elementos do teste procedimental e elementos do problema de planejamento em IA não clássico.

Teste procedimental	**Planejamento em IA**
Procedimento $proc \in Proc$	Objeto do tipo **t_procedure**
Parâmetro $pa \in Pa$	Objeto do tipo **t_parameter**
Tipo de parâmetro $pt \in Pt$	Objeto do tipo **t_parameter_type**
Tipo de retorno $rt \in Rt$	Objeto do tipo **t_return_type**
Nível $l \in L$	Objeto do tipo **t_level**
Testador $t \in T$	Objeto do tipo **t_tester**
Prioridade de integração de procedimento $pr \in Pr$	Objeto do tipo **t_priority**
Prioridade pr que deve ser atribuída ao testador no momento atual (estado)	Predicado **priority ?pr - t_priority**
Prioridade pr do procedimento $proc$	Predicado **p_priority ?proc - t_procedure pr - t_priority**
Chamada de operação entre os procedimentos $proc$ e $proc2$	Predicado **operation_call ?proc ?proc2 - t_procedure**
Testador t contido no ambiente de teste	Predicado **tester ?t - t_tester**
Procedimento $proc$ atribuído a um testador em um *sprint* finalizado	Predicado **done_all ?proc - t_procedure**
Procedimento $proc$ atribuído a um testador t no *sprint* atual	Predicado **done_tester ?proc - t_procedure ?t - t_tester**
Nivel l com procedimentos sendo integrado no momento atual	Predicado **level ?l - t_level**
Nível l do procedimento $proc$	Predicado **p_level ?proc - t_procedure ?l - t_level**
Parâmetro pa do procedimento $proc$	Predicado **p_parameter ?proc - t_procedure ?pa - t_parameter**
Tipo pt do parâmetro pa	Predicado **pa_type ?pa - t_parameter ?pt - t_parameter_type**
Tipo de retorno rt do procedimento $proc$	Predicado **p_return ?proc - t_procedure ?rt - t_return_type**
Checagem do parâmetro pa do procedimento $proc$	Predicado **checked_parameter ?pa - t_parameter ?proc - t_procedure**
Checagem das informações do procedimento $proc$	Predicado **checked_procedure_information ?proc - t_procedure**
Indicação da integração do procedimento $proc$	Predicado **integrated_procedure ?proc - t_procedure**
Tamanho $size$ do procedimento $proc$	Função **size ?proc - t_procedure**
Capacidade de teste cap do testador t em um *sprint*	Função **capacity ?t - _tester**
Capacidade máxima TS_{max} de teste em um *sprint*	Função **max_capacity**
Peso w da prioridade pr	Função **weight ?pr - t_priority**
Soma dos pesos w das prioridades pr de todos os procedimentos $proc \in Proc$	Função **sum**
Checar o parâmetro pa e seu tipo pt do procedimento $proc$	Ação **PROCEDURE_PARAMETER (?proc - t_procedure ?pa - t_parameter ?pt - t_parameter_type)**
Consolidar a checagem de parâmetros do procedimento $proc$	Ação **CHECKED_PARAMETER_INFORMATION (?proc - t_procedure)**
Atribuir a integração do procedimento $proc$ de prioridade pr, tipo de retorno rt e pertencente ao nível l ao testador t também responsável pelo teste do subsistema gerado	Ação **INTEGRATION_AND_TEST (?proc - t_procedure ?pr - t_priority ?t - t_tester ?rt - t_return_type ?l - t_level)**[a, b]
Atualizar a prioridade atual para a próxima prioridade	Ação **INCREASE_PRIORITY**
Trocar o testador em um *sprint*	Ação **CHANGE_TESTER**[a]
Finalizar o *sprint* atual e preparar o ambiente para o próximo *sprint*	Ação **NEW_SPRINT**[a]

Continua na próxima página

Tabela 4.1: Mapeamento entre elementos do teste procedimental e elementos do problema de planejamento em IA não clássico (continuação da página anterior).

Teste procedimental	**Planejamento em IA**
Atualizar a capacidade do testador t ao término de um *sprint*	Ação **RESET_CAPACITY (?t - t_tester)**[a]
Trocar nível l atual para o próximo nível	Ação **CHANGE_LEVEL (?l - t_level)**[b]

[a]Elemento ajustável para cenários com múltiplos testadores.
[b]Elemento ajustável para cenários com as estratégias de integração *top-down* ou *bottom-up*.

As próximas subseções detalham os elementos da coluna "Planejamento em IA" da Tabela 4.1 instanciados com as informações do sistema procedimental ilustrado na Figura 4.5. Esse sistema possui seis procedimentos, todos com $size = 3$, estruturados em três níveis. Foram definidos parâmetros e tipos de retorno para cada procedimento. O ambiente foi definido com seis prioridades (pr_1 a pr_6), *sprints* com $TS_{max} = 6$ e dois testadores t_1 e t_2, ambos com capacidade de teste $cap = 6$.

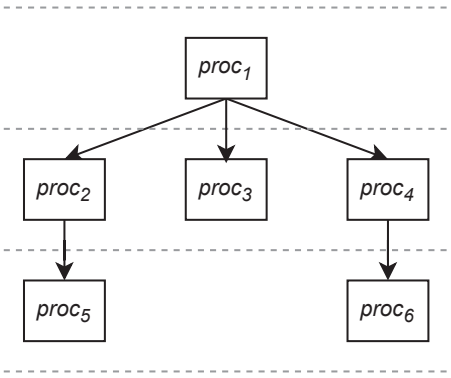


Figura 4.5: Exemplo de sistema procedimental usado para instanciar a representação PDDL do teste procedimental. Fonte: Adaptado de Myers et al. (2011).

4.3.3 Representação do domínio PDDL

Esta subseção descreve o domínio PDDL ajustável para as estratégias de integração *top-down* e *bottom-up* e para ambientes com um testador ou mais de um testador. Os códigos apresentados nesta subseção são voltados para um e dois testadores. Os elementos destacados em cinza nos códigos são incluídos na representação para os casos com dois testadores. A partir da lógica apresentada, o domínio pode ser adaptado para cenários contendo mais de dois testadores.

Os Códigos D.1 e D.4 do Apêndice D apresentam as representações completas do domínio PDDL para as estratégias *top-down* e *bottom-up*, respectivamente.

4.3.3.1 Tipos, predicados e funções

O Código 4.1 contém os tipos do domínio. Os procedimentos $proc \in Proc$, os testadores $t \in T$ e as prioridades $pr \in P_r$ são representados como objetos dos tipos `t_procedure`, `t_tester` e `t_priority`, respectivamente. Objetos do tipo `t_level` representam os níveis $l \in L$. Os parâmetros de entrada $pa \in Pa$, os tipos de parâmetros $pt \in Pt$ e os tipos de retorno $rt \in Rt$ são, respectivamente, objetos dos tipos `t_parameter`, `t_parameter_type` e `t_return_type`. Esses objetos instanciam o conjunto P de predicados do Código 4.2.

Código 4.1: Tipos do domínio PDDL do teste de integração procedimental.

```
1  (:types
2    t_procedure
3    t_tester
4    t_priority
5    t_level
6    t_parameter
7    t_parameter_type
8    t_return_type
9  )
```

O predicado `priority ?pr - t_priority` (linha 2, Código 4.2) assegura que *pr* seja a maior prioridade possível e que o procedimento com essa prioridade tenha precedência na integração. Dessa forma, a prioridade *pr* deve ser atribuída ao testador no momento atual (estado). Ainda, esse predicado também garante que um único procedimento $proc_i$ seja selecionado para a integração. O predicado `p_priority ?proc - t_procedure ?pr - t_priority` (linha 3, Código 4.2) associa uma prioridade *pr* a um procidimento *proc*.

Código 4.2: Predicados do domínio PDDL do teste de integração procedimental.

```
1  (:predicates
2    (priority ?pr - t_priority)
3    (p_priority ?proc - t_procedure ?pr - t_priority)
4    (operation_call ?proc ?proc2 - t_procedure)
5    (tester ?t - t_tester)
6    (done_all ?proc - t_procedure)
7    (done_tester ?proc - t_procedure ?t - t_tester)
8    (level ?l - t_level)
9    (p_level ?proc - t_procedure ?l - t_level)
10   (p_parameter ?proc - t_procedure ?pa - t_parameter)
11   (pa_type ?pa - t_parameter ?pt - t_parameter_type)
12   (p_return ?proc - t_procedure ?rt - t_return_type)
13   (checked_parameter ?pa - t_parameter ?proc - t_procedure)
14   (checked_procedure_information ?proc - t_procedure)
15   (integrated_procedure ?proc - t_procedure)
16 )
```

O predicado `operation_call ?proc ?proc2 - t_procedure` (linha 4, Código 4.2) foi modelado para indicar que um procedimento *proc* chama operações de um procidimento *proc*2. O predicado `tester ?t - t_tester` (linha 5, Código 4.2) representa o testador $t \in T$ responsável pelo teste no momento atual (estado).

A atribuição de procedimentos a testadores foi modelada em dois predicados. O predicado `done_all ?proc - t_procedure` (linha 6, Código 4.2) estabelece que o procedimento *proc* foi atribuído a um testador em um *sprint* finalizado. O predicado `done_tester ?proc - t_procedure ?t - t_tester` (linha 7, Código 4.2) indica que o procedimento *proc* foi atribuído ao testador *t* no *sprint* atual.

Informações sobre os níveis hierárquicos foram modelados em dois predicados. O predicado `level ?l - t_level` indica que o nível *l* é o nível atual que está sendo testado e o predicado `p_level ?proc - t_procedure ?l - t_level` indica o nível *l* no qual o procedimento *proc* está inserido (linhas 8 e 9, Código 4.2).

Os parâmetros e os tipos de parâmetros e de retorno foram representados em três predicados. O predicado `p_parameter ?proc - t_procedure ?pa - t_parameter` indica que *pa* é um parâmetro do procedimento *proc* (linha 10, Código 4.2). O predicado `pa_type ?pa - t_parameter ?pt - t_parameter_type` mostra o tipo *tp* do pa-

râmetro *pa* (linha 11, Código 4.2). O tipo *rt* retornado por *proc* é indicado pelo predicado `p_return ?proc - t_procedure ?rt - t_return_type` (linha 12, Código 4.2).

Alguns predicados representam o progresso da integração (linhas 13 a 15, Código 4.2). O predicado `checked_parameter ?pa - t_parameter ?proc - t_procedure` indica que um parâmetro *pa* de um procedimento *proc* já foi verificado. O predicado `checked_procedure_information ?proc - t_procedure` indica que as informações do procedimento *proc* já foram verificadas. O predicado `integrated_procedure ?proc - t_procedure` representa que o procedimento *proc* foi integrado após a verificação de suas informações.

Os elementos numéricos foram modelados como funções (conjunto F), conforme o Código 4.3. A função `size ?proc - t_procedure` representa o tamanho do procedimento *proc* (linha 2, Código 4.3). A capacidade de teste de um testador *t* e a capacidade máxima de teste de um *sprint* foram modeladas, respectivamente, como as funções `capacity ?t - t_tester` e `max_capacity` (linhas 3 e 4, Código 4.3). A função `weight ?pr - t_priority` indica o peso *w* da prioridade *pr*. A função `sum` armazena a soma dos pesos das prioridades.

Após a declaração desses elementos iniciais, o domínio PDDL contém a descrição de oito ações que formam o conjunto A. Essas ações representam a checagem de parâmetros dos procedimentos, o teste de integração dos procedimentos, a atualização das prioridades de integração, a troca de testadores em um *sprint*, a inicialização de um novo *sprint*, a atualização das capacidades de teste em um novo *sprint* e a troca de níveis hierárquicos.

Código 4.3: Funções do domínio PDDL do teste de integração procedimental.

```
1 (:functions
2   (size ?proc - t_procedure)
3   (capacity ?t - t_tester)
4   (max_capacity)
5   (weight ?pr - t_priority)
6   (sum)
7 )
```

4.3.3.2 Ações para checar os parâmetros dos procedimentos

A ação `PROCEDURE_PARAMETER` verifica individualmente cada parâmetro de um procedimento antes que o procedimento seja considerado para integração. Assim, essa ação atua como uma etapa intermediária no processo de preparação do procedimento para o teste. A representação em PDDL da ação `PROCEDURE_PARAMETER` é descrita pelo Código 4.4. Essa ação tem como parâmetros um procedimento *proc*, um parâmetro *pa* e o tipo do parâmetro *pt*. Suas pré-condições são:

- o parâmetro *pa* deve estar associado ao procedimento *proc* (linha 5, Código 4.4);
- o parâmetro *pa* deve possuir um tipo *pt* (linha 6, Código 4.4); e
- o parâmetro *pa* do procedimento *proc* ainda não foi verificado (linha 7, Código 4.4).

Caso as pré-condições sejam satisfeitas, a execução da ação gera como efeito:

- o parâmetro *pa* é indicado como verificado (linha 10, Código 4.4).

Código 4.4: Ação PROCEDURE_PARAMETER do domínio PDDL do teste de integração procedimental.

```
1  (:action PROCEDURE_PARAMETER
2   :parameters (?proc - t_procedure ?pa - t_parameter ?pt - t_parameter_type)
3
4   :precondition (and
5                   (p_parameter ?proc ?pa)
6                   (pa_type ?pa ?pt)
7                   (not (checked_parameter ?pa ?proc))
8                 )
9
10  :effect (checked_parameter ?pa ?proc)
11 )
```

A ação CHECKED_PARAMETER_INFORMATION valida que todos os parâmetros associados a um procedimento foram verificados e permite, então, que ocorra o teste após a integração desse procedimento. O Código 4.5 descreve essa ação em PDDL. Essa ação tem como único parâmetro um procedimento $proc$ e suas pré-condições são:

- o procedimento $proc$ ainda não foi integrado (linha 5, Código 4.5);
- as informações do procedimento $proc$ ainda não foram verificadas (linha 6, Código 4.5); e
- para todos os parâmetros pa, ou eles não são associados ao procedimento $proc$ ou já foram checados (linhas 7 a 10, Código 4.5).

Se todas essas pré-condições são satisfeitas, o efeito da execução da ação é:

- as informações do procedimento $proc$ são marcadas como verificadas (linhas 13, Código 4.5).

Código 4.5: Ação CHECKED_PARAMETER_INFORMATION do domínio PDDL do teste de integração procedimental.

```
1  (:action CHECKED_PARAMETER_INFORMATION
2   :parameters (?proc - t_procedure)
3
4   :precondition (and
5                   (not (integrated_procedure ?proc))
6                   (not (checked_procedure_information ?proc))
7                   (forall (?pa - t_parameter)
8                    (or
9                     (not (p_parameter ?proc ?pa))
10                    (checked_parameter ?pa ?proc)))
11                )
12
13  :effect (checked_procedure_information ?proc)
14 )
```

As ações PROCEDURE_PARAMETER e CHECKED_PARAMETER_INFORMATION formam um mecanismo estruturado que promove a verificação incremental dos parâmetros de todos os procedimentos $proc \in Proc$. Esse mecanismo garante que apenas procedimentos cujos parâmetros foram totalmente verificados sejam integrados.

4.3.3.3 *Ação para integrar procedimentos e testar subsistema*

A ação INTEGRATION_AND_TEST atribui a um testador a integração de um procedimento e o teste após cada integração. A representação em PDDL dessa ação é descrita no Código 4.6. Os parâmetros dessa ação são um procedimento *proc*, uma prioridade de integração *pr*, um testador *t*, um tipo de retorno *rt* e um nível *l*.

Código 4.6: Ação INTEGRATION_AND_TEST para o domínio PDDL do teste de integração procedimental.

```
1  (:action INTEGRATION_AND_TEST
2   :parameters (?proc - t_procedure ?pr - t_priority ?t - t_tester
3                ?rt - t_return_type ?l - t_level)
4
5   :precondition (and
6                   (priority ?pr)
7                   (tester ?t)
8                   (level ?l)
9                   (p_priority ?proc ?pr)
10                  (p_level ?proc ?l)
11                  (p_return ?proc ?rt)
12                  (checked_procedure_information ?proc)
13                  (not (integrated_procedure ?proc))
14                  (>= (capacity ?t) (size ?proc))
15                  (forall (?tester - t_tester)
16                    (not (done_tester ?proc ?tester)))
17                  (forall (?proc2 - t_unit)
18                    (or
19                      (not (operation_call ?proc ?proc2))
20                      (done_tester ?proc2 ?t)
21                      (done_all ?proc2)))
22                )
23
24   :effect (and
25             (done_tester ?proc ?t)
26             (decrease (capacity ?t) (size ?proc))
27             (increase (sum) (weight ?pr))
28             (not (priority ?pr))
29             (priority PR1)
30             (integrated_procedure ?proc)
31             (when (tester tester1)
32               (and
33                 (not (tester tester1))
34                 (tester tester2)))
35             (when (tester tester2)
36               (and
37                 (not (tester tester2))
38                 (tester tester1)))
39           )
40  )
```

As pré-condições da ação INTEGRATION_AND_TEST são:

- o predicado `priority ?pr` deve ser verdadeiro para a prioridade *pr* (linha 6, Código 4.6);
- o predicado `tester ?t` deve ser verdadeiro para o testador *t* (linha 7, Código 4.6);
- o predicado `level ?l` deve ser verdadeiro para o nível *l* (linha 8, Código 4.6);

- o procedimento *proc* deve ter a prioridade de integração *pr* (linha 9, Código 4.6);
- o procedimento *proc* deve estar no nível *l* (linha 10, Código 4.6);
- o procedimento *proc* deve ter o valor de retorno *rt* (linha 11, Código 4.6);
- as informações dos parâmetros do procedimento *proc* já foram checadas (linha 12, Código 4.6);
- o procedimento *proc* ainda não foi integrado (linha 13, Código 4.6);
- a capacidade do testador *t* deve ser menor ou igual ao tamanho do procedimento *proc* (linha 14, Código 4.6);
- o procedimento *proc* não pode ter sido atribuído a um testador previamente, ou seja, o predicado `done_tester ?proc ?tester` deve ser falso para todos os testadores $t \in T$ (linhas 15 e 16, Código 4.6); e
- todos os procedimentos *proc*2 chamados pelo procedimento *proc* devem ter sido atribuídos a um testador em um *sprint* anterior ou já foram atribuídos ao testador *t* no *sprint* atual (linhas 17 a 21, Código 4.6). Essa pré-condição é necessária para testes baseados na estratégia *bottom-up*.

Se todas as pré-condições são satisfeitas, a ação é executada, tendo como efeitos:

- O procedimento *proc* é atribuído ao testador *t* (linha 25, Código 4.6);
- a capacidade do testador *t* é decrementada com o valor do tamanho do procedimento *proc* (linha 26, Código 4.6);
- o valor da função `sum` é incrementado com o peso da prioridade *pr* (linha 27, Código 4.6);
- a prioridade *pr* é atualizada para a prioridade mais alta (linhas 28 e 29, Código 4.6);
- o procedimento *proc* é indicado como integrado (linha 30, Código 4.6); e
- a atribuição de procedimentos é alternada entre os testadores disponíveis (linhas 31 a 38, Código 4.6);

4.3.3.4 Ação para atualizar a prioridade de integração

A ação `INCREASE_PRIORITY` é responsável por atualizar as prioridades de integração após a atribuição dos procedimentos. O Código 4.7 apresenta a representação dessa ação em PDDL com uma instância que considera três prioridades (pr_1, pr_2 e pr_3). Essa ação não tem parâmetros ou pré-condições e seu único efeito é incrementar a prioridade *pr* após a atribuição do procedimento com a *pr*. Dessa forma, a próxima atribuição será feita com um procedimento com a prioridade $pr + 1$.

Na instância do Código 4.7, se um procedimento com a prioridade pr_1 foi atribuído a um testador para integração (linha 7), o predicado `priority pr1` deve ser negado (linha 9). Essa negação é feita para que um procedimento com a prioridade pr_2 seja integrado na próxima atribuição, indicada pelo predicado `priority pr2` verdadeiro (linha 11).

A atualização de prioridades estabelecida na ação INCREASE_PRIORITY ocorre até que os procedimentos com todas as prioridades $p_r \in P_r$ sejam atribuídos a um testador. Após a integração de todos os procedimentos com todas as prioridades, a prioridade é atualizada para pr_0 (linha 22, Código 4.7).

Código 4.7: Ação INCREASE_PRIORITY para o domínio PDDL do teste de integração procedimental.

```
1  (:action INCREASE_PRIORITY
2    :parameters ()
3
4    :precondition (and )
5
6    :effect (and
7              (when (priority PR1)
8                (and
9                  (not (priority PR1))
10                 (priority PR2)))
11             (when (priority PR2)
12               (and
13                 (not (priority PR2))
14                 (priority PR3)))
15             (when (priority PR3)
16               (and
17                 (not (priority PR3))
18                 (priority PR0)))
19           )
20 )
```

4.3.3.5 Ação para trocar o testador

A ação CHANGE_TESTER é responsável por realizar a troca de testadores quando há mais de um testador envolvido no teste. O Código 4.8 apresenta a representação dessa ação em PDDL. Essa ação não tem pré-condições ou efeitos. Seus efeitos são trocar os testadores (linhas 7 a 16, Código 4.8) quando a capacidade do testador atual atinge o valor máximo. Ao realizar essa troca, a atribuição ocorre a partir da prioridade mais alta (linhas 19 a 21, Código 4.8).

Código 4.8: Ação CHANGE_TESTER para o domínio PDDL do teste de integração procedimental.

```
1  (:action CHANGE_TESTER
2    :parameters ()
3    :precondition (and )
4
5    :effect (and
6              (when (tester tester1)
7                (and
8                  (not (tester tester1))
9                  (tester tester2)))
10             (when (tester tester2)
11               (and
12                 (not (tester tester2))
13                 (tester tester1)))
14             (forall (?pr - t_priority)
15               (not (priority ?pr)))
16             (priority PR1))
17 )
```

4.3.3.6 *Ação para iniciar um novo sprint*

A ação NEW_SPRINT finaliza o *sprint* atual e prepara o ambiente para o próximo *sprint*. A representação em PDDL dessa ação é apresentada no Código 4.9. Essa ação não possui parâmetros. Sua pré-condição é atribuir o valor zero para a capacidade de todos os testadores $t \in T$ (linhas 5 e 6, Código 4.9). Após essa pré-condição ser satisfeita, a ação gera como efeitos:

- o predicado done_all é indicado como verdadeiro pra todo procedimento *proc* (linhas 10 a 12, Código 4.9);
- a capacidade de todos os testadores *t* é atualizada com o valor da capacidade máxima de teste por *sprint* (linhas 13 a 16, Código 4.9);
- o predicado priority é atualizado com o valor da maior prioridade (linhas 17 a 19, Código 4.9); e
- é realizada a indicação de qual testador deve iniciar o teste no próximo *sprint* (linha 20, Código 4.9).

Código 4.9: Ação NEW_SPRINT para o domínio PDDL do teste de integração procedimental.

```
1  :action NEW_SPRINT
2    :parameters ()
3
4    :precondition (and
5                    (forall (?t - t_tester)
6                      (= (capacity ?t) 0))
7                  )
8
9    :effect (and
10             (forall (?proc - t_procedure ?t - t_tester)
11               (when (done_tester ?proc ?t)
12                 (done_all ?proc)))
13             (forall (?t - t_tester)
14               (and
15                 (assign (capacity ?t)(max_capacity))
16                 (not (tester ?t))))
17             (forall (?pr - t_priority)
18               (not (priority ?pr)))
19             (priority PR1)
20             (tester tester1)
21            )
22 )
```

4.3.3.7 *Ação para atualizar a capacidade de teste*

A ação RESET_CAPACITY tem como objetivo reiniciar a capacidade de teste de um testador. A descrição em PDDL dessa ação é apresentada no Código 4.10. O parâmetro da RESET_CAPACITY é um testador *t* e suas pré-condições são:

- o predicado tester deve ser verdadeiro para o testador *t* (linha 5, Código 4.10); e
- a prioridade atual deve ser pr_0 (linha 6, Código 4.10).

Se todas as pré-condições são satisfeitas, a ação é executada, tendo como efeitos:

- a capacidade do testador *t* é atribuída com o valor 0 (linha 10, Código 4.10);
- o predicado **priority** é atualizado com o valor da maior prioridade (linhas 11 e 12, Código 4.10); e
- é realizada a troca de testadores em casos em que mais de um testador esteja disponível (linhas 13 a 20, Código 4.10).

Código 4.10: Ação RESET_CAPACITY para o domínio PDDL do teste de integração procedimental.

```
1  (:action RESET_CAPACITY
2   :parameters (?t - t_tester)
3
4   :precondition (and
5                   (tester ?t)
6                   (priority PR0)
7                 )
8
9   :effect (and
10            (assign (capacity ?t) 0)
11            (not (priority PR0))
12            (priority PR1)
13            (when (tester tester1)
14              (and
15                (not (tester tester1))
16                (tester tester2)))
17            (when (tester tester2)
18              (and
19                (not (tester tester2))
20                (tester tester1)))
21          )
22 )
```

4.3.3.8 Ação para trocar o nível hierárquico

A ação `CHANGE_LEVEL` controla a transição entre níveis hierárquicos do SUT. Essa ação assegura que a integração progrida de forma sequencial e ordenada por meio dos níveis hierárquicos, seguindo a estrutura definida em cada estratégia de integração. Essa ação também garante que todos os procedimentos de um nível sejam integrados antes que o teste avance para o nível subsequente. A descrição em PDDL da ação `CHANGE_LEVEL` é apresentada no Código 4.11. Essa ação tem como parâmetro um nível *l* e suas pré-condições são:

- o predicado `level ?l` deve ser verdadeiro para o nível *l* (linha 5, Código 4.11); e
- para todos os procedimentos $proc$, ou eles não estão no nível *l* ou já foram integrados (linhas 6 a 9, Código 4.11).

A ação é executada caso essas pré-condições sejam satisfeitas, gerando como efeitos:

- o nível *l* atual é negado (linha 13, Código 4.11);
- é realizada a troca para o próximo nível de acordo com a estratégia de integração definida para o teste (linhas 14 a 21, Código 4.11). O exemplo simula um SUT com três níveis (l_1, l_2 e l_3) percorridos a partir de uma integração *top-down* (primeiro nível em direção ao último). Na integração *bottom-up* a troca de níveis ocorre no sentido inverso; e

- o predicado `priority` é atualizado para a prioridade mais alta (linhas 22 a 24, Código 4.11).

Código 4.11: Ação CHANGE_LEVEL para o domínio PDDL do teste de integração procedimental.

```
1  (:action CHANGE_LEVEL
2   :parameters (?l - t_level)
3
4   :precondition (and
5                   (level ?l)
6                   (forall (?proc - t_procedure)
7                    (or
8                     (not (p_level ?proc ?l))
9                     (integrated_procedure ?proc)))
10                )
11
12  :effect (and
13            (not (level ?l))
14            (when (level L1)
15             (and
16              (not (level L1))
17              (level L2)))
18            (when (level L2)
19             (and
20              (not (level L2))
21              (level L3)))
22            (forall (?pr - t_priority)
23             (not (priority ?pr)))
24            (priority PR1))
25 )
```

4.3.4 Representação do problema PDDL

Esta subseção descreve o problema PDDL para o teste de integração procedimental. Nesse contexto, cada arquivo de problema PDDL é caracterizado pelos elementos de um sistema SUT_p e de um ambiente Am_p específicos. Os Códigos D.2 e D.5 apresentam, respectivamente, instâncias do problema PDDL para as estratégias *top-down* e *bottom-up* considerando o sistema procedimental da Figura 4.5.

Parte dos elementos do SUT e do ambiente são representados com a declaração de objetos (Conjunto O). Esses objetos representam os procedimentos $proc \in Proc$, os parâmetros de entrada $pa \in Pa$, os tipos de parâmetros $pt \in Pt$, os tipos de retorno $rt \in Rt$, os níveis $l \in L$, os testadores $t \in T$ e as prioridades de integração $pr \in Pr$. Os demais elementos são representados pelos predicados e funções (conjuntos A e F) inicializados no estado inicial s_0.

Em s_0, são estabelecidos o testador que iniciará os testes (predicado `tester`), a primeira prioridade considerada (predicado `priority`), o nível pelo qual a integração será iniciada (predicado `level`) e as chamadas de operações entre os procedimentos (predicado `operation_call`). Na sequência, para cada $proc \in Proc$, são indicados os níveis (predicado `p_level`), as prioridades de integração (predicado `p_priority`), os parâmetros e seus tipos (predicados `p_parameter` e `pa_type`) e os tipos de retorno (predicado `p_return`).

A declaração de s_0 é finalizada com a inicialização das funções. São indicados os tamanhos dos procedimentos (função `size`), os pesos das prioridades (função `weight`), a capacidade máxima de teste por *sprint* (função `max_capacity`) e a capacidade dos testadores

(função `capacity`). A função `sum` é inicializada com o valor *0*, indicando que s_0 equivale ao momento anterior ao início do teste, com os testadores sem procedimentos atribuídos para teste.

O estado final s_g é alcançado quando a função `sum` atinge o valor da soma dos pesos w de todas as prioridades pr associadas aos procedimentos $proc_i \in Proc$, com $0 < i \leq |Proc|$. Quando um procedimento $proc_i$ é atribuído a um testador t, o valor da função `sum` é incrementado com o peso w da prioridade pr_i de $proc_i$. Assim, o objetivo se aproxima quanto maior o valor de `sum`. Dessa forma, s_g equivale ao momento em que todos os procedimentos foram integrados após serem atribuídos a um testador e um teste foi realizado após cada integração.

O domínio e o problema PDDL descritos nesta seção, instanciados com o sistema da Figura 4.5, geraram os planos de IA apresentados no Apêndice D. O plano de IA do Código D.3 exemplifica a integração *top-down*, em que as ações indicam a integração a partir do procedimento $proc_1$ em direção ao $proc_6$. No plano de IA para a estratégia *bottom-up*, apresentado no Código D.6, as ações indicam que a integração é realizada a partir dos procedimentos de nível hierárquico mais baixo ($proc_5$ e $proc_6$) em direção ao procedimento $proc_1$.

4.4 MODELAGEM 2: TESTE DE INTEGRAÇÃO ORIENTADO A OBJETOS

Os elementos definidos na Seção 4.3 basearam uma adaptação da modelagem para sistemas orientados a objetos (OO). Esta seção apresenta a modelagem decorrente dessa adaptação, considerando o teste intermétodos, o teste interclasses e o contexto delimitado na Seção 4.1.

4.4.1 Formalização

Nesta modelagem, um SUT OO é caracterizado pela tupla $SUT_{oo} = (C, Atr, Ta, M, Arg, Targ, Rt_m, Me)$, onde:

- $C = \{c_1(size_1, Atr_1, M_1, pr_{c_1}), c_2(size_2, At_2, M_2, pr_{c_2}), ...\}$ é um conjunto finito de **classes**, sendo cada c_i caracterizada por seu **tamanho** $size_i$, seus **atributos** Atr_i, seus métodos M_i e sua **prioridade de integração** pr_{c_i}, sendo $pr_{c_i} \in Pr_c$;
- $Atr = \{atr_1(at_1), atr_2(at_2), ...\}$ é um conjunto finito de **atributos**, onde cada atr_i possui um **tipo de atributo** at_i, sendo $at_i \in At$;
- $At = \{at_1, at_2, ...\}$ é um conjunto finito de **tipos de atributos**;
- $M = \{m_1(size_1, Arg_1, pr_{m1}, rt_{m_1}), m_2(size_2, Arg_2, pr_{m2}, rt_{m_2}), ...\}$ é um conjunto finito de **métodos**, sendo cada m_i caracterizado por seu **tamanho** $size_i$, seus **argumentos** Arg_i, sua **prioridade de integração** pr_{mi} e seu **tipo de retorno** rt_{m_i}, sendo $pr_{mi} \in Pr_m$ e $rt_{m_i} \in Rt_m$;
- $Arg = \{arg_1(argt_1), arg_2(argt_2), ...\}$ é um conjunto finito de **argumentos**, onde cada arg_i possui um **tipo de argumento** t_{arg_i}, sendo $argt_i \in ArgT$;
- $Argt = \{argt_1, argt_2, ...\}$ é um conjunto finito de **tipos de argumentos**;
- $Rt_m = \{rt_{m_1}, rt_{m_2}, ...\}$ é um conjunto finito de **tipos de retorno**;
- $Me = \{(m_1, m_2), (m_1, m_3), ...\}$ é um conjunto finito de **trocas de mensagens**, onde cada par (m_i, m_j) representa que o método m_i realiza uma troca de mensagem com o método m_j, com $m_j \in M - \{m_i\}$.

Um sistema SUT_{oo} pertence a um ambiente $Am_{oo} = (T, Pr_m, Pr_c, TS_{max})$, onde:

- $T = \{t_1(cap_1), t_2(cap_2), ..., \}$ é um conjunto finito de **testadores**, onde cada t_i possui uma **capacidade de teste** cap_i;
- $Pr_m = \{pr_{m_1}(w_1), pr_{m_2}(w_2), ...\}$ é um conjunto finito de **prioridades de integração de métodos**, onde cada pr_{m_i} possui um **peso** w_i;
- $Pr_c = \{pr_{c_1}(w_1), pr_{c_2}(w_2), ...\}$ é um conjunto finito de **prioridades de integração de classes**, onde cada pr_{c_i} possui um **peso** w_i;
- TS_{max} é **capacidade máxima de teste** em um *sprint*.

Os testes de integração são realizados sobre as classes $c \in C$ e os métodos $m \in M$ de um sistema SUT_{oo}. Os atributos $atr \in Atr$, os tipos de atributos $at \in At$ e os tipos de retorno $rt \in t$ das classes são definidos na especificação do SUT. Essa especificação também determina os argumentos $arg \in Arg$ e seus respectivos tipos $argt \in Argt$ nos métodos. A integração é iniciada por métodos ou classes que possuem maior prioridade associada.

O teste intermétodos é guiado por prioridades de integração atribuídas aos métodos $pr_m \in Pr_m$, enquanto o teste interclasses é conduzido com base nas prioridades de integração das classes $pr_c \in Pr_c$. Ambas as prioridades possuem pesos w associados, que influenciam na indicação do término do teste. Esses testes são organizados em *sprints*, cuja capacidade de teste é limitada por um valor máximo TS_{max}.

Cada método e classe possui um tamanho $size$ que reflete o esforço necessário para realizar seu teste. Um testador $t \in T$ está apto a realizar o teste de um método m ou uma classe c caso $size(m) \leq cap(t)$ ou $size(c) \leq cap(t)$. As atribuições de métodos ou classes são alternadas entre os testadores $t \in T$ disponíveis e as capacidades desses testadores são reiniciadas a cada *sprint*. As atribuições ocorrem após a checagem de informações de métodos e classes. Essa formalização não abrange herança, encapsulamento, abstração e polimorfismo.

4.4.2 Mapeamento

Com base na formalização da Seção 4.4.1, a Tabela 4.2 mapeia os elementos do teste OO em elementos de um problema de planejamento em IA não-clássico P_{ianc} em PDDL. A representação em PDDL dos elementos da coluna "Planejamento em IA" e a descrição do plano de IA resultante dessa representação são detalhados nas próximas subseções. A Tabela 4.2 indica elementos ajustáveis para cenários com múltiplos testadores e elementos específicos para os testes intermétodos e interclasses. Elementos sem indicações são usados em ambos os testes.

Tabela 4.2: Mapeamento entre elementos do teste OO e elementos do problema de planejamento em IA não clássico.

Teste OO	**Planejamento em IA**
Classe $c \in C$	Objeto do tipo **t_class**
Atributo $atr \in Atr$	Objeto do tipo **t_attribute**[c]
Tipo de atributo $at \in At$	Objeto do tipo **t_attribute_type**[c]
Método $m \in M$	Objeto do tipo **t_method**
Argumento $arg \in Arg$	Objeto do tipo **t_argument**[b]
Tipo de argumento $argt \in Argt$	Objeto do tipo **t_argument_type**[b]
Tipo de retorno de método $rt_m \in Rt_m$	Objeto do tipo **t_return_type**[b]
Testador $t \in T$	Objeto do tipo **t_tester**
Prioridade de integração de método $pr_m \in Pr_m$	Objeto do tipo **t_mpriority**[b]

Continua na próxima página

Tabela 4.2: Mapeamento entre elementos do teste OO e elementos do problema de planejamento em IA não clássico (continuação da página anterior).

Teste OO	**Planejamento em IA**
Prioridade de integração de classe $pr_c \in Pr_c$	Objeto do tipo **t_cpriority**[c]
Prioridade de integração de método pr_m que deve ser atribuída ao testador no momento atual (estado)	Predicado **mpriority ?prm - t_mpriority**[b]
Prioridade de integração de método pr_m do método m	Predicado **method_priority ?m - t_method ?prm - t_mpriority**[b]
Prioridade de integração de classe pr_c que deve ser atribuída ao testador no momento atual (estado)	Predicado **cpriority ?prc - t_cpriority**[c]
Prioridade de integração de classe pr_c da classe c	Predicado **class_priority ?c - t_class ?prc - t_cpriority**[c]
Testador t contido no ambiente de teste	Predicado **tester ?t - t_tester**
Troca de mensagem entre os métodos m e me	Predicado **message ?m ?me - t_method**
Método m atribuído a um testador em um *sprint* finalizado	Predicado **done_all ?m - t_method**[b]
Método m atribuído a um testador t no *sprint* atual	Predicado **done_tester ?m - t_method ?t - t_tester**[b]
Classe c atribuída a um testador em um *sprint* finalizado	Predicado **done_all ?c - t_class**[c]
Classe c atribuída a um testador t no *sprint* atual	Predicado **done_tester ?c - t_class ?t - t_tester**[c]
Classe c com métodos sendo integrados no momento atual	Predicado **class ?c - t_class**[b]
Método m da classe c	Predicado **class_method ?m - t_method ?c - t_class**
Argumento arg do método m	Predicado **method_argument ?m - t_method ?arg - t_argument**[b]
Tipo $argt$ do argumento arg	Predicado **arg_type ?arg - t_argument ?argt - t_argument_type**[b]
Tipo de retorno rt_m do método m	Predicado **method_return ?m - t_method ?rtm - t_return_type**[b]
Atributo atr da classe c	Predicado **class_attribute ?c - t_class ?atr - t_attribute**[c]
Tipo at do atributo atr	Predicado **attribute_type ?atr - t_attribute ?at - t_attribute_type**[c]
Checagem do argumento arg do método m	Predicado **checked_argument ?arg - t_argument ?m - t_method**[b]
Checagem das informações dos argumentos do método m	Predicado **checked_argument_information ?m - t_method**[b]
Checagem do atributo atr da classe c	Predicado **checked_attribute ?atr - t_attribute ?c - t_class**[c]
Checagem das informações dos atributos da classe c	Predicado **checked_attribute_information ?c - t_class**[c]
Checagem do método m da classe c	Predicado **checked_method ?m - t_method ?c - t_class**[c]
Checagem das informações dos métodos da classe c	Predicado **checked_method_information ?c - t_class**[c]
Indicação da integração do método m	Predicado **integrated_method ?m - t_method**[b]
Indicação da integração da classe c	Predicado **integrated_class ?c - t_class**[c]
Capacidade de teste cap do testador t em um *sprint*	Função **capacity ?t - t_tester**
Capacidade máxima TS_{max} de teste em um *sprint*	Função **max_capacity**
Tamanho $size$ do método m	Função **method_size ?m - t_method**[b]
Peso w da prioridade de integração de método pr_m	Função **mpriority_weight ?prm - t_mpriority**[b]
Tamanho $size$ da classe c	Função **class_size ?c - t_class**[c]
Peso w da prioridade de integração de classe pr_c	Função **cpriority_weight ?prc - t_cpriority**[c]
Soma dos pesos w das prioridades pr_m ou pr_c de todos os métodos $m \in M$ ou de todas as classes $c \in C$	Função **sum**
Checar o argumento arg e seu tipo at do método m	Ação **METHOD_ARGUMENT (?m - t_method ?arg - t_argument ?at - t_argument_type)**[b]

Continua na próxima página

Tabela 4.2: Mapeamento entre elementos do teste OO e elementos do problema de planejamento em IA não clássico (continuação da página anterior).

Teste OO	**Planejamento em IA**
Consolidar a checagem de argumentos do método m	Ação **CHECKED_ARGUMENT_INFORMATION (?m - t_method)**[b]
Checar o método m da classe c	Ação **CLASS_METHOD (?c - t_class ?m - t_method)**[c]
Consolidar a checagem de métodos da classe c	Ação **CHECKED_METHOD_INFORMATION (?c - t_class)**[c]
Checar o atributo atr da classe c	Ação **CLASS_ATTRIBUTE (?c - t_class ?atr - t_attribute ?at - t_attribute_type)**[c]
Consolidar a checagem de métodos da classe c	Ação **CHECKED_ATTRIBUTE_INFORMATION (?c - t_class)**[c]
Atribuir a integração do método m de prioridade prm, tipo de retorno rtm e pertencente à classe c ao testador t também responsável pelo teste do subsistema gerado	Ação **METHOD_INTEGRATION_AND_TEST (?m - t_method ?prm - t_mpriority ?t - t_tester ?c - t_class ?rtm - t_return_type)**[a]
Atribuir a integração da classe c de prioridade prc ao testador t também responsável pelo teste do subsistema $Proc'$ gerado	Ação **CLASS_INTEGRATION_AND_TEST (?c - t_class ?prc - t_cpriority ?t - t_tester)**[a]
Atualizar a prioridade atual para a próxima prioridade	Ação **INCREASE_PRIORITY**
Trocar o testador em um *sprint*	Ação **CHANGE_TESTER**[a]
Finalizar o *sprint* atual e preparar o ambiente para o próximo *sprint*	Ação **NEW_SPRINT**[a]
Atualizar a capacidade do testador t ao término de um *sprint*	Ação **RESET_CAPACITY (?t - t_tester)**[a]
Trocar classe c atual para a próxima classe a ser integrada	Ação **CHANGE_CLASS (?C - t_classs)**[b]

[a]Elemento ajustável para cenários com múltiplos testadores.
[b]Elemento específico do teste intermétodos.
[c]Elemento específico do teste interclasses.

As próximas subseções detalham a representação em PDDL dos elementos da coluna "Planejamento em IA" da Tabela 4.2 considerando o sistema OO ilustrado em um diagrama de classes simplificado na Figura 4.6. O sistema OO utilizado é composto por oito classes. Foram definidos atributos, tipos, métodos e argumentos para compor o sistema. As classes e os métodos foram associados a oito prioridades distintas. O ambiente delimitado contém dois testadores com capacidade de teste $cap = 6$ que realizam os testes em *sprints* com $TS_{max} = 6$.

4.4.3 Representação do domínio PDDL

Esta subseção descreve o domínio PDDL adaptável para os testes de integração intermétodos e interclasses e para ambientes com um ou mais testadores. Os códigos apresentados nesta subseção são referentes a um ambiente com dois testadores. A adaptação para ambientes com mais de dois testadores segue a lógica dos códigos destacados em cinza na Subseção 4.3.3. Elementos específicos para os testes intermétodos e interclasses são indicados por comentários[5] nos códigos desta subseção. Os Códigos D.7 e D.10 do Apêndice D descrevem instâncias completas do domínio PDDL para os testes intermétodos e interclasses, respectivamente.

4.4.3.1 Tipos, predicados e funções

Os tipos do domínio são apresentados no Código 4.12. Objetos dos tipos `t_class`, `t_method` e `t_tester` representam, respectivamente as classes $c \in C$, os métodos $m \in M$ e

[5]As linhas iniciadas por "**;**" são comentários.

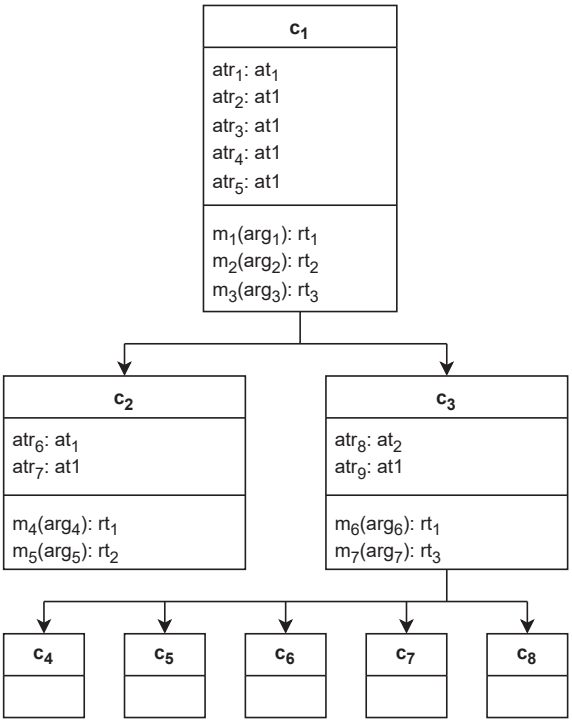


Figura 4.6: Exemplo de sistema OO usado para instanciar a representação PDDL do teste OO. Fonte: Adaptado de Siegel (1996).

os testadores $t \in T$ (linha 2, Código 4.12). Além desses objetos, a representação de cada tipo de teste contém alguns tipos de objetos específicos. A representação do teste intermétodos possui outros quatro objetos (linhas 5 a 8, Código 4.12), enquanto o teste interclasses contém outros três objetos (linhas 11 a 13, Código 4.12).

Código 4.12: Predicados do domínio PDDL do teste de integração OO intermétodos e interclasses.

```
1  (:types
2   t_class t_method t_tester
3
4   ;intermétodos
5   t_mpriority
6   t_argument
7   t_argument_type
8   t_return_type
9
10  ;interclasses
11  t_cpriority
12  t_attribute
13  t_attribute_type
14 )
```

As prioridades de integração de métodos $pr_m \in Pr_m$ e as prioridades de integração de classes $pr_c \in Pr_c$ são objetos dos tipos `t_mpriority` e `t_cpriority`, respectivamente. Os objetos `t_argument` e `t_argument_type` representam os argumentos $arg \in Arg$ e seus respectivos tipos $argt \in Argt$. O tipo de retorno dos métodos $rt_m \in Rt_m$ é modelado como um objeto do tipo `t_return_type`. Atributos $atr \in Atr$ e seus tipos $at \in At$ são, respectivamente, objetos dos tipos `t_attribute` e `t_attribute_type`.

O Código 4.13 descreve o conjunto P de predicados. O testador $t \in T$ responsável pelo teste é representado pelo predicado `tester ?t - t_tester`. A troca de mensagens entre

dois métodos *m* e *me* é modelada com o predicado `message ?m ?me - t_method`. O predicado `class_method ?m - t_method ?c - t_class` indica que o método *m* pertence à classe *c*. Além desses predicados em comum, cada tipo de teste possui um conjunto de objetos específicos.

Código 4.13: Predicados do domínio PDDL do teste de integração OO intermétodos e interclasses.

```
1  (:predicates
2    (tester ?t - t_tester)
3    (message ?m ?me - t_method)
4    (class_method ?m - t_method ?c - t_class)
5
6    ;intermétodos
7    (mpriority ?prm - t_mpriority)
8    (method_priority ?m - t_method ?prm - t_mpriority)
9    (done_all ?m - t_method)
10   (done_tester ?m - t_method ?t - t_tester)
11   (class ?c - t_class)
12   (integrated_method ?m - t_method)
13   (method_argument ?m - t_method ?arg - t_argument)
14   (arg_type ?arg - t_argument ?argt - t_argument_type)
15   (method_return ?m - t_method ?rtm - t_return_type)
16   (checked_argument ?arg - t_argument ?m - t_method)
17   (checked_argument_information ?m - t_method)
18
19   ;interclasses
20   (cpriority ?prc - t_cpriority)
21   (class_priority ?c - t_class ?prc - t_cpriority)
22   (done_all ?c - t_class)
23   (done_tester ?c - t_class ?t - t_tester)
24   (integrated_class ?c - t_class)
25   (class_attribute ?c - t_class ?atr - t_attribute)
26   (attribute_type ?atr - t_attribute ?at - t_attribute_type)
27   (checked_attribute ?atr - t_attribute ?c - t_class)
28   (checked_attribute_information ?c - t_class)
29   (checked_method ?m - t_method ?c - t_class)
30   (checked_method_information ?c - t_class)
31 )
```

A representação do teste intermétodos contém outros onze predicados (linhas 7 a 17, Código 4.13). A prioridade de integração de métodos pr_m atual considerada é representada no predicado `mpriority ?prm - t_mpriority`, enquanto a prioridade pr_m de cada método *m* é indicada pelo predicado `method_priority ?m - t_method ?prm - t_mpriority`. Dois predicados foram modelados para indicar a atribuição de métodos aos testadores.

O predicado `done_all ?m - t_method` indica que o método *m* já foi atribuído a algum testador em algum *sprint* anterior. O predicado `done_tester ?m - t_method ?t - t_tester` indica que o método *m* foi atribuído ao testador *t* para teste no *sprint* atual. O predicado `class ?c - t_class` especifica a classe *c* considerada no teste atual. O predicado `integrated_method ?m - t_method` indica que o método *m* foi integrado e que foi realizado um teste após essa integração.

Informações sobre sobre os métodos são representadas em três predicados. O predicado `method_argument ?m - t_method ?arg - t_argument` indica que o argumento *arg* pertence ao método *m*. O tipo *argt* do argumento *arg* é representado pelo predicado `arg_type ?arg - t_argument ?argt - t_argument_type`. O predicado pre-

dicado `method_return ?m - t_method ?rtm - t_return_type` indica o tipo de retorno rt_m do método m.

A verificação das informações dos métodos é modelada em dois predicados. O predicado `checked_argument ?arg - t_argument ?m - t_method` indica que o argumento *arg* do método m já foi verificado. O predicado `checked_argument_information ?m - t_method` indica que todas as informações do método m foram checadas, permitindo assim a integração do método.

Os predicados específicos do teste interclasses são apresentados entre as linhas 20 e 30 do Código 4.13. O predicado `cpriority ?prc - t_cpriority` representa a prioridade de integração de classes pr_c considerada no momento atual do teste. O predicado `class_priority ?c - t_class ?prc - t_cpriority` associa a prioridade pr_c a uma classe c. A atribuição de classes c a testadores t, representada pelos predicados `done_all ?c - t_class` e `done_tester ?c - t_class ?t - t_tester`, segue a lógica definida no contexto dos métodos.

O predicado `integrated_class ?c - t_class` indica que a classe c foi integrada e um teste foi executado após sua integração. O predicado `class_attribute ?c - t_class ?atr - t_attribute` associa o atributo *atr* à classe c. Os tipos *at* de cada atributo *atr* são representados pelo predicado `attribute_type ?atr - t_attribute ?at - t_attribute_type`.

Os demais predicados modelam a checagem das informações das classes c. O predicado `checked_attribute ?atr - t_attribute ?c - t_class` indica que o atributo *atr* foi verificado e o predicado `checked_method ?m - t_method ?c - t_class` indica que o método m foi verificado. A consolidação da checagem de atributos é feita pelo predicado `checked_attribute_information ?c - t_class`. O predicado `checked_method_information ?c - t_class` consolida a checagem dos métodos.

As valorações numéricas são expressas pelo conjunto F de funções apresentado no Código 4.14. Dois predicados modelam as capacidades de teste: a função `capacity ?t - t_tester` indica a capacidade de teste individual do testador t e a função `max_capacity` indica a capacidade máxima de teste suportada em um *sprint*. A função *sum* armazena a soma dos pesos da prioridades de integração.

Código 4.14: Funções do domínio PDDL do teste de integração OO intermétodos e interclasses.

```
1  (:functions
2    (capacity ?t - t_tester)
3    (max_capacity)
4    (sum)
5
6    ;intermétodos
7    (method_size ?m - t_method)
8    (mpriority_weight ?prm - t_mpriority)
9
10   ;interclasses
11   (class_size ?c - t_class)
12   (cpriority_weight ?prc - t_cpriority)
13 )
```

As demais funções são específicas conforme o tipo de teste. No teste intermétodos, a função `method_size ?m - t_method` indica o tamanho do método m e a função `mpriority_weight ?prm - t_mpriority` determina o peso da prioridade pr_m. No teste interclasses, o tamanho da classe c é indicado na função

`class_size ?c - t_class` e o preso da prioridade pr_c é indicado pelo predicado `cpriority_weight ?prc - t_cpriority`.

Além desses elementos iniciais, o domínio PDDL contém a declaração do conjunto A de ações. O domínio do teste intermétodos é composto por oito ações, enquanto o domínio do teste interclasses é estruturado em nove ações.

4.4.3.2 Ações para checar argumentos dos métodos

A ação `METHOD_ARGUMENT` representa a verificação de um argumento específico de um método. O Código 4.15 apresenta a representação dessa ação em PDDL. Os parâmetros dessa ação são um método m, um argumento arg e um tipo de argumento $argt$. As pré-condições da ação `METHOD_ARGUMENT` são:

- o método m possui o argumento arg (linha 5, Código 4.15);
- o argumento arg tem o tipo $argt$ (linha 6, Código 4.15); e
- o argumento arg contido no método m ainda não foi verificado (linha 7, Código 4.15).

Se as pré-condições são satisfeitas, a execução da ação gera como efeito:

- o argumento arg é marcado como verificado para o método m (linha 10, Código 4.15).

Código 4.15: Ação METHOD_ARGUMENT do domínio PDDL do teste de integração OO intermétodos.

```
1  (:action METHOD_ARGUMENT
2   :parameters (?m - t_method ?arg - t_argument ?argt - t_argument_type)
3
4   :precondition (and
5                   (method_argument ?m ?argt)
6                   (arg_type ?arg ?at)
7                   (not (checked_argument ?arg ?m))
8                 )
9
10  :effect (checked_argument ?arg ?m)
11 )
```

A ação `CHECKED_ARGUMENT_INFORMATION` representa a verificação completa dos argumentos de um método. A representação em PDDL dessa ação é descrita no Código 4.16. A ação `CHECKED_ARGUMENT_INFORMATION` tem como parâmetro um método m e suas pré-condições são:

- o método m ainda não foi integrado (linha 5, Código 4.16);
- as informações dos argumentos do método m ainda não foram verificadas (linha 6, Código 4.16); e
- para todos os parâmetros arg, pelo menos uma das condições é atendida: o método m não possui o argumento arg ou o argumento arg já foi verificado para o método m (linhas 7 a 10, Código 4.16).

Se essas pré-condições são atendidas, a ação é executada gerando como efeito:

- o método *m* é marcado como tendo suas informações verificadas (linha 13, Código 4.16).

Código 4.16: Ação CHECKED_ARGUMENT_INFORMATION do domínio PDDL do teste de integração OO intermétodos.

```
1  (:action CHECKED_ARGUMENT_INFORMATION
2   :parameters (?m - t_method)
3
4   :precondition (and
5                   (not (integrated_method ?m))
6                   (not (checked_argument_information ?m))
7                   (forall (?arg - t_argument)
8                    (or
9                     (not (method_argument ?m ?arg))
10                    (checked_argument ?arg ?m)))
11                )
12
13  :effect (checked_argument_information ?m)
14 )
```

Essas ações fazem parte de um modelo que busca assegurar que os métodos passem por uma verificação antes de serem considerados prontos para a integração no teste intermétodos. Primeiro, cada argumento de um método é verificado individualmente por `METHOD_ARGUMENT`. Depois, a ação `CHECKED_ARGUMENT_INFORMATION` certifica que todos os argumentos foram analisados antes de marcar as informações do método como verificadas. Em conjunto, essas ações asseguram que ocorra a integração apenas com métodos com argumentos verificados.

4.4.3.3 Ações para checar métodos das classes

A ação `CLASS_METHOD` representa a verificação de um método específico em uma classe. O Código 4.17 apresenta essa ação em PDDL. Os parâmetros da ação `CLASS_METHOD` são uma classe *c* e um método *m* e suas pré-condições são:

- o método *m* está associado à classe *c* (linha 5, Código 4.17); e
- o método *m* ainda não foi verificado para a classe *c* (linha 6, Código 4.17).

A execução dessa ação gera como efeito:

- o método *m* é indicado como verificado para a classe *c* (linha 9, Código 4.17).

Código 4.17: Ação CLASS_METHOD do domínio PDDL do teste de integração OO interclasses.

```
1  (:action CLASS_METHOD
2   :parameters (?c - t_class ?m - t_method)
3
4   :precondition (and
5                   (class_method ?m ?c)
6                   (not (checked_method ?m ?c))
7                 )
8
9   :effect (checked_method ?m ?c)
10 )
```

A ação CHECKED_METHOD_INFORMATION é responsável por consolidar a verificação de todos os métodos de uma classe específica. O Código 4.18 descreve a representação em PDDL dessa ação. A ação CHECKED_METHOD_INFORMATION tem como parâmetro uma classe *c* e suas pré-condições são:

- a classe *c* ainda não foi integrada (linha 5, Código 4.18);
- as informações dos métodos da classe *c* ainda não foram verificadas (linha 6, Código 4.18); e
- para todos os métodos *m*, pelo menos uma das seguintes condições deve ser verdadeira: o método *m* não está associado à classe *c* ou o método *m* já foi verificado para a classe *c* (linhas 7 a 10, Código 4.18).

Caso essas pré-condições sejam satisfeitas, a execução dessa ação tem como efeito:

- as informações dos métodos da classe *c* são marcadas como verificadas (linha 13, Código 4.18).

Código 4.18: Ação CHECKED_METHOD_INFORMATION do domínio PDDL do teste de integração OO interclasses.

```
1  (:action CHECKED_METHOD_INFORMATION
2    :parameters (?c - t_class)
3
4    :precondition (and
5                    (not (integrated_class ?c))
6                    (not (checked_method_information ?c))
7                    (forall (?m - t_method)
8                      (or
9                        (not (class_method ?m ?c))
10                       (checked_method ?m ?c)))
11                 )
12
13   :effect (checked_method_information ?c)
14 )
```

As ações CLASS_METHOD e CHECKED_METHOD_INFORMATION modelam um processo de verificação de métodos dentro de uma classe no contexto do teste interclasses. Essas ações promovem, respectivamente, verificações individuais dos métodos e a consolidação dessas verificações. A execução dessas ações garante que apenas classes cujos métodos foram verificados sejam integradas.

4.4.3.4 Ações para checar atributos das classes

A ação CLASS_ATTRIBUTE representa a verificação de um atributo específico de uma classe. A representação em PDDL dessa ação é apresentada no Código 4.19. Os parâmetros da ação CLASS_ATTRIBUTE são uma classe *c*, um atributo *atr* e um tipo de atributo *at*. As pré-condições dessa ação são:

- o atributo *atr* pertence à classe *c* (linha 5, Código 4.19);
- o atributo *atr* tem um tipo *at* (linha 6, Código 4.19); e

- o atributo *atr* ainda não foi verificado para a classe *c* (linha 7, Código 4.19).

Se essas pré-condições são satisfeitas, a ação é executada gerando como efeito:

- o atributo *atr* é marcado como verificado para a classe *c*.

Código 4.19: Ação CLASS_ATTRIBUTE do domínio PDDL do teste de integração OO interclasses.

```
1  (:action CLASS_ATTRIBUTE
2   :parameters (?c - t_class ?atr - t_attribute ?at - t_attribute_type)
3
4   :precondition (and
5                   (class_attribute ?c ?atr)
6                   (attribute_type ?atr ?at)
7                   (not (checked_attribute ?atr ?c))
8                 )
9
10  :effect (checked_attribute ?atr ?c)
11 )
```

A ação CHECKED_ATTRIBUTE_INFORMATION representa a verificação completa das informações de atributos de uma classe. Essa ação tem como parâmetro uma classe *c* e suas pré-condições são:

- a classe *c* ainda não foi integrada (linha 5, Código 4.20);
- as informações dos atributos da classe *c* ainda não foram verificadas (linha 6, Código 4.20); e
- para todos os atributos *atr*, pelo menos uma das seguintes condições deve ser atendida: a classe *c* não tem o atributo *atr* ou o atributo *atr* já foi verificado para a classe *c* (linhas 7 a 10, Código 4.20).

Caso essas pré-condições sejam satisfeitas, a execução da ação gera como efeito:

- as informações dos atributos da classe *c* são marcadas como verificadas (linha 13, Código 4.20).

Código 4.20: Ação CHECKED_ATTRIBUTE_INFORMATION do domínio PDDL do teste de integração OO interclasses.

```
1  (:action CHECKED_ATTRIBUTE_INFORMATION
2   :parameters (?c - t_class)
3
4   :precondition (and
5                   (not (integrated_class ?c))
6                   (not (checked_attribute_information ?c))
7                   (forall (?atr - t_attribute)
8                     (or
9                       (not (class_attribute ?c ?atr))
10                      (checked_attribute ?atr ?c)))
11                )
12
13  :effect (checked_attribute_information ?c)
14 )
```

As ações CLASS_ATTRIBUTE e CHECKED_ATTRIBUTE_INFORMATION estabelecem um fluxo para a verificação de atributos dentro de uma classe antes de sua integração no teste interclasses. Essas ações permitem a análise individual de atributos e a indicação da finalização da análise de atributos, respectivamente. Em conjunto, essas ações garantem que apenas classes com os atributos verificados sejam integradas.

4.4.3.5 Ações para integrar métodos e classes e testar subsistemas

A ação METHOD_INTEGRATION_AND_TEST atribui a integração de um método e o teste após a integração a um testador. Assim, essa ação é contextualizada no teste intermétodos. O Código 4.21 detalha a representação em PDDL da ação METHOD_INTEGRATION_AND_TEST. Essa ação possui como parâmetros um método m, uma prioridade de integração de métodos pr_m, um testador t, uma classe c e um tipo de retorno rt_m.

Código 4.21: Ação METHOD_INTEGRATION_AND_TEST do domínio PDDL do teste de integração OO intermétodos.

```
1  (:action METHOD_INTEGRATION_AND_TEST
2   :parameters (?m - t_method ?prm - t_mpriority ?t - t_tester
3                ?c - t_class ?rtm - t_return_type)
4
5   :precondition (and
6                   (mpriority ?prm)
7                   (tester ?t)
8                   (class ?c)
9                   (method_priority ?m ?prm)
10                  (class_method ?m ?c)
11                  (method_return ?m ?rtm)
12                  (>= (capacity ?t) (method_size ?m))
13                  (not (integrated_method ?m))
14                  (checked_argument_information ?m)
15                  (forall (?tester - t_tester)
16                    (not (done_tester ?m ?tester)))
17                 )
18
19   :effect (and
20             (done_tester ?m ?t)
21             (decrease (capacity ?t) (method_size ?m))
22             (increase (sum) (mpriority_weight ?prm))
23             (not (mpriority ?prm))
24             (mpriority PRM1)
25             (integrated_method ?m)
26             (when (tester tester1)
27               (and
28                 (not (tester tester1))
29                 (tester tester2)))
30             (when (tester tester2)
31               (and
32                 (not (tester tester2))
33                 (tester tester1)))
34           )
35  )
```

As pré-condições da ação METHOD_INTEGRATION_AND_TEST são:

- o predicado mpriority ?prm deve ser verdadeiro para a prioridade de integração pr_m (linha 6, Código 4.21);

- o predicado `tester ?t` deve ser verdadeiro para o testador *t* (linha 7, Código 4.21);
- o predicado `class ?c` deve ser verdadeiro para a classe *c* (linha 8, Código 4.21);
- o método *m* possui a prioridade de integração pr_m (linha 9, Código 4.21);
- o método *m* pertence à classe *c* (linha 10, Código 4.21);
- o método *m* tem um tipo de retorno rt_m (linha 11, Código 4.21);
- a capacidade do testador *t* deve ser menor ou igual ao tamanho do método *m* (linha 12, Código 4.21);
- o método *m* ainda não foi integrado (linha 13, Código 4.21);
- as informações dos argumento do método *m* já foram verificadas (linha 14, Código 4.21); e
- o método *m* não foi atribuído a nenhum testador *t* no *sprint* atual (linhas 15 e 16, Código 4.21).

Os efeitos da execução da ação `METHOD_INTEGRATION_AND_TEST` são:

- o método *m* é indicado como atribuído ao testador *t* (linha 20, Código 4.21);
- a capacidade do testador *t* é decrementada com o tamanho do método *m* (linha 21, Código 4.21);
- a função `sum` é incrementada com o peso da prioridade pr_m (linha 22, Código 4.21);
- a prioridade pr_m é atualizada para a prioridade mais alta (linhas 23 e 24, Código 4.21);
- o método *m* é indicado como integrado (linha 25, Código 4.21); e
- a atribuição de métodos é alternada entre os testadores disponíveis (linhas 26 a 33, Código 4.21).

A ação `CLASS_INTEGRATION_AND_TEST` é responsável por atribuir a integração de uma classe e a realização subsequente do teste a um testador. Essa ação é inserida em cenários de teste interclasses. O Código 4.22 descreve a representação em PDDL dessa ação. A ação `CLASS_INTEGRATION_AND_TEST` tem como parâmetros uma classe *c*, uma prioridade de integração pr_c e um testador *t* e suas pré-condições são:

- o predicado `cpriority ?prc` deve ser verdadeiro para a prioridade pr_c (linha 5, Código 4.22);
- o predicado `tester ?t` deve ser verdadeiro para o testador *t* (linha 6, Código 4.22);
- a classe *c* tem prioridade pr_c (linha 7, Código 4.22);
- a capacidade do testador *t* deve ser menor ou igual ao tamanho da classe *c* (linha 8, Código 4.22);
- a classe *c* ainda não foi integrada (linha 9, Código 4.22);

- as informações dos atributos da classe c foram verificadas (linha 10, Código 4.22);
- as informações dos métodos da classe c foram verificadas (linha 11, Código 4.22); e
- a classe c ainda não foi atribuída a nenhum testador t no *sprint* atual (linhas 12 e 13, Código 4.22).

A ação `CLASS_INTEGRATION_AND_TEST` tem como efeitos:

- a classe c é atribuída ao testador t (linha 17, Código 4.22);
- a capacidade do testador t é reduzida com valor do tamanho da classe c (linha 18, Código 4.22);
- o peso da prioridade pr_c é acumulado na função `sum` (linha 19, Código 4.22);
- a prioridade pr_c é atualizada com a prioridade mais alta (linhas 20 e 21, Código 4.22);
- a classe c é indicada como integrada (linha 22, Código 4.22); e
- a atribuição de classes é alternada entre os testadores disponíveis (linhas 23 a 30, Código 4.22).

Código 4.22: Ação CLASS_INTEGRATION_AND_TEST do domínio PDDL do teste de integração OO interclasses.

```
1  (:action CLASS_INTEGRATION_AND_TEST
2    :parameters (?c - t_class ?prc - t_cpriority ?t - t_tester)
3
4    :precondition (and
5                    (cpriority ?prc)
6                    (tester ?t)
7                    (class_priority ?c ?prc)
8                    (>= (capacity ?t) (class_size ?c))
9                    (not (integrated_class ?c))
10                   (checked_attribute_information ?c)
11                   (checked_method_information ?c)
12                   (forall (?tester - t_tester)
13                     (not (done_tester ?c ?tester)))
14                 )
15
16   :effect (and
17             (done_tester ?c ?t)
18             (decrease (capacity ?t) (class_size ?c))
19             (increase (sum) (cpriority_weight ?prc))
20             (not (cpriority ?prc))
21             (cpriority PRC1)
22             (integrated_class ?c)
23             (when (tester tester1)
24               (and
25                 (not (tester tester1))
26                 (tester tester2)))
27             (when (tester tester2)
28               (and
29                 (not (tester tester2))
30                 (tester tester1)))
31           )
32 )
```

O processo de integração para os testes intermétodos e interclasses são estruturados, respectivamente, pelas ações METHOD_INTEGRATION_AND_TEST e CLASS_INTEGRATION_AND_TEST. Essas ações asseguram que métodos e classes sejam integrados após a verificação completa de suas informações e que as atribuições a testadores respeitam as prioridades e as capacidades de teste do ambiente. Ainda, essas ações garantem que o balanceamento do teste seja alcançado alternando atribuições entre os testadores disponíveis.

4.4.3.6 Ação para trocar a classe

A ação CHANGE_CLASS representa a troca de contexto entre classes no teste de integração intermétodos. Essa ação garante que uma classe só seja considerada finalizada (ou seja, removida do conjunto de classes ativas) quando todos os seus métodos tiverem sido integrados. A partir disso, a ação CHANGE_CLASS realiza uma transição entre as classes no processo de integração. O Código 4.23 apresenta a representação dessa ação em PDDL.

Código 4.23: Ação CHANGE_CLASS do domínio PDDL do teste de integração OO intermétodos.

```
1  (:action CHANGE_CLASS
2   :parameters (?c - t_class)
3
4   :precondition (and
5                   (class ?c)
6                   (forall (?m - t_method)
7                     (or
8                       (not (class_method ?m ?c))
9                       (integrated_method ?m)))
10                )
11
12  :effect (and
13            (not (class ?c))
14            (when (class C1)
15              (and
16                (not (class C1))
17                (class C2)))
18            (when (class C2)
19              (and
20                (not (class C2))
21                (class C3)))
22            (forall (?prc - t_cpriority)
23              (not (cpriority ?prc)))
24            (cpriority PRC1)
25          )
26 )
```

A ação CHANGE_CLASS tem como parâmetro uma classe *c* e suas pré-condições são:

- o predicado `class ?c` deve ser verdadeiro para a classe *c* (linha 5, Código 4.23); e
- para todos os métodos *m*, pelo menos uma das seguintes condições deve ser atendida: o método *m* não pertence à classe *c* ou o método *m* já foi integrado (linhas 6 a 9, Código 4.23).

Se essas pré-condições são atendidas, a ação é executada tendo como efeitos:

- a classe *c* atual é negada (linha 13, Código 4.23);

- é realizada a troca para uma nova classe cujos métodos serão integrados (linhas 14 a 21, Código 4.23). O exemplo apresentado é um sistema contendo três classes (c_1, c_2 e c_3); e
- o predicado `cpriority` é atualizado para a prioridade mais alta (linhas 22 a 25, Código 4.23).

4.4.3.7 Outras ações

Além das ações descritas nesta subseção, o domínio PDDL para os testes intermétodos e interclasses possuem outras quatro ações. Essas ações são responsáveis por atualizar a prioridade de integração (`INCREASE_PRIORITY`), trocar o testador (`CHANGE_TESTER`), iniciar um novo *sprint* (`NEW_SPRINT`) e atualizar a capacidade de teste (`RESET_CAPACITY`). Tais ações seguem, respectivamente, as lógicas apresentadas nos Códigos 4.7, 4.8, 4.9, e 4.10. Conforme o tipo de teste, essas ações são ajustadas com as prioridades de integração de métodos $pr_m \in Pr_m$ ou as prioridades de integração de classes $pr_c \in Pr_c$ disponíveis.

4.4.4 Representação do problema PDDL

Esta subseção descreve o problema PDDL para o teste de integração OO. Nesse contexto, cada problema PDDL é definido por elementos de um sistema SUT_{oo} e um ambiente Am_{oo}. Os Códigos D.8 e D.11 apresentam instâncias do problema PDDL para os testes intermétodos e interclasses, respectivamente. Essas instâncias se referem ao sistema OO ilustrado na Figura 4.6.

Nesse problema PDDL, os objetos representam os métodos $m \in M$, as classes $c \in C$, os testadores $t \in T$, as prioridades de integração de métodos $pr_m \in Pr_m$, as prioridades de integração de classes $pr_c \in Pr_c$, os argumentos $arg \in Arg$, os tipos de argumentos $argt \in Argt$, os tipos de retorno $rt_m \in Rt_m$, os atributos $atr \in Atr$ e os tipos de atributos $at \in At$.

O estado inicial s_0 é caracterizado pelo momento que antecede o início do teste. No s_0, é feita a indicação das prioridades consideradas inicialmente (predicados `mpriority` e `cpriority`), o testador que iniciará o teste (predicado `tester`) e a classe que será inicialmente integrada (predicado `class`). A descrição do s_0 é finalizada com a inicialização das demais ações e funções (conjuntos A e F).

O estado final s_g equivale ao momento em que todos os métodos ou classes foram integrados. No teste intermétodos, esse momento é alcançado quando o valor da função `sum` atinge o valor da soma dos pesos w de todas as prioridades pr_m associadas aos métodos $m_i \in M$, com $0 < i \leq |M|$. No teste interclasses, s_g é alcançado quando a função `sum` atinge o valor da soma dos pesos w de todas as prioridades pr_c das classes $c_j \in C$, com $0 < j \leq |C|$.

O valor da função `sum` é incrementado a cada atribuição de método ou classe a um testador. Assim, o final do teste se aproxima a medida em que o valor de `sum` aumenta. Os Códigos D.9 e D.12 apresentam, respectivamente, os planos de IA para os testes intermétodos e interclasses gerados a partir da representação PDDL descrita nesta seção. Esses planos de IA foram obtidos com o planejador Metric-FF a partir das representações instanciadas com o SUT ilustrado na Figura 4.6.

4.5 CONSIDERAÇÕES DO CAPÍTULO

Este capítulo apresentou a abordagem TI-PIA delineada a partir de duas direções: a) aspectos que motivam a aplicação prática do planejamento em IA abordados nos Capítulos 1 e 2; e b) lacunas encontradas na literatura discutidas no Capítulo 3. O contexto apresentado na

Seção 4.1 descreve um cenário de teste de integração que possui elementos que caracterizam um problema complexo, justificando o uso do planejamento em IA da seguinte forma:

- **grande número de variáveis e parâmetros**: o contexto abrange sistemas procedimentais e OO. O número de unidades desses sistemas pode aumentar proporcionalmente conforme o escopo do projeto. Assim, o teste pode envolver muitos parâmetros de entrada e dados de saída, ambos com tipos variados de dados;

- **vários objetivos**: o contexto associa objetivos aos *sprints*, aos paradigmas de programação e às prioridades de integração. Cada *sprint* tem como propósito testar as unidades implementadas no ciclo atual e preparar o SUT para a próxima iteração do desenvolvimento. Além disso, os paradigmas de programação determinam tipos de integração com objetivos distintos. Por fim, a prioridade de integração pode influenciar o objetivo originalmente estabelecido pelo tipo de integração adotado no teste;

- **múltiplas restrições**: o contexto estabelece que a execução do teste de integração depende diretamente dos resultados das fases iniciais de desenvolvimento e dos testes de unidade. Além disso, o tipo de integração adotado determina o ponto de partida e a sequência subsequente do processo. Essa sequência também pode ser influenciada pela prioridade de integração. Adicionalmente, a execução do teste está sujeita a restrições temporais e à capacidade de teste disponível em cada *sprint*;

- **necessidade de otimização de recursos**: o contexto considera a execução do teste de integração em *sprints* com recursos limitados, tanto em termos de infraestrutura quanto de equipe. Para otimizar o uso desses recursos, são estabelecidas as capacidades de teste dos testadores. Tais capacidades definem um limite máximo de testes que pode ser conduzido em cada *sprint*;

- **alto risco e custos elevados**: o contexto envolve sistemas procedimentais e OO que não são direcionados a um domínio de aplicação específico. No entanto, certos domínios, como saúde e segurança, podem apresentar alto risco e demandar um custo elevado para desenvolvimento e teste;

- **atividades conjuntas com outras pessoas**: o cenário determina o trabalho paralelo de gerente de teste e vários testadores durante cada *sprint*. Além disso, o teste de integração é entendido como um processo intermediário, então é associado aos papeis envolvidos em atividades que precedem e sucedem esses testes como desenvolvedores, equipes de teste e usuários; e

- **atividades sincronizadas**: o contexto considera a realização de testes simultâneos por múltiplos testadores em cada *sprint*.

Após a delimitação do cenário, a abordagem TI-PIA foi definida conforme a descrição da Seção 4.2. As lacunas da literatura destacadas na Seção 3.4 basearam a definição da estrutura de geração de planos de integração e da abordagem. As lacunas la_1, la_2, la_3, la_7, la_{10} e la_{11} fundamentaram a definição da estrutura descrita na Seção 4.2, enquanto as lacunas la_2, la_4, la_5, la_6, la_8, la_9 e la_{12} embasaram a definição das modelagens apresentadas nas Seções 4.3 e 4.4.

A estrutura incorpora o plano de teste (la_1) e o plano de integração (la_3) como elementos essenciais para a realização do teste. Dessa forma, seus principais usuários são os gerentes de teste e os testadores (la_2). Além disso, a estrutura define formatos para os arquivos de entrada e saída, como XML e texto simples (la_{10}). Foram incluídas atividades específicas para o tratamento

de arquivos, incluindo a eliminação de ações usadas como *log* e a representação dos planos de integração em esquemas gráficos (la_{11}). A estrutura indica um repositório que fornece *stubs* e *drivers* necessários para o teste (la_7).

Além do gerenciamento de arquivos, a estrutura estabelece um mapeamento entre os elementos do teste e os elementos PDDL utilizados para instanciar os arquivos PDDL. Esse mapeamento possibilita a definição de uma atividade que organiza essa correspondência de forma estruturada. Além disso, a estrutura especifica como o plano de IA é ajustado para compor o plano de integração, removendo ações usadas apenas para *log* do planejador e que não são necessárias para a execução do teste.

A modelagem inclui a representação dos testadores (la_2) e foi elaborada com o objetivo de explicitar os diferentes tipos de integração conforme o tipo do SUT (la_4). Nesse sentido, a modelagem é adaptável tanto para as estratégias *top-down* quanto *bottom-up* (la_5), além de suportar testes intermétodos e interclasses (la_6). Assim, a modelagem também é ajustável para sistemas procedimentais e OO (la_9), incorporando elementos específicos desses sistemas, como parâmetros, chamadas de operações, atributos, trocas de mensagens e tipos de dados (la_8).

A modelagem contém elementos que representam os *sprints* que podem ser adaptados para um método ágil de desenvolvimento (la_{12}). As características do contexto foram representadas por seis funções numéricas. Nesse sentido, entende-se que as funcionalidades da PDDL2.1 são essenciais para cobrir as lacunas encontradas e garantir a completude da representação do teste. Assim, o uso dessa versão da linguagem se justifica, mesmo envolvendo problemas com maior complexidade computacional para a geração de planos de IA em comparação ao planejamento clássico.

As modelagens permitem o ajuste do número de testadores e dos tipos de integração. Essa característica proporciona ao usuário maior flexibilidade para realizar adaptações conforme o projeto em desenvolvimento. Além disso, alguns elementos do contexto aproximam a abordagem do uso prático na indústria, como o teste OO, a divisão de ambientes e a possibilidade de reutilização de unidades na integração.

A abordagem foi contextualizada com o desenvolvimento de unidades que implementam requisitos funcionais. Características de outras versões da PDDL, como tempo e não determinismo, podem contribuir para o teste de requisitos não funcionais, como desempenho e segurança. Ao mapear requisitos funcionais para unidades, a abordagem também considera o desenvolvimento de unidades voltadas para reutilização.

A separação de responsabilidades entre o planejamento e a execução do teste sugere que a abordagem pode contribuir para a rastreabilidade do teste. Além disso, a distribuição entre múltiplos testadores pode ser ampliada para múltiplas equipes de teste, permite que a abordagem seja direcionada para o teste de sistema, como segurança e usabilidade. Por fim, as modelagens definidas para a abordagem podem se enquadrar em outras estruturas da literatura, como a baseada em componentes apresentada na Subseção 3.2.4.

Diante do exposto, observa-se que as representações PDDL, os planos de IA e os planos de integração são os elementos centrais da abordagem TI-PIA. Os módulos da estrutura objetivam instanciar os arquivos PDDL, gerar os planos de IA e definir os plano de integração com base nas informações obtidas. Diante disso, o próximo capítulo apresenta avaliações focadas nesses elementos.

5 AVALIAÇÃO DA ABORDAGEM

Este capítulo apresenta avaliações que estudaram a viabilidade das modelagens do teste de integração com planejamento em IA e dos planos de integração da abordagem TI-PIA. Assim, o foco da avaliação é a parte da abordagem destacada em cinza na Figura 4.2. A partir da metodologia da Seção 5.1, são apresentados os três estudos de viabilidade que avaliaram a abordagem (Seções 5.2, 5.3 e 5.4) e as ameaças à validade dos mesmos (Seção 5.5).

A Tabela 5.1 sumariza os estudos com a estrutura "Introdução, Método, Resultados e Discussão" (IMRaD, do inglês "*Introduction, Methods, Results, and Discussion*") (Wu, 2011). Para cada estudo, são respondidas as seguintes perguntas: Introdução) "*por que o estudo foi realizado?*"; Métodos) "*como o estudo foi realizado?*" e "*quem realizou o estudo?*"; Resultados) "*o quê foi encontrado no estudo?*"; e Discussão) "*e então, o quê os resultados significam?*".

Tabela 5.1: Sumarização dos estudos de viabilidade com a estrutura IMRaD.

Estudo	Por quê?	Como?	Quem?	O quê?	E então?
1) Estudo 1, Seção 5.2	Para avaliar a viabilidade de características da representação PDDL do teste procedimental descrita na Seção 4.3	Um estudo de viabilidade com quatro cenários que avaliaram separadamente as características: a) atribuição das unidades; b) capacidades de teste dos testadores e dos *sprints*; c) balanceamento da atribuição dos procedimentos; e d) as chamadas de operações	Autor (ponto de vista de um gerente de teste)	**Pontos fortes**: a representação PDDL se mostrou viável para a geração de planos de IA factíveis nos cenários avaliados. **Pontos fracos**: a falta de informações específicas dos SUTs na representação PDDL podem dificultar o uso dos planos de integração na execução do teste	Foi verificado que os planos de integração, mesmo sem as informações dos procedimentos, podem auxiliar o gerenciamento do teste. O Estudo 2 foi definido para investigar a representação PDDL incluindo essas informações.
2) Estudo 2, Seção 5.3	Para avaliar a viabilidade da instanciação da representação PDDL do teste procedimental descrita na Seção 4.3 e da representação PDDL do teste OO descrita na Seção 4.4 usando projetos de desenvolvimento	Um estudo de viabilidade com quatro cenários que avaliou a instanciação das representações PDDL com informações de projetos de um sistema procedimental e de um sistema OO	Autor (ponto de vista de um gerente de teste)	**Pontos fortes**: as representações PDDL geraram planos de IA factíveis para os testes *top-down*, *bottom-up*, intermétodos e interclasses. **Pontos fracos**: a performance da geração dos planos de IA é escalável conforme o número de objetos instanciados	Foi verificado que os planos de integração contendo as informações das unidades são mais legíveis e auxiliam a operacionalização do teste. Foi indicada a possibilidade de gerar planos de integração para subsistemas para lidar com a escalabilidade do problema.
3) Estudo 3, Seção 5.4	Para avaliar a viabilidade da aplicação da representação PDDL do teste procedimental descrita na Seção 4.3 na manutenção de software	Um estudo de viabilidade com três cenários que avaliou a representação PDDL na manutenção de software decorrente de alterações, inclusões e remoções de unidades	Autor (ponto de vista de um gerente de teste)	**Pontos fortes**: a representação PDDL gerou planos de IA factíveis para diferentes contextos de manutenção de software. Os planos de integração podem ser adaptados processos que ocorrem após a implantação do SUT **Pontos fracos**: essa adaptação dos planos de integração pode requer ajustes na representação PDDL e ferramentas adicionais	Foi verificado que o uso dos planos de integração pode se adequar às atividades de manutenção de software e evolução de software

Os materiais utilizados e gerados nos estudos apresentados neste capítulo estão disponíveis em um repositório do Zenodo[1] (*link*: `https://zenodo.org/records/14920065`).

[1]Zenodo é um repositório digital de acesso aberto que permite o armazenamento e o compartilhamento de materiais associados a um identificador digital único e a uma determinada licença. Disponível em: `https://zenodo.org`

5.1 METODOLOGIA

Esta seção descreve a metodologia usada na avaliação da abordagem por meio de estudos de viabilidade. Um **estudo de viabilidade** é um estudo experimental que avalia uma nova tecnologia para refinar a solução e gerar hipóteses associadas ao seu uso e não necessariamente obter respostas definitivas (Strauss e Corbin, 1998). Em conjunto, os estudos apresentados neste capítulo têm como **objetivo geral** avaliar a viabilidade das representações PDDL e os planos de integração da abordagem TI-PIA. Assim, são **objetivos específicos da avaliação**:

a) Avaliar as representações PDDL em diferentes cenários;

b) Discutir a qualidade dos planos de integração gerados; e

c) Investigar a utilização dos planos de integração em contextos da ES.

Diante desses objetivos, o **objeto** principal de cada estudo são as representações PDDL de teste de integração definidas nas Seções 4.3 e 4.4. Os **materiais** dos estudos incluem os artefatos e o planejador da abordagem TI-PIA. Os artefatos são os arquivos de domínio e problema PDDL instanciados com as características dos cenários delimitados, os planos de IA gerados pelo planejador e os planos de integração. O planejador utilizado é o Metric-FF (Hoffmann, 2003a,b).

Ainda como materiais dos estudos, foram utilizados simulações de projetos e projetos de desenvolvimento de sistemas procedimentais e sistemas OO. As simulações foram obtidas em Myers et al. (2011). Os projetos foram desenvolvidos pelo autor em disciplinas do curso de Bacharelado em Ciência da Computação (BCC) da UFPR. As representações gráficas dos planos de integração e das estruturas dos projetos foram feitas na ferramenta de criação de diagramas draw.io (disponível em: `https://draw.io`).

As execuções do Metric-FF foram realizadas com a configuração padrão desse planejador em um **ambiente** que consiste em um computador Intel Core i5-2400 CPU @ 3,10 GHz × 4 com 16 GB de memória e sistema operacional LMDE 5 Elsie Cinnamon 5.6.8. O tempo de execução é indicado em segundos e foi obtido com o comando *time*. Os **métodos** utilizados para a realização de cada estudo consistem na definição do objetivo, das etapas e das atividades de cada etapa.

O **objetivo** de cada estudo foi estruturado com o *Goal-Question-Metric* (GQM) (Basili e Rombach, 1988). Os objetivos são apresentados em suas respectivas seções. Cada estudo foi conduzido em **etapas** baseadas na metodologia para experimentos em ES proposta por Wohlin et al. (2000). As etapas definidas são: 1) Preparação, etapa para definição e organização dos cenários a serem avaliados; 2) Execução, etapa que coleta os dados para análise; e 3) Discussão, etapa que analisa os resultados que pode combinar métodos quantitativos e qualitativos.

A partir das etapas definidas, os estudos foram divididos em quatro **atividades** indicadas na Figura 5.1. Os objetivos de cada atividade são apresentados a seguir.

- **Atividade i) Definição dos cenários de teste**: esta atividade objetiva definir cenários de teste contendo sistemas procedimentais, sistemas OO e ambientes com características variadas. Os cenários contêm unidades com arranjos variados de tamanhos e prioridades de integração. Conforme o aspecto avaliado em cada estudo, características dos ambientes de cada cenário foram adaptadas;

- **Atividade ii) Instanciação da representação PDDL**: esta atividade objetiva instanciar as representações PDDL descritas nas Seções 4.3 e 4.4 usando as informações dos cenários de teste definidos na atividade (i). As representações PDDL foram ajustadas com a remoção de predicados ou ações conforme as características dos cenários e os objetivos dos estudos;

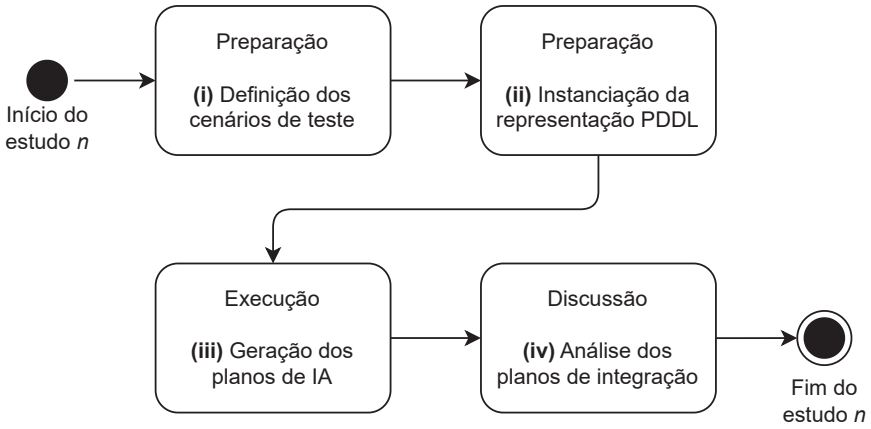


Figura 5.1: Etapas e atividades dos estudos de viabilidade.

- **Atividade iii) Geração dos planos de IA**: esta atividade objetiva gerar planos de IA com o planejador Metric-FF utilizando como entrada os arquivos PDDL modelados para cada cenário na atividade (ii); e
- **Atividade iv) Análise dos planos de integração**: esta atividade objetiva analisar os esquemas dos planos de integração gerados a partir dos planos da atividade (iii). Essa análise busca verificar se os planos indicam uma sequência adequada de integração de unidades de acordo com as características de cada cenário de teste, discutir a qualidade dos planos, debater a aplicabilidade dos planos, identificar as limitações da representação PDDL e indicar possibilidades de extensões e melhorias para as representações.

5.2 ESTUDO 1: AVALIAÇÃO DE ELEMENTOS DAS MODELAGENS

O Estudo 1 busca avaliar a viabilidade de elementos contidos na lógica da representação PDDL do teste procedimental apresentada na Seção 4.3. A Tabela 5.2 apresenta o objetivo desse estudo com o GQM. As próximas subseções detalham o Estudo 1 a partir das etapas e atividades indicadas na Figura 5.1.

Tabela 5.2: Objetivo do Estudo 1 estruturado com o GQM.

Analisar	planos de integração gerados a partir de uma simplificação da representação PDDL apresentada na Seção 4.3
Com o propósito de	avaliar
Em relação a	viabilidade de representação das características: atribuição de unidades; capacidades de teste dos testadores e dos *sprints*; balanceamento de atribuição do teste; e chamadas de operações entre unidades
Do ponto de vista de	gerente de teste
No contexto de	planejamento do teste de integração

5.2.1 Preparação e execução do Estudo 1

- **Atividade i) Definição dos cenários de teste**: foram definidos quatro cenários de teste para avaliar separadamente as características da representação PDDL do teste procedimental apresentada na Seção 4.3. Para isso, foi considerada uma versão simplificada dessa representação

para destacar as características do estudo. A Figura 5.2 ilustra as características do SUT procedimental considerado neste estudo.

O SUT ilustrado na Figura 5.2 é uma estrutura hierárquica de controle descrita por Myers et al. (2011). Esse sistema é composto por doze procedimentos que realizam doze chamadas de operações entre si. Para este estudo, a formalização desse sistema foi simplificada para $SUT_p = (Proc, Co)$, onde:

- $Proc = \{proc_1(size_1, pr), proc_2(size_2, pr), ..., proc_{12}(size_{12}, pr)\}$ é o conjunto de procedimentos e seus respectivos tamanhos e prioridades de integração; e
- $Co = \{(proc_1, proc_2), (proc_1, proc_3), ..., (proc_8, proc_{12})\}$ é o conjunto de chamadas de operações.

O ambiente de teste $Am_p = (T, Pr, TS_{max})$ foi caracterizado neste estudo como:

- $T = \{t_1(cap_1), t_2(cap_2), t_3(cap_3), t_4(cap_4)\}$ é o conjunto de testadores;
- $Pr = \{pr_0(w_0), pr_1(w_1), pr_2(w_2), ..., pr_{12}(w_{12})\}$ é o conjunto de prioridades de integração; e
- TS_{max} é a capacidade máxima de teste por *sprint*.

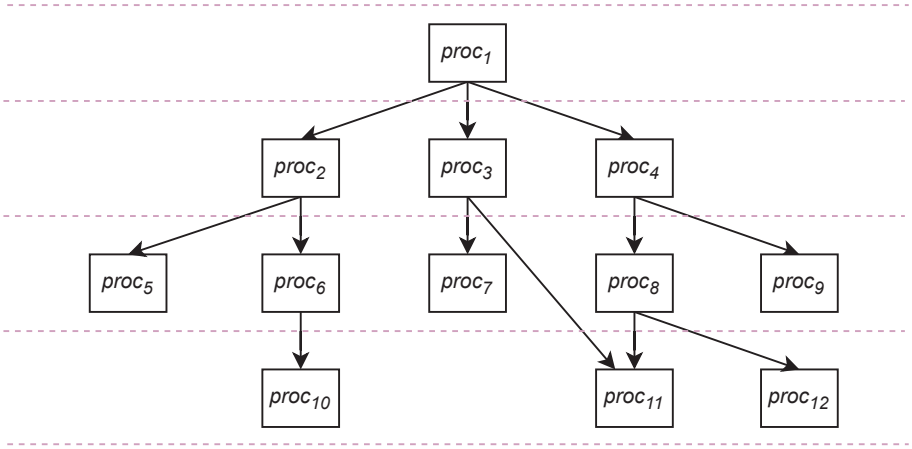


Figura 5.2: Características do SUT utilizado no Estudo 1. Fonte: Adaptado de Myers et al. (2011).

Para cada cenário de teste, os elementos do SUT_p da Figura 5.2 e do ambiente Am_p foram adaptadas conforme a característica avaliada. Essas adaptações incluem definições dos elementos: tamanhos $size_i$ dos procedimentos; prioridades de integração $pr_i \in Pr$; número de testadores de T ($|T|$); capacidades de teste cap_i; valor da capacidade máxima TS_{max}; e chamadas de operações Co. A Tabela 5.3 apresenta as instanciações desses elementos nos cenários de teste 1 a 4 (C1 a C4). Detalhes dessa instanciação são apresentados a seguir.

- **Atividade i) Definição do cenário de teste 1**: esse cenário avalia a atribuição dos procedimentos ao testador. Esse cenário contém os doze procedimentos do SUT da Figura 5.2 desconsiderando as chamadas de operações. Os procedimentos foram configurados com o mesmo tamanho ($size = 1$) e associadas a prioridades com valores crescentes (pr_1 a pr_{12}). O cenário foi ajustado para um único testador t_1, com capacidade de teste $cap_1 = 12$. A capacidade máxima dos *sprints* foi configurada em $TS_{max} = 12$. A Figura 5.3 ilustra a estrutura do SUT do cenário 1.

Elemento	C1 Atribuição de procedimentos	C2 Capacidade de teste	C3 Com balanceamento e sem balanceamento	C4 Procedimentos chamados e que chamam operações
$size_i$	Constante ($size_i = 1$)	Variável ($size_i = 3$ a $size_i = 12$)	Constante ($size_i = 1$)	Constante ($size_i = 1$)
pr_i	Variável (pr_1 a pr_{12})	Constante (pr_1)	Constante (pr_1)	Constante (pr_1)
$\|T\|$	$\|T\| = 1$	$\|T\| = 1$ e $\|T\| = 2$	$\|T\| = 4$	$\|T\| = 1$ e $\|T\| = 2$
cap_i	$cap_i = 12$	$cap_i = 12$	$cap_i = 12$	$cap_i = 3$
TS_{max}	$TS_{max} = 12$	$TS_{max} = 12$	$TS_{max} = 12$	$TS_{max} = 3$
Co	Desconsiderado	Desconsiderado	Desconsiderado	Considerado

Tabela 5.3: Instanciação dos elementos para os quatro cenários de teste do Estudo 1.

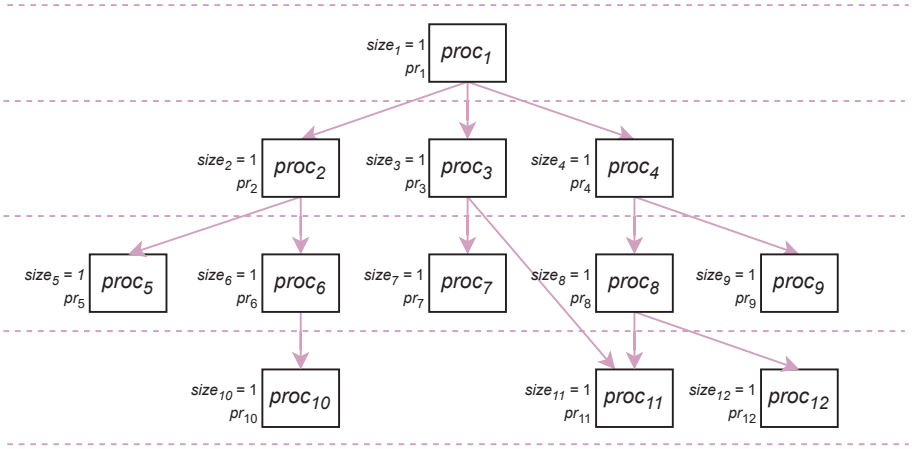


Figura 5.3: Cenário de teste 1 do Estudo 1: características do SUT.

• **Atividade i) Definição do cenário de teste 2**: esse cenário avalia a adequação da atribuição dos procedimentos conforme as capacidades de teste cap_i e TS_{max}. O SUT desse cenário consiste em doze procedimentos com a mesma prioridade (pr_1) desconsiderando as chamadas de operações. Os tamanhos $size_i$ foram atribuídos com valores entre 3 e 12. Esse cenário foi executado com um testador e com dois testadores t_1 e t_2, ambos com capacidade de teste $cap_i = 12$. O *sprint* foi configurado com $TS_{max} = 12$. A Figura 5.4 apresenta as características do SUT do cenário 2.

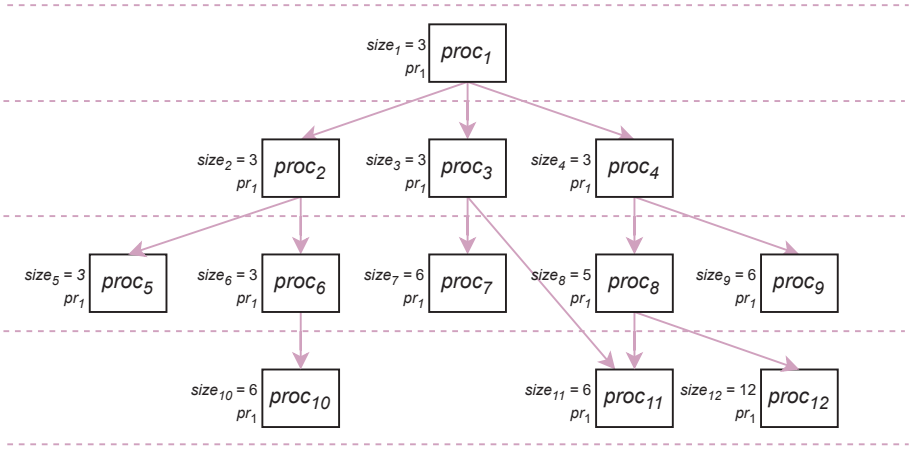


Figura 5.4: Cenário de teste 2 do Estudo 1: características do SUT.

• **Atividade i) Definição do cenário de teste 3**: esse cenário avalia o balanceamento da atribuição de unidades entre os testadores. Para isso, foi considerado um SUT com doze procedimentos, todos com o mesmo tamanho ($size_i = 1$) e a mesma prioridade pr_1. O cenário foi definido com *sprints* de $TS_{max} = 12$ e testadores com capacidade de teste $cap_i = 12$. Essa configuração foi ajustada para testes com um e com quatro testadores (t_1, t_2, t_3 e t_4). A estrutura do SUT do cenário 3 é ilustrada na Figura 5.5.

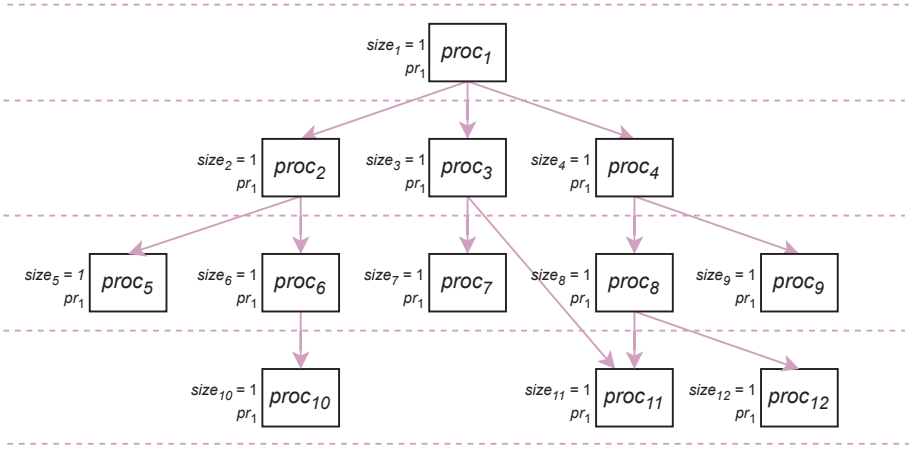


Figura 5.5: Cenário de teste 3 do Estudo 1: características do SUT.

• **Atividade i) Definição do cenário de teste 4**: esse cenário avalia as chamadas de operações Co entre os procedimentos. Esse cenário considera um SUT com doze procedimentos de tamanho $size_i = 1$ e prioridade pr_1. O cenário contém um ou dois testadores t_1 e t_2, ambos com capacidade de teste $cap_i = 3$, e *sprints* com $TS_{max} = 3$. Foi realizada a verificação dos procedimentos que chamam operações do procedimento em teste e a verificação dos procedimentos que são chamados pelo procedimento em teste. A Figura 5.6 apresenta o SUT do cenário 4.

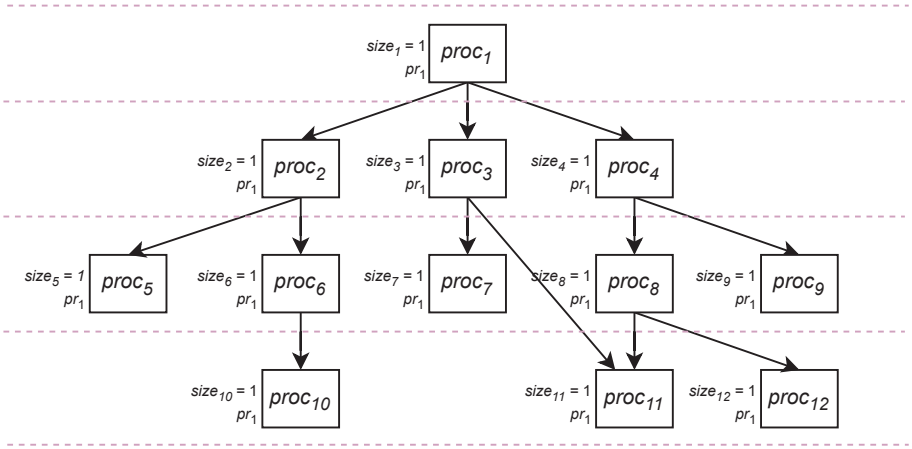


Figura 5.6: Cenário de teste 4 do Estudo 1: características do SUT.

• **Atividade ii) Instanciação da representação PDDL**: após as definições das características de cada cenário, as características dos SUTs e dos ambientes de teste foram utilizadas para instanciar a representação PDDL do teste procedimental.

• **Atividade iii) Geração dos planos de IA**: após a instanciação da representação PDDL, os arquivos do problema PDDL e do domínio PDDL de cada cenário foram usados como entrada no planejador Metric-FF para a geração dos planos de IA.

5.2.2 Resultados do Estudo 1

Os resultados encontrados durante a geração dos planos de IA foram tabulados e são encontrados na Tabela 5.4. Essa tabela indica as características de cada cenário (coluna "Características"), o tempo de execução do planejador (coluna "Tempo"), a quantidade de estados percorridos (coluna "Estados") e o número de ações do plano de IA (coluna "Ações"). O tempo é indicado em segundos (s).

Cenário	Características	Tempo	Estados	Ações
C1	Um testador	0,006s	50	35
C2	Um testador	0,004s	31	18
	Dois testadores	0,007s	30	16
C3	Quatro testadores, sem balanceamento	0,004s	13	12
	Quatro testadores, com balanceamento	0,006s	13	12
C4	Um testador, procedimentos chamados	0,005s	23	15
	Dois testadores, procedimentos chamados	0,009s	37	19
	Um testador, procedimentos que chamam	0,005s	24	15
	Dois testadores, procedimentos que chamam	0,008s	42	18

Tabela 5.4: Síntese dos resultados do Estudo 1.

A Tabela E.1 do Apêndice E amplia a apresentação dos resultados do Estudo 1 indicando um excerto do plano de IA (coluna "Plano de IA"). Para destacar a atribuição de unidades e o início dos *sprints*, as ações que indicam o incremento das prioridades, a troca de testadores e a reinicialização das capacidades foram omitidas na Tabela E.1.

No cenário 1, o planejador levou 0,006s para gerar um plano de IA com 35 ações a partir de 50 estados percorridos. O plano de integração resultante distribui as unidades para o Testador t_1 ao longo de três *sprints*. Essa distribuição foi realizada adequadamente para os tamanhos dos procedimentos e as capacidade de teste. Os procedimentos $proc_1$, $proc_2$, $proc_3$ e $proc_4$ foram atribuídos no *Sprint* #1, $proc_5$, $proc_6$, $proc_7$ e $proc_8$ no *Sprint* #2 e $proc_9$, $proc_{10}$, $proc_{11}$ e $proc_{12}$ no *Sprint* #3.

O Código 5.1 mostra um excerto do plano de IA obtido no cenário 1. A partir das ações desse plano, foi gerado de forma manual um esquema que representa graficamente o plano de integração correspondente. A Figura 5.7 apresenta o esquema para o cenário 1 a partir do plano de IA do Código 5.1. Os planos de IA e os esquemas do plano de integração foram gerados para todos os cenários do Estudo 1 e são apresentados na Seção E.1 do Apêndice E.

O plano de IA para o cenário C2, considerando um Testador t_1, foi gerado em 0,004 segundos, consistindo em 18 ações derivadas de 31 estados. A partir desse plano de IA, o plano de integração foi estruturado em cinco *sprints*, conforme ilustrado na Figura E.2(a). O plano de integração estabelece a integração dos procedimentos $proc_1$, $proc_2$, $proc_3$ e $proc_4$ no *Sprint* #1; $proc_5$, $proc_6$ e $proc_7$ no *Sprint* #2; $proc_8$ e $proc_9$ no *Sprint* #3; $proc_{10}$ e $proc_{11}$ no *Sprint* #4; e $proc_{12}$ no *Sprint* #5. Com base nesses resultados, verifica-se que:

- no *Sprint* #1, quatro procedimentos com o mesmo tamanho ($size = 3$) foram integrados, atingindo a capacidade máxima de cada *sprint* ($TS_{max} = 12$);

- no *Sprint* #2, três procedimentos de tamanhos variados ($size = 3$ e $size = 6$) foram integrados, atingindo o $TS_{max} = 12$ do *sprint*;
- no *Sprint* #3, dois procedimentos ($size = 5$ e $size = 6$) foram integrados, mas o valor $TS_{max} = 12$ não foi alcançado. Como não havia outro procedimento com tamanho compatível para completar a capacidade de teste, a atribuição prosseguiu no próximo *sprint*;
- no *Sprint* #4, dois procedimentos com $size = 6$ atingiram a capacidade máxima $TS_{max} = 12$; e
- no *Sprint* #5, um procedimento ($size = 12$) ocupou toda a capacidade de teste do *sprint*.

Código 5.1: Cenário de teste 1 do Estudo 1: excerto do plano de IA.

```
0: INTEGRATION_AND_TEST PROC1 PR1 T1
2: INTEGRATION_AND_TEST PROC2 PR2 T1
4: INTEGRATION_AND_TEST PROC3 PR3 T1
6: INTEGRATION_AND_TEST PROC4 PR4 T1
7: NEW_SPRINT
12: INTEGRATION_AND_TEST PROC5 PR5 T1
14: INTEGRATION_AND_TEST PROC6 PR6 T1
16: INTEGRATION_AND_TEST PROC7 PR7 T1
18: INTEGRATION_AND_TEST PROC8 PR8 T1
19: NEW_SPRINT
28: INTEGRATION_AND_TEST PROC9 PR9 T1
30: INTEGRATION_AND_TEST PROC10 PR10 T1
32: INTEGRATION_AND_TEST PROC11 PR11 T1
34: INTEGRATION_AND_TEST PROC12 PR12 T1
```

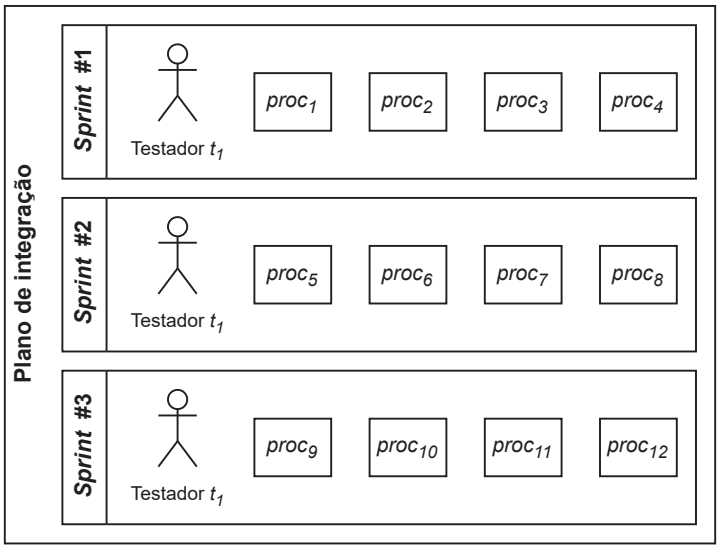


Figura 5.7: Cenário de teste 1 do Estudo 1: plano de integração.

No C2, em um ambiente com dois testadores (Testador t_1 e Testador t_2), o plano de IA foi gerado em 0,007 segundos e contém 16 ações obtidas de 30 estados. O plano de integração resultante organiza o teste dos procedimentos entre os testadores em três *sprints*, conforme a distribuição da Figura E.2(b). A distribuição ocorre alternadamente entre os dois testadores, sem atribuições para o Testador t_2 no *Sprint* #3.

Os planos de integração do cenário C3 são apresentados na Figura E.3. Para um ambiente sem balanceamento, o planejador levou 0,004s para gerar um plano de IA com 12 ações, percorrendo 13 estados. Nesse caso, todos os procedimentos ($proc_1$ a $proc_{12}$) foram atribuídos ao Testador t_1 em um único *sprint* (Figura E.3(a)). Com o balanceamento ativado, a distribuição dos procedimentos ocorre entre quatro testadores, como mostra a Figura E.3(b). Nesse caso, o plano de IA foi gerado em 0,006s, contendo 12 ações após percorrer 13 estados.

O plano de IA do cenário C4 com um testador e checagem dos procedimentos chamados pelo procedimento em teste foi gerado em 0,005 segundos. Esse plano de IA contém 15 ações encontradas entre 23 estados. Conforme indica a Figura E.4(a), os procedimentos foram atribuídos ao Testador t_1 em quatro *sprints*. No *Sprint* #1, foram atribuídos $proc_7$, $proc_{10}$ e $proc_6$. O *Sprint* #2 indica a integração de $proc_5$, $proc_{12}$ e $proc_{11}$. No *Sprint* #3 ocorreu a atribuição de $proc_8$, $proc_3$ e $proc_2$. Por fim, o $Sprint$ #4 atribui $proc_9$, $proc_4$ e $proc_1$.

Considerando C4 com dois testadores e checagem dos procedimentos chamados, o plano de IA foi gerado em 0,009 segundos, contendo 19 ações entre 37 estados. Como mostra a Figura E.4(b), os procedimentos foram atribuídos em três *sprints*. No *Sprint* #1, o Testador t_1 recebeu a integração de $proc_{12}$, $proc_{10}$ e $proc_6$, enquanto o Testador t_2 recebeu $proc_{11}$, $proc_7$ e $proc_3$. No *Sprint* #2, o Testador t_1 ficou responsável por $proc_8$ e $proc_5$ e o Testador $t2$ por $proc_9$ e $proc_2$. No *Sprint* #3, o Testador t_1 ficou responsável por $proc_4$ e o Testador t_2 por $proc_1$.

Considerando o C4 com um testador e checagem dos procedimentos que chamam o procedimento em teste, o plano de IA foi gerado em 0,005 segundos, sendo composto por ações 15 a partir de 24 estados. Como ilustra a Figura E.4(c), o planejador distribuiu os procedimentos para o Testador t_1 em quatro *sprints*. Os procedimentos $proc_1$, $proc_2$ e $proc_3$ foram atribuídos no *Sprint* #1, seguidos dos procedimentos $proc_6$, $proc_{10}$ e $proc_4$ no *Sprint* #2, $proc_9$, $proc_8$ e $proc_{12}$ no *Sprint* #3 e $proc_{11}$, $proc_7$ e $proc_5$ no *Sprint* #4.

Para o caso de C4 com checagem dos procedimentos que chamam e dois testadores, o plano de IA foi gerado em 0,008 segundos, com 18 ações em 42 estados. A Figura E.4(d) apresenta o plano de integração para esse caso. No *Sprint* #1, o Testador t_1 ficou responsável por $proc_1$, $proc_2$ e $proc_3$, sem atribuições para o Testador t_2. No *Sprint* #2, $proc_6$, $proc_{10}$ e $proc_7$ foram atribuídos ao Testador #1, enquanto $proc_4$, $proc_8$ e $proc_{12}$ foram atribuídos ao Testador #2. No *Sprint* #3, o Testador t_1 integrou $proc_{11}$ e o Testador t_2 integrou $proc_9$.

5.2.3 Discussão dos resultados do Estudo 1

- **Atividade iv) Análise dos planos de integração**: após a catalogação dos resultados do estudo, foram realizadas análises dos principais pontos observados. Como mostra a coluna 'Plano de IA" da Tabela E.1, foram gerados planos de IA factíveis que definem sequências adequadas para a integração de cada cenário. Esses planos seguem as prioridades de integração (cenário 1), as capacidades de teste dos testadores e dos *sprints* (cenário 2), o balanceamento da atribuição de teste (cenário 3) e as chamadas de operações entre procedimentos (cenário 4).

A coluna "Tempo" da Tabela E.1 mostra que os tempos de execução do planejador variaram entre 0,004s e 0,009s. Os maiores tempos ocorreram em cenários com múltiplos testadores, como os cenários 2 e 4. A coluna "Estados" destaca que os planos de IA foram encontrados entre 13 e 50 estados. Nesse sentido, cenários com mais testadores envolvem espaços de estados maiores, como no cenário 4. Já a coluna "Ações" indica que os planos de IA contêm entre 12 e 35 ações, geralmente proporcionais ao número de estados percorridos, como no cenário 1. Esses resultados sugerem indícios sobre a escalabilidade do problema.

A representação avaliada neste estudo parte da premissa de que todos os procedimentos estão disponíveis para integração. Isso implica que a modelagem não incorpora o uso de *stubs* e *drivers*. No entanto, na prática, algumas unidades (procedimentos, métodos ou classes) podem

estar indisponíveis durante a integração, exigindo o uso desses elementos. Nesses casos, a troca de unidades por *stubs* ou *drivers* pode seguir a ordem de atribuição indicada nos planos de IA.

Neste estudo, foram atribuídos valores contantes e variáveis para os elementos da representação PDDL, como os tamanhos $size_i$ e as prioridades pr_i. A instanciação desses elementos pode depender de mecanismos auxiliares, como ferramentas e algoritmos adicionais. Dessa forma, fatores externos, como a corretude e a eficácia desses mecanismos, podem influenciar os resultados usando a representação PDDL com elementos instanciados.

Os planos de integração gerados no cenário 4 e ilustrados na Figura E.4 respeitaram a verificação dos procedimentos chamados pelo procedimento em teste. Esses planos indicam que um procedimento $proc_i$ foi atribuído a um testador somente após a atribuição de todos os procedimentos $proc_i$ que ele chama operações. Essa verificação é um pré-requisito para a execução da estratégia de integração *bottom-up*. Dessa forma, o estudo indicou que a representação PDDL é viável para essa estratégia.

Este estudo desconsiderou informações dos procedimentos, como parâmetros e tipos de retorno. Os resultados sugerem que os planos de integração sem essas informações são mais eficazes para o gerenciamento dos testes. Para melhorar a operacionalização e facilitar a execução, essas informações foram incluídas na representação PDDL. A avaliação da representação com essas informações é detalhada na Seção 5.3.

5.3 ESTUDO 2: INSTANCIAÇÃO DAS MODELAGENS EM PROJETOS DE DESENVOLVIMENTO

O Estudo 2 busca avaliar a viabilidade da instanciação das representações PDDL descritas nas Seções 4.3 e 4.4 usando informações de projetos de um sistema procedimental e de um sistema OO, respectivamente. A Tabela 5.5 descreve o objetivo desse estudo com o GQM. As etapas e as atividades do Estudo 2 são apresentadas nas próximas subseções seguindo as definições da Figura 5.1.

Tabela 5.5: Objetivo do Estudo 2 estruturado com o GQM.

Analisar	planos de integração gerados a partir das representações PDDL apresentadas nas Seções 4.3 e 4.4
Com o propósito de	avaliar
Em relação a	viabilidade de instanciação das representações em projetos de: sistema procedimental; e sistema OO
Do ponto de vista de	gerente de teste
No contexto de	planejamento do teste de integração

5.3.1 Preparação e execução do Estudo 2

• **Atividade i) Definição dos cenários de teste**: foram definidos quatro cenários para avaliar separadamente os planos de integração gerados com a representação PDDL do teste procedimental descrita na Seção 4.3 e a representação PDDL do teste OO apresentada na Seção 4.4. Para isso, essas representações foram instanciadas com informações de projetos de um sistema procedimental e de um sistema OO.

• **Atividade i) Definição dos cenários de teste 1 e 2**: os cenários de teste 1 e 2 correspondem, respectivamente, às estratégias de integração *top-down* e *bottom-up*. Esses cenários consideram todos os elementos de um sistema procedimental $SUT_p = (Proc, Pa, Pt, Rt, Co, L)$ e

de um ambiente $Am_p = (T, Pr, TS_{max})$ descritos na Seção 4.3. O SUT_p utilizado nesses cenários é uma versão do jogo *River Raid* com interface gráfica. A Figura 5.8 ilustra os procedimentos ($proc_i$), os parâmetros (pa_i) e os tipos de retorno (rt_i) desse sistema.

River Raid é um jogo de tiro no qual o jogador controla uma nave, enfrenta inimigos, coleta combustível e tenta manter suas vidas ao longo da partida. O SUT_p *River Raid* foi desenvolvido em linguagem C durante a disciplina Oficina de Computação do curso de Bacharelado em Ciência da Computação (BCC) da UFPR. A Tabela 5.6 apresenta as instâncias consideradas nos cenários 1 e 2, abrangendo os 23 procedimentos $proc_i$ do SUT, uma descrição das funcionalidades desses procedimentos, seus tamanhos $size_i$ e suas respectivas prioridades pr_i.

Tabela 5.6: Cenários de teste 1 e 2 do Estudo 2: instanciação do SUT *River Raid*.

$proc_i$	**Descrição**	$size_i$	pr_i
$proc_1$: main	Procedimento principal que inicializa o jogo e controla o *loop* da partida	$size_1 = 6$	pr_1
$proc_2$: inicializa_nc	Inicializa a biblioteca para a interface gráfica	$size_2 = 3$	pr_4
$Proc_3$: inicializa_partida	Cria uma nova partida	$size_3 = 6$	pr_2
$proc_4$: gerencia_partida	Gerencia o estado da partida	$size_4 = 8$	pr_1
$proc_5$: desenha_graficos	Desenha os elementos gráficos na tela	$size_5 = 7$	pr_3
$proc_6$: infos_nave	Mostra informações da nave, como vidas, combustível e pontuação	$size_6 = 2$	pr_4
$proc_7$: info_controles	Exibe os controles do jogo na tela	$size_7 = 2$	pr_4
$proc_8$: mostra_score	Exibe a pontuação e a data no final do jogo	$size_8 = 4$	pr_4
$proc_9$: desaloca_jogo	Libera a memória alocada para os elementos do jogo	$size_9 = 4$	pr_4
$proc_{10}$: inicializa_nave	Inicializa as coordenadas da nave, o número de vidas, o combustível e a pontuação inicial	$size_{10} = 5$	pr_2
$proc_{11}$: inicializa_projetil	Define as coordenadas do projétil	$size_{11} = 4$	pr_2
$proc_{12}$: inicializa_inimigo	Define as coordenadas do inimigo	$size_{12} = 4$	pr_2
$proc_{13}$: inicializa_combustivel	Define as coordenadas do combustível	$size_{13} = 4$	pr_2
$proc_{14}$: gerencia_nave	Captura entrada do teclado para atualizas a posição da nave ou disparar um projétil	$size_{14} = 6$	pr_1
$proc_{15}$: gerencia_inimigo	Atualiza a posição vertical dos inimigos movendo-os para baixo	$size_{15} = 2$	pr_1
$proc_{16}$: gerencia_combustivel	Gerencia o movimento e a quantidade de combustível	$size_{16} = 2$	pr_1
$proc_{17}$: gerencia_projetil	Atualiza a posição vertical dos projéteis movendo-os para cima	$size_{17} = 2$	pr_1
$proc_{18}$: colidiu_nave_inimigo	Verifica se houve uma colisão entre a nave do jogador e um inimigo	$size_{18} = 3$	pr_1
$proc_{19}$: colidiu_tiro_inimigo	Verifica se um projetil colidiu com um inimigo	$size_{19} = 3$	pr_1
$proc_{20}$: colidiu_nave_combustivel	Verifica se a nave colidiu com um objeto de combustível	$size_{20} = 3$	pr_1
$proc_{21}$: colidiu_tiro_combustivel	Verifica se um projetil colidiu com um objeto de combustível	$size_{21} = 3$	pr_1
$proc_{22}$ desenha_cenario	Desenha o cenário do jogo na tela	$size_{22} = 4$	pr_3
$proc_{23}$: define_cores	Inicializa as cores utilizadas no jogo	$size_{23} = 3$	pr_3

Os tamanhos indicados na coluna "$size_i$" da Tabela 5.6 foram definidos com base no número de linhas de código de cada $proc_i$. Procedimentos com mais linhas foram classificados como de maior tamanho: aqueles com até 5 linhas receberam $size_i = 2$; de 6 a 10 linhas, $size_i = 3$; de 11 a 15 linhas, $size_i = 4$; de 16 a 20 linhas; $size_i = 5$; de 21 a 25 linhas, $size_i = 6$, de 26 a 30 linhas, $size_i = 7$; e procedimentos com mais de 30 linhas, $size_i = 8$.

As prioridades de integração da coluna "pr_i" da Tabela 5.6 foram definidas com base na hierarquia dos procedimentos. No nível l_2, a prioridade foi determinada pelo número de chamadas realizadas, atribuindo maior prioridade aos procedimentos que invocam mais operações. No nível l_3, a prioridade foi definida considerando a relevância dos procedimentos chamados, de forma que aqueles invocados por procedimentos de maior prioridade também receberam prioridade elevada. Os cenários 1 e 2 envolvem dois testadores, t_1 e t_2, ambos com capacidade de teste $cap_i = 8$. Cada *sprint* possui uma capacidade máxima de teste $TS_{max} = 8$.

- **Atividade i) Definição dos cenários de teste 3 e 4**: os cenários de teste 3 e 4 abordam, respectivamente, o teste intermétodos e o teste interclasses. Esses cenários consideram todos os elementos de um sistema OO $SUT_{oo} = (C, Atr, Ta, M, Arg, Targ, Rt_m, Me)$ e de um ambiente $Am_{oo} = (T, Pr_m, Pr_c, TS_{max})$ descritos na Seção 4.4.

O SUT_{oo} utilizado nos cenários 3 e 4 representa uma rede social simplificada, permitindo que usuários interajam, formem amizades, participem de grupos, publiquem postagens e realizem

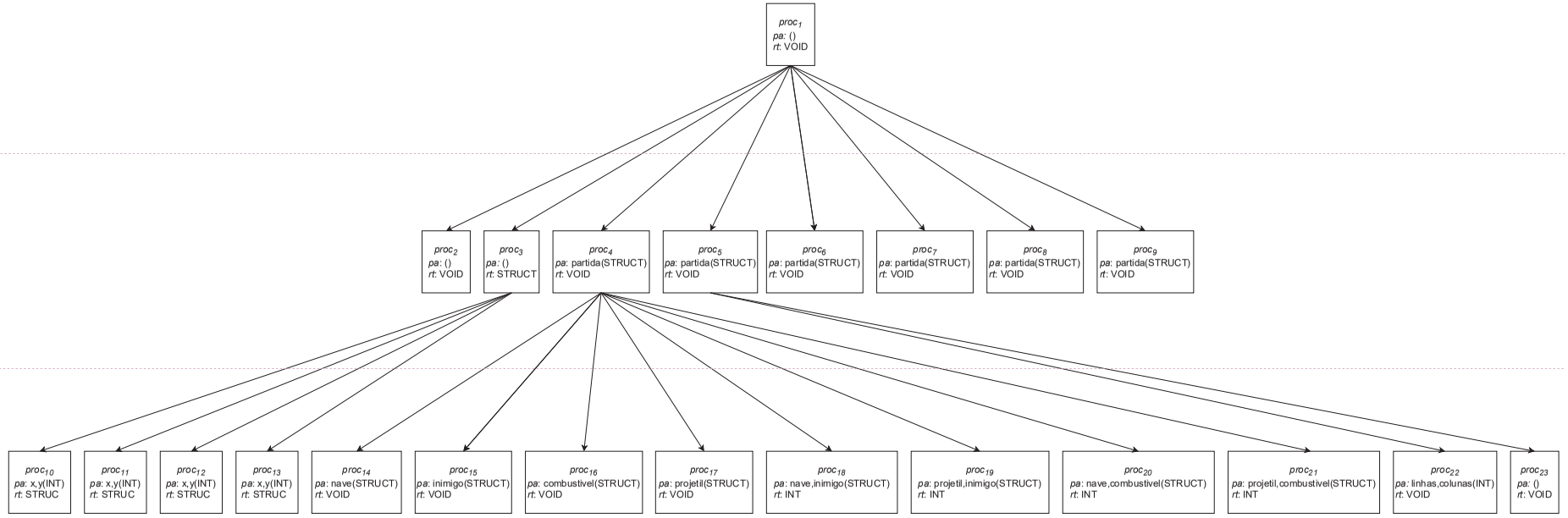


Figura 5.8: Cenários de teste 1 e 2 do Estudo 2: características do SUT *River Raid*.

ações como comentar ou curtir publicações. Esse SUT_{oo} foi desenvolvido em Java para a disciplina Técnicas Alternativas de Programação do curso de BCC da UFPR. A Figura 5.9 ilustra a estrutura do SUT_{oo} Rede Social em um diagrama de classes simplificado.

A Tabela 5.7 apresenta a instanciação considerada nos cenário 3 e 4, incluindo as classes c_i e os métodos m_i do SUT_{oo} Rede Social, seus respectivos tamanhos ($size_i$) e suas prioridades pr_{c_i} e pr_{m_i}. Classes com um maior número de atributos e métodos receberam tamanhos e prioridades pr_{c_i} mais elevadas. Os métodos de cada classe foram distribuídos alternadamente entre três níveis de prioridade (pr_{m_1}, pr_{m_2} e pr_{m_3}). Além disso, todos os métodos foram atribuídos com tamanho fixo de $size_i = 1$.

Tabela 5.7: Cenários de teste 3 e 4 do Estudo 2: instanciação do SUT Rede Social.

Classe c_i / Método m_i	**Descrição**	$size_i$	pr_{c_i} / pr_{m_i}
c_1: Programa	Classe principal que executa o sistema	$size_1 = 1$	pr_{c_3}
c_2: Group	Representa um grupo na rede social	$size_2 = 3$	pr_{c_1}
c_3: Members	Gerencia os membros de um grupo	$size_3 = 2$	pr_{c_2}
c_4: Post	Representa uma postagem na rede social	$size_4 = 3$	pr_{c_1}
c_5: Person	Representa uma pessoa na rede social	$size_5 = 2$	pr_{c_2}
c_6: Friendship	Gerencia as amizades entre pessoas	$size_6 = 2$	pr_{c_2}
c_7: Observer	Interface para classes que precisam ser notificadas sobre atualizações	$size_7 = 1$	pr_{c_3}
c_8: Subject	Interface para classes que gerenciam observadores	$size_8 = 1$	pr_{c_3}
m_1: main	Método principal que inicializa a aplicação, cria usuários, grupos e posts	$size_1 = 1$	pr_{m_1}
m_2: getGroupName	Retorna o nome do grupo	$size_2 = 1$	pr_{m_1}
m_3: setGroupName	Define o nome do grupo	$size_3 = 1$	pr_{m_2}
m_4: getGroupLogin	Retorna o *login* do grupo	$size_4 = 1$	pr_{m_3}
m_5: setGroupLogin	Define o *login* do grupo	$size_5 = 1$	pr_{m_1}
m_6: postGroup	Publica uma postagem no grupo	$size_6 = 1$	pr_{m_2}
m_7: notifyObservers	Notifica os membros do grupo sobre uma nova postagem	$size_7 = 1$	pr_{m_3}
m_8: setLike	Registra uma reação de um membro ao post do grupo	$size_8 = 1$	pr_{m_1}
m_9: update	Atualiza o grupo com um novo post	$size_9 = 1$	pr_{m_2}
m_{10}: update	Atualiza o grupo com uma nova postagem e referência ao autor	$size_{10} = 1$	pr_{m_3}
m_{11}: setMembers	Adiciona um membro ao grupo	$size_{11} = 1$	pr_{m_1}
m_{12}: removeMembers	Remove um membro do grupo	$size_{12} = 1$	pr_{m_2}
m_{13}: printMember	Exibe os membros do grupo	$size_{13} = 1$	pr_{m_3}
m_{14}: update	Atualiza um membro com uma nova postagem e referência ao autor	$size_{14} = 1$	pr_{m_1}
m_{15}: update	Atualiza um membro com uma nova postagem	$size_{15} = 1$	pr_{m_2}
m_{16}: setNewComment	Adiciona um comentário à postagem	$size_{16} = 1$	pr_{m_1}
m_{17}: setLike	Registra uma reação na postagem	$size_{17} = 1$	pr_{m_2}
m_{18}: Posts	Cria uma nova postagem	$size_{18} = 1$	pr_{m_3}
m_{19}: notifyObservers	Notifica os observadores sobre a nova postagem	$size_{19} = 1$	pr_{m_1}
m_{20}: registerObserver	Registra um observador na postagem	$size_{20} = 1$	pr_{m_2}
m_{21}: removeObserver	Remove um observador da postagem	$size_{21} = 1$	pr_{m_3}
m_{22}: getPersonLogin	Retorna o *login*	$size_{22} = 1$	pr_{m_1}
m_{23}: setPersonLogin	Define o *login*	$size_{23} = 1$	pr_{m_2}
m_{24}: getPersonName	Retorna o nome	$size_{24} = 1$	pr_{m_3}
m_{25}: setPersonName	Define o nome	$size_{25} = 1$	pr_{m_1}
m_{26}: update	Atualiza a pessoa com uma novo postagem e referência ao autor	$size_{26} = 1$	pr_{m_2}
m_{27}: update	Atualiza a pessoa com uma nova postagem	$size_{27} = 1$	pr_{m_3}
m_{28}: setMembers	Adiciona um membro ao grupo	$size_{28} = 1$	pr_{m_1}
m_{29}: removeMembers	Remove um membro do grupo	$size_{29} = 1$	pr_{m_2}
m_{30}: printMember	Exibe os membros do grupo	$size_{30} = 1$	pr_{m_3}
m_{31}: update	Atualiza um membro com uma nova postagem e referência ao autor	$size_{31} = 1$	pr_{m_1}
m_{32}: update	Atualiza um membro com uma nova postagem	$size_{32} = 1$	pr_{m_2}
m_{33}: update	Atualiza o observador com uma nova postagem e referência ao autor	$size_{33} = 1$	pr_{m_1}
m_{34}: update	Atualiza o observador apenas com uma nova postagem	$size_{34} = 1$	pr_{m_2}
m_{35}: registerObserver	Registra um novo observador	$size_{35} = 1$	pr_{m_1}
m_{36}: removeObserver	Remove um observador	$size_{36} = 1$	pr_{m_2}
m_{37}: notifyObservers	Notifica os observadores sobre uma nova postagem	$size_{37} = 1$	pr_{m_3}

A Figura 5.9 ilustra os atributos atr_i, os tipos de atributos at_i, os argumentos arg_i, os tipos de argumentos $argt_i$, os tipos de retorno rt_{m_i} do SUT_{oo} Rede Social. O ambiente Am_{oo} dos cenários 3 e 4 inclui dois testadores, t_1 e t_2, cada um com capacidade de teste $cap_i = 6$. Além disso, a capacidade máxima de teste por *sprint* foi definida como $TS_{max} = 6$.

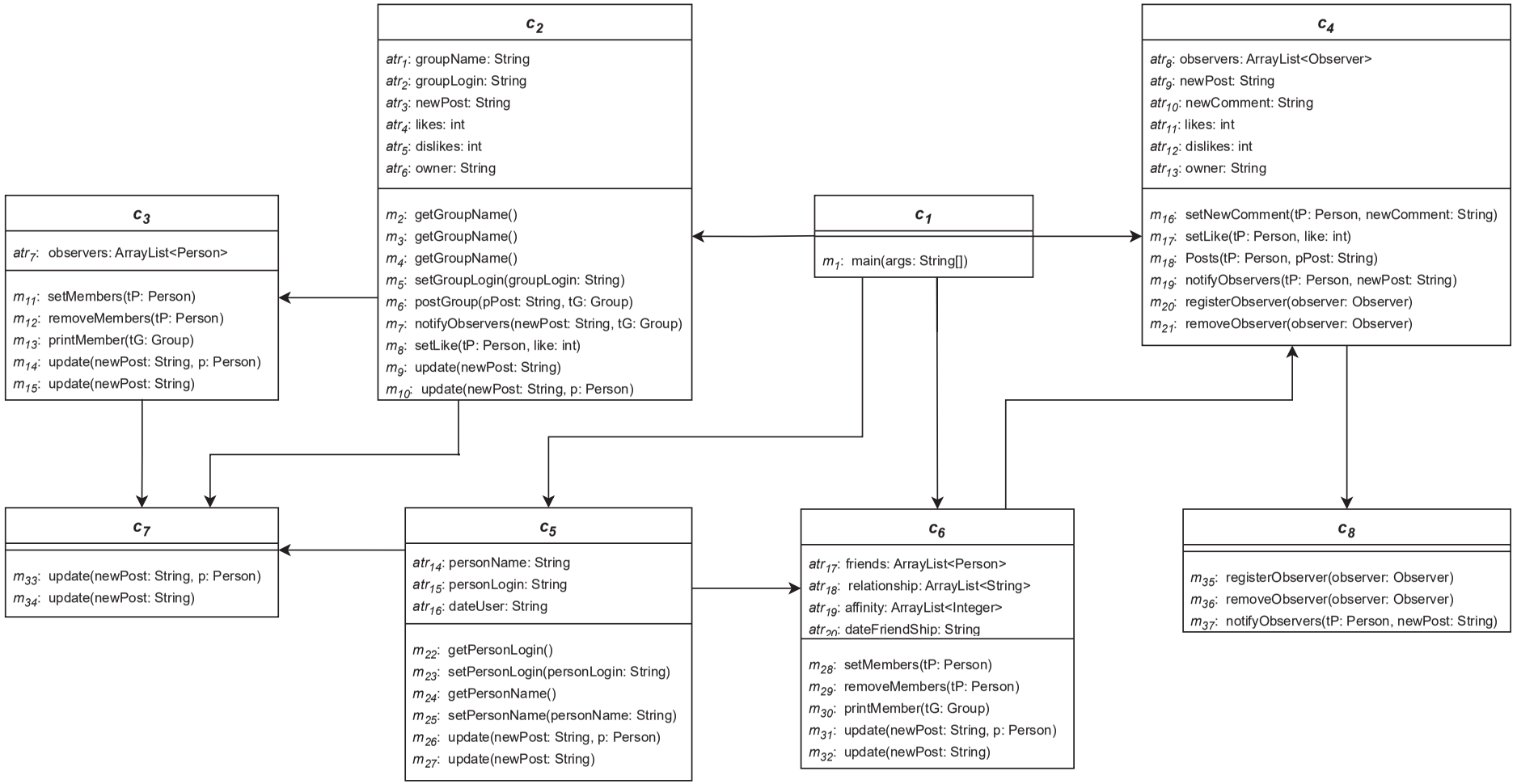


Figura 5.9: Cenários de teste 3 e 4 do Estudo 2: características do SUT Rede Social em um diagrama de classes simplificado.

• **Atividade ii) Instanciação da representação PDDL**: após a definição dos cenários, foi realizada a instanciação das representações PDDL. As informações indicadas na Tabela 5.6 e na Figura 5.8 foram usadas na instanciação da representação PDDL do teste procedimental. As informações apresentadas na Tabela 5.7 e as características descritas na Figura 5.9 instanciaram a representação PDDL do teste OO.

• **Atividade iii) Geração dos planos de IA**: após a instanciação das representações PDDL para cada cenário, foi realizada a geração dos planos com o planejador Metric-FF.

5.3.2 Resultados do Estudo 2

Os resultados encontrados durante a geração dos planos de IA em cada cenário foram tabulados e são apresentados na Tabela 5.8. Essa tabela apresenta as características dos cenários (coluna "Características"), o tempo necessário para a geração do plano de IA (coluna "Tempo"), o número de estados percorridos (coluna "Estados") e a quantidade de ações contidas no plano de IA (coluna "Ações"). O tempo é indicado em segundos (s).

Cenário	Características	Tempo	Estados	Ações
C1	Procedimental, *top-down*	0m40,432s	382645	126
C2	Procedimental, *bottom-up*	0m56,229s	531311	137
C3	OO, intermétodos	1m56,945s	393592	167
C4	OO, interclasses	0m11,791s	197634	55

Tabela 5.8: Síntese dos resultados do Estudo 2.

A Tabela E.2, localizada no Apêndice E, amplia a apresentação dos resultados com excertos dos planos de IA (coluna "Plano de IA"). Essa tabela não apresenta ações que indicam a checagem das informações, o incremento das prioridades, a atualização das capacidades de teste, a troca de testador e a troca de classes. Os excertos exibidos são adaptações das saídas geradas pelo planejador. Essas saídas indicam as informações de cada método ou classe em uma única ação. Essas ações foram posteriormente replicadas e instanciadas com os dados de cada SUT.

Os planos de IA e os esquemas gráficos que ilustram o plano de integração correspondentes foram gerados para todos os cenários do Estudo 2. Devido ao tamanho, os planos e os esquemas são apresentados apenas na Seção E.2 do Apêndice E.

O plano de IA para o cenário 1 foi gerado em 0m40,432s. Esse plano contém 126 ações, encontradas a partir de 382.645 estados. Os procedimentos foram distribuídos aos testadores em seis *sprints*, conforme ilustrado na Figura E.5(a). No *Sprint* #1, os procedimentos $proc_6$ e $proc_3$ foram atribuídos ao Testador t_1, enquanto $proc_1$ e $proc_7$ foram atribuídos ao Testador t_2. No *Sprint* #2, t_1 ficou responsável por $proc_5$ e t_2 por $proc_4$. No *Sprint* #3, $proc_2$ e $proc_9$ foram atribuídos a t_1, enquanto $proc_8$, $proc_{15}$ e $proc_6$ foram designados a t_2.

Ainda em relação ao plano de integração do cenário 1 (Figura E.5(a)), no *Sprint* #4, os procedimentos $proc_{14}$ e $proc_{17}$ foram atribuídos a t_1, enquanto $proc_{18}$ e $proc_{19}$ foram designados a t_2. No *Sprint* #5, t_1 ficou responsável por $proc_{20}$ e $proc_{22}$, enquanto t_2 assumiu $proc_{10}$ e $proc_{23}$. Por fim, no *Sprint* #6, os procedimentos $proc_{11}$ e $proc_{12}$ foram atribuídos ao Testador t_1, enquanto $proc_{13}$ e $proc_{22}$ foram designados ao Testador t_2.

No cenário 2, o plano de IA foi gerado pelo planejador em 0m56,229s. Esse plano de IA consiste em 137 ações, encontradas a partir de um total de 531311 estados. A Figura E.5(b) ilustra o plano de integração decorrente desse plano de IA. Nesse cenário, a integração foi

distribuída em seis *sprints*. No *Sprint* #1, os procedimentos $proc_{10}$ e $proc_{16}$ foram atribuídos ao Testador t_1, enquanto $proc_{15}$, $proc_{18}$ e $proc_{19}$ foram atribuídos ao Testador t_2. No *Sprint* #2, $proc_{14}$ foi atribuído a t_1, e $proc_{17}$, $proc_{23}$ e $proc_{20}$ foram designados a t_2.

Dando continuidade ao plano de integração do cenário 2 (Figura E.5(b)), no *Sprint* #3, o Testador t_1 ficou responsável por $proc_{13}$ e $proc_{14}$, enquanto o Testador t_2 assumiu $proc_{11}$ e $proc_{22}$. No *Sprint* #4, os procedimentos $proc_{21}$, $proc_2$ e $proc_6$ foram atribuídos a t_1, enquanto $proc_4$ foi designado a t_2. Durante o *Sprint* #5, $proc_3$ e $proc_7$ foram indicados para t_1, e $proc_8$ e $proc_9$ para t_2. Por fim, no *Sprint* #6, $proc_5$ e $proc_1$ foram atribuídos a t_1 e t_2, respectivamente.

O plano de IA para o cenário 3 foi gerado em 1m56,945s. Esse plano de IA contém 167 ações identificadas ao longo de 393.592 estados percorridos. A Figura E.6(a) representa graficamente o plano de integração desse cenário. Os métodos foram distribuídos entre os testadores t_1 e t_2 ao longo de dez *sprints*. Nos primeiros nove *sprints*, cada testador recebeu a atribuição de dois métodos. No *Sprint* #10, o método m_{36} foi designado ao Testador t_1, enquanto t_2 não recebeu nenhuma atribuição.

No cenário 4, o plano de IA foi gerado pelo planejador em 0m11,791s. Esse plano de IA contém 55 ações identificadas ao longo de 197.634 estados percorridos. A Figura E.6(b) ilustra o plano de integração decorrente desse plano de IA. As classes foram distribuídas entre os testadores em dois *sprints*. No *Sprint* #1, as classes c_3 e c_4 foram atribuídas ao Testador t_1, enquanto as classes c_2, c_7 e c_5 foram designadas ao Testador t_2. No *Sprint* #2, o Testador t_1 ficou responsável pelas classes c_6 e c_8, enquanto o Testador t_2 recebeu a classe c_1.

5.3.3 Discussão dos resultados do Estudo 2

• **Atividade iv) Análise dos planos de integração**: após a catalogação dos dados, foi realizada uma análise dos resultados. A coluna "Plano de IA" da Tabela E.2 revela que foram gerados planos de IA factíveis para os cenários de integração *top-down* (cenário 1), *bottom-up* (cenário 2), intermétodos (cenário 3) e interclasses (cenário 4). Em todos os casos, os planos de integração resultantes definem sequências adequadas para a integração, alinhadas às características de cada cenário. Esses resultados sugerem que a aplicação da abordagem TI-PIA é viável em contextos que envolvem SUTs desenvolvidos segundo diferentes paradigmas.

A coluna "Tempo" da Tabela E.2 mostra que os tempos de geração dos planos de IA variaram de 0m11,791s a 1m56,945s. O maior tempo foi registrado no cenário C4, que apresenta a maior quantidade de objetos representados. O número de estados variou de 197.634 a 531.311, conforme evidenciado na coluna "Estados" da Tabela E.2. Ainda, conforme a coluna "Ações", a quantidade de ações nos planos de IA variou entre 55 e 167. As maiores quantidades de estados e ações também estão associadas a cenários com um grande número de objetos.

Em todos os cenários do Estudo 2, foram realizadas tentativas para a geração dos planos com as representações PDDL completamente instanciadas com as informações dos SUTs das Figuras 5.8 e 5.9. Esse processo indicou que a escalabilidade do problema está diretamente relacionada à quantidade de objetos instanciados. Isso sugere que a abordagem TI-PIA pode ser mais eficiente em projetos de pequeno e médio porte ou em projetos com um menor número de variáveis. Como alternativa para projetos que envolvem muitos objetos, propõe-se a geração de planos de integração individuais para subsistemas.

No contexto do Estudo 2, a inteligibilidade refere-se à facilidade com que um plano de IA pode ser compreendido pelos interessados. A formatação das ações e a sequência de apresentação visam facilitar a leitura e a interpretação dos planos de IA por gerentes de teste e testadores. Dessa forma, entende-se que os planos de IA e os planos de integração são artefatos que podem auxiliar nas atividades de planejamento e execução do teste de integração. A clareza na apresentação dessas informações é importante para o desempenho das equipes de teste.

A parametrização delimitada para as ações `PROCEDURE_PARAMETER`, `CLASS_METHOD`, `CLASS_ATTRIBUTE` e `METHOD_ARGUMENT` busca indicar de forma compreensível as informações das unidades. Durante a execução do teste, essas informações podem auxiliar o testador a escolher os casos de teste adequados para cada unidade e a reconhecer rapidamente a configuração das unidades simuladas. No planejamento do teste, essas informações contribuem para que o gerente de teste direcione os recursos de forma mais eficaz.

5.4 ESTUDO 3: INVESTIGAÇÃO DAS MODELAGENS NA MANUTENÇÃO DE SOFTWARE

O Estudo 3 busca investigar a viabilidade de aplicação da representação PDDL e a geração dos planos de integração em contextos variados de manutenção de software. O objetivo desse estudo é apresentado na Tabela 5.9 com o GQM. As próximas subseções apresentam as etapas e as atividades do Estudo 3 indicadas na Figura 5.1.

Tabela 5.9: Objetivo do Estudo 3 estruturado com o GQM.

Analisar	planos de integração gerados a partir da representação PDDL apresentada na Seção 4.3
Com o propósito de	avaliar
Em relação a	viabilidade de aplicação da representação na manutenção de software envolvendo alteração, inclusão e exclusão de procedimentos
Do ponto de vista de	gerente de teste
No contexto de	planejamento do teste de integração

5.4.1 Preparação e execução do Estudo 3

• **Atividade i) Definição dos cenários de teste**: o Estudo 3 considera o SUT_p ilustrado na Figura 5.10. Esse sistema é composto por 12 procedimentos ($proc_1$ a $proc_{12}$) e um conjunto Co contendo 12 chamadas de operações. Os procedimentos estão organizados em quatro níveis hierárquicos (l_1, l_2, l_3 e l_4). As prioridades pr_i foram definidas com base no número de procedimentos subordinados a cada $proc_i$. Assim, foram atribuídas maiores prioridades aos procedimentos que possuem mais procedimentos subordinados.

As prioridades mais baixas possuem um peso $w = 1$. À medida que a prioridade aumenta, os pesos são incrementados em 1. Como indica a Figura 5.10, todos os procedimentos $proc_i$ possuem tamanho constante ($size_i = 3$). O ambiente Am_p desse estudo inclui um testador t_1 com capacidade de teste $cap_1 = 6$ e a capacidade máxima por *sprint* é $TS_{max} = 6$. Para cada procedimento $proc_i$, foram definidos parâmetros pa_i, tipos de parâmetros pt_i e tipos de retorno rt_i. Essas configurações do SUT e do ambiente resultam na versão 1 (v_1) do plano de integração.

Durante a atividade (i), foram definidos três cenários de teste baseados na estratégia de integração *top-down*. Esses cenários consideram processos de manutenção do SUT_p após sua implantação. Cada cenário resulta em uma versão 2 (v_2) do plano de integração.

• **Atividade i) Definição do cenário de teste 1**: esse cenário corresponde a um processo de manutenção do SUT_p após a modificação de determinados procedimentos. A Figura 5.11 ilustra a configuração do SUT_p para esse cenário. Nele, os procedimentos $proc_3$, $proc_5$ e $proc_{11}$ foram alterados e, por isso, receberam a prioridade de integração mais alta (pr_1). Os demais procedimentos nos mesmos níveis dos procedimentos alterados tiveram suas prioridades ajustadas de forma proporcional, conforme indicado na Figura 5.10.

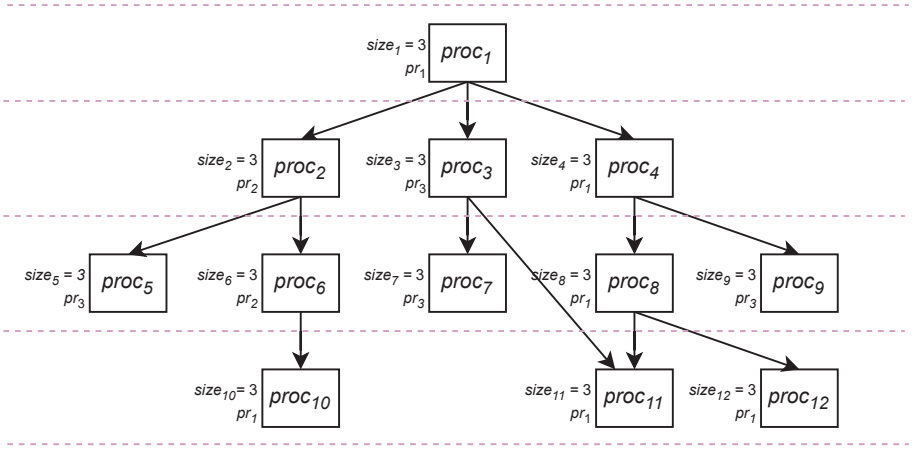


Figura 5.10: Características do SUT utilizado no Estudo 3. Fonte: Adaptado de Myers et al. (2011).

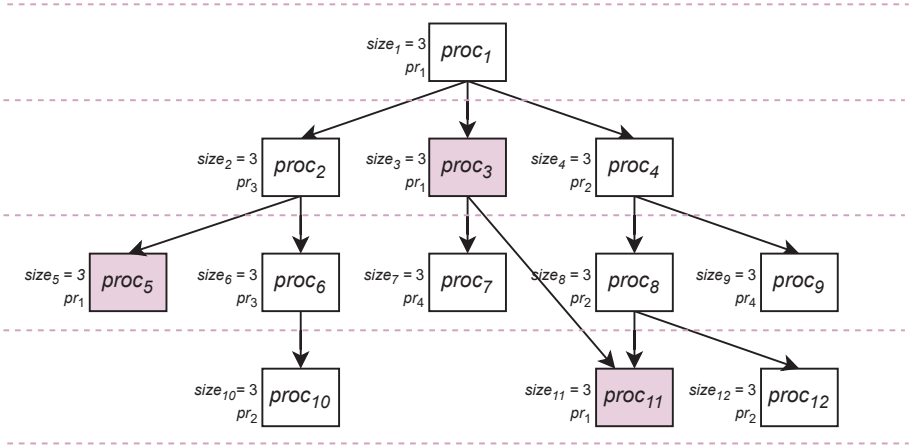


Figura 5.11: Cenário de teste 1 do Estudo 3: características do SUT.

• **Atividade i) Definição do cenário de teste 2**: esse cenário representa a manutenção do SUT_p após a inclusão de um novo procedimento. A Figura 5.12 ilustra a configuração do SUT_p para esse cenário. Nele, o procedimento $proc_{13}$ foi adicionado como subordinado ao procedimento $proc_9$ e recebeu a prioridade de integração mais alta (pr_1). Com essa inclusão, as prioridades dos procedimentos no mesmo nível de $proc_{13}$, assim como a de $proc_9$, foram ajustadas proporcionalmente.

• **Atividade i) Definição do cenário de teste 3**: esse cenário corresponde a um processo de manutenção após a exclusão de um procedimento. Nesse cenário, os procedimentos $proc_{11}$ e $proc_{12}$ foram removidos do SUT_p, o que resultou na eliminação das chamadas de operação $(proc_3, proc_{11})$, $(proc_8, proc_{11})$ e $(proc_8, proc_{12})$. Como consequência, os procedimentos $proc_3$ e $proc_8$ precisam ser revisados e recebem a prioridade mais alta. Além disso, as prioridades dos demais procedimentos nos níveis l_2 e l_3 foram ajustadas. A Figura 5.13 ilustra a configuração do SUT_p nesse cenário.

• **Atividade ii) Instanciação da representação PDDL**: após as definições de cada cenário, as informações das Figuras 5.10, 5.11, 5.12 e 5.13 foram utilizadas para instanciar a representação PDDL do teste procedimental apresentada na Seção 4.3.

• **Atividade iii) Geração dos planos de IA**: a partir da representação PDDL instanciada com as características de cada cenário, os planos de IA foram gerados com o planejador Metric-FF.

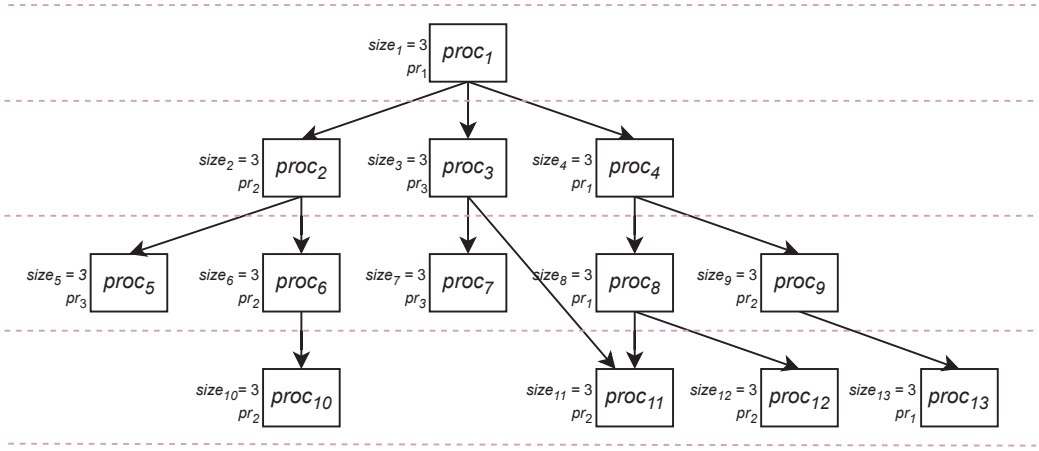


Figura 5.12: Cenário de teste 2 do Estudo 3: características do SUT.

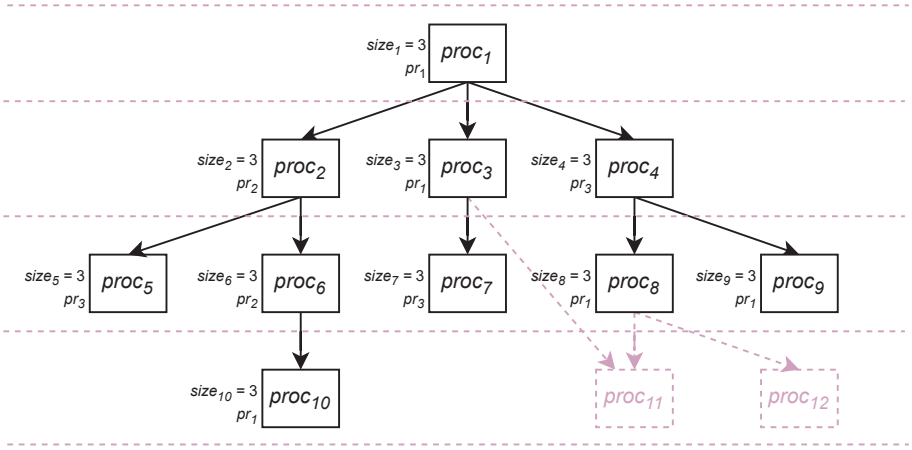


Figura 5.13: Cenário de teste 3 do Estudo 3: características do SUT.

5.4.2 Resultados do Estudo 3

Os resultados encontrados durante a geração dos planos foram organizados e são apresentados na Tabela 5.10. Essa tabela apresenta as características de cada cenário (coluna "Características"), o tempo de geração dos planos de IA (coluna "Tempo"), o número de estados percorridos (coluna "Estados"), o número de ações dos planos de IA (coluna "Ações"). O tempo é indicado em segundos (s).

Cenário	Características	Tempo	Estados	Ações
Cenário geral	Desenvolvimento	0,033s	859	48
C1	Alteração de procedimento	0,132s	5344	52
C2	Inclusão de procedimento	0,047s	1377	53
C3	Remoção de procedimento	0,018s	591	43

Tabela 5.10: Síntese dos resultados do Estudo 3.

A Tabela E.3 do Apêndice E apresenta excertos dos planos de IA de cada cenário (coluna "Plano de IA"). Para destacar as atribuições de procedimentos realizadas a partir das

características de cada cenário, a coluna "Plano de IA" da Tabela E.2 exibe apenas as ações que indicam a integração dos procedimentos e o início de novos *sprints*.

O plano de IA para o cenário 1 foi gerado em 0,132s, contendo 52 ações identificadas em um total de 5.344 estados. A Figura E.7(b) ilustra o plano de integração para esse cenário. Esse plano prevê a realização da integração dos procedimentos pelo Testador t_1 em três *sprints*. No *Sprint* #1, foram atribuídos os procedimentos $proc_1$, $proc_3$, $proc_4$ e $proc_2$; no *Sprint* #2, $proc_5$, $proc_8$, $proc_6$ e $proc_7$; e, por fim, no *Sprint* #3, $proc_9$, $proc_{11}$, $proc_{12}$ e $proc_{10}$.

A Figura 5.14 compara a primeira versão (v_1) do plano de integração gerado durante o desenvolvimento do SUT_p representado na Figura 5.10 com a segunda versão (v_2) do plano de integração baseado nos resultados do cenário 1. Essa comparação entre os esquemas gráficos das versões dos planos de integração foi realizada para todos os cenários deste estudo. Os demais esquemas são apresentados na Seção E.3 do Apêndice E.

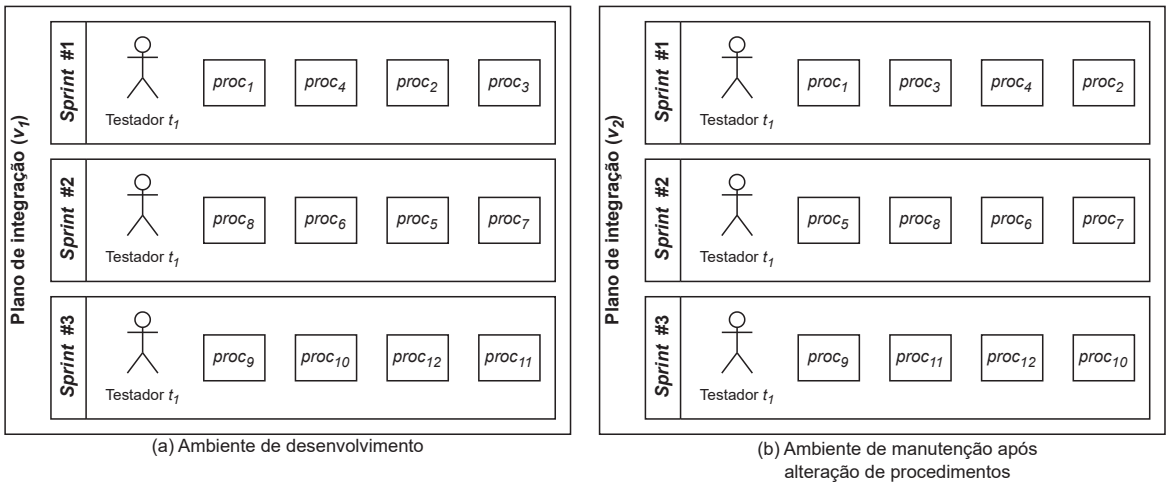


Figura 5.14: Cenário de teste 1 do Estudo 3: plano de integração.

No cenário 2, o plano de IA foi gerado pelo planejador em 0,047s, contendo 53 ações identificadas após a busca em 1.377 estados. A Figura E.8(b) ilustra o plano de integração correspondente. Nesse cenário, a integração foi distribuída ao Testador t_1 ao longo de quatro *sprints*. No *Sprint* #1, foram atribuídos os procedimentos $proc_1$, $proc_4$, $proc_2$ e $proc_3$. O *Sprint* #2 incluiu $proc_8$, $proc_6$, $proc_9$ e $proc_7$, enquanto no *Sprint* #3 foram designados $proc_5$, $proc_{13}$, $proc_{11}$ e $proc_{12}$. Por fim, no *Sprint* #4, foi atribuído o procedimento $proc_{10}$.

A geração do plano de IA para o cenário 3 ocorreu em 0,018s, resultando em 43 ações identificadas após a busca em 591 estados. A Figura E.9(b) ilustra a representação do plano de integração desse cenário. A integração foi atribuída ao Testador t_1 ao longo de três *sprints*. No *Sprint* #1, foram designados os procedimentos $proc_1$, $proc_3$, $proc_2$ e $proc_4$. O *Sprint* #2 incluiu $proc_8$, $proc_9$, $proc_6$ e $proc_5$, enquanto no *Sprint* #3 foram atribuídos $proc_7$ e $proc_{10}$.

5.4.3 Discussão dos resultados do Estudo 3

• **Atividade iv) Análise dos planos de integração**: após a catalogação dos dados sobre a geração dos planos, foi realizada uma análise dos principais resultados. com base na coluna "Plano de IA" da Tabela E.3, observa-se que foram gerados planos de IA factíveis para cenários de manutenção envolvendo alteração de procedimentos (cenário 1), inclusão de procedimentos (cenário 2) e remoção de procedimentos (cenário 3). Em todos esses casos, os planos de integração resultantes contêm ações que indicam corretamente uma ordem coerente com as características dos cenários.

A adequação dos planos de integração pode ser verificada por meio da comparação entre a primeira versão do plano de integração (v_1) e os planos de integração gerados para cada cenário (v_2), conforme ilustram as Figuras E.7, E.8 e E.9. No plano de integração do cenário 1, houve priorização da integração dos procedimentos modificados ($proc_3$, $proc_6$ e $proc_{11}$) em relação a v_1. No cenário 2, a integração do procedimento incluído ($proc_{13}$) foi priorizada. Já no cenário 3, os procedimentos $proc_3$ e $proc_8$, diretamente afetados pela remoção de outros procedimentos, foram priorizados na integração.

A abordagem TI-PIA foi desenvolvida para o contexto da fase de teste de integração do Modelo V (Figura 2.3) e contextualizada em *sprints*. No entanto, os resultados do Estudo 3 indicam que a abordagem pode ser explorada em processos da engenharia de software que ocorrem após a implantação do SUT, como os processos de manutenção. Diante dos resultados obtidos nesse contexto, compreende-se que a abordagem também pode apoiar atividades de versionamento de artefatos e evolução de software.

O Estudo 3 destacou que o versionamento do plano de integração pode contribuir para a rastreabilidade das mudanças no software. O versionamento de artefatos, parte do gerenciamento de configuração, pode viabilizar uma análise comparativa entre diferentes versões do software e facilitar a identificação dos impactos dessas mudanças. Nesse sentido, ferramentas de controle de versão, como o GitHub[2], poderiam ser utilizadas para armazenar e gerenciar versões dos planos de integração, garantindo o histórico e a recuperação de versões anteriores.

O Estudo 3 comparou o plano de integração elaborado para o desenvolvimento do SUT e versões do mesmo plano decorrentes de alterações no código. Essa aplicação dos planos de integração indica que a abordagem pode ser direcionada para os testes de regressão, auxiliando a verificar se alterações não afetaram adversamente o comportamento do SUT. Consequentemente, a abordagem pode contribuir para práticas de evolução contínua do software, assegurando que as mudanças sejam refletidas nos planos de integração.

5.5 AMEAÇAS À VALIDADE

Durante a condução dos estudos de viabilidade apresentados neste capítulo, foram identificadas algumas ameaças à validade dos resultados. Essas ameaças foram classificadas em validade interna, validade externa, validade de conclusão e validade do constructo, conforme a categorização de Wohlin et al. (2000). Ao longo da realização de cada estudo, foram adotadas estratégias para minimizar a influência dessas ameaças e reduzir possíveis riscos associados.

As ameaças à validade interna podem afetar o viés de um estudo devido a fatores de causalidade. Uma ameaça interna identificada foi o fato dos SUTs considerados no Estudo 1 e no Estudo 3 (Seções 5.2 e 5.4, respectivamente) serem simulações de projetos. Para mitigar esse viés nos resultados, as simulações de SUTs foram obtidas a partir de literatura consolidada sobre teste de software, como Myers et al. (2011).

As ameaças à validade externa estão relacionadas à generalização dos resultados. Uma ameaça externa identificada foi a condução dos três estudos em um ambiente acadêmico. O uso de artefatos e SUTs simplificados, sem a consideração de fatores organizacionais, pode fazer com que o ambiente acadêmico não reflita a realidade da prática industrial. Para reduzir os riscos associados a esse aspecto, o Estudo 2 (Seção 5.3) incluiu projetos de desenvolvimento de sistemas de software. Além disso, os cenários de teste foram projetados para replicar os experimentos em contextos variados, permitindo a comparação dos resultados obtidos.

As ameaças à validade de conclusão podem comprometer a capacidade de um estudo de gerar conclusões corretas. A principal ameaça de conclusão identificada foi o número reduzido

[2]Disponível em: `https://github.com`

de cenários de teste considerados nos três estudos apresentados neste capítulo. Esse fato pode limitar a identificação estatística de padrões nos dados obtidos. Assim, os resultados alcançados devem ser interpretados como indícios e não como conclusões definitivas.

As ameaças à validade do constructo dizem respeito à correspondência entre variáveis e conceitos teóricos. Uma ameaça do constructo identificada foi a ausência de participantes externos à pesquisa nos estudos conduzidos neste capítulo. Para minimizar os riscos decorrentes dessa limitação, todos os cenários de teste foram revisados por pares. Dois pesquisadores discutiram e refinaram iterativamente as definições desses cenários. Além disso, os resultados obtidos em cada estudo foram analisados e debatidos pelos mesmos pesquisadores.

5.6 CONSIDERAÇÕES DO CAPÍTULO

Este capítulo apresentou três estudos que avaliaram a viabilidade das representações PDDL e dos planos de integração da abordagem TI-PIA. As avaliações indicaram que a abordagem alcança a factibilidade da geração de planos de IA destacada por Ghallab et al. (2004). Os estudos alcançaram os objetivos específicos da avaliação da seguinte forma:

a) Avaliar as representações PDDL em diferentes cenários de teste de integração. Os estudos contemplaram tanto sistemas procedimentais quanto sistemas OO. Assim, a avaliação incluiu as estratégias de integração *top-down* e *bottom-up*, além dos testes intermétodos e interclasses. Os cenários analisados consideraram múltiplos testadores com diferentes capacidades de teste, variações nas chamadas de operações, tamanhos distintos de unidades e diferentes prioridades de integração.

   No Estudo 2, os tamanhos dos procedimentos foram definidos pelo número de linhas de código, enquanto o tamanho das classes foi determinado pela quantidade de atributos e métodos. As prioridades de integração foram estabelecidas com base no número de chamadas realizadas, na relevância dos procedimentos chamados e na quantidade de atributos e métodos. No Estudo 3, atribuiu-se maior prioridade a procedimentos alterados, novos procedimentos e procedimentos impactados por exclusões.

b) Discutir a qualidade dos planos de integração gerados. A análise da qualidade dos planos de integração foi conduzida sob a perspectiva da adequação funcional e da inteligibilidade. Do ponto de vista da adequação funcional, consideraram-se planos de integração de qualidade aqueles cujas ações estavam alinhadas às características dos cenários. Em relação à inteligibilidade, o Estudo 2 discutiu como as informações contidas nos planos de integração os tornam artefatos de software mais legíveis para os interessados.

c) Investigar a utilização dos planos de integração em contextos da ES. Os estudos avaliaram cenários em que as unidades foram atribuídas a *sprints*, aproximando o teste da realidade dos métodos ágeis de desenvolvimento de software. O Estudo 3 abordou a geração de planos de integração considerando processos de manutenção de software após a implantação do SUT. Embora a abordagem não tenha sido proposta para a manutenção, o Estudo 3 mostrou a viabilidade de seu uso no versionamento de artefatos e na evolução de software.

6 CONSIDERAÇÕES FINAIS

Durante esta tese, trabalhamos em um contexto de teste de integração que se mostra como um problema complexo que justifica o uso do planejamento em IA. A partir desse problema e de lacunas da literatura, propusemos a abordagem TI-PIA para o teste de integração de sistemas procedimentais e sistemas orientados a objetos (OO). Definimos uma estrutura de geração de planos de integração dividida em módulos cujos elementos centrais são representações do teste de integração com uma linguagem de planejamento em IA.

Esta tese foi guiada pela pergunta de pesquisa: "*Como o planejamento em IA pode auxiliar nas atividades de planejamento do teste de integração?*". Identificamos que o planejamento em IA permite que elementos do SUT, do teste de integração, do projeto e do processo de desenvolvimento de software sejam estruturados e representados para a geração de planos de integração. Esses planos são construídos com base nas informações coletadas pelo gerente de teste e podem servir como artefatos de apoio ao planejamento do teste. Além disso, podem auxiliar o testador na parametrização e operacionalização da execução do teste.

Os objetivos específicos desta tese foram alcançados da seguinte forma:

a) Delinear um panorama da utilização no planejamento em IA no teste de software. Conduzimos uma revisão bibliográfica que identificou 35 estudos sobre o uso do planejamento em IA no teste de software. A análise desses estudos permitiu identificar características dos SUTs, artefatos para instanciação das representações, atividades, fases e técnicas de teste utilizadas, linguagens de planejamento em IA, planejadores e planos. Durante a revisão, destacamos as principais motivações, vantagens e desvantagens do uso do planejamento em IA no teste de software.

b) Caracterizar soluções que aplicam o planejamento em IA no teste de integração. Realizamos uma análise de oito estudos relacionados ao teste de integração com planejamento em IA. Essa análise detalhou as representações do teste e os planos gerados nesses estudos. A partir de uma classificação por similaridade de conteúdo, categorizamos os estudos em estruturas baseadas em domínio, erros, UML e componentes. Com a análise realizada, indicamos lacunas de pesquisa usadas na definição da abordagem.

c) Propor uma estrutura de geração de planos de integração. Com base nas lacunas identificadas na literatura, definimos uma estrutura que é composta por módulos com os seguintes subobjetivos: (1) Planejamento do teste, módulo que coleta informações do SUT e do ambiente em documentações para instanciar representações PDDL; (2) Planejador, módulo responsável por acionar um planejador externo e gerar os planos de IA a partir das representações PDDL instanciadas; e (3) Execução do teste, módulo que engloba o plano de integração derivado do plano de IA e a infraestrutura necessária para a execução dos testes.

d) Representar o teste de integração em linguagem de planejamento em IA. Definimos modelagens para o teste de integração procedimental e OO, ambas representadas em PDDL 2.1. A modelagem do teste procedimental é ajustável às estratégias de integração *top-down* e *bottom-up*, enquanto a do teste OO se adapta aos testes intermétodos e interclasses. Além disso, as representações PDDL definidas consideram ambientes com um ou múltiplos testadores.

e) Avaliar elementos da abordagem em estudos experimentais. Realizamos a avaliação da abordagem a partir de três estudos de viabilidade. O Estudo 1 avaliou a viabilidade de elementos da lógica das representações PDDL. O Estudo 2 avaliou a viabilidade da instanciação em projetos de um sistema procedimental e de um sistema OO. O Estudo 3 avaliou a viabilidade da aplicação das representações na manutenção de software. Essas avaliações indicaram que as representações PDDL viabilizaram a geração de planos de IA factíveis e alinhados às características dos cenários avaliados.

As contribuições desta tese abrangem aspectos conceituais, metodológicos e técnicos. No aspecto *conceitual*, a revisão bibliográfica forneceu um panorama do estado da arte sobre o uso do planejamento em IA no teste de software, evidenciando sua aplicação no teste de integração. Essa análise permitiu a formulação de estruturas que relacionam esses conceitos e a identificação de lacunas de pesquisa. Além disso, a representação do conhecimento que ampliou os elementos do plano de integração constitui outra contribuição conceitual relevante.

A contribuição *metodológica* desta tese está relacionada à orientação sobre como realizar o teste de integração a partir da abordagem. Já a contribuição *técnica* refere-se à disponibilização dos artefatos gerados ao longo da pesquisa em repositórios abertos, que podem basear a aplicação da abordagem em outros contextos. As publicações científicas indicadas no Apêndice F contribuem para a divulgação do conhecimento gerado nesta tese.

Além das contribuições citadas, encontramos planos de integração mais completos do que os identificados na literatura. Nesta tese, incluímos nas representações dos planos gerados elementos do SUT, do teste de integração e do projeto e processo de desenvolvimento de software que não são considerados por outros autores. Entendemos que a geração do plano de integração com mais elementos indica a complexidade demandada pelo planejamento em IA.

Com relação aos SUTs, incluímos no plano de integração: parâmetros de entrada; tipos de dados; variáveis de saída; chamadas de operações entre os procedimentos; e as trocas de mensagens entre métodos. Quanto ao teste de integração, representamos no plano de integração: estratégia *top-down*; estratégia *bottom-up*; teste intermétodos; teste interclasses; e prioridades de integração específicas para os contextos procedimental e OO. No que diz respeito ao processo e ao projeto de software, representamos: *sprints*; capacidades de teste; e múltiplos testadores.

Esta tese buscou garantir rigor, responsabilidade e reprodutibilidade. Asseguramos o *rigor* com a fundamentação teórica baseada na literatura, com o uso de metodologias consolidadas nos estudos de viabilidade e com o processo de revisão por pares em todas as etapas. Atingimos a *responsabilidade* com o compartilhamento dos artefatos em repositórios abertos, proporcionando transparência à pesquisa. Viabilizamos a *reprodutibilidade* dos estudos realizados com a disponibilização dos artefatos e a descrição detalhada da abordagem.

Esta tese explorou como representar em PDDL elementos do SUT, do teste de integração e do ambiente. Durante o trabalho, conduzimos um estudo criterioso e detalhado que alcançou uma representação abrangente desses elementos. As avaliações dessa representação indicaram a factibilidade da geração de planos de IA e a viabilidade dos planos de integração. Dessa forma, entendemos que os estudos realizados e a abordagem delineada são promissores e necessários para a implementação de ferramentas automáticas de teste de integração usando IA.

O uso da abordagem pode ser adaptado para cenários de métodos prescritivos e de métodos ágeis de desenvolvimento. Esperamos que a abordagem possa ser adotada por usuários com pouca familiaridade em IA, já que a representação é parcialmente pré-configurada. Acreditamos que a abordagem não esteja limitada a nenhum tipo específico de linguagem ou *framework* de desenvolvimento.

As limitações desta tese incluem a falta de automatização na abordagem, o que pode impactar sua aplicabilidade prática. Além disso, as modelagens são restritas aos aspectos gerais

da integração procedimental e OO, sem aprofundar particularidades específicas desses testes. Outra limitação está na definição das representações, que foram baseadas nas funcionalidades da PDDL 2.1. O uso dessa versão da linguagem pode restringir o das representações em contextos que demandam recursos presentes em versões mais avançadas da linguagem.

Outras limitações identificadas são referentes aos estudos de viabilidade realizados. Os estudos conduzidos não envolveram participantes externos, o que pode ter limitado a diversidade das perspectivas analisadas. Além disso, a ausência de investigações voltadas para a execução do teste impede uma avaliação mais completa da abordagem em cenários práticos. Por fim, a utilização de valores constantes e variáveis para as prioridades pode ter restringido a análise dos resultados obtidos, não refletindo situações encontradas na prática do teste de integração.

6.1 TRABALHOS FUTUROS

Indicamos como trabalhos futuros decorrentes desta tese:

- **Desenvolver uma ferramenta automatizada para a abordagem**: elaborar uma ferramenta com interface gráfica que realize a instanciação dinâmica da representação PDDL. Além disso, essa ferramenta pode ser responsável pelo processamento dos planos de IA gerados pelo planejador, eliminando as ações utilizadas como *log* e direcionando as informações relevantes para a construção de um plano de integração. Por fim, a ferramenta também pode exibir uma representação gráfica desse plano de integração.

- **Utilizar extensões do planejamento em IA na abordagem**: as modelagens podem ser ajustadas para incorporar funcionalidades específicas de diferentes versões da PDDL indicados na Seção 2.4.2, como as noções temporais da PDDL2.1 e as restrições fortes e fracas da PDDL3.0. Outra possibilidade é a introdução do não determinismo indicado na Tabela 2.1. Essas modificações podem contribuir para uma representação mais precisa do sequenciamento das atividades envolvidas no teste.

- **Adaptar a abordagem para a integração baseada em colaborações**: as modelagens podem ser ajustadas para viabilizar o teste de integração baseado em colaborações. Nesse contexto, uma colaboração é definida como um grupo de unidades que interagem para executar uma ação do SUT.

- **Adaptar a abordagem para a integração de microsserviços e sistemas de software completos**: as modelagens podem ser expandidas para contemplar a integração de microsserviços e sistemas de software completos, contribuindo para a melhoria da interoperabilidade.

- **Ampliar a modelagem OO da abordagem**: A modelagem do teste orientado a objetos pode ser ampliada com a representação de aspectos inerentes a sistemas OO, como herança, polimorfismo e encapsulamento, além de aprofundar a dinâmica de troca de mensagens.

- **Conduzir estudos sobre a execução do teste de integração**: novos estudos podem explorar o módulo (3) Execução do teste, ilustrado na Figura 4.2. Esses estudos podem investigar a associação dos planos de integração com a geração de dados de teste, o uso de *stubs* e *drivers* e a execução de casos de teste, contribuindo para a validação prática da abordagem na execução do teste.

- **Realizar estudos envolvendo usuários**: novos estudos podem avaliar os artefatos da abordagem sob a perspectiva dos usuários. Esses estudos podem analisar indicadores do Modelo TAM (*Technology Acceptance Model*) (Davis, 1989; Venkatesh e Davis, 2000), como utilidade percebida, facilidade de uso e intenção de uso. Além disso, esses estudos podem investigar a abordagem em relação às vantagens, desvantagens e motivações para o uso do planejamento em IA no teste de software apresentadas na Tabela 3.1.

- **Associar a abordagem à integração contínua**: estudos futuros podem explorar a utilização dos planos de integração gerados pela abordagem em conjunto com práticas de integração contínua, por meio do uso de ferramentas automáticas de gerenciamento de configuração de software.

- **Associar a abordagem à gerência de implementação de requisitos**: o mapeamento entre requisitos e unidades pode ser expandido no contexto da abordagem, aprimorando a rastreabilidade no desenvolvimento de software. Nesse sentido, o Design Socialmente Consciente (DSC) (Baranauskas, 2014) pode ser integrado à abordagem, viabilizando o uso de requisitos que consideram aspectos sociais.

- **Instanciar a representação PDDL com prioridades**: novos estudos podem definir critérios, métricas ou heurísticas para estabelecer prioridades usadas na instanciação da representação PDDL. Para isso, podem ser consideradas ferramentas de apoio, como o Jenkins (Jenkins, 2025) e o SonarQube (SonarQube, 2025), que possibilitam a análise de fatores como o número de linhas de código e a complexidade do software.

- **Realizar experimentos de geração de planos**: novos experimentos podem ser conduzidos com o objetivo de gerar planos de integração utilizando outros planejadores (por exemplo, não determinísticos) e outras formas de representação ou teorias (por exemplo, técnicas de *machine learning*). Os planos obtidos nesses experimentos podem ser usados em análises comparativas com os planos gerados pelo planejador determinístico considerado na abordagem.

REFERÊNCIAS

Albore, A., Palacios, H. e Geffner, H. (2009). A translation-based approach to contingent planning. Em *International Joint Conferences on Artificial Intelligence*, volume 9, páginas 1623–1628.

Anderson, J. S. e Fickas, S. (1989). A proposed perspective shift: Viewing specification design as a planning problem. *ACM SIGSOFT Software Engineering Notes*, 14(3):177–184. DOI: 10.1145/75199.75227.

Andrews, A. K. A., Zhu, C., Scheetz, M., Dahlman, E. e Howe, A. E. (2002). Ai planner assisted test generation. *Software Quality Journal*, 10(3):225–259. DOI: 10.1023/A:1021686406575.

Baranauskas, M. C. C. (1993). Procedimento, função, objeto ou lógica? Linguagens de programação vistas pelos seus paradigmas. *Computadores e Conhecimento: Repensando a Educação. Campinas, SP, Gráfica Central da Unicamp*.

Baranauskas, M. C. C. (2014). Socially aware computing. Em *Proceedings of International Conference on Engineering and Computer Education*, volume 6.

Basili, V. R. e Rombach, H. D. (1988). The TAME project: Towards improvement-oriented software environments. *IEEE Transactions on SE*, 14(6):758–773. DOI: 10.1109/32.6156.

Beizer, B. (1990). *Software testing techniques - 2 ed.* Itp - Media. ISSN: 1850328803.

Binder, R. (2000). *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional. ISSN: 0201809389.

Binder, R. (2018). Sei observations and reference model for software integration labs. *Software Engineering Institute, Carnegie Mellon University*. DOI: 10.13140/RG.2.2.17661.20968.

Blum, A. L. e Furst, M. L. (1997). Fast planning through planning graph analysis. *Artificial intelligence*, 90(1-2):281–300. DOI: 10.1016/S0004-3702(96)00047-1.

Boddy, M. S., Gohde, J., Haigh, T. e Harp, S. A. (2005). Course of action generation for cyber security using classical planning. Em *ICAPS*, páginas 12–21. ISBN: 1577352203.

Bonet, B. e Geffner, H. (2011). Planning under partial observability by classical replanning: Theory and experiments. Em *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence*, páginas 1936–1941. DOI: 10.5591/978-1-57735-516-8/IJCAI11-324.

Borges Ruy, F., Almeida Falbo, R. d., Perini Barcellos, M., Dornelas Costa, S. e Guizzardi, G. (2016). Seon: A software engineering ontology network. Em *European Knowledge Acquisition Workshop*, páginas 527–542. Springer. DOI: 10.1007/978-3-319-49004-5_34.

Bozic, J., Kleine, K., Simos, D. E. e Wotawa, F. (2017). Planning-based security testing of the SSL/TLS protocol. Em *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, páginas 347–355. IEEE. DOI: 10.1109/ICSTW.2017.63.

Bozic, J., Tazl, O. A. e Wotawa, F. (2019). Chatbot testing using AI planning. Em *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)*, páginas 37–44. IEEE. DOI: 10.1109/AITest.2019.00-10.

Bozic, J. e Wotawa, F. (2015). Purity: a Planning-based secURITY testing tool. Em *2015 IEEE International Conference on Software Quality, Reliability and Security-Companion*, páginas 46–55. IEEE. DOI: 10.1109/QRS-C.2015.19.

Bozic, J. e Wotawa, F. (2018a). Planning-based security testing for chatbots. *INFORMACIJSKA DRUŽBA-IS 2018*, página 23. DOI: 10.1007/s11219-019-09469-y.

Bozic, J. e Wotawa, F. (2018b). Planning-based security testing of web applications. Em *Proceedings of the 13th International Workshop on Automation of Software Test*, páginas 20–26. ACM. DOI: 10.1145/3194733.3194738.

Bozic, J. e Wotawa, F. (2019). Software testing: According to plan! Em *IEEE International Conference on Software Testing, Verification and Validation Workshops*, páginas 23–31. IEEE. DOI: 10.1109/ICSTW.2019.00028.

Bozic, J. e Wotawa, F. (2020). Planning-based security testing of web applications with attack grammars. *Software Quality Journal*, 28(1):307–334. DOI: 10.1007/s11219-019-09469-y.

Burnstein, I. (2006). *Practical software testing: a process-oriented approach*. Springer Science & Business Media. ISBN: 0-387-95131-8.

Corbin, J. e Strauss, A. (2014). *Basics of qualitative research: Techniques and procedures for developing grounded theory*. Sage publications. ISBN: 9781412906449.

Davis, F. D. (1989). Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology. *MIS quarterly*, páginas 319–340.

Delamaro, M. E., Maldonado, J. C. e Jino, M. (2016). *Introdução ao teste de software. 2. ed.* Elsevier. ISBN: 978-85-352-8352-5.

Ding, Y., Zhang, Y., Yuan, G., Jiang, S. e Dai, W. (2022). Progress on class integration test order generation approaches: A systematic literature review. *Information and Software Technology*, página 107133. DOI: 10.1016/j.infsof.2022.107133.

Do, M. e Kambhampati, S. (2003). Sapa: A multi-objective metric temporal planner. *Journal of Artificial Intelligence Research*, 20:155–194. DOI: 10.1613/jair.1156.

Edelkamp, S. e Hoffmann, J. (2004). Pddl2. 2: The language for the classical part of the 4th international planning competition. Relatório técnico, Technical Report 195, University of Freiburg.

Engström, E., Storey, M.-A., Runeson, P., Höst, M. e Baldassarre, M. T. (2020). How software engineering research aligns with design science: a review. *Empirical Software Engineering*, 25(4):2630–2660. DOI: 10.1007/s10664-020-09818-7.

Feather, M. S. e Smith, B. (2001). Automatic generation of test oracle from pilot studies to application. *Automated Software Engineering*, 8(1):31–61. DOI: 10.1023/A:1008711707946.

Fikes, R. E. e Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208. DOI: 10.1016/0004-3702(71)90010-5.

Fox, M. e Long, D. (2003). Pddl2. 1: An extension to pddl for expressing temporal planning domains. *Journal of artificial intelligence research*, 20:61–124. DOI: 10.1613/jair.1129.

Galler, S. J., Zehentner, C. e Wotawa, F. (2010). AIana: An AI planning system for test data generation. Em *Proceedings of the 1st Workshop on Testing Object-Oriented Systems*, páginas 1–8. DOI: 10.1145/1890692.1890697.

Garousi, V., Felderer, M. e Mäntylä, M. V. (2019). Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Information and software technology*, 106:101–121. DOI: 10.1016/j.infsof.2018.09.006.

Garousi, V., Rainer, A., Lauvås Jr, P. e Arcuri, A. (2020). Software-testing education: A systematic literature mapping. *Journal of Systems and Software*, 165:110570. DOI: 10.1016/j.jss.2020.110570.

Gerevini, A. e Long, D. (2005). Plan constraints and preferences in pddl3: The language of the fifth international planning competition. university of brescia. Relatório técnico, Technical Report, University of Brescia.

Ghallab, M., Nau, D. e Traverso, P. (2004). *Automated Planning: theory and practice*. Elsevier. ISNB: 1558608567.

Gouveia, L. B. e Ranito, J. (2004). *Sistemas de informação de apoio à gestão*. Sociedade Portuguesa de Inovação. ISBN: 972-8589-43-3.

Gupta, M., Fu, J., Bastani, F. B., Khan, L. R. e Yen, I.-L. (2007). Rapid goal-oriented automated software testing using MEA-graph planning. *Software Quality Journal*, 15(3):241–263. DOI: 10.1007/s11219-007-9018-3.

Halsey, K., Long, D. e Fox, M. (2004). Crikey-a temporal planner looking at the integration of scheduling and planning. Em *Workshop on Integrating Planning into Scheduling, ICAPS*, páginas 46–52. Citeseer.

Harrold, M. J. e Rothermel, G. (1994). Performing data flow testing on classes. *ACM SIGSOFT Software Engineering Notes*, 19(5):154–163. DOI: 10.1145/195274.195402.

Helmert, M. (2008). Changes in PDDL 3.1. `https://ipc08.icaps-conference.org/deterministic/PddlExtension.html`. Acesso em: 20 maio 2024.

Hoffmann, J. (2003a). Metric-FF. Disponível em: `https://fai.cs.uni-saarland.de/hoffmann/metric-ff.html`. Acesso em: 20 jul. 2023.

Hoffmann, J. (2003b). The Metric-FF planning system: translating "ignoring delete lists" to numeric state variables. *Journal of artificial intelligence research*, 20:291–341. DOI: 10.1613/jair.1144.

Hoffmann, J. e Nebel, B. (2001). The FF planning system: fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302. DOI: 10.1613/jair.855.

Howe, A. E., von Mayrhauser, A. e Mraz, R. T. (1995). Test sequences as plans: An experiment in using an AI planner to generate system tests. Em *Proceedings of the 10th Knowledge-Based Software Engineering Conference*, páginas 184–191. IEEE. DOI: 10.1109/KBSE.1995.490134.

Howe, A. E., Von Mayrhauser, A. e Mraz, R. T. (1997). Test case generation as an AI planning problem. Em *Knowledge-Based Software Engineering*, páginas 77–106. Springer. DOI: 10.1007/978-0-585-34714-1_5.

Hsu, C.-W., Wah, B. W., Huang, R. e Chen, Y. (2006). Handling soft constraints and goals preferences in SGPlan. Em *Proceedings of the ICAPS Workshop on Preferences and Soft Constraints in Planning*.

Humphrey, W. S. (1989). *Managing the software process*. Addison-Wesley Longman Publishing Co., Inc. ISBN: 0-201-18095-2.

Jenkins (2025). Jenkins: The leading open-source automation server. Acesso em: 24 fev. 2025.

Jensen, R. M. (2003). *Efficient BDD-based planning for non-deterministic, fault-tolerant, and adversarial domains*. Carnegie Mellon University.

Kitchenham, B. e Charters, S. (2007). Guidelines for performing systematic literature reviews in software engineering. *Citeseer*. Technical Report, EBSE-2007-01.

Koehler, J., Nebel, B., Hoffmann, J. e Dimopoulos, Y. (1997). Extending planning graphs to an adl subset. Em *European Conference on Planning*, páginas 273–285. Springer. DOI: 10.1007/3-540-63912-8_92.

Leitner, A. e Bloem, R. (2005). Automatic testing through planning. *Technische Universität Graz, Institute for Software Technology, Tech. Rep*.

Lewis, W. E. (2004). *Software testing and continuous quality improvement*. Auerbach publications. ISBN: 0-8493-2524-2.

Li, L., Wang, D., Shen, X. e Yang, M. (2009). A method for combinatorial explosion avoidance of AI planner and the application on test case generation. Em *International Conference on Computational Intelligence and Software Engineering*, páginas 1–4. IEEE. DOI: 10.1109/CISE.2009.5365557.

Lima, L. F. d. (2020). Teste de intrusão para aplicações web: um método com planejamento em inteligência artificial. Dissertação (Mestrado em Informática) - Programa de Pós-graduação em Informática, Universidade Federal do Paraná. Disponível em: `https://hdl.handle.net/1884/66754`.

Lima, L. F. d. (2023). Uma abordagem de teste de integraçao com planejamento em inteligência artificial. Em *Anais do XXVI Congresso Ibero-Americano em Engenharia de Software*, páginas 261–268. SBC. DOI: 10.5753/cibse.2023.24710.

Lima, L. F. d., Grégio, A. R. A. e Peres, L. M. (2020a). Teste de intrusão para aplicações web: um método com planejamento em inteligência artificial. Em *Mostra de Trabalhos de Pós-graduandos em Ciência da Computação do Paraná do II Fórum dos Programas de Pós-Graduação em Computação do Paraná (II ForPPGC-PR)*.

Lima, L. F. d., Horstmann, M. C., Neto, D. N., Grégio, A. R., Silva, F. e Peres, L. M. (2020b). On the challenges of automated testing of web vulnerabilities. Em *2020 IEEE 29th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, páginas 203–206. IEEE. DOI: 10.1109/WETICE49692.2020.00047.

Lima, L. F. d., Huve, C. A. e Peres, L. M. (2020c). Software product quality evaluation guide for electronic health record systems. Em *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*, páginas 108–113. DOI: 10.1145/3422392.3422478.

Lima, L. F. d., Meireles, F. S. S. e Peres, L. M. (2021). Relatório técnico de estudos sobre sistemas de saúde de código aberto. *Departamento de Informática, Universidade Federal do Paraná*. Disponível em: https://zenodo.org/records/14176028.

Lima, L. F. d., Meireles, F. S. S. e Peres, L. M. (2023). Tecnologias de engenharia de software para o desenvolvimento de sistemas de saúde de código aberto: um mapeamento sistemático da literatura. *Journal of Health Informatics*, 15(Especial). DOI: 10.59681/2175-4411.v15.iEspecial.2023.1079.

Lima, L. F. d. e Peres, L. M. (2021a). Guia de aplicação de um protocolo de mapeamento sistemático para busca de aplicativos de saúde em repositórios não-acadêmicos - Perfil desenvolvedor de software. *Departamento de Informática, Universidade Federal do Paraná*. DOI: 10.5281/zenodo.11620098.

Lima, L. F. d. e Peres, L. M. (2021b). Guia de aplicação de um protocolo de mapeamento sistemático para busca de aplicativos de saúde em repositórios não-acadêmicos - Perfil profissional de saúde. *Departamento de Informática, Universidade Federal do Paraná*. DOI: 10.5281/zenodo.11620098.

Lima, L. F. d. e Peres, L. M. (2021c). Proposta de uso de técnicas de inteligência artificial para a avaliação de qualidade de sistemas de saúde. *Mostra de Trabalhos de Pós-graduandos em Ciência da Computação do Paraná do III Fórum dos Programas de Pós-Graduação em Computação do Paraná*.

Lima, L. F. d. e Peres, L. M. (2021d). Protocolo de mapeamento sistemático para busca de aplicativos de saúde em repositórios não-acadêmicos. Em *Anais do I Workshop de Práticas de Ciência Aberta para Engenharia de Software*, páginas 7–12. SBC. DOI: 10.5753/openscience.2021.17138.

Lima, L. F. d. e Peres, L. M. (2022). A protocol based on systematic mapping studies for searching health applications in non-academic repositories. *SBC Reviews on Computer Science*, 2(1). DOI: 10.5753/reviews.2022.2724.

Lima, L. F. d. e Peres, L. M. (2023). Teste de integração com planejamento em inteligência artificial. Em *V Fórum dos Programas de Pós-Graduação em Computação do Paraná (V ForPPGC-PR)*.

Lima, L. F. d., Silva, F., Grégio, A. R. A. e Peres, L. M. (2020d). A systematic literature mapping of artificial intelligence planning in software testing. Em *International Conference on Software Technologies*, páginas 152–159. DOI: 10.5220/0009829501520159.

Masiero, P. C., Lemos, O. A. L., Ferrari, F. C. e Maldonado, J. C. (2006). Teste de software orientado a objetos e a aspectos: teoria e prática. *Atualizações em informática. Rio de Janeiro: Ed. PUC*, páginas 13–72.

McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D. e Wilkins, D. (1998). PDDL - The Planning Domain Definition Language. *Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational*.

Meireles, F. S. S., Lima, L. F. d. e Peres, L. M. (2022). Catálogo de tecnologias abertas de telessaúde. *Departamento de Informática, Universidade Federal do Paraná*. Disponível em: https://acervodigital.ufpr.br/handle/1884/79572.

Memon, A. M., Pollack, M. E. e Soffa, M. L. (1999). Using a goal-driven approach to generate test cases for GUIs. Em *Proceedings of the 1999 International Conference on Software Engineering*, páginas 257–266. IEEE. DOI: 10.1145/302405.302632.

Memon, A. M., Pollack, M. E. e Soffa, M. L. (2000). A planning-based approach to GUI testing. *Proceedings of the 13th International Software/Internet Quality Week*.

Memon, A. M., Pollack, M. E. e Soffa, M. L. (2001). Hierarchical GUI test case generation using automated planning. *IEEE Transactions on Software Engineering*, 27(2):144–155. DOI: 10.1109/32.908959.

Meneguzzi, F. (2010). JavaGP - Java Implementation of Graphplan. `https://github.com/pucrs-automated-planning/javagp`. Acesso em: 25 nov. 2024.

Mraz, R. T., Howe, A. E., von Mayrhauser, A. e Li, L. (1995). System testing with an AI planner. Em *Proceedings of the 6th International Symposium on Software Reliability Engineering*, páginas 96–105. IEEE. DOI: 10.1109/ISSRE.1995.497648.

Myers, G. J. (1979). The art of software testing. *ISBN: 0-471-04328-1*.

Myers, G. J., Sandler, C. e Badgett, T. (2011). *The art of software testing, 3ª edition*. John Wiley & Sons. ISBN: 978-1-118-03196-4.

Obes, J. L., Sarraute, C. e Richarte, G. (2013). Attack planning in the real world. *arXiv: 1306.4044*.

Paradkar, A. M., Sinha, A., Williams, C., Johnson, R. D., Outterson, S., Shriver, C. e Liang, C. (2007). Automated functional conformance test generation for semantic web services. Em *IEEE International Conference on Web Services (ICWS 2007)*, páginas 110–117. IEEE. DOI: 10.1109/ICWS.2007.48.

Pednault, E. P. (1989). ADL: Exploring the middle ground between STRIPS and the situation calculus. *Kr*, 89:324–332. ISBN: 1558600329.

Penberthy, J. S. e Weld, D. S. (1992). UCPOP: A sound, complete, partial order planner for ADL. *Kr*, 92:103–114. ISBN: 1558602623.

Penberthy, J. S. e Weld, D. S. (1993). UCPOP. Disponível em: `http://aiweb.cs.washington.edu/ai/ucpop.html`. Acesso em: 20 jul. 2023.

Pereira, F. C., Neto, G. B., De Lima, L. F., Silva, F. e Peres, L. M. (2022). A tool for software requirement allocation using artificial intelligence planning. Em *2022 IEEE 30th International Requirements Engineering Conference (RE)*, páginas 257–258. DOI: 10.1109/RE54965.2022.00032.

Pressman, R. S. e Maxim, B. R. (2021). *Engenharia de software - 9ª edição*. AMGH. ISBN: 978-6558040101.

Razavi, N., Farzan, A. e McIlraith, S. A. (2014). Generating effective tests for concurrent programs via AI automated planning techniques. *International Journal on Software Tools for Technology Transfer*, 16(1):49–65. DOI: 10.1007/s10009-013-0277-y.

Richter, S. e Westphal, M. (2008). The LAMA planner using landmark counting in heuristic search. Em *Proceedings of IPC*, volume 14.

Russell, S. J. e Norvig, P. (2022). *Artificial intelligence: A modern approach - Fourth Edition*. Pearson Education. ISNB: 1-292-40113-3.

Sarraute, C., Buffet, O. e Hoffmann, J. (2012). POMDPs make better hackers: Accounting for uncertainty in penetration testing. Em *Twenty-Sixth AAAI Conference on Artificial Intelligence*. DOI: 10.1609/aaai.v26i1.8363.

Scheetz, M., von Mayrhauser, A. e France, R. (1999). Generating test cases from an OO model with an AI planning system. Em *Proceedings of the 10th International Symposium on Software Reliability Engineering*, páginas 250–259. IEEE. 10.1109/ISSRE.1999.809330.

Schnelte, M. (2009). Generating test cases for timed systems from controlled natural language specifications. Em *IEEE International Conference on Secure Software Integration and Reliability Improvement*, páginas 348–353. IEEE. DOI: 10.1109/SSIRI.2009.58.

Schnelte, M. e Güldali, B. (2010). Test case generation for visual contracts using AI planning. *INFORMATIK 2010. Service Science–Neue Perspektiven für die Informatik. Band 2*. ISBN: 978-3-88579-270-3.

SEON (2017). Seon: Software engineering ontology network (versão 1.0.5). `https://dev.nemo.inf.ufes.br/seon`. Acesso em: 28 jun. 2022.

Shmaryahu, D., Shani, G., Hoffmann, J. e Steinmetz, M. (2018). Simulated penetration testing as contingent planning. Em *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 28. DOI: 10.1609/icaps.v28i1.13902.

Siegel, S. (1996). *Object oriented software testing: a hierarchical approach*. John Wiley & Sons, Inc. ISBN: 0-471-13749-9.

Silva, C. E. d. e Lemos, R. d. (2011). Dynamic plans for integration testing of self-adaptive software systems. Em *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-managing Systems*, páginas 148–157. DOI: 10.1145/1988008.1988029.

Simos, D. E., Bozic, J., Garn, B., Leithner, M., Duan, F., Kleine, K., Lei, Y. e Wotawa, F. (2019). Testing TLS using planning-based combinatorial methods and execution framework. *Software Quality Journal*, 27(2):703–729. DOI: 10.1007/s11219-018-9412-z.

SonarQube (2025). Sonarqube: Continuous inspection for code quality. Acesso em: 24 fev. 2025.

Souza, É. F. d., Falbo, R. d. A. e Vijaykumar, N. L. (2017). Roost: Reference ontology on software testing. *Applied Ontology*, 12(1):59–90. DOI: 10.3233/AO-170177.

Spillner, A. e Bremenn, H. (2002). The w-model. strengthening the bond between development and test. Em *Int. Conf. on Software Testing, Analysis and Review*, páginas 15–17.

Strauss, A. e Corbin, J. (1998). *Basics of qualitative research techniques*. Citeseer. ISBN: 0803959397.

Venkatesh, V. e Davis, F. D. (2000). A Theoretical Extension of the Technology Acceptance Model: Four Longitudinal Field Studies. *Management science*, 46(2):186–204.

Vincenzi, A. M. R. (2004). *Orientaçao a objeto: Definiçao, implementaçao e análise de recursos de teste e validaçao*. Tese de doutorado - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo. DOI: 10.11606/T.55.2004.tde-17082004-122037.

von Mayrhauser, A., Scheetz, M. e Dahlman, E. (1999). Generating goal-oriented test cases. Em *Compsac*, página 110. IEEE. DOI: 10.1109/CMPSAC.1999.812687.

von Mayrhauser, A., Scheetz, M., Dahlman, E. e Howe, A. E. (2000). Planner based error recovery testing. Em *Proceedings of the 11th International Symposium on Software Reliability Engineering*, páginas 186–195. IEEE. DOI: 10.1109/ISSRE.2000.885871.

W3C (2008). Extensible Markup Language (XML). `https://www.w3.org/XML`. Acesso em: 28 fev. 2022.

Wazlawick, R. (2013). *Engenharia de software: Conceitos e práticas*, volume 1. Elsevier Brasil. ISBN: 978-85-352-6120-2.

Wohlin, C. (2014). Guidelines for snowballing in systematic literature studies and a replication in software engineering. Em *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, páginas 1–10. DOI: 10.1145/2601248.2601268.

Wohlin, C., Runeson, P., Host, M., Ohlsson, M. C. e Regnell, B. (2000). Experimentation in software engineering - An introduction. *Doedrecht the Netherlands*. ISSN: 1384-6469.

Wotawa, F. e Bozic, J. (2014). Plan it! Automated security testing based on planning. Em *IFIP International Conference on Testing Software and Systems*, páginas 48–62. Springer. DOI: 10.1007/978-3-662-44857-1_4.

Wu, J. (2011). Improving the writing of research papers: IMRAD and beyond. DOI: 10.1007/s10980-011-9674-3.

Würsch, M., Ghezzi, G., Hert, M., Reif, G. e Gall, H. C. (2012). Seon: a pyramid of ontologies for software evolution and its applications. *Computing*, 94(11):857–885. DOI: 10.1007/s00607-012-0204-1.

Yen, I.-L., Bastani, F. B., Mohamed, F., Ma, H. e Linn, J. (2002). Application of AI planning techniques to automated code synthesis and testing. Em *Proceedings of the 14th IEEE International Conference on Tools with Artificial Intelligence*, páginas 131–137. IEEE. DOI: 10.1109/TAI.2002.1180797.

Younes, H. L. e Littman, M. L. (2004). PPDDL1. 0: An extension to PDDL for expressing planning domains with probabilistic effects. *Techn. Rep. CMU-CS-04-162*, 2:99.

APÊNDICE A – REVISÃO BIBLIOGRÁFICA DA LITERATURA

Esta revisão bibliográfica da literatura investiga estudos que aplicam técnicas de planejamento em IA no teste de software. São apresentados a metodologia utilizada na condução da revisão (Seção A.1), os resultados da extração de dados nos estudos (Seção A.2) e as ameaças à validade da revisão (Seção A.3).

A.1 METODOLOGIA

Esta revisão bibliográfica objetiva obter um panorama da aplicação do planejamento em IA no teste de software. Assim, a revisão é direcionada pela seguinte questão de pesquisa: "*Como o planejamento em IA tem apoiado o teste de software?*". A Figura A.1 descreve os processos de seleção e análise de estudos.

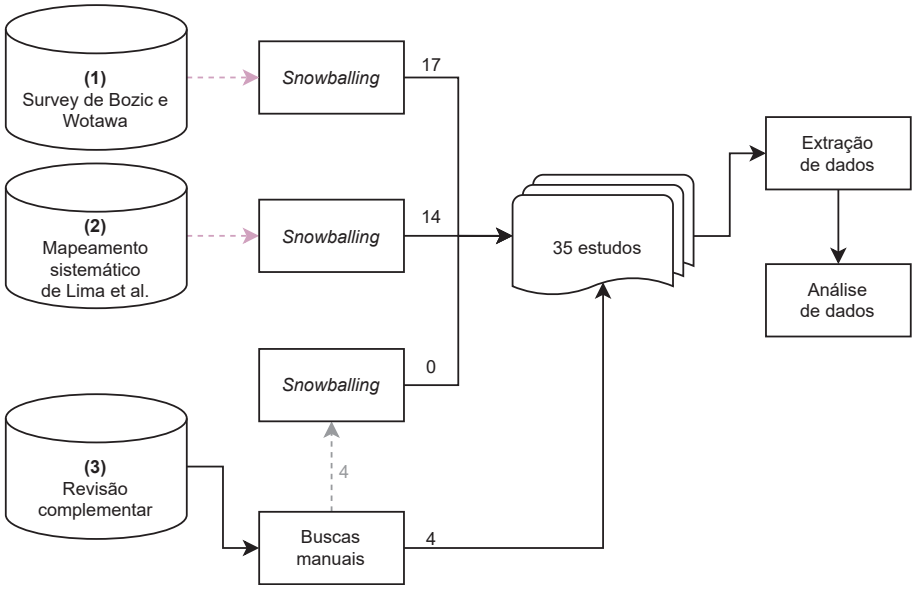


Figura A.1: Metodologia da seleção de estudos da revisão bibliográfica.

A seleção de estudos foi conduzida com processos de *snowballing* reverso (do inglês, *backward snowballing*). De acordo com Wohlin (2014), *snowballing* reverso consiste na análise de referências de estudos relevantes para identificar outros estudos elegíveis para uma pesquisa. Foram utilizadas as referências de três fontes principais para uma iteração do *snowballing* reverso: (1) o *survey* de Bozic e Wotawa (2019); (2) o mapeamento sistemático da literatura de Lima et al. (2020d); e (3) uma revisão bibliográfica complementar.

O *snowballing* identificou 17 estudos nas referências de Bozic e Wotawa (2019) (fonte 1) e 14 estudos a partir das referências de Lima et al. (2020d) (fonte 2). A revisão complementar (fonte 3) foi realizada com buscas manuais no Google Acadêmico[1] em junho de 2022, sendo posteriormente revisada em novembro de 2024. Essas buscas consideraram palavras-chave que derivam da expressão de busca usada por Lima et al. (2020d): ((“teste de software” *OR* “*software testing*”) *AND* (“planejamento em IA” *OR* “*AI planning*”)).

[1]Disponível em: `https://scholar.google.com.br`

Foram identificados 4 estudos nas buscas manuais. O *snowballing* realizado nas referências desses 4 estudos não identificou outros estudos elegíveis. Dessa maneira, a seleção de estudos identificou 35 estudos para análise. O Apêndice B sumariza os estudos identificados. Foram incluídos estudos escritos em português e inglês. O idioma inglês foi considerado por ser adotado pela grande maioria das conferências sobre o assunto da pesquisa e o idioma português por ser a língua nativa dos autores.

Foram incluídos estudos revisados por pares como artigos de periódicos e conferências e dissertações. Estudos classificados como literatura cinzenta (Garousi et al., 2019), como *preprints* e relatórios técnicos, não foram incluídos. Também foram desconsiderados mapeamentos ou revisões sistemáticas, revisões bibliográficas, *surveys* e compilações de outros estudos já incluídos.

Foram definidas questões sobre sistemas sob teste, artefatos, atividades de teste, fases de teste, técnicas de teste, linguagens de planejamento, planejadores e planos de IA. A Tabela A.1 descreve o objetivo de cada questão, as questões elaboradas para cada aspecto e as possíveis respostas de cada questão.

Tabela A.1: Questões da revisão bibliográfica estruturadas com o GQM.

Objetivo	**Questão**	**Respostas**
Identificar características, funcionalidades, contextos de uso, plataformas de execução e paradigmas de programação do SUT	**Q1. Sistemas sob teste**) Quais SUTs têm sido usados no teste de software com planejamento em IA?	Questão aberta
Identificar artefatos textuais ou gráficos usados para obter informações usadas na representação do conhecimento ou geração do planos de IA	**Q2. Artefatos**) Quais artefatos têm sido usados para gerar planos de IA?	Questão aberta
Identificar quais atividades de teste são consideradas nos estudos	**Q3. Atividades de teste**) Quais atividades de teste têm sido apoiadas por planejamento em IA?	Categorização das atividade de teste de Souza et al. (2017): (a) planejamento do teste; (b) design de casos de teste; (c) codificação do teste; (d) execução do teste; ou (f) análise do teste
Identificar em quais fases de teste os estudos são conduzidos	**Q4. Fases de teste**) Quais fases de teste têm sido apoiadas por planejamento em IA?	Categorização das fases de teste de Lewis (2004): (a) teste de unidade; (b) teste de integração; (c) teste de sistema; ou (d) teste de aceitação
Identificar quais técnicas de teste são consideradas nos estudos	**Q5. Técnicas de teste**) Quais técnicas de teste têm sido associadas ao planejamento em IA?	Categorização das técnicas de teste de Myers et al. (2011): (a) teste funcional; (b) teste estrutural; ou (c) teste baseado em erros
Identificar quais linguagens de planejamento são usadas nas representações do teste de software	**Q6. Linguagens de planejamento**) Quais linguagens de planejamento em IA têm sido usadas para representar o teste de software?	Questão aberta
Identificar quais planejadores são usados, desenvolvidos ou adaptados nos estudos	**Q7. Planejadores**) Quais planejadores têm sido usados para apoiar o teste de software?	Questão aberta
Identificar quais representações são modeladas como ações dos planos de IA	**Q8. Planos de IA**) Quais representações têm sido associadas às ações dos planos de IA?	Questão aberta

As questões foram parcialmente baseadas na teoria de teste descrita pela ontologia *Reference Ontology on Software Testing* (ROoST) (Würsch et al., 2012; Borges Ruy et al., 2016; Souza et al., 2017; SEON, 2017). A ontologia ROoST é integrada a uma rede de ontologias da

área da engenharia de software e inclui a representação dos seguintes elementos envolvidos no teste de software: ambiente, artefatos, atividades, fases e técnicas. **Q1**, **Q6** e **Q7** derivam do elemento *ambiente*, **Q2** e **Q8** de *artefatos*, **Q3** de *atividades*, **Q4** de *fases* e **Q5** de *técnicas*.

Foram realizadas leituras dos textos completos de cada estudo para identificar os dados que respondem as questões da Tabela A.1. Os dados obtidos foram inicialmente inseridos em uma planilha e organizados em tabelas na Seção A.2. Em seguida, foi realizada a análise de dados apresentada na Seção 3.1 do Capítulo 3. Essa análise objetiva interpretar os dados obtidos na extração e identificar tendências e lacunas relacionados ao uso do planejamento em IA no teste de software.

Informações gerais dos estudos, como veículo de publicação e instituição dos autores, uma síntese da extração de dados e a documentação da codificação e categorização dos relatos descritos na Seção 3.1 estão disponíveis em uma planilha[2] do *Google Drive*[3] (*link*: `https://docs.google.com/spreadsheets/d/1M0DQOx42pmDumAGIf5TbbYeIbSJo0NX5PaNzw2BL4bo/edit?usp=sharing`).

A.2 RESULTADOS

Esta seção apresenta os resultados da extração de dados nos estudos selecionados. A Tabela A.2 sumariza as respostas das questões da Tabela A.1 conforme as informações declaradas nos estudos. Dados não reportados e identificados com análises complementares da documentação, arquitetura ou código disponibilizados são apresentados entre parênteses. Dados não especificados que não foram identificados na análise complementar são indicados por "NE".

Tabela A.2: Síntese da extração de dados dos trabalhos relacionados.

Estudo	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Anderson e Fickas (1989)	Sistema de gerenciamento (funcionalidade)	Especificação do SUT	NE	NE	(Funcional)	NE	NE	Conjunto de operadores
Mraz et al. (1995)	Biblioteca (funcionalidade)	Comandos da interface	(Design de casos de teste)	(Unidade, integração e sistema)	Baseado em domínio (Funcional)	(ADL)	UCPOP	Casos de teste
Howe et al. (1995)	Biblioteca (funcionalidade)	Comandos da interface	(Design de casos de teste)	(Unidade, integração e sistema)	Baseado em domínio (Funcional)	(ADL)	UCPOP	Casos de teste
Howe et al. (1997)	Biblioteca (funcionalidade)	Comandos da interface	(Design de casos de teste)	(Unidade, integração e sistema)	Baseado em domínio (Funcional)	(ADL)	UCPOP	Casos de teste
Scheetz et al. (1999)	Biblioteca (funcionalidade)	Diagrama de classes UML	(Design de casos de teste)	(Unidade, integração e sistema)	Funcional	(ADL)	UCPOP	Casos de teste
von Mayrhauser et al. (1999)	Biblioteca (funcionalidade)	Diagrama de classes UML	(Design de casos de teste)	(Unidade, integração e sistema)	Orientado a objetivo (Funcional)	(ADL)	UCPOP	Casos de teste
Memon et al. (1999)	Interface HC (funcionalidade)	Especificação da interface	(Design de casos de teste)	NE	(Funcional)	(ADL)	IPP	Casos de teste
Memon et al. (2000)	Interface HC (funcionalidade)	Especificação da interface	(Design de casos de teste)	NE	Oráculo (Funcional)	(ADL)	IPP	Casos de teste
von Mayrhauser et al. (2000)	Biblioteca (funcionalidade)	Especificação do SUT	(Design de casos de teste)	(Unidade, integração e sistema)	Mutação, Baseado em erro	(ADL)	UCPOP	Casos de teste
Memon et al. (2001)	Interface HC (funcionalidade)	Especificação da interface	(Design de casos de teste)	NE	(Funcional)	(ADL)	IPP	Casos de teste

Continua na próxima página

[2]Esta revisão foca em estudos conceitualmente próximos ao escopo desta tese e que apresentam motivações e contribuições para o desenvolvimento da pesquisa. A planilha disponibilizada apresenta uma versão estendida da revisão que inclui: i) uma síntese de outros estudos com abordagens de planejamento do teste de integração e; ii) uma versão ampliada da extração de dados.

[3]*Google Drive* é uma plataforma de armazenamento, compartilhamento e colaboração para edição de arquivos. Disponível em: `https://www.google.com/drive`

Tabela A.2: Síntese da extração de dados dos trabalhos relacionados (continuação da página anterior).

Estudo	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
Feather e Smith (2001)	Sistema autônomo (caract.)	Especificação do design	Planejamento	NE	(Funcional)	NE	NE	Atividades
Andrews et al. (2002)	Biblioteca (funcionalidade)	Diagrama de classes UML	(Design de casos de teste)	(Unidade, integração e sistema)	Orientado a objetivo (Funcional)	(ADL)	UCPOP	Casos de teste
Yen et al. (2002)	Sistema embarcado (caract.)	Especificação da hierarquia	(Design de casos de teste e codificação)	(Unidade e sistema)	(Funcional)	NE	NE	Regras
Leitner e Bloem (2005)	Sistema Eiffel (OO, paradigma)	Código	Execução	Unidade	Baseado em contrato (Estrutural)	ExtNADL	Bifrost	Instruções
Boddy et al. (2005)	Sistema Web (plataforma)	Informações da rede e ataques	Planejamento e execução	(Sistema)	Intrusão (Funcional)	PDDL	Metric-FF	*Exploits*
Paradkar et al. (2007)	Serviço de Web semântica (caract.)	Especificação funcional	Planejamento, (design) e execução	(Sistema)	Baseado em modelo Funcional	NE	Planejador *Graphplan*	Operações
Gupta et al. (2007)	Sistema de simulação (caract.)	Descrição do sistema	(Planejamento e design de casos de teste)	(Sistema)	Orientado a objetivo (Funcional)	NE	Planejador *Graphplan*	Dados de teste
Schnelte (2009)	Sistema temporal (caract.)	Requisitos funcionais	(Design de casos de teste)	(Sistema)	Funcional	NE	NE	Casos de teste
Li et al. (2009)	Interface HC (funcionalidade)	(Especificação da interface)	(Design de casos de teste)	(Sistema)	(Funcional)	NE	MF-IPP	Casos de teste
Galler et al. (2010)	Aplicação industrial Java (caract., OO, paradigma)	Especificação de design e código	(Design de casos de teste)	Unidade	Orientado a objetivo (Estrutural)	PDDL	NE	Dados de teste
Schnelte e Güldali (2010)	Sistema de agendamento (caract.)	Contratos visuais	(Design de casos de teste)	Unidade	Baseado em modelo (Funcional)	PDDL	LAMA	Casos de teste
Silva e Lemos (2011)	Sistema autoadap. (OO, caract., paradigma)	Configuração dos componentes	Planejamento e execução	Integração	Baseado em componente (Funcional)	PDDL	SGPlan	Ordem de integração e teste
Sarraute et al. (2012)	Sistema de rede (caract.)	Descrição do sistema	Planejamento	(Sistema)	Intrusão (Funcional)	NE	NE	Ataque
Obes et al. (2013)	Sistema de rede (caract.)	Especificação do sistema e ataque	Planejamento e execução	(Sistema)	Intrusão (Funcional)	PDDL	Metric-FF e SGPlan	Caminho de ataque
Wotawa e Bozic (2014)	Aplicação Web (plataforma)	Especificação da rede, ataque e vulnerabilidade	(Design de casos de teste e execução)	(Sistema)	Intrusão (Funcional)	PDDL	Metric-FF	Casos de teste
Razavi et al. (2014)	NE	Padrão de violações	(Execução)	(Sistema)	(Estrutural)	PDDL	FF	Execução viável
Bozic e Wotawa (2015)	Aplicação Web (plataforma)	Especificação da rede, ataque e vulnerabilidade	(Design de casos de teste) e execução	(Sistema)	Baseado em modelo, segurança (Funcional)	PDDL	Metric-FF	Casos de teste
Bozic et al. (2017)	Protocolo de rede (funcionalidade)	Funcionalidades e terminologias do protocolo TLS	(Design de casos de teste) e execução	(Sistema)	Intrusão (funcional)	PDDL	Metric-FF	Casos de teste
Bozic e Wotawa (2018b)	Aplicação Web (plataforma)	Informações do sistema	(Design de casos de teste) e execução	(Sistema)	Intrusão (Funcional)	PDDL	JavaGP	Casos de teste
Shmaryahu et al. (2018)	Sistema de rede (caract.)	Informações da rede	(Planejamento)	(Sistema)	Intrusão (Funcional)	PDDL	NE	*Exploits*
Bozic e Wotawa (2018a)	Aplicação Web (plataforma)	Especificação do sistema e ataque	(Design de casos de teste) e execução	Sistema	Intrusão (Funcional)	PDDL	Metric-FF	Casos de teste
Bozic et al. (2019)	Aplicação Web (plataforma)	Texto com interações	(Design de casos de teste) e execução	(Sistema)	Funcional	PDDL	NE	Casos de teste
Simos et al. (2019)	Protocolo de rede (funcionalidade)	Eventos do protocolo TLS	(Design de casos de teste) e execução	(Sistema)	Segurança (Funcional)	PDDL	JavaGP	Casos de teste
Bozic e Wotawa (2020)	Aplicação Web (plataforma)	Informações do sistema e ataque	(Design de casos de teste) e execução	(Sistema)	Intrusão (Funcional)	PDDL	Metric-FF	Casos de teste
Lima (2020)	Aplicação Web (plataforma)	Documentação das ferramentas e do SUT	Planejamento e execução	Sistema	Intrusão, Funcional	PDDL	Metric-FF	Configuração de ferramentas

A.2.1 Q1. Sistemas sob teste

Conforme indicado na Tabela A.3, os resultados da Q1 identificaram SUTs com dez características distintas. A característica mais recorrente observada é a aplicação no contexto de sistemas de redes, presente em 8,57% dos estudos (n = 3). Em seguida, os protocolos de rede equivalem a 5,71% dos estudos (n = 2). As demais características encontradas, como sistemas autônomos, embarcados e temporais, correspondem a 2,86% dos estudos (n = 1) cada.

Com relação aos paradigmas de programação, foram identificados 8,57% dos estudos (n = 3) com sistemas OO. A principal plataforma de uso dos sistemas é a Web, identificada em 22,86% dos estudos (n = 8). As funcionalidades com maior destaque nos sistemas são bibliotecas em 20% (n = 7) e interfaces HC em 11,43% dos estudos (n = 4).

Tabela A.3: Resultados da Q1: "*Quais SUTs têm sido usados no teste de software com planejamento em IA?*".

Sistema sob teste		**Referências**
Característica	Sistema autônomo	Feather e Smith (2001)
	Sistema embarcado	Yen et al. (2002)
	Serviço de semântica	Paradkar et al. (2007)
	Sistema de simulação	Gupta et al. (2007)
	Sistema temporal	Schnelte (2009)
	Aplicação industrial	Galler et al. (2010)
	Sistema de agendamento	Schnelte e Güldali (2010)
	Sistema autoadaptativo	Silva e Lemos (2011)
	Protocolo de rede	(Bozic et al., 2017), Simos et al. (2019)
	Sistema de rede	Sarraute et al. (2012), Obes et al. (2013), Shmaryahu et al. (2018)
Paradigma	Orientado a objetos	Leitner e Bloem (2005), Galler et al. (2010), Silva e Lemos (2011)
Plataforma	Web	Boddy et al. (2005), Wotawa e Bozic (2014), Bozic e Wotawa (2015), Bozic e Wotawa (2018b), Bozic e Wotawa (2018a), Bozic et al. (2019), Bozic e Wotawa (2020), Lima (2020)
Funcionalidade	Gerenciamento	Anderson e Fickas (1989)
	Interface HC	Memon et al. (1999), Memon et al. (2000), Memon et al. (2001), Li et al. (2009)
	Biblioteca	Mraz et al. (1995), Howe et al. (1995), Howe et al. (1995), Scheetz et al. (1999), von Mayrhauser et al. (1999), von Mayrhauser et al. (2000), Andrews et al. (2002)

Razavi et al. (2014) não forneceram informações suficientes para identificar aspectos do SUT utilizado. Alguns SUTs foram categorizados em dois aspectos, como o sistema utilizado por Silva e Lemos (2011) que se enquadra como um sistema autoadaptativo e um sistema OO.

A.2.2 Q2. Artefatos

Com os resultados da Q2 foi possível identificar dezesseis tipos de artefatos usados como base para a geração dos planos de IA. A Tabela A.4 sumariza os resultados referentes aos artefatos identificados. A coluna "Fase de desenvolvimento" indica a fase de desenvolvimento de software de geração dos artefatos encontrados, seguindo a categorização apresentada na Figura 2.3 da Subseção 2.1.4.

Os resultados mostraram que 25,71% dos estudos (n = 9) utilizam artefatos com descrições e especificações gerais do SUT. Por exemplo, a proposta de Lima (2020) utiliza documentos de especificação do SUT para identificar informações do SUT, como a necessidade

Tabela A.4: Resultados da Q2: "*Quais artefatos têm sido usados para gerar planos de IA?*".

Fase de desenvolvimento	Artefato	Referências
Modelagem de requisitos	Requisitos e especificação funcional	Paradkar et al. (2007), Schnelte (2009)
	Descrição e especificação do SUT	Anderson e Fickas (1989), von Mayrhauser et al. (2000), Gupta et al. (2007), Sarraute et al. (2012), Obes et al. (2013), Bozic e Wotawa (2018b), Shmaryahu et al. (2018), Bozic e Wotawa (2020), Lima (2020)
Design de arquitetura	Especificação da hierarquia	Yen et al. (2002)
	Padrão de violações	Razavi et al. (2014)
	Texto com interações	Bozic et al. (2019)
	Documentação das ferramentas	Lima (2020)
	Contratos visuais	Schnelte e Güldali (2010)
	Especificação do design	Feather e Smith (2001), Galler et al. (2010)
	Informações do protocolo de rede	Bozic et al. (2017), Simos et al. (2019)
	Comandos da interface	Mraz et al. (1995), Howe et al. (1995), Howe et al. (1997)
	Informações da rede	Boddy et al. (2005), Wotawa e Bozic (2014), Bozic e Wotawa (2015), Bozic e Wotawa (2018b)
	Especificação da interface	Memon et al. (1999), Memon et al. (2000), Memon et al. (2001), Li et al. (2009)
	Informações do ataque	Boddy et al. (2005), Obes et al. (2013), Wotawa e Bozic (2014), Bozic e Wotawa (2015), Shmaryahu et al. (2018), Bozic e Wotawa (2020)
Design de componentes	Configuração de componentes	Silva e Lemos (2011)
	Diagrama de classes UML	Scheetz et al. (1999), von Mayrhauser et al. (1999), Andrews et al. (2002)
Geração de código	Código	Leitner e Bloem (2005), Galler et al. (2010)

do uso de *login* e senha para autenticação. Em seguida, 17,14% dos estudos (n = 6) usam artefatos com informações para simular ataques contra o SUT no contexto de teste de segurança.

Os artefatos usados abrangem elementos sobre interfaces HC e redes como: especificações das funcionalidades da interface HC, em 11,43% dos estudos (n = 4); descrições dos comandos da interface HC, incluindo informações das abas e menus, em 8,57% dos estudos (n = 3); e documentos com informações da rede, como *hosts* e servidores, também em 11,43% (n = 4). Outros artefatos de destaque são os diagramas de classes UML, observados em 8,57% dos estudos (n = 3). Os demais artefatos correspondem a 2,86% (n = 1) ou 5,71% (n = 2) cada.

Os artefatos criados na fase de design de arquitetura representam 74,29% dos estudos (n = 26). Os artefatos gerados na fase de modelagem de requisitos correspondem a 31,43% dos estudos (n = 11). Os artefatos criados nas fases de design de componentes e geração de código correspondem a 11,43% (n = 4) e 5,71% (n = 2), respectivamente.

A.2.3 Q3. Atividades de teste

De acordo com os resultados da Q3 apresentados na Tabela A.5, 71,43% dos estudos (n = 25) aplicam o planejamento em IA na atividade de design de casos de teste. Esses estudos

incluem, por exemplo, propostas de geração de dados de teste (Galler et al., 2010) e geração de casos de teste (Memon et al., 2001).

Tabela A.5: Resultados da Q3: "*Quais atividades de teste têm sido apoiadas por planejamento em IA?*".

Atividade de teste	Referências
Planejamento do teste	Feather e Smith (2001), Boddy et al. (2005), Paradkar et al. (2007), Gupta et al. (2007), Silva e Lemos (2011), Sarraute et al. (2012), Obes et al. (2013), Shmaryahu et al. (2018), Lima (2020)
Design de casos de teste	Mraz et al. (1995), Howe et al. (1995), Howe et al. (1995), Scheetz et al. (1999), von Mayrhauser et al. (1999), Memon et al. (1999), Memon et al. (2000), von Mayrhauser et al. (2000), Memon et al. (2001), Andrews et al. (2002), Yen et al. (2002), Paradkar et al. (2007), Gupta et al. (2007), Schnelte (2009), Li et al. (2009), Galler et al. (2010), Schnelte e Güldali (2010), Wotawa e Bozic (2014), Bozic e Wotawa (2015), Bozic et al. (2017), Bozic e Wotawa (2018b), Bozic e Wotawa (2018a), Bozic et al. (2019), Simos et al. (2019), Bozic e Wotawa (2020)
Codificação do teste	Yen et al. (2002)
Execução do teste	Leitner e Bloem (2005), Boddy et al. (2005), Paradkar et al. (2007), Silva e Lemos (2011), Obes et al. (2013), Wotawa e Bozic (2014), Razavi et al. (2014), Bozic e Wotawa (2015), Bozic et al. (2017), Bozic e Wotawa (2018b), Bozic e Wotawa (2018a), Bozic et al. (2019), Simos et al. (2019), Bozic e Wotawa (2020), Lima (2020)
Análise do teste	Não identificado

A segunda atividade de teste mais presente nos estudos é a de execução do teste, com 42,86% (n = 15). Em seguida, foi possível identificar 25,71% dos estudos (n = 9) no contexto da atividade de planejamento do teste. Por exemplo, Silva e Lemos (2011) geram planos de IA que indicam uma ordem para integrar e testar unidades, sendo esse um elemento definido durante o planejamento do teste de integração. Propostas no contexto de codificação do teste correspondem a 2,86% (n = 1). Não foram identificadas propostas do contexto da atividade de análise do teste.

A.2.4 Q4. Fases de teste

Os resultados da Q4 apresentados na Tabela A.6 mostraram que 74,29% dos estudos (n = 26) ocorrem na fase de teste de sistema. Um dos objetivos dessa fase é verificar se requisitos não-funcionais, como segurança, estão adequados na versão final do software (Lewis, 2004). Assim, todos os estudos realizados no contexto de teste de segurança foram categorizados como teste de sistema. Por exemplo, Bozic e Wotawa (2020) geram planos de IA cujas ações são instruções que indicam a existência de vulnerabilidades no SUT.

Tabela A.6: Resultados da Q4: "*Quais fases de teste têm sido apoiadas por planejamento em IA?*".

Fase de teste	Referências
Teste de unidade	Mraz et al. (1995), Howe et al. (1995), Howe et al. (1995), Scheetz et al. (1999), von Mayrhauser et al. (1999), von Mayrhauser et al. (2000), Andrews et al. (2002), Yen et al. (2002), Leitner e Bloem (2005), Galler et al. (2010), Schnelte e Güldali (2010)
Teste de integração	Mraz et al. (1995), Howe et al. (1995), Howe et al. (1995), Scheetz et al. (1999), von Mayrhauser et al. (1999), von Mayrhauser et al. (2000), Andrews et al. (2002), Silva e Lemos (2011)
Teste de sistema	Mraz et al. (1995), Howe et al. (1995), Howe et al. (1995), Scheetz et al. (1999), von Mayrhauser et al. (1999), von Mayrhauser et al. (2000), Andrews et al. (2002), Yen et al. (2002), Boddy et al. (2005), Paradkar et al. (2007), Gupta et al. (2007), Schnelte (2009), Li et al. (2009), Sarraute et al. (2012), Obes et al. (2013), Wotawa e Bozic (2014), Razavi et al. (2014), Bozic e Wotawa (2015), Bozic et al. (2017), Bozic e Wotawa (2018b), Shmaryahu et al. (2018), Bozic e Wotawa (2018a), Bozic et al. (2019), Simos et al. (2019), Bozic e Wotawa (2020), Lima (2020)
Teste de aceitação	Não identificado

A segunda fase de teste mais recorrente é o teste de unidade, identificada em 31,43% dos estudos (n = 11). Por exemplo, os planos de IA gerados por Mraz et al. (1995) representam casos de teste para testar comandos entendidos como unidades do SUT. A fase de teste de integração foi identificada em 22,86% dos estudos (n = 8). Por exemplo, Scheetz et al. (1999) geram planos de IA que representam casos de teste para o teste de conjuntos de classes integradas (subsistemas).

Não foram identificadas propostas para a fase de teste de aceitação. Estudos como Anderson e Fickas (1989), Memon et al. (1999), Memon et al. (2000), Memon et al. (2001) e

Feather e Smith (2001) não informaram dados suficientes para estabelecer uma associação a uma fase de teste específica.

A.2.5 Q5. Técnicas de teste

Os resultados sumarizados na Tabela A.7 indicam que 88,57% dos estudos (n = 31) apresentam soluções apoiadas por técnicas de teste funcional. O tipo de teste apoiado por essa técnica com maior destaque é o teste de intrusão, observado em 28,57% dos estudos (n = 10). São exemplos de propostas nesse contexto os estudos de Bozic e Wotawa (2015) e Shmaryahu et al. (2018).

Tabela A.7: Resultados da Q5: "*Quais técnicas de teste têm sido associadas ao planejamento em IA?*".

Técnica de teste		**Referências**
Teste funcional	Oráculo	Memon et al. (2000)
	Baseado em componente	Silva e Lemos (2011)
	Segurança	Bozic et al. (2017), Simos et al. (2019)
	Baseado em modelo	Paradkar et al. (2007), Schnelte e Güldali (2010)
	Orientado a objetivo	von Mayrhauser et al. (1999), Andrews et al. (2002), Gupta et al. (2007)
	Baseado em domínio	Mraz et al. (1995), Howe et al. (1995), Howe et al. (1995)
	Intrusão	Boddy et al. (2005), Sarraute et al. (2012), Obes et al. (2013), Wotawa e Bozic (2014), Bozic e Wotawa (2015), Bozic e Wotawa (2018b), Shmaryahu et al. (2018), Bozic e Wotawa (2018a), Bozic e Wotawa (2020), Lima (2020)
	Não especificado	Anderson e Fickas (1989), Scheetz et al. (1999), Memon et al. (1999), Memon et al. (2001), Feather e Smith (2001), Yen et al. (2002), Schnelte (2009), Li et al. (2009), Bozic et al. (2019)
Teste estrutural	Baseado em contrato	Leitner e Bloem (2005)
	Orientado a objetivo	Galler et al. (2010)
	Não especificado	Razavi et al. (2014)
Teste baseado em erros	Mutação	von Mayrhauser et al. (2000)

Outros tipos de teste baseados em técnicas funcionais são testes de segurança, orientados a objetivo e baseados em domínio e modelo. Finalmente, testes baseados em componentes e testes de oráculo correspondem a 2,86% (n = 1) cada. Na sequência, o uso de teste estrutural foi identificado em 8,57% dos estudos (n = 3), abrangendo propostas de testes baseados em contrato e orientados a objetivo. O estudo de von Mayrhauser et al. (2000) usa a técnica de teste baseado em erros associando planejamento em IA e conceitos de teste de mutação.

A.2.6 Q6. Linguagens de planejamento

Conforme indicado na Tabela A.8, os resultados da Q6 indicaram o uso de três linguagens de planejamento em IA. A principal linguagem utilizada é a PDDL, identificada em 45,71% dos estudos (n = 16). A segunda linguagem em destaque é a ADL, com 28,57% (n = 10). A linguagem ExtNADL corresponde a 2,86% (n = 1) dos estudos.

Foi possível identificar a utilização de três versões da PDDL. A versão PDDL1.2 foi usada em quatro estudos (Galler et al. (2010), Schnelte e Güldali (2010), Razavi et al. (2014), Simos et al. (2019)). Onze estudos utilizaram a versão PDDL2.1: Boddy et al. (2005), Lima (2020), Silva e Lemos (2011), Obes et al. (2013), Wotawa e Bozic (2014), Bozic e Wotawa (2015), Bozic et al. (2017), Bozic e Wotawa (2018b), Bozic e Wotawa (2018a), Bozic et al. (2019), Bozic e Wotawa (2020).

Shmaryahu et al. (2018) utilizaram uma versão conhecida como PDDL contingente (Albore et al., 2009; Bonet e Geffner, 2011). Essa variação da linguagem busca representar

problemas a partir de informações incompletas e ações sensoriais. Não foi identificado o uso das versões PDDL2.2, PDDL3.0 e PDDL3.1 nos estudos analisados.

Tabela A.8: Resultados da Q6: "*Quais linguagens de planejamento em IA têm sido usadas para representar teste de software?*".

Linguagem	Referências
ExtNADL	Leitner e Bloem (2005)
ADL	Mraz et al. (1995), Howe et al. (1995), Howe et al. (1995), Scheetz et al. (1999), von Mayrhauser et al. (1999), Memon et al. (1999), Memon et al. (2000), von Mayrhauser et al. (2000), Memon et al. (2001), Andrews et al. (2002)
PDDL	Boddy et al. (2005), Galler et al. (2010), Schnelte e Güldali (2010), Silva e Lemos (2011), Obes et al. (2013), Wotawa e Bozic (2014), Razavi et al. (2014), Bozic e Wotawa (2015), Bozic et al. (2017), Bozic e Wotawa (2018b), Shmaryahu et al. (2018), Bozic e Wotawa (2018a), Bozic et al. (2019), Simos et al. (2019), Bozic e Wotawa (2020), Lima (2020)

A.2.7 Q7. Planejadores

Conforme apresentado na Tabela A.9, os resultados da Q7 identificaram a utilização de nove planejadores distintos. O planejador com maior destaque é o Metric-FF, usado em 22,86% dos estudos (n = 8). Outros planejadores para problemas em PDDL, como o SGPlan e o JavaGP, correspondem a 5,71% dos estudos (n = 2).

O planejador UCPOP foi utilizado em 20% dos estudos (n = 7). O UCPOP é um planejador desenvolvido por Penberthy e Weld (1992) para a resolução de problemas em ADL. O uso do planejador IPP, também aplicado em problemas ADL, corresponde a 8,57% dos estudos (n = 3). Outros planejadores correspondem a 2,86% (n = 1) cada: FF, LAMA, MF-IPP e Bifrost.

Tabela A.9: Resultados da Q7: "*Quais planejadores têm sido usados para apoiar o teste de software?*".

Planejador	Referências
Bifrost	Leitner e Bloem (2005)
MF-IPP	Li et al. (2009)
LAMA	Schnelte e Güldali (2010)
FF	Razavi et al. (2014)
JavaGP	Bozic e Wotawa (2018b), Simos et al. (2019)
SGPlan	Silva e Lemos (2011), Obes et al. (2013)
IPP	Memon et al. (1999), Memon et al. (2000), Memon et al. (2001)
UCPOP	Mraz et al. (1995), Howe et al. (1995), Howe et al. (1995), Scheetz et al. (1999), von Mayrhauser et al. (1999), von Mayrhauser et al. (2000), Andrews et al. (2002)
Metric-FF	Boddy et al. (2005), Obes et al. (2013), Wotawa e Bozic (2014), Bozic e Wotawa (2015), Bozic et al. (2017), Bozic e Wotawa (2018a), Bozic e Wotawa (2020), Lima (2020)

Vale ressaltar que alguns estudos como Yen et al. (2002), Feather e Smith (2001), Schnelte (2009), Sarraute et al. (2012) não indicam o uso de linguagens de planejamento em IA ou planejadores específicos. Outros estudos mencionam o algoritmo de planejamento em IA considerado nas soluções. Por exemplo, Paradkar et al. (2007) e Gupta et al. (2007) utilizam o algoritmo *Graphplan* de Blum e Furst (1997).

A.2.8 Q8. Planos de IA

Os resultados da Q8 apresentados na Tabela A.10 revelaram doze representações distintas nos planos de IA. A maioria dos estudos, correspondendo a 60% (n = 21), geram planos de IA que representam casos de teste. Diante do destaque no contexto de segurança, 5,71% (n = 2) dos

estudos geram planos de IA que representam caminhos de ataque contendo sequências de ações que indicam um ataque bem sucedido, como em Obes et al. (2013).

Também em 5,71% dos estudos (n = 2) foi possível identificar planos de IA que representam dados de teste e *exploits*. As demais representações de planos de IA equivalem a 2,86% (n = 1) dos estudos, sendo eles: configurações de ferramentas, ações do SUT, operações para execução do SUT, operações do SUT, ordem de integração, instruções, regras, atividades e conjunto de operadores.

Tabela A.10: Resultados da Q8: "*Quais representações têm sido associadas às ações dos planos de IA?*".

Plano de IA	**Referências**
Conjunto de operadores	Anderson e Fickas (1989)
Atividades	Feather e Smith (2001)
Regras	Yen et al. (2002)
Instruções	Leitner e Bloem (2005)
Operações do SUT	Paradkar et al. (2007)
Ordem de integração	Silva e Lemos (2011)
Operações para execução	Razavi et al. (2014)
Configurações de ferramentas	Lima (2020)
Exploits	Boddy et al. (2005), Shmaryahu et al. (2018)
Dados de teste	Gupta et al. (2007), Galler et al. (2010)
Caminhos de ataque	Sarraute et al. (2012), Obes et al. (2013)
Casos de teste	Mraz et al. (1995), Howe et al. (1995), Howe et al. (1995), Scheetz et al. (1999), von Mayrhauser et al. (1999), Memon et al. (1999), Memon et al. (2000), von Mayrhauser et al. (2000), Memon et al. (2001), Andrews et al. (2002), Schnelte (2009), Li et al. (2009), Schnelte e Güldali (2010), Wotawa e Bozic (2014), Bozic e Wotawa (2015), Bozic et al. (2017), Bozic e Wotawa (2018b), Bozic e Wotawa (2018a), Bozic et al. (2019), Simos et al. (2019), Bozic e Wotawa (2020)

A.3 AMEAÇAS À VALIDADE

Algumas ameaças à validade foram identificas durante a seleção de estudos e a análise de resultados desta revisão bibliográfica. Durante esse processos, medidas foram tomadas para minimizar a influência dessas limitações e ameaças.

Uma ameaça ao processo de seleção dos estudos envolve a realização da revisão bibliográfica de forma não sistemática. Kitchenham e Charters (2007) sugerem que, antes de realizar um estudo sistemático, é importante confirmar a necessidade de tal estudo. Como indicado na Seção A.1, foram identificadas duas revisões da literatura previamente publicadas sobre o tema estudado.

Dessa forma, foram utilizados processos de *snowballing* usando as revisões da literatura encontradas como fonte, em vez de estabelecer um protocolo formal de estudo sistemático. Wohlin (2014) destaca que o *snowballing* ajuda a conduzir pesquisas adicionais com base em outros estudos relevantes publicados.

Não foi realizada uma avaliação de qualidade dos estudos selecionados. A seleção de estudos ocorreu principalmente com processos de *snowballing* nas fontes (1) e (2), indicadas na Seção A.1. Essas fontes consistem em revisões de literatura revisadas por pares, indicando que foram submetidas a processos rigorosos e identificaram estudos de qualidade.

Outra ameaça está relacionada ao período de publicação dos estudos selecionados. Os estudos identificados pelas fontes (1) e (2), descritas na Seção A.1, foram publicados até 2020. A revisão complementar (fonte (3) indicada na Seção A.1) investigou estudos publicados até o último trimestre de 2024. Não foram incluídos estudos publicados após esse período. A análise de estudos ocorreu durante o segundo semestre de 2023 e o primeiro trimestre de 2024.

A análise de dados podem introduzir viés aos resultados. Para minimizar esta ameaça, a revisão bibliográfica e a análise de estudos foram conduzidas por dois pesquisadores. Inicialmente, um pesquisador conduziu a extração de dados para responder as questões descritas na Tabela A.1. Posteriormente, um segundo pesquisador revisou a extração de dados. Os dois pesquisadores discutiram as divergências da extração de dados e chegaram a um consenso. Ambos pesquisadores realizaram a análise de dados.

A.4 CONSIDERAÇÕES

A partir da análise de 35 estudos, esta revisão bibliográfica da literatura que delineou um panorama do uso do planejamento em IA no teste de software, destacou tendências e lacunas nessa área de pesquisa e identificou motivações para o desenvolvimento desta tese.

Os resultados de cada questão analisada na revisão mostraram que:

- **Q1. Sistemas sob teste**: os estudos consideraram SUTs com dez características distintas, SUTs desenvolvidos com paradigmas de programação procedimental e OO, SUTs executados em plataformas Web e SUTs focados principalmente em funcionalidades de gerenciamento, interfaces HC e bibliotecas;
- **Q2. Artefatos**: os estudos incorporaram artefatos de todo o processo de desenvolvimento de software, destacando artefatos criados durante as fases iniciais de modelagem de requisitos e design de arquitetura;
- **Q3. Atividades de teste**: os estudos aplicaram planejamento de IA para apoiar as atividades de planejamento do teste, design de casos de teste, codificação do teste e execução do teste, com foco principal nas atividades de design de casos de teste e execução do teste;
- **Q4. Fases de teste**: os estudos associaram o planejamento de IA às fases de teste de unidade, teste de integração e teste de sistema, sendo o teste de sistema a principal fase apoiada;
- **Q5. Técnicas de teste**: os estudos combinaram planejamento de IA com técnicas de teste funcional, estrutural e baseado em erros, sendo considerados principalmente testes apoiados por técnicas funcionais;
- **Q6. Linguagens de planejamento**: os estudos utilizaram três linguagens de planejamento de IA (ExtNADL, ADL e PDDL), com o uso principal de linguagens de planejamento não-clássico (ADL e PDDL) para representar o teste de software;
- **Q7. Planejadores**: os estudos usaram nove planejadores (Bifrost, MF-IPP, LAMA, FF, JavaGP, SGPlan, IPP, UCPOP e Metric-FF) e citaram a linguagem de planejamento, a disponibilidade de código-fonte e fatores de desempenho como motivações para a seleção dessas ferramentas; e
- **Q8. Planos de IA**: os estudos geraram planos de IA ocorre a partir de representações que mapeiam os elementos de teste como objetos, as associações entre esses elementos como pré-condições e efeitos e os processos de teste em cada contexto como ações.

APÊNDICE B – SÍNTESE DOS ESTUDOS DA REVISÃO BIBLIOGRÁFICA

A Tabela B.1 apresenta os 35 estudos selecionados na revisão bibliográfica analisada no Capítulo 3 e descrita no Apêndice A. As propostas dos estudos são sumarizadas adiante.

Tabela B.1: Estudos sobre planejamento em IA no teste de software.

Referência	Título
Anderson e Fickas (1989)	*A proposed perspective shift: Viewing specification design as a planning problem*
Mraz et al. (1995)	*System testing with an AI planner*
Howe et al. (1995)	*Test sequences as plans: An experiment in using an AI planner to generate system tests*
Howe et al. (1997)	*Test case generation as an AI planning problem*
Scheetz et al. (1999)	*Generating test cases from an OO model with an AI planning system*
von Mayrhauser et al. (1999)	*Generating goal-oriented test cases*
Memon et al. (1999)	*Using a goal-driven approach to generate test cases for GUIs*
Memon et al. (2000)	*A planning-based approach to GUI testing*
von Mayrhauser et al. (2000)	*Planner based error recovery testing*
Memon et al. (2001)	*Hierarchical GUI test case generation using automated planning*
Feather e Smith (2001)	*Automatic generation of test oracle from pilot studies to application*
Andrews et al. (2002)	*AI planner assisted test generation*
Yen et al. (2002)	*Application of AI planning techniques to automated code synthesis and testing*
Leitner e Bloem (2005)	*Automatic testing through planning*
Boddy et al. (2005)	*Course of action generation for cyber security using classical planning*
Paradkar et al. (2007)	*Automated functional conformance test generation for semantic web services*
Gupta et al. (2007)	*Rapid goal-oriented automated software testing using MEA-graph planning*
Schnelte (2009)	*Generating test cases for timed systems from controlled natural language specifications*
Li et al. (2009)	*A method for combinatorial explosion avoidance of AI planner and the application on test case generation*
Galler et al. (2010)	*AIana: An AI planning system for test data generation*
Schnelte e Güldali (2010)	*Test case generation for visual contracts using AI planning*
Silva e Lemos (2011)	*Dynamic plans for integration testing of self-adaptive software systems*
Sarraute et al. (2012)	*POMDPs make better hackers: Accounting for uncertainty in penetration testing*
Obes et al. (2013)	*Attack planning in the real world*
Wotawa e Bozic (2014)	*Plan it! Automated security testing based on planning*
Razavi et al. (2014)	*Generating effective tests for concurrent programs via AI automated planning techniques*
Bozic e Wotawa (2015)	*PURITY: a Planning-based secURITY testing tool*
Bozic et al. (2017)	*Planning-based security testing of the SSL/TLS protocol*
Bozic e Wotawa (2018b)	*Planning-based security testing of web applications*
Shmaryahu et al. (2018)	*Simulated penetration testing as contingent planning*
Bozic e Wotawa (2018a)	*Planning-based security testing for chatbots*
Bozic et al. (2019)	*Chatbot testing using AI planning*
Simos et al. (2019)	*Testing TLS using planning-based combinatorial methods and execution framework*
Bozic e Wotawa (2020)	*Planning-based security testing of web applications with attack grammars*
Lima (2020)	Teste de intrusão para aplicações web: um método com planejamento em inteligência artificial

Bozic e Wotawa (2019) indicam o uso do planejamento em IA em teste de software desde 1989 com o desenvolvimento do programa *Automated Specifier And Planner* (ASAP) por Anderson e Fickas (1989). O ASAP usa planejamento em IA para descobrir falhas em um SUT. O ASAP é executado a partir da especificação do SUT contendo a descrição de operadores, condições iniciais, objetivos e condições proibidas. Os planos de IA gerados indicam que o SUT pode conter uma falha que ocasiona uma violação estabelecida pelas condições especificadas.

Mraz et al. (1995) e Howe et al. (1995) propõem modelagens da especificação de um SUT com planejamento em IA. Essa especificação inclui os comandos sintáticos e semânticos do SUT, suas condições iniciais e um estado objetivo que estabelece o que deve ser testado no SUT. A geração de planos de IA ocorre com o planejador UCPOP. Esses planos de IA indicam sequências de comandos executados que representam casos de teste.

Howe et al. (1997) elaboram uma abordagem que utiliza uma especificação contendo os comandos das funcionalidades de um SUT. A partir de uma representação baseada nessa

especificação, o planejador UCPOP gera planos de IA que representam casos de teste. Esses autores realizam um estudo comparativo entre essa abordagem e a ferramenta *Sleuth*. Esse estudo indica a inclusão do tratamento de casos de teste inválidos e testes de subdomínios do SUT como possibilidades de melhorias para a abordagem.

Scheetz et al. (1999), von Mayrhauser et al. (1999) e Andrews et al. (2002) propõem abordagens de geração de casos de teste que usam descrições do SUT obtidas em diagramas de classes e de estado UML. Os diagramas são divididos em subdiagramas que contêm objetivos individuais. Então, os objetivos definidos e as condições iniciais do SUT são representados como um problema de planejamento em IA. Essa representação é usada como entrada do planejador UCPOP para a geração dos planos de IA que correspondem a casos de teste.

Memon et al. (1999) propõem uma abordagem para a geração de casos de teste de interfaces gráficas humano/computador (HC). Essa abordagem inclui uma representação com planejamento em IA da hierarquia de funcionalidades da interface do SUT. Essa abordagem integra o planejador IPP que gera planos de IA que indicam uma sequência de interações com a interface que atingem o objetivo estabelecido em um determinado cenário de teste. Memon et al. (2000) e Memon et al. (2001) automatizam essa abordagem com a ferramenta PATHS.

O estudo de von Mayrhauser et al. (2000) propõe uma abordagem para a geração de casos de teste para recuperação de erros de SUTs críticos, como dispositivos médicos. Essa abordagem associa técnicas de planejamento em IA com teste baseado em erros, realizando uma representação de operadores de mutação em termos de pré-condições e efeitos. Os planos de IA, gerados pelo planejador UCPOP, indicam os erros esperados para cada operador a partir de uma especificação do SUT.

Feather e Smith (2001) descrevem estudos que resultam na ferramenta *Planchecker* usada para a geração automática de testes de oráculo. *Planchecker* utiliza um planejador para a geração de planos de IA a partir de condições iniciais, objetivos e restrições do SUT. Esses planos de IA representam conjuntos de atividades que são enviadas e analisadas automaticamente por uma base de dados integrada à ferramenta. Essa base de dados compara o plano de IA com informações previamente armazenadas sobre o design do SUT para identificar anomalias.

Yen et al. (2002) apresentam uma abordagem com planejamento em IA para automatizar a síntese de código e o teste durante o desenvolvimento de sistemas. Essa abordagem considera a hierarquia de componentes de um SUT para a definição e representação de um espaço de estados. A partir dessa representação, um planejador gera planos de IA que indicam sequências de especificações que podem ser usadas na síntese de código ou na geração de casos de teste.

Leitner e Bloem (2005) apresentam uma estratégia que utiliza o planejamento em IA para a identificação de *bugs* em sistemas Eiffel. A partir da representação do código em termos de pré-condições e efeitos, o planejador Bifrost gera planos de IA que correspondem a sequências de instruções que satisfazem uma determinada pré-condição. A chamada por esse código é considerada válida caso o plano de IA seja gerado. Caso contrário, um *bug* é identificado e interpretado, possibilitando que uma nova solução seja criada com aprendizado.

Paradkar et al. (2007) apresentam uma abordagem automática para a geração de casos de teste em serviços Web. Os SUTs são representados em termos de entradas, saídas, pré-condições e efeitos. Assim, essa representação descreve um objetivo para o teste com base em um modelo de falhas. Esses autores desenvolvem um componente de planejador baseado no algoritmo de planejamento *Graphplan*, que usa como entrada a representação dos SUTs. Os planos de IA gerados representam sequências de invocações do serviço que são usadas como casos de teste.

Schnelte (2009) descreve uma abordagem que utiliza o planejamento em IA para a geração automática de casos de teste a partir de especificações de requisitos em linguagem natural. Os casos de teste são gerados a partir de especificações de requisitos mapeados para um

modelo formal de linguagem natural controlada. Esses autores exemplificam a abordagem em um sistema automotivo.

Li et al. (2009) propõem a ferramenta MF-IPP para a geração automática de casos de teste. Essa ferramenta foi desenvolvida a partir de modificações no planejador IPP para permitir o uso simultâneo de múltiplos arquivos de entrada. Assim, a entrada é decomposta em arquivos menores antes da execução da ferramenta proposta. Essa estratégia tem como objetivo evitar a explosão combinatorial geralmente ocasionada na execução de planejadores.

Schnelte e Güldali (2010) propõem uma abordagem de geração automática de casos de teste para teste de unidade. Essa abordagem associa planejamento em IA com contratos visuais do SUT. Esses contratos especificam o comportamento do SUT antes e após sua execução. Esses contratos são traduzidos para a linguagem PDDL e usados com entrada do planejador LAMA. Esses autores realizam experimentos de escalabilidade que indicam resultados favoráveis para a abordagem em comparação a outras estratégias de teste baseadas em modelos.

Silva e Lemos (2011) apresentam uma abordagem para indicar uma ordenação de integração de sistemas autoadaptativos usando planejamento em IA. A modelagem PDDL definida é instanciada com informações de um diagrama da configuração arquitetural dos componentes SUT. Os planos de IA gerados indicam a ordem de integração e teste desses componentes. Na sequência, ocorre a execução de casos de teste associados aos componentes de acordo com a ordem indicada no plano de IA.

O planejamento em IA também é usado em propostas para a geração de dados de teste. Gupta et al. (2007) propõem o algoritmo MEA-*Graphplan* para a geração automática de dados de teste. O MEA-*Graphplan* usa o algoritmo clássico *Graphplan* como base para propor adaptações na forma de expansão do grafo de planejamento. Além desse algoritmo, os autores elaboram um modelo formal para a representação dos SUTs.

Galler et al. (2010) apresentam a abordagem AIana para geração de dados de teste para SUTs orientados a objeto. Essa abordagem utiliza um planejador para criar planos de IA a partir de uma representação em PDDL. O plano de IA contém uma sequência de instanciação para os objetos envolvidos que satisfaz as pré-condições determinadas. Em seguida, ocorre a tradução desses planos de IA para a linguagem Java e as instâncias indicadas são usadas nos testes.

Boddy et al. (2005) usam planejamento em IA para detectar vulnerabilidades em redes com risco de ataques. Para isso, esses autores propõem a ferramenta BAMS integrada ao planejador Metric-FF. Essa ferramenta utiliza como entrada uma modelagem em PDDL com o modelo de rede, métodos de ataque e objetivos de um invasor. Dessa forma, o plano de IA gerado representa uma hipótese de ataque contra a rede a partir das informações contidas na modelagem.

O planejamento em IA também é usado em testes de intrusão. Esses testes simulam ataques contra um SUT para identificar falhas exploráveis conhecidas como vulnerabilidades. Sarraute et al. (2012) propõem uma estratégia que utiliza o planejamento em IA para gerar ataques contra um SUT. A representação proposta inclui informações sobre a configuração da rede e sobre as dependências entre as possibilidades de ataque contra o SUT.

Obes et al. (2013) propõem um modelo de ataque para realizar simulações de testes de intrusão. Esse estudo integra um modelo em PDDL, os planejadores Metric-FF e SGPlan e um *framework* de teste de intrusão. A representação PDDL contém informações do SUT e parâmetros de execução do *framework*. Os planos de IA indicam sequências de passos para um ataque contra o SUT. Assim, esses planos de IA são usados para instrumentar o *framework* para a execução de testes de intrusão.

Razavi et al. (2014) desenvolvem um *framework* para prever violações em execuções de SUTs concorrentes. Os critérios investigados pelo *framework* são condições de corrida, violações de atomicidade e leituras nulas. Os padrões dessas violações são modelados como um problema

de planejamento em IA e comparados com uma execução do SUT. Esse *framework* integra o planejador FF para a geração de planos de IA que representam caminhos executáveis do SUT. Logo, a análise do plano de IA permite identificar a existência de *bugs* no SUT.

Wotawa e Bozic (2014) propõem uma abordagem que inclui uma representação em PDDL de padrões de ataques das vulnerabilidades Injeção de SQL e *Cross-site Scripting* (XSS). Essa representação mapeia possíveis ações de um invasor como ações em PDDL. Informações do SUT, como endereço e parâmetros, são indicadas como a condição inicial. Esses autores desenvolvem um algoritmo que integra o planejador Metric-FF que gera planos de IA indicando interações necessárias com o SUT para a identificação da vulnerabilidade.

Como continuidade do estudo de Wotawa e Bozic (2014), Bozic e Wotawa (2018b) propõem um *framework* para a execução do teste e Bozic e Wotawa (2020) elaboram uma abordagem para mapear planos de IA em uma gramática que resulta em vetores de ataque. Em Bozic e Wotawa (2015), esses autores descrevem a ferramenta *Planning-based secURITY testing* (PURITY) (Bozic e Wotawa, 2015). Essa ferramenta mapeia as ações do planos de IA gerados na representação de Wotawa e Bozic (2014) como casos de teste e os executa em SUTs Web.

Bozic et al. (2017) apresentam um abordagem que aplica o planejamento em IA em testes de protocolos de segurança. Essa abordagem inclui uma representação PDDL usando funcionalidades e terminologias do protocolo *Transport Layer Security* (TLS). Os planos de IA gerados indicam a detecção ou a exploração de possíveis falhas de segurança na implementação desse protocolo. Assim, as ações desses planos de IA são entendidas como casos de teste abstratos, posteriormente concretizados por um *framework* e executados no SUT.

Shmaryahu et al. (2018) elaboram um modelo PDDL de teste de intrusão para a detecção de falhas em redes. Esse modelo representa informações da rede com planejamento contingente. Essa técnica de planejamento em IA possibilita a inclusão de características adicionais na modelagem, como probabilidades. Dessa forma, a representação pode ser estendida para um modelo de processos de decisão de Markov parcialmente observáveis.

Bozic e Wotawa (2018a) propõem uma abordagem para teste de intrusão de sistemas Web de assistência virtual, conhecidos como *chatbots*. Essa abordagem realiza testes das vulnerabilidades Injeção de SQL e XSS. Dessa forma, esses autores adaptam a abordagem proposta por Wotawa e Bozic (2014) para um novo domínio de aplicação.

Bozic et al. (2019) propõem uma abordagem para automatizar o teste de requisitos funcionais de *chatbots*. Nessa abordagem, possíveis interações dos usuários com o *chatbot* são mapeadas como ações em PDDL. O plano de IA gerado representa uma sequência de consultas do usuário que são traduzidas como casos de teste. A execução dos casos de teste visa verificar a exatidão dos dados fornecido pelo *chatbot* ao usuário.

Simos et al. (2019) desenvolvem uma estratégia para geração e execução automática de casos de teste para o protocolo de segurança TLS. Essa estratégia usa o planejamento em IA para gerar planos de IA que representam sequências abstratas de eventos desse protocolo. O planejador utilizado é o JavaGP, o qual oferece suporte para modelagens em PDDL. As informações dos planos de IA são posteriormente transformadas em sequências com valores concretos para a execução do protocolo.

Lima (2020) propõe um método para testes de intrusão em aplicações Web utilizando planejamento em IA. Esse método usa o planejador Metric-FF para gerar planos de IA que indicam sequências de configurações de ferramentas necessárias para cada cenário de teste. Esses planos de IA são utilizados para instanciar módulos que executam testes para a identificação das vulnerabilidades Injeção de SQL e XSS.

APÊNDICE C – ESTRUTURAS DE TESTE DE INTEGRAÇÃO COM PLANEJAMENTO EM IA

A análise apresentada a seguir tem como objetivo definir estruturas de teste de integração com planejamento em IA com base em estudos que associam esses conceitos. Esses estudos foram encontrados na revisão bibliográfica do Apêndice A. A partir da metodologia da Seção C.1, são definidos quatro tipos de estruturas (Seção C.2) e são apresentadas ameaças à validade dessa definição (Seção C.3).

C.1 METODOLOGIA

Esta análise tem como objetivo principal definir estruturas de teste de integração baseadas em planejamento em IA. A Figura C.1 descreve a metodologia utilizada para atingir esse objetivo. Conforme indicado, a definição da estrutura parte da análise de 8 estudos que utilizam o planejamento em IA no teste de integração. Esses estudos foram identificados na amostra de 35 estudos selecionados na revisão bibliográfica da literatura do Apêndice A.

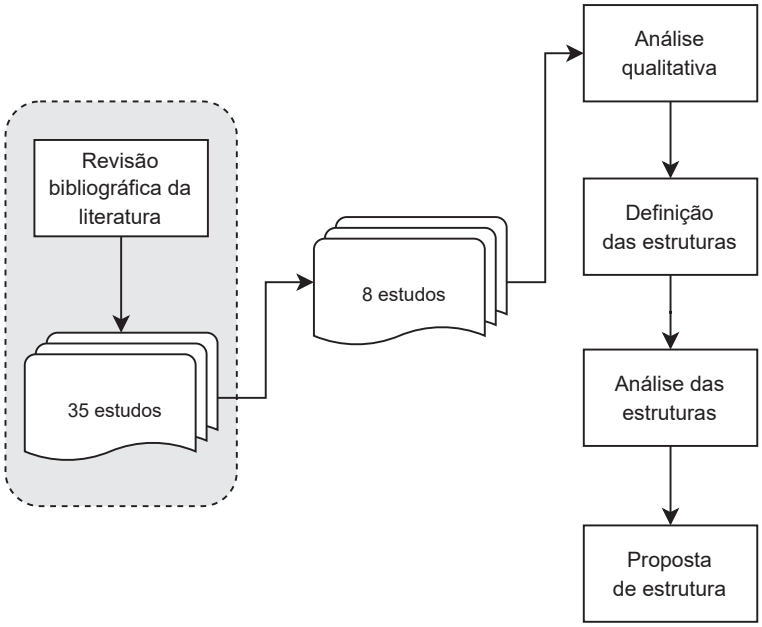


Figura C.1: Metodologia para a definição das estruturas de teste de integração.

Os 8 estudos analisados são apresentados na Tabela C.1. A partir da leitura completa desses estudos, foram identificadas as representações com planejamento em IA, as atividades, as documentações e as ferramentas usadas na condução do teste. A Subseção 3.2 detalha esses dados retirados dos estudos. A partir de uma análise qualitativa desses dados, foram definidas as estruturas de teste. Conceitos teóricos das Seções 2.1 e 2.4 complementam essas definições.

Os dados retirados dos estudos foram agregados com base na similaridade de conteúdo, resultando na categorização de quatro tipos distintos de estruturas descritas na Seção C.2. As estruturas são estruturadas em módulos e representadas graficamente conforme as definições da Seção 4.2. Cada estrutura é dividida em três módulos: (1) Planejamento do teste; (2) Planejador; e (3) Execução do teste.

Tabela C.1: Estudos sobre planejamento em IA no teste de integração

Referência	Título
Mraz et al. (1995)	*System testing with an AI planner*
Howe et al. (1995)	*Test sequences as plans: An experiment in using an AI planner to generate system tests*
Howe et al. (1997)	*Test case generation as an AI planning problem*
Scheetz et al. (1999)	*Generating test cases from an OO model with an AI planning system*
von Mayrhauser et al. (1999)	*Generating goal-oriented test cases*
von Mayrhauser et al. (2000)	*Planner based error recovery testing*
Andrews et al. (2002)	*AI planner assisted test generation*
Silva e Lemos (2011)	*Dynamic plans for integration testing of self-adaptive software systems*

Após o delineamento das estruturas, foi realizada uma análise apresentada na Seção 3.3 do Capítulo 3. A partir da análise, foram encontradas lacunas sobre o teste de software, o teste de integração, os SUTs, os elementos das estruturas e o modelo de desenvolvimento de software. Essas lacunas basearam a definição de elementos da abordagem apresentada no Capítulo 4.

C.2 RESULTADOS

As estruturas de teste identificadas baseiam-se em domínio, UML, erros e componentes. A Tabela C.2 indica as referências, uma descrição e o entendimento de unidade em cada estrutura. As próximas subseções descrevem as estruturas e apresentam suas representações gráficas baseadas na organização da Figura 4.3.

Tabela C.2: Visão geral das estruturas de teste de integração baseadas em planejamento em IA.

Estrutura	Referências	Descrição	Unidades
Domínio, Subseção C.2.1	Mraz et al. (1995), Howe et al. (1995) e Howe et al. (1997)	Proposta baseada nas funcionalidades de um SUT, descritas em termos de comandos parametrizados, representadas como um problema de planejamento. O conjunto dessas funcionalidades é chamado de domínio.	Um ou mais comandos agregados
UML, Subseção C.2.2	Scheetz et al. (1999), von Mayrhauser et al. (1999) e Andrews et al. (2002)	Proposta baseada nas informações de diagramas de classes UML de um SUT representadas como um problema de planejamento.	Classes
Erros, Subseção C.2.3	von Mayrhauser et al. (2000)	Proposta baseada na aplicação de operadores de mutação na representação de um domínio de um SUT como um problema de planejamento.	Um ou mais comandos agregados
Componentes, Subseção C.2.4	Silva e Lemos (2011)	Proposta baseada na representação da configuração arquitetural de componentes como um problema de planejamento.	Componentes

C.2.1 Estrutura baseada em domínio

A estrutura baseada em domínio foi definida a partir de Mraz et al. (1995), Howe et al. (1995) e Howe et al. (1997). A Figura C.2 ilustra a estrutura delineada para esse contexto.

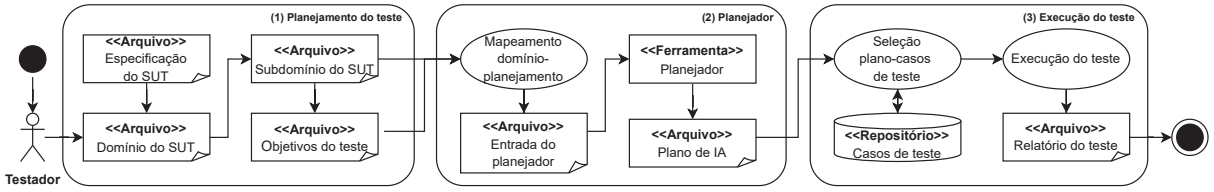


Figura C.2: Visão geral da estrutura baseada em domínio.

A abordagem começa no módulo **(1) Planejamento do teste**, com o testador (*Testador*[1]) obtendo um domínio (≪Arquivo≫ *Domínio do SUT*) a partir da especificação do SUT (≪Arquivo≫ *Especificação do SUT*). Um domínio descreve comandos parametrizados que estabelecem funcionalidades específicas do SUT. Nessa abordagem, um ou mais comandos agregados formam uma unidade do SUT.

Uma parametrização específica desses comandos ou a seleção de um subconjunto de comandos pode determinar um subdomínio (≪Arquivo≫ *Subdomínio do SUT*). Esses subdomínios podem ser associados a subsistemas gerados pela integração de unidades. Cada funcionalidade de um subdomínio gera um objetivo distinto para o teste (≪Arquivo≫ *Objetivos do teste*). Assim, definir um objetivo de teste equivale a estabelecer uma sequência válida de comandos com parâmetros significativos para testar a funcionalidade vinculada ao subdomínio.

No módulo **(2) Planejador**, o subdomínio e os objetivos do teste são mapeados como um problema de planejamento em IA por meio de um mecanismo de conversão (*Mapeamento domínio-planejamento*). Durante esse processo, os comandos são mapeados como ações, as condições que satisfazem a execução de um comando são mapeadas como pré-condições, enquanto os efeitos da ações expressam condições verdadeiras após a execução de um comando.

Nessa representação, o estado inicial contém um conjunto de comandos parametrizados anteriores à execução da funcionalidade. Já o estado final expressa a parametrização específica alcançada após a execução dos comandos. Esse mapeamento resulta em um arquivo (≪Arquivo≫ *Entrada do planejador*) que é submetido ao planejador (≪Ferramenta≫ *Planejador*), o qual produz um plano de IA (≪Arquivo≫ *Plano de IA*) para o problema representado. Os planos de IA indicam uma sequência de comandos parametrizados para o teste da funcionalidade.

No módulo **(3) Execução do teste**, os planos de IA passam por um mecanismo (*Seleção plano-casos de teste*) que associa o conjunto de comandos indicados nas ações aos casos de teste necessários para executar o teste. Por fim, esses casos de teste são recuperados em um repositório (≪Repositório≫ *Casos de teste*) e posteriormente executados (*Execução do teste*). Os resultados das execuções são descritos em um relatório (≪Arquivo≫ *Relatório do teste*).

C.2.2 Estrutura baseada em UML

A estrutura baseada em UML, ilustrada na Figura C.3, foi estabelecida a partir das abordagens de Scheetz et al. (1999), von Mayrhauser et al. (1999) e Andrews et al. (2002).

No módulo **(1) Planejamento do teste**, o testador (*Testador*[2]) define um domínio (≪Arquivo≫ *Domínio do SUT*) derivado da especificação funcional do SUT (≪Arquivo≫ *Especificação do SUT*). Utilizando o domínio e outras outras documentações disponíveis, como o manual do usuário, a arquitetura conceitual do SUT é descrita em diagramas de classes ou estados UML (≪Arquivo≫ *Diagrama UML*).

[1]Esta subseção destaca os elementos da Figura C.2.
[2]Esta subseção destaca os elementos da Figura C.3.

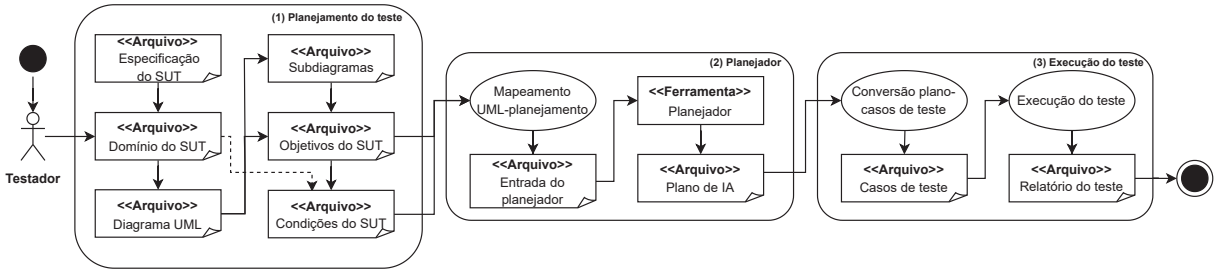


Figura C.3: Visão geral da estrutura baseada em UML.

Os diagramas UML podem ser divididos em subdiagramas (≪Arquivo≫ *Subdiagramas*), cada um contendo uma ou mais classes integradas que representam as unidades do SUT. A partir do modelo UML, é possível estabelecer objetivos para os testes (≪Arquivo≫ *Objetivos do teste*). Esses objetivos abrangem classes individuais ou grupos de classes e englobam suas interações e relações descritas nos diagramas. O domínio e os objetivos de teste são usados para definir condições (≪Arquivo≫ *Condições*) incluídas posteriormente na representação do problema.

As condições podem ser persistentes, descrevendo condições fixas para o teste, ou variáveis, que abrangem configurações do SUT que mudam de acordo com a tarefa. No módulo **(2) Planejador**, os objetivos de teste e as condições são mapeadas para uma linguagem de planejamento em IA por um mecanismo de conversão (*Mapeamento UML-planejamento*). Nessa abordagem, as condições são indicadas nos estados inicial e final do problema e os métodos das classes descritas nos diagramas são mapeados como as ações.

A partir da representação gera-se um arquivo (≪Arquivo≫ *Entrada do planejador*) utilizado como entrada do planejador (≪Ferramenta≫ *Planejador*), que então constroi o plano de IA (≪Arquivo≫ *Plano de IA*). No módulo **(3) Execução do teste**, um processo de conversão (*Conversão plano-casos de teste*) mapeia as ações do plano de IA em um conjunto de casos de teste executáveis (≪Arquivo≫ *Casos de teste*). Então, os casos de teste são executados (*Execução do teste*) resultando em um relatório (≪Arquivo≫ *Relatório do teste*).

C.2.3 Estrutura baseada em erros

A estrutura baseada em erros foi estabelecida a partir da proposta de von Mayrhauser et al. (2000), sendo parcialmente baseada nas estruturas baseadas em domínio e UML. Dessa forma, essa estrutura incorpora características similares às demais abordagens no que diz respeito ao teste de integração, como o teste de subconjuntos de comandos, a parametrização de subdomínios e a definição de objetivos de teste associados com a interações de unidades. A Figura C.4 ilustra a estrutura baseada em erros.

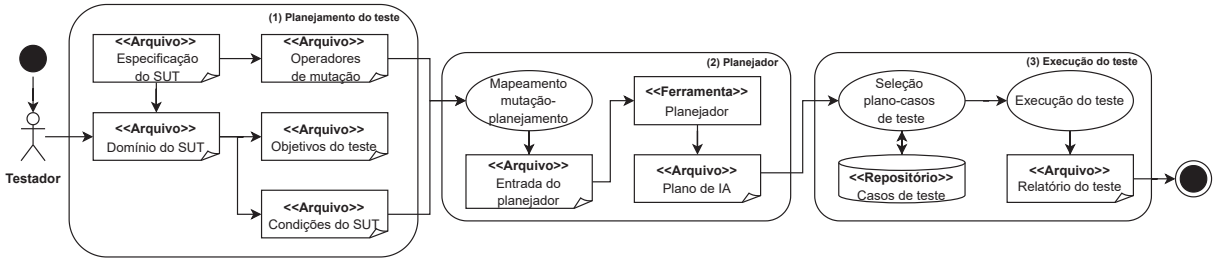


Figura C.4: Visão geral da estrutura baseada em erros.

No módulo **(1) Planejamento do teste**, o testador (*Testador*[3]) estabelece um domínio (≪Arquivo≫ *Domínio do SUT*) usando informações da especificação do SUT (≪Arquivo≫ *Especificação do SUT*) situado em contextos críticos. A partir do domínio, os objetivos do teste (≪Arquivo≫ *Objetivos do teste*) e as condições iniciais persistentes e variáveis (≪Arquivo≫ *Condições do SUT*) são definidos.

Essa abordagem adapta conceitos de teste de mutação. Para isso, a partir da especificação do SUT, é gerado um documento listando os principais operadores de mutação (≪Arquivo≫ *Operadores de mutação*). No módulo **(2) Planejador**, um mecanismo de conversão (*Mapeamento mutação-planejamento*) é usado para criar uma representação formal do SUT, fundamentada nos objetivos de teste, nas condições e nos operadores de mutação. Assim, essa representação tem como objetivo criar uma representação errônea do SUT (≪Arquivo≫ *Entrada do planejador*).

Os operadores de mutação são aplicados nas pré-condições e efeitos das ações. Esses operadores abrangem a inclusão e a remoção de parâmetros ou predicados, alterações nos valores de parâmetros, substituição de parâmetros, substituição de nomes de ações, troca de conectivos lógicos e negação de predicados. Então, a representação é usada como entrada do planejador (≪Ferramenta≫ *Planejador*) para gerar o plano de IA (≪Arquivo≫ *Plano de IA*).

Então, no módulo **(3) Execução do teste**, um mecanismo (*Seleção plano-casos de teste*) seleciona em um repositório (≪Repositório≫ *Casos de teste*) os casos de teste indicados nas ações dos planos de IA. Ao utilizar os operadores de mutação, as ações indicam diferentes casos de teste quando comparadas com a descrição original do SUT. Isso garante que a execução do teste (*Execução do teste*) ocorra com casos de teste que representam testes de recuperação de erro. A saída gerada é um relatório (≪Arquivo≫ *Relatório do teste*) com os resultados do teste.

C.2.4 Estrutura baseada em componentes

A estrutura baseada em componentes foi definida a partir do trabalho de Silva e Lemos (2011), focando na reconfiguração do SUT decorrente da integração de componentes de sistemas autoadaptativos. A Figura C.5 ilustra essa estrutura.

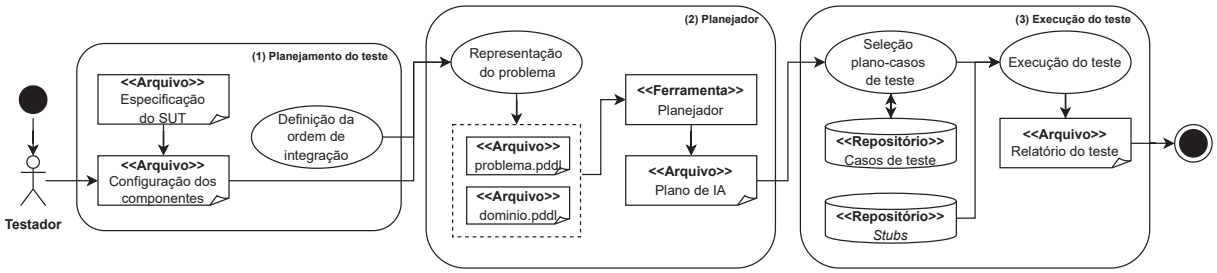


Figura C.5: Visão geral da estrutura baseada em componentes.

No módulo **(1) Planejamento do teste**, o testador (*Testador*[4]) usa a especificação do SUT (≪Arquivo≫ *Especificação do SUT*) para definir um diagrama que representa a configuração arquitetural dos componentes do SUT (≪Arquivo≫ *Configuração dos componentes*). Cada componente é visto como uma unidade e pode ser entendido como uma porção reutilizável do SUT composta por interfaces bem definidas. A estrutura inclui um mecanismo (*Definição da ordem de integração)* responsável por determinar uma ordenação para integrar os componentes.

No módulo **(2) Planejador**, as informações da configuração são usadas para definir o problema de planejamento em IA (*Representação do problema*). Essa representação (≪Arquivo≫

[3]Esta subseção destaca os elementos da Figura C.4.

[4]Esta subseção destaca os elementos da Figura C.5.

problema.pddl e ≪Arquivo≫ *dominio.pddl*) é feita em PDDL e permite que a ordenação seja indicada por valores numéricos. Nessa representação, o estado inicial estabelece a etapa atual do teste e indica a ordem de integração dos componentes envolvidos. O estado final é alcançado quando todos os componentes foram integrados e testados.

Utilizando a representação PDDL, o planejador (≪Ferramenta≫ *Planejador*) gera um plano de IA (≪Arquivo≫ *Plano de IA*) que especifica a ordem de integração e de teste dos componentes. No módulo **(3) Execução do teste**, ocorre a seleção de casos de teste (*Seleção plano-casos de teste)* em um repositório (≪Repositório≫ *Casos de teste*). Um repositório (≪Repositório≫ *Stubs*) disponibiliza os *stubs* necessários para a execução do teste (*Execução do teste*). Finalmente, os casos de teste são executados em conformidade com a ordem indicada no plano de IA, gerando um relatório de resultados (≪Arquivo≫ *Relatório do teste*).

C.3 AMEAÇAS À VALIDADE

Algumas limitações e ameaças à validade foram identificadas durante o delineamento das estruturas. No decorrer desta análise, buscou-se ativamente diminuir potenciais riscos.

Parte das elementos das estruturas foram baseados em informações adicionais às apresentadas nos estudos. As estruturas da Seção C.2 foram definidas principalmente a partir de informações dos estudos indicados na Tabela C.1 e descritos na Seção 3.2. Certos aspectos foram definidos a partir dos conceitos da Seção 2.1, como a inclusão do artefato de relatório de teste. Como parte da padronização, as estruturas também incluem os elementos da Figura 4.3.

O período de publicação dos estudos pode ser entendido como uma ameaça à validade dos resultados. Os estudos analisados foram identificados na revisão bibliográfica do Apêndice A e publicados entre 1995 e 2011. Embora a revisão tenha destacado o planejamento de IA em estudos recentes, os resultados não indicaram estudos atuais no contexto de teste de integração. Esse fato motivou a investigação nesta análise a partir de propostas mais antigas.

As interpretações usadas na definição de estruturas podem enviesar os resultados. Para minimizar as ameaças decorrentes disso, as estruturas da Seção C.2 foram revisadas por pares. Dois pesquisadores discutiram e refinaram as definições das estruturas em um processo iterativo. Ainda, as definições das estruturas foram parcialmente baseadas nos resultados da revisão bibliográfica do Apêndice A, também conduzida e revisada por pares.

As estruturas das Seções C.2 foram ilustradas em diagramas de fluxo. Entende-se que essa definição como parte de um projeto preliminar de implementação dessas estruturas. Portanto, esta análise focou na organização das estruturas e em detalhes de implementação. Por esse motivo, não foi conduzido um estudo de validação. Após as futuras etapas de implementação, espera-se realizar estudos aplicando as estruturas em casos concretos inspirados na indústria. Dessa forma, será possível observar o auxílio das estruturas na prática.

C.4 CONSIDERAÇÕES

Com base em oito estudos encontrados na revisão bibliográfica, foram delineados quatro tipos de estruturas de teste de integração com planejamento em IA. Ao analisar essas estruturas, foi possível observar que essas soluções têm como principais características otimizar a seleção de casos de teste, identificar dependências entre componentes, priorizar testes críticos e representar o conhecimento de diferentes domínios. A análise dessas estruturas destacou lacunas de pesquisa usadas como base para a definição da abordagem do Capítulo 4.

APÊNDICE D – INSTÂNCIAS DAS REPRESENTAÇÕES PDDL DA ABORDAGEM

As seções a seguir apresentam exemplos de instâncias para as representações PDDL descritas nas Seções 4.3 e 4.4 do Capítulo 4. As instâncias das Seções D.1 e D.2 se referem ao sistema procedimental ilustrado na Figura 4.5. As instâncias das Seções D.3 e D.4 foram obtidas a partir do sistema OO da Figura 4.6. Todas as instâncias apresentadas consideram ambientes com dois testadores.

D.1 INSTÂNCIA DO TESTE DE INTEGRAÇÃO *TOP-DOWN*

Código D.1: Domínio PDDL para uma instância do teste de integração *top-down*.

```
(define (domain integration_topdown)
  (:requirements
    :typing
    :disjunctive-preconditions
    :fluents
    :negative-preconditions)

  (:types
    t_procedure
    t_tester
    t_priority
    t_level
    t_parameter
    t_parameter_type
    t_return_type
  )

  (:predicates
    (priority ?pr - t_priority)
    (p_priority ?proc - t_procedure ?pr - t_priority)
    (operation_call ?proc ?proc2 - t_procedure)
    (tester ?t - t_tester)
    (done_all ?proc - t_procedure)
    (done_tester ?proc - t_procedure ?t - t_tester)
    (level ?l - t_level)
    (p_level ?proc - t_procedure ?l - t_level)
    (p_parameter ?proc - t_procedure ?pa - t_parameter)
    (pa_type ?pa - t_parameter ?pt - t_parameter_type)
    (p_return ?proc - t_procedure ?rt - t_return_type)
    (checked_parameter ?pa - t_parameter ?proc - t_procedure)
    (checked_procedure_information ?proc - t_procedure)
    (integrated_procedure ?proc - t_procedure)
  )

  (:functions
    (size ?proc - t_procedure)
    (capacity ?t - t_tester)
    (max_capacity)
    (weight ?pr - t_priority)
    (sum))

  (:action PROCEDURE_PARAMETER
```

```
  :parameters (?proc - t_procedure ?pa - t_parameter
               ?pt - t_parameter_type)

  :precondition (and
                  (p_parameter ?proc ?pa)
                  (pa_type ?pa ?pt)
                  (not (checked_parameter ?pa ?proc))
                )

  :effect (checked_parameter ?pa ?proc)
)

(:action CHECKED_PARAMETER_INFORMATION
  :parameters (?proc - t_procedure)

  :precondition (and
                  (not (integrated_procedure ?proc))
                  (not (checked_procedure_information ?proc))
                  (forall (?pa - t_parameter)
                    (or
                      (not (p_parameter ?proc ?pa))
                      (checked_parameter ?pa ?proc)))
                )

  :effect (checked_procedure_information ?proc)
)

(:action INTEGRATION_AND_TEST

  :parameters (?proc - t_procedure ?pr - t_priority
               ?t - t_tester ?rt - t_return_type ?l - t_level)

  :precondition (and
                  (priority ?pr)
                  (tester ?t)
                  (level ?l)
                  (p_priority ?proc ?pr)
                  (p_level ?proc ?l)
                  (p_return ?proc ?rt)
                  (checked_procedure_information ?proc)
                  (not (integrated_procedure ?proc))
                  (>= (capacity ?t) (size ?proc))
                  (forall (?tester - t_tester)
                    (not (done_tester ?proc ?tester)))
                )

  :effect (and
            (done_tester ?proc ?t)
            (decrease (capacity ?t) (size ?proc))
            (increase (sum) (weight ?pr))
            (not (priority ?pr))
            (priority PR1)
            (when (tester tester1)
              (and (not (tester tester1)) (tester tester2)))
            (when (tester tester2)
              (and (not (tester tester2)) (tester tester1)))
            (integrated_procedure ?proc)
          )
```

```
)

(:action INCREASE_PRIORITY
 :parameters ()

 :precondition (and )

 :effect (and
           (when (priority PR1)
            (and (not (priority PR1)) (priority PR2)))
           (when (priority PR2)
            (and (not (priority PR2)) (priority PR3)))
           (when (priority PR3)
            (and (not (priority PR3)) (priority PR4)))
           (when (priority PR4)
            (and (not (priority PR4)) (priority PR5)))
           (when (priority PR5)
            (and (not (priority PR5)) (priority PR6)))
           (when (priority PR6)
            (and (not (priority PR6)) (priority PR0)))
         )
)

(:action CHANGE_TESTER
 :parameters ()

 :precondition (and )

 :effect (and
           (when (tester tester1)
            (and (not (tester tester1)) (tester tester2)))
           (when (tester tester2)
            (and (not (tester tester2)) (tester tester1)))
           (forall (?pr - t_priority)
            (not (priority ?pr)))
           (priority PR1)
         )
)

(:action NEW_SPRINT
 :parameters ()

 :precondition (and
                 (forall (?t - t_tester)
                   (= (capacity ?t) 0))
               )

 :effect (and
           (forall (?proc - t_procedure ?t - t_tester)
            (when (done_tester ?proc ?t) (done_all ?proc)))
           (forall (?t - t_tester)
            (and
             (assign (capacity ?t)(max_capacity))
             (not (tester ?t))))
           (tester tester1)
           (forall (?pr - t_priority)
            (not (priority ?pr)))
           (priority PR1)
```

```
            )
  )

  (:action RESET_CAPACITY
    :parameters (?t - t_tester)

    :precondition (and
                    (priority PR0)
                    (tester ?t)
                  )

    :effect (and
              (assign (capacity ?t) 0)
                (when (tester tester1)
                  (and (not (tester tester1)) (tester tester2)))
                (when (tester tester2)
                  (and (not (tester tester2)) (tester tester1)))
                (not (priority PR0))
                (priority PR1)
             )
  )

  (:action CHANGE_LEVEL
    :parameters (?l - t_level)

    :precondition (and
                    (level ?l)
                    (forall (?proc - t_procedure)
                      (or
                        (not (p_level ?proc ?l))
                        (integrated_procedure ?proc)))
                  )

    :effect (and
              (not (level ?l))
                (when (level L1)
                  (and (not (level L1)) (level L2)))
                (when (level L2)
                  (and (not (level L2)) (level L3)))
            )
  )
)
```

Código D.2: Problema PDDL para uma instância do teste de integração *top-down*.

```
(define (problem integration_topdown)
  (:domain integration_topdown)

  (:objects
    PROC1 PROC2 PROC3 PROC4 PROC5 PROC6 - t_procedure
    tester1 tester2 - t_tester
    PR0 PR1 PR2 PR3 PR4 PR5 PR6 - t_priority
    L1 L2 L3 - t_level
    PA1 PA2 PA3 - t_parameter
    PT1 PT2 - t_parameter_type
    RT1 RT2 RT3 - t_return_type
  )

```

```
  (:init
    (priority PR1)
    (tester tester1)
    (level L1)
    (operation_call PROC1 PROC2)
    (operation_call PROC1 PROC3)
    (operation_call PROC1 PROC4)
    (operation_call PROC2 PROC5)
    (operation_call PROC4 PROC6)
    (p_priority PROC1 PR1)
    (p_priority PROC2 PR2)
    (p_priority PROC3 PR3)
    (p_priority PROC4 PR4)
    (p_priority PROC5 PR5)
    (p_priority PROC6 PR6)
    (p_level PROC1 L1)
    (p_level PROC2 L2)
    (p_level PROC3 L2
    (p_level PROC4 L2)
    (p_level PROC5 L3)
    (p_level PROC6 L3)
    (p_parameter PROC1 PA1)
    (p_parameter PROC1 PA2)
    (p_parameter PROC1 PA3)
    (p_parameter PROC2 PA1)
    (p_parameter PROC3 PA1)
    (p_parameter PROC4 PA1)
    (p_parameter PROC5 PA1)
    (p_parameter PROC6 PA1)
    (pa_type PA1 PT1)
    (pa_type PA2 PT1)
    (pa_type PA3 PT2)
    (p_return PROC1 RT1)
    (p_return PROC2 RT1)
    (p_return PROC3 RT2)
    (p_return PROC4 RT2)
    (p_return PROC5 RT3)
    (p_return PROC6 RT3)
    (= (size PROC1) 3)
    (= (size PROC2) 3)
    (= (size PROC3) 3)
    (= (size PROC4) 3)
    (= (size PROC5) 3)
    (= (size PROC6) 3)
    (= (weight PR6) 1)
    (= (weight PR5) 2)
    (= (weight PR4) 3)
    (= (weight PR3) 4)
    (= (weight PR2) 5)
    (= (weight PR1) 6)
    (= (max_capacity) 6)
    (= (capacity tester1) 6)
    (= (capacity tester2) 6)
    (= (sum) 0)
  )

  (:goal (and (= (sum) 21)))
)
```

Código D.3: Plano de IA para uma instância do teste de integração *top-down*.

```
0: PROCEDURE_PARAMETER PROC1 PA1 PT1
1: PROCEDURE_PARAMETER PROC1 PA2 PT1
2: PROCEDURE_PARAMETER PROC1 PA3 PT2
3: CHECKED_PARAMETER_INFORMATION PROC1
4: INTEGRATION_AND_TEST PROC1 PR1 TESTER1 RT1 L1
5: PROCEDURE_PARAMETER PROC2 PA1 PT1
6: INCREASE_PRIORITY
7: CHANGE_LEVEL L1
8: CHECKED_PARAMETER_INFORMATION PROC2
9: INTEGRATION_AND_TEST PROC2 PR2 TESTER2 RT1 L2
10: INCREASE_PRIORITY
11: PROCEDURE_PARAMETER PROC4 PA1 PT1
12: PROCEDURE_PARAMETER PROC3 PA1 PT1
13: INCREASE_PRIORITY
14: INCREASE_PRIORITY
15: CHECKED_PARAMETER_INFORMATION PROC4
16: INTEGRATION_AND_TEST PROC4 PR4 TESTER1 RT2 L2
17: INCREASE_PRIORITY
18: INCREASE_PRIORITY
19: CHECKED_PARAMETER_INFORMATION PROC3
20: INTEGRATION_AND_TEST PROC3 PR3 TESTER2 RT2 L2
21: PROCEDURE_PARAMETER PROC5 PA1 PT1
22: CHANGE_LEVEL L2
23: CHECKED_PARAMETER_INFORMATION PROC5
24: NEW_SPRINT
25: INCREASE_PRIORITY
26: INCREASE_PRIORITY
27: INCREASE_PRIORITY
28: INCREASE_PRIORITY
29: INTEGRATION_AND_TEST PROC5 PR5 TESTER1 RT3 L3
30: INCREASE_PRIORITY
31: INCREASE_PRIORITY
32: INCREASE_PRIORITY
33: INCREASE_PRIORITY
34: PROCEDURE_PARAMETER PROC6 PA1 PT1
35: INCREASE_PRIORITY
36: CHECKED_PARAMETER_INFORMATION PROC6
37: INTEGRATION_AND_TEST PROC6 PR6 TESTER2 RT3 L3
```

D.2 INSTÂNCIA DO TESTE DE INTEGRAÇÃO *BOTTOM-UP*

Código D.4: Domínio PDDL para uma instância do teste de integração *bottom-up*.

```
(define (domain integration_bottomup)
  (:requirements
   :typing
   :disjunctive-preconditions
   :fluents
   :negative-preconditions)

  (:types
   t_procedure
   t_tester
   t_priority
   t_level
   t_parameter
   t_parameter_type
   t_return_type
  )

  (:predicates
   (priority ?pr - t_priority)
   (p_priority ?proc - t_procedure ?pr - t_priority)
   (operation_call ?proc ?proc2 - t_procedure)
   (tester ?t - t_tester)
   (done_all ?proc - t_procedure)
   (done_tester ?proc - t_procedure ?t - t_tester)
   (level ?l - t_level)
   (p_level ?proc - t_procedure ?l - t_level)
   (p_parameter ?proc - t_procedure ?pa - t_parameter)
   (pa_type ?pa - t_parameter ?pt - t_parameter_type)
   (p_return ?proc - t_procedure ?rt - t_return_type)
   (checked_parameter ?pa - t_parameter ?proc - t_procedure)
   (checked_procedure_information ?proc - t_procedure)
   (integrated_procedure ?proc - t_procedure)
  )

  (:functions
   (size ?proc - t_procedure)
   (capacity ?t - t_tester)
   (max_capacity)
   (weight ?pr - t_priority)
   (sum)
  )

  (:action PROCEDURE_PARAMETER
   :parameters (?proc - t_procedure ?pa - t_parameter
                ?pt - t_parameter_type)

   :precondition (and
                   (p_parameter ?proc ?pa)
                   (pa_type ?pa ?pt)
                   (not (checked_parameter ?pa ?proc))
                 )

   :effect (checked_parameter ?pa ?proc)
  )
```

```
55
56  (:action CHECKED_PARAMETER_INFORMATION
57   :parameters (?proc - t_procedure)
58
59   :precondition (and
60                  (not (integrated_procedure ?proc))
61                  (not (checked_procedure_information ?proc))
62                  (forall (?pa - t_parameter)
63                   (or
64                    (not (p_parameter ?proc ?pa))
65                    (checked_parameter ?pa ?proc)))
66                )
67
68   :effect (checked_procedure_information ?proc)
69  )
70
71  (:action INTEGRATION_AND_TEST
72   :parameters (?proc - t_procedure ?pr - t_priority
73                ?t - t_tester ?rt - t_return_type ?l - t_level)
74
75   :precondition (and
76                  (priority ?pr)
77                  (tester ?t)
78                  (level ?l)
79                  (p_priority ?proc ?pr)
80                  (p_level ?proc ?l)
81                  (p_return ?proc ?rt)
82                  (checked_procedure_information ?proc)
83                  (not (integrated_procedure ?proc))
84                  (>= (capacity ?t) (size ?proc))
85                  (forall (?tester - t_tester)
86                   (not (done_tester ?proc ?tester)))
87                  (forall (?proc2 - t_procedure)
88                   (or
89                    (not (operation_call ?proc ?proc2))
90                    (done_tester ?proc2 ?t)
91                    (done_all ?proc2)))
92                )
93
94   :effect (and
95             (done_tester ?proc ?t)
96             (decrease (capacity ?t) (size ?proc))
97             (increase (sum) (weight ?pr))
98             (not (priority ?pr))
99             (priority PR1)
100            (when (tester tester1)
101             (and (not (tester tester1)) (tester tester2)))
102            (when (tester tester2)
103             (and (not (tester tester2)) (tester tester1)))
104            (integrated_procedure ?proc)
105           )
106 )
107
108 (:action INCREASE_PRIORITY
109  :parameters ()
110
111  :precondition (and )
112
```

```
  :effect (and
            (when (priority PR1)
              (and (not (priority PR1)) (priority PR2)))
            (when (priority PR2)
              (and (not (priority PR2)) (priority PR3)))
            (when (priority PR3)
              (and (not (priority PR3)) (priority PR4)))
            (when (priority PR4)
              (and (not (priority PR4)) (priority PR5)))
            (when (priority PR5)
              (and (not (priority PR5)) (priority PR6)))
            (when (priority PR6)
              (and (not (priority PR6)) (priority PR0)))
          )
)

(:action CHANGE_TESTER
  :parameters ()

  :precondition (and )

  :effect (and
            (when (tester tester1)
              (and (not (tester tester1)) (tester tester2)))
             (when (tester tester2)
              (and (not (tester tester2)) (tester tester1)))
             (forall (?pr - t_priority)
              (not (priority ?pr)))
             (priority PR1)
           )
)

(:action NEW_SPRINT
  :parameters ()

  :precondition (and
                  (forall (?t - t_tester)
                    (= (capacity ?t) 0))
                )

  :effect (and
            (forall (?proc - t_procedure ?t - t_tester)
              (when (done_tester ?proc ?t) (done_all ?proc)))
            (forall (?t - t_tester)
              (and
                (assign (capacity ?t)(max_capacity))
                (not (tester ?t))))
            (tester tester1)
            (forall (?pr - t_priority)
              (not (priority ?pr)))
            (priority PR1)
          )
)

(:action RESET_CAPACITY
  :parameters (?t - t_tester)

  :precondition (and
```

```
                    (priority PR0)
                    (tester ?t)
                  )

    :effect (and
              (assign (capacity ?t) 0)
               (when (tester tester1)
                 (and (not (tester tester1)) (tester tester2)))
               (when (tester tester2)
                 (and (not (tester tester2)) (tester tester1)))
              (not (priority PR0))
              (priority PR1)
            )
  )

  (:action CHANGE_LEVEL
    :parameters (?l - t_level)

    :precondition (and
                    (level ?l)
                    (forall (?proc - t_procedure)
                      (or
                        (not (p_level ?proc ?l))
                        (integrated_procedure ?proc)))
                  )

    :effect (and
              (not (level ?l))
               (when (level L3)
                 (and (not (level L3)) (level L2)))
               (when (level L2)
                 (and (not (level L2)) (level L1)))
            )
  )
)
```

Código D.5: Problema PDDL para uma instância do teste de integração *bottom-up*.

```
(define (problem integration_bottomup)
  (:domain integration_bottomup)

  (:objects
    PROC1 PROC2 PROC3 PROC4 PROC5 PROC6 - t_procedure
    tester1 tester2 - t_tester
    PR0 PR1 PR2 PR3 PR4 PR5 PR6 - t_priority
    L1 L2 L3 - t_level
    PA1 PA2 PA3 - t_parameter
    PT1 PT2 - t_parameter_type
    RT1 RT2 RT3 - t_return_type
  )

  (:init
    (priority PR1)
    (tester tester1)
    (level L3)
    (operation_call PROC1 PROC2)
    (operation_call PROC1 PROC3)
    (operation_call PROC1 PROC4)
```

```
    (operation_call PROC2 PROC5)
    (operation_call PROC4 PROC6)
    (p_priority PROC1 PR1)
    (p_priority PROC2 PR2)
    (p_priority PROC3 PR3)
    (p_priority PROC4 PR4)
    (p_priority PROC5 PR5)
    (p_priority PROC6 PR6)
    (p_level PROC1 L1)
    (p_level PROC2 L2)
    (p_level PROC3 L2)
    (p_level PROC4 L2)
    (p_level PROC5 L3)
    (p_level PROC6 L3)
    (p_parameter PROC1 PA1)
    (p_parameter PROC1 PA2)
    (p_parameter PROC1 PA3)
    (p_parameter PROC2 PA1)
    (p_parameter PROC3 PA1)
    (p_parameter PROC4 PA1)
    (p_parameter PROC5 PA1)
    (p_parameter PROC6 PA1)
    (pa_type PA1 PT1)
    (pa_type PA2 PT1)
    (pa_type PA3 PT2)
    (p_return PROC1 RT1)
    (p_return PROC2 RT1)
    (p_return PROC3 RT2)
    (p_return PROC4 RT2)
    (p_return PROC5 RT3)
    (p_return PROC6 RT3)
    (= (size PROC1) 3)
    (= (size PROC2) 3)
    (= (size PROC3) 3)
    (= (size PROC4) 3)
    (= (size PROC5) 3)
    (= (size PROC6) 3)
    (= (weight PR6) 1)
    (= (weight PR5) 2)
    (= (weight PR4) 3)
    (= (weight PR3) 4)
    (= (weight PR2) 5)
    (= (weight PR1) 6)
    (= (max_capacity) 6)
    (= (capacity tester1) 6)
    (= (capacity tester2) 6)
    (= (sum) 0)
  )

  (:goal (and (= (sum) 21)))
)
```

Código D.6: Plano de IA para uma instância do teste de integração *bottom-up*.

```
0: PROCEDURE_PARAMETER PROC6 PA1 PT1
1: PROCEDURE_PARAMETER PROC5 PA1 PT1
2: PROCEDURE_PARAMETER PROC2 PA1 PT1
3: CHECKED_PARAMETER_INFORMATION PROC6
```

```
4: CHECKED_PARAMETER_INFORMATION PROC5
5: INCREASE_PRIORITY
6: INCREASE_PRIORITY
7: INCREASE_PRIORITY
8: INCREASE_PRIORITY
9: CHECKED_PARAMETER_INFORMATION PROC2
10: INCREASE_PRIORITY
11: INTEGRATION_AND_TEST PROC6 PR6 TESTER1 RT3 L3
12: PROCEDURE_PARAMETER PROC3 PA1 PT1
13: CHECKED_PARAMETER_INFORMATION PROC3
14: INCREASE_PRIORITY
15: INCREASE_PRIORITY
16: INCREASE_PRIORITY
17: INCREASE_PRIORITY
18: INTEGRATION_AND_TEST PROC5 PR5 TESTER2 RT3 L3
19: INCREASE_PRIORITY
20: CHANGE_LEVEL L3
21: INCREASE_PRIORITY
22: INTEGRATION_AND_TEST PROC3 PR3 TESTER1 RT2 L2
23: INCREASE_PRIORITY
24: INTEGRATION_AND_TEST PROC2 PR2 TESTER2 RT1 L2
25: PROCEDURE_PARAMETER PROC4 PA1 PT1
26: CHECKED_PARAMETER_INFORMATION PROC
27: NEW_SPRINT
28: INCREASE_PRIORITY
29: INCREASE_PRIORITY
30: INCREASE_PRIORITY
31: INTEGRATION_AND_TEST PROC4 PR4 TESTER1 RT2 L2
32: PROCEDURE_PARAMETER PROC1 PA1 PT1
33: PROCEDURE_PARAMETER PROC1 PA2 PT1
34: PROCEDURE_PARAMETER PROC1 PA3 PT2
35: CHANGE_TESTER
36: CHANGE_LEVEL L2
37: CHECKED_PARAMETER_INFORMATION PROC1
38: INTEGRATION_AND_TEST PROC1 PR1 TESTER1 RT1 L1
```

D.3 INSTÂNCIA DO TESTE DE INTEGRAÇÃO INTERMÉTODOS

Código D.7: Domínio PDDL para uma instância do teste de integração intermétodos.

```
(define (domain integration_methods)
  (:requirements
   :typing
   :disjunctive-preconditions
   :fluents
   :negative-preconditions)

  (:types
   t_class
   t_method
   t_tester
   t_mpriority
   t_argument
   t_argument_type
   t_return_type
  )

  (:predicates
    (mpriority ?prm - t_mpriority)
    (method_priority ?m - t_method ?prm - t_mpriority)
    (message ?m ?me - t_method)
    (tester ?t - t_tester)
    (done_all ?m - t_method)
    (done_tester ?m - t_method ?t - t_tester)
    (class ?c - t_class)
    (integrated_method ?m - t_method)
    (class_method ?m - t_method ?c - t_class)
    (method_argument ?m - t_method ?arg - t_argument)
    (arg_type ?arg - t_argument ?argt - t_argument_type)
    (method_return ?m - t_method ?rtm - t_return_type)
    (checked_argument ?arg - t_argument ?m - t_method)
    (checked_argument_information ?m - t_method)
  )

  (:functions
    (method_size ?m - t_method)
    (capacity ?t - t_tester)
    (max_capacity)
    (mpriority_weight ?prm - t_mpriority)
    (sum)
  )

  (:action METHOD_ARGUMENT
    :parameters (?m - t_method ?arg - t_argument ?argt - t_argument_type)

    :precondition (and
                    (method_argument ?m ?arg)
                    (arg_type ?arg ?argt)
                    (not (checked_argument ?arg ?m))
                  )

    :effect (checked_argument ?arg ?m)
  )

```

```
(:action CHECKED_ARGUMENT_INFORMATION
 :parameters (?m - t_method)

 :precondition (and
                 (not (integrated_method ?m))
                 (not (checked_argument_information ?m))
                 (forall (?arg - t_argument)
                  (or
                   (not (method_argument ?m ?arg))
                   (checked_argument ?arg ?m)))
                )

      :effect (checked_argument_information ?m)
)

(:action METHOD_INTEGRATION_AND_TEST
 :parameters (?m - t_method ?prm - t_mpriority ?t - t_tester
              ?c - t_class ?rtm - t_return_type)

 :precondition (and
                 (mpriority ?prm)
                 (tester ?t)
                 (class ?c)
                 (method_priority ?m ?prm)
                 (class_method ?m ?c)
                 (method_return ?m ?rtm)
                 (>= (capacity ?t) (method_size ?m))
                 (not (integrated_method ?m))
                 (checked_argument_information ?m)
                 (forall (?tester - t_tester)
                  (not (done_tester ?m ?tester)))
                )

 :effect (and
           (done_tester ?m ?t)
           (decrease (capacity ?t) (method_size ?m))
           (increase (sum) (mpriority_weight ?prm))
           (not (mpriority ?prm))
           (mpriority PRM1)
           (integrated_method ?m)
           (when (tester tester1)
            (and (not (tester tester1)) (tester tester2)))
           (when (tester tester2)
            (and (not (tester tester2)) (tester tester1)))
         )
)

(:action INCREASE_PRIORITY
 :parameters ()

 :precondition (and )

 :effect (and
           (when (mpriority PRM1)
            (and (not (mpriority PRM1)) (mpriority PRM2)))
           (when (mpriority PRM2)
            (and (not (mpriority PRM2)) (mpriority PRM3)))
           (when (mpriority PRM3)
```

```
                (and (not (mpriority PRM3)) (mpriority PRM4)))
              (when (mpriority PRM4)
                (and (not (mpriority PRM4)) (mpriority PRM5)))
              (when (mpriority PRM5)
                (and (not (mpriority PRM5)) (mpriority PRM6)))
              (when (mpriority PRM6)
                (and (not (mpriority PRM6)) (mpriority PRM7)))
              (when (mpriority PRM7)
                (and (not (mpriority PRM7)) (mpriority PRM0)))
            )
  )

  (:action CHANGE_TESTER
    :parameters ()

    :precondition (and )

    :effect (and
              (when (tester tester1)
                (and (not (tester tester1)) (tester tester2)))
              (when (tester tester2)
                (and (not (tester tester2)) (tester tester1)))
              (forall (?prm - t_mpriority)
                (not (mpriority ?prm)))
              (mpriority PRM1)
            )
  )

  (:action NEW_SPRINT
    :parameters ()

    :precondition (and
                    (forall (?t - t_tester)
                      (= (capacity ?t) 0))
                  )

  :effect (and
            (forall (?m - t_method ?t - t_tester)
              (when (done_tester ?m ?t) (done_all ?m)))
            (forall (?t - t_tester)
              (and
                (assign (capacity ?t)(max_capacity))
                (not (tester ?t))))
            (tester tester1)
            (forall (?prm - t_mpriority)
              (not (mpriority ?prm)))
            (mpriority PRM1)
          )
  )

 (:action RESET_CAPACITY
   :parameters (?t - t_tester)

   :precondition (and
                   (mpriority PRM0)
                   (tester ?t)
                 )

```

```
  :effect (and
            (assign (capacity ?t) 0
            (when (tester tester1)
              (and (not (tester tester1)) (tester tester2)))
            (when (tester tester2)
              (and (not (tester tester2)) (tester tester1)))
            (not (mpriority PRM0))
            (mpriority PRM1)
          )
  )

  (:action CHANGE_CLASS
    :parameters (?c - t_class)

    :precondition (and
                    (class ?c)
                    (forall (?m - t_method)
                      (or
                        (not (class_method ?m ?c))
                        (integrated_method ?m)))
                  )

    :effect (and
              (not (class ?c))
              (when (class C1)
                (and (not (class C1)) (class C2)))
              (when (class C2)
                (and (not (class C2)) (class C3)))
              (when (class C3)
                (and (not (class C3)) (class C4)))
              (when (class C4)
                (and (not (class C4)) (class C5)))
              (when (class C5)
                (and (not (class C5)) (class C6)))
              (when (class C6)
                (and (not (class C6)) (class C7)))
              (when (class C7)
                (and (not (class C7)) (class C8)))
            )
  )
)
```

Código D.8: Problema PDDL para uma instância do teste de integração intermétodos.

```
(define (problem integration_methods)
  (:domain integration_methods)

  (:objects
    M1 M2 M3 M4 M5 M6 M7 - t_method
    C1 C2 C2 C3 C4 C5 C6 C7 C8 - t_class
    tester1 tester2 - t_tester
    PRM0 PRM1 PRM2 PRM3 PRM4 PRM5 PRM6 PRM7 - t_mpriority
    ARG1 ARG2 ARG3 ARG4 ARG5 ARG6 ARG7 - t_argument
    ARGT1 ARGT2 - t_argument_type
    RTM1 RTM2 RTM3 - t_return_type
  )

  (:init
```

```
    (mpriority PRM1)
    (tester tester1)
    (class C1)
    (message M1 M2)
    (message M1 M3)
    (message M1 M4)
    (message M2 M5)
    (message M4 M6)
    (method_priority M1 PRM1)
    (method_priority M2 PRM2)
    (method_priority M3 PRM3)
    (method_priority M4 PRM4)
    (method_priority M5 PRM5)
    (method_priority M6 PRM6)
    (method_priority M7 PRM7)
    (class_method M1 C1)
    (class_method M2 C1)
    (class_method M3 C1)
    (class_method M4 C2)
    (class_method M5 C2)
    (class_method M6 C3)
    (class_method M7 C3)
    (method_argument M1 ARG1)
    (method_argument M2 ARG2)
    (method_argument M3 ARG3)
    (method_argument M4 ARG4)
    (method_argument M5 ARG5)
    (method_argument M6 ARG6)
    (method_argument M7 ARG7)
    (arg_type ARG1 ARGT1)
    (arg_type ARG2 ARGT1)
    (arg_type ARG3 ARGT2)
    (arg_type ARG4 ARGT2)
    (arg_type ARG5 ARGT1)
    (arg_type ARG6 ARGT1)
    (arg_type ARG7 ARGT2)
    (method_return M1 RTM1)
    (method_return M2 RTM2)
    (method_return M3 RTM3)
    (method_return M4 RTM1)
    (method_return M5 RTM2)
    (method_return M6 RTM1)
    (method_return M7 RTM3)
    (= (method_size M1) 3)
    (= (method_size M2) 3)
    (= (method_size M3) 3)
    (= (method_size M4) 3)
    (= (method_size M5) 3)
    (= (method_size M6) 3)
    (= (method_size M7) 3)
    (= (mpriority_weight PRM7) 1)
    (= (mpriority_weight PRM6) 2)
    (= (mpriority_weight PRM5) 3)
    (= (mpriority_weight PRM4) 4)
    (= (mpriority_weight PRM3) 5)
    (= (mpriority_weight PRM2) 6)
    (= (mpriority_weight PRM1) 7)
    (= (max_capacity) 6)
```

```
    (= (capacity tester1) 6)
    (= (capacity tester2) 6)
    (= (sum) 0)
  )

  (:goal (and (= (sum) 28)))
)
```

Código D.9: Plano de IA para uma instância do teste de integração intermétodos.

```
0: METHOD_ARGUMENT M3 ARG3 ARGT2
1: METHOD_ARGUMENT M2 ARG2 ARGT1
2: METHOD_ARGUMENT M1 ARG1 ARGT1
3: CHECKED_ARGUMENT_INFORMATION M2
4: INCREASE_PRIORITY
5: CHECKED_ARGUMENT_INFORMATION M1
6: METHOD_INTEGRATION_AND_TEST M2 PRM2 TESTER1 C1 RTM2
7: METHOD_INTEGRATION_AND_TEST M1 PRM1 TESTER2 C1 RTM1
8: INCREASE_PRIORITY
9: INCREASE_PRIORITY
10: CHECKED_ARGUMENT_INFORMATION M3
11: METHOD_INTEGRATION_AND_TEST M3 PRM3 TESTER1 C1 RTM3
12: INCREASE_PRIORITY
13: METHOD_ARGUMENT M5 ARG5 ARGT1
14: INCREASE_PRIORITY
15: METHOD_ARGUMENT M4 ARG4 ARGT2
16: CHECKED_ARGUMENT_INFORMATION M5
17: INCREASE_PRIORITY
18: INCREASE_PRIORITY
19: CHANGE_CLASS C1
20: METHOD_INTEGRATION_AND_TEST M5 PRM5 TESTER2 C2 RTM2
21: CHECKED_ARGUMENT_INFORMATION M4
22: NEW_SPRINT
23: INCREASE_PRIORITY
24: INCREASE_PRIORITY
25: INCREASE_PRIORITY
26: METHOD_INTEGRATION_AND_TEST M4 PRM4 TESTER1 C2 RTM1
27: INCREASE_PRIORITY
28: INCREASE_PRIORITY
29: INCREASE_PRIORITY
30: INCREASE_PRIORITY
31: METHOD_ARGUMENT M6 ARG6 ARGT1
32: INCREASE_PRIORITY
33: CHANGE_CLASS C2
34: CHECKED_ARGUMENT_INFORMATION M6
35: METHOD_INTEGRATION_AND_TEST M6 PRM6 TESTER2 C3 RTM1
36: INCREASE_PRIORITY
37: INCREASE_PRIORITY
38: INCREASE_PRIORITY
39: INCREASE_PRIORITY
40: INCREASE_PRIORITY
41: METHOD_ARGUMENT M7 ARG7 ARGT2
42: INCREASE_PRIORITY
43: CHECKED_ARGUMENT_INFORMATION M7
44: METHOD_INTEGRATION_AND_TEST M7 PRM7 TESTER1 C3 RTM3
```

D.4 INSTÂNCIA DO TESTE DE INTEGRAÇÃO INTERCLASSES

Código D.10: Domínio PDDL para uma instância do teste de integração interclasses.

```
(define (domain integration_classes)
  (:requirements
    :typing
    :disjunctive-preconditions
    :fluents
    :negative-preconditions)

  (:types
    t_class
    t_method
    t_tester
    t_cpriority
    t_attribute
    t_attribute_type
  )

  (:predicates
    (cpriority ?prc - t_cpriority)
    (class_priority ?c - t_class ?prc - t_cpriority)
    (message ?m ?me - t_method)
    (tester ?t - t_tester)
    (done_all ?c - t_class)
    (done_tester ?c - t_class ?t - t_tester)
    (integrated_class ?c - t_class)
    (class_method ?m - t_method ?c - t_class)
    (class_attribute ?c - t_class ?atr - t_attribute)
    (attribute_type ?atr - t_attribute ?at - t_attribute_type)
    (checked_attribute ?atr - t_attribute ?c - t_class)
    (checked_attribute_information ?c - t_class)
    (checked_method ?m - t_method ?c - t_class)
    checked_method_information ?c - t_class)
  )

  (:functions
    (class_size ?c - t_class)
    (capacity ?t - t_tester)
    (max_capacity)
    (cpriority_weight ?prc - t_cpriority)
    (sum)
  )

  (:action CLASS_METHOD
    :parameters (?c - t_class ?m - t_method)

    :precondition (and
                    (class_method ?m ?c)
                    (not (checked_method ?m ?c))
                  )

    :effect (checked_method ?m ?c)
  )

  (:action CHECKED_METHOD_INFORMATION
     :parameters (?c - t_class)
```

```
55
56      :precondition (and
57                      (not (integrated_class ?c))
58                      (not (checked_method_information ?c))
59                      (forall (?m - t_method)
60                        (or
61                          (not (class_method ?m ?c))
62                          (checked_method ?m ?c)))
63                    )
64
65    :effect (checked_method_information ?c)
66  )
67
68  (:action CLASS_ATTRIBUTE
69    :parameters (?c - t_class ?atr - t_attribute ?at - t_attribute_type)
70
71    :precondition (and
72                    (class_attribute ?c ?atr)
73                    (attribute_type ?atr ?at)
74                    (not (checked_attribute ?atr ?c))
75                  )
76
77    :effect (checked_attribute ?atr ?c)
78  )
79
80  (:action CHECKED_ATTRIBUTE_INFORMATION
81    :parameters (?c - t_class)
82
83    :precondition (and
84                    (not (integrated_class ?c))
85                    (not (checked_attribute_information ?c))
86                    (forall (?atr - t_attribute)
87                      (or
88                        (not (class_attribute ?c ?atr))
89                        (checked_attribute ?atr ?c)))
90                  )
91
92    :effect (checked_attribute_information ?c)
93  )
94
95  (:action CLASS_INTEGRATION_AND_TEST
96    :parameters (?c - t_class ?prc - t_cpriority ?t - t_tester)
97
98    :precondition (and
99                    (cpriority ?prc)
100                   (tester ?t)
101                   (class_priority ?c ?prc)
102                   (>= (capacity ?t) (class_size ?c))
103                   (not (integrated_class ?c))
104                   (checked_attribute_information ?c)
105                   (checked_method_information ?c)
106                   (forall (?tester - t_tester)
107                     (not (done_tester ?c ?tester)))
108                 )
109
110   :effect (and
111             (done_tester ?c ?t)
112             (decrease (capacity ?t) (class_size ?c))
```

```
113               (increase (sum) (cpriority_weight ?prc))
114               (not (cpriority ?prc))
115               (cpriority PRC1)
116               (when (tester tester1)
117                 (and (not (tester tester1)) (tester tester2)))
118               (when (tester tester2)
119                 (and (not (tester tester2)) (tester tester1)))
120               (integrated_class ?c)
121             )
122   )
123
124   (:action INCREASE_PRIORITY
125     :parameters ()
126
127     :precondition (and )
128
129     :effect (and
130               (when (cpriority PRC1)
131                 (and (not (cpriority PRC1)) (cpriority PRC2)))
132               (when (cpriority PRC2)
133                 (and (not (cpriority PRC2)) (cpriority PRC3)))
134               (when (cpriority PRC3)
135                 (and (not (cpriority PRC3)) (cpriority PRC4)))
136               (when (cpriority PRC4)
137                 (and (not (cpriority PRC4)) (cpriority PRC5)))
138               (when (cpriority PRC5)
139                 (and (not (cpriority PRC5)) (cpriority PRC6)))
140               (when (cpriority PRC6)
141                 (and (not (cpriority PRC6)) (cpriority PRC7)))
142               (when (cpriority PRC7)
143                 (and (not (cpriority PRC7)) (cpriority PRC8)))
144               (when (cpriority PRC8)
145                 (and (not (cpriority PRC8)) (cpriority PRC0)))
146             )
147   )
148
149   (:action CHANGE_TESTER
150     :parameters ()
151
152     :precondition (and )
153
154     :effect (and
155               (when (tester tester1)
156                 (and (not (tester tester1)) (tester tester2)))
157               (when (tester tester2)
158                 (and (not (tester tester2)) (tester tester1)))
159               (forall
160                 (?prc - t_cpriority)
161                 (not (cpriority ?prc)))
162               (cpriority PRC1)
163             )
164   )
165
166   (:action NEW_SPRINT
167     :parameters ()
168
169     :precondition (and
170                     (forall (?t - t_tester)
```

```
                    (= (capacity ?t) 0))
                )

  :effect (and
            (forall (?c - t_class ?t - t_tester)
              (when (done_tester ?c ?t) (done_all ?c)))
            (forall (?t - t_tester)
              (and
                (assign (capacity ?t)(max_capacity))
                (not (tester ?t))))
            (tester tester1)
            (forall (?prc - t_cpriority)
              (not (cpriority ?prc)))
            (cpriority PRC1)
          )
  )

  (:action RESET_CAPACITY
    :parameters (?t - t_tester)

    :precondition (and
                    (cpriority PRC0)
                    (tester ?t)
                  )

    :effect (and
              (assign (capacity ?t) 0)
              (when (tester tester1)
                (and (not (tester tester1)) (tester tester2)))
              (when (tester tester2)
                (and (not (tester tester2)) (tester tester1)))
              (not (cpriority PRC0))
              (cpriority PRC1)
            )
  )
)
```

Código D.11: Problema PDDL para uma instância do teste de integração interclasses.

```
(define (problem integration_classes)
  (:domain integration_classes)

  (:objects
    M1 M2 M3 M4 M5 M6 M7 - t_method
    C1 C2 C2 C3 C4 C5 C6 C7 C8 - t_class
    tester1 tester2 - t_tester
    PRC0 PRC1 PRC2 PRC3 PRC4 PRC5 PRC6 PRC7 PRC8 - t_cpriority
    ATR1 ATR2 ATR3 ATR4 ATR5 ATR6 ATR7 ATR8 ATR9 - t_attribute
    AT1 AT2 - t_attribute_type
  )

  (:init
    (cpriority PRC1)
    (tester tester1)
    (message M1 M2)
    (message M1 M3)
    (message M1 M4)
    (message M2 M5)
```

```
    (message M4 M6)
    (class_priority C1 PRC1)
    (class_priority C2 PRC2)
    (class_priority C3 PRC3)
    (class_priority C4 PRC4)
    (class_priority C5 PRC5)
    (class_priority C6 PRC6)
    (class_priority C7 PRC7)
    (class_priority C8 PRC8)
    (class_method M1 C1)
    (class_method M2 C1)
    (class_method M3 C1)
    (class_method M4 C2)
    (class_method M5 C2)
    (class_method M6 C3)
    (class_method M7 C3)
    (class_attribute C1 ATR1)
    (class_attribute C1 ATR2)
    (class_attribute C1 ATR3)
    (class_attribute C1 ATR4)
    (class_attribute C1 ATR5)
    (class_attribute C2 ATR6)
    (class_attribute C2 ATR7)
    (class_attribute C3 ATR8)
    (class_attribute C3 ATR9)
    (attribute_type ATR1 AT1)
    (attribute_type ATR2 AT1)
    (attribute_type ATR3 AT1)
    (attribute_type ATR4 AT1)
    (attribute_type ATR5 AT1)
    (attribute_type ATR6 AT1)
    (attribute_type ATR7 AT1)
    (attribute_type ATR8 AT2)
    (attribute_type ATR9 AT1)
    (= (class_size C1) 3)
    (= (class_size C2) 3)
    (= (class_size C3) 3)
    (= (class_size C4) 3)
    (= (class_size C5) 3)
    (= (class_size C6) 3)
    (= (class_size C7) 3)
    (= (class_size C8) 3)
    (= (cpriority_weight PRC8) 1)
    (= (cpriority_weight PRC7) 2)
    (= (cpriority_weight PRC6) 3)
    (= (cpriority_weight PRC5) 4)
    (= (cpriority_weight PRC4) 5)
    (= (cpriority_weight PRC3) 6)
    (= (cpriority_weight PRC2) 7)
    (= (cpriority_weight PRC1) 8)
    (= (max_capacity) 6)
    (= (capacity tester1) 6)
    (= (capacity tester2) 6)
    (= (sum) 0)
  )

  (:goal and (= (sum) 36)))
)
```

Código D.12: Plano de IA para uma instância do teste de integração interclasses.

```
0: CLASS_ATTRIBUTE C3 ATR8 AT2
1: CLASS_ATTRIBUTE C3 ATR9 AT1
2: CLASS_METHOD C3 M6
3: CLASS_METHOD C3 M7
4: CLASS_ATTRIBUTE C2 ATR6 AT1
5: CLASS_ATTRIBUTE C2 ATR7 AT1
6: CLASS_METHOD C2 M4
7: CLASS_METHOD C2 M5
8: CLASS_ATTRIBUTE C1 ATR1 AT1
9: CLASS_ATTRIBUTE C1 ATR2 AT1
10: CLASS_ATTRIBUTE C1 ATR3 AT1
11: CLASS_ATTRIBUTE C1 ATR4 AT1
12: CLASS_ATTRIBUTE C1 ATR5 AT1
13: CLASS_METHOD C1 M1
14: CLASS_METHOD C1 M2
15: CLASS_METHOD C1 M3
16: CHECKED_ATTRIBUTE_INFORMATION C2
17: CHECKED_METHOD_INFORMATION C2
18: INCREASE_PRIORITY
19: CHECKED_ATTRIBUTE_INFORMATION C1
20: CHECKED_METHOD_INFORMATION C1
21: CLASS_INTEGRATION_AND_TEST C2 PRC2 TESTER1
22: CLASS_INTEGRATION_AND_TEST C1 PRC1 TESTER2
23: INCREASE_PRIORITY
24: INCREASE_PRIORITY
25: INCREASE_PRIORITY
26: CHECKED_ATTRIBUTE_INFORMATION C4
27: CHECKED_METHOD_INFORMATION C4
28: CLASS_INTEGRATION_AND_TEST C4 PRC4 TESTER1
29: INCREASE_PRIORITY
30: INCREASE_PRIORITY
31: CHECKED_ATTRIBUTE_INFORMATION C3
32: CHECKED_METHOD_INFORMATION C3
33: CLASS_INTEGRATION_AND_TEST C3 PRC3 TESTER2
34: CHECKED_ATTRIBUTE_INFORMATION C5
35: CHECKED_METHOD_INFORMATION C5
36: NEW_SPRINT
37: INCREASE_PRIORITY
38: INCREASE_PRIORITY
39: INCREASE_PRIORITY
40: INCREASE_PRIORITY
41: CLASS_INTEGRATION_AND_TEST C5 PRC5 TESTER1
42: INCREASE_PRIORITY
43: INCREASE_PRIORITY
44: INCREASE_PRIORITY
45: INCREASE_PRIORITY
46: INCREASE_PRIORITY
47: CHECKED_ATTRIBUTE_INFORMATION C6
48: CHECKED_METHOD_INFORMATION C6
49: CLASS_INTEGRATION_AND_TEST C6 PRC6 TESTER2
50: INCREASE_PRIORITY
51: INCREASE_PRIORITY
52: INCREASE_PRIORITY
53: INCREASE_PRIORITY
54: INCREASE_PRIORITY
55: INCREASE_PRIORITY
56: CHECKED_ATTRIBUTE_INFORMATION C7
```

```
57: CHECKED_METHOD_INFORMATION C7
58: CLASS_INTEGRATION_AND_TEST C7 PRC7 TESTER1
59: INCREASE_PRIORITY
60: INCREASE_PRIORITY
61: INCREASE_PRIORITY
62: INCREASE_PRIORITY
63: INCREASE_PRIORITY
64: INCREASE_PRIORITY
65: INCREASE_PRIORITY
66: CHECKED_ATTRIBUTE_INFORMATION C8
67: CHECKED_METHOD_INFORMATION C8
68: CLASS_INTEGRATION_AND_TEST C8 PRC8 TESTER2
```

APÊNDICE E – SÍNTESE DOS ESTUDOS DE VIABILIDADE

A seguir, são apresentados os resultados dos estudos de viabilidade do Capítulo 5 e os esquemas dos planos de integração decorrentes dos planos de IA gerados nesses estudos.

E.1 SÍNTESE DO ESTUDO 1

Tabela E.1: Resultados do Estudo 1.

Cenário	Características	Tempo	Estados	Ações	Plano de IA
C1	Um testador	0,006s	50	35	0: INTEGRATION_AND_TEST PROC1 PR1 T1 2: INTEGRATION_AND_TEST PROC2 PR2 T1 4: INTEGRATION_AND_TEST PROC3 PR3 T1 6: INTEGRATION_AND_TEST PROC4 PR4 T1 7: NEW_SPRINT 12: INTEGRATION_AND_TEST PROC5 PR5 T1 14: INTEGRATION_AND_TEST PROC6 PR6 T1 16: INTEGRATION_AND_TEST PROC7 PR7 T1 18: INTEGRATION_AND_TEST PROC8 PR8 T1 19: NEW_SPRINT 28: INTEGRATION_AND_TEST PROC9 PR9 T1 30: INTEGRATION_AND_TEST PROC10 PR10 T1 32: INTEGRATION_AND_TEST PROC11 PR11 T1 34: INTEGRATION_AND_TEST PROC12 PR12 T1
C2	Um testador	0,004s	31	18	0: INTEGRATION_AND_TEST PROC1 PR1 T1 1: INTEGRATION_AND_TEST PROC2 PR1 T1 2: INTEGRATION_AND_TEST PROC3 PR1 T1 3: INTEGRATION_AND_TEST PROC4 PR1 T1 4: NEW_SPRINT 5: INTEGRATION_AND_TEST PROC5 PR1 T1 6: INTEGRATION_AND_TEST PROC6 PR1 T1 7: INTEGRATION_AND_TEST PROC7 PR1 T1 8: NEW_SPRINT 9: INTEGRATION_AND_TEST PROC8 PR1 T1 10: INTEGRATION_AND_TEST PROC9 PR1 T1 13: NEW_SPRINT 14: INTEGRATION_AND_TEST PROC10 PR1 T1 15: INTEGRATION_AND_TEST PROC11 PR1 T1 16: NEW_SPRINT 17: INTEGRATION_AND_TEST PROC12 PR1 T1
C2	Dois testadores	0,007s	30	16	0: INTEGRATION_AND_TEST PROC1 PR1 T1 1: INTEGRATION_AND_TEST PROC2 PR1 T2 2: INTEGRATION_AND_TEST PROC3 PR1 T1 3: INTEGRATION_AND_TEST PROC4 PR1 T2 4: INTEGRATION_AND_TEST PROC5 PR1 T1 5: INTEGRATION_AND_TEST PROC7 PR1 T2 6: INTEGRATION_AND_TEST PROC6 PR1 T1 7: NEW_SPRINT 8: INTEGRATION_AND_TEST PROC8 PR1 T1 9: INTEGRATION_AND_TEST PROC9 PR1 T2 10: INTEGRATION_AND_TEST PROC10 PR1 T1 11: INTEGRATION_AND_TEST PROC11 PR1 T2 14: NEW_SPRINT 15: INTEGRATION_AND_TEST PROC12 PR1 T1
C3	Quatro testadores Sem balanceamento da atribuição	0,004s	13	12	0: INTEGRATION_AND_TEST PROC1 PR1 T1 1: INTEGRATION_AND_TEST PROC2 PR1 T1 2: INTEGRATION_AND_TEST PROC3 PR1 T1 3: INTEGRATION_AND_TEST PROC4 PR1 T1 4: INTEGRATION_AND_TEST PROC5 PR1 T1 5: INTEGRATION_AND_TEST PROC6 PR1 T1 6: INTEGRATION_AND_TEST PROC7 PR1 T1 7: INTEGRATION_AND_TEST PROC8 PR1 T1 8: INTEGRATION_AND_TEST PROC9 PR1 T1 9: INTEGRATION_AND_TEST PROC10 PR1 T1 10: INTEGRATION_AND_TEST PROC11 PR1 T1 11: INTEGRATION_AND_TEST PROC12 PR1 T1
C3	Quatro testadores Com balanceamento da atribuição	0,006s	13	12	0: INTEGRATION_AND_TEST PROC1 PR1 T1 1: INTEGRATION_AND_TEST PROC2 PR1 T2 2: INTEGRATION_AND_TEST PROC3 PR1 T3 3: INTEGRATION_AND_TEST PROC4 PR1 T4 4: INTEGRATION_AND_TEST PROC5 PR1 T1 5: INTEGRATION_AND_TEST PROC6 PR1 T2 6: INTEGRATION_AND_TEST PROC7 PR1 T3 7: INTEGRATION_AND_TEST PROC8 PR1 T4 8: INTEGRATION_AND_TEST PROC9 PR1 T1 9: INTEGRATION_AND_TEST PROC10 PR1 T2 10: INTEGRATION_AND_TEST PROC11 PR1 T3 11: INTEGRATION_AND_TEST PROC12 PR1 T4

Continua na próxima página

Tabela E.1: Resultados do Estudo 1 (continuação da página anterior).

Cenário	Características	Tempo	Estados	Ações	Plano de IA
C4	Um testador Checagem dos procedimentos chamados	0,005s	23	15	0: INTEGRATION_AND_TEST PROC7 PR1 T1 1: INTEGRATION_AND_TEST PROC10 PR1 T1 2: INTEGRATION_AND_TEST PROC6 PR1 T1 3: NEW_SPRINT 4: INTEGRATION_AND_TEST PROC5 PR1 T1 5: INTEGRATION_AND_TEST PROC12 PR1 T1 6: INTEGRATION_AND_TEST PROC11 PR1 T1 7: NEW_SPRINT 8: INTEGRATION_AND_TEST PROC8 PR1 T1 9: INTEGRATION_AND_TEST PROC3 PR1 T1 10: INTEGRATION_AND_TEST PROC2 PR1 T1 11: NEW_SPRINT 12: INTEGRATION_AND_TEST PROC9 PR1 T1 13: INTEGRATION_AND_TEST PROC4 PR1 T1 14: INTEGRATION_AND_TEST PROC1 PR1 T1
C4	Dois testadores Checagem dos procedimentos chamados	0,009s	37	19	0: INTEGRATION_AND_TEST PROC12 PR1 T1 1: INTEGRATION_AND_TEST PROC11 PR1 T2 2: INTEGRATION_AND_TEST PROC10 PR1 T1 3: INTEGRATION_AND_TEST PROC7 PR1 T2 5: INTEGRATION_AND_TEST PROC3 PR1 T2 6: INTEGRATION_AND_TEST PROC6 PR1 T1 7: NEW_SPRINT 8: INTEGRATION_AND_TEST PROC8 PR1 T1 9: INTEGRATION_AND_TEST PROC9 PR1 T2 10: INTEGRATION_AND_TEST PROC5 PR1 T1 12: INTEGRATION_AND_TEST PROC2 PR1 T1 15: NEW_SPRINT 16: INTEGRATION_AND_TEST PROC4 PR1 T1 18: INTEGRATION_AND_TEST PROC1 PR1 T1
C4	Um testador Checagem dos procedimentos que chamam	0,005s	24	15	0: INTEGRATION_AND_TEST PROC1 PR1 T1 1: INTEGRATION_AND_TEST PROC2 PR1 T1 2: INTEGRATION_AND_TEST PROC3 PR1 T1 3: NEW_SPRINT 4: INTEGRATION_AND_TEST PROC6 PR1 T1 5: INTEGRATION_AND_TEST PROC10 PR1 T1 6: INTEGRATION_AND_TEST PROC4 PR1 T1 7: NEW_SPRINT 8: INTEGRATION_AND_TEST PROC9 PR1 T1 9: INTEGRATION_AND_TEST PROC8 PR1 T1 10: INTEGRATION_AND_TEST PROC12 PR1 T1 11: NEW_SPRINT 12: INTEGRATION_AND_TEST PROC11 PR1 T1 13: INTEGRATION_AND_TEST PROC7 PR1 T1 14: INTEGRATION_AND_TEST PROC5 PR1 T1
C4	Dois testadores Checagem dos procedimentos que chamam	0,008s	42	18	0: INTEGRATION_AND_TEST PROC1 PR1 T1 2: INTEGRATION_AND_TEST PROC2 PR1 T1 4: INTEGRATION_AND_TEST PROC3 PR1 T1 7: NEW_SPRINT 8: INTEGRATION_AND_TEST PROC6 PR1 T1 9: INTEGRATION_AND_TEST PROC4 PR1 T2 10: INTEGRATION_AND_TEST PROC10 PR1 T1 11: INTEGRATION_AND_TEST PROC8 PR1 T2 12: INTEGRATION_AND_TEST PROC7 PR1 T1 13: INTEGRATION_AND_TEST PROC12 PR1 T2 14: NEW_SPRINT 15: INTEGRATION_AND_TEST PROC11 PR1 T1 16: INTEGRATION_AND_TEST PROC9 PR1 T2 17: INTEGRATION_AND_TEST PROC5 PR1 T1

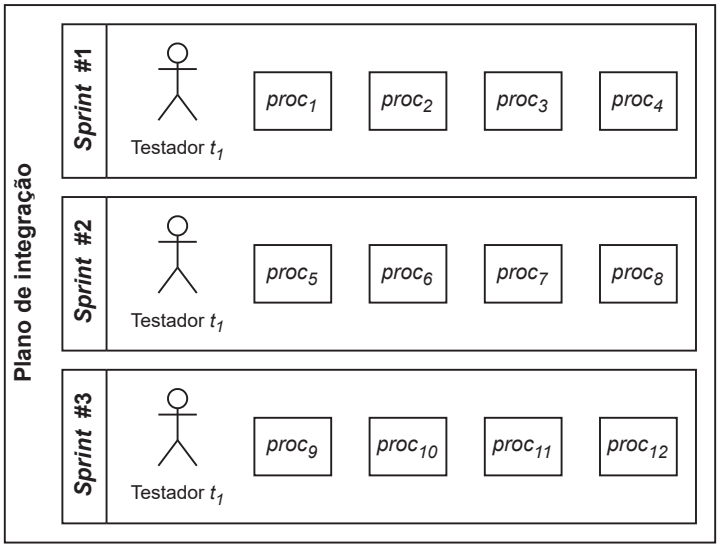
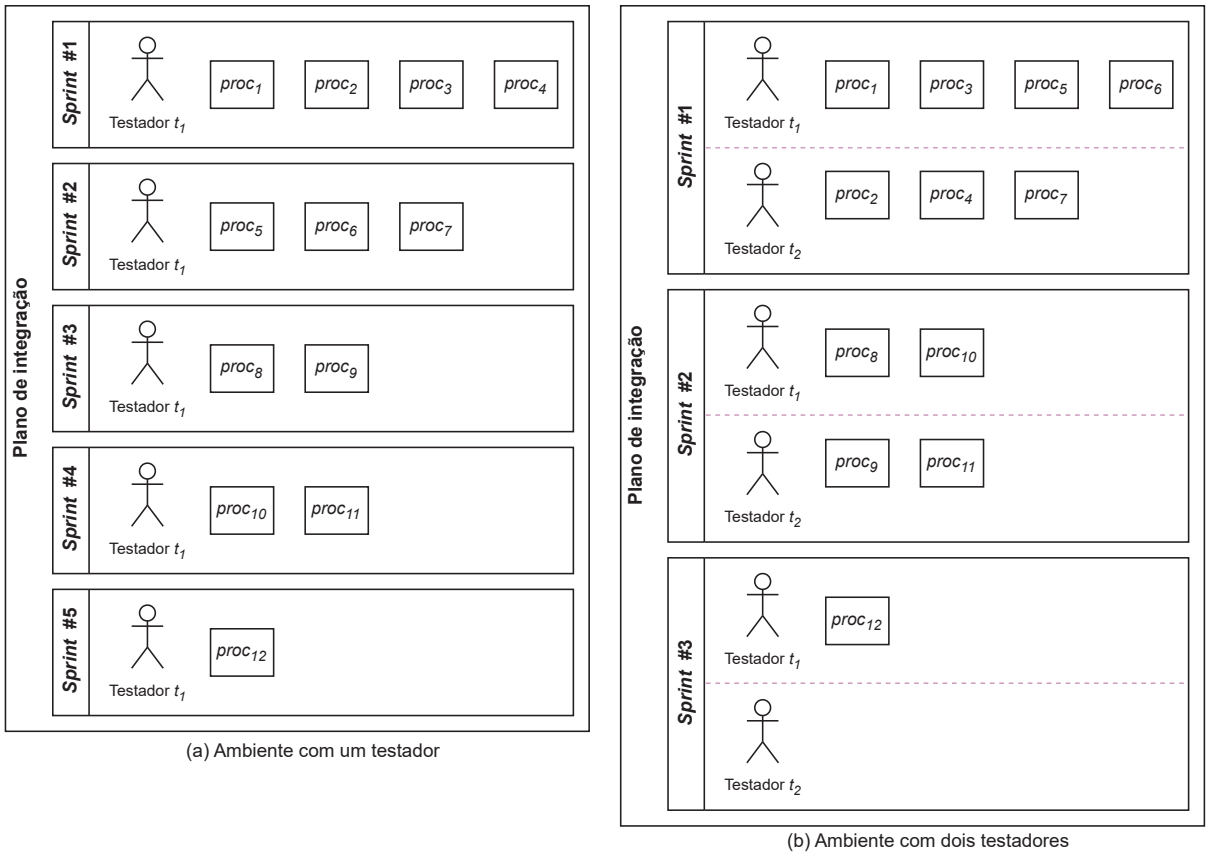


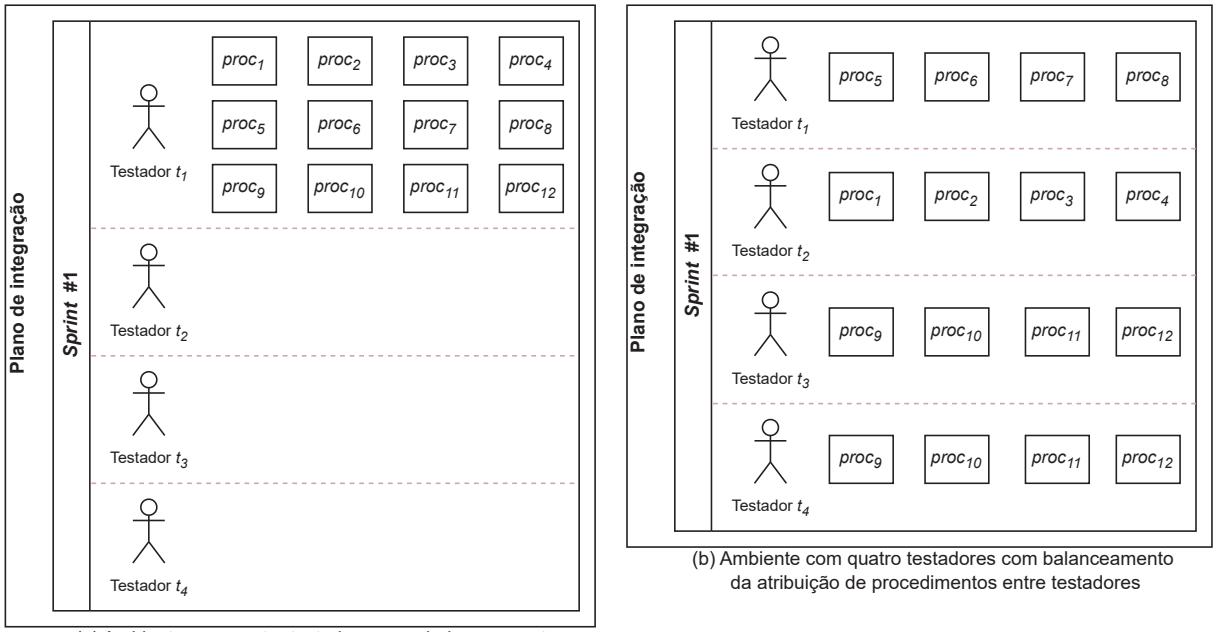
Figura E.1: Cenário de teste 1 do Estudo 1: plano de integração.



(a) Ambiente com um testador

(b) Ambiente com dois testadores

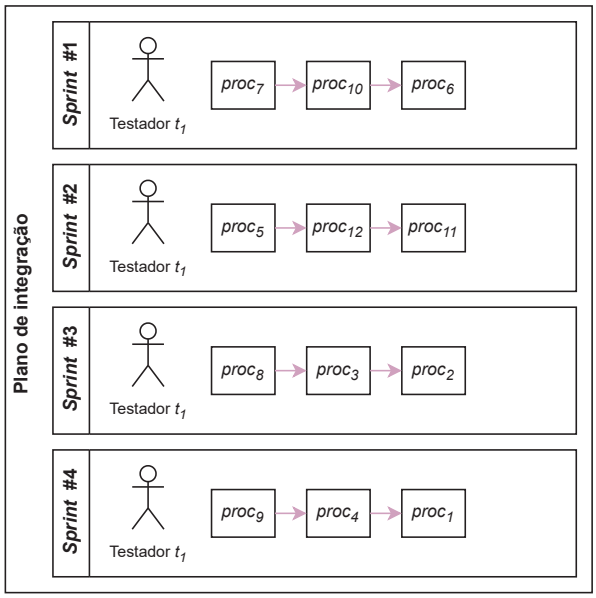
Figura E.2: Cenário de teste 2 do Estudo 1: planos de integração.



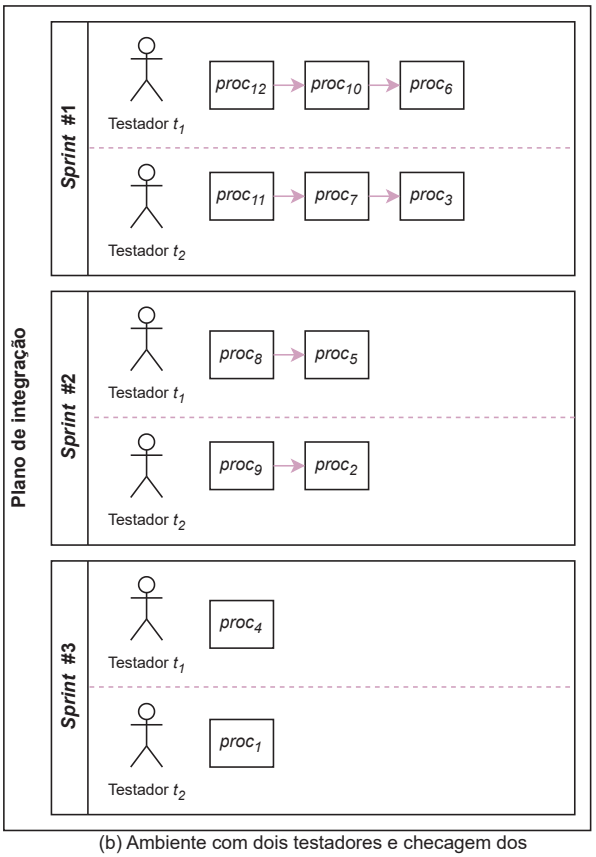
(a) Ambiente com quatro testadores sem balanceamento da atribuição de procedimentos entre testadores

(b) Ambiente com quatro testadores com balanceamento da atribuição de procedimentos entre testadores

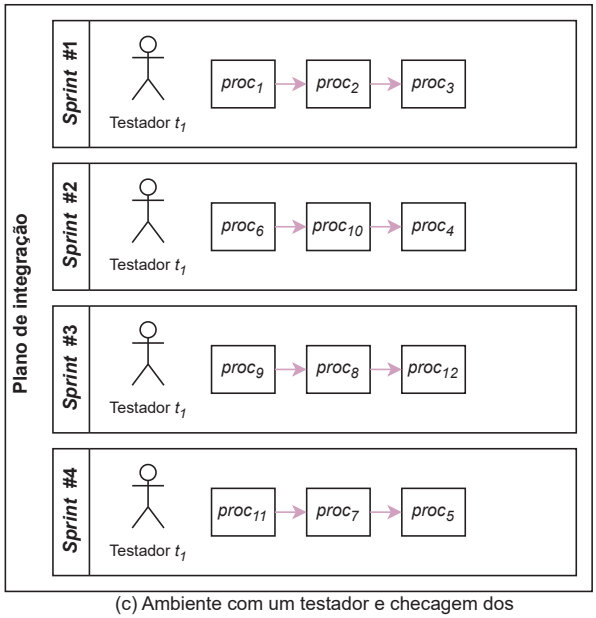
Figura E.3: Cenário de teste 3 do Estudo 1: planos de integração.



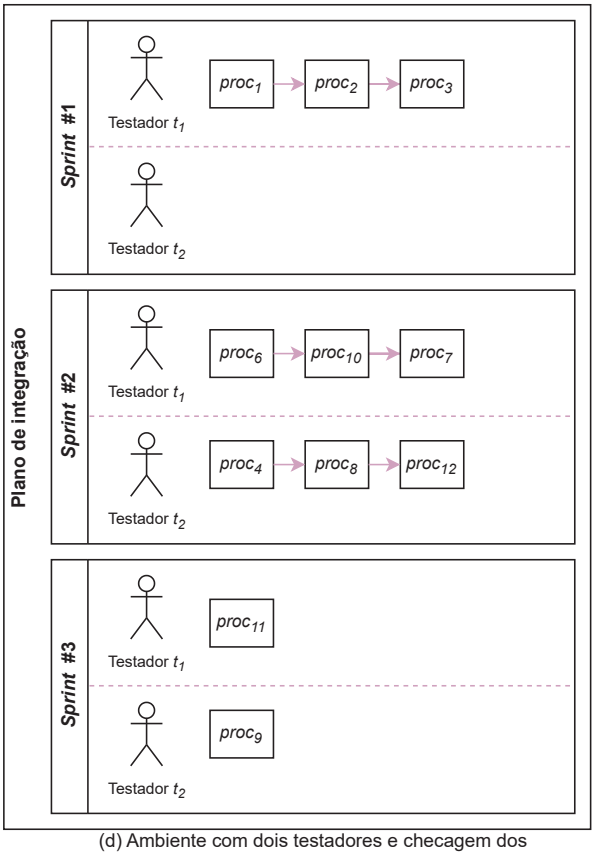
(a) Ambiente com um testador e checagem dos procedimentos chamados pelo procedimento em teste



(b) Ambiente com dois testadores e checagem dos procedimentos chamados pelo procedimento em teste



(c) Ambiente com um testador e checagem dos procedimentos que chamam o procedimento em teste



(d) Ambiente com dois testadores e checagem dos procedimentos que chamam o procedimento em teste

Figura E.4: Cenário de teste 4 do Estudo 1: planos de integração.

E.2 SÍNTESE DO ESTUDO 2

Tabela E.2: Resultados do Estudo 2.

Cenário	Características	Tempo	Estados	Ações	Plano de IA
C1	Teste procedimental Estratégia *top-down*	0m40,432s	382645	126	0: PROCEDURE_PARAMETER PROC8 PARTIDA STRUCT 1: PROCEDURE_PARAMETER PROC7 PARTIDA STRUCT 2: PROCEDURE_PARAMETER PROC6 PARTIDA STRUCT 3: PROCEDURE_PARAMETER PROC5 PARTIDA STRUCT 4: PROCEDURE_PARAMETER PROC14 NAVE STRUCT 5: PROCEDURE_PARAMETER PROC4 PARTIDA STRUCT 16: PROCEDURE_PARAMETER PROC16 COMBUSTIVEL STRUCT 17: PROCEDURE_PARAMETER PROC15 INIMIGO STRUCT 21: INTEGRATION_AND_TEST PROC1 PR1 T2 R_VOID L1 22: PROCEDURE_PARAMETER PROC10 X INT 23: PROCEDURE_PARAMETER PROC10 Y INT 24: PROCEDURE_PARAMETER PROC21 COMBUSTIVEL STRUCT 25: PROCEDURE_PARAMETER PROC21 PROJETIL STRUCT 26: PROCEDURE_PARAMETER PROC20 NAVE STRUCT 27: PROCEDURE_PARAMETER PROC20 COMBUSTIVEL STRUCT 28: PROCEDURE_PARAMETER PROC19 INIMIGO STRUCT 29: PROCEDURE_PARAMETER PROC19 PROJETIL STRUCT 30: PROCEDURE_PARAMETER PROC18 NAVE STRUCT 31: PROCEDURE_PARAMETER PROC18 INIMIGO STRUCT 32: PROCEDURE_PARAMETER PROC17 PROJETIL STRUCT 43: PROCEDURE_PARAMETER PROC9 PARTIDA STRUCT 45: INTEGRATION_AND_TEST PROC6 PR4 T1 R_VOID L2 49: INTEGRATION_AND_TEST PROC7 PR4 T2 R_VOID L2 51: INTEGRATION_AND_TEST PROC3 PR2 T1 R_STRUCT L2 52: NEW_SPRINT 55: INTEGRATION_AND_TEST PROC5 PR3 T1 R_VOID L2 56: INTEGRATION_AND_TEST PROC4 PR1 T2 R_VOID L2 62: NEW_SPRINT 66: INTEGRATION_AND_TEST PROC2 PR4 T1 R_VOID L2 70: INTEGRATION_AND_TEST PROC8 PR4 T2 R_VOID L2 74: INTEGRATION_AND_TEST PROC9 PR4 T1 R_VOID L2 76: INTEGRATION_AND_TEST PROC15 PR1 T2 R_VOID L3 78: INTEGRATION_AND_TEST PROC16 PR1 T2 R_VOID L3 84: NEW_SPRINT 85: INTEGRATION_AND_TEST PROC14 PR1 T1 R_VOID L3 86: INTEGRATION_AND_TEST PROC18 PR1 T2 R_INT L3 87: INTEGRATION_AND_TEST PROC17 PR1 T1 R_VOID L3 88: INTEGRATION_AND_TEST PROC19 PR1 T2 R_INT L3 95: NEW_SPRINT 96: INTEGRATION_AND_TEST PROC20 PR1 T1 R_INT L3 98: INTEGRATION_AND_TEST PROC10 PR2 T2 R_STRUCT L3 99: INTEGRATION_AND_TEST PROC21 PR1 T1 R_INT L3 100: PROCEDURE_PARAMETER PROC11 X INT 101: PROCEDURE_PARAMETER PROC11 Y INT 105: INTEGRATION_AND_TEST PROC23 PR3 T2 R_VOID L3 111: NEW_SPRINT 114: INTEGRATION_AND_TEST PROC11 PR2 T1 R_STRUCT L3 115: PROCEDURE_PARAMETER PROC13 X INT 116: PROCEDURE_PARAMETER PROC13 Y INT 119: INTEGRATION_AND_TEST PROC13 PR2 T2 R_STRUCT L3 120: PROCEDURE_PARAMETER PROC12 X INT 121: PROCEDURE_PARAMETER PROC12 Y INT 124: INTEGRATION_AND_TEST PROC12 PR2 T1 R_STRUCT L3 126: PROCEDURE_PARAMETER PROC22 LINHAS INT 127: PROCEDURE_PARAMETER PROC22 COLUNAS INT 130: INTEGRATION_AND_TEST PROC22 PR3 T2 R_VOID L3

Continua na próxima página

Tabela E.2: Resultados do Estudo 2 (continuação da página anterior).

Cenário	Características	Tempo	Estados	Ações	Plano de IA
C2	Teste procedimental Estratégia *bottom-up*	0m56,229s	531311	137	0: PROCEDURE_PARAMETER PROC10 X INT 1: PROCEDURE_PARAMETER PROC10 Y INT 2: PROCEDURE_PARAMETER PROC21 COMBUSTIVEL STRUCT 3: PROCEDURE_PARAMETER PROC21 PROJETIL STRUCT 4: PROCEDURE_PARAMETER PROC20 NAVE STRUCT 5: PROCEDURE_PARAMETER PROC20 COMBUSTIVEL STRUCT 6: PROCEDURE_PARAMETER PROC19 INIMIGO STRUCT 7: PROCEDURE_PARAMETER PROC19 PROJETIL STRUCT 8: PROCEDURE_PARAMETER PROC18 NAVE STRUCT 9: PROCEDURE_PARAMETER PROC18 INIMIGO STRUCT 10: PROCEDURE_PARAMETER PROC17 PROJETIL STRUCT 11: PROCEDURE_PARAMETER PROC16 COMBUSTIVEL STRUCT 12: PROCEDURE_PARAMETER PROC15 INIMIGO STRUCT 13: PROCEDURE_PARAMETER PROC14 NAVE STRUCT 23: INTEGRATION_AND_TEST PROC10 PR2 T1 R_STRUCT L3 24: INTEGRATION_AND_TEST PROC15 PR1 T2 R_VOID L3 25: INTEGRATION_AND_TEST PROC16 PR1 T1 R_VOID L3 27: INTEGRATION_AND_TEST PROC18 PR1 T2 R_INT L3 29: INTEGRATION_AND_TEST PROC19 PR1 T2 R_INT L3 35: NEW_SPRINT 36: INTEGRATION_AND_TEST PROC14 PR1 T1 R_VOID L3 37: INTEGRATION_AND_TEST PROC17 PR1 T2 R_VOID L3 42: INTEGRATION_AND_TEST PROC23 PR3 T2 R_VOID L3 44: INTEGRATION_AND_TEST PROC20 PR1 T2 R_INT L3 45: PROCEDURE_PARAMETER PROC13 X INT 46: PROCEDURE_PARAMETER PROC13 Y INT 47: PROCEDURE_PARAMETER PROC11 X INT 48: PROCEDURE_PARAMETER PROC11 Y INT 56: NEW_SPRINT 58: INTEGRATION_AND_TEST PROC13 PR2 T1 R_STRUCT L3 60: INTEGRATION_AND_TEST PROC11 PR2 T2 R_STRUCT L3 61: PROCEDURE_PARAMETER PROC12 X INT 62: PROCEDURE_PARAMETER PROC12 Y INT 65: INTEGRATION_AND_TEST PROC12 PR2 T1 R_STRUCT L3 67: PROCEDURE_PARAMETER PROC22 LINHAS INT 68: PROCEDURE_PARAMETER PROC22 COLUNAS INT 69: PROCEDURE_PARAMETER PROC4 PARTIDA STRUCT 72: INTEGRATION_AND_TEST PROC22 PR3 T2 R_VOID L3 74: NEW_SPRINT 75: INTEGRATION_AND_TEST PROC21 PR1 T1 R_INT L3 84: INTEGRATION_AND_TEST PROC2 PR4 T1 R_VOID L2 88: PROCEDURE_PARAMETER PROC6 PARTIDA STRUCT 92: INTEGRATION_AND_TEST PROC6 PR4 T1 R_VOID L2 93: INTEGRATION_AND_TEST PROC4 PR1 T2 R_VOID L2 94: NEW_SPRINT 96: INTEGRATION_AND_TEST PROC3 PR2 T1 R_STRUCT L2 97: PROCEDURE_PARAMETER PROC8 PARTIDA STRUCT 98: PROCEDURE_PARAMETER PROC7 PARTIDA STRUCT 100: PROCEDURE_PARAMETER PROC9 PARTIDA STRUCT 101: PROCEDURE_PARAMETER PROC5 PARTIDA STRUCT 107: INTEGRATION_AND_TEST PROC8 PR4 T2 R_VOID L2 111: INTEGRATION_AND_TEST PROC7 PR4 T1 R_VOID L2 115: INTEGRATION_AND_TEST PROC9 PR4 T2 R_VOID L2 117: NEW_SPRINT 120: INTEGRATION_AND_TEST PROC5 PR3 T1 R_VOID L2 123: INTEGRATION_AND_TEST PROC1 PR1 T2 R_VOID L1

Continua na próxima página

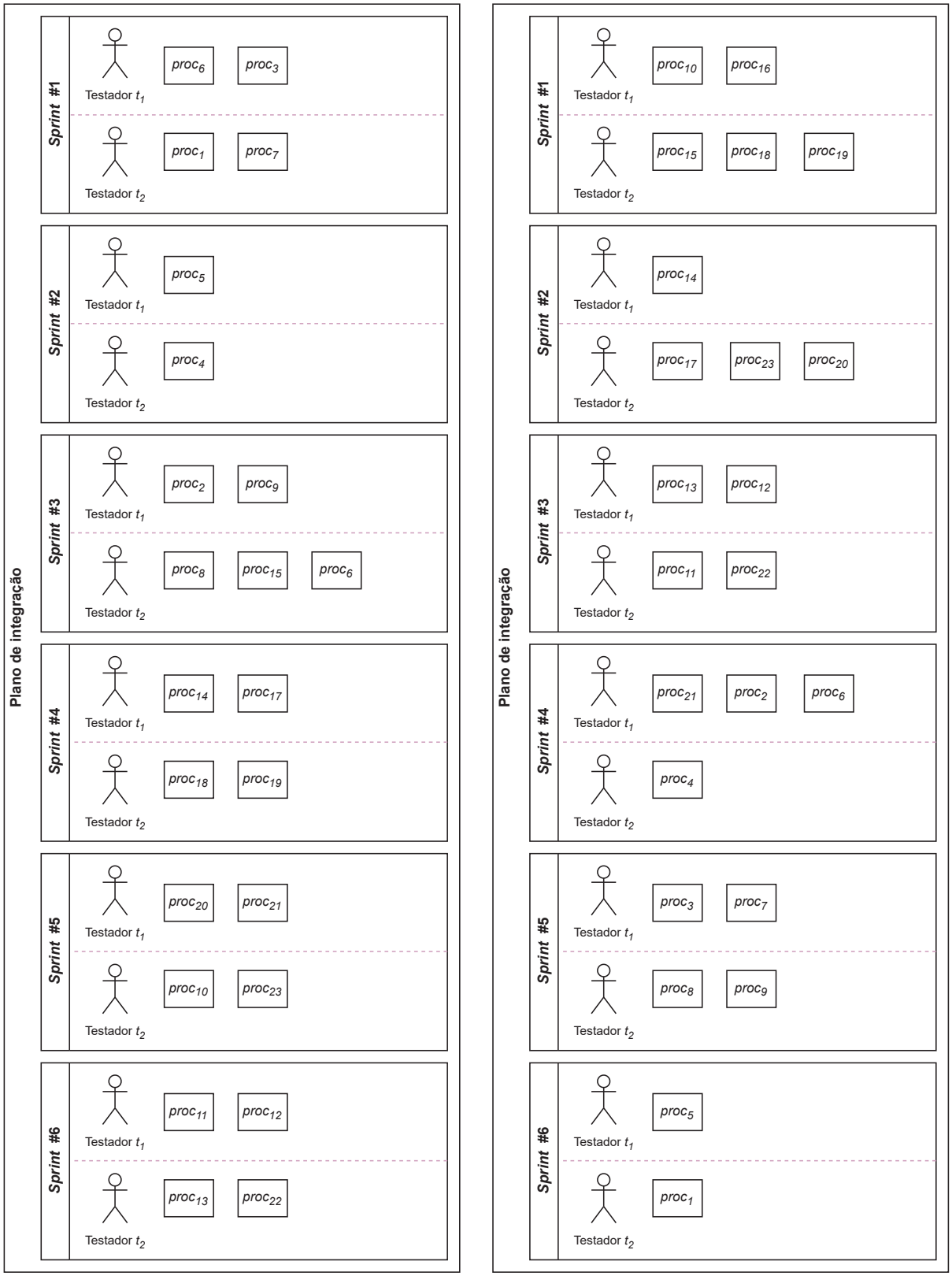
Tabela E.2: Resultados do Estudo 2 (continuação da página anterior).

Cenário	Características	Tempo	Estados	Ações	Plano de IA
C3	Teste OO Teste intermétodos	1m56,945s	393592	167	0: METHOD_ARGUMENT M10 NEWPOST STRING 1: METHOD_ARGUMENT M10 P PERSON 2: METHOD_ARGUMENT M7 NEWPOST STRING 3: METHOD_ARGUMENT M7 TG GROUP 4: METHOD_ARGUMENT M9 NEWPOST STRING 5: METHOD_ARGUMENT M6 PPOST STRING 6: METHOD_ARGUMENT M6 TG GROUP 7: METHOD_ARGUMENT M3 GROUPNAME STRING 8: METHOD_ARGUMENT M1 ARG STRING 9: METHOD_ARGUMENT M11 TP PERSON 10: METHOD_ARGUMENT M8 TP PERSON 11: METHOD_ARGUMENT M8 LIKE INT 12: METHOD_ARGUMENT M5 GROUPLOGIN STRING 26: METHOD_INTEGRATION_AND_TEST M1 PRM1 T2 C1 RTM1 28: METHOD_INTEGRATION_AND_TEST M2 PRM1 T1 C2 RTM1 31: METHOD_INTEGRATION_AND_TEST M4 PRM3 T2 C2 RTM1 33: METHOD_INTEGRATION_AND_TEST M3 PRM2 T1 C2 RTM1 34: NEW_SPRINT 37: METHOD_INTEGRATION_AND_TEST M7 PRM3 T1 C2 RTM1 39: METHOD_INTEGRATION_AND_TEST M6 PRM2 T2 C2 RTM1 41: METHOD_ARGUMENT M14 NEWPOST STRING 42: METHOD_ARGUMENT M14 P PERSON 44: METHOD_INTEGRATION_AND_TEST M10 PRM3 T1 C2 RTM1 45: METHOD_ARGUMENT M15 NEWPOST STRING 46: METHOD_ARGUMENT M12 TP PERSON 47: METHOD_INTEGRATION_AND_TEST M5 PRM1 T2 C2 RTM1 50: NEW_SPRINT 52: METHOD_INTEGRATION_AND_TEST M8 PRM1 T1 C2 RTM1 53: METHOD_ARGUMENT M13 TG GROUP 56: METHOD_INTEGRATION_AND_TEST M9 PRM2 T2 C2 RTM1 59: METHOD_INTEGRATION_AND_TEST M12 PRM2 T1 C3 RTM1 61: METHOD_INTEGRATION_AND_TEST M15 PRM2 T2 C3 RTM1 62: NEW_SPRINT 64: METHOD_ARGUMENT M17 TP PERSON 65: METHOD_ARGUMENT M17 LIKE INT 66: METHOD_ARGUMENT M19 TP PERSON 67: METHOD_ARGUMENT M19 NEWPOST STRING 68: METHOD_ARGUMENT M16 TP PERSON 69: METHOD_ARGUMENT M16 NEWCOMMENT STRING 71: METHOD_INTEGRATION_AND_TEST M13 PRM3 T1 C3 RTM1 77: METHOD_ARGUMENT M18 TP PERSON 78: METHOD_ARGUMENT M18 PPOST STRING 79: METHOD_ARGUMENT M20 OBSERVER OBSERVER 80: METHOD_INTEGRATION_AND_TEST M11 PRM1 T1 C3 RTM1 81: METHOD_INTEGRATION_AND_TEST M14 PRM1 T2 C3 RTM1 88: METHOD_INTEGRATION_AND_TEST M18 PRM3 T2 C4 RTM1 89: NEW_SPRINT 91: METHOD_INTEGRATION_AND_TEST M17 PRM2 T1 C4 RTM1 93: METHOD_INTEGRATION_AND_TEST M20 PRM2 T2 C4 RTM1 95: METHOD_ARGUMENT M21 OBSERVER OBSERVER 98: METHOD_INTEGRATION_AND_TEST M21 PRM3 T1 C4 RTM1 99: METHOD_ARGUMENT M25 PERSONNAME STRING 100: METHOD_INTEGRATION_AND_TEST M16 PRM1 T2 C4 RTM1 103: NEW_SPRINT 104: METHOD_INTEGRATION_AND_TEST M19 PRM1 T1 C4 RTM1 106: METHOD_ARGUMENT M23 PERSONLOGIN STRING 109: METHOD_INTEGRATION_AND_TEST M23 PRM2 T2 C5 RTM1 110: METHOD_ARGUMENT M26 NEWPOST STRING 111: METHOD_ARGUMENT M26 P PERSON 114: METHOD_INTEGRATION_AND_TEST M26 PRM2 T1 C5 RTM1 115: METHOD_ARGUMENT M27 NEWPOST STRING 119: METHOD_INTEGRATION_AND_TEST M27 PRM3 T2 C5 RTM1 120: NEW_SPRINT 121: METHOD_INTEGRATION_AND_TEST M22 PRM1 T1 C5 RTM1 123: METHOD_ARGUMENT M28 TP PERSON 126: METHOD_INTEGRATION_AND_TEST M24 PRM3 T2 C5 RTM1 127: METHOD_ARGUMENT M31 NEWPOST STRING 128: METHOD_ARGUMENT M31 P PERSON 129: METHOD_INTEGRATION_AND_TEST M25 PRM1 T1 C5 RTM1 133: METHOD_INTEGRATION_AND_TEST M28 PRM1 T2 C6 RTM1 134: METHOD_ARGUMENT M32 NEWPOST STRING 136: NEW_SPRINT 138: METHOD_ARGUMENT M29 TP PERSON 140: METHOD_INTEGRATION_AND_TEST M29 PRM2 T1 C6 RTM1 142: METHOD_INTEGRATION_AND_TEST M32 PRM2 T2 C6 RTM1 143: METHOD_INTEGRATION_AND_TEST M31 PRM1 T1 C6 RTM1 144: METHOD_ARGUMENT M30 TG GROUP 145: METHOD_ARGUMENT M33 NEWPOST STRING 146: METHOD_ARGUMENT M33 P PERSON 150: METHOD_INTEGRATION_AND_TEST M30 PRM3 T2 C6 RTM1 153: NEW_SPRINT 154: METHOD_INTEGRATION_AND_TEST M33 PRM1 T1 C7 RTM1 155: METHOD_ARGUMENT M34 NEWPOST STRING 158: METHOD_INTEGRATION_AND_TEST M34 PRM2 T1 C7 RTM1 160: METHOD_ARGUMENT M35 OBSERVER OBSERVER 163: METHOD_INTEGRATION_AND_TEST M35 PRM1 T2 C8 RTM1 164: METHOD_ARGUMENT M37 TP PERSON 165: METHOD_ARGUMENT M37 NEWPOST STRING 166: METHOD_ARGUMENT M36 OBSERVER OBSERVER 171: METHOD_INTEGRATION_AND_TEST M37 PRM3 T2 C8 RTM1 173: NEW_SPRINT 175: METHOD_INTEGRATION_AND_TEST M36 PRM2 T1 C8 RTM1

Continua na próxima página

Tabela E.2: Resultados do Estudo 2 (continuação da página anterior).

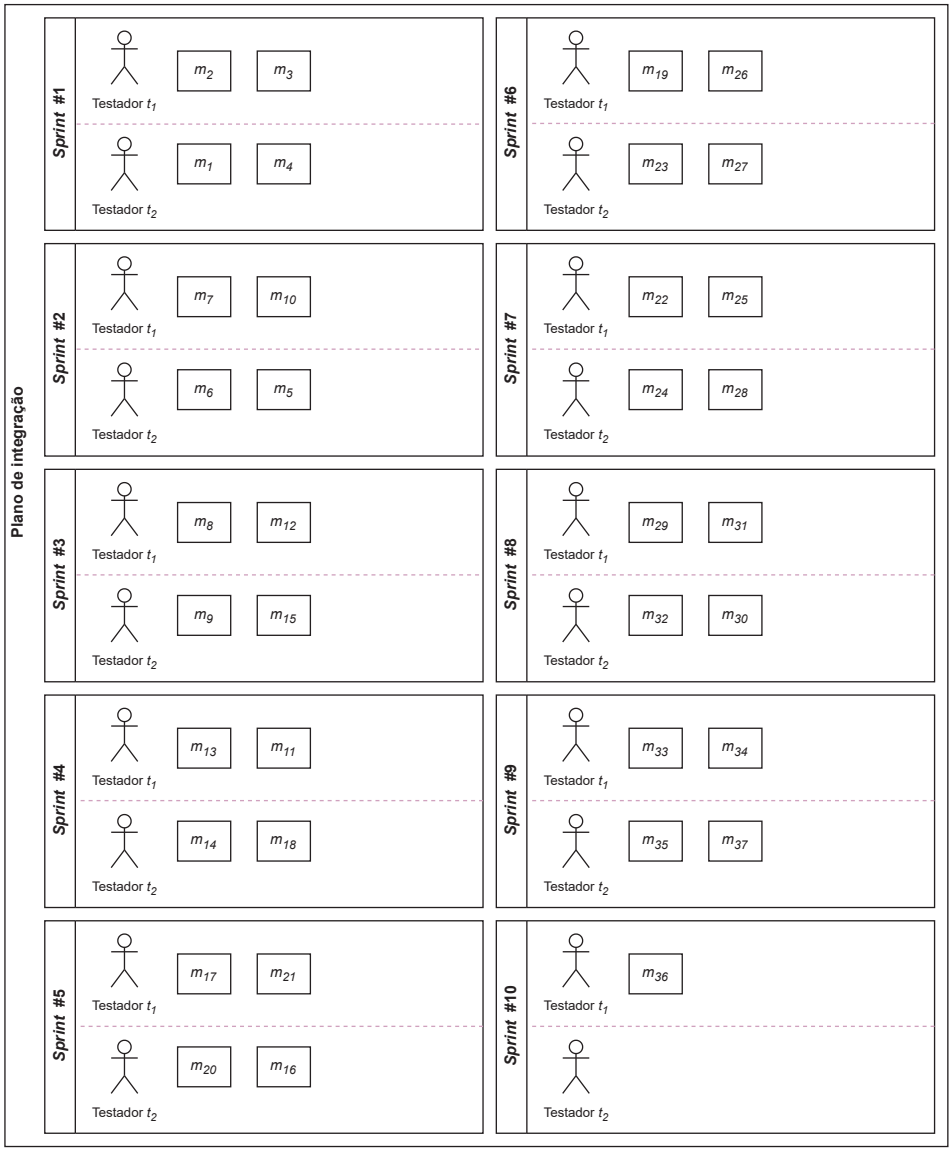
Cenário	Características	Tempo	Estados	Ações	Plano de IA
C4	Teste OO Teste interclasses	0m11,791s	197634	55	0: CLASS_ATTRIBUTE C3 OBSERVERS ARRAYLIST 1: CLASS_METHOD C3 M11 2: CLASS_METHOD C3 M12 3: CLASS_METHOD C3 M13 4: CLASS_METHOD C3 M14 5: CLASS_METHOD C3 M15 6: CLASS_ATTRIBUTE C4 OBSERVERS ARRAYLIST 7: CLASS_ATTRIBUTE C4 NEWPOST STRING 8: CLASS_ATTRIBUTE C4 NEWCOMMENT STRING 9: CLASS_ATTRIBUTE C4 LIKES INT 10: CLASS_ATTRIBUTE C4 DESLIKES INT 11: CLASS_ATTRIBUTE C4 OWNER STRING 12: CLASS_METHOD C4 M16 13: CLASS_METHOD C4 M17 14: CLASS_METHOD C4 M18 15: CLASS_METHOD C4 M19 16: CLASS_METHOD C4 M20 17: CLASS_METHOD C4 M21 18: CLASS_ATTRIBUTE C2 GROUPNAME STRING 19: CLASS_ATTRIBUTE C2 GROUPLOGIN STRING 20: CLASS_ATTRIBUTE C2 NEWPOST STRING 21: CLASS_ATTRIBUTE C2 LIKES INT 22: CLASS_ATTRIBUTE C2 DESLIKES INT 23: CLASS_ATTRIBUTE C2 OWNER STRING 24: CLASS_METHOD C2 M2 25: CLASS_METHOD C2 M3 26: CLASS_METHOD C2 M4 27: CLASS_METHOD C2 M5 28: CLASS_METHOD C2 M6 29: CLASS_METHOD C2 M7 30: CLASS_METHOD C2 M8 31: CLASS_METHOD C2 M9 32: CLASS_METHOD C2 M10 38: CLASS_INTEGRATION_AND_TEST C3 PRC2 T1 41: CLASS_INTEGRATION_AND_TEST C2 PRC1 T2 42: CLASS_INTEGRATION_AND_TEST C4 PRC1 T1 43: CLASS_ATTRIBUTE C7 ATR1 AT1 44: CLASS_METHOD C7 M33 45: CLASS_METHOD C7 M34 46: CLASS_ATTRIBUTE C5 PERSONNAME STRING 47: CLASS_ATTRIBUTE C5 PERSONLOGIN STRING 48: CLASS_ATTRIBUTE C5 DATEUSER STRING 49: CLASS_METHOD C5 M22 50: CLASS_METHOD C5 M23 51: CLASS_METHOD C5 M24 52: CLASS_METHOD C5 M25 53: CLASS_METHOD C5 M26 54: CLASS_METHOD C5 M27 59: CLASS_INTEGRATION_AND_TEST C7 PRC3 T2 64: CLASS_INTEGRATION_AND_TEST C5 PRC2 T2 67: CLASS_ATTRIBUTE C6 FRIENDS ARRAYLIST 68: CLASS_ATTRIBUTE C6 RELATIONSHIP ARRAYLIST 69: CLASS_ATTRIBUTE C6 AFFINITY ARRAYLIST 70: CLASS_ATTRIBUTE C6 DATEFRIENDSHIP STRING 71: CLASS_METHOD C6 M28 72: CLASS_METHOD C6 M29 73: CLASS_METHOD C6 M30 74: CLASS_METHOD C6 M31 75: CLASS_METHOD C6 M32 78: NEW_SPRINT 82: CLASS_INTEGRATION_AND_TEST C6 PRC2 T1 84: CLASS_ATTRIBUTE C1 ATR1 AT1 85: CLASS_METHOD C1 M1 89: CLASS_INTEGRATION_AND_TEST C1 PRC3 T2 91: CLASS_ATTRIBUTE C8 ATR1 AT1 92: CLASS_METHOD C8 M35 93: CLASS_METHOD C8 M36 94: CLASS_METHOD C8 M37 90: CLASS_INTEGRATION_AND_TEST C8 PRC3 T1



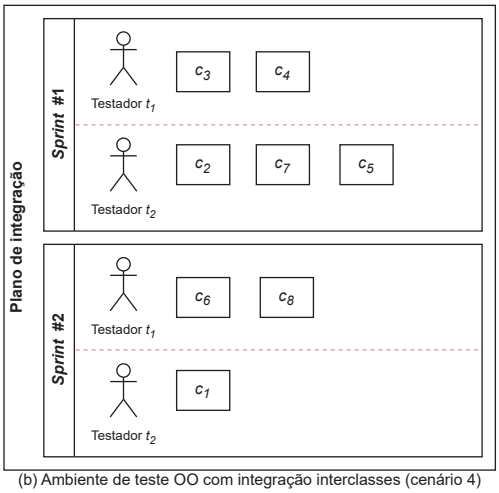
(a) Ambiente de teste procedimental com integração *top-down* (cenário 1)

(b) Ambiente de teste procedimental com integração *bottom-up* (cenário 2)

Figura E.5: Cenários de teste 1 e 2 do Estudo 2: planos de integração.



(a) Ambiente de teste OO com integração intermétodos (cenário 3)



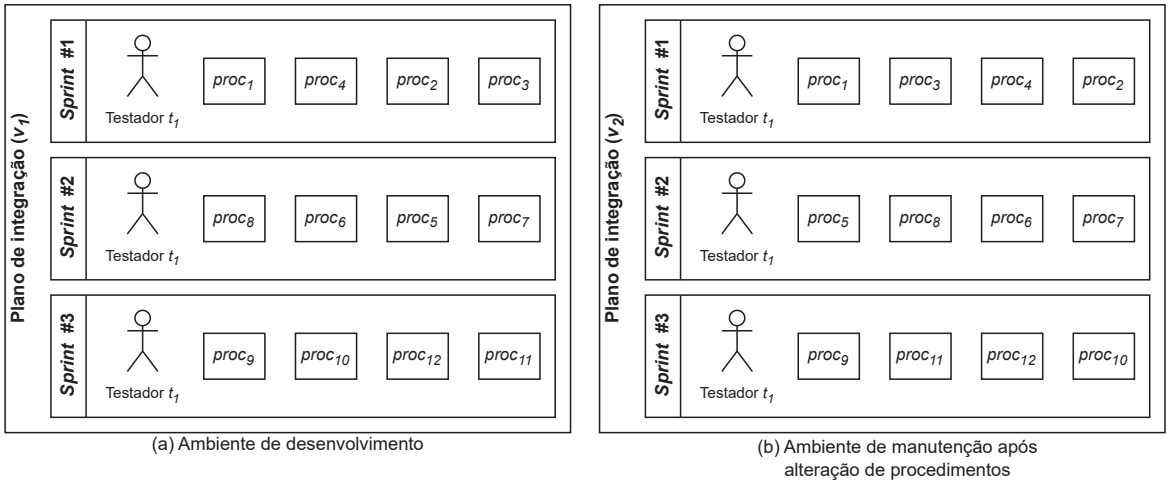
(b) Ambiente de teste OO com integração interclasses (cenário 4)

Figura E.6: Cenários de teste 3 e 4 do Estudo 2: planos de integração.

E.3 SÍNTESE DO ESTUDO 3

Tabela E.3: Resultados do Estudo 3.

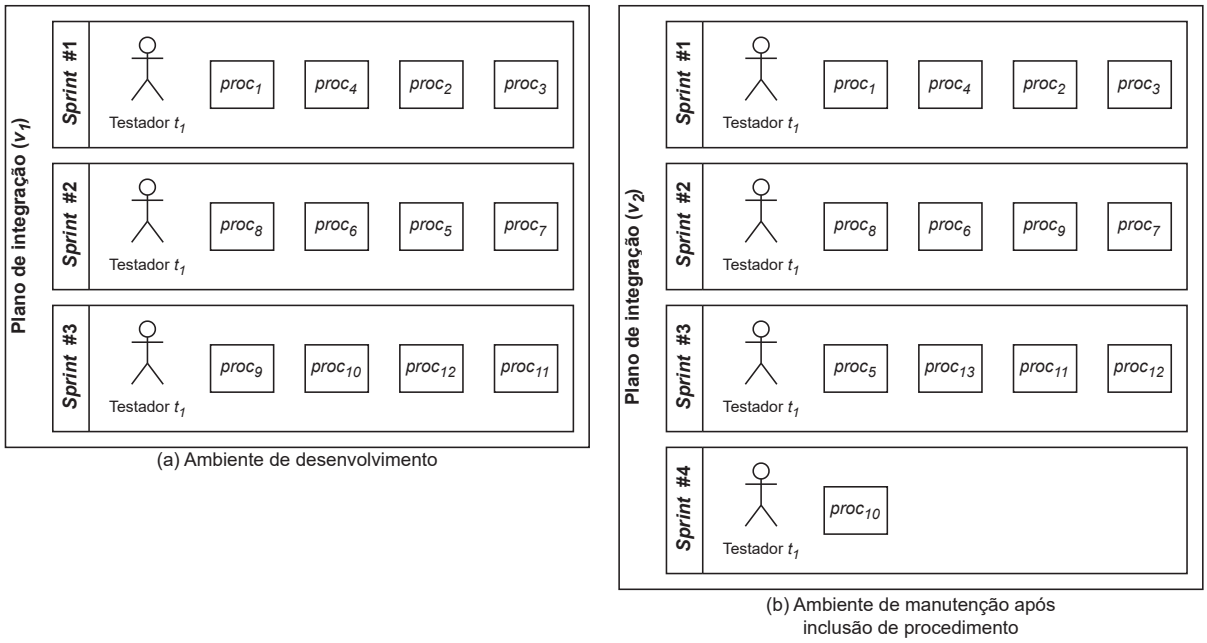
Cenário	Características	Tempo	Estados	Ações	Plano de IA
Cenário geral	Desenvolvimento	0,033s	859	48	10: INTEGRATION_AND_TEST PROC1 PR1 T1 RT1 L1 13: INTEGRATION_AND_TEST PROC4 PR1 T1 RT1 L2 16: INTEGRATION_AND_TEST PROC2 PR2 T1 RT1 L2 19: INTEGRATION_AND_TEST PROC3 PR3 T1 RT1 L2 21: NEW_SPRINT 24: INTEGRATION_AND_TEST PROC8 PR1 T1 RT2 L3 29: INTEGRATION_AND_TEST PROC6 PR2 T1 RT2 L3 31: INTEGRATION_AND_TEST PROC5 PR3 T1 RT2 L3 35: INTEGRATION_AND_TEST PROC7 PR3 T1 RT2 L3 37: NEW_SPRINT 38: INTEGRATION_AND_TEST PROC9 PR3 T1 RT3 L3 41: INTEGRATION_AND_TEST PROC10 PR1 T1 RT3 L4 44: INTEGRATION_AND_TEST PROC12 PR1 T1 RT3 L4 47: INTEGRATION_AND_TEST PROC11 PR1 T1 RT3 L4
C1	Manutenção Alteração de procedimento	0,132s	5344	52	8: INTEGRATION_AND_TEST PROC1 PR1 T1 RT1 L1 15: INTEGRATION_AND_TEST PROC3 PR1 T1 RT1 L2 17: INTEGRATION_AND_TEST PROC4 PR2 T1 RT1 L2 20: INTEGRATION_AND_TEST PROC2 PR3 T1 RT1 L2 23: NEW_SPRINT 28: INTEGRATION_AND_TEST PROC5 PR1 T1 RT2 L3 30: INTEGRATION_AND_TEST PROC8 PR2 T1 RT2 L3 32: INTEGRATION_AND_TEST PROC6 PR3 T1 RT2 L3 39: INTEGRATION_AND_TEST PROC7 PR4 T1 RT2 L3 41: NEW_SPRINT 42: INTEGRATION_AND_TEST PROC9 PR4 T1 RT3 L3 45: INTEGRATION_AND_TEST PROC11 PR1 T1 RT3 L4 48: INTEGRATION_AND_TEST PROC12 PR2 T1 RT3 L4 51: INTEGRATION_AND_TEST PROC10 PR2 T1 RT3 L4
C2	Manutenção Inclusão de procedimento	0,047s	1377	53	10: INTEGRATION_AND_TEST PROC1 PR1 T1 RT1 L1 14: INTEGRATION_AND_TEST PROC4 PR1 T1 RT1 L2 16: INTEGRATION_AND_TEST PROC2 PR2 T1 RT1 L2 18: INTEGRATION_AND_TEST PROC3 PR3 T1 RT1 L2 20: NEW_SPRINT 24: INTEGRATION_AND_TEST PROC8 PR1 T1 RT2 L3 28: INTEGRATION_AND_TEST PROC6 PR2 T1 RT2 L3 30: INTEGRATION_AND_TEST PROC9 PR2 T1 RT3 L3 35: INTEGRATION_AND_TEST PROC7 PR3 T1 RT2 L3 37: NEW_SPRINT 38: INTEGRATION_AND_TEST PROC5 PR3 T1 RT2 L3 41: INTEGRATION_AND_TEST PROC13 PR1 T1 RT3 L4 47: INTEGRATION_AND_TEST PROC11 PR2 T1 RT3 L4 48: INTEGRATION_AND_TEST PROC12 PR2 T1 RT3 L4 51: NEW_SPRINT 52: INTEGRATION_AND_TEST PROC10 PR2 T1 RT3 L4
C3	Manutenção Remoção de procedimento	0,018s	591	43	10: INTEGRATION_AND_TEST PROC1 PR1 T1 RT1 L1 14: INTEGRATION_AND_TEST PROC3 PR1 T1 RT1 L2 16: INTEGRATION_AND_TEST PROC2 PR2 T1 RT1 L2 18: INTEGRATION_AND_TEST PROC4 PR3 T1 RT1 L2 20: NEW_SPRINT 22: INTEGRATION_AND_TEST PROC8 PR1 T1 RT2 L3 25: INTEGRATION_AND_TEST PROC9 PR1 T1 RT3 L3 27: INTEGRATION_AND_TEST PROC6 PR2 T1 RT2 L3 35: INTEGRATION_AND_TEST PROC5 PR3 T1 RT2 L3 37: NEW_SPRINT 38: INTEGRATION_AND_TEST PROC7 PR3 T1 RT2 L3 42: INTEGRATION_AND_TEST PROC10 PR1 T1 RT3 L4



(a) Ambiente de desenvolvimento

(b) Ambiente de manutenção após alteração de procedimentos

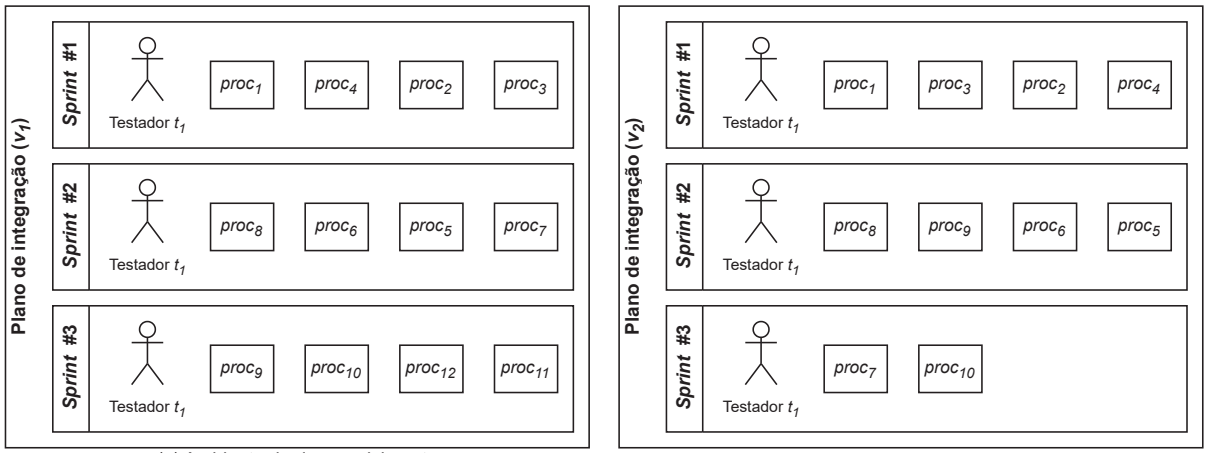
Figura E.7: Cenário de teste 1 do Estudo 3: plano de integração.



(a) Ambiente de desenvolvimento

(b) Ambiente de manutenção após inclusão de procedimento

Figura E.8: Cenário de teste 2 do Estudo 3: plano de integração.



(a) Ambiente de desenvolvimento

(b) Ambiente de manutenção após remoção de procedimentos

Figura E.9: Cenário de teste 3 do Estudo 3: plano de integração.

APÊNDICE F – PUBLICAÇÕES E PRODUÇÕES TÉCNICAS

Esta tese resultou nas seguintes publicações:

1. **Lima e Peres (2023)**: Lima, Luis F. de e Peres, Leticia M. (2023). Teste de integração com planejamento em inteligência artificial. V Fórum dos Programas de Pós-Graduação em Computação do Paraná (V ForPPGC-PR).

2. **Lima (2023)**: Lima, Luis F. de (2023). Uma abordagem de teste de integração com planejamento em inteligência artificial. XXVI Congresso Ibero-Americano em Engenharia de Software (CIbSE 2023). páginas 261-268. DOI: 10.5753/cibse.2023.24710.

Publicações relacionadas ao tema da tese submetidas ou em preparação:

1. *Artificial Intelligence Planning and Software Testing: An Extended Bibliographic Review*.

2. *A Survey AI Planning-based Integration Testing Architectures*.

3. *Software Requirement Allocation: An Automatic Tool Based On Artificial Intelligence Planning*.

4. *Artificial Intelligence Planning Models for Generation of Integration Testing Plans*.

Outras publicações desenvolvidas no âmbito do grupo de pesquisa do Lab FAES publicadas durante o desenvolvimento da tese:

1. **Lima et al. (2020c)**: Lima, Luis F. de, Huve, Cristiane A. G. e Peres, Leticia M. (2020). *Software product quality evaluation guide for electronic health record systems*. XXXIV Simpósio Brasileiro de Engenharia de Software (SBES 2020). páginas 108-113. DOI: 10.1145/3422392.3422478.

2. **Lima et al. (2020d)**: Lima, Luis F. de, Silva, Fabiano, Grégio, André R. A. e Peres, Leticia M. (2020). *A systematic literature mapping of artificial intelligence planning in software testing*. *15th International Conference on Software Technologies* (ICSOFT 2020). páginas 152-159. DOI: 10.5220/0009829501520159.

3. **Lima et al. (2020b)**: Lima, Luis F. de, Horstmann, Matheus C., Neto, David N., Grégio, André R., Silva, Fabiano e Peres, Leticia M. (2020). *On the challenges of automated testing of web vulnerabilities*. *IEEE 29th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises* (WETICE 2020). páginas 203-206. IEEE. DOI: 10.1109/WETICE49692.2020.00047.

4. **Lima et al. (2020a)**: Lima, Luis F. de, Grégio, André R. A. e Peres, Leticia M. (2020). Teste de intrusão para aplicações web: um método com planejamento em inteligência artificial. Mostra de Trabalhos de Pós-graduandos em Ciência da Computação do Paraná do II Fórum dos Programas de Pós-Graduação em Computação do Paraná (II ForPPGC-PR).

5. **Lima e Peres (2021c)**: Lima, Luis F. de e Peres, Leticia M. (2021). Proposta de uso de técnicas de inteligência artificial para a avaliação de qualidade de sistemas de saúde. Mostra de Trabalhos de Pós-graduandos em Ciência da Computação do Paraná do III Fórum dos Programas de Pós-Graduação em Computação do Paraná (III ForPPGC-PR).

6. **Pereira et al. (2022)**: Pereira, Fernanda C., Neto, Gerhard B., Lima, Luis F. de, Silva, Fabiano e Peres, Leticia M. (2022). *A tool for software requirement allocation using artificial intelligence planning*. *IEEE 30th International Requirements Engineering Conference* (RE 2022). páginas 257-258. IEEE. DOI: RE54965.2022.00032.

Publicações e produções técnicas realizadas sob o escopo do Programa Estratégico Emergencial de Prevenção e Combate a Surtos, Endemias, Epidemias e Pandemias da CAPES:

1. **Lima e Peres (2021a)**: Lima, Luis F. de e Peres, Leticia M. (2021). Guia de aplicação de um protocolo de mapeamento sistemático para busca de aplicativos de saúde em repositórios não-acadêmicos - Perfil desenvolvedor de software. DOI: 10.5281/zenodo.11620098.

2. **Lima e Peres (2021b)**: Lima, Luis F. de e Peres, Leticia M. (2021). Guia de aplicação de um protocolo de mapeamento sistemático para busca de aplicativos de saúde em repositórios não-acadêmicos - Perfil profissional de saúde. DOI: 10.5281/zenodo.11620098.

3. **Lima e Peres (2021d)**: Lima, Luis F. de e Peres, Leticia M. (2021). Protocolo de mapeamento sistemático para busca de aplicativos de saúde em repositórios não-acadêmicos. I Workshop de Práticas de Ciência Aberta para Engenharia de Software (OpenScienSE 2021). páginas 7-12. SBC. DOI: 10.5753/opensciense.2021.17138.

4. **Lima et al. (2021)**: Lima, Luis F. de, Meireles, Flávia S. S. e Peres, Leticia M. (2021). Relatório técnico de estudos sobre sistemas de saúde de código aberto. Disponível em: https://zenodo.org/records/14176028.

5. **Lima e Peres (2022)**: Lima, Luis F. de e Peres, Leticia M. (2022). *A protocol based on systematic mapping studies for searching health applications in non-academic repositories*. *SBC Reviews on Computer Science* (ROCS), 2(1). páginas 1-12. DOI: 10.5753/reviews.2022.2724.

6. **Meireles et al. (2022)**: Meireles, Flávia S. S., Lima, Luis F. de e Leticia M. (2022). Catálogo de tecnologias abertas de telessaúde. Disponível em: https://acervodigital.ufpr.br/handle/1884/79572.

7. **Lima et al. (2023)**: Lima, Luis F. de, Meireles, Flávia S. S. e Peres, Leticia M. (2023). Tecnologias de engenharia de software para o desenvolvimento de sistemas de saúde de código aberto: um mapeamento sistemático da literatura. *Journal of Health Informatics* (JHI), 15(Especial). páginas 1-14. DOI: 10.59681/2175-4411.v15.iEspecial.2023.1079.