UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE AUTOMAÇÃO E SISTEMAS
CURSO DE GRADUAÇÃO EM ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Christopher de Carvalho

Desenvolvimento de uma PoC para recomendação de diagnósticos em configuradores de produtos baseados em restrições

Florianópolis
2021

Christopher de Carvalho

Desenvolvimento de uma PoC para recomendação de diagnósticos em configuradores de produtos baseados em restrições

Relatório final da disciplina DAS5511 (Projeto de Fim de Curso) como Trabalho de Conclusão do Curso de Graduação em Engenharia de Controle e Automação da Universidade Federal de Santa Catarina em Florianópolis.
Orientador: Prof. Ricardo José Rabelo, Dr.
Supervisor: Jacques Politi, Eng.

Florianópolis
2021

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Carvalho, Christopher de
Desenvolvimento de uma PoC para recomendação de diagnósticos em configuradores de produtos baseados em restrições / Christopher de Carvalho ; orientador, José Ricardo Rabelo, 2021.
95 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Engenharia de Controle e Automação, Florianópolis, 2021.

Inclui referências.

1. Engenharia de Controle e Automação. 2. Configurador de produtos baseado em restrições. 3. Tecnologias de recomendação. 4. Cálculo de diagnósticos personalizados. 5. Problemas de Satisfação de Restrições. I. Rabelo, José Ricardo. II. Universidade Federal de Santa Catarina. Graduação em Engenharia de Controle e Automação. III. Título.

Christopher de Carvalho

Desenvolvimento de uma PoC para recomendação de diagnósticos em configuradores de produtos baseados em restrições

Esta monografia foi julgada no contexto da disciplina DAS5511 (Projeto de Fim de Curso) e aprovada em sua forma final pelo Curso de Graduação em Engenharia de Controle e Automação

Florianópolis, 16 de agosto de 2021.

Prof. Hector Bessa Silveira, Dr.
Coordenador do Curso

Banca Examinadora:

Prof. José Ricardo Rabelo, Dr.
Orientador
UFSC/CTC/DAS

Jacques Politi, Eng.
Supervisor
Empresa WEG S.A

Prof. João Carlos Espíndola Ferreira, Dr.
Avaliador
UFSC/CTC/EMC

Prof. Marcelo de Lellis Costa de Oliveira, Dr.
Presidente da Banca
UFSC/CTC/DAS

AGRADECIMENTOS

à WEG, pela oportunidade de estágio, infraestrutura e notável preocupação e respeito para com seus colaboradores;

ao Jacques, pela supervisão e apoio durante o período de estágio, e por garantir que eu me sentisse à vontade na empresa;

ao Gustavo, por me guiar no desenvolvimento deste trabalho, e pela indicação de livros, artigos e ideias;

ao Lucas, pela amizade, ajuda, conselhos e os diálogos interessantes nos intervalos de almoço;

aos colegas da turma 16.1 de automação, pelo companheirismo e amizade durante a trajetória de graduação. Sem vocês, eu provavelmente teria desistido no meio do caminho;

ao prof. Rabelo, por aceitar ser meu orientador, pela prontidão nas respostas e conselhos importantes em momentos difíceis;

aos professores do DAS, pelo ensino de altíssima qualidade, dedicação ao trabalho e constantes esforços para melhorar o curso de automação;

à Luiza, pela companhia e carinho;

aos meus pais, Paulo e Estela, por todo o suporte, dedicação incondicional, e por sempre acreditarem no meu potencial;

RESUMO

Atualmente, observa-se um aumento crescente no número de vendas realizadas pela internet. A oferta de produtos cada vez mais complexos tornou os sistemas de configuração de produtos em importante ferramentas que auxiliam os usuários a customizarem produtos de acordo com suas necessidades. A configuração de produtos pode ser modelada como um Problema de Satisfação de Restrições (CSP), onde o processo de configuração é uma busca interativa. No decorrer deste processo, é possível a ocorrência de situações em que um dado conjunto de requisitos do usuário é inconsistente (possui contradições lógicas) com o subjacente conjunto de restrições que modela o produto, tal que nenhuma solução de configuração possa ser encontrada. Neste cenário, é necessário que a aplicação providencie um meio para a resolução das inconsistências geradas. A abordagem mais comum é através do cálculo e apresentação de diagnósticos, que definem alternativas para a remoção de determinadas seleções do usuário que impedem a obtenção de uma solução válida. Neste trabalho, buscando melhorar a experiência de utilização do configurador, implementa-se uma técnica de recomendação que possibilita que o cálculo de diagnósticos seja realizado de maneira personalizada ao usuário corrente da aplicação, com base na similaridade entre as seleções definidas pelo mesmo e as configurações realizadas em sessões prévias do sistema. Para tal, foi desenvolvida uma aplicação Prova de Conceito (PoC), cujo objetivo é demonstrar a técnica e ser posteriormente integrada ao *framework* de configuração de produtos da empresa.

Palavras-chave: Configuradores de produtos baseados em restrições. Tecnologias de recomendação. Cálculo de diagnósticos personalizados. Problemas de Satisfação de Restrições.

ABSTRACT

Nowadays, it can be observed an increase in the number of sales made over the internet. Increasingly complex products are offered, which turns product configuration systems into important tools that help end users customize products according to their needs. Product configuration can be modeled as a Constraint Satisfaction Problem (CSP), where the configuration process is an interactive search. During this process, situations may occur where a given set of user requirements is inconsistent (has logical contradictions) with the underlying set of constraints that model the product, so that no configuration solution can be found. In this scenario, it is necessary for the application to provide a means to resolve the generated inconsistencies. The most common approach is through the calculation and presentation of diagnoses, which are alternatives to remove certain user requirements that prevent the obtainment of a valid solution. In order to improve the configuration experience, from the user point of view, this works implements the addition of a recommendation technique that enables the proposition of personalized diagnostics to the current user, based on the similarity between its requirements and complete configurations from previous sessions. For this purpose, a PoC application is developed to demonstrate the technique and later be integrated into the company's product configuration framework.

Keywords: Constraint-based product configurators. Recommendation technologies. Personalized diagnoses. Constraint Satisfaction Problems.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

B2B	*Business to Business*
B2C	*Business to Consumer*
BOM	*Bill of Materials*
CAD	*Computer Aided Design*
CPQ	*Configure, price and quote*
CSP	*Constraint Satisfaction Problem*
EIB	*Equal-Is-Better*
ERP	*Enterprise Resource Planning*
FIFO	*First In, First Out*
HSDAG	*Hitting Set Directed Acyclic Graph*
kNN	*k-Nearest-Neighbors*
LIB	*Lower-Is-Better*
MAUT	*Multi-Attribute Utility Theory*
MIB	*More-Is-Better*
MRP	*Material Requirement Planning*
MVC	*Model-View-Controller*
NIB	*Near-Is-Better*
ORM	*Object-Relational Mapping*
PoC	*Proof of Concept*
POJO	*Plain Old Java Object*
SSR	*Server Side Rendering*
UML	*Unified Modeling Language*
WIP	*Work In Progress*

SUMÁRIO

1 INTRODUÇÃO

Nos primórdios, o desenvolvimento de objetos complexos, como utensílios e ferramentas, foi fundamental para a evolução de nossa espécie, ao compensar nossa relativa fragilidade física e permitir-nos a realizar tarefas que não conseguiríamos somente com a utilização das mãos. Avançando um pouco no tempo, uma das facetas da interação entre pessoas e objetos pode ser vista no consumo de produtos. Produtos são objetos especializados, com valor de mercado, que possuem a finalidade de suprir um desejo ou uma necessidade do consumidor (FERNANDES, 2019), e observa-se que o consumo dos mesmos tornou-se um aspecto importante na vida da maioria das pessoas.

Também intrínseco ao ser-humano é a necessidade de diferenciar-se dos demais, e, no âmbito do consumo de produtos, esse aspecto da psique humana se manifesta no desejo que os consumidores têm pela personalização de seus bens de consumo (GILMORE; PINE II, 1997). Quando adveio o modo de produção em massa, no início do século XX, as oportunidades de personalização dos produtos eram bastante limitadas, o que ficou simbolizado pela famosa frase de Henry Ford: “Seu Ford Modelo T pode ter qualquer cor, desde que seja preto”. Nas últimas décadas, porém, o cenário mudou completamente. Visando manter aquecido o espírito de competitividade, tem se pronunciado uma movimentação geral por parte das empresas, onde estas têm buscado se tornarem cada vez mais centradas no consumidor (GILMORE; PINE II, 1997). Em resposta à crescente demanda por produtos cada vez mais personalizáveis/customizáveis, surgiu, no início da década de 90, o conceito de customização em massa, do inglês *mass customization*, cuja implementação requer a reimaginação do modo de produção nas fábricas, de forma a suportar, em diferentes graus de complexidade, a personalização dos bens de consumo a serem produzidos. A customização em massa é uma evolução natural do conceito de produção em massa, sendo definida em Tseng e Jiao (2001) como “a produção de bens e serviços para atender às necessidades individuais dos consumidores com eficiência próxima a de produção em massa”, e constitui numa das problemáticas que a indústria 4.0 contribui para solucionar. É uma tendência observada não só na produção direcionada aos consumidores finais, *Business to Consumer* (B2C), mas também no mercado entre empresas, *Business to Business* (B2B), onde existe uma alta demanda pela personalização de produtos especialmente complexos (GRAFMÜLLER; HABICHT, 2017).

Apesar de a customização em massa estar tradicionalmente associada às tecnologias flexíveis de manufatura e ao desenvolvimento de arquiteturas de produção modulares (PILLER; BLAZEK, 2014), uma atividade fundamental para o sucesso de qualquer empreitada nesse modo de produção, no cenário atual, é a concepção, projeto e implementação de configuradores de produtos (HOTZ *et al.*, 2014).

Um configurador de produtos é um *software*, a ser disponibilizado pelo fabricante do produto ao consumidor interessado, que contém embarcado um modelo do produto oferecido. Partindo de um estado inicial, onde nenhuma ou poucas características estão definidas, o configurador recebe *inputs* fornecidos pelo usuário, através de uma sequência de passos denominada como processo de configuração (Seção 3.1.2). Esses passos são percorridos até que a configuração do produto seja finalizada, ou seja, todas as características necessárias do produto estejam definidas.

O processo de configuração é definido por Hesselfeldt (2019) como "um processo interativo, onde o usuário escolhe uma *feature*, a ser validada através de uma *engine* com regras de configuração, antes de permitir que usuário efetue a próxima escolha sobre as *features* restantes de seu produto. A aparente simplicidade desse processo esconde a complexidade na construção e validação das regras utilizadas para definir o comportamento do configurador de produtos".

Além da necessidade de modelar o produto e integrar a solução de configuração no sistema de TI da empresa, uma tarefa ao implementar tais sistemas é a de providenciar uma interface apropriada ao usuário, que seja fácil de entender e intuitiva de operar, sem a necessidade de treinamento prévio. A interface também deve oferecer suporte à navegação das seleções, potencialmente complexas, as quais devem ser definidas pelos usuários durante processos de configuração. Quanto mais eficiente for o sistema de configuração nestes aspectos, maior a probabilidade de que o usuário fique satisfeito com a experiência de utilização do sistema, e, consequentemente, maior o número de vendas.

Uma situação comum em processos de configuração é quando os requisitos do usuário corrente (aquele que está interagindo com o sistema) não podem ser atendidos por nenhuma das configurações disponibilizadas pelo configurador. Tal situação pode ocorrer, por exemplo, porque os requisitos especificados estão em falta no sistema de produção, ou não foram previstos no projeto do configurador, ou ainda porque são infactíveis segundo a modelagem do produto. Hotz *et al.* (2014) sugere que o registro das sessões de configuração que foram efetuadas, não só as concluídas, mas também as interrompidas, pode fornecer *insights* valiosos para as empresas, pois possibilitam o levantamento de uma base de conhecimento sobre os usuários, do inglês *customer knowledge base*, que permite inferir tendências de mercado e até mesmo guiar o desenvolvimento de novos produtos.

Especialmente no caso da configuração de produtos complexos, é comum que o usuário não saiba de antemão o que realmente deseja, mas que defina os requisitos no escopo do processo de configuração, podendo modificar suas escolhas múltiplas vezes no decorrer da sessão. Tais mudanças podem gerar estados de inconsistência, onde determinadas restrições do modelo do produto ficam insatisfeitas, devido à combinação de seleções incongruentes. Segundo Hotz *et al.* (2014), um sistema de configuração

que possa ser chamado de “inteligente” deve estar apto a tratar essas situações de alguma forma. Nesse projeto, a solução proposta se dá por meio da recomendação de alternativas de reparo personalizadas ao usuário corrente (Seção 2.4).

Na próxima seção, os objetivos do projeto são definidos, e este capítulo introdutório finaliza com informações sobre a estrutura do documento.

1.1 OBJETIVOS

O principal objetivo do projeto descrito nesse documento é o desenvolvimento de uma aplicação *Proof of Concept* (PoC), do inglês para “prova de conceito”, que demonstre a recomendação de diagnósticos em configuradores de produtos baseados em restrições.

Este projeto faz parte de um esforço maior, por parte da empresa, no sentido de atualizar seus configuradores de produtos baseados em regras para configuradores de produtos baseados em restrições (detalhes na Seção 2.3.3). Posteriormente à conclusão deste trabalho, pretende-se realizar a integração da aplicação desenvolvida ao *framework* de configuração de produtos baseados em restrições que está em fase de desenvolvimento na empresa.

A adição da recomendação de diagnósticos é apenas uma dentre as muitas técnicas de recomendação passíveis de serem adicionadas em configuradores de produtos (Seção 3.3), e busca melhorar a experiência dos usuários finais na utilização dos mesmos, especificamente no cenário de resolução de conflitos (Seção 3.1.2.2). Tanto a recomendação de diagnósticos quanto outros conceitos específicos que foram mencionados serão detalhados na fundamentação teórica (Capítulo 3). Do objetivo principal, derivam-se os seguintes objetivos específicos:

- Pesquisar e estudar diferentes formas de efetuar a recomendação de diagnósticos (e escolher uma para implementar na aplicação);
- planejar e desenvolver uma aplicação para demonstrar a abordagem selecionada;
- implementar e testar os algoritmos de detecção de conflitos e cálculo de diagnósticos com suporte às heurísticas de recomendação;
- integrar a aplicação desenvolvida como um módulo de recomendação ao *framework* de configuração principal (fora do escopo da monografia).

1.2 ESTRUTURA DO DOCUMENTO

Este documento está dividido em sete capítulos. No capítulo 2, é feita uma contextualização da empresa, então introduz-se o tema de configuradores de produtos,

buscando facilitar ao leitor o entendimento de como o trabalho desenvolvido insere-se dentro de um projeto maior de atualização dos configuradores da empresa.

No capítulo 3, apresenta-se a fundamentação teórica, levantada a partir dos estudos e investigação da literatura, conduzidos para aquisição do conhecimento necessário para efetuar o desenvolvimento. No capítulo 4, são apresentadas as ferramentas utilizadas, metodologia de desenvolvimento empregada, especificação de requisitos e atividades planejadas.

No capítulo 5, é apresentada a implementação efetiva da aplicação, relatando-se as diversas etapas do processo de desenvolvimento, com os resultados sendo apresentados no capítulo 6. Por fim, o capítulo 7 traz as conclusões finais e sugestões de tópicos a serem investigados em trabalhos futuros.

2 CONTEXTUALIZACAO

Este projeto de fim de curso foi desenvolvido em paralelo à realização de um estágio obrigatório na empresa WEG, em Jaraguá do Sul, na seção de Desenvolvimento de Sistemas de Engenharia do departamento de TI. O período de duração do estágio foi de cinco meses, de 16/03/2021 até 13/08/2021.

O primeiro mês de estágio foi dedicado à ambientação na empresa e ao estudo das tecnologias utilizadas pela mesma. Em função do período de pandemia, houve uma redução do contingente em expediente presencial no departamento, com a maioria dos colaboradores da TI exercendo suas atividades de modo remoto, o que acabou dificultando um pouco a interação, mas também oferecendo a oportunidade de experienciar e lidar com esse tipo de situação.

Foi logo definido que o projeto seria feito com base no configurador de produtos em desenvolvimento pela empresa (Seção 2.3.3). O escopo inicial era amplo, e a proposta era desenvolver alguma atividade no intuito de auxiliar no processo de transição de configuradores baseados em regras para configuradores baseados em restrições (Seção 2.3). Durante o refinamento, o escopo foi alterado para o desenvolvimento de algum recurso avançado de configuração de produtos, e buscou-se definir um projeto que fosse passível de ser realizado por completo no período de realização do estágio, adequando-se às necessidades de um PFC.

Dentre as possibilidades levantadas, foi escolhida a adição de tecnologias de recomendação ao configurador, especificamente através do desenvolvimento e posterior integração de uma aplicação demonstrativa para a recomendação de diagnósticos, sendo uma melhoria que busca especificamente aprimorar a experiência do usuário final, sendo especialmente útil no contexto da configuração de produtos complexos (FELFERNIG *et al.*, 2006).

2.1 A EMPRESA

A WEG é uma empresa sediada na cidade de Jaraguá do Sul, em Santa Catarina. Foi fundada em 1961 por Werner, Eggon e Geraldo, cujos nomes formam o acrônimo da empresa. Constitui numa das maiores fabricantes de equipamentos elétricos do mundo. É conhecida principalmente por fazer motores elétricos, e contém uma ampla gama de produtos, como geradores, transformadores, disjuntores, tintas, vernizes, dentre outros. Tem como missão o “crescimento contínuo e sustentável, mantendo a simplicidade” (WEG, 2021b), e almeja ser, cada vez mais, uma referência global em máquinas elétricas.

O êxito da empresa pode ser inferido ao analisar alguns números recentes, retirados de WEG (2021c): a WEG teve faturamento de R$ 17,47 bilhões em 2020, tem filiais em 36 países, fábricas em 12 países, estando presente em 5 continentes. Tem

um portfólio com mais de 1200 linhas de produtos, e mais de 33.300 colaboradores e 3.600 engenheiros.

Em constante processo de atualização, metade dos faturamentos são provenientes de suas mais recentes inovações no âmbito de soluções digitais (CARVALHO, 2021). Um exemplo é o *WEG Digital Solutions*, ecossistema que conecta e integra equipamentos e sensores, aplicando conceitos importantes para o desenvolvimento da indústria de ponta, como plataforma *IoT*, recursos avançados de conectividade, inteligência artificial e *softwares* voltados a indústria 4.0. A busca por versatilidade também pode ser observada pelas recentes aquisições de *startups* na área de tecnologia, como a MVisia e a BirminD (RENNER, 2020). Outro projeto interessante que mostra o comprometimento da empresa em acelerar e viabilizar o desenvolvimento de soluções em indústria 4.0 no país é a recente parceria com a Nokia, no desenvolvimento de uma fábrica automatizada que deve servir como piloto de testes para uma rede privada 5G e conceitos de automação em nuvem (ANAND, 2021).

Nesse contexto de inovação, com a digitalização permeando todos os setores produtivos da empresa, existe uma demanda para que as tecnologias empregadas em produção também sejam atualizadas de forma a acompanhar o setor de pesquisa e desenvolvimento. Com a maior parte das vendas sendo atualmente realizada pela *internet*, aplicações de configuração de produtos são utilizadas pela maioria dos clientes que desejam comprar algum produto com a WEG, constituindo num ponto crucial de negócio, portanto a necessidade pela atualização constante dessas aplicações. Um configurador de produtos é um subsistema oferecido pela solução de *Configure, price and quote* (CPQ) da empresa, a ser detalhado a seguir.

2.2 SISTEMAS CPQ

Segundo a definição encontrada em Techopedia (2021), CPQ (sigla do inglês para “configurar, precificar e cotar”) é um termo usado na indústria para descrever sistemas de *software* que auxiliam os vendedores na cotação e configuração de produtos completos. Algumas empresas especializadas oferecem soluções no âmbito de CPQ, como a Tacton (ORSVÄRN; AXLING, 1999), ConfigIT (MOELLER *et al.*, 2001), EngCon (HOTZ; GÜNTER, 2014), dentre outras. Algumas dessas soluções comerciais tem foco maior ou até mesmo exclusivo em apenas um dos componentes do acrônimo que forma o CPQ.

Neste trabalho, o foco do sistema desenvolvido é exclusivamente no componente de configuração do CPQ, denominado simplesmente de ”configurador de produtos“. Na etapa de desenvolvimento em que atualmente se encontra, o *framework* de configuração da empresa é completamente agnóstico às noções de precificação e cotação de componentes. Pela perspectiva de produção, ao finalizar uma sessão de configuração, o configurador precisa apenas gerar e encaminhar uma lista de mate-

riais (BOM, detalhado na Seção 3.1.2) ao setor de produção, com as atividades de precificação e cotação a serem realizadas por outros sistemas fornecidos pela solução de *Enterprise Resource Planning* (ERP) adotada pela empresa.

2.3 CONFIGURADORES DE PRODUTOS

No contexto de configuração, a complexidade de um produto está diretamente relacionada ao grau de customização oferecido, isto é, de quantas maneiras diferentes o usuário pode personalizar o seu produto. Segundo Blumöhr *et al.* (2011), especialmente na configuração de produtos muito complexos, pode ocorrer um problema de explosão combinatória das alternativas de características a serem manejadas pelo sistema, como mostra a Tabela 1.

Tabela 1 – Explosão combinatória ao juntar *n* características com *m* valores.

n \ m	1	2	3	4	5	…	10	…	15
"linear"	*1*	*2*	*3*	*4*	*5*	…	*10*	…	*15*
"quadratic"	*1*	*2*	*9*	*16*	*25*	…	*100*	…	*225*
2	**2**	**4**	**8**	**16**	**32**	…	**1024**	…	**32,768**
"e ≈ 2.718 …"	*≈ 2.7*	*≈ 7.4*	*≈ 20*	*≈ 55*	*≈ 148*	…	*≈ 22,026*	…	*≈ 3,269,017*
3	**3**	**9**	**27**	**81**	**243**	…	**59,049**	…	**14,348,907**
5	**5**	**25**	**125**	**625**	**3.125**	…	**9,765,625**	…	**30,517,578,125**
7	**7**	**49**	**343**	**2401**	**16,807**	…	**282,475,249**	…	**4,747,561,509,943**

Fonte – (BLUMÖHR *et al.*, 2011).

Devido ao problema de exploração combinatória, toda solução de CPQ completa deve oferecer uma *engine* de configuração, incorporada no configurador de produtos, a qual é responsável por tratar esse problema. Quanto a finalidade de configuradores, Binder *et al.* (2020) sumariza dizendo que “o configurador de produtos trata do desafio de permitir ao usuário combinar partes e componentes de um determinado produto, visando assim chegar numa solução viável de configuração”.

Uma breve história sobre o desenvolvimento de tecnologias de configuração de produtos é apresentada em Hotz *et al.* (2014), podendo-se dividi-las em duas categorias: configuradores baseados em regras e configuradores baseados em restrições, a serem detalhados a seguir.

2.3.1 Configuradores baseados em regras

Configuradores baseados em regras são provenientes da primeira geração de configuradores, desenvolvidos na década de 70 com base nos resultados de pesquisa em inteligência artificial na década anterior (BLUMÖHR *et al.*, 2011). Nessa abordagem, é necessária a antecipação prévia de todas as possíveis formas de se configurar um produto, através de verificações no formato *if-then-else*, que determinam o que fazer em cada cenário. Um configurador baseado em regras pode ter na casa das centenas de milhares de regras, que devem ser sequenciadas de forma a garantir uma ordem de verificação correta durante os processos de configuração.

A Figura 1 mostra a típica declaração de uma regra, no configurador baseado em regras *R1/XCON*. A parte de condição (*IF*) descreve a configuração corrente, e a parte de ação (*THEN*) define que ações tomar neste contexto.

Figura 1 – Declaração de uma regra no *R1/XCON*.

```
ASSIGN-POWER-SUPPLY-1

IF:THE MOST CURRENT ACTIVE CONTEXT IS ASSIGNING A POWERSUPPLY
   AND AN SBIMODULE OF ANY TYPE HAS BEEN PUT IN A CABINET
   AND THE POSITION IT OCCUPIES IN THE CABINET IS KNOWN
   AND THERE IS SPACE IN THE CABINET FOR A POWER SUPPLY
   AND THERE IS NO AVAILABLE POWER SUPPLY
   AND THE VOLTAGE AND FREQUENCY OF THE COMPONENTS IS KNOWN

THEN: FIND A POWER SUPPLY OF THAT VOLTAGE AND FREQUENCY
      AND ADD IT TO THE ORDER
```

Fonte – (HOTZ *et al.*, 2014).

Um grande problema dos configuradores baseados em regras é sua falta de flexibilidade, evidenciada sempre que o administrador de regras (engenheiro do conhecimento responsável por dar manutenção ao sistema) precisa adicionar, alterar ou remover regras do configurador (HOTZ *et al.*, 2014). Essa desvantagem é decorrente do acoplamento entre o modelo de domínio e o processo de resolução, intrínseco à utilização de regras. Ao adicionar uma nova regra, por exemplo, o administrador deve revisar com cuidado todas as regras anteriores, e verificar se a posição de inserção desta nova regra pode gerar um efeito cascata de atualização e reordenação (SALOWAY *et al.*, 1987).

Devido aos esforços necessários, também é comum ver, na prática, a simples inserção no final do código de uma regra que invalide a anterior, seguido pela adição da nova regra. Porém, a longo prazo, essa abordagem pode acabar sobrecarregando o sistema, piorando o tempo de resposta e dificultando os processos de manutenção.

2.3.2 Configuradores baseados em restrições

Desenvolvidos nas décadas de 80 e 90, também são conhecidos como configuradores baseados em modelo, pois requerem uma modelagem do produto a ser configurado (FALKNER *et al.*, 2017). Conseguem manejar um conjunto complexo de restrições de configuração, dessa forma tratando o problema de explosão combinatória. Constituem numa aplicação prática de *Constraint Satisfaction Problem* (CSP), subárea importante da Inteligência Artificial, ao problema de configuração de produtos (Seção 3.1.1).

No contexto de configuração, é recomendado não usar os termos “regra” e “restrição” como sinônimos, de modo a evitar possíveis confusões (TACTON, 2021a). A Figura 2 mostra a declaração de uma restrição na linguagem de satisfação de restrições *MiniZinc*, que envolve as variáveis *usage* e *number_cpus*, para um configurador de computadores.

Figura 2 – Declaração de uma restrição na linguagem *MiniZinc*.

```
int : multimedia = 1;
int : scientific = 2;
int : internet = 3;
var 1 .. 3: usage ;
var 1 .. 16: number_cpus ;
constraint (usage == multimedia) -> (number_cpus >= 4);
```

Fonte – (FALKNER *et al.*, 2017).

Existem diferentes abordagens para a representação de conhecimento dentro de configuradores baseados em restrições. A mais simples é efetuar a representação do modelo de configuração dos produtos com CSPs estáticos, a ser exemplificada na Seção 3.1.1. Posteriormente, foram criadas abordagens mais complexas, como a representação em CSPs dinâmicos, que permite a ativação/inativação de variáveis, conforme o contexto, e também a representação com CSPs generativos, que permite modelar o produto e as restrições num estilo orientado a objetos, definindo uma hierarquia entre os componentes. Neste projeto, visto que a escolha por uma abordagem de representação não interfere no processo de recomendação de diagnósticos, a representação estática (mais simples) foi escolhida.

Em contraposição aos configuradores baseados em regras, a abordagem baseada em restrições propicia uma separação completa entre o domínio do produto e a resolução do problema de configuração (HOTZ *et al.*, 2014), permitindo assim que um mesmo configurador possa ser utilizado para domínios diferentes, como evi-

denciado pelo *framework* de configuração S'UPREME da Siemens, em Haselböck e Schenner (2014). A Tabela 2 apresenta algumas vantagens da abordagem baseada em restrições com relação à baseada em regras.

Tabela 2 – Comparação entre configuradores baseados em regras e configuradores baseados em restrições

Baseado em regras	**Baseado em restrições**
De construção difícil e manutenção toma bastante tempo.	Construçao e manutenção mais simples.
A escolha de módulos/funcionalidades deve ser efetuada numa ordem predefinida.	Liberdade de efetuar as escolhas em qualquer sequência.
'Modo batelada', i.e. as escolhas são definidas como inputs, depois executa-se o algoritmo, com a possível consequência de que escolhas inconsistentes foram efetuadas.	'Modelo interativo', i.e. as consequências são derivadas imediatamente a cada escolha do usuário.

Fonte – Traduzido pelo autor de (JORGENSEN, 2009).

2.3.3 Configuradores de produtos na WEG

A empresa fabrica produtos cuja especificação de atributos é consideravelmente complexa, como bombas, compressores, ventiladores, transformadores e motores para usos específicos (WEG, 2021a). Além das linhas produzidas em massa, também são oferecidos produtos customizáveis de acordo com as necessidades do cliente, que por sua vez pode optar dentre diversas características, como, para o caso de motores, uso, forma, tamanho, potência, etc.

Atualmente, os configuradores utilizados em produção são baseados em regras, todavia a empresa possui um *framework* de configuração baseado em restrições em estado de desenvolvimento, porém ainda é necessária a implementação de melhorias e novas funcionalidades, bem como a tradução do conjunto de regras atuais, que chegam na casa de centenas de milhares, para um conjunto de restrições. Só após a conclusão dessas atividades é que se poderá dar início ao processo de substituição do configurador atual, baseado em regras, para o novo configurador, baseado em restrições. Uma das melhorias planejadas é a adição de tecnologias de recomendação ao configurador de produtos, a ser uma das contribuições realizadas por este trabalho.

2.3.4 Soluções comerciais de configuradores

A tecnologia de configuração por restrições já está bem consolidada no mercado, como relatado em Falkner *et al.* (2017), que conta a trajetória de 25 anos de aplica-

ção bem-sucedida de configuradores baseados em restrições pela empresa alemã Siemens. Uma das aplicações mencionadas é um configurador para um sistema de intertravamento ferroviário para a Áustria, que está em operação desde 1989 até hoje. Outra aplicação é um configurador para *switches* de telecomunicação.

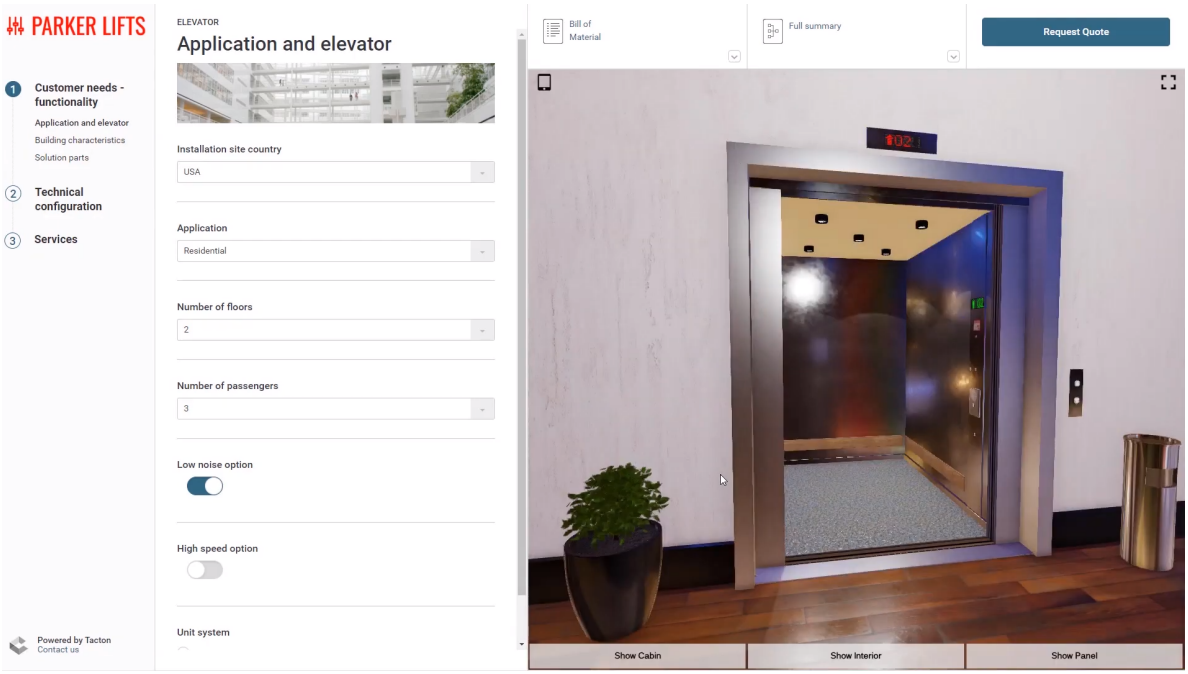
Entre as soluções de CPQ mencionadas na seção Seção 2.2, uma que representa o que está disponível de mais avançado no mercado atualmente é o configurador de produtos oferecido pela empresa sueca Tacton. Um caso de sucesso do mesmo foi evolução dos configuradores baseados em regras para configuradores baseados em restrições da empresa ABB, líder mundial em automação e robótica. A ABB estava trabalhando com dois configuradores diferentes para sua linha de acessórios, cobrindo apenas 20% dos produtos. Efetuaram a aquisição e transição para o Tacton CPQ e conseguiram cobrir 100% da linha dentro de quatro meses, representando o que antes necessitava de 500.000 regras com apenas 150 restrições (TACTON, 2021d).

O vídeo "*First Glance at Tacton CPQ*" (TACTON, 2021b) mostra a utilização de um configurador de ponta, a partir da perspectiva de um usuário final sem expertise no domínio do produto, no caso a configuração de elevadores. A interface gráfica contém uma integração com o *software* de *Computer Aided Design* (CAD) 3D Solidworks, exibindo uma prévia do produto final, atualizada em tempo real em resposta às seleções efetuadas pelo usuário (ver Figura 3). Uma funcionalidade avançada oferecida por esse configurador é permitir, através de realidade aumentada (usando a câmera do celular), visualizar uma prévia de como ficaria a instalação do elevador no local pretendido (TACTON, 2021c).

Outra funcionalidade avançada é a possibilidade de realizar configuração baseada em necessidade (do inglês *needs-based configuration*), onde o usuário responde a uma série de perguntas, informando aquilo que precisa (i.e., requisitos funcionais), e uma solução configuração é então proposta pelo configurador com base nas respostas fornecidas pelo usuário. Naturalmente, também é mantida a possibilidade de configuração tradicional, a partir da inserção das especificações técnicas do elevador. Essa funcionalidade é muito interessante para aprimorar a experiência do usuário final, principalmente quando o configurador é utilizado por usuários não-técnicos, sendo uma possível área de desenvolvimento de uma trabalho futuro para o *framework* de configuração da empresa.

Do ponto de vista estratégico, ao surgir a necessidade de atualização do configurador de produtos de uma empresa, devem ser discutidas as possibilidades de adquirir uma solução comercial ou desenvolver uma solução proprietária de configuração. Existem muitas bibliotecas *open-source* para a resolução de CSPs, em constante manutenção pela comunidade, as quais podem ser aplicadas à finalidade de configuração de produtos. A escolha pelo desenvolvimento de uma solução proprietária oferece um ganho de flexibilidade, permitindo uma implementação especificamente moldada

Figura 3 – Configurador de elevadores da *Tacton*.



Fonte – (TACTON, 2021b).

conforme as necessidades da empresa. Além disso, também pode evitar eventuais problemas de integração com os outros sistemas, já que o desenvolvimento é efetuado “do zero” e traz oportunidades de crescimento aos próprios desenvolvedores.

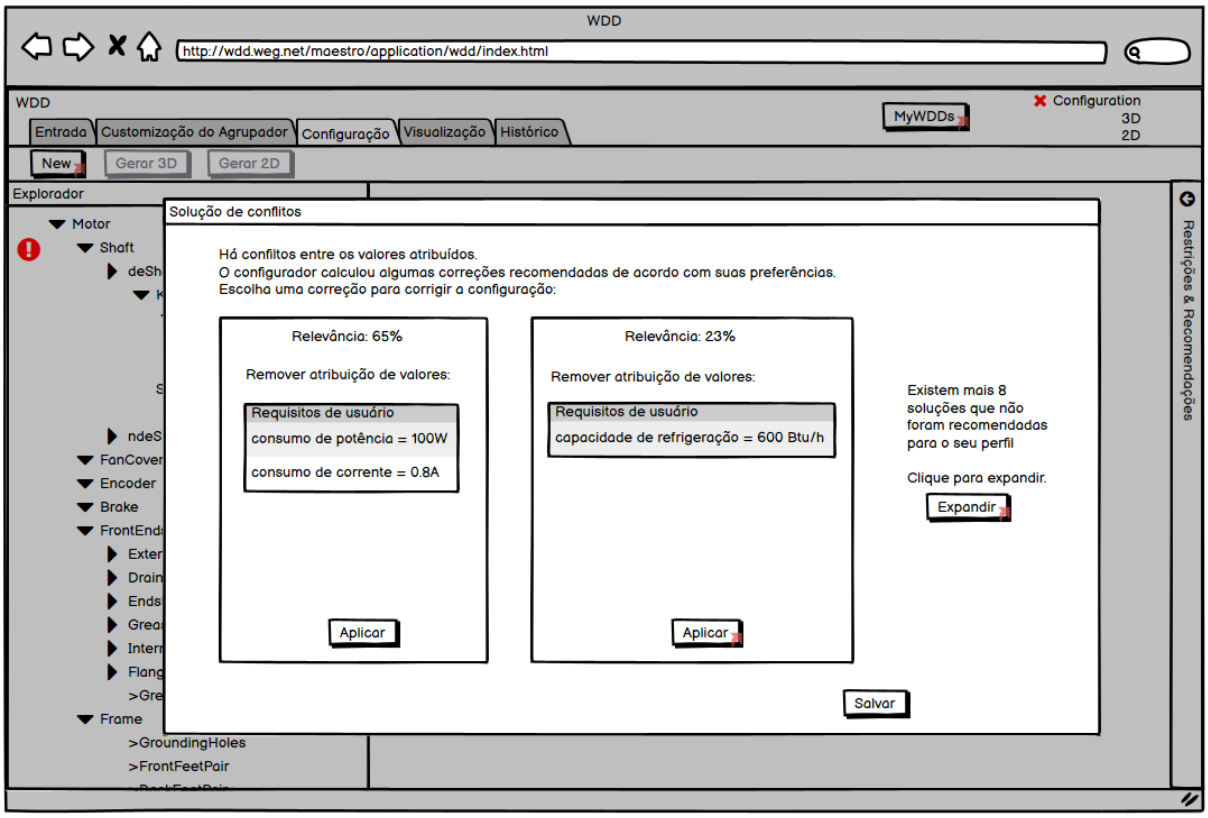
2.4 DESCRIÇÃO DO PROBLEMA E MOTIVAÇÃO

O sistema a ser desenvolvido é uma aplicação para implementar e demonstrar uma técnica de recomendação de diagnósticos. Para exemplificar e descrever o problema, supõe-se o cenário em que o representante de uma empresa que fabrica geladeiras esteja configurando um compressor no configurador de produtos da WEG, e que, durante esse processo, acabe definindo uma capacidade de refrigeração incompatível com as especificações de consumo. Nessa situação, é dito que um estado de inconsistência foi atingido. Supondo que o configurador identifique dez diagnósticos, onde cada diagnóstico conste numa alternativa de remoção de um ou mais requisitos previamente definidos pelo usuário, o representante deverá escolher a aplicação de um dos diagnósticos apresentados para dar prosseguimento ao processo de configuração.

Buscando melhorar a experiência do usuário, é interessante fazer com que o configurador dê ênfase às alternativas identificadas como mais relevantes ao usuário corrente. Por exemplo, se o sistema tiver acesso a um banco de dados, que contenha informações sobre as preferências dos usuários, poderia identificar o representante

(e os produtos que já configurou) e propor a remoção de características não pertinentes a compressores de geladeira, desta forma ajudando-o não somente a retomar o processo de configuração, mas também a chegar na solução (configuração) desejada mais rapidamente. A Figura 4 exemplifica essa situação, mostrando a interface de configuração do ponto de vista do representante, na situação de conflito mencionada. Supondo que dez diagnósticos são calculados, apenas os dois de maior relevância, ou seja, os mais recomendados, são apresentados ao usuário, enquanto os demais são ocultos, evitando assim poluir a tela e confundir o usuário.

Figura 4 – Exemplo de resolução de conflitos com recomendação.



Fonte – WEG.

A melhoria na experiência do usuário é o principal ganho obtido com a adição da recomendação de diagnósticos. Não é uma funcionalidade indispensável do configurador, mas está presente nos configuradores mais avançados atualmente disponíveis no mercado (Seção 2.3.4).

Observa-se que o exemplo dado para efetuar a recomendação de diagnósticos, baseando-se em preferências de usuário, é apenas uma das possíveis abordagens para esse fim. Poder-se-ia, por exemplo, considerar as experiências prévias de todos os usuários, dentre outras técnicas. O estudo e a implementação de uma técnica de recomendação foi parte da investigação conduzida neste trabalho, e acabou-se

optando por implementar a recomendação de diagnósticos baseada em similaridade (Seção 3.3.2.1).

Sumarizando, deve ser desenvolvida uma aplicação que demonstre o cálculo de diagnósticos perante um conjunto de requisitos inconsistentes, apresentando-os ao usuário corrente de forma personalizada, seguindo alguma técnica de recomendação.

3 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta a fundamentação teórica da aplicação desenvolvida neste trabalho, obtida como resultado da investigação literária conduzida sobre o tema de sistemas de recomendação no apoio à resolução de conflitos em configuradores de produtos. A Seção 3.1 introduz o CSP, estrutura matemática em que se fundamentam os configuradores de produtos baseados em restrições. A Seção 3.2 apresenta os algoritmos estudados e posteriormente implementados para efetuar a detecção de conflitos e cálculo de diagnósticos em situações de inconsistência durante processos de configuração. A Seção 3.3 introduz a integração de tecnologias de recomendação em configuradores de produtos, com foco na recomendação de diagnósticos, principal tópico da implementação.

3.1 CONSTRAINT SATISFACTION PROBLEMS

Constraint Satisfaction Problem (CSP), do inglês para "problema de satisfação de restrições", é uma abordagem matemática para a resolução de problemas, os quais são definidos por um conjunto de variáveis cuja atribuição de valores está submetida (deve satisfazer) a um determinado conjunto de restrições (RUSSELL; NORVIG, 2020).

Um CSP pode ser definido como uma tripla (V, D, C), onde $V = \{v_1, v_2, ..., v_n\}$ representa o conjunto de variáveis, $D = \{dom(v_1), dom(v_2), ..., dom(v_n)\}$ representa o domínio correspondente das variáveis, e $C = \{c_1, c_2, ..., c_m\}$ representa o conjunto de restrições impostas às variáveis de V.

O espaço de solução $S = \{s_1, s_2, ..., s_p\}$ representa o conjunto de todas as possíveis soluções do CSP, onde cada solução s_i é uma atribuição de valores das variáveis contidas em V - cada variável recebe um valor do seu domínio associado, de tal forma que todas as restrições de C sejam satisfeitas, isto é, nenhuma restrição seja quebrada/violada.

Falkner *et al.* (2017) observa que, devido à estrita e simples definição dos CSPs, muitas técnicas de raciocínio eficientes e robustas foram desenvolvidas para a resolução destes problemas, como propagação de restrições (Seção 3.1.2.1), busca completa, busca local com heurísticas, dentre outras. Essas técnicas são apresentadas detalhadamente em Russell e Norvig (2020), juntamento com a modelagem de alguns problemas clássicos em CSP, como o problema das oito rainhas, o problema da coloração de mapas, e a régua de Golomb.

3.1.1 Aplicação de CSP para configuração de produtos

De acordo com Brailsford *et al.* (1999), tecnologias de restrições são uma das aplicações mais bem-sucedidas de inteligência artificial em ambientes industriais, apli-

cáveis para problemas de configuração, agendamento e sistemas de recomendação. Ademais, (HOTZ *et al.*, 2014) afirma que representações baseadas em restrições, através da modelagem em CSP, são amplamente utilizadas na definição de modelos de configuração.

Como visto na Seção 2.3, o problema da configuração de produtos pode ser definido como "o desafio de encontrar um conjunto de componentes que atenda a um conjunto de restrições" (BINDER *et al.*, 2020). Desta forma, as variáveis do CSP representam os componentes ou as partes de um determinado produto. Cada domínio contém os valores possíveis para a respectiva variável (componente). Um ponto importante é a divisão do conjunto de restrições C em dois subconjuntos: C_{KB} e C_{UR}.

$C_{KB} = \{c_1, c_2, ..., c_q\}$ são as restrições relativas à base de conhecimento (*knowledge-base constraints*), instrísecas ao domínio do produto. A base de conhecimento KB consiste na união dos conjuntos $V \cup D \cup C_{KB}$, e contém as entidades do modelo mais as restrições sobre as mesmas. O espaço de solução S consiste no conjunto com todas as possíveis configurações válidas. As restrições contidas em C_{KB} são definidas na forma de expressões lógicas ou relacionais entre as variáveis de V, de forma a restringir o espaço de solução S e consequentemente o número de soluções (configurações) possíveis (FALKNER *et al.*, 2017).

Já o conjunto C_{UR} (*user requirement constraints*) contém restrições que representam as escolhas feitas pelo usuário durante o processo de configuração (a ser examinado na Seção 3.1.2). Diferente das expressões potencialmente complexas contidas em C_{KB}, as restrições de C_{UR} geralmente são atribuições simples que representam a escolha do usuário por uma determinada característica ou componente do produto (HOTZ *et al.*, 2014). Desta forma, quando o conjunto C_{UR} contiver uma atribuição válida e completa das variáveis de V, este será correspondente a um elemento do espaço de soluções S. "Válida" significa que nenhum valor atribuído a uma variável infringe qualquer restrição de C_{KB}, e "completa" significa que todas as variáveis estão com seus respectivos valores atribuídos (HOTZ *et al.*, 2014).

Sumarizando, a atividade de configurar um produto, modelada como um CSP, consiste na aplicação de um conjunto de requisitos do usuário C_{UR} a uma base de conhecimento KB. A menos que essa base de conhecimento esteja em processo de construção ou manutenção (ver Seção 3.2), assume-se que as restrições de C_{KB} sejam consistentes entre si, e que eventuais inconsistências no processo de configuração sejam sempre induzidas por restrições de C_{UR} (requisitos do usuário). Para facilitar o entendimento das definições apresentadas, providencia-se um exemplo de modelagem em CSP que ilustra um problema de configuração de bicicletas:

$$V = \{tipo, cor, marchas\}\,. \tag{1}$$

$$D = \{dom(tipo) = \{urbana, speed, montain\}, \\ dom(cor) = \{azul, preto, vermelho, branco\} \\ dom(marchas) = \{1, 18, 21\}\}. \tag{2}$$

$$C_{KB} = \{c_1 : tipo = urbana \rightarrow marchas = 1, \\ c_2 : tipo = speed \rightarrow marchas = 21, \\ c_3 : marchas = 1 \rightarrow tipo \neq mountain\}. \tag{3}$$

Na Equação (1), tem-se a definição dos componentes (variáveis) de um configurador de bicicletas, que suporta a escolha do tipo de bicicleta, cor e número de marchas. Em (2), define-se o domínio de cada variável de (1). A Equação (3) contém as restrições da base de conhecimento: c_1 define que toda bicicleta do tipo "urbana" é de marcha única, c_2 define que toda bicicleta do tipo "speed" tem câmbio de 21 marchas, e por fim c_3 define que as bicicletas do tipo "mountain" não podem ser de marcha única, portanto configuráveis tanto com câmbios de 18 quanto de 21 marchas. Observa-se que não há nenhuma restrição quanto à cor da bicicletas, logo estas podem receber qualquer cor de $dom(cor)$, independente do tipo ou do número de marchas. Uma possível solução para este problema de configuração poderia ser atingida a partir do seguinte conjunto de requisitos de usuário:

$$C_{UR} = \{r_1 : tipo = mountain, r_2 : cor = vermelho, r_3 : marchas = 18\}. \tag{4}$$

Verifica-se que nenhuma restrição $r_i \in C_{UR}$ contradiz quaisquer restrições de C_{KB}. Além disso, todas as variáveis de V foram instanciadas. Portanto, a configuração definida por C_{UR} é tanto válida quanto completa, e pode-se dizer que o processo de configuração foi solucionado. Mais detalhes sobre este processo serão apresentados na Seção 3.1.2.

3.1.2 O processo de configuração

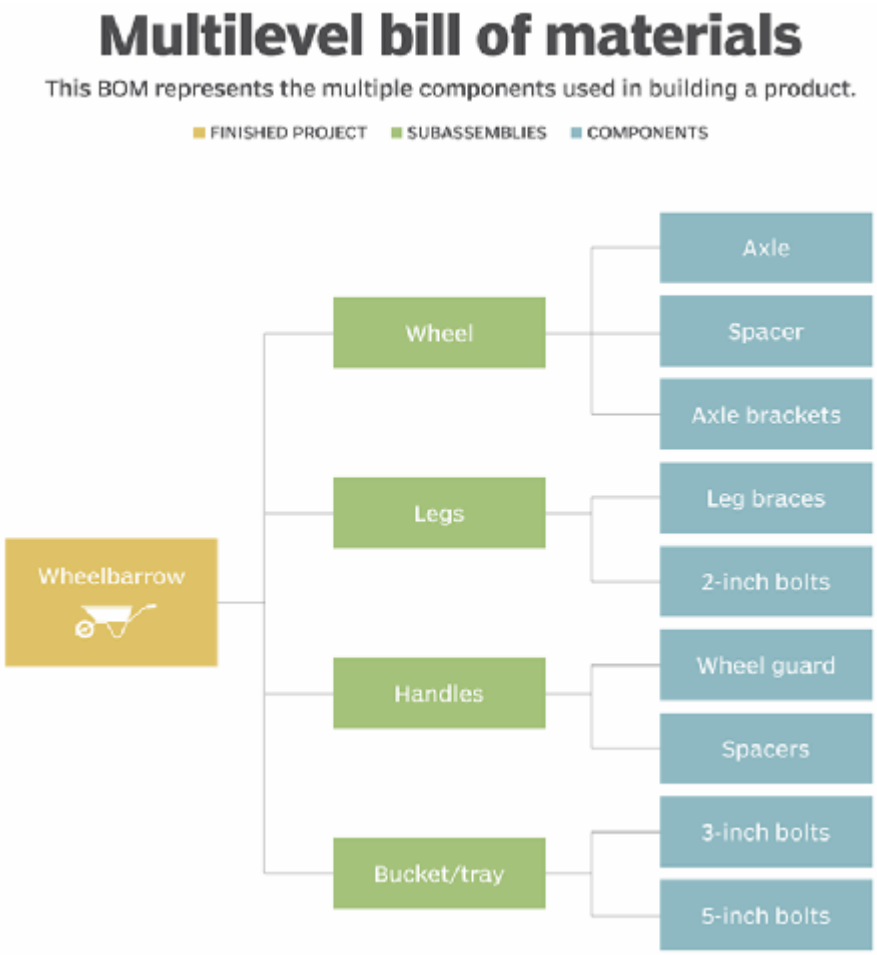
O processo de configuração, também chamado de tarefa de configuração (JUNKER, 2006), ocorre quando o usuário interage com o configurador de produtos para configurar o produto desejado. Geralmente ocorre numa interface gráfica, seja pelo navegador ou através de um *software* disponibilizado pela empresa. Ao iniciar a configuração, o conjunto de requisitos de usuário C_{UR} está vazio, pois ainda nenhuma característica do produto foi selecionada pelo mesmo (também é possível efetuar a inicialização de atributos com valores *default* - Seção 3.3.1).

A cada nova escolha do usuário, uma nova restrição de seleção, isto é, uma simples atribuição de valor a uma variável do produto, é instanciada e adicionada ao conjunto C_{UR}, e o processo de propagação de restrições, a ser detalhado na próxima

seção, é efetuado pelo configurador. Caso a seleção do usuário não leve a um estado de inconsistência (Seção 3.1.2.2), o processo de configuração continua até que todas as características necessárias do produto sejam definidas e o usuário confirme a configuração obtida.

Após a finalização de um processo de configuração, uma *Bill of Materials* (BOM) é gerada e encaminhada ao setor de produção. A BOM consiste numa estrutura padrão para modelar informacionalmente um produto, listando peças, materiais, partes e componentes necessários para manufaturá-lo, de forma a especificar a relação entre o produto final (já montado) e seus componentes. A BOM é processada por um sistema *Material Requirement Planning* (MRP), responsável por separar o que deve ser comprado daquilo que é produzido internamente, bem como planejar as atividades de manufatura, compra e entrega do produto. A Figura 5 mostra um exemplo de uma BOM multinível na forma de um grafo para a construção de um carrinho-de-mão.

Figura 5 – Exemplo de BOM.



Fonte – (KAKADE, 2021).

3.1.2.1 Propagação de restrições

A propagação de restrições, aqui referenciada simplesmente por “propagação”, é uma das técnicas de raciocínio possíveis para a resolução de CSPs (mencionadas no início desta seção), sendo a técnica empregada pela biblioteca de programação por restrições *Choco* (JUSSIEN *et al.*, 2008), utilizada neste projeto. A propagação de uma restrição é uma transformação que reduz o espaço de busca, sem modificar as soluções do problema (BESSIERE, 2006). Pode ser feita através da redução dos domínios das variáveis, fortalecimento de restrições ou criação de novas restrições auxiliares. A redução no espaço de busca facilita a resolução do problema e também serve para efetuar a verificação de que nenhuma restrição de C_{KB} foi violada com a adição de um requisito de usuário. Este último é um dos principais propósitos para sua utilização na configuração de produtos (JUNKER, 2006).

Como exemplo simples de propagação, considera-se um problema contendo duas variáveis, x_1 e x_2, ambas definidas com domínios contendo os números inteiros de 1..10, e uma restrição $c_1 : x_1 >= x_2$. Supondo que seja adicionado um requisito de usuário, definido por $r_1 : x_2 = 5$. Ao propagar essa restrição, altera-se o domínio de x_2 para 5, e o domínio de x_1 para 5..10. Desta forma, é possível ver como a manipulação do domínio das variáveis permite identificar quais são os valores a serem selecionados para x_1 que são consistentes com sua escolha para x_2, ou seja, quais as opções de seleção para os componentes restantes são consistentes com as escolhas previamente efetuadas. Esse tipo de indicação pode ser efetuada através de algum artifício da interface gráfica, que diferencie as seleções restantes que são consistentes (caminham em direção à solução do problema) daquelas que são inconsistentes (levam a um estado de inconsistência, a ser detalhado a seguir).

Formalmente, a propagação trabalha com a aplicação de consistência local, através da verificação de consistência de nodos, arcos e caminhos. Estes detalhes podem ser estudados em Bessiere (2006). Do ponto de vista da configuração de produtos, a função mais importante da técnica de propagação é aplicar as seleções efetuadas (interativamente) pelo usuário. A realização das operações necessárias para efetuar as propagações é delegada à biblioteca de programação por restrições.

3.1.2.2 Estado de inconsistência

Um estado de inconsistência, também conhecido como *over-constrained state* (JUNKER, 2004), é um estado passível de ser atingido durante uma sessão de configuração, em que há uma ou mais contradições lógicas entre as atribuições de valores definidas pelo usuário (representadas por restrições de C_{UR}) e as restrições da base de conhecimento C_{KB}. Como mencionado na Seção 3.1.1, assume-se que todas as restrições da base de conhecimento C_{KB} são consistentes entre si. Consequente-

mente, qualquer inconsistência é sempre induzida por uma seleção do usuário, ou seja, uma restrição $r_i \in C_{UR}$. Já um estado de consistência é simplesmente um estado sem nenhuma contradição entre C_{UR} e C_{KB}.

Ao entrar em um estado de inconsistência, o processo de configuração é efetivamente "travado", não permitindo a seleção de novos atributos do produto. A tarefa do configurador, então, é a de providenciar um meio para que o usuário possa sair desta situação de "nenhuma solução pode ser encontrada" e retomar o processo de configuração. Dada a natureza incremental deste processo, uma solução trivial seria simplesmente desfazer a última seleção efetuada, pois esta sempre terá participação no conflito que gerou o estado de inconsistência. Contudo, essa abordagem não funcionaria na prática, pois propõe desfazer uma escolha que o usuário acabou de realizar (mesmo sabendo ser uma escolha inconsistente, caso a interface indique isto). Assim, é necessário providenciar outra solução para indicar ao usuário qual ou quais seleções previamente efetuadas devem ser removidas para que a configuração retorne a um estado consistente. Também seria possível remover todas as seleções feitas pelo usuário, efetivamente reiniciando o processo, mas essa solução é bastante improdutiva.

É importante mencionar que existem técnicas avançadas de configuração que permitem o raciocínio mesmo na presença de inconsistências, como é apresentado em Nöhrer e Egyed (2010). Ao permitir que conflitos (inconsistências) existam durante o processo de configuração, abordagem conhecida como *living with conflicts*, o intuito é postegar a resolução de conflitos às etapas finais do processo de configuração, quando há uma maior interação do usuário com o sistema e dessa forma é possível entender melhor as intenções do mesmo, para então propor uma solução. Porém, são técnicas relativamente complexas de serem implementadas, e a solução mais observada, nos artigos estudados, é a de optar pela resolução imediata das inconsistências.

Principalmente no caso da configuração de produtos complexos, muitas vezes o usuário não tem ideia daquilo que realmente deseja (HOTZ *et al.*, 2014), e também é comum que possa mudar suas intenções para determinadas características do produto múltiplas vezes durante uma mesma sessão de configuração. Nesses casos, a propagação de variáveis se mostra especialmente útil, pois permite ao configurador identificar previamente e apresentar ao usuário quais seleções que levam a um estado de inconsistência. Visando impedir completamente o alcance de tais estados, poder-se-ia, após cada propagação, impedir a seleção de opções inconsistentes, mas a desvantagem dessa abordagem é que, caso o usuário tenha qualquer ressalva a fazer durante o processo de configuração, seria necessário que ele desfizesse manualmente suas seleções prévias até conseguir uma combinação que o permita efetuar a seleção desejada.

A partir dos pontos mencionados, conclui-se que é desejável permitir o atingimento de estados de inconsistência durante sessões de configuração. Contudo, para

viabilizar isto, é necessário prover ao configurador recursos para que saia de estados de inconsistência, e a forma convencional de se fazê-lo é através da implementação de algoritmos para cálculo de diagnósticos e detecção de conflitos, a serem vistos na próxima seção.

3.2 CÁLCULO DE DIAGNÓSTICOS

O cálculo de diagnósticos é uma técnica que permite determinar quais elementos de C_{UR} são responsáveis pelo atingimento do estado de inconsistência, para que o configurador possa então indicar ao usuário qual ou quais seleções prévias devem ser removidas para que o processo de configuração possa ser retomado. Também pode ser utilizado para verificar se as restrições que modelam o produto são consistentes entre si, como investigado em Felfernig *et al.* (2004) e Felfernig *et al.* (2006), mas esta aplicação está fora do escopo deste trabalho.

Para facilitar a compreensão do cálculo de diagnósticos, utiliza-se como exemplo o cenário introduzido no artigo Felfernig *et al.* (2013b), que consiste num configurador de automóveis, domínio comumente usado como exemplo também em outros artigos da área. Segue a definição do modelo em CSP:

$$V = \{type, fuel, skibag, 4\text{-}wheel, pdc\}, \tag{5}$$

$$\begin{aligned} D = \{&dom(type) = \{city, limo, combi, xdrive\}, \\ &dom(fuel) = \{4l, 6l, 10l\} \\ &dom(skibag) = \{yes, no\} \\ &dom(4\text{-}wheel) = \{yes, no\} \\ &dom(pdc) = \{yes, no\}\,. \end{aligned} \tag{6}$$

O modelo tem cinco variáveis que representam cinco componentes relevantes para o configurador de automóveis: *type* define o tipo do veículo, *fuel* define o consumo de combustível a cada 100*km*, *skibag* define a presença de um porta-esqui dentro do carro, 4-*wheel* define se a tração é 4x4, e *pdc* (*park distance control*) define se a inclusão do sensor de estacionamento. As variáveis representam todos os possíveis requisitos do usuário, e as combinações possíveis das mesmas são restringidas pela aplicação das restrições da base de conhecimento, definidas por:

$$\begin{aligned} C_{KB} = \{&c_1 : 4\text{-}wheel = yes \Rightarrow type = xdrive, \\ &c_2 : skibag = yes \Rightarrow type \neq city, \\ &c_3 : fuel = 4l \Rightarrow type = city, \\ &c_4 : fuel = 6l \Rightarrow type \neq xdrive, \\ &c_5 : type = city \Rightarrow fuel \neq 10l\}\,. \end{aligned} \tag{7}$$

Um ponto importante a ser analisado é que todas as restrições de C_{KB} são consistentes entre si, isto é, não entram em contradição. Além disso, no início de um processo de configuração, as restrições não forçam uma escolha *a priori* de nenhum componente, isto é, ao aplicar C_{KB} sobre V e D e efetuar uma propagação, o domínio das variáveis permanece o mesmo (D não sofre alterações).

O artigo ainda providencia como exemplo uma sessão de configuração, onde supõe que o usuário corrente tenha selecionado valores para cada uma das cinco variáveis e entrado em um estado de inconsistência. O conjunto de requisitos do usuário fornecido é dado por

$$\begin{aligned} C_{UR} = \{ & c_6 : 4\text{-}wheel = yes, \\ & c_7 : fuel = 6l, \\ & c_8 : type = city, \\ & c_9 : skibag = yes, \\ & c_{10} : pdc = yes \}. \end{aligned} \tag{8}$$

Nota-se uma pequena diferença de notação: no artigo, os requisitos do usuário são definidos a partir de c_{q+1}, onde q é o número da última restrição de C_{KB}, enquanto na notação introduzida na Seção 3.1.1 deste capítulo, os requisitos do usuário foram definidos utilizando a letra r.

Ao analisar o conjunto C_{UR}, é possível inferir a presença de contradições, ou seja, que alguns requisitos do usuário são inconsistentes com a base de conhecimento C_{KB}. Por exemplo, c_6 está em conflito com c_8, pois uma das implicações da restrição c_1 é de que a escolha por 4-*wheel* (tração 4x4) implica no tipo do automóvel ser *xdrive*. Nota-se que todas as cinco variáveis foram instanciadas, portanto a configuração é completa. Se nem tivessem sido instanciadas, a configuração seria incompleta, mas mesmo assim ainda poderia haver conflitos no conjunto C_{UR}.

Mesmo em modelos simples como este do artigo, pode ser difícil analisar a satisfação das restrições de C_{KB} sem o auxílio de algoritmos. Para identificar quais restrições de C_{UR} devem ser removidas para que o processo retorne a um estado consistente, é necessário efetuar o cálculo de diagnósticos (Seção 3.2). Porém, antes de apresentar os algoritmos que efetuam este processo, introduz-se a seguir a definição do conceito de conflitos no contexto de uma sessão de configuração em estado de inconsistência.

3.2.1 Conjuntos de mínimos conflitos

Adaptando a definição apresentada em Hotz *et al.* (2014) para o contexto e notação utilizados neste documento, um conjunto de conflitos $CS = \{c_a, c_b, ..., c_n\}$ é um subconjunto de C_{UR} tal que $C_{KB} \cup CS$ seja inconsistente. Essa verificação é

denominada como “propriedade de conflito”. Reitera-se que as restrições de C_{KB} são consistentes entre si, e que C_{UR} representa o conjunto de requisitos do usuário que gerou uma ou mais contradições durante o processo de propagação. Um conjunto de conflito *CS* é mínimo sse. não exista $CS' \subset CS$ que satisfaça a propriedade de conflito.

No exemplo de configuração de automóveis de Felfernig *et al.* (2013b), com a aplicação dos requisitos dados pela Equação 8, os seguintes conjuntos de conflitos podem ser identificados: $CS_1 = \{c_6, c_7\}$, $CS_2 = \{c_8, c_9\}$, e $CS_3 = \{c_6, c_8\}$. Esses conjuntos não permitem a determinação de uma solução de configuração, pois $CS_1 \cup C_{KB}$, $CS_2 \cup C_{KB}$ e $CS_3 \cup C_{KB}$ são todos inconsistentes. Também são conjuntos mínimos, visto que nenhum de seus possíveis subconjuntos são inconsistentes com C_{KB}. Logo, os conjuntos *CS* podem ser, mais especificamente, denominados como *conjuntos de mínimos conflitos* (HOTZ *et al.*, 2014).

Nota-se que cada conjunto de mínimo conflito pode ser resolvido simplesmente pela remoção de um de seus elementos. Esse fato é considerado pelo algoritmo que calcula os diagnósticos, apresentado na Seção 3.2.2, cujo objetivo é determinar um conjunto de diagnósticos (a não ser confundido com conjunto de mínimos conflitos) que contenha os requisitos do usuário a serem removidos para que todos os *CS* sejam resolvidos.

Na situação-exemplo, um possível conjunto de diagnósticos é

$$\text{diagnósticos} = \{\Delta_1, \Delta_2, \Delta_3\}, \tag{9}$$

onde $\Delta_1 = \{c_6, c_8\}$, $\Delta_2 = \{c_6, c_9\}$, e $\Delta_3 = \{c_7, c_8\}$. A remoção de requisitos, sugerida por cada diagnóstico Δ, resolve todos os conflitos *CS*. Por esta razão, também é possível chamar esse conjunto de “alternativas de reparo” do estado inconsistente. O processo de cálculo que determinou o conjunto será apresentado após a introdução dos algoritmos.

Um ponto importante a ser observado é que existem dois algoritmos a serem utilizados: o algoritmo que calcula diagnósticos, HSDAG, introduzido por Reiter *et al.* (1987), e o algoritmo que determina conjuntos de mínimos conflitos, *QuickXplain*, introduzido por Junker (2004). No processo de resolução de um estado de inconsistência, o algoritmo HSDAG é executado apenas uma vez, e invoca o *QuickXPlain* diversas vezes durante sua execução. Isso acontece porque o *QuickXPlain* retorna apenas um conjunto de mínimos conflitos por computação. Desta forma, para identificar todos os conjuntos de mínimos conflitos (e elaborar o conjunto de diagnósticos a partir dos mesmos), é necessário combinar os dois algoritmos, como apresentado a seguir.

3.2.2 Algoritmo HSDAG para o cálculo de diagnósticos

O algoritmo *Hitting Set Directed Acyclic Graph* (HSDAG) é um algoritmo de busca em largura (*breadth-first search*) que efetua a determinação de diagnósticos.

Devido à busca ser em largura, os diagnósticos determinados são de mínima cardinalidade, isto é, diagnósticos mínimos com o menor número possível de restrições $r_i \in C_{UR}$ inclusas (HOTZ *et al.*, 2014). O algoritmo também é completo, pois retorna todos os diagnósticos contidos em C_{UR}, e consiste na construção de um grafo acíclico dirigido por *hitting sets*. *Hitting sets* são efetivamente os diagnósticos, conjuntos que acertam (do inglês *hit*) todos os conjuntos de mínimos conflitos *CS*. A estrutura do grafo é estabelecida por repetidas ativações de um algoritmo de determinação de conflitos (no caso, o *QuickXPlain*), e da análise das diferentes possibilidades de resolver o conjunto retornado.

A Figura 6 contém o pseudo-código para a construção do HSDAG. O parâmetro *n* é opcional, e permite impor uma condição de parada, caso deseja-se limitar o número de diagnósticos calculados. A função *theorem prover* (TP) representa o algoritmo auxiliar que determina e retorna os conjuntos de mínimos conflitos. No caso deste projeto e do artigo de onde foi retirada a figura, o TP é o *QuickXPlain*.

Figura 6 – Algoritmo HSDAG.

Algorithm HSDAG($C_{\mathrm{u}R}$,C_{KB}, n): Δ

HSDAG returns $\leq n (n \geq 0)$ minimal diagnoses Δ (bag)
for a given set of inconsistent user requirements ($C_{\mathrm{u}R}$)
{C_{KB}: configuration knowledge base}
{H: paths of the diagnosis search tree}
{CS: min. conflict set returned by theorem prover (TP)}
$\Delta = \emptyset$; $H = \emptyset$;
repeat
  $\Delta' \leftarrow first(H)$;
  $CS \leftarrow TP((C_{\mathrm{u}R} - \Delta') \cup C_{KB})$;
  if $isEmpty(CS)$ **then**
    $\Delta \leftarrow \Delta \cup \Delta'$;
    $H \leftarrow deleteall(\Delta', H)$;
    n ← n - 1;
  else
    for all X in CS **do**
      $H \leftarrow H \cup \{\Delta' \cup \{X\}\}$;
    end for
    $H \leftarrow delete(\Delta', H)$;
  end if
until (H = $\emptyset$ or n = 0);
return Δ;

Fonte – Adaptado de (FELFERNIG *et al.*, 2013b).

A seguir, o algoritmo *QuickXPlain* é apresentado, então será exibido o grafo de

busca construído ao realizar a aplicação do HSDAG aos C_{UR} e C_{KB} definidos nas equações 8 e 7, respectivamente, o qual deve determinar o conjunto de diagnósticos descrito pela Equação 9.

3.2.3 Algoritmo QuickXPlain para determinação de mínimos conflitos

O *QuickXPlain* (JUNKER, 2004) é um algoritmo usado para a determinação de conjuntos de mínimos conflitos, baseado no conceito de "divisão e conquista". A ideia é que, sempre que for detectado que a primeira mentade de um conjunto de restrições *C* a ser analisada é inconsistente, a segunda metade deste conjunto é omissa, ou seja, desconsiderada do processo de determinação de conflitos. Essa abordagem é eficiente para desconsiderar restrições (requisitos do usuário) que não participam do conflito (HOTZ *et al.*, 2014).

A Figura 7 mostra o pseudo-código do *QuickXPlain*. Considerando a notação introduzida neste documento e o contexto de resolução de estados de inconsistência durante processos de configuração, *C* pode ser substituído por C_{UR}. *AC* (*all constraints*) trata-se de C_{UR} + C_{KB}, e *B* (*background constraints*) é equivalente a C_{KB}. *CS* é o conjunto de mínimos conflitos a ser retornado pelo algoritmo (apenas um conjunto é retornado a cada execução).

Figura 7 – Algoritmo QuickXPlain.

Algorithm 7.2 – QUICKXPLAIN

```
1  func QUICKXPLAIN(C ⊆ AC, B = AC − C) : CS
2  if isEmpty(C) or inconsistent(B) return ∅;
3  else return QX(∅, C, B);

4  func QX(D, C = {c_1..c_q}, B) : CS
5  if D ≠ ∅ and inconsistent(B) return ∅;
6  if singleton(C) return C;
7  k = ⌈q/2⌉;
8  C_1 = {c_1..c_k}; C_2 = {c_{k+1}..c_q};
9  CS_1 = QX(C_2, C_1, B ∪ C_2);
10 CS_2 = QX(CS_1, C_2, B ∪ CS_1);
11 return(CS_1 ∪ CS_2);
```

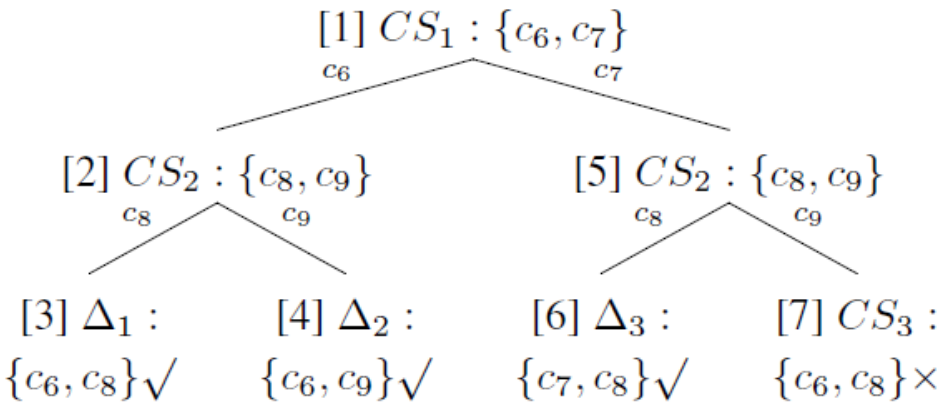
Fonte – (HOTZ *et al.*, 2014).

Ao aplicar o HSDAG, aliado ao *QuickXPlain*, para realizar o cálculo de diagnósticos para o estado de inconsistência descrito pelos C_{UR} e C_{KB} das equações 8 e 7, o grafo de busca da Figura 8 é construída. A ordem de exploração é identificada pelos

números entre colchetes. Em cada nodo, temos o conjunto de mínimos conflitos *CS* retornado pelo *QuickXPlain*.

O primeiro *CS* retornado é $CS_1 = \{c_6, c_7\}$. Deste, criam-se duas arestas: uma representa a remoção do elemento c_6, e a outra a remoção de c_7. Por convenção, a aresta da esquerda é explorada primeiro. Aplica-se o *QuickXPlain* considerando a remoção de c_6, e $CS_2 = \{c_8, c_9\}$ é retornado, então cria-se uma aresta para cada elemento de CS_2. A primeira aresta é verificada ao aplicar o *QuickXPlain* considerando a remoção de c_6 e c_8 (contidos no caminho), que retorna um conjunto vazio, indicando que nenhuma consistência foi detectada. Logo, $\{c_6, c_8\}$ é o primeiro diagnóstico a ser determinado. O mesmo repete-se para o caminho $c_6 \rightarrow c_9$, que resulta na determinação de $\Delta 2$.

Figura 8 – Árvore de busca gerada pelo cálculo de diagnósticos.



Fonte – (FELFERNIG *et al.*, 2013b).

Após a determinação de $\Delta 2$, o algoritmo retorna ao primeiro nodo e o caminho representado por c_7 é explorado. No nodo [5], obtém-se novamente CS_2, e deste criam-se duas arestas. Na aresta da esquerda, verifica-se a remoção de c_7 e c_8, obtendo-se $\Delta 3$. Na aresta da direita, verifica-se que o conjunto CS_3 retornado já está contido num dos caminhos explorados ($c_6 \rightarrow c_8$), portando o nodo [7] é encerrado.

Com essa execução do HSDAG, o processo de cálculo de diagnósticos é concluído, e as três alternativas de reparo, representadas pela Equação 9, devem ser apresentadas para a escolha do usuário. Δ_1 indica a remoção de c_6 : 4-*wheel* = *yes* e c_8 : *type* = *city*, Δ_2 indica a remoção de c_6 : 4-*wheel* = *yes* e c_9 : *skibag* = *yes*, e Δ_3 indica a remoção de c_7 : *fuel* = 6*l* e c_8 : *type* = *city*. A aplicação de qualquer um dos diagnósticos resolve todos os mínimos conflitos, e retorna a aplicação para um estado consistente.

Um ponto importante a ser observado é que a geração do grafo com o HSDAG, ainda sem recomendação, não segue nenhuma heurística, seguindo a convenção de

uma busca em largura convencional. Dependendo da ordem em que o usuário define seus requisitos, o primeiro *CS* retornado poderia ser CS_2, por exemplo, e a ordem de determinação de diagnósticos teria sido diferente. A ideia da recomendação de diagnóstico, como proposta em Felfernig *et al.* (2013b), está em adicionar uma heurística nesse processo de busca, de forma a obter os diagnósticos de maneira ordenada, onde os primeiros diagnósticos são considerados de maior relevância ao usuário corrente. A técnica será apresentada na Seção 3.3.2, após uma breve panorama sobre a integração de tecnologias de recomendação em configuradores de produtos.

3.3 INTEGRAÇÃO DE TECNOLOGIAS DE RECOMENDAÇÃO

Com o aumento da complexidade de customização dos produtos oferecidos, surgiu uma demanda por técnicas inteligentes que proativamente auxiliam o usuário de aplicações de configuração a encontrar uma solução que anteda às suas necessidades. Segundo Hotz *et al.* (2014), um efeito colateral da oferta de produtos muito customizáveis é que o excesso de complexidade pode sobrecarregar a capacidade do usuário de explorar o espaço de solução, dificultando sua chegada a uma decisão. Logo, tecnologias de recomendação são propícias de serem aplicadas no contexto de configuração, porque permitem a busca por itens relevantes em conjuntos grandes e complexos.

De acordo com Felfernig e Stettinger (2015), sistemas de recomendação tem o potencial de melhorar os processos de negócios, por exemplo, através da redução de erros durante o registro de pedidos, ou pela redução de esforços ao prestar assessoria aos clientes. Além disso, ao interagir com estes sistemas, o usuário pode ganhar, por conta própria, um maior entendimento sobre o domínio do produto, e como consequência, menor esforços são necessários quanto a explicação de aspectos básicos dos produtos a serem configurados.

Existem diferentes tipos de tecnologias de recomendação que podem ser aplicadas em configuradores de produtos, seja para a recomendação de configurações completas ou para indicar a seleção de atributos individuais ao usuário. A seguir, são apresentadas as abordagens de recomendação de *defaults* e ranqueamento, e então a recomendação de alternativas de reparo, sendo o foco deste trabalho.

3.3.1 Recomendação de *defaults* e ranqueamento

Um *default* (do inglês para “padrão”) é uma característica do produto que é préselecionada pelo configurador, e representa alternativas recomendadas, significativas ou compatíveis com as preferências do usuário atual (HOTZ *et al.*, 2014). Um dos objetivos ao usar esta técnica é permitir que o usuário necessite especificar apenas as características que são importantes para ele, cabendo ao sistema determinar automa-

ticamente os valores restantes (FALKNER *et al.*, 2011). Listam-se algumas diferentes abordagens para efetuar a recomendação de *defaults*:

- Recomendação estática de *defaults*: a recomendação estática pode ser feita para parâmetros cuja recomendação de valores independe do contexto de configuração. Pressupõe-se que estes valores sejam sempre desejados pelos usuários. McSherry (2005) introduziu a ideia de efetuar a recomendação estática usando métricas que consideram, por exemplo, se é desejável que determinado atributo seja maximizado (*more-is-better*, e.g., eficiência energética), minimizado (*less-is-better*, e.g., preço), dentre outras possibilidades.
- Recomendação de *defaults* baseada em regras: necessita da definição explícita de regras, acopladas ao domínio do produto, que determinam recomendações dependendo do contexto de configuração corrente (FALKNER *et al.*, 2011). Nessa abordagem, os *default* são determinados com base em requisitos que já foram especificados pelo usuário. Um exemplo, resgatando o configurador de bicicletas introduzido na Seção 3.1.1, seria a incorporação da seguinte regra: “se *uso* for definido como *infantil*, recomenda-se a escolha de um par de rodinhas auxiliares”.
- Recomendação colaborativa de *defaults*: nesta abordagem, o sistema efetua recomendação de valores para características do produto que ainda não foram selecionadas pelo usuário, a partir da exploração de informações sobre sessões prévias de configuração (efetuadas por outros usuários), sendo comum a aplicação de algoritmos de classificação como o *k-Nearest-Neighbors* (kNN) (FELFERNIG *et al.*, 2010). Essa abordagem é semelhante ao que foi implementado neste trabalho, porém recomenda a seleção de características que ainda não foram definidas pelo usuário, enquanto neste trabalho efetua-se a recomendação da remoção de certas características em estados de inconsistência.

Além da recomendação de *defaults*, outra aplicação de tecnologias de recomendação em configuradores de produtos é o ranqueamento de resultados, que se mostra útil quando existem muitas alternativas de solução para um dado conjunto de requisitos de usuário. Nessa situação, é interessante prover alguma heurística de ranqueamento para destacar as soluções que sejam consideradas de maior relevância ao usuário (HOTZ *et al.*, 2014).

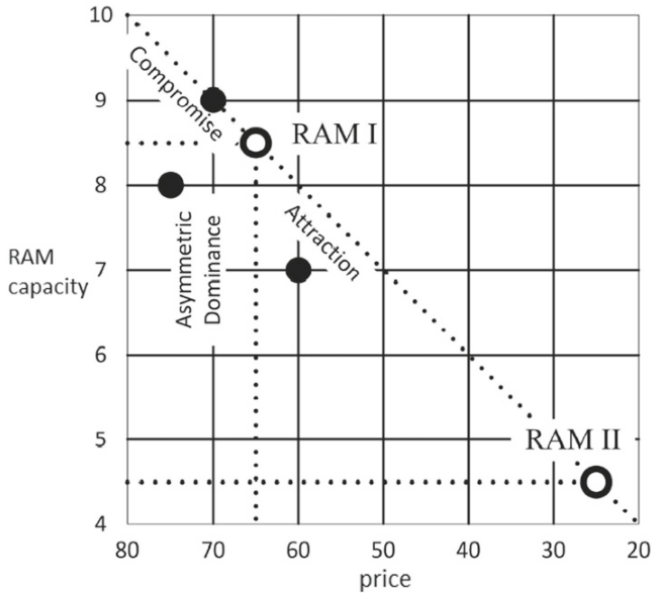
Sobre a interação de usuários com interfaces, existem alguns fenômenos da psique humana que devem ser considerados durante o desenvolvimento de sistemas de suporte à decisão (recomendação). Alguns destes fenômenos, abordados em Mandl *et al.* (2014), são:

- *Decoy effects*: consiste na adição de configurações inferiores ou superiores na lista de produtos, visando influenciar o processo decisório do usuário. A Figura 9

mostra diferentes categorias de itens *decoy* (chamarizes), discutidos em Mandl *et al.* (2014), usando como domínio-exemplo a configuração de computadores, e relacionando a capacidade de memória RAM com o preço.

- *Serial position effects*: trata do estudo de como o posicionamento de determinados itens em uma lista influencia na decisão do usuário. Um efeito observado por Crowder (2014) é o chamado *primacy effect*, que evidencia que usuários estão mais propensos a explorar itens que aparecem no início ou no final de listas do que aqueles que estão em posições medianas.
- *Status quo effect*: segundo Mandl *et al.* (2014), pesquisas no processo decisório de pessoas indicam que elas possuem uma forte tendência a manter o *status quo* ao terem que efetuar escolhas dentre alternativas, e isto aplica-se ao contexto da configuração de produtos. Uma consequência é que decisões propostas por especialistas (ou pelo próprio configurador) são aceitas com maior facilidade pelo usuário.

Figura 9 – Diferentes tipos de *decoy effects*.



Fonte – (MANDL *et al.*, 2014)

Por fim, a última aplicação de tecnologias de recomendação em configuradores de produtos a ser apresentada é a recomendação de diagnósticos, cuja implementação e demonstração é o objetivo da aplicação a ser desenvolvida.

3.3.2 Recomendação de diagnósticos

Quando uma sessão de configuração entra em estado de inconsistência e nenhuma solução pode ser determinada para o conjunto de requisitos definidos pelo usuário, a ação recomendada é realizar o cálculo de diagnósticos, e apresentar os resultados ao usuário na forma de alternativas de reparo (FELFERNIG *et al.*, 2013a). A abordagem convencional para realizar o cálculo de diagnósticos foi apresentada na Seção 3.2, determinando diagnósticos de mínima-cardinalidade sem a integração com tecnologias de recomendação (FELFERNIG *et al.*, 2004).

Felfernig *et al.* (2009) afirma que, especialmente no caso de produtos complexos, é bastante comum que um grande número de diagnósticos alternativos sejam possíveis, e a recomendação de diagnósticos, também conhecida como “personalização de diagnósticos” ou ”recomendação de alternativas de reparo”, busca identificar quais são as alternativas mais relevantes ao usuário do configurador. Geralmente, o usuário não quer ou não está apto a avaliar conjuntos muitos grandes de alternativas. Além disso, pode ser impraticável a determinação de todos os diagnósticos possíveis, devido a tempos de execução inaceitáveis dos algoritmos que calculam diagnósticos. Por esta razão, existe uma demanda por abordagens que ajudam a ordenar e/ou reduzir as alternativas de diagnósticos. Segundo Felfernig *et al.* (2013b), o objetivo ao empregar técnicas de recomendação de diagnósticos é o de manter o processo de avaliação e seleção de diagnósticos o mais simples o possível.

A primeira abordagem para melhorar o tempo de execução do cálculo de diagnósticos é limitar, estaticamente, o número de diagnósticos a serem calculados e apresentados ao usuário. Porém, esta abordagem não garante que os diagnósticos apresentados sejam representativos do ponto de vista do usuário. Uma solução melhor, proposta por O’Sullivan *et al.* (2007), é limitar dinamicamente a quantidade de alternativas que são exibidas ao usuário, através da implementação do conceito de *representative explanations* (do inglês para “explicações representativas”), que consiste num critério a ser satisfeito, onde cada elemento contido em pelo menos um diagnóstico, considerando todos os possíveis, também deve estar contido no conjunto de diagnósticos a ser apresentado ao usuário.

Felfernig *et al.* (2013b) mostra como integrar técnicas de recomendação para realizar a determinação personalizada de diagnósticos. Para tal, propõe realizar uma alteração no algoritmo HSDAG, de forma a permitir a adição de heurísticas para guiar a construção do grafo durante o processo de cálculo de diagnósticos. O algoritmo resultante, mostrado na Figura 10, foi chamado de *PDiag* pelos autores do artigo.

Ao comparar com o *PDiag* com o HSDAG convencional (apresentado na Seção 3.2.2, ver Figura 6), nota-se a simples adição de uma função $sort(H, crit)$ no final do *loop* de busca. O objetivo dessa função é reordenar os caminhos H da árvore de busca a partir de um determinado critério *crit*. Para implementação deste critério, o

Figura 10 – Algoritmo PDiag.

Algorithm 1 PDIAG($C_{\mathrm{u}R}$,C_{KB}, $crit$, n): Δ

{PDIAG returns $\leq n (n \geq 0)$ personalized minimal diagnoses Δ (bag) for a given set of inconsistent user requirements ($C_{\mathrm{u}R}$)using the preference criteria defined in $crit$.}
{C_{KB}: configuration knowledge base}
{H: paths of the diagnosis search tree}
{CS: min. conflict set returned by theorem prover (TP)}
$\Delta = \emptyset$; $H = \emptyset$;
repeat
  $\Delta' \leftarrow first(H)$;
  $CS \leftarrow TP((C_{\mathrm{u}R} - \Delta') \cup C_{KB})$;
  if $isEmpty(CS)$ **then**
    $\Delta \leftarrow \Delta \cup \Delta'$;
    $H \leftarrow deleteall(\Delta', H)$;
    n ← n - 1;
  else
    for all X in CS **do**
      $H \leftarrow H \cup \{\Delta' \cup \{X\}\}$;
    end for
    $H \leftarrow delete(\Delta', H)$;
    $H \leftarrow sort(H, crit)$;
  end if
until (H = $\emptyset$ or n = 0);
return Δ;

Fonte – Adaptado de (FELFERNIG *et al.*, 2013b)

artigo apresenta quatro abordagens: diagnóstico baseado em similaridade (*similarity-based*), diagnóstico baseado em utilidade (*utility-based*), diagnóstico baseado em probabilidade (*probability-based*) e diagnóstico baseado em conjunto (*ensemble-based*). Destas, apenas o diagnóstico baseado em similaridade, a ser detalhado a seguir, foi implementado neste trabalho. As demais abordagens não foram implementadas, pois, além da indisponibilidade de tempo, necessitariam da implementação de funcionalidades complementares cuja adição ao configurador da empresa não era planejada, no momento da realização do estágio.

3.3.2.1 Personalização de diagnósticos baseada em similaridade

Assim como a recomendação colaborativa de *defaults* (Seção 3.3.1), a personalização de diagnósticos baseada em similaridade, proposta por Felfernig *et al.* (2013b), necessita de uma base de dados que contenha informações sobre as configurações

que foram realizadas em sessões prévias, por outros usuários do sistema. A ideia é utilizar essas configurações para guiar o cálculo de diagnósticos, de forma que os diagnósticos obtidos estejam ordenados de acordo com sua relevância para o usuário corrente.

Para exemplificar, supõe-se que um determinado conjunto de requisitos de usuário $C_{UR} = \{r_1, r_2\}$, ao ser aplicado na configuração de um produto, gere um estado de inconsistência. Efetua-se então o cálculo de diagnósticos de maneira convencional, e são obtidos, nesta ordem, os diagnósticos $\Delta_1 = \{r_1\}$ e $\Delta_2 = \{r_2\}$. Se a aplicação de Δ_1 leva a uma configuração muito diferente das configurações previamente efetuadas, enquanto a aplicação de Δ_2 leva a uma configuração similar a uma configuração previamente efetuada por outro usuário, a aplicação de diagnóstico baseado em similaridade faria com que Δ_2 fosse determinado antes de Δ_1. Logo, Δ_2 seria a primeira sugestão de reparo apresentada ao usuário, com a justificativa de que Δ_2 é mais relevante do que Δ_1, porque retorna a um estado de configuração mais similar (segundo as métricas que foram utilizadas) a uma configuração previamente efetuada e finalizada no sistema, que é provavelmente parecida ou igual a que o usuário da sessão atual deseja chegar.

No exemplo dado, com apenas dois diagnósticos, a utilidade da ordenação baseada em similaridade não fica evidente. Porém, no contexto de produtos complexos, o número de diagnósticos pode chegar a algumas dezenas (FELFERNIG *et al.*, 2009). Efetuar a ordenação de diagnósticos e fornecer justificativas ao usuário auxilia na seleção por uma alternativa de diagnóstico, assim melhorando a experiência de configuração. A justificativa poderia ser feita, por exemplo, pela apresentação dos valores de similaridades calculados para cada diagnóstico.

3.3.2.1.1 Base de configurações completas

A recomendação de diagnósticos baseada em similaridade é pautada na ideia de obter diagnósticos que levam a configurações semelhantes às que já foram feitas por outros usuários. Para tal, é necessária a elaboração e manutenção de uma base de configurações completas, a qual é utilizada para a construção de uma tabela de similaridade, onde, para cada registro de configuração completa s_i, calcula-se sua similaridade com o conjunto de requisitos do usuário da sessão atual. Nota-se que, como C_{UR} é inconsistente, e toda s_i é completa e válida, nenhuma configuração s_i terá 100% de similaridade com C_{UR}.

Na prática, quando a base ainda não foi suficientemente populada, podem surgir problemas de *cold-start* (JANNACH *et al.*, 2010), muito comuns em sistemas de recomendação, quando não se tem dados em quantidade ou qualidade suficiente para efetuar recomendações úteis ao usuário. Para contornar esse problema, é uma solução é simplesmente desativar o módulo de recomendação para os primeiros usuários do configurador, registrando as configurações que são finalizadas, até que a base

fique suficientemente preenchida. Outra opção seria popular artificialmente a base de configurações, antes de efetuar o *deploy* do configurador.

Retomando o configurador de automóveis, introduzido na Seção 3.2, definido pelas Equações 5, 6, 7 e 8, Felfernig *et al.* (2013b) providencia uma base de configurações completas com uma tabela contendo oito registros (Tabela 3), onde cada sessão s_i representa uma sessão de configuração previamente finalizada por um usuário do sistema.

Tabela 3 – Base de configurações completas para o configurador de automóveis.

SESSION s_i	TYPE	FUEL	SKIBAG	4-WHEEL	PDC
s_1	city	4l	no	no	yes
s_2	city	4l	no	no	no
s_3	xdrive	10l	yes	yes	yes
s_4	limo	6l	no	no	yes
s_5	combi	6l	no	no	no
s_6	xdrive	10l	no	yes	yes
s_7	limo	6l	yes	no	no
s_8	combi	6l	yes	no	no

Fonte – (FELFERNIG *et al.*, 2013b).

Antes de mostrar a tabela de similaridade que foi calculada no artigo, é necessário introduzir as métricas de similaridade, as quais são utilizadas na construção da tabela.

3.3.2.1.2 Métricas de similaridade

Os valores de similaridade são determinados com base em quatro métricas em nível de atributo (MCSHERRY, 2005), da mesma forma como é feito na recomendação estática de *defaults* (Seção 3.3.1). A métrica de similaridade em nível de atributo consiste numa função $s(a_i, c_i)$, que determina a similaridade entre cada atributo a_i da sessão s_k e o correspondente requisito do usuário $c_i \in C_{UR}$. Para cada atributo do produto, uma das seguintes métricas de similaridade deve ser utilizada, dependendo de seu tipo:

$$MIB : s(a_i, c_i) = \frac{a_i - min(a_i)}{max(a_i) - min(a_i)}, \tag{10}$$

$$LIB : s(a_i, c_i) = \frac{max(a_i) - val(a_i)}{max(a_i) - min(a_i)}, \tag{11}$$

$$NIB : s(a_i, c_i) = 1 - \left| \frac{val(c_i) - val(a_i)}{max(a_i) - min(a_i)} \right| , \tag{12}$$

$$EIB : s(a_i, c_i) = \begin{cases} 1, & \text{if } val(c_i) = val(a_i) \\ 0, & \text{otherwise} \end{cases} . \tag{13}$$

A Equação (10) define a métrica *More-Is-Better* (MIB), usada para atributos que o usuário deseja maximizar, como, por exemplo, a *eficiência energética*. A Equação (11) define a métrica *Lower-Is-Better* (LIB), para atributos que o usuário deseja minimizar, como o *preço*. A Equação (12) define a métrica *Near-Is-Better* (NIB), usada para atributos que devem ser similares ao requisito do usuário, como a *potência* desejada para um motor. E por fim, a Equação (13) define a métrica *Equal-Is-Better* (EIB), usada para calcular a similaridade de atributos não-numéricos, selecionáveis dentro de um conjunto de opções distintas (enumerações).

Observa-se que as três primeiras métricas, MIB, LIB e NIB têm aplicação destinada a atributos numéricos, isto é, que seguem uma escala de valor intrínseca. No domínio do configurador de automóveis, cujas variáveis são *type*, *fuel*, *skibag*, 4-*wheel* e *pdc*, o único atributo numérico é *fuel* (consumo de combustível), que usa a métrica LIB no cálculo de similaridade. Os demais atributos são enumerações, e utilizam a métrica EIB.

A similaridade total, descrita pela Equação (14), entre a configuração definida por $c = C_{UR}$ e a tupla $a = s_k$ da tabela de configurações completas é calculada por uma soma ponderada das similaridades em nível de atributo. Segundo Felfernig *et al.* (2013b), a importância dos atributos é tipicamente baseada em *Multi-Attribute Utility Theory* (MAUT) (WINTERFELDT; EDWARDS, 1986), porém, caso esta informação não esteja disponível, os atributos podem ser definidos com pesos homogêneos, $w(c_i) = 1/num(atributos)$, de forma que o somatório resulte em 100%.

Uma observação importante é que, caso o C_{UR} da sessão corrente não fosse completo, isto é, algumas variáveis não tivessem sido definidas, as colunas referentes a estes atributos, na Tabela 3, deveriam ser desconsideradas, porque não é possível calcular uma similaridade para atributos que ainda não foram definidos.

$$sim(c, a) = \sum_{i=1}^{n} s(c_i, a_i) * w(c_i) \tag{14}$$

No artigo Felfernig *et al.* (2013b), o peso de cada atributo do configurador de automóveis é definido em porcentagem (ver Tabela 4). Estes valores foram definidos a título de exemplo, com o intuito de gerar uma hierarquia de importância entre os atributos configuráveis dos automóveis.

Tabela 4 – Peso dos atributos do configurador de automóveis.

TYPE	FUEL	SKIBAG	4-WHEEL	PDC
50.0	5.0	10.0	30.0	5.0

Fonte – (FELFERNIG *et al.*, 2013b).

3.3.2.1.3 Tabela de similaridade

Ao calcular a similaridade entre as entradas da base de configurações completas (Tabela 3) e os requisitos inconsistentes definidos pela Equação (8) (C_{UR} = {c_6 : 4-*wheel* = *yes*, c_7 : *fuel* = 6*l*, c_8 : *type* = *city*, c_9 : *skibag* = *yes*, $c_1$0 : *pdc* = *yes*), considerando-se a métrica LIB para *fuel*, EIB para os demais atributos, e os pesos definidos pela Tabela 4, obtém-se a tabela de similaridade, retratada na Tabela 5.

Tabela 5 – Tabela de similaridade em nível de atributos para a configuração de automóveis.

measure	*EIB*	*LIB*	*EIB*	*EIB*	*EIB*	———
SESSION s_i	TYPE	FUEL	SKIBAG	4-WHEEL	PDC	*sim(c,a)*
s_1	1.0	1.0	0.0	0.0	1.0	0.6
s_2	1.0	1.0	0.0	0.0	0.0	0.55
s_3	0.0	0.0	1.0	1.0	1.0	0.18
s_4	0.0	0.67	0.0	0.0	1.0	0.08
s_5	0.0	0.67	0.0	0.0	0.0	0.03
s_6	0.0	0.0	0.0	1.0	1.0	0.35
s_7	0.0	0.67	1.0	0.0	0.0	0.13
s_8	0.0	0.67	1.0	0.0	0.0	0.13
$c = C_{uR}$	c_8 city	c_7 6l	c_9 yes	c_6 yes	c_{10} yes	———

Fonte – Adaptado de (FELFERNIG *et al.*, 2013b).

A Tabela 5 contém os valores de similaridade calculados em nível de atributo, sendo somados e ponderados para gerar a similaridade total (coluna *sim*(*c*, *a*)) de cada sessão de configuração prévia s_i. A similaridade total é o único valor a ser utilizado nas consultas. Substituindo os valores em nível de atributo pelos registros da base de configurações completas, obtém-se a tabela de similaridade retratada na Tabela 6. Essa versão é a efetivamente utilizada para guiar o cálculo de diagnósticos, como será exemplificado de forma prática, a seguir.

Tabela 6 – Tabela de similaridade para cenário da configuração de automóveis.

SESSION s_i	TYPE	FUEL	SKIBAG	4-WHEEL	PDC	*sim(c,a)*
s_1	city	4l	no	no	yes	0.6
s_2	city	4l	no	no	no	0.55
s_3	xdrive	10l	yes	yes	yes	0.18
s_4	limo	6l	no	no	yes	0.08
s_5	combi	6l	no	no	no	0.03
s_6	xdrive	10l	no	yes	yes	0.35
s_7	limo	6l	yes	no	no	0.13
s_8	combi	6l	yes	no	no	0.13
$c = C_{uR}$	c_8 city	c_7 6l	c_9 yes	c_6 yes	c_{10} yes	—

Fonte – Adaptado de (FELFERNIG *et al.*, 2013b).

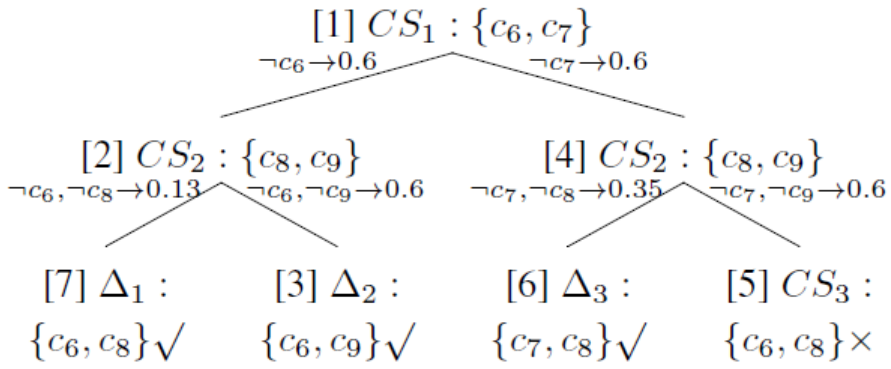
3.3.2.1.4 Aplicação do PDiag com o critério de similaridade

A tabela de similaridade retratada pela Tabela 6 é consultada pela função *sort*, do algoritmo *PDiag*, da seguinte maneira: os requisitos de usuário que definem os caminhos da árvore de busca, durante o cálculo de diagnósticos, são usados para filtrar a tabela, consultando-se o máximo valor de similaridade encontrado ao desconsiderar as entradas que contém os requisitos contidos no caminho. O máximo valor de similaridade retornado é então associado ao caminho. A regra é simplesmente expandir primeiro os caminhos com maior valor de similaridade. Desta forma, a exploração do grafo de busca não segue mais uma busca em largura (*breadth-first*), e sim uma busca com heurística, definida pelo critério de similaridade.

Para exemplificar este processo, aplica-se o algoritmo *PDiag* para calcular os diagnósticos para a situação de inconsistência definida pelo C_{UR} fornecido, usando o critério de similaridade, e obtém-se a árvore de busca mostrada na Figura 11. Os nodos são idênticos aos que foram obtidos aplicando o HSDAG convencional (sugere-se ao leitor comparar com a Figura 8), porém a ordem de exploração, denotada pelos números entre colchetes, foi alterada. Além disso, observa-se que o nome das arestas segue o padrão “atributos filtrados na tabela de similaridade” $\rightarrow$ “máximo valor de similaridade retornado”. O passo a passo da construção e exploração dos primeiros nodos da árvore será apresentado a seguir.

Do primeiro nodo, que consiste no primeiro conjunto de mínimo conflito retornado pelo *QuickXPlain*, originam-se duas arestas: uma referente à remoção de c_6 : 4-*wheel* = *yes* e a outra referente à remoção de c_7 : *fuel* = 6*l*. Para a primeira, uma consulta é feita na tabela de similaridade, considerando a negação de c_6, como

Figura 11 – Árvore de busca gerada pelo cálculo de diagnósticos usando a heurística de similaridade.



Fonte – Adaptado de (FELFERNIG *et al.*, 2013b).

mostra a Tabela 7. Todos os s_i que são inconsistentes com a negação de c_6, ou seja, que possuem 4-*wheel* = *yes*, são desconsiderados nesta consulta. Dos valores restantes, retorna-se o maior valor de similaridade encontrado, no caso, 0.6, circulado em verde.

Tabela 7 – Consulta na tabela de similaridade por $\neg c_6$.

SESSION s_i	TYPE	FUEL	SKIBAG	4-WHEEL	PDC	*sim(c,a)*
s_1	city	4l	no	no	yes	0.6
s_2	city	4l	no	no	no	0.55
~~s_3~~	~~xdrive~~	~~10l~~	~~yes~~	~~yes~~	~~yes~~	~~0.18~~
s_4	limo	6l	no	no	yes	0.08
s_5	combi	6l	no	no	no	0.03
~~s_6~~	~~xdrive~~	~~10l~~	~~no~~	~~yes~~	~~yes~~	~~0.35~~
s_7	limo	6l	yes	no	no	0.13
s_8	combi	6l	yes	no	no	0.13

Fonte – Adaptado de (FELFERNIG *et al.*, 2013b).

Para a segunda aresta, é realizada uma consulta onde se desconsideram as entradas consistentes com o atributo definido por c_7 : *fuel* = 6, e novamente o maior valor de similaridade encontrado é 0.6, como mostra a Tabela 8.

Como ambas arestas que partem do nodo raiz possuem a mesma atribuição de valor, convenciona-se expandir a da esquerda primeiro. Aplica-se o *QuickXPlain* considerando a remoção de c_6, e obtém-se $CS_2 = \{c_8, c_9\}$. A aresta da esquerda consulta a tabela de similaridade aplicando a negação de c_6 : 4-*wheel* = *yes* e c_8 : *type* = *city*.

Tabela 8 – Consulta na tabela de similaridade por $\neg c_7$.

SESSION s_i	TYPE	FUEL	SKIBAG	4-WHEEL	PDC	*sim(c,a)*
s_1	city	4l	no	no	yes	0.6
s_2	city	4l	no	no	no	0.55
s_3	xdrive	10l	yes	yes	yes	0.18
~~s_4~~	~~limo~~	~~6l~~	~~no~~	~~no~~	~~yes~~	~~0.08~~
~~s_5~~	~~combi~~	~~6l~~	~~no~~	~~no~~	~~no~~	~~0.03~~
s_6	xdrive	10l	no	yes	yes	0.35
~~s_7~~	~~limo~~	~~6l~~	~~yes~~	~~no~~	~~no~~	~~0.13~~
~~s_8~~	~~combi~~	~~6l~~	~~yes~~	~~no~~	~~no~~	~~0.13~~

Fonte – Adaptado de (FELFERNIG *et al.*, 2013b).

Essa consulta é ilustrada na Tabela 9. Desta vez, o maior valor de similaridade encontrado é 0.13, o qual é associado à aresta definida pelo caminho $(\neg c_6, \neg c_8)$.

Tabela 9 – Consulta na tabela de similaridade por $\neg c_6 \wedge \neg c_8$.

SESSION s_i	TYPE	FUEL	SKIBAG	4-WHEEL	PDC	*sim(c,a)*
~~s_1~~	~~city~~	~~4l~~	~~no~~	~~no~~	~~yes~~	~~0.6~~
~~s_2~~	~~city~~	~~4l~~	~~no~~	~~no~~	~~no~~	~~0.55~~
~~s_3~~	~~xdrive~~	~~10l~~	~~yes~~	~~yes~~	~~yes~~	~~0.18~~
s_4	limo	6l	no	no	yes	0.08
s_5	combi	6l	no	no	no	0.03
~~s_6~~	~~xdrive~~	~~10l~~	~~no~~	~~yes~~	~~yes~~	~~0.35~~
s_7	limo	6l	yes	no	no	0.13
s_8	combi	6l	yes	no	no	0.13

Fonte – Adaptado de (FELFERNIG *et al.*, 2013b).

Na aresta direita do nodo [2], a tabela de similaridade é consultada filtrando-se os registros em consistência com c_6 : 4-*wheel* = *yes* e c_9 : *skibag* = *yes*, como ilustrado na Tabela 10, e encontra-se novamente 0.6 como maior valor de similaridade.

Esse procedimento segue para a construção do restante do grafo, até que todos (ou *n*) os diagnósticos tenham sido determinados. Nota-se que a tabela de similaridade é construída só uma vez, antes de iniciar a construção do grafo, mas é consultada a cada geração de aresta na árvore de busca. Como pode ser observado pela ordem de exploração dos nodos, os diagnósticos foram obtidos na ordem $\Delta_2 \rightarrow \Delta_3 \rightarrow \Delta_1$, ao invés de $\Delta_1 \rightarrow \Delta_2 \rightarrow \Delta_3$, que havia sido obtido pela aplicação convencional do HSDAG, na Seção 3.2.2. Logo, supõe-se que, conforme a base de configurações

Tabela 10 – Consulta na tabela de similaridade por $\neg c_6 \wedge \neg c_9$.

SESSION s_i	TYPE	FUEL	SKIBAG	4-WHEEL	PDC	*sim(c,a)*
s_1	city	4l	no	no	yes	0.6
s_2	city	4l	no	no	no	0.55
~~s_3~~	~~xdrive~~	~~10l~~	~~yes~~	~~yes~~	~~yes~~	~~0.18~~
s_4	limo	6l	no	no	yes	0.08
s_5	combi	6l	no	no	no	0.03
~~s_6~~	~~xdrive~~	~~10l~~	~~no~~	~~yes~~	~~yes~~	~~0.35~~
~~s_7~~	~~limo~~	~~6l~~	~~yes~~	~~no~~	~~no~~	~~0.13~~
~~s_8~~	~~combi~~	~~6l~~	~~yes~~	~~no~~	~~no~~	~~0.13~~

Fonte – Adaptado de (FELFERNIG *et al.*, 2013b).

completas e as métricas de similaridade empregadas, o diagnóstico Δ_2 é o mais relevante a ser apresentado para o usuário, seguido pelo diagnóstico Δ_3, e por último Δ_1. Pode-se imaginar, num contexto com dezenas de diagnósticos, como a ordenação de diagnósticos poderia ser útil para auxiliar o usuário do configurador a escolher a alternativa mais recomendada.

3.3.2.2 Outras formas de personalização de diagnósticos

Além do diagnóstico baseado em similaridade, Felfernig *et al.* (2013b) também mostra como realizar diagnósticos com base na utilidade, probabilidade e em conjunto.

3.3.2.2.1 Diagnóstico baseado em utilidade

O diagnóstico em utilidade pressupõe a existência de uma base com preferências de usuários. A ideia é que o usuário do configurador, previamente ao processo de configuração, tenha definido valores de importância aos atributos do produto. Isto poderia ser feito indiretamente, com o sistema inferindo estes valores a partir de uma série de perguntas a serem respondidas, ou de forma direta, com o usuário definindo quais atributos considera mais importante, preenchendo um formulário semelhante à Tabela 4. Todavia, nota-se que a Tabela 4 é uma definição estática da importância dos atributos, que independente da sessão ou do usuário, enquanto na abordagem por utilidade, seria necessário armazenar estes valores para cada usuário do sistema. A ideia da técnica é, ao realizar o cálculo de diagnósticos, sugerir primeiramente a remoção de atributos definidos como de pouca importância pelo usuário corrente.

3.3.2.2.2 Diagnóstico baseado em probabilidade

O diagnóstico baseado em probabilidade necessita da implementação do que pode-se chamar de uma base de conflitos, a qual armazena configurações incompletas e inconsistentes de sessões prévias no sistema, com a decisão que foi tomada nestas ocasiões. Desta forma, poder-se-ia inferir, estatisticamente, qual seria a provável alternativa de reparo a ser escolhida pelo usuário, e basear a heurística de exploração de árvore do PDiag nestes valores de probabilidade.

Uma possível desvantagem é que a base de conflitos pode não ser útil para outros fins além de viabilizar a recomendação baseada em probabilidade. Já a base de configurações completas, a ser utilizada pela recomendação por similaridade, pode ser aproveitada do histórico de compras dos usuários, que provavelmente está presente em qualquer sistema de vendas *online*.

3.3.2.2.3 Diagnóstico baseado em conjunto

O diagnóstico baseado em conjunto consiste simplesmente na combinação das três abordagens anteriores. Aplica-se cada uma individualmente, e através de uma função que realiza a soma ponderada das recomendações, chamada no artigo de *majority voting*, obtém-se a uma ordenação final dos diagnósticos, como é retratado na Tabela 11. Naturalmente, esta é a abordagem mais custosa de ser implementada, do ponto de vista computacional, pois necessita da execução das outras três abordagens de recomendação.

Tabela 11 – *Majority voting* para diagnóstico baseado em conjunto.

METHOD / POSITION	1	2	3
utility-based	Δ_2	Δ_3	Δ_1
probability-based	Δ_3	Δ_1	Δ_2
similarity-based	Δ_2	Δ_3	Δ_1
ensemble-based	Δ_2	Δ_3	Δ_1

Fonte – (FELFERNIG *et al.*, 2013b).

Além das técnicas de recomendação, Felfernig *et al.* (2013b) também traz uma avaliação empírica das mesmas com base em dois conjuntos de dados, um de configuração de computadores e outro de serviços financeiros. Foi realizada uma comparação da qualidade de predição e desempenho entre as diferentes abordagens de recomendação, obtendo resultados muito semelhantes, mas todas se saíram melhores do que o cálculo de diagnósticos convencional (sem personalização).

Neste capítulo, foi apresentada a fundamentação teórica obtida para a realização deste trabalho, iniciando com a apresentação dos CSPs, diminuiu-se gradualmente o escopo, passando pela configuração de produtos e sistemas de recomendação, até as técnicas de recomendação de diagnósticos, onde as diferentes abordagens estudadas foram apresentadas, com foco especial na recomendação baseada em similaridade. Essa última foi selecionada para ser implementada na aplicação demonstrativa desenvolvida neste trabalho, cujo planejamento é apresentado no próximo capítulo.

4 PROJETO DA POC

Neste capítulo, apresentam-se as principais ferramentas utilizadas no desenvolvimento da aplicação, bem como a arquitetura proposta, atores, casos de uso, especificação de requisitos, atividades estipuladas e a metodologia de desenvolvimento empregada.

O planejamento inicial era desenvolver a funcionalidade de recomendação de diagnósticos diretamente como um módulo no *framework* de configuração por restrições atualmente em desenvolvimento na empresa (referido aqui simplesmente por *framework*). Todavia, optou-se pelo desenvolvimento de uma aplicação externa, inicialmente desvinculada do *framework*, para servir como uma PoC para a recomendação de diagnósticos. A razão disso é que o *framework* possui muitas camadas adicionais sobre a biblioteca *Choco-Solver*, o que dificultaria a implementação e verificação dos algoritmos a serem desenvolvidos. Dessa forma, definiu-se que a atividade de integração da aplicação ao *framework* seria realizada logo após a conclusão deste trabalho.

4.1 FERRAMENTAS

Este projeto foi desenvolvido com base nas tecnologias que já eram utilizadas no *framework* de configuração principal. O desenvolvimento do *back-end* da aplicação foi feito com a linguagem de programação orientada a objetos Java. O ambiente de desenvolvimento integrado (IDE) utilizado foi o IntelliJ IDEA, desenvolvido pela JetBrains, que atualmente consiste numa das escolhas mais populares para desenvolvimento em Java.

A aplicação foi estruturada usando o *framework* Spring, que facilita usar os padrões de injeção de dependências (DI) e inversão de controle (IoC), permitindo a construção de sistemas desacoplados (BAELDUNG, 2021). Além disso, o Spring também provê diversas funcionalidades *out-of-the-box*, como um servidor Apache Tomcat pré-configurado, e todo um conjunto de classes e anotações para o rápido desenvolvimento dos controladores e serviços necessários para rodar aplicações *web*.

A biblioteca de programação por restrições *Choco-Solver*, também desenvolvida em Java, foi utilizada para implementar o *core* do configurador de produtos. Essa biblioteca foca na resolução em geral de CSPs, podendo ser utilizada para o desenvolvimento de configuradores de produtos (Seção 5.1).

Para versionamento de código, foi usado o *GitHub*, gerenciador de repositório de *software* baseado em *git* (CHACON; STRAUB, 2014). O uso de versionamento é considerado uma *best practice* para equipes de desenvolvimento, pois permite a manutenção de um histórico dos arquivos, automação de processos de integração e testes, desenvolvimento simultâneo (através de *branching* e *merging*), dentre outras funcionalidades úteis ao desenvolvimento de *software*.

Para efetuar o gerenciamento do projeto, foi utilizado o JIRA, desenvolvido pela *Atlassian*, que é uma plataforma com ferramentas de planejamento e roteiro para equipes de desenvolvimento. O JIRA suporta o emprego de metodologias ágeis, especificamente *Scrum* e *Kanban*, sendo esta última escolhida como metodologia de desenvolvimento a ser utilizada neste projeto (Seção 4.6).

Para a criação dos protótipos de telas da interface do configurador, foi usado o *software Balsamiq Mockups*, que é uma ferramenta para construção de *wireframes* para páginas *web*. Já para o desenvolvimento das telas, a *engine* de renderização de *templates Thymeleaf* foi utilizada, a qual possibilitou a sincronização do que era exibido no *front-end* ao estado de configuração, armazenado no servidor. Ao contrário de sua conhecida alternativa JSP, o *Thymeleaf* permite a inserção de *tags* não invasivas no HTML, melhorando a legibilidade do código e sendo, discutivelmente, mais fácil de aprender.

Como banco de dados da aplicação, foi utilizado o banco de dados em memória H2, que é bastante leve e ideal para cenários de testes, tal como uma aplicação PoC. Para interação com o mesmo, o módulo *Spring Data* foi utilizado, o qual inclui o *Hibernate*, solução de *Object-Relational Mapping* (ORM) que implementa a especificação JPA e efetua diretamente a comunicação com o H2.

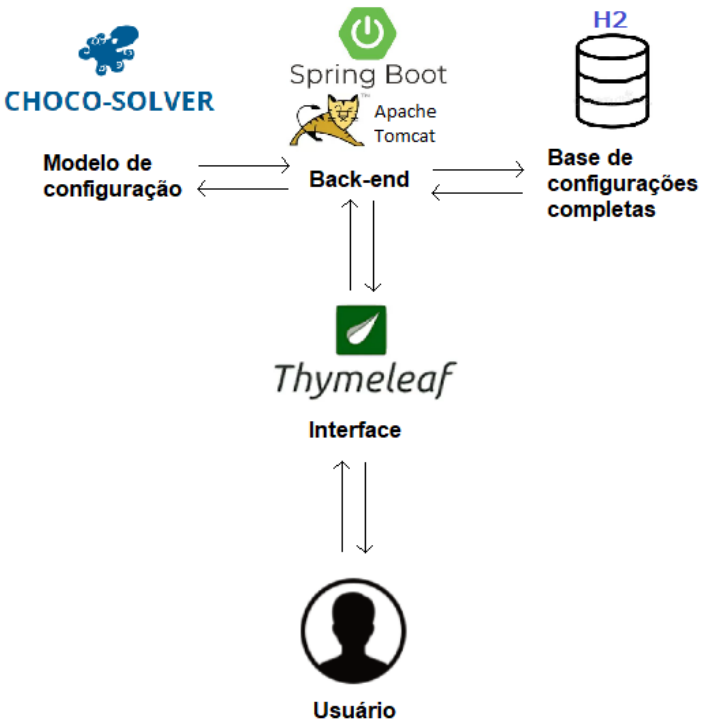
4.2 ARQUITETURA DO SISTEMA

O sistema a ser desenvolvido, a partir das ferramentas listadas na seção anterior, consiste numa aplicação *web* a ser hospedada em nuvem através do serviço *Heroku*. O intuito de uma aplicação para prova de conceito é ser o mais simples o possível, e servir para demonstrar, neste caso, a integração de tecnologias de recomendação em configuradores de produtos baseados em restrições, especificamente através da implementação da técnica de recomendação de diagnósticos baseada em similaridade (apresentada na Seção 3.3.2.1).

A Figura 12 traz uma visão geral do sistema proposto. O *back-end* consiste no componente central da arquitetura, ligando os demais. O banco de dados H2 armazena as características de produtos obtidos em sessões de configuração finalizadas. As instanciações de modelos de produto durante processos de configuração, de acordo com suas características e restrições, são realizadas pela API da biblioteca *Choco-Solver*, e o *Thymeleaf* efetua a ligação entre a interface com o usuário e o *back-end* da aplicação.

Complementando a visão geral do sistema, o diagrama da Figura 13 retrata em *Unified Modeling Language* (UML) a arquitetura proposta para o sistema. A aplicação deverá seguir o padrão *Model-View-Controller* (MVC), separando-se em três camadas conectadas: a camada de visão (*view*), onde ocorre a apresentação dos dados e interação com o usuário através de uma interface, a camada de modelo (*model*), que

Figura 12 – Visão geral do sistema.



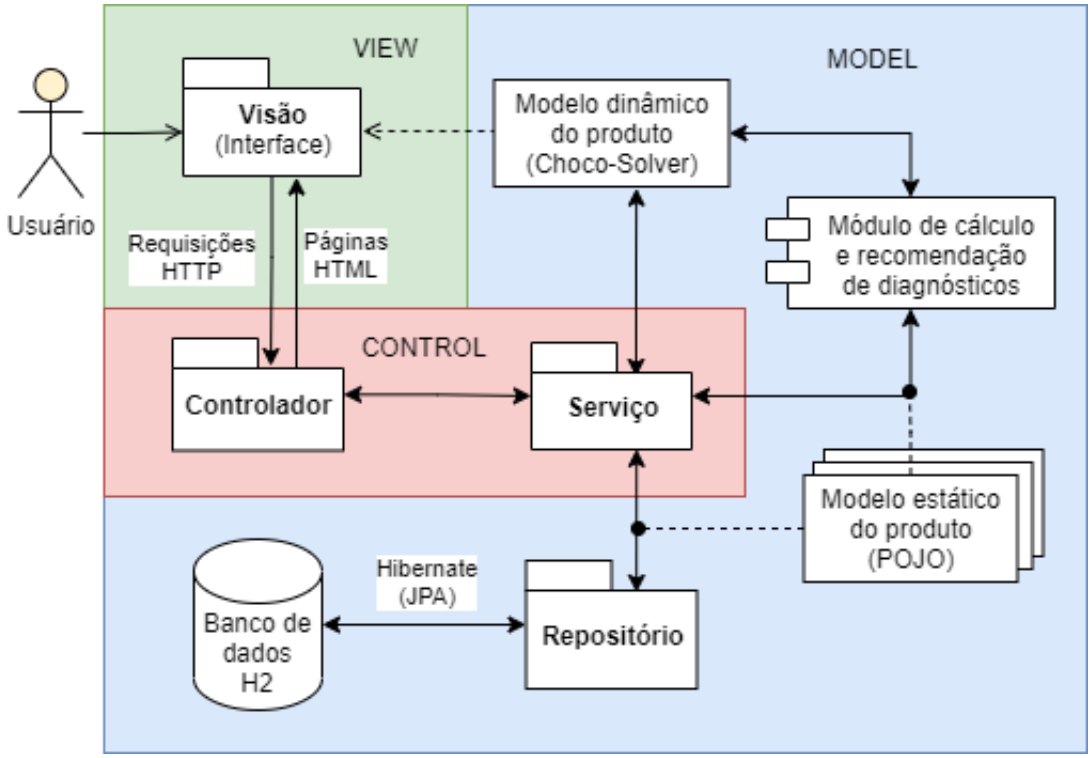
Fonte – Elaborado pelo autor.

consiste na parte lógica da aplicação, armazenando o estado de configuração corrente, e a camada de controle (*controller*), que faz a mediação entre as outras duas.

Existem dois modelos que são usados para representar as configurações, um estático e outro dinâmico. O modelo estático é usado para definir instâncias de configurações completas, representando o produto como um objeto Java convencional (*Plain Old Java Object* (POJO)). Essas configurações são salvas no banco de dados ou resgatadas para auxiliar no processo de recomendação de diagnósticos com base em similaridade. Já o modelo dinâmico tem seus atributos definidos como tipos específicos da biblioteca *Choco-Solver*, sendo utilizado para representar o estado de configuração corrente, sendo este o modelo que é reproduzido “em tempo real” na interface e manipulado pelo usuário.

Outro diagrama importante é o diagrama de *deployment* (implantação), retratado na Figura 14, que mostra como a arquitetura proposta será efetivamente executada, e possibilita a visualização de como os componentes de *hardware* e *software* do sistema estarão interligados.

Figura 13 – Diagrama de arquitetura do sistema.



Fonte – Elaborado pelo autor.

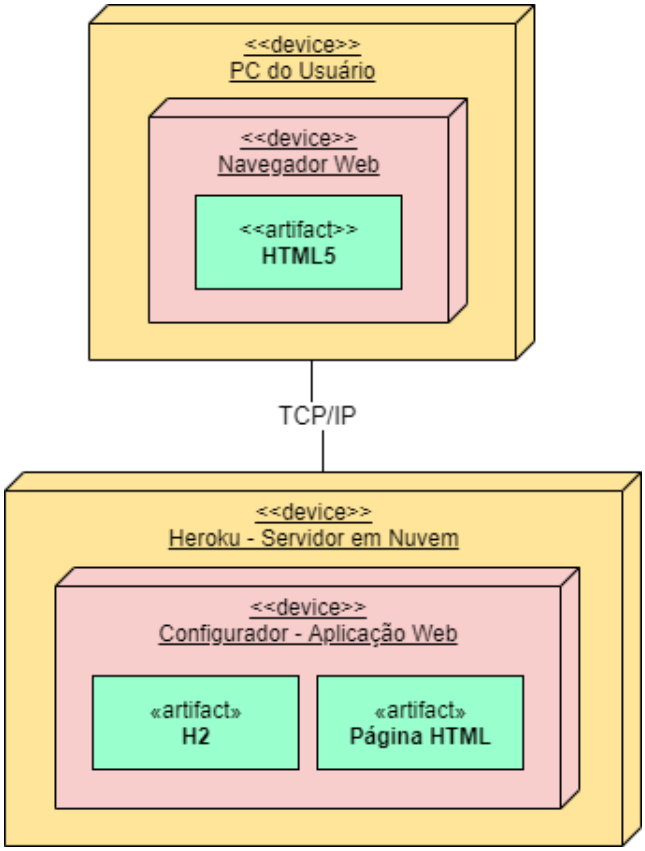
4.3 ATORES E CASOS DE USO

De acordo com Hotz *et al.* (2014), existem pelo menos três perfis bem definidos de usuários em um sistema de configuração de produtos:

- Especialista do domínio (*domain-specialist*): engenheiro com conhecimento técnico especializado no produto, incluindo suas possíveis características e peculiaridades. Conhece implicitamente as restrições do produto;
- Engenheiro do conhecimento (*knowledge-engineer*): também conhecido como administrador de regras, é o engenheiro com experiência em programação responsável por extrair os conhecimentos do especialista e traduzí-los em regras a serem inseridas no sistema de configuração. Também é responsável por dar manutenção ao sistema.
- Usuário final: utilizador do sistema de configuração. Pode ser um cliente interno ou externo da empresa, podendo ter ou não conhecimento técnico sobre o produto a ser configurado.

O fluxo de informação entre o especialista do domínio e o engenheiro do conhecimento, se não for bem-planejado, pode gerar um gargalo conhecido como *knowledge-*

Figura 14 – Diagrama de implantação do sistema.



Fonte – Elaborado pelo autor.

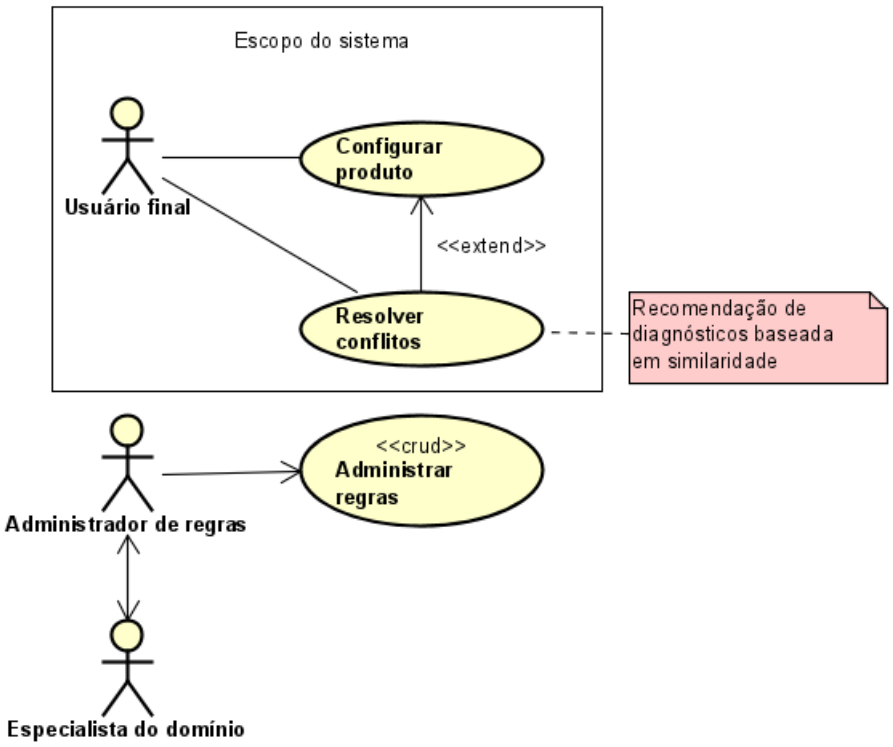
acquisition bottleneck (HOTZ *et al.*, 2014). Este problema pode estar presente tanto na fase de concepção do configurador quanto em fases de manutenção, sendo mitigado quando o engenheiro do conhecimento busca adquirir também conhecimento técnico sobre o domínio. O usuário final também pode ajudar na concepção e manutenção do sistema, ao providenciar *feedback* da sua experiência de utilização do sistema.

Como a PoC busca demonstrar uma técnica de recomendação de diagnósticos, que serve especificamente para a resolução de conflitos passíveis de ocorrerem durante sessões de configuração, o único ator considerado no sistema proposto é o usuário final, que desempenha o caso de uso “Configurar produto”, como mostra a Figura 15. Nota-se que o fragmento “Resolver conflitos” foi destacado, porque consiste numa etapa crítica do processo de configuração de um produto, em que pretende-se demonstrar a técnica da recomendação de diagnósticos.

A interação do usuário com a aplicação, com o intuito de configurar um produto, é descrita pelos seguintes passos:

1. O usuário acessa a página do configurador, que deve representar o estado inicial de configuração (nenhum componente selecionado);

Figura 15 – Atores e casos de uso do sistema.



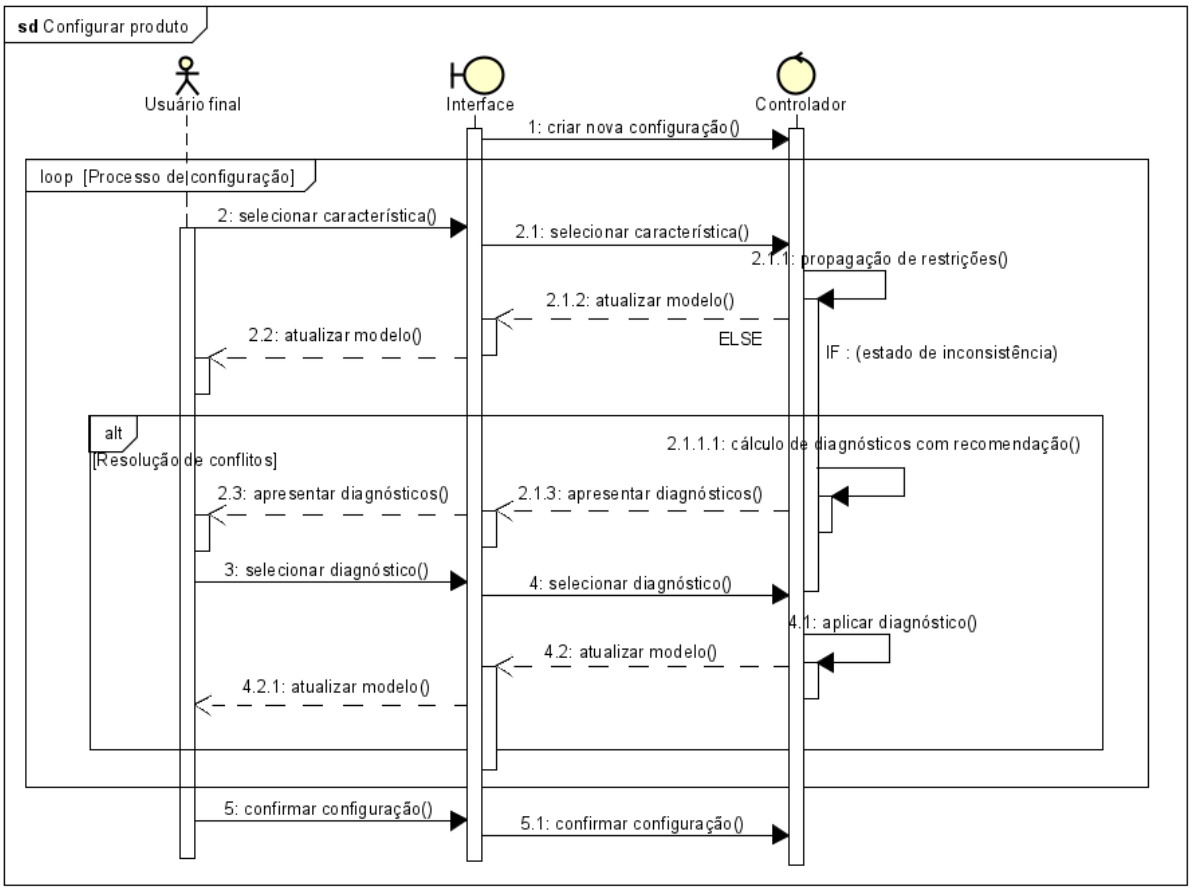
Fonte – Elaborado pelo autor.

2. o usuário realiza o processo de configuração, selecionando os requisitos que deseja para cada componente do computador;
3. *se desejar*, o usuário pode reiniciar o processo de configuração a qualquer momento, retornando para o estado inicial;
4. *se* um estado de inconsistência for atingido, o processo de cálculo de diagnósticos é efetuado, e as alternativas de reparo são apresentadas ao usuário na ordem recomendada. Este, no que lhe concerne, escolhe uma das alternativas ou simplesmente opta por desfazer a última seleção efetuada;
5. o usuário, após obter uma configuração completa e válida, finaliza o processo de configuração;
6. o configurador exibe a tela de resultados e apresenta a BOM da configuração finalizada pelo usuário, oferecendo também a opção de iniciar uma nova sessão.

A partir dos passos enumerados, constrói-se um diagrama de sequência de sistema, retratado na Figura 16, o qual permite visualizar a sequência de eventos e

respostas entre os atores e outros componentes do sistema (WAZLAWICK, 2015).

Figura 16 – Diagrama de sequência do caso de uso Configurar produto.



Fonte – Elaborado pelo autor.

4.4 ESPECIFICAÇÃO DE REQUISITOS

A partir dos casos de uso e do diagrama de sequência, podem-se levantar os requisitos do sistema. Os requisitos são as propriedades que este deve ter para atender às especificações solicitadas, sendo divididos em requisitos funcionais e requisitos não-funcionais. Os requisitos funcionais definem o que o sistema deve efetivamente fazer, enquanto os requisitos não-funcionais correspondem a restrições sobre como tais funcionalidades devem ser executadas (WAZLAWICK, 2015).

Nesse trabalho, logo após a definição da técnica que seria utilizada para efetuar as recomendações de diagnósticos, foi discutido quais seriam as funcionalidades relevantes a serem implementadas na aplicação demonstrativa, e os seguintes requisitos foram levantados:

1. possibilitar que o usuário selecione incrementalmente as características do produto a ser configurado;
2. providenciar *feedback* para a interação do usuário, indicando: seleções do usuário, seleções efetuadas pelo configurador e alternativas que levam para estados de inconsistência.
3. quando alcançado um estado de inconsistência, guiar o usuário para fora do mesmo, efetuando o cálculo e apresentação de diagnósticos;
4. gerar uma BOM ao final de um processo de configuração.
5. (não funcional) a apresentação de diagnósticos deve seguir a técnica de recomendação por similaridade, considerando a configuração da sessão atual e uma base de configurações completas;
6. (não funcional) a interface entre usuário e configurador deve se dar através de uma página *web* amigável, com imagens e botões;

Do ponto de vista técnico, o requisito 3 é o mais desafiador, pois requer a implementação das tecnologias de recomendação no processo de configuração. Também é o requisito mais importante, pois é o foco da demonstração do sistema. Já o requisito 2 consiste no maior desafio da implementação do *front-end*, pois requer a integração entre o modelo de configuração que roda no servidor e as opções a serem exibidas ao usuário.

Na próxima seção, são apresentadas as atividades que foram definidas para efetuar o desenvolvimento da aplicação.

4.5 ATIVIDADES PLANEJADAS

A primeira atividade executada foi a realização de um estudo sobre a integração de tecnologias de recomendação em configuradores de produtos, através da leitura de livros e artigos relacionados ao tema. A partir dos conhecimentos adquiridos, tinha-se como objetivo propor uma estratégia para a recomendação de alternativas de reparo durante processos de configuração, e como resultado foi definida a implementação da técnica de recomendação de diagnósticos por similaridade (Seção 3.3.2).

Logo após o anteprojeto, diversas atividades de desenvolvimento foram elencadas como sub-tarefas do projeto na plataforma JIRA. No desenrolar do mesmo, algumas atividades foram adicionadas, alteradas ou removidas. A última versão do *Backlog*, lugar onde se armazenam as atividades no JIRA, compunha-se pelas seguintes atividades:

- estudar sobre configuradores de produtos;

- estudar sobre a integração de sistemas de recomendação em configuradores de produtos;
- definir uma estratégia para recomendação de alternativas de reparo;
- elaborar modelo de testes para os algoritmos;
- corrigir *bugs* no projeto base;
- elaborar base de configurações completas;
- implementar recomendação de diagnósticos por similaridade;
- desenvolver interface do configurador;
- hospedar a PoC numa plataforma em nuvem;
- integrar a PoC no *framework*;
- desenvolver tela de resolução de conflitos para o configurador principal;
- evoluir o modelo para um produto oferecido pela WEG;

Nota-se que a atividade “implementar recomendação de diagnósticos por similaridade” e “desenvolver interface do configurador” se desdobram nas sub-atividades desenvolvidas nas seções 5.2 e 5.3, respectivamente. Além disso, as atividades posteriores a “hospedar a PoC numa plataforma em nuvem” ficaram de fora do escopo deste trabalho.

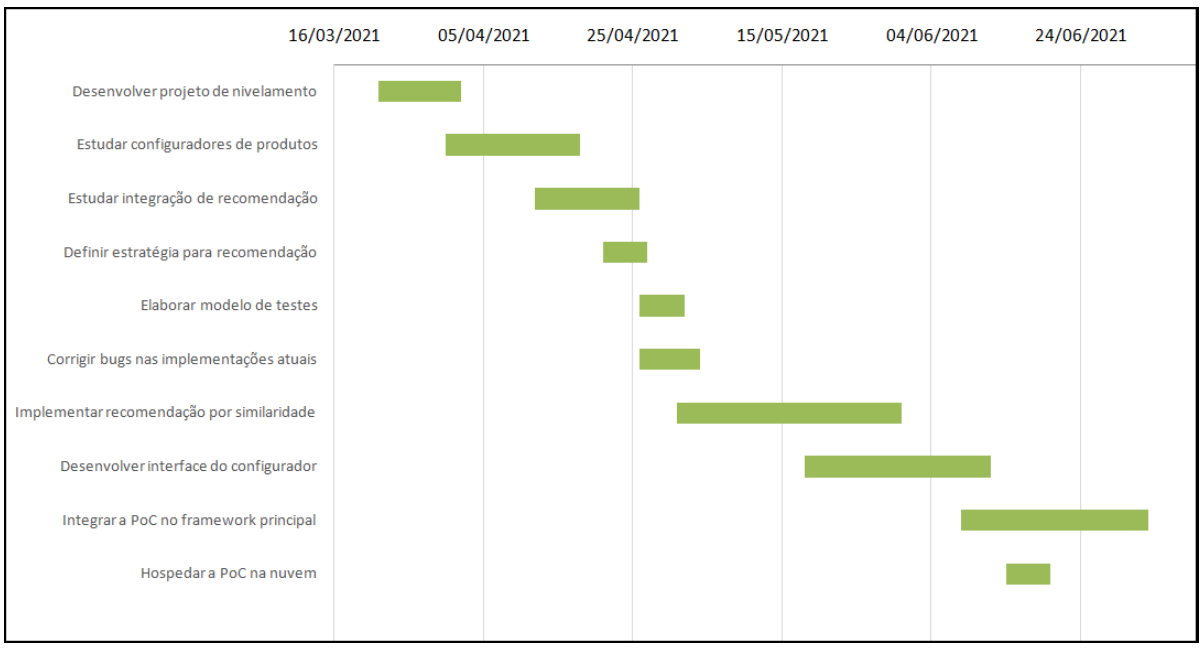
Motivado pela necessidade de elaboração de relatórios executivos, foi também construído um diagrama de Gantt (Figura 17), contendo as atividades especificamente relacionadas ao desenvolvimento da PoC, dispondo-as temporalmente conforme as respectivas estimativas de esforço e ordem de execução.

4.6 METODOLOGIA DE DESENVOLVIMENTO

O projeto foi desenvolvido utilizando a metodologia de desenvolvimento ágil *Kanban*, cuja principal característica é o quadro *Kanban*, que contém cartões que representam os itens de trabalho. Esses cartões são organizados no quadro e fluem através dos estágios do fluxo de trabalho, representados por colunas. Na versão mais básica, o quadro possui três colunas: *To Do* (a fazer), *In Progress* (em andamento) e *Done* (*concluído*).

A Figura 18 mostra o quadro *Kanban* utilizado no projeto, em uma determinada etapa do desenvolvimento. Nota-se que as colunas *To Do* e *Done* foram renomeadas para *Open* e *Closed*, respectivamente.

Figura 17 – Diagrama de Gantt.



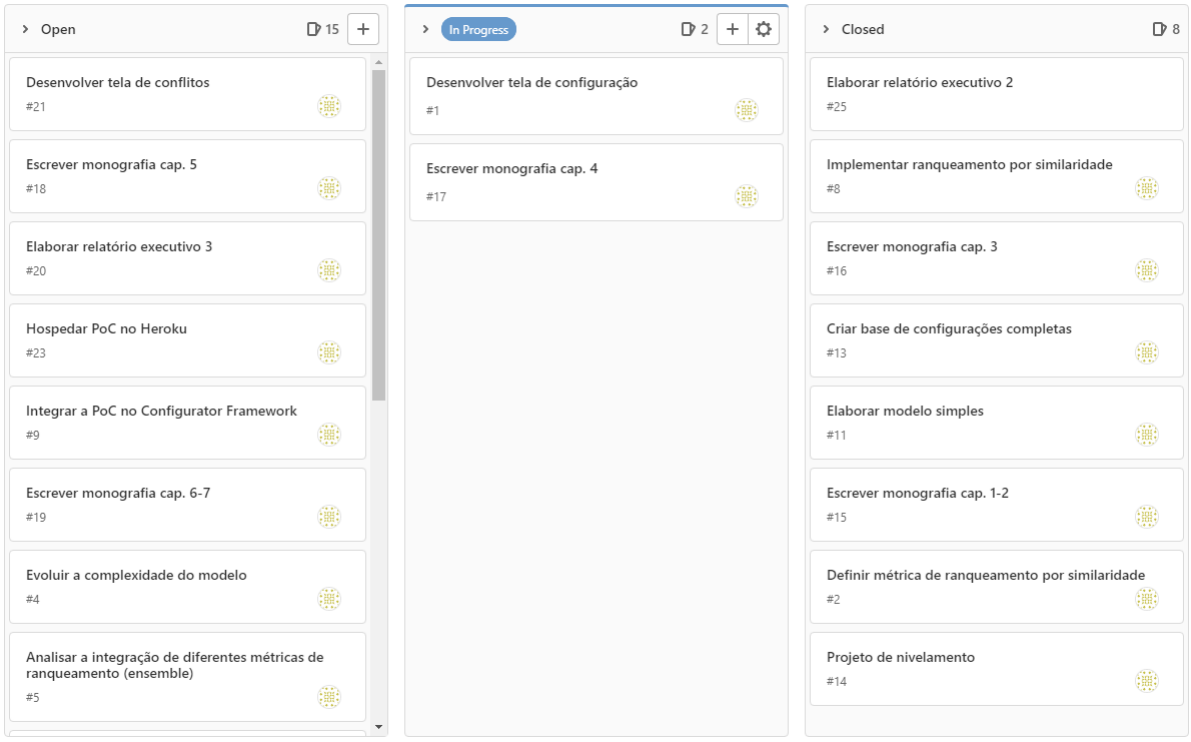
Fonte – Elaborado pelo autor.

Na metodologia Kanban, são empregados diversos conceitos, como a não diferenciação das funções entre os membros da equipe, fluxo contínuo e entrega contínua. As principais métricas são o tempo de espera, tempo de ciclo e *Work In Progress* (WIP), que é o limite de cartões que podem estar numa determinada coluna em simultâneo. Mais detalhes e especializações da metodologia podem ser conferidos em Brechner (2015).

Apesar do desenvolvimento do projeto se dar de forma individual, decidiu-se que o *Kanban* seria adequado à proposta, visto que um dos princípios da metologia é focar na visualização do trabalho a ser efetuado (*visualize your work*). Esse princípio, além de fomentar a organização pessoal, facilitou a comunicação com os supervisores, a qual era efetuada através de reuniões semanais, onde se discutiam os resultados obtidos e o planejamento de atividades a serem desenvolvidas.

Neste capítulo, foi apresentado todo o projeto para o desenvolvimento da aplicação PoC, cuja execução é relatada no próximo capítulo.

Figura 18 – Quadro *Kanban*.



Fonte – Elaborado pelo autor.

5 DESENVOLVIMENTO

Neste capítulo, é apresentado o desenvolvimento efetivo da aplicação de PoC para a recomendação de diagnósticos, que pode ser dividido em três partes: a primeira (Seção 5.1) consiste na elaboração do modelo de testes e na implementação dos algoritmos que efetuam o cálculo de diagnósticos da maneira convencional. A segunda (Seção 5.2) consiste na implementação das funcionalidades necessárias para realizar o cálculo de diagnósticos com recomendação por similaridade, e a terceira (Seção 5.3) consiste na elaboração das telas da interface de usuário e na integração das mesmas com o *back-end* da aplicação.

5.1 IMPLEMENTAÇÃO INICIAL

O desenvolvimento do código iniciou tomando como ponto de partida um repositório de testes fornecido por meu co-supervisor na empresa, buscando aproveitar as implementações dos algoritmos *QuickXPlain* e HSDAG previamente realizadas, porém haviam alguns *bugs* a serem corrigidos. Esse repositório fora usado antes mesmo do início do desenvolvimento do *framework* de configuração principal, e suas classes interagiam diretamente com a biblioteca *Choco-Solver*.

5.1.1 Algoritmos para o cálculo de diagnósticos convencional

A primeira atividade de desenvolvimento foi corrigir alguns *bugs* da implementação do *QuickXPlain* que havia no repositório, pois alguns testes foram realizados e os diagnósticos esperados não eram obtidos. A razão identificada foi que, a partir das restrições inicialmente aplicadas ao CSP, C_{UR} e C_{KB}, o *solver* deriva restrições implícitas, que auxiliam o processo de resolução. Porém, ao aplicar o *QuickXPlain*, é necessário efetuar sistematicamente o desligamento e religamento das restrições, e não estava sendo possível reativar as restrições que eram geradas implicitamente. A solução obtida foi gravar o estado do modelo antes de efetuar a postagem das restrições, desta forma podendo inferir quais restrições foram geradas implicitamente e evitar o desligamento das mesmas.

Todas as dificuldades da implementação, no âmbito da resolução de CSPs, se devem ao fato de que a biblioteca *Choco-Solver* não é voltada especificamente à configuração de produtos, e sim à resolução em geral de CSPs, onde o foco está em achar uma solução para os problemas. O processo de identificar restrições inconsistentes, o qual é realizado através da elaboração de diagnósticos, requer uma implementação de algoritmos específicos a este propósito, a qual foi realizada com base nos pseudo-códigos encontrados na literatura (Seção 3.2).

A Figura 19 traz a implementação já corrigida do *QuickXPlain*. A classe na

qual este algoritmo está contido também possui algumas funções auxiliares que não aparecem na figura, como o `refresh()`, que prepara o modelo para uma nova aplicação do *QuickXPlain*, e o construtor da classe, que guarda as restrições implicitamente geradas para contornar o problema mencionado anteriormente.

Figura 19 – Implementação do algoritmo *QuickXPlain*.

```
public List<Constraint> qx(List<Constraint> b, List<Constraint> c) {
    refresh();
    b.forEach(cstr -> cstr.setEnabled(true));
    c.forEach(cstr -> cstr.setEnabled(true));

    return c.isEmpty() || model.getSolver().solve() ? new ArrayList<>() : qx(new ArrayList<>(), c, b);
}

public List<Constraint> qx(List<Constraint> d, List<Constraint> c, List<Constraint> b) {

    if (!d.isEmpty()) {
        refresh();
        b.forEach(cstr -> cstr.setEnabled(true));
        if (!model.getSolver().solve()) {
            return new ArrayList<>();
        }
    }
    if (c.size() == 1) {
        return c;
    }
    List<Constraint> c1 = c.subList(0, (c.size() + 1) / 2);
    List<Constraint> c2 = c.subList((c.size() + 1) / 2, c.size());
    List<Constraint> cs1 = qx(c2, c1, ListUtils.union(b, c2));
    List<Constraint> cs2 = qx(cs1, c2, ListUtils.union(b, cs1));
    return ListUtils.union(cs1, cs2);
}
```

Fonte – Elaborado pelo autor.

O *QuickXPlain* é aliado ao HSDAG para realizar o cálculo de diagnósticos da maneira convencional, sendo equivalente à utilização do *PDiag* com a recomendação desativada, cuja implementação será mostrada na Seção 5.2. A próxima atividade desenvolvida foi a elaboração do modelo de testes, que corresponde ao domínio de configuração a ser utilizado como exemplo na PoC.

5.1.2 Elaboração do modelo de testes

Como visto na Seção 2.3.2, uma das vantagens da utilização de um configurador baseado em restrições é a separação entre a declaração do modelo de domínio e o processo de resolução do problema de configuração. Por esta razão, não foi preciso usar como modelo de testes um produto complexo da empresa, o que demandaria muito tempo de estudo e poderia dificultar o objetivo demonstrativo da aplicação. O

modelo mais simples de configuração real que se cogitou utilizar, por exemplo, tem em torno de 400 características, gerando cerca de 400 mil combinações tabulares, o que seria intratável com o tempo de estágio disponível. Logo, optou-se por utilizar um modelo relativamente simples, que pudesse servir para o propósito de apresentar as tecnologias de recomendação.

Em alguns artigos relacionados ao tema de cálculo de diagnósticos, foi observada, por exemplo, a utilização de domínios como configuradores de computadores (ATAS *et al.*, 2019), serviços financeiros (FELFERNIG *et al.*, 2009), e automóveis (FELFERNIG *et al.*, 2013b).

O modelo elaborado consiste num cenário de configuração de computadores. A definição das variáveis e seus respectivos domínios foram inspirada nos modelos-exemplo utilizados em Wotawa *et al.* (2015) e Jannach *et al.* (2001). Seguindo a notação introduzida na Seção 3.1.1, define-se o modelo como:

$$V = \{uso, gabinete, processador, drive\text{-}optico, placa\text{-}video, memoria, hdd, ssd\}, \quad (15)$$

$$\begin{aligned} D = \{&dom(uso) = \{basico, desenvolvimento, games\}, \\ &dom(gabinete) = \{mini, desktop, gamer\} \\ &dom(processador) = \{dual\text{-}core, quad\text{-}core\} \\ &dom(drive-optico) = \{sim, nao\} \\ &dom(placa-video) = \{sim, nao\} \\ &dom(memoria) = \{4GB, 8GB, 16GB\} \\ &dom(hdd) = \{sim, nao\} \\ &dom(ssd) = \{sim, nao\}\}. \end{aligned} \quad (16)$$

O modelo possui oito variáveis, as quais representam oito componentes relevantes para a montagem de um computador. Outros componentes como placa-mãe e fonte foram desconsiderados da modelagem, assumindo-se, para fins de simplificação, que são de tipos universais. Pela definição do domínio das variáveis, na equação 16, percebe-se que a maioria pode ser representada por valores *booleanos*, enquanto o restante deve receber uma atribuição dentre uma enumeração de três opções.

Na Figura 20, mostra-se a implementação das variáveis e seus respectivos domínios utilizando a biblioteca *Choco-Solver*. Como o *solver* abstrai todos os valores ao tipo *Integer* (inteiros), foi necessário representar os nomes definidos na Equação (16) através de enumerações, as quais seguem a mesma ordenação de definição dos domínios, por exemplo, o domínio da variável *uso*, {*basico*, *desenvolvimento*, *games*}, equivale à enumeração $\{1, 2, 3\}$.

Figura 20 – Implementação das variáveis e domínios do modelo.

```
public Model model = new Model( name: "Computer");
public IntVar uso = model.intVar( name: "uso",  lb: 1, Uso.values().length);
public IntVar gabinete = model.intVar( name: "gabinete",  lb: 1, Gabinete.values().length);
public IntVar processador = model.intVar( name: "processador",  lb: 1, Processador.values().length);
public BoolVar driveOptico = model.boolVar( name: "driveOptico");
public BoolVar placaVideo = model.boolVar( name: "placaVideo");
public IntVar memoria = model.intVar( name: "memoria",  lb: 1, Memoria.values().length);
public BoolVar hdd = model.boolVar( name: "hdd");
public BoolVar ssd = model.boolVar( name: "ssd");
```

Fonte – Elaborado pelo autor.

Se nenhuma restrição é considerada, são possíveis 864 configurações completas. Já ao aplicar as restrições da base de conhecimento, definidas por:

$$
\begin{aligned}
C_{KB} = \{ & c_1 : (hdd = sim) \vee (ssd = sim), \\
& c_2 : uso = basico \Rightarrow (gabinete \neq gamer) \wedge \\
& (processador = dual\text{-}core) \wedge (memoria = 4GB), \\
& c_3 : uso = desenvolvimento \Rightarrow (memoria = 16GB) \wedge (ssd = sim), \\
& c_4 : uso = games \Rightarrow (placa\text{-}video = sim) \wedge \\
& (processador = quad\text{-}core) \wedge (memoria \neq 4GB), \\
& c_5 : gabinete = mini \Rightarrow (placa\text{-}video = nao) \wedge \\
& ((hdd = sim) \oplus (ssd = sim)), \\
& c_6 : gabinete = gamer \Rightarrow (drive\text{-}optico = nao) \} ,
\end{aligned}
\tag{17}
$$

o número de configurações completas e válidas é de 62. Analisando as restrições definidas em (17), c_1 força a escolha de pelo menos uma opção de armazenamento, enquanto c_2, c_3 e c_4 definem algumas seleções obrigatórias conforme o uso do computador. Já c_5 define que o gabinete do tipo *mini* não tem espaço para placa de vídeo e pode receber apenas um tipo de armazenamento ($\oplus$ é o símbolo para a operação lógica XOR), e por fim c_6 define que o gabinete do tipo *gamer* não comporta um drive óptico. A Figura 21 mostra a implementação dessas restrições através da API da biblioteca *Choco-Solver*.

Observa-se que todas as restrições de C_{KB} são consistentes entre si, isto é, não entram em contradição. Além disso, no início da configuração, as restrições não forçam uma escolha *a priori* de nenhum componente, isto é, ao aplicar C_{KB} sobre V e D e efetuar uma propagação, o domínio das variáveis permanece o mesmo. Este fator permite com que toda possível atribuição de valor a uma variável, considerando

Figura 21 – Implementação das restrições da base de conhecimento.

```
private final Constraint c1 = hdd.or(ssd).decompose();
private final Constraint c2 = uso.eq( y: 1).imp(gabinete.ne( y: 3).and(processador.eq( y: 1))
        .and(memoria.eq( y: 1))).decompose();
private final Constraint c3 = uso.eq( y: 2).imp(memoria.eq( y: 3).and(ssd)).decompose();
private final Constraint c4 = uso.eq( y: 3).imp(placaVideo.and(processador.eq( y: 2))
        .and(memoria.ne( y: 1))).decompose();
private final Constraint c5 = gabinete.eq( y: 1).imp(placaVideo.not().and(hdd.xor(ssd))).decompose();
private final Constraint c6 = gabinete.eq( y: 3).imp(driveOptico.not()).decompose();
```

Fonte – Elaborado pelo autor.

os domínios iniciais definidos na Equação (16), apareça em pelo menos uma das soluções possíveis.

5.1.3 Exemplo de cálculo de diagnósticos sem recomendação

Para testar os algoritmos implementados para o cálculo de diagnósticos, considerando o modelo da configuração de computadores introduzido na seção anterior, foram desenvolvidas algumas classes de testes, as quais podem ser conferidas no repositório do projeto (DE CARVALHO, 2021b). Com o intuito de ilustrar como o cálculo de diagnósticos está sendo efetuado de forma a seguir as técnicas apresentadas no Capítulo 3, apresenta-se nesta seção um dos testes efetuados.

Suponha que, em uma data sessão de configuração, o conjunto de requisitos do usuário seja dado por

$$\begin{aligned} C_{UR} = \{ & r_1 : \mathit{hdd} = \mathit{sim}, \\ & r_2 : \mathit{drive\text{-}optico} = \mathit{sim}, \\ & r_3 : \mathit{memoria} = 4\mathit{GB}, \\ & r_4 : \mathit{uso} = \mathit{games}, \\ & r_5 : \mathit{gabinete} = \mathit{mini}, \\ & r_6 : \mathit{ssd} = \mathit{sim} \}. \end{aligned} \tag{18}$$

Nota-se que apenas seis das oito variáveis do computador foram instanciadas, portanto a configuração é incompleta. Também é inválida, pois alguns requisitos do usuário são inconsistentes com a base de conhecimento C_{KB} (Equação (17)), por exemplo, r_4 está em conflito com r_3, porque uma das implicações da restrição c_4 é de que o uso para *games* necessita de pelo menos $8GB$ de memória. A Figura 22 mostra a definição desse conjunto de requisitos com a biblioteca *Choco-Solver*.

Com a aplicação dos requisitos dados pela Equação 18, a partir de uma análise cuidadosa, pode-se identificar os seguintes conjuntos de mínimos conflitos (Seção 3.2.1): $CS_1 = \{r_4, r_5\}$, $CS_2 = \{r_1, r_5, r_6\}$ e $CS_3 = \{r_3, r_4\}$. Esses conjuntos não permi-

Figura 22 – Implementação do C_{UR} inconsistente.

```
Constraint r1 = hdd.eq(TRUE).decompose();
Constraint r2 = driveOptico.eq(TRUE).decompose();
Constraint r3 = memoria.eq(Memoria.QUATRO_GB.getValue()).decompose();
Constraint r4 = uso.eq(Uso.GAMES.getValue()).decompose();
Constraint r5 = gabinete.eq(Gabinete.MINI.getValue()).decompose();
Constraint r6 = ssd.eq(TRUE).decompose();
```

Fonte – Elaborado pelo autor.

tem a determinação de uma solução de configuração, pois $CS_1 \cup C_{KB}$, $CS_2 \cup C_{KB}$, e $CS_3 \cup C_{KB}$ são todos inconsistentes. Também são conjuntos mínimos, pois nenhum de seus possíveis subconjuntos é inconsistente com C_{KB}. O conjunto de diagnósticos esperado como resultado do teste, a ser calculado pelo algoritmo HSDAG, é dado por

$$\text{diagnósticos} = \{\Delta_1 = \{r_1, r_4\}, \Delta_2 = \{r_4, r_5\}, \Delta_3 = \{r_4, r_6\}, \Delta_4 = \{r_3, r_5\}\}\,. \tag{19}$$

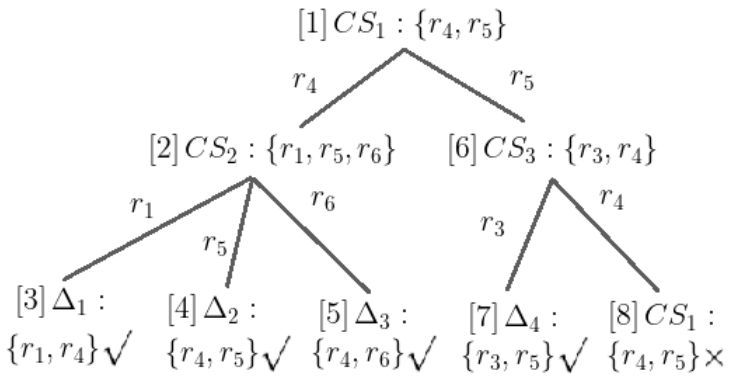
Nota-se que a remoção das restrições (requisitos de usuário), sugerida por qualquer um dos diagnósticos, resolve todos os conjuntos de mínimos conflitos *CS*.

Da mesma forma como foi apresentado na Seção 3.2.2, aplica-se as implementações do HSDAG e *QuickXPlain* para realizar o cálculo de diagnósticos para o estado de inconsistência definido pelos C_{UR} e C_{KB} das equações 18 e 17, obtendo-se a árvore de busca retratada na Figura 23. A ordem de exploração é identificada pelos números entre colchetes. Como o HSDAG não implementa recomendação (ordenação de diagnósticos), a exploração segue o rumo de uma busca em largura convencional. Em cada nodo, tem-se o conjunto de mínimos conflitos *CS* retornado pelo algoritmo *QuickXPlain*. A Figura 24 contém o *log* textual da execução do algoritmo *PDiag* com a recomendação desativada (equivalente ao HSDAG), após efetuar o cálculo de diagnósticos para a mesma situação, descrevendo o grafo de busca da Figura 23.

O primeiro *CS* retornado é $CS_1 = \{r_4, r_5\}$. Desse, criam-se duas arestas, que representam a remoção dos elementos r_4 e r_5. Por convenção, a aresta da esquerda é explorada primeiro. Aplica-se o *QuickXPlain* considerando a remoção de r_4, e $CS_2 = \{r_1, r_5, r_6\}$ é retornado. Então, cria-se uma aresta para cada elemento de CS_2, e a primeira aresta é verificada ao aplicar o *QuickXPlain* considerando a remoção de r_1 e r_4, que retorna um conjunto vazio, indicando que nenhuma consistência foi detectada. Logo, $\{r_1, r_4\}$ é o primeiro diagnóstico a ser determinado. O mesmo repete-se para as outras duas arestas provenientes de CS_2, resultando nas determinações de $\Delta 2$ e $\Delta 3$.

Após a determinação de $\Delta 3$, o algoritmo retorna ao primeiro nodo e o caminho representado por r_5 é explorado. Obtém-se $CS_3 = \{r_3, r_4\}$, e desse nodo criam-se duas

Figura 23 – Árvore de busca por diagnósticos com recomendação desativada.



Fonte – Elaborado pelo autor com base em (FELFERNIG *et al.*, 2013b).

Figura 24 – Execução do cálculo de diagnósticos sem heurística.

```
"C:\Program Files\Java\jdk1.8.0_281\bin\java.exe" ...
Conflict Set 1: [ARITHM ([uso = 3]), ARITHM ([gabinete = 1])]
Selected path: not{ [ARITHM ([uso = 3])] }
Conflict Set 2: [ARITHM ([hdd = 1]), ARITHM ([gabinete = 1]), ARITHM ([ssd = 1])]
Adding to path: ARITHM ([hdd = 1])
Adding to path: ARITHM ([gabinete = 1])
Adding to path: ARITHM ([ssd = 1])
Selected path: not{ [ARITHM ([gabinete = 1])] }
Conflict Set 3: [ARITHM ([memoria = 1]), ARITHM ([uso = 3])]
Adding to path: ARITHM ([memoria = 1])
Adding to path: ARITHM ([uso = 3])
Selected path: not{ [ARITHM ([uso = 3]), ARITHM ([hdd = 1])] }
Diagnostic 1: [ARITHM ([uso = 3]), ARITHM ([hdd = 1])]
Selected path: not{ [ARITHM ([uso = 3]), ARITHM ([gabinete = 1])] }
Diagnostic 2: [ARITHM ([uso = 3]), ARITHM ([gabinete = 1])]
Selected path: not{ [ARITHM ([uso = 3]), ARITHM ([ssd = 1])] }
Diagnostic 3: [ARITHM ([uso = 3]), ARITHM ([ssd = 1])]
Selected path: not{ [ARITHM ([gabinete = 1]), ARITHM ([memoria = 1])] }
Diagnostic 4: [ARITHM ([gabinete = 1]), ARITHM ([memoria = 1])]
Selected path: not{ [ARITHM ([gabinete = 1]), ARITHM ([uso = 3])] }
Path contained -> closing node
Diagnostics: [[r1, r4], [r4, r5], [r4, r6], [r3, r5]]
```

Fonte – Elaborado pelo autor.

arestas. Na aresta da esquerda, verifica-se a remoção de r_3 e r_5, obtendo-se $\Delta 4$. Na aresta da direita, verifica-se que o caminho $\{r_4, r_5\}$ já é representado pelo nodo raiz, portanto o nodo 8 é fechado e a busca é encerrada.

A partir do teste apresentado, e também de outros que foram realizados, verificou-se que os algoritmos implementados estavam efetuando corretamente o cálculo de diagnósticos da maneira convencional. A próxima etapa de desenvolvimento, apresentada na seção seguinte, foi a implementação das funcionalidades necessárias para efetuar o cálculo ordenado de diagnósticos, desta forma integrando tecnologias de

recomendação ao configurador de produtos.

5.2 IMPLEMENTAÇÃO DA RECOMENDAÇÃO POR SIMILARIDADE

Para fazer a recomendação de diagnósticos baseada em similaridade, tiveram de ser implementadas as funcionalidades abordadas na Seção 3.3.2.1, da fundamentação teórica, consistindo na elaboração de uma base de configurações completas, definição das métricas de similaridade, criação de uma classe responsável por calcular e armazenar a tabela de similaridade, e a implementação do algoritmo *PDiag*.

5.2.1 Base de configurações completas

Para efetuar a recomendação de diagnósticos com base em similaridade, é necessário ter uma base de configurações completas, a qual é utilizada para calcular o valor de similaridade entre a configuração corrente e as configurações prévias. Para acompanhar o modelo de testes, foi construída a base retratada na Tabela 12, que consiste basicamente numa planilha em formato `.csv`, a ser importada no configurador e salva no banco de dados em memória H2.

Tabela 12 – Base de configurações completas.

uso	gabinete	processador	driveOptico	placaVideo	memoria	hdd	ssd	replicações
basico	mini	dual-core	sim	nao	4GB	sim	nao	1
basico	mini	dual-core	nao	nao	4GB	nao	sim	1
basico	desktop	dual-core	sim	sim	4GB	sim	nao	1
basico	desktop	dual-core	nao	nao	4GB	nao	sim	1
desenvolvimento	mini	quad-core	nao	nao	16GB	nao	sim	1
desenvolvimento	desktop	dual-core	sim	sim	16GB	sim	sim	1
desenvolvimento	desktop	quad-core	sim	nao	16GB	nao	sim	1
desenvolvimento	desktop	quad-core	nao	nao	16GB	sim	sim	1
desenvolvimento	gamer	dual-core	nao	sim	16GB	nao	sim	1
desenvolvimento	gamer	quad-core	nao	nao	16GB	sim	sim	1
games	desktop	quad-core	sim	sim	8GB	sim	nao	1
games	desktop	quad-core	nao	sim	16GB	sim	sim	1
games	gamer	quad-core	nao	sim	8GB	sim	nao	1
games	gamer	quad-core	nao	sim	16GB	sim	nao	1
games	gamer	quad-core	nao	sim	16GB	sim	sim	1

Fonte – Elaborado pelo autor.

A base tem 15 entradas artificialmente construídas, as quais representam configurações (seleções) bem-sucedidas efetuadas em sessões prévias do sistema. O objetivo da recomendação de diagnósticos por similaridade é retornar o estado de configuração da sessão corrente a um estado o mais similar o possível a algum que tenha ocorrido previamente no sistema. Os registros foram gerados com o intuito de representar as seguintes tendências:

- Usuários que selecionam *uso* = *basico* geralmente optam por configurações mais simples;
- usuários que selecionam *uso* = *desenvolvimento* geralmente optam por *gabinete* = *desktop* e não escolhem placa de vídeo;
- usuários que querem *uso* = *games* geralmente escolhem *gabinete* = *gamer* e *memoria* = 16*GB*.

Um ponto a ser observado é que existe uma coluna de *replicações*, que indica quantas vezes uma mesma configuração foi efetuada. Em Felfernig *et al.* (2013b), não existe nenhuma menção quanto à ponderação desse fator na hora de calcular as recomendações por similaridade. Em discussões com a equipe, concluiu-se que o número de *replicações* era um fator importante a ser considerado, mas não faria sentido tentar incorporá-lo no cálculo de similaridade, pois conceitualmente a similaridade entre duas determinadas configurações é sempre a mesma, independente do número de vezes que uma ou outra tenha sido efetuada.

Portanto, decidiu-se postergar a consideração do número de *replicações* para uma atividade a ser desenvolvida futuramente, após o início do processo de integração da aplicação ao *framework*. Com a adição de novas técnicas de recomendação de diagnósticos, seria possível combinar as recomendações, através da abordagem *ensemble-based diagnosis*, abordada na Seção 3.3.2.2.

A geração artificial de registros foi empregada no contexto de testes para gerar a base que fundamenta as recomendações por similaridade. Após o desenvolvimento da interface de usuário e posterior disponibilização da PoC, o intuito é que a base seja atualizada dinamicamente, conforme os usuários utilizam o configurador. Desta forma, as novas configurações submetidas passariam a ser consideradas nas recomendações posteriores.

5.2.2 Implementação do algoritmo PDiag

O algoritmo *PDiag*, como apresentado na Seção 3.3.2.1, consiste basicamente numa pequena alteração a ser introduzida no algoritmo HSDAG. A implementação realizada do algoritmo é apresentada na Figura 25. Para compactar e facilitar a visualização, alguns *prints* que serviam para efetuar o *log* textual da construção do grafo de busca foram omitidos.

O principal detalhe da implementação está na utilização da classe `PriorityQueue`, do próprio Java, para armazenar os caminhos do grafo de busca a serem explorados. Essa classe fornece uma estrutura de fila que admite a utilização de um comparador para definir a prioridade de saída dos itens contidos. Desta forma, a cada vez que um novo item, no caso um caminho do grafo de busca, é inserido na lista, um

Figura 25 – Implementação do algoritmo PDiag.

```
public List<Diagnostic> PDiag(List<Constraint> backgroundConstraints, List<Constraint> userRequirements,
                              List<Constraint> auxiliaryConstraints, boolean withRecommendation) {

    List<Diagnostic> diagnoses = new ArrayList<>();
    int diagnosticNumber = 0;
    PDiagAdvisor advisor = new PDiagAdvisor(userRequirements);
    List<Constraint> conflictSet = QuickXplain.getInstance(model, auxiliaryConstraints)
                                       .qx(backgroundConstraints, userRequirements);

    Queue<List<Constraint>> pathQueue = withRecommendation ? new PriorityQueue<>(advisor) : new PriorityQueue<>();
    pathQueue.addAll(conflictSet.stream().map(Arrays::asList).collect(Collectors.toList()));

    while (!pathQueue.isEmpty()) {
        if(diagnosticNumber == maxDiagnosis) { break; }
        List<Constraint> path = pathQueue.poll();
        List<Constraint> testConstraintsMinusPath = userRequirements.stream().filter(c -> !path.contains(c))
                .collect(Collectors.toList());
        conflictSet = QuickXplain.getInstance(model, auxiliaryConstraints)
                              .qx(backgroundConstraints, testConstraintsMinusPath);

        if (conflictSet.isEmpty() && diagnoses.stream().map(Diagnostic::getPath).noneMatch(path::containsAll)) {
            diagnoses.add(new Diagnostic(path));
        } else {
            conflictSet.forEach(c -> {
                List<Constraint> newPath = new ArrayList<>(path);
                newPath.add(c);
                pathQueue.offer(newPath);
            });
        }
    }
    return diagnoses;
}
```

Fonte – Elaborado pelo autor.

comparador é acionado para definir qual dos caminhos deve ser explorado primeiro. Previamente, na implementação do HSDAG, era usada uma fila convencional, que operava na abordagem *First In, First Out* (FIFO). Na Figura 25, nota-se que o parâmetro `withRecommendation` define se a busca deve ser realizada com ou sem a recomendação (ordenação) de diagnósticos. Caso a busca seja sem recomendação, a `PriorityQueue` não recebe um comparador ao ser instanciada, funcionando efetivamente como uma estrutura de fila convencional.

A Figura 26 contém a implementação do comparador, numa classe que foi denominada de `PDiagAdvisor`. A funcionalidade de comparação é implementada pela função `compare()`, que compara dois caminhos ao realizar uma consulta na tabela de similaridade. Através desta consulta, é possível decidir qual caminhos deve receber a maior prioridade, e assim ordenar a `PriorityQueue`. Algumas linhas foram removidas para facilitar a visualização.

As consultas são realizadas através da invocação do método *getMaxSimilarityValue*, mostrado na Figura 27, provido pela classe que constrói a tabela de similaridade. Este método recebe um caminho (*path*), definido por uma lista de restrições (neste

Figura 26 – Implementação do comparador.

```
public class PDiagAdvisor implements Comparator<List<Constraint>> {

    private final SimilarityTable similarityTable;

    public PDiagAdvisor(List<Constraint> userRequirements) {
        similarityTable = new SimilarityTable(userRequirements);
    }

    @Override
    public int compare(List<Constraint> path1, List<Constraint> path2) {

        double maxSimilarityValuePath1 = similarityTable.getMaxSimilarityValue(path1);

        double maxSimilarityValuePath2 = similarityTable.getMaxSimilarityValue(path2);

        return (int) Math.signum(maxSimilarityValuePath2 - maxSimilarityValuePath1);
    }
}
```

Fonte – Elaborado pelo autor.

contexto, requisitos de usuário), e basicamente efetua uma consulta na tabela de similaridade, seguindo exatamente a abordagem que foi apresentada na Seção 3.3.2.1.

Figura 27 – Função de consulta por máxima similaridade.

```
public double getMaxSimilarityValue(List<Constraint> path) {
    // cria uma cópia da tabela de similaridade, a ser usada nesta consulta
    List<Configuration> filteredConfigurations = new ArrayList<>(configurations);
    path.forEach(constraint -> {
        String attribute = extractAttribute(constraint).getKey();
        Integer value = extractAttribute(constraint).getValue();
        configurations.forEach(entry -> {
            try {
                if(entry.getClass().getDeclaredField(attribute).get(entry).equals(value)) {
                    filteredConfigurations.remove(entry);
                }
            } catch (Exception e) {
                e.printStackTrace();
            } }); });
    // da tabela restante, retorna o máximo valor de similaridade
    return Collections.max(filteredConfigurations, Comparator.comparing(Configuration::getSimilarity))
                .getSimilarity();
}
```

Fonte – Elaborado pelo autor.

Antes de iniciar o desenvolvimento da tabela de similaridade, foi necessário implementar as métricas de similaridade, introduzidas na Seção 3.3.2.1.2, bem como definir quais métricas deverão ser utilizadas no domínio de configuração de compu-

tadores (Seção 5.1.2). Como esse último não possui atributos numéricos, apenas enumerações, não existe nenhuma escala intrínseca de valor em seus atributos, e quando o usuário define o tipo do gabinete, por exemplo, supõe-se que o mais similar seja preferido. Por esta razão, a métrica EIB foi utilizada para todos os atributos do modelo de testes. Ademais, por simplicidade, os pesos das variáveis foram definidos de maneira homogênea: sendo 8 variáveis, o peso de cada uma foi definido como 12.5%.

5.2.3 Implementação da tabela de similaridade

Optou-se por não apresentar neste documento o algoritmo que realiza a construção da tabela de similaridade, devido ao seu tamanho. Porém, este pode ser conferido no repositório do projeto, disponibilizado em De Carvalho (2021b). Basicamente, o processo de construção da tabela de similaridade é definido pelas seguintes etapas:

1. Recuperar a base de configurações completas, conforme o domínio de configuração;
2. considerar, das configurações completas, apenas os atributos (colunas) do produto que, na sessão atual de configuração, já foram definidos pelo usuário ou sejam resultantes da propagação de restrições;
3. efetuar o cálculo de similaridade da configuração atual com cada registro da base de configurações completas, em nível de atributo;
4. efetuar a soma ponderada das similaridades previamente calculadas, obtendo assim a similaridade total da configuração atual com cada sessão prévia S_i.

Após a construção da tabela, esta deve ser armazenada em memória, para que as consultas posteriores possam ser realizadas pelo `PDiagAdvisor`, durante a execução do algoritmo PDiag com a recomendação de diagnósticos ativada.

5.2.4 Exemplo de cálculo de diagnósticos com personalização

Para a mesma situação apresentada na Seção 5.1.3, efetua-se o processo de cálculo de diagnósticos, desta vez com a ordenação dos diagnósticos baseada em similaridade. Com a recomendação ativada, ao identificar-se um estado de inconsistência, o primeiro passo realizado pelo configurador é a construção da tabela de similaridade. A Figura 28 mostra a tabela de similaridade com os valores de similaridade em nível de atributo, a partir da base de configurações completas introduzida na Seção 5.2.1.

Observa-se que, como todos os atributos do modelo utilizam a métrica EIB, todos os valores de similaridade calculados em nível de atributo são *booleans*, ou seja, 0 ou 1. Dessa tabela, apenas os valores de similaridade total (coluna da direita) são

Figura 28 – Tabela de similaridade em nível de atributo para a configuração de computadores.

User Requirements:

hdd	driveOptico	memoria	uso	gabinete	ssd
sim	sim	4GB	games	mini	sim

Similarity Table Construction:

Session si	uso	gabinete	memoria	ssd	driveOptico	hdd	similarity
s1	0,00	1,00	1,00	0,00	1,00	1,00	0,500
s2	0,00	1,00	1,00	1,00	0,00	0,00	0,375
s3	0,00	0,00	1,00	0,00	1,00	1,00	0,375
s4	0,00	0,00	1,00	1,00	0,00	0,00	0,250
s5	0,00	1,00	0,00	1,00	0,00	0,00	0,250
s6	0,00	0,00	0,00	1,00	1,00	1,00	0,375
s7	0,00	0,00	0,00	1,00	1,00	0,00	0,250
s8	0,00	0,00	0,00	1,00	0,00	1,00	0,250
s9	0,00	0,00	0,00	1,00	0,00	0,00	0,125
s10	0,00	0,00	0,00	1,00	0,00	1,00	0,250
s11	1,00	0,00	0,00	0,00	1,00	1,00	0,375
s12	1,00	0,00	0,00	1,00	0,00	1,00	0,375
s13	1,00	0,00	0,00	0,00	0,00	1,00	0,250
s14	1,00	0,00	0,00	0,00	0,00	1,00	0,250
s15	1,00	0,00	0,00	1,00	0,00	1,00	0,375

Fonte – Elaborado pelo autor.

consultados pelo `PDiagAdvisor`. Para facilitar a visualização, substituem-se os valores de similaridade em nível de atributo pelo próprio nome dos atributos selecionados em cada sessão. Dessa forma, obtém-se a tabela de similaridade mostrada na Figura 29, a qual representa a estrutura consultada durante a realização do cálculo de diagnósticos.

Ao aplicar o *PDiag* para realizar o cálculo de diagnósticos com recomendação, a árvore de busca da Figura 30 é construída. A ordem de exploração dos nodos é identificada pelos números entre colchetes. A explicação de como funciona a busca com heurística já foi apresentada detalhadamente na Seção 3.3.2.1. Basicamente, os caminhos de maior similaridade são considerados mais relevantes e explorados primeiro. Na execução sem recomendação (Seção 5.1.3), foram obtidos os diagnósticos na ordenação $\Delta 1, \Delta 2, \Delta 3, \Delta 4$, porém agora a ordem obtida foi $\Delta 3, \Delta 1, \Delta 2, \Delta 4$.

Ao analisar os requisitos do usuário da sessão atual, dados por r_1 : *hdd* = *sim*, r_2 : *drive-optico* = *sim*, r_3 : *memoria* = $4GB$, r_4 : *uso* = *games*, r_5 : *gabinete* = *mini* e r_6 : *ssd* = *sim*, em conjunto com a tabela de similaridade da Figura 29, percebe-se que a ordenação resultante faz sentido, pois o diagnóstico $\Delta 3$ recomenda a remoção de r_4 e r_6. Tal remoção leva a configuração atual ao estado mais similar a uma configuração prévia contida na base, no caso s_1, com a qual possui uma similaridade de 50%, sendo provavelmente a configuração desejada pelo usuário atual. Já os demais diagnósticos são, nesse cenário, todos equivalentes, pois remetem a configurações com $37,5\%$ de similaridade.

Figura 29 – Tabela de similaridade para o cenário da configuração de computadores.

User Requirements:

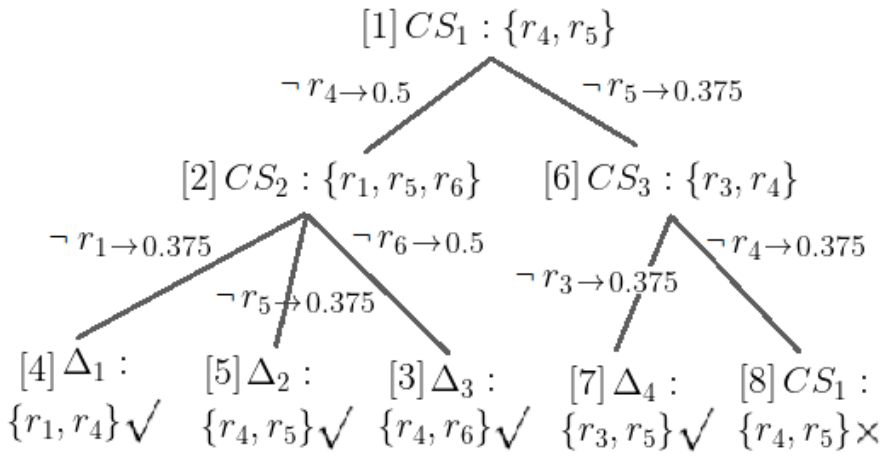
hdd	driveOptico	memoria	uso	gabinete	ssd
sim	sim	4GB	games	mini	sim

Similarity Table:

Session si	uso	gabinete	memoria	ssd	driveOptico	hdd	similarity
s1	básico	mini	4GB	não	sim	sim	0,500
s2	básico	mini	4GB	sim	não	não	0,375
s3	básico	desktop	4GB	não	sim	sim	0,375
s4	básico	desktop	4GB	sim	não	não	0,250
s5	desenvolvimento	mini	16GB	sim	não	não	0,250
s6	desenvolvimento	desktop	16GB	sim	sim	sim	0,375
s7	desenvolvimento	desktop	16GB	sim	sim	não	0,250
s8	desenvolvimento	desktop	16GB	sim	não	sim	0,250
s9	desenvolvimento	gamer	16GB	sim	não	não	0,125
s10	desenvolvimento	gamer	16GB	sim	não	sim	0,250
s11	games	desktop	8GB	não	sim	sim	0,375
s12	games	desktop	16GB	sim	não	sim	0,375
s13	games	gamer	8GB	não	não	sim	0,250
s14	games	gamer	16GB	não	não	sim	0,250
s15	games	gamer	16GB	sim	não	sim	0,375

Fonte – Elaborado pelo autor.

Figura 30 – Árvore de busca por diagnósticos com recomendação ativada.



Fonte – Elaborado pelo autor com base em (FELFERNIG *et al.*, 2013b).

A Figura 31 exibe o *log* textual gerado pelo algoritmo após a sua execução, o qual retrata a árvore de busca exibida na figura anterior.

Com o processo de cálculo de diagnósticos com recomendação por similaridade funcionando, o próximo passo foi o desenvolvimento da interface do configurador com o usuário, a ser apresentado na seção a seguir.

Figura 31 – Execução do algoritmo PDiag.

```
Conflict Set 1: [ARITHM ([uso = 3]), ARITHM ([gabinete = 1])]
Selected path: not{ [ARITHM ([uso = 3])] }, similarity = 0,500
Conflict Set 2: [ARITHM ([hdd = 1]), ARITHM ([gabinete = 1]), ARITHM ([ssd = 1])]
Adding to path: ARITHM ([hdd = 1])
Adding to path: ARITHM ([gabinete = 1])
Adding to path: ARITHM ([ssd = 1])
Selected path: not{ [ARITHM ([uso = 3]), ARITHM ([ssd = 1])] }, similarity = 0,500
Diagnostic 1: [ARITHM ([uso = 3]), ARITHM ([ssd = 1])]
Selected path: not{ [ARITHM ([uso = 3]), ARITHM ([hdd = 1])] }, similarity = 0,375
Diagnostic 2: [ARITHM ([uso = 3]), ARITHM ([hdd = 1])]
Selected path: not{ [ARITHM ([uso = 3]), ARITHM ([gabinete = 1])] }, similarity = 0,375
Diagnostic 3: [ARITHM ([uso = 3]), ARITHM ([gabinete = 1])]
Selected path: not{ [ARITHM ([gabinete = 1])] }, similarity = 0,375
Conflict Set 3: [ARITHM ([memoria = 1]), ARITHM ([uso = 3])]
Adding to path: ARITHM ([memoria = 1])
Adding to path: ARITHM ([uso = 3])
Selected path: not{ [ARITHM ([gabinete = 1]), ARITHM ([memoria = 1])] }, similarity = 0,375
Diagnostic 4: [ARITHM ([gabinete = 1]), ARITHM ([memoria = 1])]
Selected path: not{ [ARITHM ([gabinete = 1]), ARITHM ([uso = 3])] }, similarity = 0,375
Path contained -> closing node
Diagnostics: [[r4, r6], [r1, r4], [r4, r5], [r3, r5]]
```

Fonte – Elaborado pelo autor.

5.3 DESENVOLVIMENTO DA INTERFACE DO CONFIGURADOR

Como visto na Seção 3.3.1, a interface de um configurador de produtos consiste num ponto crítico da aplicação, contendo diversos fatores relacionados à psique humana e processos decisórios, que interferem diretamente no sucesso ou fracasso das vendas. Para esta aplicação para PoC, cujo objetivo é demonstrar um conceito e não efetivamente vender um produto, planejou-se desenvolver uma interface totalmente focada em exibir o estado da configuração atual.

Do ponto de vista técnico, havia uma escassez de conhecimento quanto às tecnologias de *front-end*, cujo estudo foi necessário. O maior desafio foi realizar a integração da interface de usuário com o modelo de configuração que roda no servidor. A *engine* de *Server Side Rendering* (SSR) *Thymeleaf*, muito comum de ser empregada com *Spring*, se mostrou bastante útil para solucionar este problema, pois permitiu com que, a cada requisição do usuário, fossem invocadas funções que verificassem o estado interno do modelo do configurador, possibilitando assim a alteração do estilo dos componentes da página de acordo.

A aplicação *web* do configurador é dividida em quatro telas: a tela de configuração, onde o usuário seleciona os componentes do computador; a tela de resolução de conflitos, onde é apresentado as alternativas de reparo calculadas com os algorit-

mos implementados, a tela de resultados, onde é exibida uma BOM da configuração finalizada, e a tela da base de configurações, que mostra a base de configurações atualizada.

Antes de iniciar o desenvolvimento efetivo das telas, o *software Balsamiq Mockups* foi utilizado para criar os protótipos das telas a serem desenvolvidas, e as versões finais, já implementadas, são apresentadas nas seções seguintes.

5.3.1 Tela de configuração

Para a tela de configuração, foi usada uma estrutura de duas colunas. Para cada componente do modelo do computador, foi inserido um *cluster*, contendo uma imagem, o nome e botões para a seleção. No lugar de botões, foi considerado usar caixas de seleção, mas devido ao baixo número de opções para cada componente, concluiu-se que a utilização de botões deixaria o processo mais intuitivo. No topo da página, um parágrafo descreve o funcionamento do configurado. O botão *restrições* abre uma janela modal, que exibe as restrições da base de conhecimento (Equação (17)), e o botão *base* leva para a tela da base de configurações.

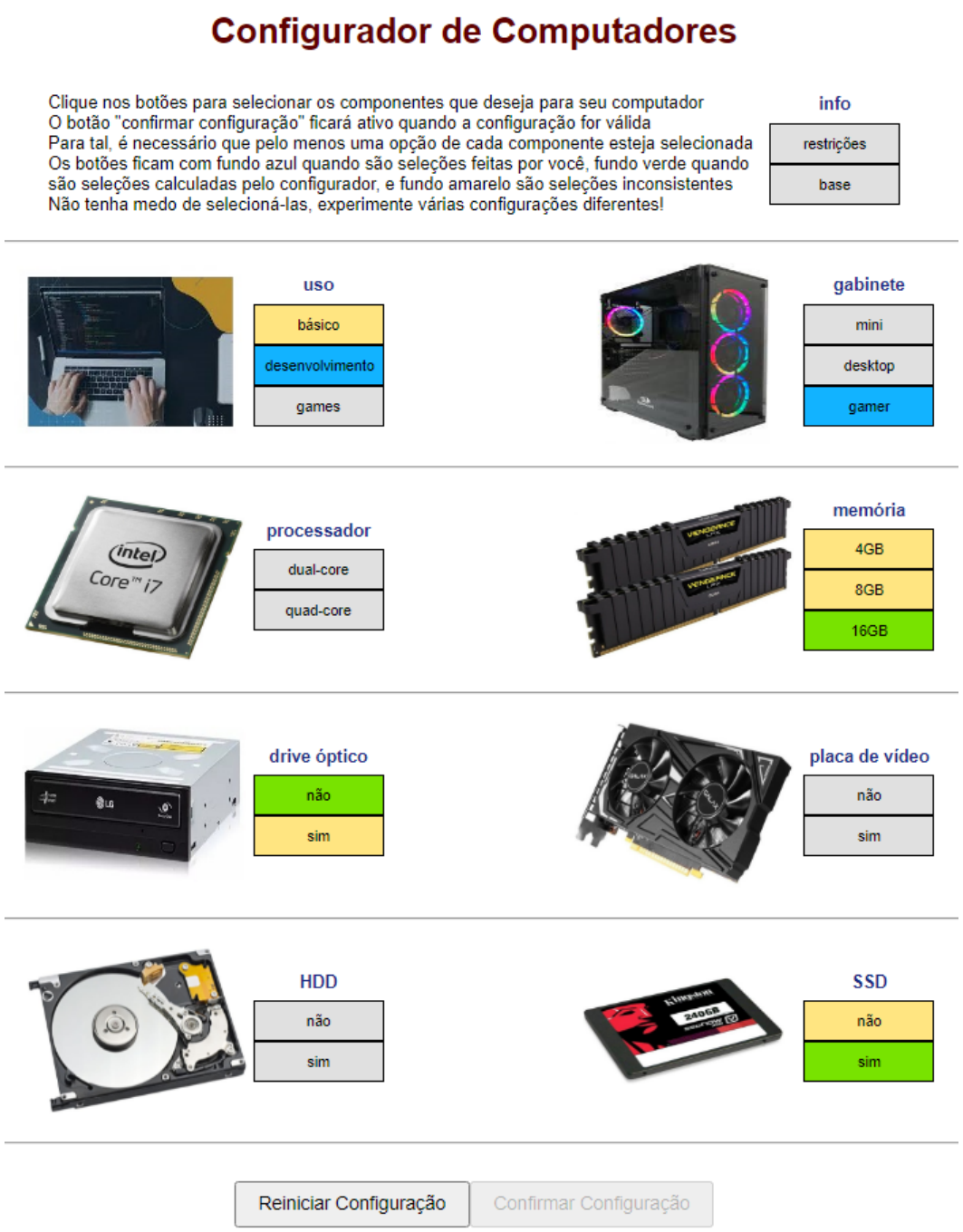
A Figura 32 mostra a tela de configuração implementada. No exemplo, o usuário selecionou as opções *uso* = *desenvolvimento* e *gabinete* = *gamer*. Para propiciar *feedback* do estado de configuração ao usuário, altera-se a cor de fundo dos botões da seguinte maneira: o botão fica com fundo azul quando é uma seleção efetuada diretamente pelo usuário, fundo verde quando é uma seleção calculada pelo configurador (através de propagação), e fundo amarelo se for uma opção prevista como inconsistente, cuja seleção levaria ao processo de resolução de conflitos. Informações sobre a utilização do configurador são apresentadas em um parágrafo, no topo da página.

Observa-se que os componentes cuja seleção restringe mais o escopo da solução, no caso *uso* e *gabinete*. foram postos na primeira linha. Essa prática evita o atingimento de estados de inconsistência. Apesar disso, o atingimento de tais estados é encorajado pelo parágrafo explicativo, visto que é a situação em que ocorre a demonstração das técnicas implementadas. A seguir, é apresentada a tela de resolução de conflitos, a ser exibida sempre que um botão amarelo (que identifica uma opção inconsistente) for selecionado.

5.3.2 Tela de resolução de conflitos

A tela de resolução de conflitos, exibida na Figura 33, pode ser considerada a tela mais importante da aplicação desenvolvida, pois nela são apresentadas as alternativas de reparo (diagnósticos) do estado inconsistência, já ordenados segundo a recomendação por similaridade entre a configuração atual e as presentes na base de configurações completas. No caso da figura, o conflito foi obtido a partir do estado ilustrado na Figura 32, com a seleção da opção *memoria* = $4GB$. Do ponto de vista

Figura 32 – Tela de configuração.

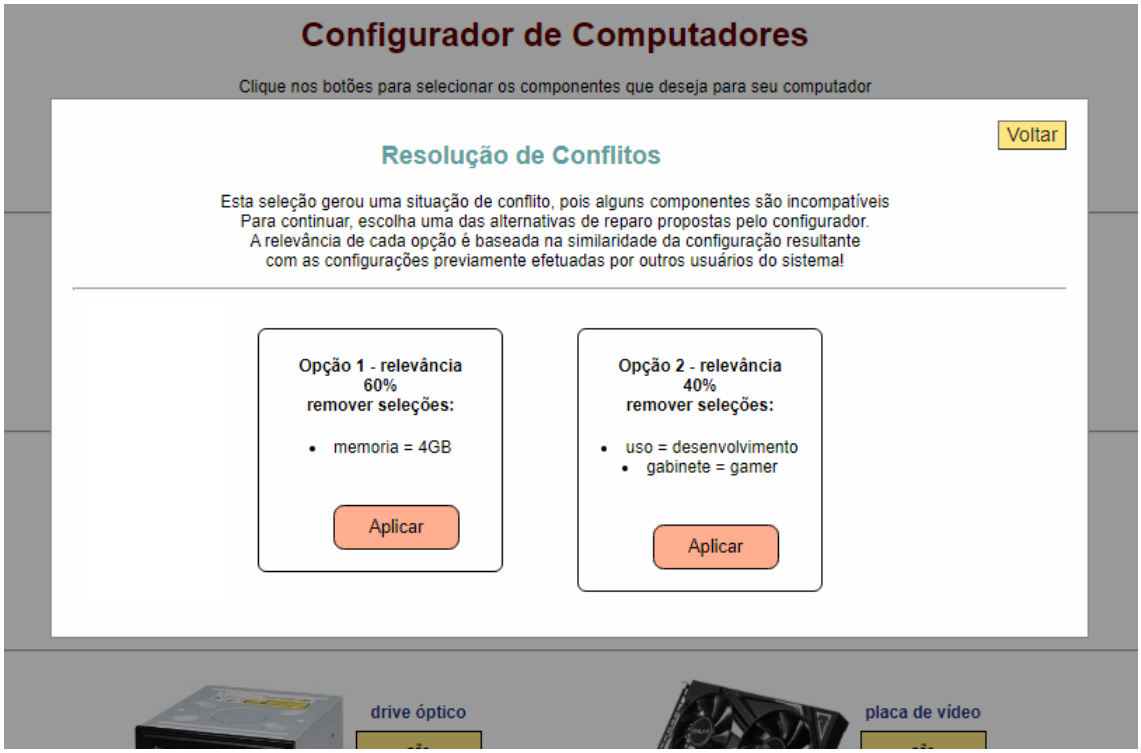


Fonte – Elaborado pelo autor.

estético, foi usado o conceito de tela modal, a ser exibida sempre que um estado de inconsistência é detectado.

Para apresentar ao usuário uma justificativa das recomendações efetuadas, mostra-se um valor de relevância para cada alternativa, proveniente do valor calculado da similaridade obtida com a aplicação da opção, porém normalizado de forma que o somatório das opções resulte em 100%. Um caso específico que foi verificado é quando

Figura 33 – Tela de resolução de conflitos.



Fonte – Elaborado pelo autor.

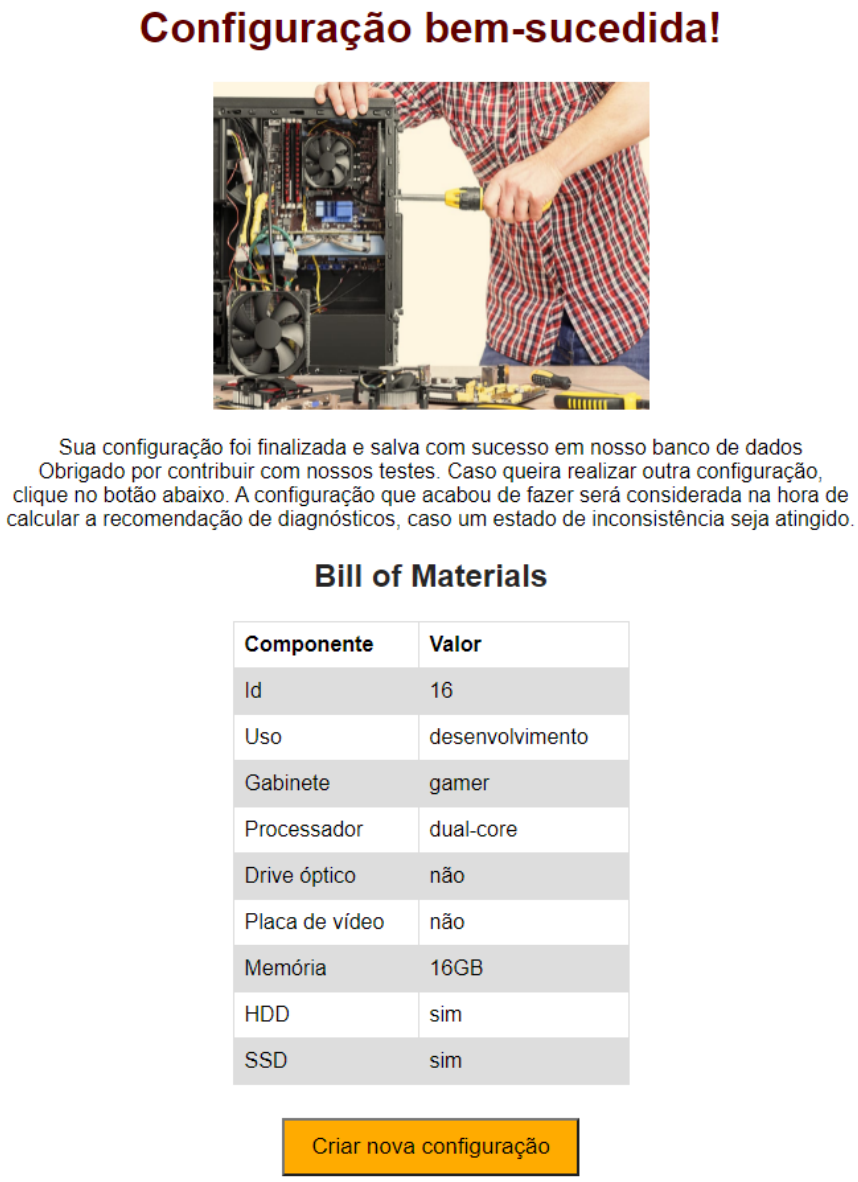
a base de configurações não possui entradas suficientes para efetuar recomendações ao usuário, onde todas as alternativas possuem zero de similaridade. Nesse caso, os diagnósticos são calculados e apresentados da mesma forma como se nenhuma tecnologia de recomendação estivesse sendo utilizada.

5.3.3 Tela de resultados

Assim que o estado de configuração é identificado como completo e válido, isto é, cada componente do computador tenha sido selecionado com valores consistentes (seja pelo usuário ou pelo configurador), o botão “Confirmar Configuração”, situado na parte inferior da tela de configuração, é habilitado. Ao clicar nele, o usuário é redirecionado para a tela de resultados, mostrada na Figura 34.

Na tela de resultados é exibida uma BOM, que lista os componentes que foram selecionadas na configuração confirmada. O usuário também tem a opção de iniciar uma nova configuração. Cada configuração completa é salva imediatamente no banco de dados, já sendo considerada nas recomendações de alternativas de reparo a serem efetuadas nas sessões seguintes.

Figura 34 – Tela de resultados.



Fonte – Elaborado pelo autor.

5.3.4 Tela da base de configurações completas

A Figura 35 mostra a tela da base de configurações completas, dinamicamente atualizada com as configurações finalizadas pelos usuários. Essa tela é acessível através do botão *base*, que leva ao *endpoint* `localhost:8080/base`. Consiste basicamente numa replicação da tabela de configurações armazenada no banco de dados H2, porém convertendo os valores do tipo *Integer* (utilizados pelo *Choco-Solver*) conforme as enumerações correspondentes.

Quando uma configuração é finalizada pelo usuário, caso esta já tenha sido efetuada previamente, a coluna *replicações* é incrementada, evitando assim a duplicação

Figura 35 – Tela da base de configurações.

Base de Configurações

Id	Uso	Gabinete	Processador	Drive óptico	Placa de vídeo	Memória	HDD	SSD	Replicações
1	básico	mini	dual-core	sim	não	4GB	sim	não	1
2	básico	mini	dual-core	não	não	4GB	não	sim	1
3	básico	desktop	dual-core	sim	sim	4GB	sim	não	1
4	básico	desktop	dual-core	não	não	4GB	não	sim	1
5	desenvolvimento	mini	quad-core	não	não	16GB	não	sim	1
6	desenvolvimento	desktop	dual-core	sim	sim	16GB	sim	sim	1
7	desenvolvimento	desktop	quad-core	sim	não	16GB	não	sim	1
8	desenvolvimento	desktop	quad-core	não	não	16GB	sim	sim	1
9	desenvolvimento	gamer	dual-core	não	sim	16GB	não	sim	1
10	desenvolvimento	gamer	quad-core	não	não	16GB	sim	sim	1
11	games	desktop	quad-core	sim	sim	8GB	sim	não	1
12	games	desktop	quad-core	não	sim	16GB	sim	sim	1
13	games	gamer	quad-core	não	sim	8GB	sim	não	1
14	games	gamer	quad-core	não	sim	16GB	sim	não	1
15	games	gamer	quad-core	não	sim	16GB	sim	sim	1
16	desenvolvimento	gamer	dual-core	não	não	16GB	sim	sim	1

Voltar

Fonte – Elaborado pelo autor.

de registros. Caso contrário, um novo registro é adicionado. Nota-se que a configuração com *id* = 16 é a mesma obtida no resultado mostrado na Figura 34.

Observa-se que a tela da base de configurações seria de uso exclusivo do administrador do sistema. Para uma aplicação real de configuração, mais funcionalidades destinadas a esse utilizador poderiam ser desenvolvidas, como um *CRUD* das restrições da base de conhecimento, monitoramento de sessões em andamento, estatísticas de recomendação, etc. Reitera-se que a proposta desenvolvida neste trabalho teve foco exclusivo na melhoria da experiência do usuário final.

No próximo capítulo, serão apresentados os resultados e desdobramentos com a implementação da PoC, bem como um relato de como foi realizado o processo de integração da mesma ao *framework* de configuração da empresa.

6 RESULTADOS

Neste capítulo, são apresentados os resultados obtidos, que incluem uma verificação dos algoritmos implementados, avaliação e recepção da aplicação desenvolvida, e as limitações observadas.

6.1 REPLICAÇÃO DO DOMÍNIO DO ARTIGO BASE

Com o intuito de comprovar que a recomendação de diagnósticos baseada em similaridade estava sendo efetuada corretamente pelos algoritmos implementados, foi desenvolvido um conjunto de testes para replicar o domínio apresentado no artigo Felfernig *et al.* (2013b), referente ao cenário da configuração de automóveis, que foi apresentado na Seção 3.2 da fundamentação teórica.

Para permitir o funcionamento do configurador, tanto com o domínio dos computadores quanto o de automóveis, foi necessário efetuar uma generalização da classe responsável por construir a tabela de similaridade, o que foi feito através do uso de interfaces e classes abstratas. Desta forma, a atividade de replicação do artigo base também ajudou a preparar a aplicação para integração no *framework*.

O teste da recomendação baseada em similaridade replicou de forma idêntica o cenário apresentado na Seção 3.3.2.1.4. A mesma base de configurações completas e conjunto de requisitos do usuário fornecidos no artigo foram utilizados. Os resultados obtidos, ilustrados pelas Figuras 36 e 37, foram iguais aos apresentados no artigo, incluindo o grafo de busca construído.

Figura 36 – Tabela de similaridade obtida para o configurador de automóveis.

User Requirements:

type	fuel	skibag	fourWheel	pdc
city	6l	yes	yes	yes

Similarity Table:

Session si	skibag	fuel	fourWheel	type	pdc	similarity
s1	0,00	1,00	0,00	1,00	1,00	0,600
s2	0,00	1,00	0,00	1,00	0,00	0,550
s3	1,00	0,00	1,00	0,00	1,00	0,450
s4	0,00	0,67	0,00	0,00	1,00	0,083
s5	0,00	0,67	0,00	0,00	0,00	0,033
s6	0,00	0,00	1,00	0,00	1,00	0,350
s7	1,00	0,67	0,00	0,00	0,00	0,133
s8	1,00	0,67	0,00	0,00	0,00	0,133

Fonte – Elaborado pelo autor.

Posteriormente, também foram desenvolvidos testes para replicar o diagnóstico convencional (sem recomendação) e o diagnóstico baseado em utilidade, também ob-

Figura 37 – Execução do PDiag para o configurador de automóveis.

```
Conflict Set 1: [ARITHM ([fuel = 6]), ARITHM ([fourWheel = 1])]
Selected path: not{ [ARITHM ([fuel = 6])] }, similarity = 0,600
Conflict Set 2: [ARITHM ([type = 1]), ARITHM ([fourWheel = 1])]
Adding to path: ARITHM ([type = 1])
Adding to path: ARITHM ([fourWheel = 1])
Selected path: not{ [ARITHM ([fourWheel = 1])] }, similarity = 0,600
Conflict Set 3: [ARITHM ([type = 1]), ARITHM ([skibag = 1])]
Adding to path: ARITHM ([type = 1])
Adding to path: ARITHM ([skibag = 1])
Selected path: not{ [ARITHM ([fuel = 6]), ARITHM ([fourWheel = 1])] }, similarity = 0,600
Conflict Set 4: [ARITHM ([type = 1]), ARITHM ([skibag = 1])]
Adding to path: ARITHM ([type = 1])
Adding to path: ARITHM ([skibag = 1])
Selected path: not{ [ARITHM ([fourWheel = 1]), ARITHM ([skibag = 1])] }, similarity = 0,600
Diagnostic 1: [ARITHM ([fourWheel = 1]), ARITHM ([skibag = 1])]
Selected path: not{ [ARITHM ([fuel = 6]), ARITHM ([fourWheel = 1]), ARITHM ([skibag = 1])] }, similarity = 0,600
Path contained -> closing node
Selected path: not{ [ARITHM ([fuel = 6]), ARITHM ([type = 1])] }, similarity = 0,450
Diagnostic 2: [ARITHM ([fuel = 6]), ARITHM ([type = 1])]
Selected path: not{ [ARITHM ([fourWheel = 1]), ARITHM ([type = 1])] }, similarity = 0,133
Diagnostic 3: [ARITHM ([fourWheel = 1]), ARITHM ([type = 1])]
Selected path: not{ [ARITHM ([fuel = 6]), ARITHM ([fourWheel = 1]), ARITHM ([type = 1])] }, similarity = 0,000
Path contained -> closing node
Diagnostics: [[c9, c6], [c8, c7], [c8, c6]]
```

Fonte – Elaborado pelo autor.

tendo os mesmos resultados de Felfernig *et al.* (2013b). Os testes realizados podem ser conferidos no repositório da aplicação (DE CARVALHO, 2021b), e comprovam que os algoritmos implementados funcionam corretamente. Das abordagens apresentadas na Seção 3.3.2.2, apenas a técnica de diagnóstico baseado em probabilidade não foi implementada, porque necessitaria do desenvolvimento de uma classe que armazenasse e tratasse uma base de conflitos, que seria algo atualmente não previsto para o configurador da empresa.

6.2 APRESENTAÇÃO DA POC

Após o desenvolvimento da aplicação, esta foi hospedada na plataforma em nuvem *Heroku*, estando disponível em De Carvalho (2021a). O trabalho foi apresentado ao meu supervisor na empresa e disponibilizado aos colegas da seção para a realização de testes. A avaliação sobre a PoC foi efetuada de maneira informal. Do ponto de vista da empresa, o principal resultado é que a PoC foi bem-sucedida nos seguintes pontos:

- Demonstrar a técnica de recomendação de diagnósticos por similaridade;
- servir como ponto de partida para a adição de novas tecnologias de recomendação ao configurador;

- fornecer novas ideias para a tela de resolução de conflitos;

Apesar da realização de testes unitários que comprovam o correto funcionamento dos algoritmos implementados, nenhum indicador de desempenho foi utilizado para avaliar quantitativa ou qualitativamente os impactos da recomendação de diagnósticos em sessões de configuração realizadas na aplicação. A razão disso é que foi pressuposto que as situações-exemplo passíveis de serem geradas a partir do domínio simples utilizado teriam poucos diagnósticos a serem calculados. Desta forma, seria inadequado tentar "provar" estatisticamente os benefícios introduzidos pela adição da recomendação de diagnósticos, pois o próprio usuário, numa aplicação de domínio simples, tem capacidade de avaliar um conjunto pequeno de alternativas.

Reitera-se que o principal objetivo da aplicação proposta, como apresentado na Seção 1.1, é o de apresentar a tecnologia de recomendação de diagnósticos. A validação da técnica já foi apresentada no artigo utilizado como base para o desenvolvimento do projeto, Felfernig *et al.* (2013b), a partir de duas avaliações empíricas com modelos complexos e centenas de usuários de teste para coleta de dados.

Após a apresentação da PoC, foram pontuadas algumas limitações da aplicação e também fornecidas sugestões de melhoria a serem implementadas nas futuras continuações do trabalho.

6.2.1 Limitações observadas

A aplicação desenvolvida não provê nenhuma forma inteligente de limitar a quantidade de diagnósticos a serem calculados e exibidos ao usuário. Essa limitação poderia ser feita dinamicamente, com a aplicação do conceito de *representative explanations*, apresentado na Seção 3.3.2. Em conjunto, poderia ser disponibilizado um botão "ver mais" na tela de resolução de conflitos, para casos em que o usuário estivesse interessado em ver alternativas além das recomendadas pelo configurador.

Outra limitação observada é que o número de replicações de uma configuração armazenada na base de configurações completas não é considerado na hora de efetuar as recomendações. Como foi discutido na Seção 5.2.1, seria adequado considerar o número de replicações através de outra técnica de recomendação, que não a baseada em similaridade.

Atividades de tratamento das limitações mencionadas foram adicionadas ao *Backlog* do JIRA e implementadas posteriormente à integração da aplicação ao *framework*.

No próximo capítulo, são dadas as conclusões finais do trabalho, bem como as perspectivas futuras e tópicos a serem investigados.

7 CONCLUSÃO

Neste trabalho, foram realizados o projeto e o desenvolvimento de uma aplicação PoC para a recomendação de diagnósticos em configuradores de produtos baseados em restrições, cujo objetivo é o de melhorar a experiência do usuário final ao utilizar o sistema de configuração. Foi apresentada uma introdução sobre a área de configuradores de produtos, e também uma contextualização da empresa onde o trabalho foi realizado, a qual está em processo de transição de seus configuradores baseados em regras para configuradores baseados em restrições. Também foi apresentada a fundamentação teórica do projeto, incluindo a aplicação de CSP para configuração de produtos, o estudo de algoritmos para realizar o cálculo de diagnósticos em estados de inconsistência, e uma investigação das diferentes formas de se adicionar tecnologias de recomendação em configuradores de produtos, com foco nas diferentes abordagens para efetuar a recomendação de diagnósticos.

Como perspectivas futuras, além da resolução das limitações mencionadas na Seção 6.2.1, propõe-se, como continuação natural deste trabalho, a implementação da técnica de diagnóstico baseado em probabilidade (Seção 3.3.2.2), e também a posterior combinação de todas as abordagens de recomendação de diagnósticos implementadas, através da técnica *ensemble-based*. Além disso, sugere-se a investigação de como poderia ser feita uma ponderação dinâmica de cada abordagem, conforme o estado do sistema (através da utilização de hiperparâmetros).

Uma funcionalidade a ser investigada futuramente, para o configurador da empresa, é a configuração baseada em necessidade, a qual já é disponibilizada em configuradores de ponta (Seção 2.3.4), e cuja principal vantagem é viabilizar a utilização do configurador para usuários sem conhecimento técnico.

Como adendo final, gostaria de compartilhar que a realização deste trabalho foi bastante importante para mim, tanto do ponto de vista pessoal quanto profissional. Tive a oportunidade de morar sozinho pela primeira vez, trabalhar numa empresa de grande porte e adquirir novas experiências. Portanto, reitero os agradecimentos iniciais e estendo-os também ao leitor.

REFERÊNCIAS

ANAND, Praharsha. **Nokia's Digital Automation Cloud will power WEG's Industry 4.0 project**. [*S.l.*: *s.n.*], 2021. Disponível em: https://www.itpro.co.uk/business-strategy/automation/359336/nokias-digital-automation-cloud-will-power-wegs-industry-40. Acesso em: 10 mai. 2021.

ATAS, Muesluem *et al.* Towards Similarity-Aware Constraint-Based Recommendation. *In*: 32ND International Conference on Industrial, Engineering & Other Applications of Applied Intelligent Systems. Graz, Austria: Springer, 2019.

BAELDUNG. **Spring Dependency Injection**. [*S.l.*: *s.n.*]. Disponível em: https://www.baeldung.com/spring-dependency-injection. Acesso em: 10 jun. 2021.

BESSIERE, Christian. Constraint Propagation. Edição: F. Rossi. **Handbook of Constraint Programming**, Elsevier, v. 2, p. 29–83, 2006.

BINDER, Alexander *et al.* Big Data Management Using Ontologies for CPQ Solutions. **Procedia Manufacturing**, v. 10, n. 52, p. 307–312, 2020.

BLUMÖHR, Uwe *et al.* **Variant Configuration with SAP**. [*S.l.*]: SAP PRESS, 2011.

BRAILSFORD, S. *et al.* Constraint satisfaction problems: algorithms and applications, 1999.

BRECHNER, Eric. **Agile Project Management with Kanban**. [*S.l.*]: Microsoft Press, 2015.

CARVALHO, Jiane. **Metade do faturamento da WEG vem de suas mais recentes inovações**. [*S.l.*: *s.n.*]. Disponível em: https://epocanegocios.globo.com/360/noticia/2020/10/metade-do-faturamento-da-weg-vem-de-suas-mais-recentes-inovacoes.html. Acesso em: 10 mai. 2021.

CHACON, Scott; STRAUB, Ben. **Pro Git**. 2th. [*S.l.*]: Apress, 2014.

CROWDER, Robert. **Principles of learning and memory**. [*S.l.*]: Psychology Press, 2014.

DE CARVALHO, Christopher. **Aplicação hospedada no Heroku**. [*S.l.*: *s.n.*]. Disponível em: `https://diag-recommender.herokuapp.com/`. Acesso em: 11 jun. 2021.

DE CARVALHO, Christopher. **Repositório do projeto no GitHub**. [*S.l.*: *s.n.*]. Disponível em: `https:/github.com/christopherdec/diag-recommender`. Acesso em: 7 jun. 2021.

FALKNER, Andreas *et al.* Recommendation technologies for configurable products. **AI Magazine**, v. 3, n. 32, p. 99–108, 2011.

FALKNER, Andreas *et al.* Twenty-Five Years of Successful Application of Constraint Technologies at Siemens. **AI Magazine**, v. 4, n. 37, p. 67–80, 2017.

FELFERNIG, Alexander *et al.* An Integrated Environment for the Development of Knowledge-Based Recommender Applications, 2006.

FELFERNIG, Alexander *et al.* Automated repair of scoring rules in constraint-based recommender systems. **AI Communications**, v. 2, n. 26, p. 15–27, 2013a.

FELFERNIG, Alexander *et al.* Consistency-based diagnosis of configuration knowledge bases. **Artificial Intelligence**, Elsevier, v. 152, n. 2, p. 213–234, 2004.

FELFERNIG, Alexander *et al.* Personalized Diagnosis for Over-Constrained Problems. *In*: 23RD International Joint Conference on Artificial Intelligence. [*S.l.*: *s.n.*], 2013b. P. 1990–1996.

FELFERNIG, Alexander *et al.* Personalized user interfaces for product configuration. *In*: INTERNATIONAL Conference on Intelligent User Interfaces. Hong Kong, China: [*s.n.*], 2010.

FELFERNIG, Alexander *et al.* Plausible repairs for inconsistent requirements. *In*: 21ST International Joint Conference on Artificial Intelligence. Pasadena, California, USA: [*s.n.*], 2009.

FELFERNIG, Alexander; STETTINGER, Martin. Conflict Management in Interactive Financial Service Selection, 2015.

FERNANDES, Joaquim. **Conceito de Produto**. [*S.l.*: *s.n.*], 2019. Disponível em: `https://agenciasi.com.br/conceito-de-produto/`. Acesso em: 7 mai. 2021.

GILMORE, James; PINE II, Joseph. **The four faces of mass customization**. [*S.l.*: *s.n.*], 1997. Disponível em: `https://hbr.org/1997/01/the-four-faces-of-mass-customization`. Acesso em: 7 mai. 2021.

GRAFMÜLLER, Leontin; HABICHT, Hagen. Current Challenges for Mass Customization on B2B Markets, 2017.

HASELBÖCK, Alois; SCHENNER, Gottfried. S'UPREME, 2014.

HESSELFELDT, David. **What is a Product Configurator?** [*S.l.*: *s.n.*], 2019. Disponível em: `https://configit.com/what-is-a-product-configurator/`. Acesso em: 7 mai. 2021.

HOTZ, Lothar *et al.* **Knowledge-based Configuration**: From Research to Business Cases. [*S.l.*]: Springer, 2014.

HOTZ, Lothar; GÜNTER, Andreas. KONWERK, 2014.

JANNACH, Dietmar *et al.* Intelligent Support for Interactive Configuration of Mass-Customized Products, 2001.

JANNACH, Dietmar *et al.* **Recommender Systems**: An Introduction. [*S.l.*]: Cambridge University Press, 2010. v. 1.

JORGENSEN, Kaj A. Product Configuration and Product Family Modelling, 2009.

JUNKER, Ulrich. Configuration. *In*: HANDBOOK of Constraint Programming. [*S.l.*]: Elsevier, 2006. v. 2, p. 837–873.

JUNKER, Ulrich. Quickxplain: Preferred explanations and relaxations for over-constrained problems, 2004.

JUSSIEN, Narendra *et al.* Choco: an Open Source Java Constraint Programming Library, 2008.

KAKADE, Shraddha. **What is bill of materials (BOM)?** [*S.l.*: *s.n.*]. Disponível em: `https://searcherp.techtarget.com/definition/bill-of-materials-BoM`. Acesso em: 1 jul. 2021.

MANDL, M. *et al.* Consumer decision making and configuration systems. *In*: KNOWLEDGE-BASED Configuration: From Research to Business Cases. [*S.l.*: *s.n.*], 2014. cap. 14, p. 181–190.

MCSHERRY, D. Incremental nearest neighbour with default preferences. *In*: INTERNATIONAL Conference on Innovative Techniques and Applications of Artificial Intelligence. Cambridge, United Kingdom: [*s.n.*], 2005. P. 246–259.

MOELLER, J *et al.* Product configuration over the internet. *In*: 6TH INFORMS Conference on Information Systems and Technology. Miami Beach, Florida: [*s.n.*], 2001.

NÖHRER, Alexander; EGYED, Alexander. Conflict Resolution Strategies During Product Configuration, 2010.

O'SULLIVAN, Barry *et al.* Representative explanations for over-constrained problems, 2007.

ORSVÄRN, K; AXLING, T. The Tacton view of configuration tasks and engines, 1999.

PILLER, Frank; BLAZEK, Paul. Core Capabilities of Sustainable Mass Customization. **Knowledge-Based Configuration**, 2014.

REITER, R. *et al.* A theory of diagnosis from first principles. **Artificial Intelligence**, v. 1, n. 32, p. 57–96, 1987.

RENNER, Maurício. **WEG segue colecionando startups**. [*S.l.*: *s.n.*], 2020. Disponível em: `https://www.baguete.com.br/noticias/02/07/2020/weg-segue-colecionando-startups`. Acesso em: 10 mai. 2021.

RUSSELL, Stuart; NORVIG, Peter. **Artificial Intelligence**: A Modern Approach. 4th. [*S.l.*]: Prentice Hall, 2020.

SALOWAY, E. *et al.* Assessing the maintainability of XCON-in-RIME: coping with the problem of very large rule-bases, 1987.

TACTON. **CPQ Concepts Explained**: Go Beyond Rules with Constraint-Based Configuration. [*S.l.*: *s.n.*]. Disponível em: https://www.tacton.com/resources/cpq-concepts-beyond-guided-rules-constraint-based-configuration/. Acesso em: 13 mai. 2021.

TACTON. **First Glance at Tacton CPQ**. [*S.l.*: *s.n.*]. Disponível em: https://www.youtube.com/watch?v=Zwh-vWkvocw. Acesso em: 10 mai. 2021.

TACTON. **The CPQ Guide for Elevator and Lift Manufacturers**. [*S.l.*: *s.n.*]. Disponível em: https://www.tacton.com/resources/cpq-guide-for-elevators-and-lifts/. Acesso em: 10 mai. 2021.

TACTON. **What is a Product Configurator?** [*S.l.*: *s.n.*]. Disponível em: https://www.tacton.com/resources/abb/. Acesso em: 10 mai. 2021.

TECHOPEDIA. **Configure Price Quote Software**. [*S.l.*: *s.n.*]. Disponível em: https://www.techopedia.com/definition/29476/configure-price-quote-software-cpq. Acesso em: 10 mai. 2021.

TSENG, Mitchell; JIAO, Roger. Mass Customization. *In*: HANDBOOK of Industrial Engineering. 3rd. Wiley, New York: John Wiley & Sons, Inc., 2001.

WAZLAWICK, Raul. **Análise e Design Orientados a Objetos Para Sistemas de Informação**. 3ª edição. Rio de Janeiro: Elsevier, 2015.

WEG. **Catálogo | WEG**. [*S.l.*: *s.n.*]. Disponível em: https://www.weg.net/catalog/weg/BR/pt/search. Acesso em: 15 jul. 2021.

WEG. **Isto é WEG | WEG**. [*S.l.*: *s.n.*]. Disponível em: https://www.weg.net/institutional/BR/pt/this-is-weg. Acesso em: 10 mai. 2021.

WEG. **WEG em Números | WEG**. [*S.l.*: *s.n.*]. Disponível em: https://www.weg.net/institutional/BR/pt/weg-in-numbers. Acesso em: 10 mai. 2021.

WINTERFELDT, D.; EDWARDS, W. Decision Analysis and Behavioral Research. **Cambridge University Press**, 1986.

WOTAWA, F. *et al.* Conflict Management for Constraint-based Recommendation. *In*: IJCAI 2015. Buenos Aires, Argentina: [*s.n.*], 2015.