Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Entendendo requisitos contextuais a partir de Personas

Danilo José Bispo Galvão 12/0114852

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientador
Prof.a Dr.a Genaína Nunes Rodrigues

Brasília
2019

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Entendendo requisitos contextuais a partir de Personas

Danilo José Bispo Galvão 12/0114852

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Prof.a Dr.a Genaína Nunes Rodrigues (Orientador)
CIC/UnB

Prof. Dr. Célia Ghedini Ralha
UnB

Prof. Dr. João Henrique Correia Pimentel
UFRPE

Prof. Dr. José Edil Guimarães de Medeiros
Coordenador do Curso de Engenharia da Computação

Brasília, 13 de dezembro de 2019

Dedicatória

Dedico esse trabalho a todas as pessoas que investiram na minha carreira, principalmente meus pais, que acreditaram em mim em momentos que nem eu mesmo acreditava. Dedico também a todos os profissionais, estudantes e professores que impactaram profundamente no meu crescimento profissional e pessoal.

Agradecimentos

Agradeço, primeiramente, a Prof. Drª Genaína Nunes Rodrigues, que me confiou esse projeto de bastante complexidade e me guiou para as reuniões com as pessoas certas em momentos de dúvidas.

Em seguida, agradeço a minha família, por todo o apoio que prestaram a mim durante toda a minha caminhada da graduação na UnB, com um agradecimento especial ao meu irmão Marcus André, que me presenteou com o computador com o qual elaborei todos os projetos da minha graduação, inclusive essa monografia.

Agradeço também a minha namorada, Helena Schubert, que, com sua paciência e sabedoria, me motiva a continuar persistindo mesmo nos momentos mais solitários.

Agradeço a Gabriela Félix Solano, Leandro Santos Bergmann e Carlos Joel Tavares, ex-alunos de graduação da UnB que me emprestaram um pouco do seu tempo e conhecimento e me auxiliaram no entendimento do domínio com seus trabalhos passados e reuniões presenciais e vários e-mails trocados.

Por fim, agradeço aos inúmeros amigos que fiz durante a estadia na Universidade de Brasília, pois suas histórias me enriqueceram e são parte do que sou hoje. Aos amigos que me ofereceram caronas tanto para ir quanto para voltar da UnB em dias de chuva, dias ensolarados e dias comuns, aos amigos que fizeram trabalhos comigo e a todos os professores que abrilhantaram não só a minha vida acadêmica, mas também minha vida pessoal.

Resumo

A engenharia de requisitos orientada a objetivos *Goal-Oriented Requirements Engineering* (GORE) define objetivos como um meio de projetar os requisitos de um dado sistema, aliado a essa abordagem, os requisitos contextuais potencializam o alcance de um *software* que atenda melhor às necessidades e demandas de usuários em um determinado contexto. *Personas* são arquétipos de grupos diversos de usuários que podem ou não conhecer o sistema em questão, para identificarmos e modificar requisitos para melhorar a experiência de usuário[1], é possível utilizar as características de uma *persona*, suas necessidades e objetivos com um *software* para elaborar um sistema mais adaptativo e mais efetivo para diversas audiências. [**Hipótese/Problema**] O projeto piStar GODA é uma ferramenta criada para modelar modelos contextuais orientados a objetivos, os quais podem ser utilizados em vários algoritmos para avaliar aspectos diferentes do dado modelo. A modelagem baseada em *personas* é outra abordagem cujo objetivo é determinar se o modelo acomoda as necessidades de diferentes usuários. Ao executarmos o algoritmo de *Achievability*(alcançabilidade, traduzido livremente para o português), é possível identificar melhorias e/ou falhas em nosso modelo contextual de objetivos(em inglês, *Contextual Goal Model* (CGM)) e melhorar a capacidade de reação desse sistema a mudanças em contextos de *personas*. **Ideia principal** Nesse trabalho, propomos adicionar a visão de modelagem baseada em *personas* à ferramenta piStar GODA, assim como o algoritmo *Achievability* para considerar possíveis formas de refinar o sistema baseado nos resultados.**Contribuição** Com essa nova funcionalidade, é gerada a extensão das funcionalidades e também o aceleramento do processo de avaliação utilizando um *website* interativo para projetar as características da *persona* e associá-las com os modelos CGM. **Results** Os resultados foram similares ao de experimentos passados [1] mas também para novos modelos CGM, comprovando que a ferramenta implementada funciona razoavelmente em várias circunstâncias. Os experimentos também mostram a versatilidade da ferramenta, com a criação e avaliação de múltiplas *personas* dentro do piStar GODA.

Palavras-chave: Engenharia de software, engenharia de requisitos, requisitos contextuais, engenharia de requisitos orientados a objetivos, design voltado a usuários

Abstract

Goal-oriented requirements engineering (GORE) defines goals as a way to design our system requirements, in that sense, we also have contextual requirements to achieve a software that answers better to users' demands and necessities in a determined context. *Personas* are an archetype of a particular group of users that may or may not be accustomed to our system [1], in order to identify or change requirements to enhance user experience, we can use a *persona's* characteristics, needs and goals to render a more effective and adaptive system for diverse audiences. [**Question/Problem**] The piStar-GODA project is a tool to model contextual goal-oriented models, which can later be used in several algorithms to evaluate different aspects of said model. Persona-based modelling is another approach to determine if our model accommodates different users' necessities. By running the algorithm of achievability of *Contextual Goal Model* (CGM) goals, we can identify improvements and/or flaws to our CGM and be able to respond to changes in persona's contexts. [**Main Idea**] In this presentation, we propose adding Persona-based modelling to the piStar-GODA tool, as well as the Achievability algorithm to assess possible ways to improve our system based on the results. [**Contribution**] With this new function, we extend piStar-GODA's functionalities and also hasten the process of evaluation by using a front-end interactive website to design our personas characteristics and associate them with our CGMs. **Results** The results were similar for past experiments [1] and also for new goal models, proving that the tool performs reasonably under several circumstances. The experiments also show the tool's versability, by creating and assessing *personas* by submitting them to multiple evaluations within piStar-GODA.

Keywords: Software engineering, requirements engineering, contextual requirements, goal-oriented requirements engineering, user-centred design

Sumário

1 Introdução **1**
1.1 Problema e hipótese . . . 2
1.2 Motivação . . . 3
1.3 Objetivos . . . 3
1.4 Organização do Trabalho . . . 4

2 Referencial Teórico **5**
2.1 Engenharia de requisitos orientada a objetivos . . . 5
2.1.1 O modelo CGM . . . 6
2.2 *Persona* . . . 6
2.2.1 Atributos . . . 8
2.2.2 *Facts*,*World Predicates* e *Statements* . . . 8
2.2.3 Contextos . . . 9
2.3 PiStar GODA . . . 10
2.4 Tecnologias utilizadas . . . 10
2.4.1 Java . . . 11
2.4.2 Javascript . . . 11
2.4.3 Backbone.js . . . 11
2.4.4 Spring Boot e REST . . . 12

3 Contribuição **13**
3.1 Objetivos da *persona* . . . 13
3.2 Arquitetura do projeto . . . 13
3.2.1 Diagrama de atividades . . . 14
3.2.2 Arquitetura de módulos . . . 14
3.2.3 Front-end e Back-end . . . 15
3.3 *Achievability* . . . 17
3.3.1 Execução do *Achievability* . . . 19
3.3.2 O algoritmo *Achievability* . . . 20

3.3.3 Requisição da aplicação *web* 21
3.3.4 Classe *Controller* 22
3.3.5 PersonaAchievability e TreeBooleanEvaluator 23
3.4 Estendendo o piStar GODA 26
3.4.1 A interface de usuário 27
3.4.2 Criação de fatos 28
3.4.3 Criação de contextos 29
3.4.4 Expressões complexas e *decomposition preview* 30
3.4.5 Criação de novas decomposições 30
3.4.6 Criação de world predicates 32
3.4.7 Pré-visualização de decomposições 32
3.4.8 Associação de contextos com o modelo CGM 33
3.4.9 Criação de *personas* 34

4 Avaliação da Ferramenta **36**
4.1 Estudo exploratório 36
4.1.1 Perguntas do estudo 38
4.2 Resultados 38
4.2.1 Caso 1: Mary Collins 38
4.2.2 Caso 2: Jennifer Smith 40
4.2.3 Caso 3: Dorothy Williams 40
4.2.4 Questões levantadas 41

5 Conclusão e Trabalhos futuros **43**
5.1 Dificuldades encontradas e lições aprendidas 43
5.1.1 Conclusão e trabalhos futuros 44

Referências **45**

Lista de Figuras

2.1 Um CGM do sistema MPERS. 7
2.2 Um exemplo de atributos e objetivos de uma *persona*. 7
2.3 Exemplo de modelagem de contextos. 10

3.1 A relação entre atores e objetivos de *personas*. 14
3.2 Diagrama de atividades do usuário, *front-end* e *back-end*. 15
3.3 Diagrama de classes do projeto. 16
3.4 Diagrama de sequência do fluxo de dados entre pacotes e classes. 18
3.5 Diagrama de classes da comunicação de dados dentro do *back-end*. 21
3.6 Tela inicial do piStar GODA. 27
3.7 piStar GODA com a visão de Personas. 28
3.8 Modal da criação de fatos. 28
3.9 Modal da criação de contextos. 29
3.10 Modal da criação de outra decomposição, junto com a modal de contextos. 31
3.11 Modal de criação de novo *world predicate*. 32
3.12 Modal de contextos após a criação de *world predicate*. 33
3.13 Modal de criação de *personas*. 35

4.1 Modelo CGM do MPERS no piStar GODA. 37
4.2 Exemplo de *Achievability* com outro CGM com a mensagem de erro. 42

Lista de Tabelas

4.1 Contextos válidos para cada persona 37

Lista de Abreviaturas e Siglas

CGM *Contextual Goal Model.*

CRGM *Contextual and Runtime Goal Model.*

CRUD *Create, Read, Update, Destroy.*

CSS *Cascading Style Sheets.*

DOM *Document Object Model.*

DTMC *Discrete-Time Markov Chain.*

GM *Goal Model.*

GODA *Goal-Oriented Dependability Analysis.*

GORE *Goal-Oriented Requirements Engineering.*

HTML *HyperText Markup Language.*

HTTP *HyperText Transfer Protocol.*

JS *JavaScript.*

JSON *JavaScript Object Notation.*

MPERS *Mobile Personal Emergency Response System.*

POJO *Plain Old Java Object.*

RE *Requirements Engineering.*

REST *Representational State Transfer.*

URL *Universal Resource Locator.*

UX *User Experience.*

Capítulo 1

Introdução

Com o crescimento exponencial de complexidade em algoritmos, a engenharia de requisitos tornou-se rapidamente uma área de grande importância na computação. Em grandes projetos, o *design* do *software* precisa ser executado eficientemente, aliando a maleabilidade com uma capacidade de comportar as necessidades de vários usuários com objetivos diferentes.

Uma necessidade do mercado é a de criar sistemas que melhorem a experiência de usuário (*User Experience* (UX), em inglês) e tornem o seu uso mais prático, intuitivo e dinâmico. O *design* voltado a usuários propõe a elaboração de *softwares* com um foco especial nas necessidades e objetivos dos usuários do sistema, onde a elaboração, planejamento e desenvolvimento do *software* é voltado para possibilitar a melhor interação do usuário com o sistema. Com o apoio da engenharia de requisitos (*Requirements Engineering* (RE)), que se dedica a criar uma linguagem ampla, interdisciplinar e aberta a modificações, novas técnicas e métodos são criados para traduzir observações informais em linguagens matemáticas de especificação para requisitos [2].

Nesse aspecto, a engenharia de requisitos baseadas em objetivos (em inglês, GORE) concebe a simplicidade de decompor um objetivo complexo em uma ou mais tarefas, estabelecendo uma zona de compreensão maior entre o cliente e o desenvolvedor, ambos capazes de comunicar em termos práticos. Com uma representação do problema em modelos de objetivos (em inglês, *Goal Model* (GM)), é possível definir uma *interface* gráfica para o usuário, onde o objetivo final do sistema se torna a raiz de uma árvore (portanto um objetivo-raiz) com um número variável de nós filhos que representam os objetivos necessários para que o objetivo final do sistema seja atingido.

Com a adição de contextos, definidos como estados parciais do mundo no qual o sistema opera que influenciam no alcance de objetivos [3], é possível adicionar mais um nível de complexidade ao GM. Muitas vezes a qualidade das interações do usuário com o sistema dependem de um contexto específico, esse conceito normalmente é ignorado em

*i** [4], *Tropos* [5], *KAOS* [6] e foi introduzido como uma forma auxiliar os modelos atuais [3]. Contextos podem mudar a qualidade dos objetivos alcançados ou até mesmo como eles são atingidos, isso é, as tarefas que devem ser executadas.

A ferramenta piStar GODA [7] é uma ferramenta rica onde o usuário é capaz de criar modelos CGM, estabelecendo as dependências entre tarefas e objetivos. A ferramenta ainda possui funcionalidades para avaliação de dependabilidade de modelos orientados a objetivos, com a recente importação das funcionalidades do *Goal-Oriented Dependability Analysis* (GODA) em 2018.

A modelagem voltada a usuários também pode utilizar de abstrações para representar as demandas de clientes em diferentes cenários: introduzindo *personas* -grupos diversos de usuários com necessidades, objetivos e diferentes contextos [8]. Tais entidades interagem com um sistema, de forma que é possível avaliar a robustez e flexibilidade do software em termos de satisfação de classes variadas de clientes. Utilizando o modelo CGM, uma forma de avaliar se a *persona* atende a um modelo orientado a objetivos é verificar se ela é capaz de realizar todos os objetivos e tarefas, levando em conta os contextos específicos, caso existam, de cada um deles. Se essa análise for bem sucedida, podemos afirmar que a *persona* é atendida pelo sistema descrito no modelo.

Um estudo exploratório foi conduzido no artigo [1], onde foram modeladas e avaliadas quatro *personas* diferentes em um modelo CGM representando um sistema de monitoramento médico. O estudo foi conduzido para avaliar a viabilidade e os benefícios do método adotado, e nele foi introduzido um algoritmo que verifica se a *persona* atende aos contextos exigidos pelo modelo é denominado *Achievability*. A abordagem utilizada mostrou que não apenas o método é útil, apontando casos de sucesso e falha nos objetivos corretos, mas também mostrou bons resultados de *performance*, evidenciando que o algoritmo pode ser utilizado como abordagem válida em problemas de modelagem de sistemas voltados a usuários. Contudo, apenas uma árvore feita no modelo CGM foi testada, isso porque ela foi feita de forma estática em uma estrutura de dados dentro do programa para avaliação. De forma análoga, as *personas* também foram definidas de antemão, sendo necessário que o usuário programe manualmente ambas as *personas* e a árvore de objetivos com os contextos correspondentes para a análise do *Achievability* em outros cenários.

1.1 Problema e hipótese

A integração da avaliação de alcançabilidade de modelos CGM sob a perspectiva de *personas* ainda não foi implementada na ferramenta piStar GODA, de forma que a avaliação desses modelos atualmente é feita manualmente ou de forma *ad-hoc* [1] onde a árvore de objetivos e tarefas foi programada manualmente, juntamente com as proposições lógicas

de contextos de cada nó, além dos contextos que a *persona* ativa. Com a integração dessa abordagem será possível computar com facilidade quais áreas de um modelo CGM possuem deficiências ou acomodam uma quantidade razoável de *personas*, permitindo o refinamento do modelo para inúmeras *personas* e tornando a experiência de uso desses usuários mais aprazível. A hipótese é que é possível integrar o processo de avaliação de *personas* em um modelo CGM dentro do piStar GODA, tornando a análise dessas *personas* mais viável para a condução de análises sistemáticas em um ambiente integrado.

1.2 Motivação

A ferramenta PiStar GODA [7] possibilita a produção de modelos de objetivos contextuais (*Contextual Goal Model (CGM)*, em inglês) e a realização de análises em cima deles. Contudo, a visão de *personas* não está integrada à ferramenta.

Com o estudo de viabilidade publicado no Artigo [1], é evidente que uma integração da modelagem baseada em *personas* é necessária para simplificar o processo de avaliação e tornar a avaliação de mais modelos de objetivos viável. A extensão da ferramenta para acomodar essa nova funcionalidade possibilitaria uma análise mais célere de um ou mais modelos de objetivos, aliados a múltiplas de *personas*, facilitando a verificação de robustez e/ou deficiências nos modelos, a condução de estudos de caso e os estudos dos níveis de atendimento de um determinado sistema.

Se a visão de *personas* for adicionada ao piStar GODA e o *Achievability* for incluído como um algoritmo que verifica se a *persona* selecionada atende aos contextos exigidos pelo modelo atual desenhado, mais análises podem ser realizadas. Adicionalmente, o esforço necessário para conduzir experimentos com essa abordagem iria diminuir.

1.3 Objetivos

Este trabalho tem por objetivo apresentar a nova extensão na ferramenta PiStar GODA como alternativa para avaliação de *personas* em modelos CGM. Os objetivos específicos das funcionalidades são divididos nas seguintes etapas:

1. O usuário ser capaz de cadastrar, editar ou apagar fatos contextuais

2. Após a criação de um ou mais fatos contextuais, poder conceber *world predicates* e contextos a partir deles

3. Associar esses contextos com tarefas e objetivos no modelo CGM

4. Criar uma *persona* e associar os contextos válidos para ela

5. Verificar, utilizando o algoritmo *Achievability* [1], isso é, se a *persona* é atendida pelo modelo, e, em caso negativo, informar qual dos objetivos ou tarefas não foram alcançados.

1.4 Organização do Trabalho

O Capítulo 2 descreve o referencial teórico necessário para compreender os conceitos gerais que foram apenas introduzidos nesse capítulo.

O Capítulo 3 aborda a contribuição proposta por esse trabalho, ou seja, como é realizada a criação de modelos de objetivos com seus respectivos contextos e a extensão na ferramenta original e a criação de *personas* dentro dela.

O Capítulo 4 trata dos resultados da implementação do algoritmo *Achievability* simulando o estudo (descrito no Artigo [1]) como um estudo exploratório.

O Capítulo 5 traz a conclusão e os resultados e apresenta propostas de futuros trabalhos que foram observados durante o desenvolvimento do projeto atual.

Capítulo 2

Referencial Teórico

Essa seção apresenta o referencial bibliográfico utilizado como base para a condução desse projeto. Serão discutidos os referenciais teóricos de engenharia de requisitos, os fundamentos de um modelo orientado a objetivos, as abstrações propostas e as suas respectivas relevâncias para demonstrar a importância científica do trabalho conduzido.

2.1 Engenharia de requisitos orientada a objetivos

A Engenharia de Requisitos (*Requirements Engineering* (RE), em inglês) [9] define processos, metodologias, linguagens e arcabouços que auxiliam as atividades de elicitação, modelagem, análise e especificação de requisitos. Tal processo busca elucidar a viabilidade e as restrições em tempo de projeto do sistema e garantir que ele seja capaz de atender as necessidades esperadas pelos seus usuários, bem como ser flexível e aceitar novas funcionalidades a partir de novos requisitos.

Com uma emergente necessidade de softwares cada vez mais personalizáveis para o usuário, isso é, que proporcionem ao usuário uma experiência que satisfaça suas necessidades e desejos respectivos, a engenharia de requisitos orientada a objetivos utiliza objetivos para elaboração, especificação, negociação e documentação dos requisitos no desenvolvimento de sistemas [3]. De forma que os *Goal Model* (GM) demonstram quais objetivos devem ser alcançados e as tarefas a serem elaboradas para o cumprimento daqueles.

Uma das alternativas dentro da engenharia de requisitos é realizar a elucidação de requisitos da forma mais transparente possível para os *stakeholders* do projeto. Dessa forma é possível representar intuitivamente o atendimento às necessidades e expectativas, a partir de uma visão orientada a objetivos [2].

A visão orientada a objetivos é uma opção inteligente, pois consegue simplificar tarefas complexas em uma linguagem compreensível para todos os envolvidos em um projeto. Tais objetivos são expressos como uma decomposição de tarefas para serem alcançados,

facilitando a leitura, aumentando a possibilidade de colaborações para o sistema em tempo de projeto e permitindo uma percepção mais clara do desenvolvimento do sistema em questão.

A análise orientada a objetivos propõe, portanto, uma abordagem prática para a elucidação de requisitos, explicitando a relação direta entre um objetivo e as tarefas necessárias para seu alcance. Por meio de um modelo com regras estabelecidas, é possível criar modelos robustos representando sistemas complexos e analisar possíveis refinamentos para completar objetivos de formas alternativas.

2.1.1 O modelo CGM

O *Contextual Goal Model* (CGM) é uma extensão do GM que adiciona a ideia de contextos ao modelo [3]. Os contextos funcionam como estados que alteram a qualidade dos objetivos a serem alcançados. Demonstrando, portanto, a relação explícita entre objetivos e as estratégias para que sejam atingidos com base nos contextos nos quais eles estão inseridos.

Por exemplo, um contexto pode ser, em um sistema inteligente de jardinagem, o estado atual de saúde da grama do jardim, o clima (chuvoso, árido, etc) ou a umidade do ambiente. Porém, os contextos precisam ser relevantes, ou seja, importantes para o sistema para serem incluídos no modelo. Neste exemplo de um sistema de jardinagem, um contexto inválido seria se o dono do sistema de jardinagem gosta de sorvete, o que não interfere no funcionamento de um sistema de jardinagem.

A Figura 2.1 exibe um exemplo de um CGM conhecido como *Mobile Personal Emergency Response System* (MPERS): Um software responsável por atender a emergências em um ambiente controlado de assistência a pacientes específicos, onde o objetivo-raiz é conhecido como "*respond to emergency*", sendo esse o objetivo final do software. Caso esse objetivo seja realizado, temos que o modelo atual atende à *persona* em questão. Também podemos ver as proposições lógicas com os contextos que são necessários para o alcance dos objetivos e/ou tarefas anotados ao lado deles.

2.2 *Persona*

Uma *persona* é um personagem fictício que representa um grupo de usuários que utiliza um determinado sistema, com atributos e objetivos independentes a serem realizados, podendo ser ou não atendidas pelo sistema em questão [8]. O conceito de *personas* favorece o *design* voltado a usuários, contando com uma visão mais humanizada do sistema, adicionalmente, proporciona uma personalização de experiência de usuário onde as necessidades e desejos de cada usuário sejam atendidas de forma quase singular dentro do ambiente do sistema.

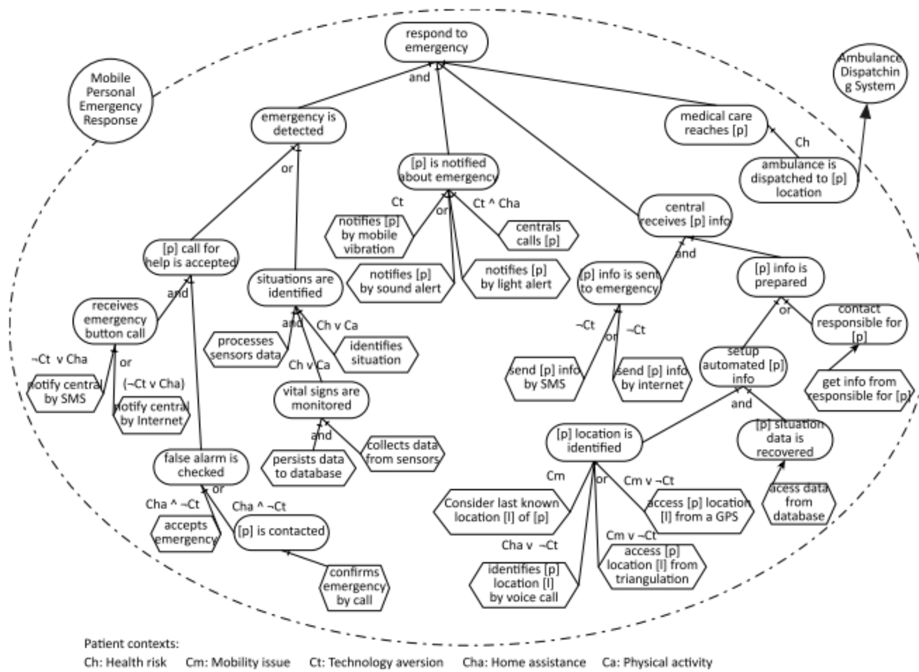

Figura 2.1: Um CGM do sistema MPERS (Fonte: [1]).

Mary Collins (Persona 1)

Age: 70 years
Profession: Retired

Attributes

- Live alone in a small house;
- Does not have a houserkeeper, only diarist every 15 days;
- Does other household chores;
- Fell at home once, but did not fracture any bone;
- Has osteoporosis type 2 at an early stage;
- Has diabetes, high blood pressure and heart problems;
- She is diurnal but wakes up twice at night to go to the bathroom;
- Has 2 childrens who lives in their homes;
- Don't have Wi-Fi at home.

Goals

- To avoid frustating experiences with technologies;
- To not to worry with her children;
- To feel safe by not falling down at home;
- To have quality of life.

Figura 2.2: Um exemplo de atributos e objetivos de uma *persona* (Fonte: [1]).

A elaboração de sistemas utilizando *personas* também facilita a detecção de requisitos não previstos durante as fases de conceitualização e pesquisa de usuário do sistema [1],

acomodando maior diversidade de usuários dentro dele. Uma *persona* é caracterizada pelos seus objetivos e atributos. Como demonstra a Figura 2.2, os objetivos e atributos da *persona* constituem as características que serão utilizadas para a criação do modelo concreto dentro do sistema.

2.2.1 Atributos

Para a modelagem dos requisitos, é necessário que estejam claros as necessidades e características da *persona*, essas serão definidas em **atributos**, que são colocados juntamente dos objetivos da *persona* ao utilizar um dado sistema. Essa abordagem permite que o modelo CGM a ser utilizado possa compreender, com visão humanizada de seus requisitos, as propriedades necessárias para atender aos seus usuários, impactando em futuras decisões do modelo do sistema, permitindo que as funcionalidades priorizem modificações em prol do usuário [1].

Na Figura 2.2, uma paciente do sistema MPERS possui como um de seus atributos: "Não gosta e não conhece nenhuma tecnologia"(traduzido para o português), essa característica impactará futuramente durante a criação de contextos. Os atributos de uma *persona* são transformados, quando relevantes, em *contextual facts*, ou fatos contextuais.

2.2.2 *Facts*, *World Predicates* e *Statements*

Como formalizado no Artigo [1], a descrição de atributos em fatos contextuais é feita através das seguintes etapas:

1. i é o id da *persona* do grupo em questão
2. $A_i \in A_1, A_2, ..., A_n$, onde A é um grupo de atributos com em variáveis nominais de i
3. Cada atributo de A_i pode ter um um fato contextual correspondente F_j, onde $i \leq j$
4. $i = \bigcup_{n=1}^{j} F_n$, a *persona* i é caracterizada como a união de F_j fatos contextuais

Quando se torna um fato contextual da *persona*, ele pode ser adicionado a uma modelagem de contextos, isso é, ele poderá constituir a proposição lógica necessária para ativar um contexto.

World Predicates são proposições lógicas que são classificadas em 2 categorias: *statement* e *fact* (*contextual fact*). Essas duas categorias são definidas pela sua verificabilidade do ator e são separadas formalmente da seguinte forma pela proposição de [3] traduzida para o português:

- ***Definição** 3 (Fact)* um *world predicate* F é um *fact* para um dado ator A se, e somente se, F puder ser verificado por A.
- ***Definição** 4 (Statement)* um *world predicate* S é um *statement* para um dado ator A se, e somente se, S **não** puder ser verificado por A.

Um *world predicate* é, de forma prática, uma proposição lógica de fatos (*Statement* ou *Fact*). Um *fact* é verificável por um ator de forma determinística, não subjetivamente. Em outras palavras, se um fato é verdadeiro para um ator, ele deve ser válido para outros [3].

Os objetivos (*goals*) podem ser visto em duas facetas, sendo especificadas, quando necessário:

1. Objetivos do modelo: dizem respeito ao modelo CGM, que são o elemento de objetivo no modelo, alcançadas através da realização das tarefas necessárias no sistema em questão.
2. Objetivos da *persona*: dizem respeito aos objetivos que a *persona* deseja alcançar, podendo ser compatíveis ou não com o sistema modelado.

2.2.3 Contextos

Contextos são definidos como uma representação parcial do mundo que abrange o sistema em questão e que contribui ou impõe obstáculos na execução de seus objetivos [3]. A ideia de contextos auxilia a visualização das possíveis alternativas de tarefas que devem ser realizadas para o alcance de um objetivo dentro de um modelo CGM.

Em termos concretos, um contexto é o conjunto de proposições lógicas de fatos e/ou *world predicates*. Na Figura 2.3, um contexto é definido pelo *world predicate* w1 e pela decomposição OR de fatos f1 a f6. Ou seja, para que esse contexto seja ativado por uma *persona*, ela precisa atender aos requisitos mínimos definidos pela decomposição lógica da Figura 2.3. No caso em questão, a *persona* precisa ter algum dos fatos de f1 a f6 **e** o *world predicate* w1, ou seja, pelo menos um dos fatos entre f7 e f9. Em seguida, o contexto será instanciado nas tarefas e objetivos relevantes dentro do CGM para analisar se os objetivos do modelo serão afetados pelos contextos operantes da *persona* [1].

O modelo [1] de um contexto proposto é, portanto, a combinação de *world predicates* e *contextual facts*, e a ativação de um dos contextos pela *persona* depende da estrutura formal definida por cada um deles.

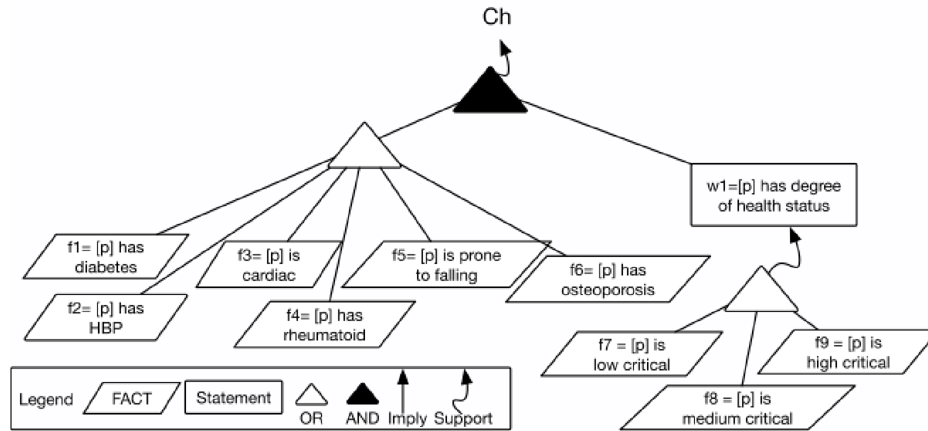

Figura 2.3: Exemplo de modelagem de contextos (Fonte: [1]).

2.3 PiStar GODA

A ferramenta PiStar GODA foi desenvolvida em 2018 no Artigo [7] como a junção das utilidades do piStar e do *Goal-Oriented Dependability Analysis* (GODA). O GODA é utilizado para gerar código *Discrete-Time Markov Chain* (DTMC) [10] a partir de CRGM automaticamente, reduzindo erros decorrentes de análises manuais, e permitindo que os analistas foquem na modelagem [7].

O *piStar* é uma ferramenta de modelagem de modelos CGM que utiliza o padrão *iStar* 2.0 de engenharia de requisitos. O projeto é feito em HTML e JavaScript e conta com a biblioteca Rappid (antigo JointJS) [11] para a modeloção.Já o GODA é um *framework* de análise de dependabilidade com modelos orientados a objetivos.

Com o PiStar GODA, é possível modelar um modelo no próprio *browser*, adicionar propriedades especiais a atores, tarefas ou objetivos e relacioná-los com vários tipos de *links* diferentes, como *AND's*, *OR's*, *qualifications* e etc. O PiStar GODA pode ser executado localmente ou ser hospedado e funciona bem em vários navegadores modernos, como o *Mozilla Firefox* versão 66.0 e *Google Chrome* versão 79.

2.4 Tecnologias utilizadas

Nessa sessão serão descritas brevemente as tecnologias que foram empregadas e faziam ou fazem parte do projeto piStar GODA e são relevantes para o trabalho atual.

2.4.1 Java

Java é uma linguagem de programação orientada a objetos criada pela Sun Microsystems, atualmente mantido pela Oracle. Com o objetivo de prover uma ampla plataforma de desenvolvimento de aplicações portáteis de alto desempenho, é uma linguagem que conta com uma máquina virtual que flexibiliza sua execução em diversos dispositivos diferentes [12].

Nesse projeto, ela faz a interface entre serviços web existentes como o PRISM-DTMC e o PARAM e os elementos do GODA [7] que fazem as análises dos dados fornecidos.

2.4.2 Javascript

JavaScript (JS) é a uma linguagem de programação interpretada de alto nível, feita de acordo com as especificações ECMAScript, comumente empregada para a programação web em conjunção com o HTML e CSS. Algumas de suas características são multi-paradigma e tipagem dinâmica fraca [13]. Sua utilização é tão popular na Web que a maioria de bibliotecas (*libraries*) e *frameworks* é baseada em JS, tais como NodeJS, Angular e Backbone.js. Nesse projeto ela é utilizada para programar os eventos *web* e é utilizada em conjunto com o Backbone.js.

2.4.3 Backbone.js

Backbone.js é uma biblioteca JS *open-source* que se destina a estruturar as aplicações associando elementos (*views*) da página com modelos chave-valor. Ela utiliza o apoio de uma API que possui funções que manipulam os eventos ocorridos nessas *views* de forma que simplifica a organização de modelos e *views* dentro de um contexto web [14].

Um dos precedentes de erros mais comuns no desenvolvimento front-end é a associação de eventos com elementos diretamente do *Document Object Model* (DOM) , o que pode causar um código rapidamente desorganizado e mal-estruturado, com múltiplas referências a apenas um elemento e seus eventos. O Backbone.js permite que todos os eventos associados a um elemento (e seus elementos-filhos) sejam acoplados em uma estrutura, facilitando o desenvolvimento de *features* complexas em elementos na aplicação.

Adicionalmente, com sua associação de modelos com a *view*, a implementação de *Create, Read, Update, Destroy* (CRUD) usando modelos pré-definidos é facilitada, uma das característica mais importantes para o reaproveitamento da biblioteca no projeto.

O Backbone.js trabalha algumas terminologias importantes, que serão mencionadas frequentemente:

1. ***Models***: São os modelos criados dentro do framework, possuem atributos, funções próprias e, se necessário, validadores. São comumente utilizados para implementação de termos recorrentes desse projeto, tais como *facts*,*contexts*, *personas*, etc.

2. ***Collections***: Similares às Collections em Java [15], são listas encadeadas de modelos, mas possuem funções de pilha como **pop** e **push** e funções de ordenação, bem como adição, remoção e atualização de valores de um dos elementos, fornecidas pela própria *framework* do Backbone.js.

3. ***Views***: As views são elementos criados no backbone que associam um *model* ou uma *collection* a um elemento HTML no DOM, permitindo também a associação deles com eventos do elemento.

Em conclusão, podemos ver o Backbone.js como uma *framework* poderosa que funciona como uma abordagem *Model-View* da interface de usuário. O Backbone possui uma dependência (a única *hard dependency*) chamada Underscore.js [16], que realiza a persistência *RESTful* e a manipulação do DOM com anotações em HTML e JS que facilitam o *binding* de modelos com elementos HTML.

2.4.4 Spring Boot e REST

O Spring Boot é uma framework *open-source* Java destinada a aceleração do *deployment* de ambientes de aplicações, com simplificação das configurações envolvidas [17]. É utilizada em nosso projeto para a ativação do *WebApp* em um servidor local e para implementação de microsserviços com REST. REST é um padrão que define a construção de serviços *web*, de forma desacoplada, criado por Roy Fielding [18]. Os serviços que estão de acordo com essa padronização são chamados de *Web services RESTful* (ou apenas *RESTful*).

Capítulo 3

Contribuição

Esta seção se dedica a detalhar a arquitetura e a implementação das novas funcionalidades dentro do piStar GODA e explica em profundidade como elas são realizadas tanto de um ponto de vista de usuário como de um ponto de vista de elaboração e execução, a fim de deixar clara a contribuição realizada nesse trabalho.

Serão brevemente discutidas as principais tecnologias utilizadas nesse projeto para a criação das novas funcionalidades e como são empregadas para desempenhar cada uma das utilidades dentro do projeto. Nesta seção, há a apresentação de partes importantes de código, com a explicação da função de cada etapa dessas partes. Os diagramas desta seção foram feitos com a ferramenta *open-source* Astah UML [19].

3.1 Objetivos da *persona*

A definição dos objetivos de uma *persona* pode ser feito a partir da definição anterior de seus atributos e objetivos, que podem ser convertidos em objetivos (*goals*), *softgoals*, ou tarefas (*tasks*), seguindo a proposta do Artigo [9]. De forma que um ator do CGM pode ser associado com diferentes *personas*, como ilustra a Figura 3.1, onde os objetivos das *personas* P1, P2 e P3 de contextos individuais C1, C2 e C3, respectivamente, podem ser instanciadas individualmente como o mesmo ator dentro de um CGM.

3.2 Arquitetura do projeto

Essa seção se dedica a dar uma visão geral da arquitetura do projeto, uma descrição breve dos arquivos e o que cada um deles faz, com o apoio de figuras que exemplificam a organização adotada. Serão explorados como os arquivos se comunicam e as *interfaces* entre eles.

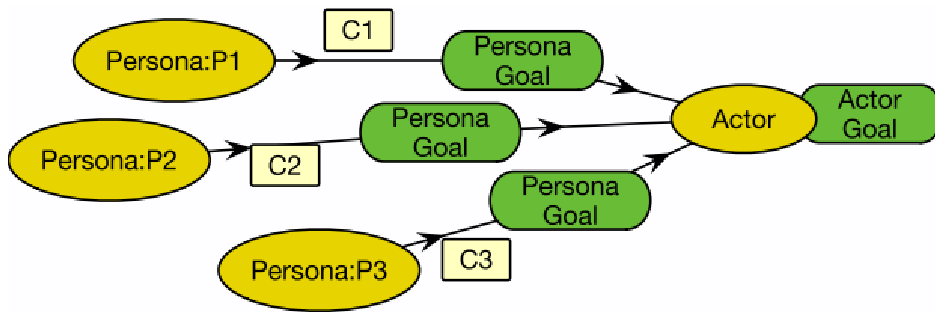

Figura 3.1: A relação entre atores e objetivos de *personas* (Fonte: [1]).

3.2.1 Diagrama de atividades

A Figura 3.2 ilustra, de forma simplificada, como é o fluxo de atividades do usuário, da aplicação *web* (*front-end*) e da aplicação Java (*back-end*) para que o algoritmo seja executado. Primeiramente, o usuário inicia o processo cadastrando fatos, contextos e *personas*, esses dados são guardados parcialmente por estruturas de dados presentes na parte *web*, que definem as listas de dados a serem exibidas para o usuário em cada uma das etapas parciais. Após a criação bem sucedida dos contextos e da associação deles com pelo menos uma *persona*, o usuário pode requisitar a execução do *Achievability*. A aplicação *web* repassa os dados para a aplicação Java, que por sua vez, converte os dados recebidos e executa o algoritmo *Achievability*. Ao finalizar, os dados são retornados para o *front-end* e o usuário pode observar o resultado.

3.2.2 Arquitetura de módulos

A arquitetura de módulos foi feita seguindo a separação simbólica, onde todos os módulos dedicados a criar a nova *interface* gráfica estão no *front-end*, ou seja, contidos dentro do pacote *webapp*. Na Figura 3.3 que apresenta as classes criadas e suas localizações, o pacote *persona*, é responsável pela criação da estrutura JS para as novas funcionalidades da aplicação *web*.

Já o pacote Java é responsável por abrigar algumas das classes que controlam os serviços (*Controller*) como também o *startup* da aplicação (*Application*). Nesse pacote, também há a classe que converte os dados contendo informações da *persona* vindos em *JavaScript Object Notation* (JSON) da aplicação (classe *Persona*). Por fim, no pacote *Persona* está a classe responsável por executar o *Achievability* (*PersonaAchievability*) e o *Evaluator* responsável por avaliar as expressões lógicas recebidas do *front-end* (*TreeBooleanEvaluator*).

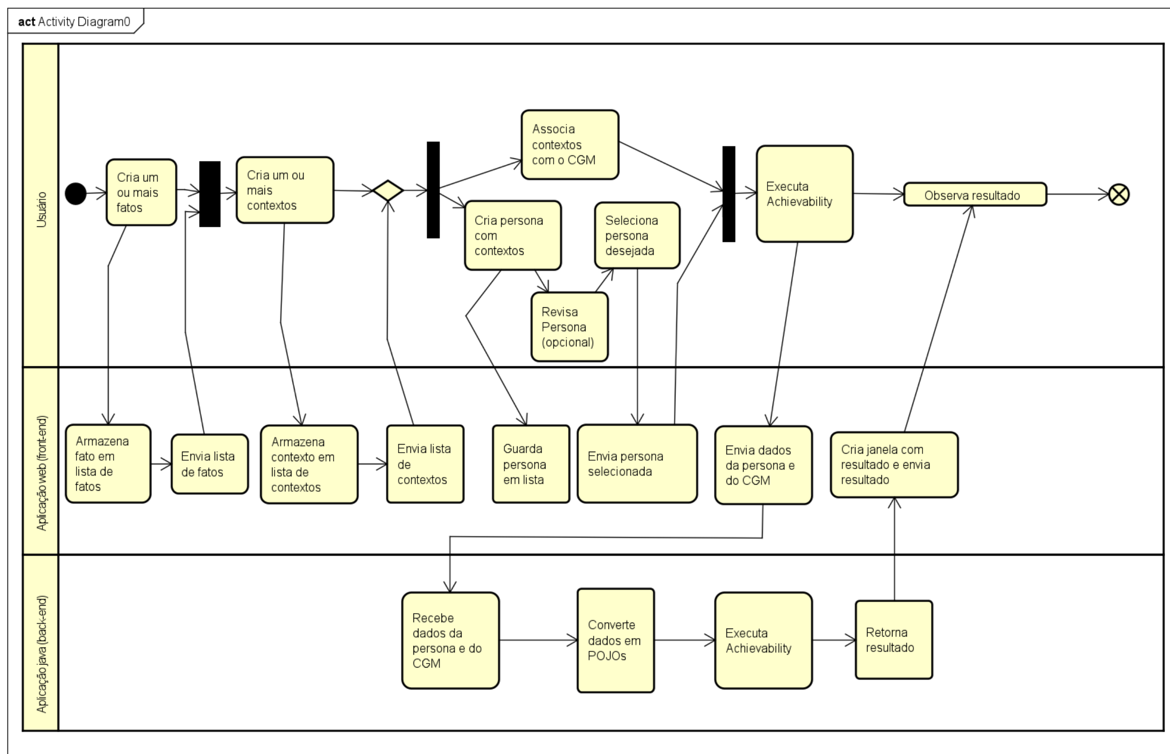

Figura 3.2: Diagrama de atividades do usuário, *front-end* e *back-end*.

3.2.3 Front-end e Back-end

Esse projeto foi separado simbolicamente em duas partes de código, são elas:

- ***Front-end (HTML, CSS e JS)***: Responsável por criar a interface gráfica adicional com as funcionalidades de criação de *personas* e realiza as requisições *HTTP*, além de armazenar modelos de todos os objetos criados e enviados para as requisições na área JavaScript do código
- **Back-end (Java/Spring)**: Responsável por manipular as informações vindas das requisições e implementar o algoritmo *Achievability*. Também é responsável por retornar os resultados obtidos com a *persona* e CGM utilizados.

Esses termos serão utilizados em diversos momentos durante esse capítulo. O fluxo de dados usual do projeto é representado, resumidamente, na lista abaixo:

1. No *front-end*:
   (a) O usuário cria fatos;
   (b) A partir dos fatos, o usuário pode criar contextos e *world predicates*;

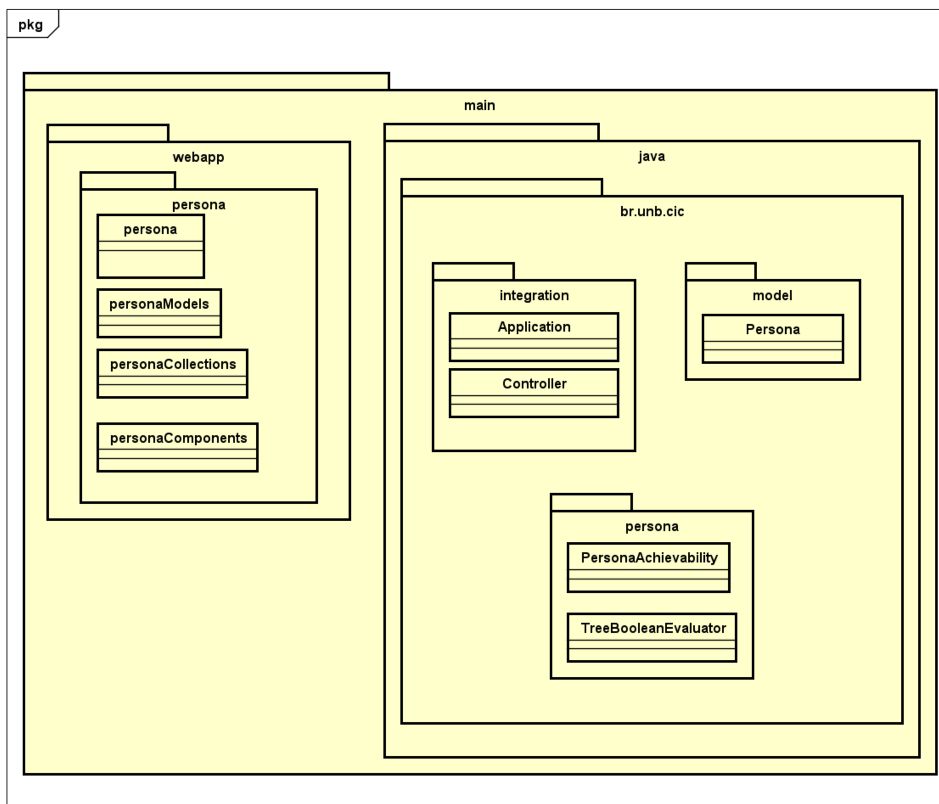

Figura 3.3: Diagrama de classes do projeto.

(c) Em seguida, associa os novos contextos em proposições lógicas a elementos do CGM;

(d) Depois, o usuário cria uma ou mais *personas*;

(e) Por fim, o usuário faz a requisição para rodar o algoritmo *Achievability*, selecionando a *persona* desejada previamente.

2. No *back-end*:

   (a) Os dados da *persona* e CGM são recebidos e convertidos em objetos Java;

   (b) O algoritmo *Achievability* percorre cada nó do objeto Java e verifica as expressões lógicas;

   (c) Dados da *persona* são guardados para retorno;

      i. Em caso de falha: O nó causador do erro é anotado para retorno;

      ii. Em caso de sucesso: Nenhum elemento é anotado;

   (d) Os resultados são retornados para o usuário

3. Ao fim da execução, uma caixa de mensagens é exibida ao usuário com o resultado final.

Mais detalhes das etapas do *front-end* serão mostrados na Seção 3.4, assim como o *back-end* será aprofundado na Seção 3.3.2, onde será explicado o funcionamento do algoritmo *Achievability*. A interface de comunicação do serviço criado para enviar e receber dados entre o *front-end* e *back-end* serão explicitados na Seção 3.3.2.

3.3 *Achievability*

Achievability (alcançabilidade, traduzido livremente para o português) é o nome do algoritmo que une as informações do modelo CGM e dos contextos da *persona* e verifica se a *persona* consegue realizar as tarefas e objetivos dentro de seus respectivos contextos válidos. Com isso procura-se alcançar o objetivo raiz, isto é, o objetivo que tem como nós-filhos todas as tarefas e objetivos do modelo. Caso esse objetivo seja alcançado, é dito que a *persona* pode ser atendida pelo sistema em questão. Caso contrário, o algoritmo também prevê mostrar o nó do modelo CGM que falhou e apresentar o erro para o usuário.

O algoritmo de *Achievability* já está implementado no Artigo [1], onde o objeto de estudo foi um sistema médico de atendimento de pacientes em um ambiente controlado; ao total foram avaliados 4 *personas*. O algoritmo faz uma iteração pela árvore de tarefas e objetivos e detecta se a *persona* possui os contextos ativados válidos para o cumprimento daquela tarefa. Em caso negativo, o algoritmo verifica se isso configura uma falha que impede a *persona* atual de conseguir alcançar o objetivo raiz e, caso a *persona* seja impossibilitada de alcançar o nó raiz por conta da falha em outros nós, o erro e o nó que falhou são exibidos.

Esse mesmo sistema também será avaliado nesse trabalho, para verificar se os resultados serão semelhantes e, portanto, avaliar o método proposto. No algoritmo original [1], a árvore foi criada de forma estática, juntamente com as *personas* que foram avaliadas. O desafio desse trabalho é flexibilizar tanto a criação do modelo CGM quanto a criação de *personas*, tornando esse tipo de avaliação mais dinâmica e criando oportunidades para a análise de diversos modelos combinados com um grupo variável de *personas*.

Como dito anteriormente, o algoritmo foi implementado linguagem Java, com os seguintes objetivos gerais, divididos em etapas:

1. O algoritmo deve receber a *persona* a ser avaliada e o modelo CGM, ambos em formato de string JSON;

2. Com as informações do modelo, o algoritmo deve fazer o *parse* da árvore em um objeto navegável Java (já implementado em outras opções do microsserviço);
3. Os dados da *persona* são convertidos em um objeto da classe *Persona.java*;
4. Os dados são submetidos para uma classe responsável (*PersonaAchievability.java*) que é responsável por realizar a iteração pelo objeto da árvore e verificar se a *persona* atual atende aos objetivos contextuais de cada elemento do modelo CGM;
5. Relatar ao usuário em caso de sucesso ou falha.

Essa implementação é explicada detalhadamente na Seção 3.3.2. A Figura 3.4 também evidencia o fluxo de dados do ponto de vista dos pacotes e classes envolvidos.

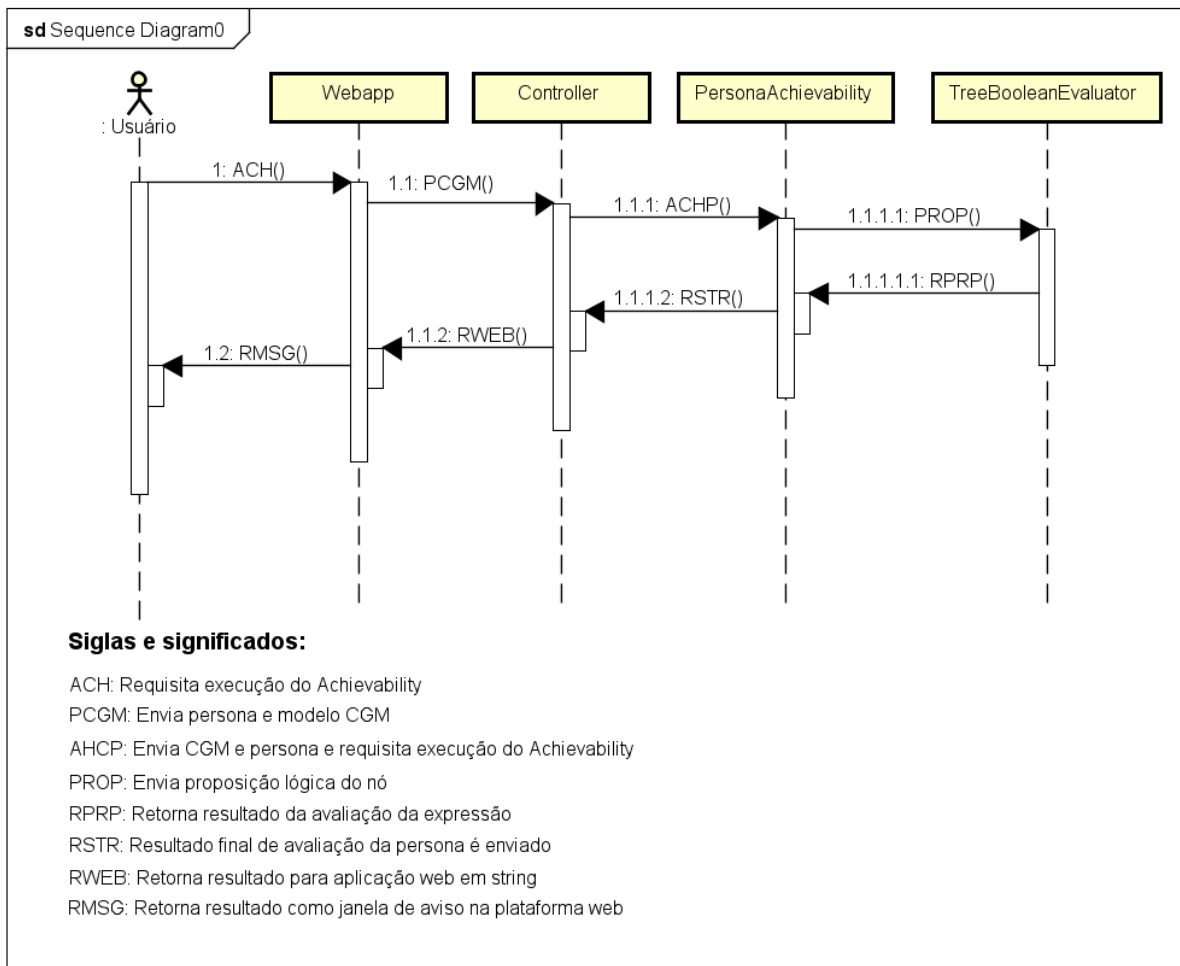

Figura 3.4: Diagrama de sequência do fluxo de dados entre pacotes e classes.

3.3.1 Execução do *Achievability*

A execução funciona da seguinte forma:

1. É importante que o usuário adicione a propriedade *"selected"* com o valor *"true"* no objetivo raiz. Isso faz com o modelo CGM seja interpretado corretamente pelo serviço Java que faz a avaliação da *persona*.

2. O usuário clica no botão *"Run Achievability algorithm"*, presente na barra lateral ou na barra superior, com a diferença de que a *persona* só pode ser selecionada anteriormente na barra lateral, caso não seja selecionada, a última *persona* selecionada será avaliada;

3. Ao clicar, são enviados para o serviço:

    - Dados da persona: JSON contendo os dados da *persona* em questão;
    - Dados do modelo CGM: JSON com um formato pré-definido pela ferramenta para *parsing* correto do modelo CGM.

4. Ao receber os dados do modelo, o código o transforma em classes Java que possuem todas as informações dos elementos presentes e podem ser manipuladas pelo usuário;

5. O algoritmo *Achievability* então obtém o objetivo raiz e inicia um *loop* a partir dele, percorrendo o modelo CGM, fazendo a análise dos contextos anotados com a sintaxe descrita na Seção 3.4.8. Mais detalhes do *loop* serão descritos na Seção 3.3.2

6. Após a análise do *Achievability*, em caso de falha, o nó que gerou a falha é retornado e em ambos os casos de sucesso ou falha, as informações da *persona* são retornadas como um *alert* na plataforma.

Durante o desenvolvimento, foi percebido que a aplicação *web* não poderia fazer uma análise confiável dos dados separadamente, isso porque:

- O JavaScript é uma linguagem com tipagem fraca, ou seja, os objetos criados nela são muito mais suscetíveis a erro que em uma linguagem estruturada, mesmo com o apoio do Backbone.js, capaz de realizar verificações de validade em modelos criados dentro da *framework*.

- A realização de algoritmos complexos na linguagem JavaScript é mais demorado pela dificuldade de *debugging* nessa linguagem, onde originalmente não há tecnologias de *debugging*, mas bibliotecas que criam uma estrutura semelhante, tornando o processo mais complicado e enfadonho pela própria natureza da tecnologia.

Por esses motivos, foi definido que a aplicação *web* não seria utilizada para essa finalidade, como o projeto já apresentava microsserviços para processamento de dados do modelo CGM. Como no caso de algoritmos que geram modelos DTMC nas linguagens PRISM e PARAM [7], o arquétipo foi reutilizado para a implementação de um novo serviço, que implementa o algoritmo *Achievability*.

3.3.2 O algoritmo *Achievability*

Essa seção mostra, de forma detalhada, como a implementação do algoritmo *Achievability* foi realizada desde o seu ponto de entrada até o retorno ao usuário. Trata dos trechos de código mais relevantes para a compreensão da estratégia adotada para criação do algoritmo e as etapas do processamento das informações.

Para suporte à nova funcionalidade no *back-end* do projeto, foram criadas as classes com as respectivas funcionalidades:

- *Persona*: Criada para receber o modelo JSON da *persona* do serviço e traduzi-lo em um *Plain Old Java Object* (POJO).
- *PersonaAchievability*: Implementa o algoritmo de Achievability dentro da plataforma.
- *TreeBooleanEvaluator*: *Evaluator* responsável por obter as expressões como *strings* e fazer o *parse* e análise dela.

As duas últimas classes estão localizadas em um pacote separado, chamado *persona*, a classe *Persona* foi incluída dentro do pacote *model* pois faz a tradução de um JSON em um POJO, como o resto das classes do pacote.

Um diagrama de classes 3.5 foi feito para mostrar como funciona a comunicação de dados com ênfase na *back-end*. O diagrama mostra o pacote *br.unb.cic* (pacote raiz do *back-end*) e seus pacotes internos. Os dados vêm do *front-end* (no diagrama, a interface *webapp*). Os objetos JSON são convertidos em objetos pelas classes *Persona.java* e pelo método *transformeToTao4MeEntities* para o caso da *persona* e para o CGM, respectivamente.

Os dados do contexto da *persona* e os *goals* do CGM são mandados para o construtor da classe *PersonaAchievability.java*, que roda o método *run*, que envia informações das expressões como *strings* a serem avaliadas nos respectivos nós do CGM para a classe *TreeBooleanEvaluator.java*, retornando um resultado a cada avaliação de expressão lógica, em caso de um resultado de falha, o nó responsável pela falha é anotado em uma variável da classe e a análise continua se o erro não comprometer o alcance do usuário ao fim da objetivo raiz. Ao fim da avaliação de todos os nós, o resultado é repassado ao método

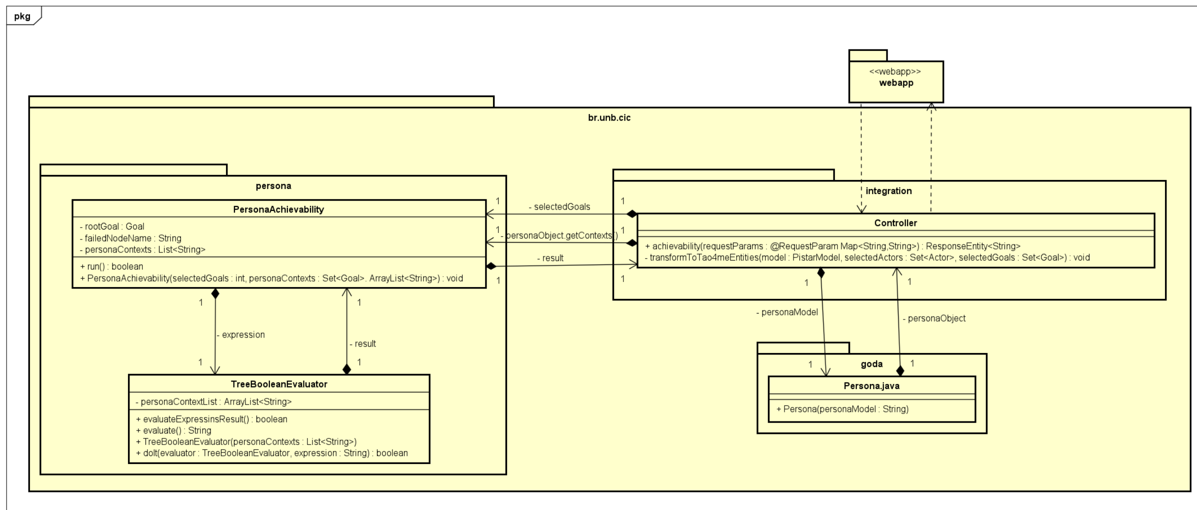

Figura 3.5: Diagrama de classes da comunicação de dados dentro do *back-end*.

Achievability novamente, que repassa o resultado para a parte *web* numa mensagem em forma de *string*.

Como será mostrado em mais detalhes na Seção 3.4, a aplicação *web* é responsável pela adição de componentes a interface gráfica que permitem a criação de fatos contextuais, contextos, *world predicates* e *personas*. Será apresentado a seguir como a aplicação *web* envia os dados necessários para processamento na linguagem Java e o subsequente retorno com os resultados.

3.3.3 Requisição da aplicação *web*

O método da requisição feito em JavaScript está exposto abaixo:

```
1  function runAchievability() {
2      return function () {
3          var model = saveModel();
4          var persona = JSON.stringify(app.selectedPersona);
5          $.ajax({
6              type: "POST",
7              url: '/achievability',
8              data: {
9                  "content": model,
10                 "persona": persona
11             },
12             success: function (data) {
13                 alert(data);
14             },
15             error: function () {
```

```
16                  alert("Error!");
17              }
18          });
19      };
20  }
```

Listing 3.1: Código da função *runAchievability()*

Das linhas 3 a 4, os dois objetos de importância são armazenados em variáveis locais que posteriormente serão enviadas como parâmetros para a requisição, são eles o modelo CGM e os dados da *persona*, convertidos em JSON. Em seguida, as linhas 5 a 7 especificam o tipo de requisição (*POST*) e o endereço *Universal Resource Locator* (URL) da requisição, as linhas 8 a 11 definem os dados a serem enviados no *payload* da mensagem. Por fim, as linhas 12 a 17 especificam os tratamentos em caso de sucesso ou falha da requisição, respectivamente. Em caso de sucesso, os dados retornados pelo método que implementa o *Achievability* em Java são exibidos como um *alert*, em caso de falha, uma janela de alerta também é mostrada, mas com a mensagem *"Error!"*.

3.3.4 Classe *Controller*

No código Java abaixo, é possível observar o trecho de código que representa mapeia a requisição *HyperText Transfer Protocol* (HTTP) que implementa o microsserviço do *Achievability*:

```
1    @RequestMapping(value = "/achievability", method = RequestMethod.
    POST)
2    public ResponseEntity<String> achievability(@RequestParam Map<String
    , String> requestParams) throws IOException {
3        String content = requestParams.get("content");
4        Gson gson = new GsonBuilder().create();
5        PistarModel model = gson.fromJson(content, PistarModel.class);
6        Set<Actor> selectedActors = new HashSet<>();
7        Set<Goal> selectedGoals = new HashSet<>();
8        transformToTao4meEntities(model, selectedActors, selectedGoals);
9        String persona = requestParams.get("persona");
10       Persona personaModel = new Persona(persona);
11       PersonaAchievability achievability = new PersonaAchievability(
    selectedGoals, personaModel.getContexts());
12       String stringReturn;
13       boolean isSuccess = achievability.run();
14       stringReturn = isSuccess? achievability.
    personaAchievabilitySuccess(personaModel) :
15               achievability.personaAchievabilityFailure(personaModel);
```

```java
16          return new ResponseEntity<String>(stringReturn, HttpStatus.OK);
```

Listing 3.2: Método *Achievability* na classe *Controller.java*

No caso do Spring, uma anotação especial é utilizada quando queremos tratar requisições REST da plataforma web fornecida por ele: a linha 1 apresenta a anotação *RequestMapping*, que permite o mapeamento do endereço da função (*/achievability*) para a função logo abaixo dessa anotação, adicionalmente, o método de requisição HTTP, que nesse caso é o POST. Essa parte é responsável por implementar o controlador (*Controller*) responsável por tratar os parâmetros provenientes dessa requisição.

A linha 2 é a assinatura do método, que define o nome da função, seu retorno e parâmetros. Os parâmetros são um *Map* de duas *strings*, correspondentes ao JSON da *persona* e o JSON do modelo CGM. Seu retorno é uma *ResponseEntity<>* de tipo genérico, como nesse caso a resposta a esse método é somente textual, o tipo parametrizado utilizado foi, necessariamente, uma *string*.

As linhas 3 a 5 criam objetos que recebem o modelo e fazem o *parse* dele para um modelo PistarModel, a linha 8 cria os objetos sensíveis para a iteração da árvore (*selectedActors* e *selectedGoals*) através do método *transformtoTao4MeEntities*. Em seguida, os dados da *persona* são obtidos de forma semelhante, mudando apenas o nome do parâmetro esperado, os dados também são convertidos em um objeto Java quando enviados para o construtor da classe *Persona*.

Finalmente, os objetos Java do modelo CGM e os contextos da *persona* são enviados para o algoritmo. Um booleano é criado para armazenar o resultado de sucesso ou falha e uma *string* de retorno é criada para retornar para o usuário a mensagem de sucesso ou erro junto com os dados da *persona*, essas informações são então mostradas num *dialog* da aplicação *web*.

Como dito anteriormente, o serviço recebe dois elementos em JSON: O do modelo CGM e os dados da *persona* a ser analisada. Esses dados vão para uma classe chamada *Controller.java*, que manipula os outros serviços como o PRISM e PARAM [7]. Após recebê-los, o programa faz imediatamente o *parse* desses dois elementos *JSON* a fim de manipulá-los. No caso da *persona*, por exemplo, o JSON é convertido em atributos *string*, como o seu nome e descrição, já os contextos são convertidos em uma lista de *strings*, cada nó correspondendo a um contexto respectivo.

3.3.5 PersonaAchievability e TreeBooleanEvaluator

Após o *parsing* da informação, a classe *PersonaAchievability.java* recebe os dois como parâmetros em seu construtor e os converte em atributos internos da classe. Em seguida, o método *run* presente no Código 3.3 é chamado, onde o vasculhamento da propriedade

creationProperty é feito desde o objetivo raiz (linha 5), onde caso ele tenha e a condição falhar, significa que o código falhou automaticamente, pois falhou no objetivo raiz, sendo inalcançável para a *persona*, caso contrário, ele inicia a iteração pelos *goals* e *tasks* (linha 12).

```
public boolean run() {
// Caso assertion condition na raiz:
        System.out.println("Goal " +
        this.rootGoal.getName() +" is being evaluated");
        if (this.rootGoal.getCreationProperty() != null) {
            if (!evaluateGoalExpression(this.rootGoal)) {
                System.out.println("The current persona
                does not meet the CGM context conditions");
                return false;
            }
        } else {
            if (!iterateThroughGoalsAndPlans(this.rootGoal)) {
                System.out.println("The current persona meets
                the CGM context conditions!");
                return true;
            } else {
                System.out.println("The current persona
                does not meet the CGM context conditions");
                return false;
            }
        }
        return true;
    }
```

Listing 3.3: Código do método *run*

A iteração funciona da seguinte forma: são descobertos o número de nós filhos com usos do método *size* nas listas encadeadas que definem os nós do CGM e é feita uma iteração por cada objeto. Essa iteração verifica se o nó possui a propriedade de contextos dentro de si, em caso positivo, ela é prontamente avaliada com os métodos *evaluateTaskExpression* ou *evaluateGoalExpression*, que são os responsáveis por remover a *string "assertion condition"* de dentro da expressão (pois não faz parte da expressão) e mandá-la para a classe *TreeBooleanEvaluator.java* que verifica se a *persona* atende àqueles contextos. Em caso de sucesso, ele continua a busca até exaurir todos os nós do modelo CGM, fazendo a checagem em todos eles caso a *persona* atenda aos contextos exigidos pelas tarefas ou objetivos.

A verificação é feita a partir da prerrogativa: "Se a *persona* não possuir esse contexto na sua lista, significa que esse contexto é falso para aquela *persona*", os operandos são

verificados e categorizados na expressão como *true* ou *false* com base na distinção feita acima. De forma semelhante, o operando *NOT* verifica se o contexto é verdadeiro para a *persona*, e em caso positivo, o operando é negado, tendo como resultado final *false* e vice-versa.

Finalmente, a proposição lógica é avaliada de forma iterativa até chegar ao final da avaliação da *string*. A avaliação é feita com apoio da biblioteca *Javaluator*, uma biblioteca responsável por avaliar *strings* de expressões lógicas com valores *true* ou *false* para cada um dos seus operandos [20].

No Código 3.4 é possível ver que o método de avaliação de expressões *evaluate* possui como parâmetros o operador e os operandos (linhas 1 e 2), bem como um objeto que representa a porção da *string* que está sendo avaliada, a avaliação funciona iterativamente, ou seja, toda vez que uma porção é avaliada, a próxima é avaliada em seguida em cada chamada do método. A justificativa da avaliação ser realizada desse jeito é de que os operandos são verificados 2 a 2, no caso de operações com *AND* e *OR*, e 1 a 1 no caso de operadores *NOT*, a condição de parada nessa iteração é o fim da expressão e consequente avaliação de todos os operandos em conjunto com os operadores, detectado pelo próprio *Javaluator*. Os operadores são identificados para definir a organização dos operandos, como dito anteriormente, o operando *NOT* é avaliado diferentemente pois, como dito anteriormente, só necessita de um operador para uma expressão completa (linhas 12 a 14), a mesma avaliação de operandos ocorre nas operações *AND* e *OR* (linhas 15 a 18) e em caso negativo para todos os operandos, uma exceção é ativada para operando inválido (linha 20).

Por fim, o resultado da operação parcial é adicionado a uma árvore responsável (linha 3) por dar o resultado final da expressão lógica (linha 27). A expressão é retornada como *string* e o método *endsWith* é utilizado para verificar quais é a cadeia de caracteres que terminam a *string* (*true* ou *false*) e, portanto, definir o resultado final da expressão.

```
1  protected String evaluate(Operator operator, Iterator<String> operands,
2                               Object evaluationContext) {
3          List<String> tree = (List<String>) evaluationContext;
4  
5          String o1 = operands.next();
6          String o2 = "";
7          if(operator != NEGATE){
8              o2 = operands.hasNext() ? operands.next() : o1;
9          }
10         String eval;
11         Boolean result;
12         if(operator == NEGATE) {
```

```
13              result = !getValue(o1);
14          }
15          else if (operator == OR) {
16              result = getValue(o1) || getValue(o2);
17          } else if (operator == AND) {
18              result = getValue(o1) && getValue(o2);
19          } else {
20              throw new IllegalArgumentException();
21          }
22          if(operator != NEGATE){
23              eval = "("+o1+" "+operator.getSymbol()+" "+o2+")="+result;
24          } else {
25              eval = "("+operator.getSymbol()+" "+o1+")="+result;
26          }
27          tree.add(eval);
28          return eval;
29      }
```

Listing 3.4: Método de avaliação de expressões lógicas

Em cada iteração, o *goal* ou *task* é mostrado no output para evidenciar o percorrimento do CGM de forma intuitiva ao usuário, além de mostrar no terminal, em caso de falha, a proposição lógica na qual a *persona* falhou (exemplo disponível no Código 4.1). Para o usuário da plataforma *web*, o percorrimento do modelo é transparente e apenas o nome do elemento que falhou (caso ocorra falha) na avaliação de contextos é mostrado.

Assim como no método *run*, os retornos das iterações são booleanos que indicam se houve falha ou não, então no caso de um retorno *true* de um dos métodos dentro de *run*, significa que houve uma falha, um atributo da classe *PersonaAchievability.java* armazena o nome do nó no qual ocorreu a falha e informa em um método após a execução do *run*. O algoritmo faz, portanto, uma busca *top-down*, iniciando-se pelo objetivo raiz, com a intenção de identificar os erros (caso existam) o mais rápido possível e informá-los ao usuário assim que eles forem identificados.

3.4 Estendendo o piStar GODA

Para a estruturação do código, foram criados quatro módulos na pasta *persona*; uma pasta contida dentro da área *webapp* (aplicação *web*) do projeto:

- ***persona***: Contém as alterações no modelo e encapsula as funções JavaScript que envolvam a modelagem baseada em *personas*. Também possui a requisição para o microsserviço que implementa o algoritmo *Achievability* em Java.

- **personaCollections**: Possui as definições de *collections* utilizadas no projeto. Alguns exemplos são:
  - *FactCollection*: A *collection* de *facts*.
  - *ContextList*: A *collection* de *contexts*.
- **personaComponents**: Esse arquivo tem as implementações de *views*, juntamente com instâncias de *models* e *collections*, criadas para interação entre eles com as *views*.
- **personaModels**: Define concretamente os modelos utilizados no projeto, alguns exemplos são *Facts*, *Context*, *Persona* e *World Predicate*

3.4.1 A interface de usuário

A tela inicial do projeto continua a mesma em comparação ao projeto original, a fim de manter o ideal de segregar a implementação de *personas* do objetivo inicial da ferramenta, não confundir o usuário com tantas opções e deixar a interface mais intuitiva. A extensão da interface seguiu as diretivas do piStar GODA e manteve a língua inglesa como a língua padrão do projeto.

Na Figura 3.6, é possível visualizar a página inicial, com sua barra de ferramentas para inserção de novos atores, objetivos ou tarefas, bem como estabelecer novas relações com os refinamentos *OR*, *AND*, entre outros dentro do modelo CGM.

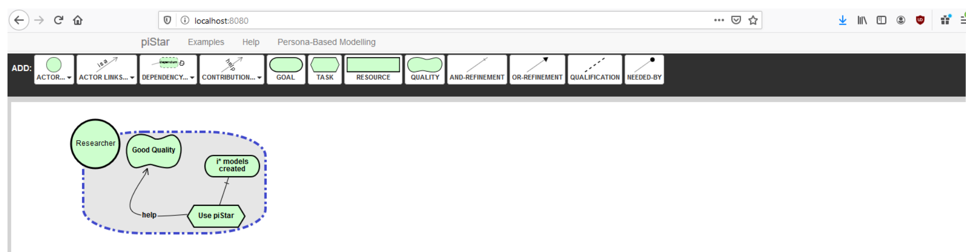

Figura 3.6: Tela inicial do piStar GODA.

Ao clicar, no botão na barra superior escrito *Persona-Based Modelling*, é ativado o ambiente de modelagem baseada em *personas* dentro da ferramenta, capaz de criar fatos, contextos, relacioná-los com o CGM da ferramenta e, por fim, criar *personas* e rodar o algoritmo *Achievability*. Estas etapas serão analisadas em profundidade nas próximas seções.

É importante constatar que novos botões são criados, e, de forma análoga, alguns anteriores são removidos, pois não entram na definição formal do CGM orientado a *personas*.

Essa mudança também foi feita para simplificar a *interface* de usuário, tornando mais simples a visualização dos elementos necessários para a modelagem baseada em *personas*

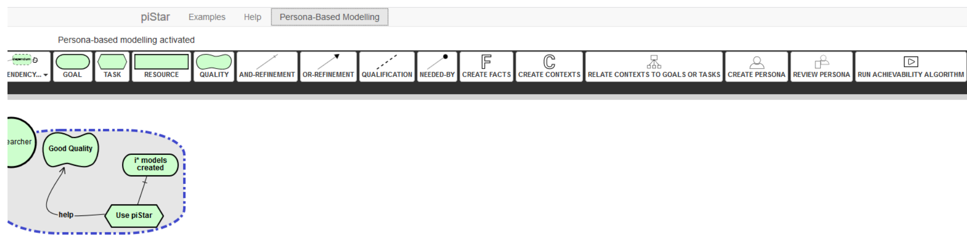

Figura 3.7: piStar GODA com a visão de Personas.

3.4.2 Criação de fatos

Os fatos são criados após o usuário clicar no botão "*Create facts*", visível na Figura 3.7. Ao clicar, o usuário será apresentado a uma modal, onde pode criar, remover e visualizar uma lista completa de fatos criados. Na Figura 3.8 a criação de três fatos foi feita para exemplificar a exibição da lista. A única restrição definida pela interface é que dois fatos não podem possuir o mesmo nome. Esta limitação é feita durante a tentativa de criar o nome de fato repetido.

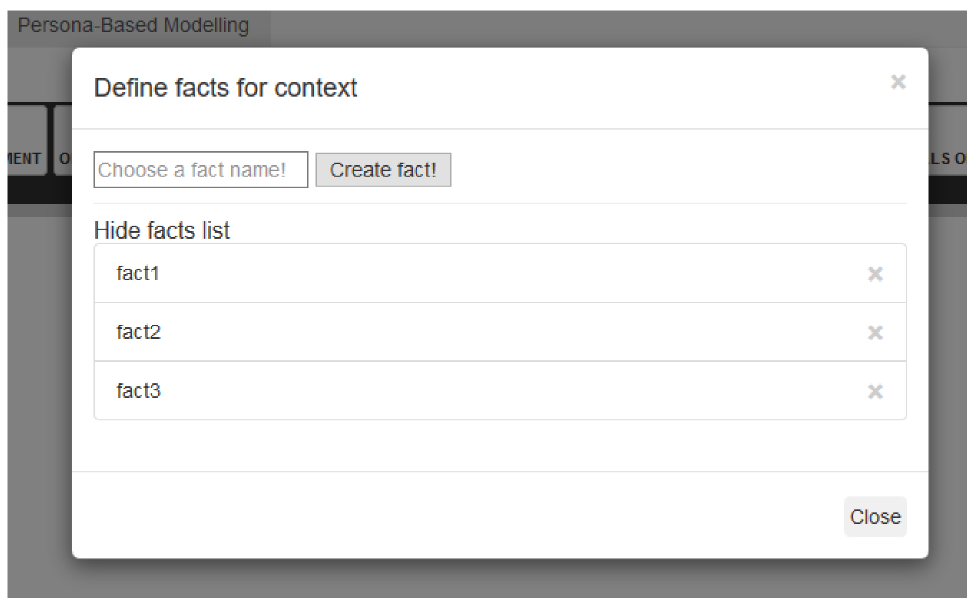

Figura 3.8: Modal da criação de fatos.

3.4.3 Criação de contextos

Da visão da interface criada, o contexto é uma expressão lógica criada a partir da proposição lógica de fatos nomeada pelo usuário. Na Figura 3.9 o estado inicial da modal mostra diversos elementos, são eles:

Figura 3.9: Modal da criação de contextos.

- Criação e edição de contextos: A criação envolve definir o nome para um contexto na lacuna mostrada na Figura 3.9. A edição somente pode ser executada quando há um ou mais nomes de contextos criados, o contexto é selecionado onde o usuário deseja remover ou trocar a proposição lógica atual dele.

- Seleção do tipo de decomposição: O tipo de decomposição define qual será o tipo de proposição lógica que dois ou mais fatos terão entre si.

  Para ilustrar como o tipo de decomposição influencia na proposição lógica, vamos supôr o seguinte cenário: Suponha que dois fatos estejam cadastrados, intitulados de f_1 e f_2, se a opção selecionada for

- AND: A expressão será $f_1 \wedge f_2$
- OR: A expressão será $f_1 \vee f_2$

É importante ressaltar que os caracteres de *AND* e *OR* foram modificados respectivamente para "&"e "|", para simplificação da leitura das expressões e evitar problemas com o *charset* permitido normalmente para páginas web. Há a alternativa de elaborar proposições lógicas mais complexas, o que será discutido em mais profundidade na Seção 3.4.4.

- Lista de *checkboxes* de fatos: A lista que mostra os fatos que podem ser selecionados em uma criação de contexto. Quando selecionados são adicionados à expressão contida na *Decomposition preview.*
- *Decomposition preview*: A *view* responsável por mostrar uma pré-visualização de como a expressão final do contexto ficará. Mais detalhes são explicados na Seção 3.4.4.
- *Selected context*: Mostra o contexto selecionado atualmente, que pode ser alterado na opção de edição de contexto.
- Botão "*Finish and create context*": Finaliza a criação ou edição de contextos, adicionando ou modificando o nome e sua proposição lógica às estruturas de dados e *views* de contextos do projeto.

3.4.4 Expressões complexas e *decomposition preview*

Durante o desenvolvimento da ferramenta, foi constatado que a criação de expressões lógicas mais complexas eram necessárias para garantir que o usuário pudesse inseri-las sem que ele mesmo tivesse que digitá-las, o que podia incorrer em erros de digitação e de *parsing* pela ferramenta. Portanto, foi decidido desenvolver um mecanismo com regras bem estabelecidas e que fosse intuitivo o suficiente para que o usuário visse suas modificações instantaneamente e pudesse alterá-las ao seu interesse.

Essa seção se dedica a explicar os diferentes tipos de decomposições complexas que podem ser realizados dentro da ferramenta, a fim de torná-la mais amigável.

3.4.5 Criação de novas decomposições

Quando o botão "*Create another decomposition*", presente na Figura 3.9, é selecionado, a modal na Figura 3.10 é apresentada ao usuário, perguntando qual será o tipo de decomposição associado à decomposição anterior. Utilizando o exemplo associado a Figura 3.10, temos que *fact1* e *fact2* foram selecionados e já foram removidos da lista de fatos

selecionáveis, isso porque não é permitido ter dois fatos iguais na mesma decomposição a não ser que eles sejam *world predicates* definidos anteriormente.

De acordo com a opção selecionada, a expressão lógica mudará da seguinte forma:

- Se for OR: a expressão será $(fact1 \land fact2) \lor fact3$
- Se for AND: a expressão será $(fact1 \land fact2) \land fact3$

A seleção do *fact3* é obrigatória (definida pela interface). Caso contrário a expressão terminaria com um operador e nenhum operando, o que caracteriza um erro de formação de expressão lógica, e portanto não permitido pela interface.

A criação de uma outra decomposição explicada acima deve ser realizada com no mínimo um fato selecionado e outro ainda não selecionado e, nesta conformidade, é impossível que seja gerada uma expressão malformada. Após a criação da nova decomposição, a decomposição de novos fatos segue a regra antiga: dois ou mais fatos separados por um operador lógico selecionado no seletor do tipo de decomposição na Seção 3.4.3.

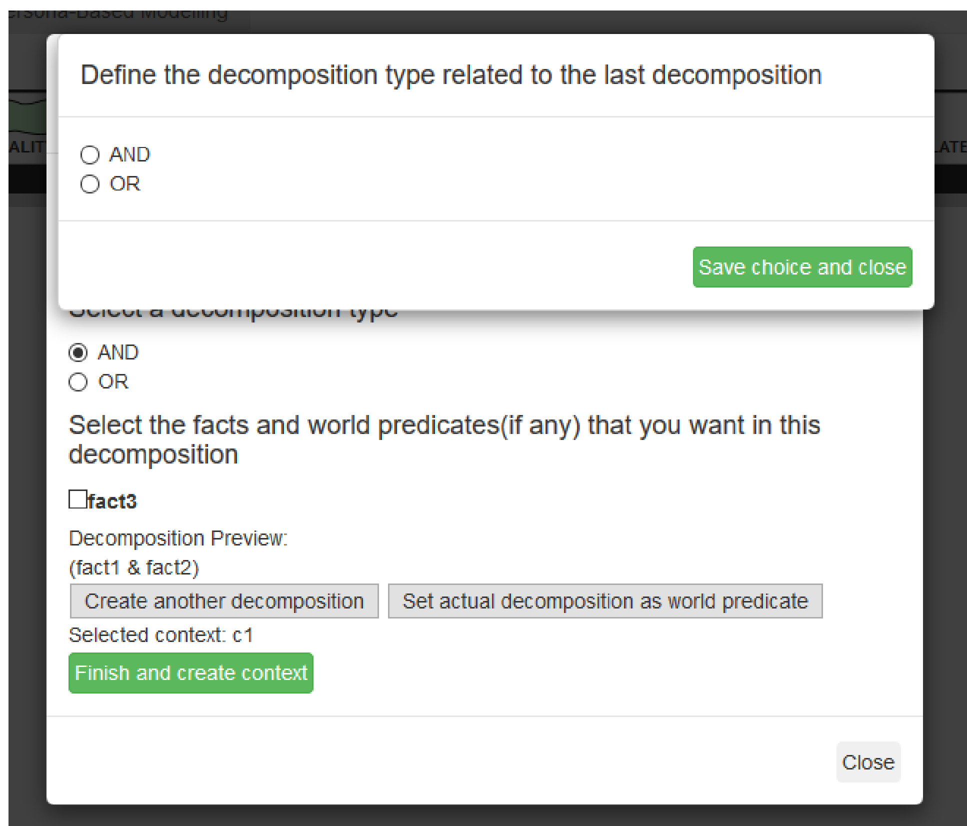

Figura 3.10: Modal da criação de outra decomposição, junto com a modal de contextos.

3.4.6 Criação de world predicates

Como comentado anteriormente, após a seleção de um ou mais fatos, uma proposição lógica é criada. Um *world predicate* é uma proposição lógica empregada normalmente em mais de um contexto. Quando o botão *"Set actual decomposition as world predicate"* é clicado, a proposição lógica é gravada e salva em um *world predicate* de nome definido pelo usuário em uma modal mostrada na Figura 3.11.

Figura 3.11: Modal de criação de novo *world predicate*.

3.4.7 Pré-visualização de decomposições

Decomposition preview é a *view* responsável por mostrar a expressão lógica que será atribuída ao contexto selecionado. Ela funciona de forma reativa aos eventos de adição e remoção de fatos à uma lista interna, isso é, não visível ao usuário, cuja utilidade é definir que fatos foram selecionados ou não.

Sua característica mais distinta é mostrar instantaneamente a mudança quando um fato ou um *world predicate* é selecionado, de modo que as alterações na decomposição fiquem claras para o usuário. Diferentemente dos contextos, *world predicates* não podem

ser removidos, mas podem ser ignorados na hora de construir novos contextos. Após serem criados, as *checkboxes* de world predicates ficam logo após as *checkboxes* da lista de fatos, como mostra a Figura 3.12.

Figura 3.12: Modal de contextos após a criação de *world predicate*.

3.4.8 Associação de contextos com o modelo CGM

Para reconhecimento da ferramenta, é necessário que os elementos do modelo CGM estejam com uma anotação que defina os contextos associados àquele elemento. Tal como no modelo proposto [1], a associação de contextos é feita apenas com *goals* ou *tasks* do modelo CGM dentro do piStar GODA.

A associação é feita através do clique da opção "*Relate contexts to goals or tasks*". Após realizar o clique, o usuário deve clicar no *goal* ou *task* desejado, que abrirá uma modal que mostrará uma lista de *checkboxes* dos contextos criados pelo usuário, permitindo uma associação semelhante com a de fatos e contextos.

Depois de realizar a associação, o elemento selecionado possuirá uma nova propriedade *creationProperty*, que permite a identificação de contextos com a sintaxe definida pelo ANTLR. A sintaxe é definida da seguinte forma:

1. Um elemento com a propriedade de nome *creationProperty* possui um valor válido se, e somente se, possuir a estrutura

   - *assertionCondition expressao*

2. Onde *expressao* é uma expressão definida com a seguinte sintaxe:

   - Para no mínimo dois operandos em *expressao*:
     - Para operadores *AND*, é utilizado o operador "&" entre os dois operandos;
     - Para operadores *OR*, é utilizado o operador "|" entre os dois operandos;
   - Para apenas um operando ou para dois ou mais operandos, desde que os operadores *AND* e *OR* tenham sido utilizados entre eles:
     - Para operadores *NOT*, é utilizado próximo ao operando desejado o operador "!".

A propriedade pode ser editada manualmente pelo usuário para inserção de operadores *NOT*, desde que siga as regras definidas acima, isso pois a UI não possui suporte para criação de expressões com esse operador, mas é capaz de reconhecê-los durante a execução do algoritmo *Achievability* desenvolvido para essa solução.

3.4.9 Criação de *personas*

A criação de *personas* pode ser feita a partir do clique na barra superior em *"Create Persona"* que abre uma modal, disposta na Figura 3.13, onde podemos inserir:

- Nome da *persona* (obrigatório): Define o nome da *persona*
- Descição da *persona* (opcional): Uma descrição que serve apenas para relembrar o usuário lembrar da *persona* em questão
- *Checkboxes* de contextos (obrigatório): *Checkboxes* que definem quais contextos a *persona* possui.

Após o cadastro da *persona*, o usuário pode revisar as informações colocadas ao clicar em *"Review Persona"* e abrir uma barra lateral, que possui uma caixa de seleção onde é possível selecionar uma das *personas* criadas e rever as informações fornecidas, bem como rodar o algoritmo *Achievability*.

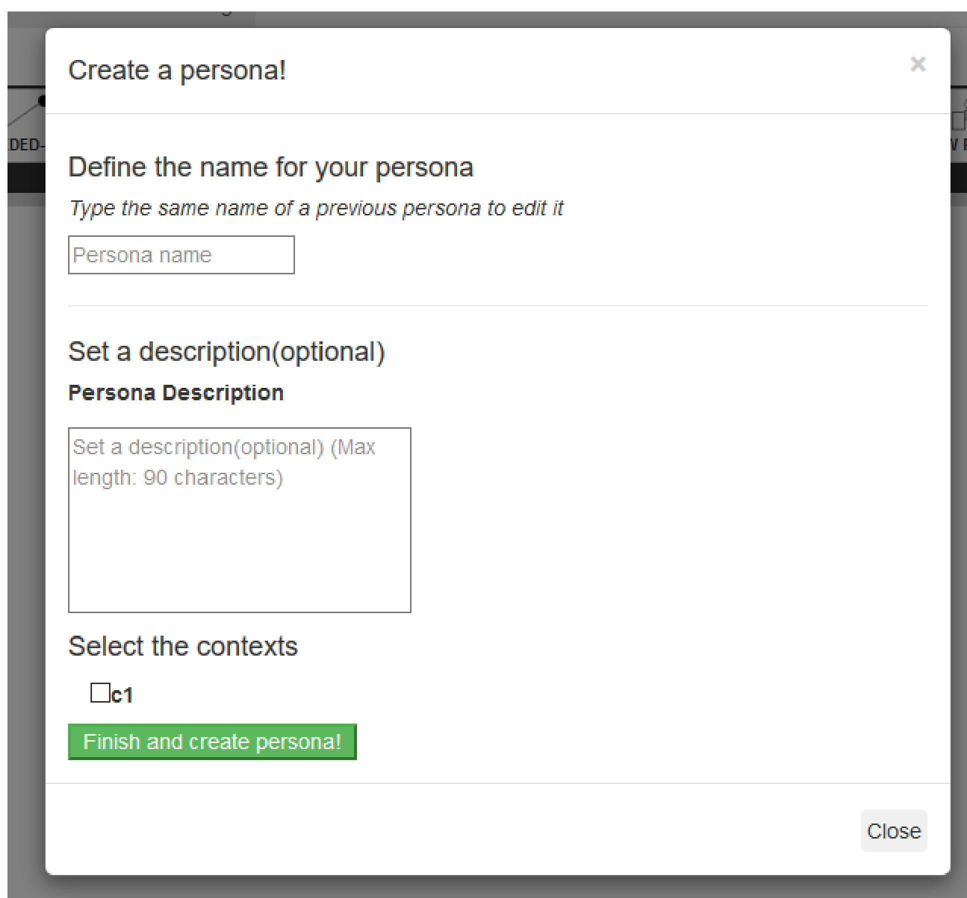

Figura 3.13: Modal de criação de *personas*.

Capítulo 4

Avaliação da Ferramenta

Este capítulo apresentará a reprodução do estudo exploratório feito no Artigo [1], a execução e os resultados obtidos a partir da implementação elaborada como extensão do piStar GODA. A ferramenta estendida está disponível no GitHub em projeto aberto pelo *link* `https://github.com/danilobispo/pistargodaintegration`.

4.1 Estudo exploratório

O estudo exploratório selecionado para avaliação e validação da implementação foi refazer o estudo utilizado em [1] dentro da nova ferramenta proposta, sendo replicado o modelo CGM utilizado, respeitando a hierarquia de tarefas e objetivos do modelo original.

No projeto anterior, foi criada uma notação que funciona com a seguinte sintaxe:

- Para um dado objetivo G_i dentre os n objetivos, é atribuído um número sequencial único i que é adicionado à sua direita, em seguida é adicionada uma descrição separada por dois pontos após o número selecionado.
  - Por exemplo: G_4: *central receives info*.
- De forma semelhante, para uma dada tarefa T_i dentre as n tarefas, um número sequencial único i é adicionado à sua direita e uma pequena descrição também é inserida para diferenciá-lo com mais facilidade.
  - Por exemplo: T_9: *notify central by sms*.

A contagem dos números válidos começa do 1 e vai até o número respectivo de tarefas e objetivos. Nesse trabalho, os objetivos e tarefas também foram enumerados de forma idêntica ao estudo anterior, simplificando a análise dos resultados.

As *personas* utilizadas para a avaliação também foram as mesmas, com a finalidade de verificar se os resultados encontrados são idênticos ao estudo anterior. Um algoritmo

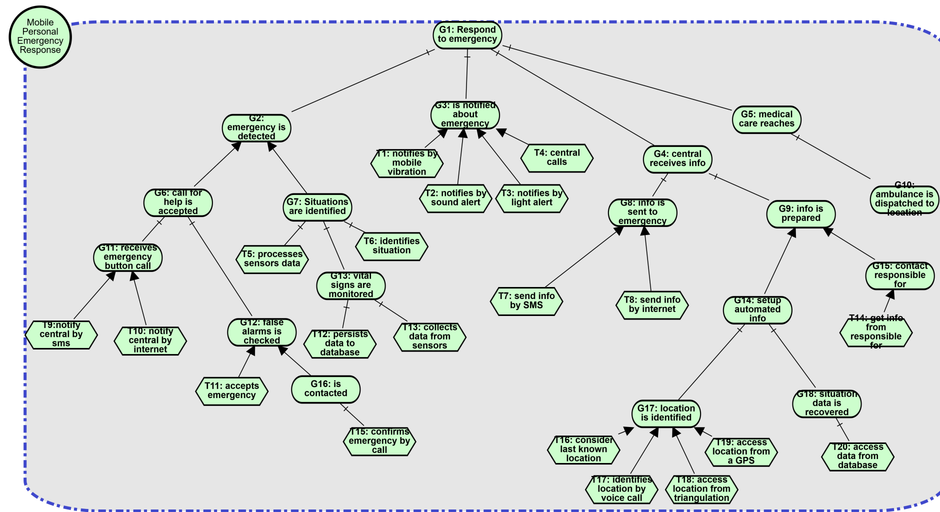

Figura 4.1: Modelo CGM do MPERS no piStar GODA.

Contexts	Mary Collins	Jennifer Smith	Dorothy Williams
C_h(*Health risk*)	Sim	Sim	Sim
C_m(*Mobility issue*)			Sim
C_t(*Technology aversion*)	Sim		
C_{ha}(*Home assistance*)	Sim	Sim	Sim
C_a(*Physical Activity*)		Sim	Sim

Tabela 4.1: Contextos válidos para cada persona

de exemplo foi feito na plataforma com o intuito de avaliar o caso de estudo rapidamente, com o modelo pré-carregado nele, bem como as *personas* e seus respectivos contextos.

No estudo exploratório atual, os fatos necessários para cada contexto foram omitidos pois são desnecessários para a criação de um exemplo funcional a ser avaliado pelo *Achievability*. Isso ocorre pois o único fator considerado durante a execução do algoritmo são os contextos da *persona*, caso o cadastro (nesse caso, feito manualmente) esteja com contextos válidos presentes no CGM, a ferramenta fará a análise correta dos dados.

A Tabela 4.1 apresenta uma pequena revisão dos contextos utilizados e quais deles cada uma das *personas* ativa. As caixas vazias representam que a *persona* não ativa aqueles contextos.

Como a Figura 4.1 não mostra os contextos válidos para cada objetivo ou tarefa do modelo CGM, é possível acompanhar com mais precisão as avaliações utilizando a Figura 2.1 para verificar quais contextos se encaixam no elemento em questão do modelo. Também é possível clicar em cada elemento e verificar se ele possui a propriedade descrita

em 3.4.8 que permite a análise daquele elemento como contexto válido, bem como a proposição exigida para aquele nó do CGM.

As descrições dos contextos foram inseridas dentro do modelo CGM em letras minúsculas (*lowercase*) para simplificar a leitura dos resultados. Adicionalmente, como esperado, nem o nome do objetivo ou sua descrição influenciam no resultado, como os dados foram inseridos manualmente, apenas as siglas que identificam unicamente as tarefas e objetivos foram mantidas em letra maiúscula, para facilitar sua identificação.

Para não haver confusão dos dados, a cada *persona* avaliada, o algoritmo possui confirmações para continuar pois é rodado três vezes para avaliar as três *personas* descritas na Tabela 4.1. Quando clicamos no exemplo, presente na barra superior *"Examples"* e em seguida na opção *"MPERS Achievability Example"*, ele carrega o modelo CGM e começa a rodar instantaneamente o algoritmo para a primeira *persona*. Em seguida, ele envia uma mensagem de confirmação para que o usuário possa confirmar a continuação das análises.

4.1.1 Perguntas do estudo

As perguntas a serem respondidas pelo estudo conduzido são:

1. É possível reproduzir o estudo feito no MPERS [1] e obter resultados idênticos?
2. A ferramenta é capaz de reproduzir outros modelos *CGM* e realizar a avaliação de *Achievability* deles?

Essas questões serão avaliadas na Seção 4.2, juntamente com outras observações relevantes durante a análise individual das três *personas*. As perguntas foram elaboradas com base nas necessidades descritas na Seção 1.2, razões que motivaram a extensão e implementação da modelagem baseada em *personas* no piStar GODA.

4.2 Resultados

4.2.1 Caso 1: Mary Collins

Mary Collins, assim como no estudo de caso anterior [1] é a única *persona* que não é capaz de atingir o objetivo raiz, por possuir o contexto C_t como válido, vários objetivos possuem contextos cujas proposições dependem desse contexto ser inválido para que sejam alcançados. Alguns exemplos são:

- T_{11}*: accepts emergency*, onde a proposição lógica é: $C_{ha} \wedge \neg C_t$
- T_7*: send info by SMS*, onde a proposição lógica é: $\neg C_t$

- G_{16}: *is contacted*, onde a proposição lógica é: $C_{ha} \land \neg C_t$
- T_8: *send info by internet*, onde a proposição lógica é: $\neg C_t$

No estudo anterior, a falha foi no objetivo G_4, onde os filhos T_7 e T_8 possuíam contextos no qual Mary Collins não se adequava, culminando na falha. Nesse algoritmo, como a iteração começa a descer pelos nós da esquerda da árvore, isso é, ele faz uma busca em profundidade a partir do nó G_2, depois do G_3 e por fim chega no nó G_4, portanto, ele avalia primeiramente com o objetivo G_{16}(filho de G_2), onde há uma falha, porém, como o objetivo pai dele é uma decomposição *OR*, é possível que um outro nó(caso exista) seja avaliado e a expressão dele seja avaliada como verdadeira, não foi o caso aqui, pois o único nó irmão na árvore possuía a mesma proposição lógica, portanto os dois nós falharam, constituindo uma falha que inviabiliza o uso do sistema por essa *persona*.

O *output* abaixo apresenta um exemplo das saídas do programa Java que evidenciam as falhas ao atender objetivos contextuais necessários:

```
1  Persona name: Mary Collins
2  Persona description: mary
3  Persona contexts: [ch, cha, ct]
4  br.unb.cic.goda.model.GoalImpl@9ece12
5  Goal G1: Respond to emergency is being evaluated
6  Goal G2: emergency is detected is being evaluated
7  Goal G6: call for help is accepted is being evaluated
8  Goal G11: receives emergency button call is being evaluated
9  Task T9:notify central by sms is being evaluated
10 Evaluation sequence for :!ct | cha
11 (! ct)=false
12 ((! ct)=false | cha)=true
13 Task T10: notify central by internet is being evaluated
14 Task T9:notify central by sms is being evaluated
15 Evaluation sequence for :!ct | cha
16 (! ct)=false
17 ((! ct)=false | cha)=true
18 Task T10: notify central by internet is being evaluated
19 Goal G12: false alarms is checked is being evaluated
20 Goal G16: is contacted is being evaluated
21 Evaluation sequence for :cha & !ct
22 (! ct)=false
23 (cha & (! ct)=false)=false
24 Error at G16: is contacted.Logical proposition is false
25 Task T15: confirms emergency by call is being evaluated
26 Task T11: accepts emergency is being evaluated
27 Evaluation sequence for :cha & !ct
28 (! ct)=false
```

```
29 (cha & (! ct)=false)=false
30 Error at T11: accepts emergency.Logical proposition is false
31 Task T11: accepts emergency is being evaluated
32 Evaluation sequence for :cha & !ct
33 (! ct)=false
34 (cha & (! ct)=false)=false
35 Error at T11: accepts emergency.Logical proposition is false
36 Error at T11: accepts emergency.Conditions not met for persona
37 The current persona does not meet the CGM context conditions
```

Listing 4.1: *output* do resultado em texto para a *persona* Mary Collins

4.2.2 Caso 2: Jennifer Smith

Na segunda *persona* avaliada, Jennifer Smith, são encontradas falhas em dois dos nós observados, são eles:

- T_1*: notifies by mobile vibration*, onde a proposição lógica é C_t
- T_4*: central calls*, onde a proposição lógica é $C_t \wedge C_{ha}$

Entretanto, essas duas tarefas participam de decomposições OR, e como tais, não configuram falha completa do algoritmo em relação a *persona* atual, e seus nós vizinhos são bem sucedidos, logo o algoritmo então continua a fazer os outros elementos até chegar ao fim das avaliações e não falhar definitivamente em nenhum objetivo ou tarefa obrigatório para a alcançabilidade do objetivo raiz.

4.2.3 Caso 3: Dorothy Williams

Nesse último caso, a *persona* também teve êxito ao alcançar o objetivo raiz, mas falhou em alguns dos nós do modelo CGM MPERS, são eles:

- *Task T16: consider last known location*, onde a proposição era $\neg C_m$
- T_1*: notifies by mobile vibration*, onde a proposição lógica é C_t
- T_4*: central calls*, onde a proposição lógica é $C_t \wedge C_{ha}$

Porém, como no último caso, os nós eram decomposições *OR* e não comprometiam o alcance do objetivo G_1: *Respond to emergency* pois não eram as únicas condições necessárias, permitindo que o caso 3 também seja um sucesso e o modelo CGM atenda à *persona*.

4.2.4 Questões levantadas

Além de verificar o resultado individual das *personas*, o estudo exploratório também foi conduzido por duas perguntas, presentes na Subseção 4.1.1. As perguntas serão respondidas com base nos resultados encontrados durante o experimento do MPERS, mas também para outro modelo utilizado anteriormente na plataforma para testes de integração do piStar GODA.

Para o item 1, a resposta é verdadeira, porém com algumas ressalvas: Na elaboração da ferramenta visual, um requisito que não foi enxergado durante as etapas de elaboração e implementação foi o da implementação da negação de um contexto, ou seja, uma negação lógica conhecida como $\neg$, porém, o *evaluator* descrito na Seção 3.3.2 é capaz de reconhecer proposições lógicas com o operador de negação, seguindo a sintaxe:

$$\text{Para um contexto } C_1\text{, a negação } \neg C_1 \text{ pode ser escrita como: } !C_1$$

A edição da propriedade que configura o contexto pode ser feita na plataforma antes de rodar o algoritmo, tornando possível adicionar a negação de contexto em um exemplo criado totalmente na plataforma, contudo, sem a assistência das novas funcionalidades criadas por esse projeto.

Já para o item 2, a ferramenta foi capaz de apresentar bons resultados para *personas* criadas para testes em modelos CGM utilizados anteriormente [7] para validação de outros aspectos relativos ao modelo. Dentro da plataforma, um *link* para um exemplo de modelo CGM com modificações nas propriedades que definem contextos foi adicionado juntamente ao exemplo de *Achievability* do estudo exploratório descrito na Seção 4.1 no último parágrafo, denominado *"Achievability Example"*.

O exemplo lida com a avaliação de uma *persona* genérica com os contextos c_1 a c_4, os contextos não possuem significância, tampouco a *persona*. Os modelos do exemplo são utilizados apenas como ilustração para avaliação do algoritmo dentro de um CGM. Após a execução, uma mensagem de erro será dada logo no primeiro nó, que faz a seguinte assertiva:

$$\textit{assertion condition } c_1 \wedge \neg c_2$$

Uma proposição lógica que irá falhar para essa *persona*, pois c_2 é um contexto que ela possui, resultando em falso no lado direito da proposição lógica, a mensagem de erro pode ser vista na Figura 4.2. Não é possível alterar os dados do CGM e tentar rodar o algoritmo a partir do clique do botão na barra superior, já que não existe *persona* criada pela plataforma e um dos argumentos do serviço seria enviado como vazio, culminando em um erro em seu funcionamento. É possível, no entanto, criar uma ou mais *personas* com os mesmos nomes de contextos descritos no CGM e avaliar o funcionamento para diferentes combinações de contextos válidos.

É possível verificar, portanto, que é possível criar outros modelos com contextos e *personas* diferentes com o apoio da ferramenta. Também é capaz de avaliar a alcançabilidade dos objetivos por diferentes grupos de usuários de forma mais dinâmica, flexível e ágil, validando o modelo adotado.

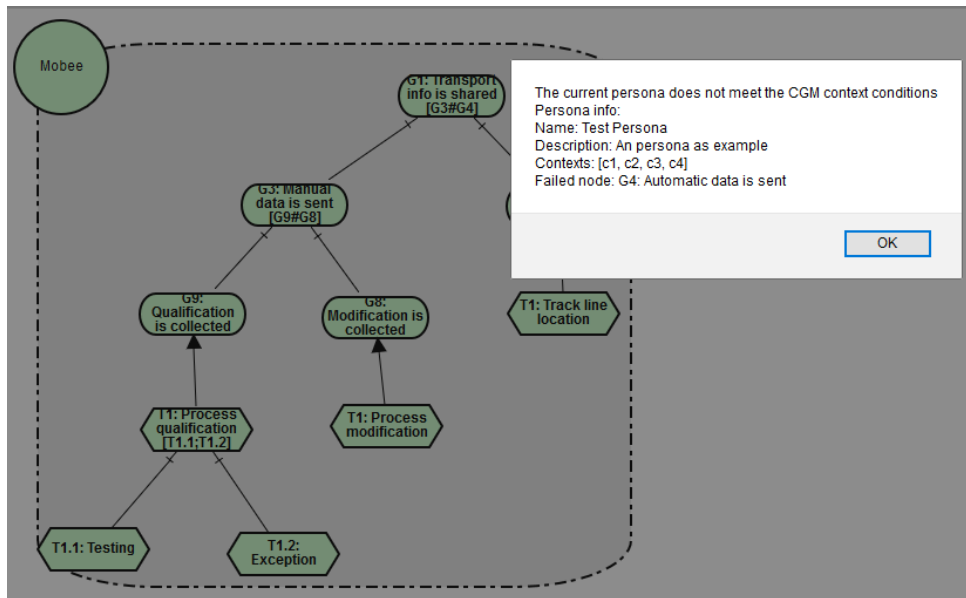

Figura 4.2: Exemplo de *Achievability* com outro CGM com a mensagem de erro.

Capítulo 5

Conclusão e Trabalhos futuros

Essa seção conclui o trabalho, mostrando as interpretações das observações inferidas dos resultados. A conclusão também aborda os possíveis trabalhos futuros que derivam desse projeto. Também serão apresentadas as dificuldades encontradas durante a implementação das novas funcionalidades e as lições que foram aprendidas, potencialmente aplicáveis em futuros projetos.

5.1 Dificuldades encontradas e lições aprendidas

Uma das grandes dificuldades encontradas durante todo o projeto foi a pouca documentação do *framework* do Backbone.js [14], as funções internas da ferramenta discutem muito fracamente em sua documentação os argumentos de função e os retornos(quando existentes) de funções, adicionalmente, alguns *softbugs* de elementos gerados com o Backbone.js durante a execução da ferramenta foram detectados e não possuíam qualquer mensagem de erro, gerando um esforço não-previsto durante as etapas de implementação e atrasando consideravelmente a entrega de algumas funcionalidades.

Uma das lições aprendidas diante das várias dificuldades encontradas dentro do Backbone.js é a de utilizar ferramentas com suporte maior e atualizações mais estáveis em próximas versões do projeto, potencialmente substituindo as utilizadas nesse trabalho.

Outra grande dificuldade foi compreender a difícil arquitetura interna de *parsing* da árvore de objetivos CGM, que foi importada para a validação dos casos de teste presentes em [7]. O código não possui documentação e possui muitos *loops* em classes internas da biblioteca do GODA assim como do Java, o que dificulta muito o *debugging* e o compreendimento completo do que acontece nas etapas do código. Felizmente, foram extraídas as funcionalidades necessárias para aplicar o algoritmo Achievability na seção Java do código com o apoio da criação de árvores hierárquicas dos modelos CGM.

5.1.1 Conclusão e trabalhos futuros

Conclui-se com o trabalho que a análise da ferramenta apresenta resultados semelhantes ao estudo anterior do Artigo [1], comprovando que a reprodução do algoritmo *Achievability* em um sistema *web* foi bem sucedida. Não obstante, a própria contribuição da integração do processo de modelagem de *personas* na ferramenta, que permite que modelos sejam refinados a partir da verificação de estudos de caso. Contudo, o projeto atual não construiu um grupo de testes para validação. É necessário elaborar um grupo de testes que avaliem a robustez da ferramenta e a validade de seus métodos, tanto de interface de usuário quanto dos serviços elaborados.

Outras melhorias na própria solução atual também são possíveis, é possível alterar o código do *Evaluator* 3.3.2 para que ele detecte quais fatos contextuais ativam um contexto e a criação de *personas* possa ser realizada apenas com base nos fatos de uma *persona*, deixando a inferência de contextos para a plataforma. Essa melhoria ajudaria os usuários que não precisariam inferir manualmente se uma *persona* ativa ou não um determinado contexto e poderiam ser auxiliadas pela ferramenta durante a verificação de estudos de caso.

A análise orientada a *personas* também abre muitas opções para a pesquisa, como por exemplo, aprimoramento de técnicas para as elucidações de fatos contextuais relevantes para sistemas, processamento de linguagens naturais para elucidação de fatos contextuais e análises variadas de melhorias para modelos de objetivos com base em resultados do algoritmo *Achievability*.

Referências

[1] Nunes Rodrigues, G., Joel Tavares C. Watanabe N. Alves C. Ali R.: *A persona-based modelling for contextual requirements*. Requirements Engineering: Foundation for Software Quality, 10753:352–368, 2018. v, vi, 2, 3, 4, 7, 8, 9, 10, 14, 17, 33, 36, 38, 44

[2] Yu, E. e J. Mylopoulos: *Why goal-oriented requirements engineering*. Requirements Engineering: Foundation for Software Quality, páginas 15–22, 1998. 1, 5

[3] Ali, R., Dalpiaz F. e P. Giorgini: *A goal-based framework for contextual requirements modeling and analysis*. P. Requirements Eng, 15:439–458, 2010. 1, 2, 5, 6, 8, 9

[4] Yu., E.S.K.: *Modelling strategic relationships for process reengineering*. Ph.D. Thesis - University of Toronto, 1995. 2

[5] P. Bresciani, A. Perini, P. Giorgini F. Giunchiglia e J. Mylopoulos: *Tropos: An agent-oriented software development methodology*. Autonomous Agents and Multi-Agent Systems, 2004. 2

[6] Anne Dardenne, Axel van Lamsweerde e Stephen Fickas: *Goal-directed requirements acquisition*. Sci. Comput. Program, 1993. 2

[7] Santos Bergmann, L.: *pistar-goda: Integração entre os projetos pistar e goda*, 2018. 2, 3, 10, 11, 20, 23, 41, 43

[8] Huang, J.: *Meet elaine: A persona-driven approach to exploring architecturally significant requirements*. IEEE Magazine, 30:18–21, 2013. 2, 6

[9] Faily, S. e I. Flchais: *Eliciting and visualising trust expectations using persona trust characteristics and goal models*. Proc. of the 6th International Workshop on Social Software Engineering, 2014. 5, 13

[10] F. Mendonça, Danilo: *Dependability verification for contextual/runtime goal modelling*, 2015. 10

[11] *Rappid: Powerful visual tools at your fingertips*. `https://www.jointjs.com/`, acesso em 2019-11-23. 10

[12] *O que é a tecnologia java e porque preciso dela?* `https://www.java.com/pt_BR/download/faq/whatis_java.xml`, acesso em 2019-11-18. 11

[13] Contributors, Mozilla: *Javascript - mozilla developer network web docs.* https://developer.mozilla.org/pt-BR/docs/Web/JavaScript, acesso em 2019-11-10. 11

[14] *Backbone.js - página principal.* https://backbonejs.org/, acesso em 2019-11-10. 11, 43

[15] *Collection - java 8 docs.* https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html, acesso em 2019-11-11. 12

[16] *Underscore.js - página principal.* https://underscorejs.org/, acesso em 2019-11-10. 12

[17] Afonso, Alexandre: *O que é spring boot?* https://blog.algaworks.com/spring-boot/, acesso em 2017-02-02. 12

[18] Fielding, Roy Thomas: *Architectural styles and the design of network-based software architectures.* https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm, acesso em 2019-12-01. 12

[19] *Astah uml - main page.* http://astah.net/editions/uml-new, acesso em 2019-12-04. 13

[20] *Javaluator- official site.* http://javaluator.sourceforge.net/en/home/, acesso em 2019-11-18. 25