PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

Gerador de história de jogo baseado em planejamento com uma interface entre Prolog e C#

Rafael Rubim Cabral

PROJETO FINAL II

ENG1133 – PROJ DE GRAD EM ENG COMPUTAÇÃO II

CENTRO TÉCNICO CIENTÍFICO – CTC

DEPARTAMENTO DE INFORMÁTICA

Curso de Graduação em Engenharia da Computação

Rio de Janeiro, dezembro de 2019

Rafael Rubim Cabral

Gerador de história de jogo baseado em planejamento com uma interface entre Prolog e C#

Projeto final apresentado ao Curso Engenharia da Computação como requisito parcial para a obtenção do título de Engenheiro de Computação.

Orientador: Bruno Feijó

Rio de Janeiro

Dezembro de 2019

Agradecimentos

A minha mãe Denise, por me inscrever no vestibular da PUC que eu não ligava em fazer.

A meus pais Denise e Márcio, por me incentivarem a fazer minhas próprias escolhas e me apoiarem até o final. E por serem os melhores pais do mundo.

A meus irmãos Ana Luísa e Pedro, por serem dois gordinhos muito engraçados que iluminam minha vida.

A minha avó Denair e à Maria Luiza, por me aguentarem nas madrugadas e por fazerem de tudo por mim.

A minha irmã e melhor amiga e em todo mundo Ana Alice, por não me deixar esquecer nunca como sou amado e por me ensinar a cuidar dos meus tesouros.

A meus irmãos Eduardo Matias, Gabriel Sarruf, Rodrigo Donato, Thor Rodrigues, Yuri Gonçalves, por que já tem 7 anos que eu tenho mais uma família.

Aos meus amigos Bernardo Ruga e Victor Pinto, pelo apoio e por me lembrarem que ninguém sofre sozinho.

A minha amiga Karen, por revisar meus erros e ajudar com minhas dúvidas.

A meus amigos Bianca Villar, Gabriela Gutierrez e Pedro Meireles, por me apoiarem e tranquilizarem.

A meus amigos Yuri, Rodrigo, Bernardo, Pedro, Alexandre e Julio, por salvarem meus dias com memes.

Ao resto de meus amigos e familiares próximos, que me fazem ser a pessoa mais feliz do mundo.

A meu orientador Bruno, por ser paciente comigo.

A meus professores, por que além de exercerem a profissão mais nobre, foram muito amáveis comigo.

A meu cachorro Merlin, por ser um fofo e por madrugar comigo.

E a todos que acreditaram em mim, quando nem eu mais estava.

Resumo

Rubim Cabral, Rafael. Feijó, Bruno. Gerador de história de jogo baseado em planejamento com uma interface entre Prolog e C#. Rio de Janeiro, 2019. 63p. Relatório de Projeto Final - Departamento de Informática. Pontifícia Universidade Católica do Rio de Janeiro.

Este trabalho implementa uma biblioteca C# de apoio à geração de histórias interativas em jogos eletrônicos, com fins de entretenimento. A biblioteca é integrada com uma implementação em Prolog. Realizou-se uma análise sobre os principais componentes de boas narrativas e a variação do *storytelling* baseado na personalidade e ações do jogador. Determinou-se uma ontologia independente de gênero de história e criaram-se ferramentas para um usuário definir o domínio de um gênero particular, os elementos específicos de uma história e uma trama. Utilizam-se essas definições para auxiliar a formação da história do jogo em tempo de execução. A biblioteca é reativa a inputs do jogador ou usuário e usa um algoritmo de planejamento interno - com base em uma função heurística configurável - para direcionar o progresso da história de forma a satisfazer a trama. Obtiveram-se resultados satisfatórios, porém não escaláveis.

Palavras-chave

Storytelling. História. Jogos. Planejador. Geração.

Abstract

Rubim Cabral, Rafael. Feijó, Bruno. Planning-based game story generator with an interface between Prolog and C#. Rio de Janeiro, 2019. 63p. Capstone Project Report - Department of Informatics. Pontifical Catholic University of Rio de Janeiro.

This work implements a C# support library for interactive story generation in electronic games, for entertainment purposes. The library is integrated with an implementation in Prolog. An analysis was performed around the main components of good narratives and the variation of storytelling based on the player's personality and actions. A story genre-independent ontology was determined and tools were created for a user to define the domain of a particular genre, the elements specific to a story and a plot. These definitions are used to assist the game's story formation in runtime. The library is reactive to player or user's inputs and uses an internal planning algorithm - based on a configurable heuristic function - to direct the story's progress in a way to satisfy the plot. Satisfactory results were obtained, though they are not scalable.

Keywords

Storytelling. Story. Games. Planner. Generation.

Sumário

1. Introdução

Existem muitos tipos de jogos cuja qualidade principal é a história experienciada pelo jogador. Dentre esses, a interatividade do enredo é uma funcionalidade muito explorada, pois grande parte dos jogadores explora a sensação de ter controle sobre o destino dos personagens.

Esses jogos oferecem várias escolhas para o jogador, possuindo uma multiplicidade de cenários que resultam das ações dele. Esse projeto explora uma alternativa para a implementação dessa funcionalidade: a geração automática da história em tempo de execução do jogo.

No presente projeto, foi desenvolvida uma biblioteca de apoio a essa geração, chamada de Gerador de Histórias, que se baseia em um algoritmo de planejamento. Para utilizar esta biblioteca, o desenvolvedor do jogo especifica os elementos deste, como os personagens, itens e lugares exploráveis, que são mapeados na biblioteca. Dados uma lista de objetivos a serem cumpridos (chamados de “trama” da história) e um conjunto de possíveis eventos, o Gerador de Histórias planeja os eventos em escala “macro” importantes para o avanço do jogo. Por exemplo, ele determina que o personagem X deve se mover para o lugar Y, ou determina que ele deve pegar o item Z. Esses eventos são obtidos em tempo de execução, para que sejam influenciados pelas ações e interação com o jogador. Por isso, o jogo deve garantir que seus elementos estão muito bem sincronizados com o modelo da biblioteca, transformando, por exemplo, um evento “personagem se move” em um comando para um NPC de fato se mover.

Esse projeto não implementa nenhum jogo, mas a biblioteca do Gerador de Histórias em C#. Essa linguagem foi escolhida porque é muito utilizada para o desenvolvimento de jogos, devido ao uso de engines e frameworks como Unity[1] e Monogame[2]. A interface em C# utiliza uma implementação em linguagem Prolog, devido a seu caráter descritivo.

A biblioteca recebe os elementos do jogo, tais como:

```
personagem(vilão).
item(tesouro).
lugar(caverna_do_tesouro).
```

[1] Disponível em: <https://unity.com/>. Acesso em 4 dez. 2019.
[2] Disponível em <http://www.monogame.net/>. Acesso em 4 dez. 2019.

Em múltiplas iterações em tempo de execução, a biblioteca retorna “comandos” para o jogo, que constituem os eventos que ocorreram na história, por exemplo:

```
vilão move_para caverna_do_tesouro
vilão pega tesouro
vilão escapa || vilão fica_e_luta
```

Na prática, os eventos não são retornados em forma textual, mas como uma lista de objetos em C# que representam eventos. Assim, é mais fácil sincronizá-los com os elementos da biblioteca e podem ser realizadas buscas (exemplo: LINQ[3]) com dados textuais.

2. Motivação

A narrativa é um recurso de linguagem essencial para a humanidade. Ela sempre usou histórias como meio de transmitir informações, desde que foi capaz de se comunicar. As pessoas constantemente trocam histórias por meio de conversas, gestos, imagens, textos e vídeos, com objetivo de relatar experiências, entreter e comover, dentre outros. Hoje, a inserção de uma abundância de livros, filmes, séries, documentários e *podcasts* na vida pessoal demonstra o apreço do indivíduo pelo uso desse recurso comunicativo. Existem guerras que foram retratadas em pinturas; etnias que se identificam com estilos de dança; cantores que narram suas próprias experiências de vida nas músicas. Todas essas expressões artísticas são enriquecidas pelo ato de contar histórias.

Mais especificamente, a narrativa pode ter mais de um objetivo útil para o locutor. Por conta disso, podemos associar a narrativa à palavra *storytelling*. CHAITIN (2019) explica que *storytelling* não serve apenas para contar acontecimentos passados, mas para apresentá-los de maneira subjetiva e estabelecer ligações entre eventos e pessoas.

O narrador pode alcançar seus objetivos mediante a empatia ou o interesse que a história provoca. Assim, há a necessidade de contar uma “história relevante". Histórias boas são mecanismo para imersão do interlocutor - fica fácil de decorar e influenciar as emoções conscientemente ou não, pela natureza apreciativa de histórias.

O *storytelling* pode ter inúmeros objetivos, tais como: ensinar; promover uma ideia, produto ou marca; influenciar e comover as pessoas; auxiliar a memorização de uma informação; comunicar ou prover suporte e treino psicológico. A ampla utilidade desse recurso é a motivação deste trabalho, com o objetivo de criar *storytelling* atraente ao interlocutor.

---

[3] LINQ significa *Language Integrated Query*, uma API e sintaxe de consultas a coleções em C#.

Em primeiro lugar, consideram-se os termos “história” e “estória” no domínio de *storytelling.* Ambos os casos tratam de narrativas, que, em termos textuais, podem possuir diferentes objetivos. As histórias têm o objetivo de informar e ensinar, enquanto as estórias, de entreter. Para provocar a imersão alvejada pelo *storytelling*, deve-se causar interesse no interlocutor. Por exemplo, ver o nome de um conhecido em um artigo de jornal motivará o leitor a inspecionar o texto. De maneira geral, porém, as estórias são mais agradáveis ao público e desejáveis de se ler. Elas mantêm o interesse do interlocutor até o final. Com essa premissa, as vantagens citadas anteriormente podem ser usadas como objetivos secundários de estórias. O entretenimento é, portanto, a ferramenta trabalhada neste projeto para imergir o interlocutor do *storytelling*. Em diante, esse será o foco do trabalho, portanto a palavra “história” será usada como substituto mais atual do termo arcaico “estória”, com foco em suas características de entretenimento.

Ainda pensando na experiência de imersão no *storytelling*, identificam-se vários meios de comunicação que evoluíram com o tempo e tornaram a experiência das histórias mais enriquecida para o público-alvo. Livros ilustrados, radionovelas, telenovelas - eles acrescentam um nível de experiência ao interlocutor (imagens, sons, vídeo), o que aumenta sua imersão. Atualmente, além da realidade virtual ainda emergente, a interação com histórias é um nível de experiência possibilitado pela rápida comunicação. Ela já é explorada pela indústria de jogos e também emerge com o conceito de filmes interativos, exemplificado por “Bandersnatch”, lançado pela Netflix em dezembro de 2018. Este projeto será desenvolvido sobre o *storytelling* interativo em jogos digitais, o que possibilita construções mais complexas do que o filme exemplificado.

Os videogames têm um apelo de entretenimento e, com base nas hipóteses anteriores, pretendem construir uma narrativa atraente aos usuários que buscam uma história envolvente. Contudo, considera-se o seguinte problema remanescente: mesmo as histórias, cujo objetivo é entreter, não podem facilmente agradar a todos. As pessoas são naturalmente diferentes e, portanto, são entretidas segundo essa mesma natureza. Elas reagem distintamente às situações que lhes são impostas e têm diversas perspectivas dos acontecimentos das histórias. Isso é ainda mais evidente em termos de emoção e impulso dos personagens e situações de discussão de moralidade e outros assuntos em que os indivíduos divergem. Eles são incomodados ou ficam empolgados diante de uma variedade de acontecimentos.

A proposta deste projeto é criar um sistema que apoie a narração de histórias que aceite um modelo de diferentes tipos de personalidades. Um dos objetivos é adaptar a história ao

jogador para despertar seu interesse, sob um domínio modelado em *storytelling*. Outro objetivo é limitar a liberdade de variação da história aos objetivos do autor (sejam estes promover sua história/entreter, gerar interesse por um assunto, fazer publicidade). Cumprir esses objetivos aparentemente conflitantes simultaneamente é um dos desafios da proposta.

3. Situação Atual

Atualmente, *storytelling* em jogos e outras formas de entretenimento que envolvem a interação com o jogador/interlocutor consideram a personalidade na evolução da história e da experiência. Trabalhos recentes têm apresentado propostas que relacionam perfis e tipos básicos de personalidade. Também há pesquisas na área de tipos de jogadores assim como diversos modelos de *storytelling* e suas taxonomias.

Existe um modelo amplamente aceito na área de psicologia de tipos básicos de personalidade. Conhecidos como “BigFive”, os cinco tipos são chamados de: “Neuroticismo”, “Extroversão”, “Abertura”, “Amabilidade” e “Conscienciosidade”. Cada um desses tipos é dividido em seis subtipos relacionados (PSI, 2010). Esse modelo determina esses tipos em uma pessoa e é utilizado em testes de personalidade, como o referenciado acima.

No paper de GERLACH (2018), é estudado um tema considerado mais controverso: a existência de perfis de personalidade. Baseado no BigFive, buscou-se *clusters* (aglomerações) de pessoas com resultados parecidos de um teste que mede esses cinco tipos de personalidade e obteve-se um subconjunto de quatro perfis relativamente bem definidos. Da mesma forma, foram descritas as limitações de seus métodos.

O paper de BARTLE (1996), por sua vez, descreve o que se chama de taxonomia de Bartle, que categoriza jogadores de acordo com sua forma de agir dentro do jogo. Relacionando a taxonomia com modelos de *storytelling*, tais como os descritos por ROE (2016) no site POND5 BLOG, pode-se focar nos níveis de *storytelling* adequados aos interesses de jogador, como o foco no ambiente/mundo da narrativa em casos de jogadores exploradores e nos diálogos/enredo para os socializadores.

Ainda é possível modelar o *storytelling* buscando os casos específicos do trabalho. É possível dividir uma história em três níveis: universo, enredo e perspectiva. Assim, pode-se tentar fixar o enredo ou universo de uma história para cumprir os objetivos narrativos e apresentá-la sob diferentes perspectivas adaptadas para o jogador. Ao mesmo tempo, pode-se fixar uma perspectiva, para firmar um certo arsenal emocional de um personagem quando o

objetivo principal do autor é fazer um jogo de suporte psicoterapêutico. Dessa maneira, o enredo pode se adaptar aos interesses do jogador, de forma gerada computacionalmente ou como recomendação da linha de história correta para a terapia. Podemos ver como exemplos o jogo "*God of war"* (2005) e a série de livros "*Percy Jackson e os olimpianos*" (2010): dois tipos de história com o universo fixo - ambos se ambientam em universos da mitologia grega - que são meios de entretenimento que provocam interesse pela mitologia grega em seus fãs. Ainda que possuam enormes imprecisões históricas, fazem o público alvo ficar muito mais bem disposto a ler fontes seguras sobre a mitologia grega.

Um trabalho já realizado sobre as características do BigFive em *storytelling* interativo é o trabalho de LIMA et al. (2018). Nele, não só os tipos de personalidade são utilizados, como também o que se chamou de comportamento do jogador, que pode variar diante de situações e está mais associado a emoções momentâneas do que seu traço de personalidade. Os algoritmos de planejamento e aprendizado de máquina sobre o modelo permitem a criação de histórias interativas consistentes.

4. Proposta e Objetivos do Trabalho

Neste projeto, pretende-se desenvolver um sistema que dê suporte a *storytelling* em jogos que aceite a personalidade dos jogadores como input. A proposta possui a pretensão de ser independente de gêneros de história ou de jogo.

Pretende-se modelar e especificar os requisitos de forma a resolver um caso de implementação interativa e variável de storytelling, dados limites ou uma coleção de possibilidades impostos pelo autor de história, não necessariamente discretos ou determinísticos. Essa modelagem deverá considerar que as possibilidades de variação de história devem ser resultados de uma função do tipo do jogador. Ou seja, a sua personalidade é um *input* e a saída é uma história que lhe é apresentada.

As preferências do jogador irão se traduzir em eventos na história, com a narrativa desenvolvida com um algoritmo de planejamento - como o utilizado por LIMA et al. (2018).

O ambiente de desenvolvimento proposto é a linguagem C# no runtime de .NET Core com a API de SWI Prolog para o suporte ao algoritmo de planejamento. Essas escolhas são justificadas na seção 8.

Para a modelagem do *storytelling*, algumas questões são consideradas em seus limites subjetivos e não determinísticos:

Quando se adiciona interação, a história continua sendo uma história? Ela perde sua essência? Até que ponto se pode dar liberdade ao jogador sem desviar de seus objetivos ou personalidade/perspectiva do personagem? Como se delimita a diferença entre enredo e perspectiva? Alterá-los não deterministicamente atrapalha o entretenimento da história?

5. Análise de Requisitos

O *storytelling* flexível e interativo possui diversas abordagens para se adaptar à personalidade do jogador. Por exemplo, pode-se imaginar uma história que se passa em uma batalha. Um jogador pode estar muito interessado no motivo da batalha e todo o jogo político que existe por trás. Outro jogador, por sua vez, pode estar mais interessado nos diálogos e dramas que acontecem entre os combatentes. Um terceiro jogador pode não se interessar em nenhum desses detalhes. Apesar do enredo fixo, essa história pode ser contada com enfoque em sua trama política, desenvolvimento emocional ou nenhum enfoque particular. Esse tipo de variação de narração poderia ser modelado, por exemplo, em cima da variação de dois parâmetros: “grau político” e “grau emocional”.

Por outro lado, há uma maneira diferente de flexibilizar uma história, mais comum em jogos: seguir diferentes enredos dependendo das escolhas e interação do jogador com o ambiente. Nesse caso, o jogo possui uma "árvore de escolhas" com enredos pré-determinados. A história é montada dinamicamente: as escolhas explícitas/ações do personagem definem o ramo da árvore que é seguido. Por exemplo, se o jogador executa uma ação, uma batalha pode ser vencida. Caso contrário, ela é perdida. A “árvore de escolhas” é limitada à criação de todos os ramos da história manualmente. Por isso, esse método é trabalhoso e tem a desvantagem de ter os pontos de ramificação facilmente distinguíveis para o jogador, o que é pior para a imersão na história.

Como alternativa para flexibilizar histórias, pode-se considerar um algoritmo que monta as histórias do zero, a partir de elementos básicos, como coisas, personagens e lugares. Essa solução é trabalhosa e aparentemente difícil de dar bons resultados - afinal, é difícil ou mesmo impossível fazer um algoritmo substituir a criatividade de bons escritores. Por outro lado, um algoritmo que apoia a geração de histórias e depende de escritores para sua geração - se alavancando da criatividade humana - pode ser concebível de maneira a dar resultados satisfatórios. Essa abordagem permitiria gerar histórias com diferentes enredos (como uma batalha ser vencida ou perdida), dependendo das ações do jogador, ou gerar histórias com

enredo fixo, mas sob diferentes perspectivas (como variações de “grau político” e “grau emocional”). Idealmente, os escritores poderiam escolher quais elementos da história desejam fixar e quais ficam a critério do gerador.

Portanto, esse trabalho pretende desenvolver um algoritmo com essas características de geração dinâmica de histórias aceitando a personalidade do jogador. Diante da magnitude desse problema, ele será desenvolvido de forma limitada, focada nos requisitos que serão definidos. Para determiná-los, há duas abordagens que podem ser utilizadas: uma delas é a “*top*” ou “abstrata”, que foca nos propósitos do algoritmo (objetivos como os que foram previamente citados). Outra é a “*bottom*” ou “concreta”, que foca no interface do algoritmo (como ele se apresenta ao usuário; como é utilizado). A definição de requisitos requer um pensamento "*top-bottom*" - o que quer dizer começar pelas abstrações para se pensar e definir sua interface. Como o gerador de histórias é complexo, em alguns momentos a abordagem "*bottom-top*" (i.e. começar pensando na interface do algoritmo para realizar as abstrações) também pode ser utilizada para garantir a viabilidade e eventual reconsideração de requisitos pensados no padrão "*top-bottom*".

Em uma visão "*top",* deseja-se fazer uma ponte entre personalidade e perspectivas de *storytelling*. A primeira coisa a se perguntar é de que se tratam essas perspectivas. Como o algoritmo deve ser independente do gênero da história, resta o que há em comum entre as histórias no escopo deste trabalho: elas almejam entreter. Dessa maneira, parte-se do princípio que o objetivo principal de uma história qualquer é gerar imersão para o interlocutor. Para construir imersão nesse escopo, é necessária uma história consistente e interessante, visando um bom entretenimento.

Alguns pontos importantes de consistência são:

- Relações entre os personagens bem definidas;
- Dinâmica/lógica do universo bem definida;
- Encadeamento consistente de eventos, sem contradições.

Para entreter e atrair interesse, existem diversas técnicas e padrões que se repetem:

- Personagem com o qual o interlocutor pode se identificar;
- Desenvolvimento emocional do personagem;
- Desenvolvimento das relações entre personagens;
- Sensação de progressão do personagem (prestígio, conhecimento, habilidades);
- Sensação de progressão da história;
- Problematização no enredo, trama;

- Reviravoltas no enredo (“reproblematização”).

A imersão é obtida com os fatores de consistência e interesse, mas também renovando o interesse do usuário, ao recompensá-lo por prestar atenção na história. Por exemplo, ela o satisfaz ao apresentar novos fatos sobre eventos que já foram relatados, explicando mistérios que antes foram colocados. Outra forma de recompensá-lo é fazer os eventos da história alterarem as relações entre os personagens. Isso é interessante para o interlocutor que conhece o relacionamento anterior entre eles. Existe um equilíbrio importante entre dar satisfação e recompensas continuamente ao usuário e gerar os problemas e tramas que o prendem na história. Assim, é importante fazer referências a fatos já narrados e ao universo da história ocasionalmente.

Esses conceitos são importantes para diversos gêneros de histórias. Por exemplo, em histórias épicas, a “Jornada do Herói” (CAMPBELL, 2004) é um clichê de progressão do personagem, que geralmente vem de origem humilde e luta até o topo. Em jogos, histórias visam trabalhar as emoções humanas, apresentar dilemas morais ou apresentar reflexões. Até as histórias mais simples almejam ser memoráveis, empregando os conceitos de consistência e interesse.

São esses os pré-requisitos da geração de histórias. O algoritmo deve construir as relações entre personagens e a lógica do universo. Ele também deve garantir a consistência no encadeamento de eventos. Os personagens devem ter traços emocionais, uma medida de progressão de suas habilidades/conquistas e medida das relações com outros personagens. A história deve ambientar uma trama que se encaixa coerentemente em seu universo e envolve o(s) personagem(ns) principal(is).

Resta definir a modelagem de "blocos de construção" básicos através dos quais o escritor expressa sua criatividade. Também é necessário conectá-los com o “*input*” de personalidade do jogador e definir parâmetros de restrição de liberdade em geração de eventos. Para facilitar essa modelagem, volta-se a abordar a parte mais concreta do projeto.

Em segunda instância, portanto, considera-se a visão “*bottom*” do algoritmo. A geração de histórias a partir de blocos é tão genérica que deve ser detalhista, incluindo todos os elementos físicos, emoções humanas, lugares da narrativa e conhecimentos gerais, culturais, atuais e históricos. Isso é quase impossível de implementar sem interferência humana que é sujeita à existência desses fatores e possui opinião e visão próprias dos fatos. Para simplificar, pode-se considerar a geração de histórias de um gênero em particular. Ele costuma possuir elementos conhecidos e esperados em suas histórias, cujas presenças nelas já refletem

aspectos culturais e conhecimentos gerais do ser humano. Por exemplo, nas fantasias medievais há reis, rainhas, príncipes, princesas, cavaleiros, magos, bruxos, monstros, arqueiros, espadas, itens mágicos, reinos, castelos, florestas, rios, lagos, política, guerras. As histórias mais famosas dessa temática consideram universos com diversas raças além de humanos, como elfos, trolls, anões, gigantes, orcs e variações parecidas. Elfos e anões, por exemplo, são encontrados na mitologia nórdica, assim como gigantes, que também existem na mitologia grega e romana. Com elementos e tramas suficientes comuns a um gênero, inevitavelmente se herdam conhecimentos humanos gerais e situações comuns em histórias. A geração de histórias pode partir, por exemplo, do fato de que “bruxa” e “cavaleiro” são usualmente antagônicos no contexto medieval. Utilizar elementos específicos de uma temática como blocos de criação de história pode fazer a narrativa ficar repetitiva. Esse problema pode ser amenizado caso elementos suficientes sejam usados.

O objetivo da proposta, contudo, é independer do gênero da história. Logo, precisa-se de uma solução sem elementos específicos, como “príncipe” e “bruxa”. Por outro lado, o usuário do algoritmo pretende desenvolver uma história específica, então ele tem uma temática em mente. Portanto, pode-se delegar a ele a tarefa de cadastrar os blocos de construção do algoritmo. Cabe a ele definir os personagens ou tipos de personagens e seus tipos de relacionamento mais comuns, como são vistos pela sociedade. O algoritmo deve disponibilizar maneiras de se expressar tais informações em um banco de conhecimento. Em termos de uma ontologia, os blocos podem ser específicos - como um personagem com nome e personalidade bem definidos - ou podem ser genéricos - como um tipo de personagem e suas características mais comuns (e.g o tipo "príncipe", cuja característica é ser corajoso). A personalidade do jogador não determinará o gênero da história, mas será utilizada em tempo de execução. A título de exemplo, um personagem principal do tipo "príncipe" pode ser gerado com personalidade parecida com a do jogador, mas com a característica de ser corajoso. Assim, ele se identificará com esse personagem. O escritor da história pode definir que tipo de personalidade de jogador implicará nas ações dos personagens, ou fixá-las para que fiquem independentes do jogador. O algoritmo também saberá fazer um pouco disso, pois saberá associar as emoções dos personagens com personalidades dos jogadores.

Deve-se discutir os propósitos do gerador de histórias, e dessa forma, os requerimentos do algoritmo. Eles são os requisitos do que deve ser possível cadastrar no banco de conhecimento. Como as personalidades do jogador são entradas do algoritmo e os personagens possuem a própria personalidade, alguns pontos da geração devem depender no

modelo de personalidade - eles serão chamados de "pontos de personalidade". O modelo de personalidade utilizado será o “*Big Five*” (PSI, 2010). Os pontos de personalidade serão tuplas com cinco valores que são números reais entre -1 e 1 cada valor representa uma dimensão do BigFive: "abertura", "conscienciosidade", "extroversão", "amabilidade" e "neuroticismo". A propensão a sentir um tipo de emoção (como "raiva") depende do ponto de personalidade. A personalidade com maior nível de “neuroticismo” do *BigFive* está mais propensa à raiva do que uma com menos. Em termos práticos, pode-se dizer que a distância entre a personalidade com maior neuroticismo e o ponto de personalidade "emoção raiva” é menor do que com um tipo com menos neuroticismo. Os pontos de personalidade serão definidos durante o desenvolvimento de requisitos, a partir de escolhas bem fundamentadas.

Para definir os requisitos do banco de conhecimentos, é necessário definir o conceito de tempo. Muitas variáveis do banco de conhecimento precisam depender do tempo, pois as histórias têm progressão e seus elementos são mutáveis, então situam-se temporalmente. Elas devem possuir início e fim, com a possibilidade de existirem eventos que ocorreram antes do início. Em todo momento da geração, o tempo segue adiante. O tempo corrente é o tempo do universo da história. Os personagens devem poder trocar informações entre si, portanto podem referenciar outros instantes de tempo: um pode contar ao outro o que ocorreu no passado ou especular sobre o futuro. A frequência de aparecimento dessas referências ao passado ou futuro podem depender dos pontos de personalidade dos personagens, pois é argumentável que pessoas ansiosas são mais preocupadas com o futuro e pessoas autoconscientes refletem muito sobre o próprio passado. Com o tempo, podem-se definir eventos: acontecimentos (ações, caracterizados por verbos) que ocorrem um momento do tempo. Simplesmente, os eventos são ações que dependem de zero, um ou mais sujeitos. Eles englobam várias ações, representando uma ocorrência importante para o desenvolvimento da trama atual.

6. Especificação de Requisitos

Em seguida, determinam-se requisitos para a ontologia do gerador de histórias. Cada elemento é detalhado a seguir:

6.1. Requisitos Funcionais

- Gerador de Histórias:

Biblioteca que cria a história do jogo de forma interativa e iterativa. Recebe como input inicial um banco de conhecimento do gênero da história a ser servida e o modelo do jogador (personalidade). Ela utiliza o estado do universo da história em cada passo da iteração, gerando novas saídas que afetam o estado do universo do jogo. Cada iteração ocorre após uma consulta à história. O gerador deve funcionar independentemente da implementação específica de um jogo. A implementação do jogo pode ser conectada à história por meio de um mapeamento do estado do universo da história com o jogo, utilizando a biblioteca.

- História:

Consiste em uma sequência de eventos que culminam no final de uma trama, coincidindo com o fim do jogo. Deve ser gerada durante o andamento de um jogo, de forma a utilizar seus elementos para criar uma narrativa incremental não obrigatoriamente determinística que avança diante da passagem do tempo, ações e interações do jogador.

- Tempo:

Antes do jogo ser iniciado, um tempo já pode ter passado, representando o passado do universo da história. Ele abriga uma sequência de eventos independente das ações do jogador. Ao final dessa sequência de eventos, o estado desse universo serve para a configuração inicial do jogo.

- Tempo corrente:

É o momento temporal atual da partida. Se inicia em zero com o início do jogo e deve ser estritamente incrementado com o decorrer dele. Pode ser o momento de ocorrência de pelo menos um evento ou situa-se entre a ocorrência deles.

A maneira de determinar a passagem do tempo corrente é específica da implementação do jogo, desde que siga essas regras. Dessa forma, o gerador deve receber o tempo corrente quando uma consulta é realizada. Um momento do passado pode ser referenciado pelo valor que o tempo corrente possuía nesse momento. Além disso, momentos anteriores ao início da história possuem tempos negativos.

- Passado:

O passado é o conjunto de todos os valores de tempo menores que o tempo corrente - incluindo valores negativos.

- Enredo:

Conjunto de todos os eventos da história, incluindo os que ocorreram antes do tempo inicial. O enredo se constrói simultaneamente com a história.

- Entidades da história:

São os elementos importantes para a história que podem ter características fixas ou variáveis durante ela - os chamados "estados da história". A entidade pode ser um personagem, uma cidade, uma arma, uma árvore ou um animal, desde que tenha um papel importante na história. Pode ser representada por um substantivo.

As entidades são criadas pelo usuário e fornecidas como input inicial do gerador. Dessa maneira, elas podem refletir os objetos e imagens que foram de fato implementados no jogo.

- Tipos e categorias das entidades:

Toda entidade tem um único tipo específico. Alguns tipos devem ser nativos da biblioteca, como personagem, lugar, animal e coisa - um objeto. Outros podem ser definidos pelo usuário. Esses tipos são importantes para diferenciar a natureza básica das entidades. Também podem ser vistos como as categorias mais genéricas.

As categorias das entidades são divisões dos tipos. Toda categoria é uma subcategoria de outra ou de um tipo, chamada de superclassificação. Por exemplo, “homem” ou ”mulher” podem ser subcategorias do tipo “personagem”; “príncipe” também pode ser uma subcategoria de “homem”. As entidades podem ter uma categoria ou estar diretamente superclassificadas em um tipo. O tipo e todas as subcategorias da divisão de uma entidade são chamadas de classificações - se “Arthur” é superclassificado em “príncipe”, então ele é classificado em “príncipe”, “homem” e “personagem”. A entidade não possui mais de uma superclassificação, o que equivale a um modelo simples de herança. Se esse modelo não for suficiente para descrever as características de um grupo de indivíduos, uma alternativa é utilizar estados da história para fazer essa caracterização.

Não há categorias nativas: todas devem ser definidas pelo usuário caso faça sentido no seu gênero de história. Elas servem para definir características e estados comuns de suas entidades.

- Valores escalares:

São valores que complementam informações sobre as entidades. Ao contrário da entidade, o valor escalar não representa nada por si só. Ele é usado no estado para agregar valor quantitativo ou qualitativo. Os valores escalares se dividem em:

    - Número:

É do tipo “número”. Pode ser usado para valores quantitativos - “a floresta e a cidade se distanciam em 3 quilômetros”.

Também pode ser usado para valores qualitativos, em estados com uma escala de referência - “o totó é bravo 7/10”.

- Valor enumerável:

É usado para valores qualitativos. Pode ser representado por um adjetivo ou substantivo, dentro de um conjunto de valores possíveis. O tipo de um escalar enumerável é o nome do conjunto enumerável em si. Considere a enumeração de valores [pequena, média, alta], chamada de “altura”. Um escalar do tipo “altura” pode ser usado em um estado como “o menino tem altura média”.

- Estado da história:

É uma informação que representa o estado de uma ou mais entidades em um momento do tempo. Também pode ser uma informação acerca do universo da história, caso qual não está ligado a nenhuma entidade. Pode ser uma característica de uma entidade ou representar uma relação entre entidades.

Pode ser representado por uma oração nominal ou uma verbal que representa um estado, como em “o menino possui a bola”. Neste exemplo, “menino” e “bola” são as entidades associadas ao estado. Outro caso é “o menino é bonito”, com uma única entidade sendo caracterizada por uma oração nominal. O estado da história pode conter zero ou mais valores escalares.

- Tipos de estado da história:

Similarmente às entidades e escalares, os estados possuem tipos. Um tipo pode ser representado por um verbo de ligação e uma característica ou um verbo intransitivo/transitivo que representa um estado, como o “possui” do exemplo anterior. Cada tipo de estado se refere a um número fixo de entidades/escalares de tipos fixos, chamados de argumentos.

Alguns tipos são nativos da biblioteca, como “conhece” e “possui”. O tipo “conhece”, por exemplo, tem dois argumentos do tipo de entidade “personagem”. Os tipos de estado podem ser representados com a notação:

- *conhece(entidade ‘personagem’, entidade ‘personagem’)*

Essa notação se chama “assinatura do tipo”. Nesse caso, um estado desse tipo pode ser representado de maneira similar:

- *conhece(joão, maria)*

Esse estado equivale à frase “joão conhece maria”. A notação é a "assinatura do estado".

O usuário também pode definir seus próprios tipos, como os exemplos a seguir:

- *seDistanciamEmQuilômetros(entidade ‘lugar’, entidade ‘lugar’, escalar ‘número’)*

A frase “a floresta e a cidade se distanciam em 3 quilômetros” é um estado desse tipo, e pode ser representada por:

- *seDistanciamEmQuilômetros(floresta, cidade, 3)*

Da mesma maneira, pode-se definir os tipos:

- *éBravoNaEscala10(entidade ‘animal’, escalar ‘número’)*
- *temAltura(entidade ‘personagem’, escalar ‘altura’)*

Os seguintes estados são desses tipos, respectivamente:

- *éBravoNaEscala10(totó, 7)*
- *temAltura(menino, alta)*

Como as entidades podem ter categorias, também é possível declarar tipos de estado que atuam apenas sobre categorias específicas. O exemplo anterior do tipo “éBravoNaEscala10” pode ser modificado para não aceitar qualquer tipo de animal, mas apenas animais da categoria “cachorro”. A assinatura desse tipo seria então:

- *éBravoNaEscala10(entidade ‘cachorro’, escalar ‘número’)*

Alguns tipos de estado podem ter mais de uma assinatura, caso existam usos com diferentes tipos de argumento. Por exemplo, o último exemplo pode ser estendido para outra categoria de animais se outra assinatura for adicionada:

- *éBravoNaEscala10(entidade ‘animal’, escalar ‘número’)*

Outra opção é não limitar o tipo da entidade ou escalar. Isso tem menos aplicações práticas, mas tem seu uso. O exemplo abaixo serve para qualquer entidade:

- *anoDeCriação(entidade, escalar ‘número’)*

Alguns tipos de estado existirão nativamente, outros serão adicionados pelo usuário. Dois exemplos de tipos nativos são:

- "*relacionamentoEntre*" - determina o relacionamento de um par de personagens.
- “*conhecePessoa*” - determina se um personagem conhece o outro.

Há ainda dois tipos de argumentos menos usuais que podem ser usados nos estados: eventos e os próprios estados. Eles são úteis para expressar como o universo lida com as informações da história. São utilizadas por outro tipo de estado nativo: “*sabeQue*”. Ele determina quando um personagem sabe que a ocorrência de um evento ou a existência de um estado é verdadeira ou não. Por exemplo:

- *sabeQue(maria, temMedoDeBarata(joão))*

Esse estado indica que Maria sabe se o estado “joão tem medo de barata” é verdadeiro. Esse tipo de estado tem duas assinaturas. Elas são:

- *sabeQue(entidade ‘personagem’, estado)*
- *sabeQue(entidade ‘personagem’, evento)*

Os estados que recebem um estado como argumento têm um nome especial - metaestados. Eles possuem um detalhe: seu argumento não pode ser um metaestado em si. Caso contrário, construções infinitas seriam possíveis:

- *sabeQue(maria, sabeQue(joão, sabeQue(maria, sabeQue(joão, ...*

Talvez até certo nível, seria útil ter metaestados que recebem metaestados. Por exemplo, se Maria souber que João está ciente de seu casamento com Ricardo, ela não precisa informá-lo das novidades. Nesse caso, um tipo de estado que representa essa construção pode ser criado:

- *sabeQueSabeQue(maria, joão, casados(maria, ricardo))*

Assim, o nível a que chega a “metainformação” fica a critério do usuário.

- Evento:

Representa uma ocorrência relevante para a história, que necessariamente altera o estado do universo da história. O evento ocorre em um momento do tempo, chamado de tempo de ocorrência. Os eventos ocorrem juntamente com a progressão do jogo. Ou seja, um evento só é criado quando de fato começa a existir. Portanto, o instante de ocorrência do evento nunca é maior que o tempo corrente, pois nesse caso ele ainda não começou a ocorrer. A duração de um acontecimento da história normalmente é irrelevante para a implementação do jogo. Nas poucas exceções, ele pode ser modelado como dois eventos - um de início e outro de final do acontecimento.

Os eventos podem ser representados por orações verbais que indicam uma ação ou um acontecimento, como “começa a chover” ou “joão conhece maria”. Portanto, eles também têm assinatura com zero ou mais argumentos:

- *chove*
- *conhece(joão, maria)*

Todo evento ocorre como resultado do disparo de um gatilho. Sua própria ocorrência pode disparar outro(s) gatilho(s) em um momento do futuro. Neste caso, ele tem um efeito que perdura no tempo. Por exemplo: “joão coloca a torta no forno” pode disparar o gatilho “a torta fica pronta” 20 minutos após sua ocorrência. Esse gatilho, por sua vez, pode causar o evento “joão pega a torta” ou “a torta queima”.

Os eventos podem ter tempos de ocorrência coincidentes - portanto podem ser simultâneos -, exceto se tiverem o mesmo tipo. Eles ocorrem independentemente da

consciência do jogador. Assim, cabe à implementação do jogo a escolha de expor os acontecimentos ao jogador de forma explícita ou omiti-los e deixar que sejam percebidos pela alteração do estado do universo da história, que pode ser espelhado para o jogador pela interface do jogo.

Cada personagem tem o próprio conhecimento da ocorrência dos eventos. Um personagem pode não estar ciente dessa ocorrência, mesmo que tenha consciência de todas as suas consequências.

- Tipos de eventos:

Todo evento possui um tipo, que existe na base de conhecimento antes da partida começar. O tipo do evento determina se ele pode ou não ocorrer, as mudanças que ele ocasionará e os gatilhos que ele pode disparar. Na geração da história, múltiplos eventos de um mesmo tipo podem ser instanciados.

Um tipo de evento pode ser representado por um verbo transitivo ou intransitivo que representa uma ação ou ocorrência. Ele também é representado por uma lista de variáveis, que representam as entidades ou escalares sobre as quais ele vai atuar. Quando um evento é instanciado, essas variáveis recebem valores que caracterizam sua ocorrência. Por exemplo, “viaja(Pessoa, Lugar1, Lugar2)” é uma representação de um tipo de evento em que uma pessoa viaja de um lugar para outro. “Pessoa”, “Lugar1” e “Lugar2” são variáveis que são instanciadas quando um evento desse tipo ocorre: “viaja(harry, londres, hogwarts)” representa um evento instanciado, a título de exemplo.

O tipo do evento determina pré requisitos do estado do universo da história para que ele ocorra. Cada tipo de evento está associado a um ou mais gatilhos. O evento de um tipo ocorre sob duas condições: seus pré requisitos são atendidos e um de seus gatilhos é ativado. O momento de ocorrência é o momento do disparo do gatilho.

O evento pode causar mudanças no estado do universo da história, removendo estados ou os adicionando. Por exemplo, o evento “joão anda da cidade para o campo” remove o estado “joão está na cidade” e adiciona o estado “joão está no campo”.

Existem tipos de eventos nativos do gerador, como um que representa um personagem entregando um objeto para outro. Esse tipo tem o pré-requisito de que os personagens estejam no mesmo lugar físico na história. Outros tipos também devem ser cadastrados pelo usuário da biblioteca, para representarem seu contexto específico.

- Estado do universo da história:

É o conjunto de todos os estados da história, ou seja, caracterizações de todas as entidades que determinam como a história se situa temporalmente. Esses estados devem ser relevantes para o andamento da história a fim de concluir a trama. Eles se desenvolvem ao longo do tempo e são meios de interesse e imersão do jogador. Os estados são gerenciados pelo gerador de histórias, mas deve haver uma maneira externa de impor seus valores de forma coerente. Pode-se definir requisitos funcionais de modelagem das entidades nativas que determinam estados nativos do gerador:

- ❏ Personagens:

Cada personagem é como se fosse um ser pensante que possui consciência própria sobre os eventos que ocorreram e outros estados da história. Um personagem possui um conjunto de outros personagens que ele conhece, e possui uma relação bem definida com cada um deles. Esses são dois tipos de estado da história. Como "*sabeQue*" é um metaestado, o personagem também tem consciência das relações entre outros personagens.

Cada personagem possui sua própria personalidade no modelo "*BigFive*". Ele também pode opcionalmente possuir uma motivação, que moverá suas ações no decorrer da trama (através de uma heurística). Se possuir motivação, ele é chamado de personagem ativo. Se não, de personagem passivo.

Cada par de personagens possui um tipo de relação, que pode ser indeterminado caso eles não se conheçam. As relações são modeladas também pelo "*BigFive*" (5 valores escalares), apesar de que o modelo não foi feito para este uso. O modelo de personalidade de cada personagem individualmente não interfere no modelo da relação. A categorização da relação (amor, amizade, rivalidade) depende do input do banco de conhecimento do gênero.

Outros estados relacionados ao personagem também podem ser criados pelo usuário da biblioteca. Nesse caso, devem-se criar características que possam impactar o andamento da história, em detrimento de atributos voláteis como emoção momentânea dele. Em suma, os principais são:

- ❖ Conhecimentos sobre o estado do universo
- ❖ Personalidade
- ❖ Relações com outros personagens
- ❖ Motivação (opcional)
- ❖ Atributos customizados (habilidades, prestígio, estado da trama)

Os personagens controlados pelo jogador são tratados como personagens quaisquer. Todavia, em tempo de implementação, pode-se forçar que ele possua os valores de conhecimento, personalidade e motivação do modelo do jogador.

- ❏ Lugares:

Locais onde a história acontece. Quase todo evento ocorre no mínimo em um lugar, mas essa não é uma restrição. Todo personagem está presente em um único lugar, em qualquer instante de tempo. Similarmente ao personagem, também pode-se criar estados customizados relacionados aos lugares (atentando às mesmas regras).

- Trama:

Deve consistir em uma sequência de objetivos que metrifica o progresso da história. Quando o último objetivo é cumprido, a história se encerra. Cada objetivo é uma lista de estados da história que devem ser verdadeiros ou falsos. Isso significa que ele é cumprido quando o estado do universo da história é adequado.

- Gatilho:

Os gatilhos são pré-requisitos de ocorrência de alguns eventos. Eles são disparados em um momento da história, contendo o tempo corrente dela - o tempo de disparo do gatilho. O disparo pode gerar ocorrências de no máximo um evento no tempo de disparo.

Os gatilhos possuem tipos que devem ser previamente registrados pelo usuário da biblioteca e associados aos eventos. O usuário pode escolher ativar um tipo de gatilho após certas ações do jogador. Um tipo gatilho pode estar associado a zero ou mais eventos.

Cada tipo de gatilho tem uma classificação: ativo ou passivo. Essa classificação afeta a maneira que o processador de eventos organiza a ocorrência deles: se um gatilho passivo é disparado, o evento só ocorre caso vá contribuir para a trama da história. Se um gatilho ativo é disparado, o evento ocorre sempre que possível.

- Consulta:

Uma consulta é um mecanismo para o gerador se comunicar com o meio externo. O usuário da biblioteca pode enviar quando quiser uma consulta (uma mensagem) para o gerador contendo o tempo corrente da história - o tempo da consulta. Então, o gerador responderá com informações sobre o estado do universo da história nesse tempo corrente. Ele tem uma política "*lazy*" de processar os eventos da história - até receber outro sinal externo (outra consulta), a geração da história estará congelada. Quando o usuário enviar outra consulta, o gerador processará todos os eventos que ocorreram no meio tempo - desde a última consulta - e

retornará novamente o estado do universo da história, informando também todos os eventos que ocorreram nesse intervalo. O tempo de consulta é informado ao gerador e deve ser estritamente maior que o enviado da última vez - uma vez que os eventos são executados, não há volta. O tempo corrente continua avançando.

Uma consulta pode estar acompanhada de uma lista de gatilhos, cujos tempos de ativação estão ordenados entre os tempos da última consulta e a atual. Esses gatilhos podem influenciar os eventos que são processados nesse intervalo. Se o tempo de ativação de um gatilho coincidir com o tempo da consulta, qualquer evento consequente dele será processado antes do estado do universo ser retornado.

6.2. Requisitos Não Funcionais

- Performance:

O gerador de histórias deve funcionar em um tempo adequado para o contexto. O usuário não precisa de respostas em tempo real, mas em geral o máximo de 60 segundos em cada iteração é razoável.

- Modularização:

O código deve ser bem modular para simplicidade e fácil compreensão. Quando possível, bibliotecas prontas de estruturas de dados devem ser preferidas.

- Versionamento:

O código deve ser versionado usando git. Cada release deve estar acompanhado de uma tag no modelo “semantic versioning”. Também deverá possuir um “changelog”.

- Documentação:

  O código deverá estar documentado externamente em um arquivo “readme”.

- Confiabilidade:

  O software deverá possuir testes automatizados de cobertura abrangente.

- Normalização de erros:

  Os erros da biblioteca deverão ser retornados em formato padronizado.

- Robustez:

  Eventuais erros semânticos da implementação serão tratados.

7. Arquitetura

A arquitetura do Gerador de Histórias se divide em dois componentes principais: o Estado do Universo da História e o Processador de Eventos, como apresentado na figura 1. O Banco de conhecimento interno ou Base de Conhecimento contém as definições de Eventos, Gatilhos, Entidades, Estados, Trama e Tipos, usados pelo Processador de Eventos. O usuário da biblioteca define seus próprios elementos não-nativos e passa como input para o Gerador por meio de um Banco de Conhecimento Inicial que ele define.

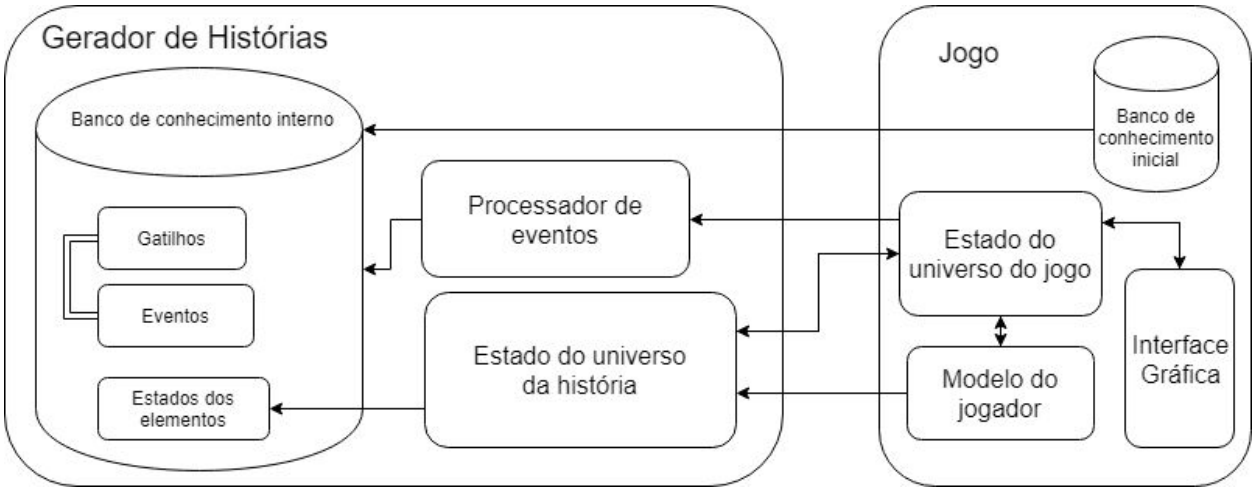

Figura 1 - arquitetura do gerador de histórias

8. Análise do Problema

O fim da história é importante, portanto ela deve progredir de forma a terminar em um estado adequado. O gerador de histórias precisa de direcionamento para montar o enredo. Essa direção é constituída pela trama escolhida pelo escritor, especificando o estado do universo final almejado. Outro fator de direcionamento é uma função heurística que controlará a geração, também definida pelo usuário.

Como a história é montada em tempo real, o jogador compartilha responsabilidade por direcioná-la. Isso ocorre por meio da ativação de gatilhos consequente de suas ações no jogo, como determinado na seção anterior.

Do ponto de vista da responsabilidade do gerador, o conjunto de estados finais que deve permanecer após a geração é obtido a partir do estado do universo inicial e uma

sequência de eventos que o altera. Para escolher os eventos que ocorrerão e satisfazer a trama, o gerador precisa planejar suas ações.

Esse tipo de problema é abordado por um campo da inteligência artificial chamado "*Automated Planning*" ou "Planejamento Automatizado" (GHALLAB et al. 2016). Tratam-se de algoritmos e estratégias para um agente planejar suas ações visando um objetivo. Nessa área, existe uma solução que se adequa bem à ontologia escolhida para gerador na especificação de requisitos: algoritmos de planejamento como o STRIPS - Stanford Research Institute Problem Solver (FIKES et al. 1971). O STRIPS é o primeiro de vários modelos usados para planejamento automatizado que utilizam um modelo de mundo, conjunto de operadores e objetivo a ser alcançado. Os operadores são funções que transformam o modelo de mundo e o objetivo é o mundo final desejado. Uma linguagem descritiva é utilizada para definir esses modelos para que um planejador possa atuar sobre o problema. A linguagem PDDL (GHALLAB et al. 1998) foi criada nesse contexto, como forma de padronizar a descrição de uma ontologia sobre a qual se planeja. O PDDL tem suporte para STRIPS básico e outros recursos sintáticos para definir os operadores de planejamento, suas condições e efeitos.

Apenas um planejador clássico não seria suficiente para o problema definido neste projeto, uma vez que a interatividade da história por influência do jogador compartilha a responsabilidade de direcionar o enredo da história. Decidiu-se utilizar um algoritmo de planejamento em conjunto com o sistema de gatilhos especificado para atender essas especificações.

9. Definição de Tecnologias

Com base nisso tudo, escolheu-se utilizar a linguagem Prolog (BLACKBURN, 2006). Prolog é uma linguagem de programação lógica descritiva utilizada em problemas de inteligência artificial, banco de dados, provas de teoremas e computação linguística - devido à aplicação de gramática formal. Ela é baseada em lógica formal de primeira ordem, portanto possui natureza descritiva muito adequada para a definição ontológica de domínio e para trabalhar com planejamento automático. A versão digital do livro "Learn Prolog Now!" (BLACKBURN, 2006) foi utilizada para aprendizado dessa linguagem. É uma fonte de informações completa sobre a sintaxe e boas práticas em Prolog.

Um dos critérios da escolha das tecnologias utilizadas foi a fácil portabilidade. O software criado neste projeto não foi produzido para mais de uma arquitetura de processador

ou sistema operacional, mas esse poderia ser considerado um requisito desejável. Na prática, o projeto foi desenvolvido em uma máquina Windows com processador de 64bits.

Para fazer a interface do gerador de histórias com o jogo, decidiu-se usar a linguagem C# - devido ao fato de que as linguagem que executam no ambiente .NET são muito utilizadas para a produção de jogos de PC. Muitos jogos são feitos em C++ e atualmente a plataforma Unity é muito utilizada com C#. O C# também é uma linguagem interpretada, portanto permite a portabilidade para outros sistemas com o uso do .NET Core - *framework* utilizada neste projeto. A IDE Visual Studio 2019 foi utilizada para o desenvolvimento, com a criação de uma solução dois projetos. O primeiro projeto é "story-generator", uma interface do gerador de histórias com o jogo, que é uma "*Class Library*" - um *proxy* para C# do "estado do universo" e "processador de eventos" (figura 1) programados em Prolog. O segundo projeto é "generator-test", um programa "*Console App*" para teste manual da biblioteca "story-generator".

Há mais de um software que implementa o Prolog. Após o estudo das opções, decidiu-se utilizar o SWI-Prolog - por ser a implementação mais amplamente utilizada, bem completa e bem documentada. O SWI-Prolog também acompanha um ambiente de integrações, bibliotecas e funcionalidades nativas, é atualizado regularmente e possui versões atuais compatíveis com os sistemas Windows (32 e 64 bits), Linux e OSX. Segundo o livro "Learn Prolog Now!", esse software possui uma "implementação sofisticada" da especificação do Prolog - como o tratamento de erros na ausência do chamado *"occurs check"* (BLACKBURN, 2006). Por último, o SWI-Prolog também possui ferramentas úteis para auxiliar o desenvolvimento de aplicações - como os predicados nativos "make/0" e "edit/1" - e implementação de procedimentos não determinísticos - com o predicado nativo "random/3". A documentação do SWI-Prolog foi a principal fonte usada para aprendizado sobre o ambiente de programação desse software (FRUEHWIRTH, 2012).

Escolheu-se usar o Visual Studio Code para programação em Prolog. Instalou-se a extensão "VSC-Prolog"[4], por ter um amplo uso - a quantidade de downloads era 31.073 na data de acesso -, suporte a SWI-Prolog (conhecimento de funcionalidades definidas por esse software que não estão no Prolog padrão) e por disponibilizar recursos úteis para o ambiente de desenvolvimento: presença de um linter, *Syntax Highlighting* (colorização de código), *snippets* de código, comandos de integração com a CLI do SWI-Prolog, *debugger* com *breakpoints/ trace* e algumas funcionalidades de *IntelliSense*, como *autocomplete*, *tooltips*, *helpers* (ex. importação e exportação) e ajuda para refatoração de código.

[4] Disponível em <https://github.com/arthwang/vsc-prolog>. Acesso em 26 out. 2019.

Para integrar o Prolog e o C#, obteve-se a biblioteca dinâmica SWIPlCs. Ela é recomendada pelo próprio site do SWI-Prolog[5]. Ela está presente no gerenciador de pacotes NuGet da Microsoft[6], mas não na versão mais atualizada. Utilizar um gerenciador de pacotes é uma ótima prática no ponto de vista da engenharia de software, pois ajuda a manter as dependências do software atualizadas e a fazer um controle de versionamento. Porém, como infelizmente os autores não mantém a biblioteca atualizada no gerenciador, baixou-se a versão mais recente dela do github[7]. Isso é ruim para a manutenção da biblioteca com futuras atualizações do SWI-Prolog, mas foi aceitável para realização deste projeto. Vale notar que o SWI-Prolog é atualizado com uma frequência razoável - na última verificação (27/10), o último commit do usuário “SWI-Prolog” foi no dia 19/10. A versão utilizada no projeto foi a 8.0.3-1 (a mais recente em 03/11). A biblioteca SWIPlCs estava atualizada, mas seu versionamento estava errado (a versão utilizada foi a 1.1.60605.0). Para funcionar com o SWI-Prolog 8.0.3-1, foi necessário fazer um *build* na máquina utilizada para o projeto, seguindo as instruções de sua documentação.

O SWI-Prolog e a biblioteca SWIPlCs possuem *deploys* diferentes para cada tipo de arquitetura de CPU (32 e 64 bit). Neste projeto, utilizou-se uma única versão: de 64 bit. Ela foi escolhida considerando a vantagem de um espaço de endereçamento maior, apesar do maior uso de memória relatado pela documentação do SWI[8]. Portar a biblioteca do Gerador de Histórias para outras arquiteturas de sistema e processador requer a mudança de ambos os *deploys* no projeto.

Para versionar o software, escolheu-se usar Git e o site Github. Um repositório privado foi criado com um .gitignore para arquivos do Visual Studio, fornecido pelo próprio Github[9]. Esse arquivo serve para listar todos os arquivos do projeto que não importam para o versionamento. Foi utilizado o cliente Git do VS Code para auxiliar nesse processo. A CLI do Git foi usada apenas para comandos diferenciados eventuais, como um *commit* com opção de *amend* ou um *rebase*, de forma a manter o repositório organizado. Alguns pedaços de código que foram usados apenas para testes foram salvos no repositório para fins demonstrativos, na pasta “POC” (*Proof of Concept*). Eles são referenciados em alguns momentos deste relatório.

---

[5] Disponível em <https://www.swi-prolog.org/contrib/CSharp.txt>. Acesso em 27 out. 2019.
[6] Disponível em <https://www.nuget.org/packages/SWI.Prolog/>. Acesso em 27 out. 2019.
[7] Disponível em <https://github.com/SWI-Prolog/contrib-swiplcs>. Acesso em 3 nov. 2019.
[8] Disponível em <https://www.swi-prolog.org/pldoc/man?section=64bits>. Acesso em 9 nov. 2019.
[9] Disponível em <https://github.com/github/gitignore/blob/master/VisualStudio.gitignore>. Acesso em 27 nov. 2019.

Normalmente o software SWI-Prolog não estaria incluído no repositório para o versionamento, porque é instalado no sistema. Porém, descobriu-se que a integração realizada pela SWIPICs funciona apenas quando as DLLs da instalação do SWI (“libswipl.dll” e suas dependências) estão em um caminho acessível pelo sistema de arquivos em tempo de execução do programa. Para desenvolvimento e debug, isso seria facilmente resolvido configurando a pasta de trabalho (“Working Directory”) do Visual Studio para o local de instalação dessas DLLs. Porém, desejou-se fazer um software auto-contido, cujo *deploy* não dependa da instalação do SWI-Prolog na máquina do usuário do Gerador de Histórias. Portanto, copiaram-se todas essas DLLs e o arquivo “boot64.prc” para um *path* relativo no *build* da aplicação - “lib/swipl” (não se considerou os requisitos legais para redistribuição desses arquivos). Apesar de não serem bibliotecas *linkadas* na interpretação do programa, são utilizadas para rodar um processo do SWI-Prolog - configurando a variável de ambiente “SWI_HOME_DIR” e a pasta de trabalho do programa em tempo de execução. Apenas os arquivos necessários foram copiados, para evitar pesar o projeto - ainda assim, 18,6MB foram ocupados por um total de 59 arquivos. Isso não é o suficiente para ultrapassar o limite de arquivos em um *commit* do Github.

Durante o desenvolvimento do software, a plataforma Trello foi utilizada para organização de tarefas. Dessa maneira, toda funcionalidade pensada foi adicionada a ela para garantir que nada ficasse esquecido.

10. Implementação do Planejador

Para utilizar um planejador para a trama da história, uma linguagem descritiva como o PDDL deve ser utilizada. O próprio Prolog é uma linguagem desse tipo, então faz sentido utilizá-lo, para não se criar mais integrações que podem comprometer a performance e simplicidade do programa. Uma POC (*Proof of Concept* - Prova de Conceito) com um algoritmo de planejamento simples foi implementada para estudar a viabilidade relacionada à eficiência de um algoritmo de planejamento em Prolog. Se mediram os tempos de execução sob diversas entradas, obtendo resultados satisfatórios para domínios com poucas variáveis.

Dessa maneira, implementou-se um planejador similar ao STRIPS, com a própria maneira formal de declarar um modelo de mundo, parecido com o PDDL (BECKER, 2015). Esse planejador está no módulo “utils/planning.pl” e seus testes estão no arquivo “tests/planning.tests.pl”. Para utilizá-lo, o usuário deve começar definindo o modelo do mundo e

os possíveis procedimentos, o que se chama “domínio”. Depois, ele deve definir fatos sobre o mundo e o objetivo que se quer alcançar. Isso se chama “problema”. Com o domínio e o problema definidos, o planejador retornará uma sequência de procedimentos que devem ser realizados para se alcançar os objetivos a partir dos fatos iniciais. Os principais elementos do planejador que compõe o domínio e o problema são:

- Objetos:

  “Coisas” que existem no mundo. Cada objeto tem um tipo definido. Por exemplo, “cidade” é um objeto cujo tipo é “lugar”. “escadas” é um objeto cujo tipo é “objeto” e assim por diante.

- Fatos:

  São informações determinísticas que representam um fato do domínio. Por exemplo, “grande(cidade)” é um fato que determina que a cidade é grande. “naPosição(escadas, cidade)” é um fato que determina que as escadas estão na cidade.

- Ações:

  São procedimentos que possuem pré-requisitos para serem executados, e quando são executados possuem um efeito - alteram os fatos. A ação “joãoVaiParaACidade” tem como pré requisito o fato “naPosição(joão, campo)”. Se essa ação for executada, ela vai remover o fato “naPosição(joão, campo)” (ele passa a ser falso) e criar o fato “naPosição(joão, cidade)” (passa a ser verdadeiro).

- Objetivo:

  Conjunto de fatos que devem ser satisfeitos pelo planejador. São divididos em duas categorias:

  - ❏ Objetivos de inclusão:

    Fatos que devem estar incluídos na lista final de fatos.

  - ❏ Objetivos de absência:

    Fatos que não podem estar na lista final de fatos.

  Se “naPosição(joão, campo)” é um objetivo de inclusão e “naPosição(maria, campo)” é um objetivo de absência, o planejador deve formular uma sequência de ações que, após serem executadas, farão o fato “naPosição(joão, campo)” ser verdadeiro e o fato “naPosição(maria, campo)” ser falso.

Similarmente ao PDDL, o domínio contém informações sobre os tipos de objetos e os procedimentos ou “ações”. A POC realizada representa um mundo com um personagem “macaco”. O macaco precisa pegar a banana que está pendurada no teto de um quarto, e o planejador precisa definir uma sequência de ações para ele atingir esse objetivo. Definiu-se a existência de uma escada que o macaco pode carregar e usar para alcançar o teto. As ações permitidas ao macaco são:

- mover(X, Y) → move o macaco da posição X para a posição Y;
- carregar(X, Y) → faz o macaco carregar as escadas da posição X para a posição Y;
- subir(X) → faz o macaco subir nas escadas na posição X;
- malhar(F, NovoF) → aumenta a força do macaco de F para NovoF;
- vencer(X) → faz o macaco pegar as bananas na posição X e vencer o desafio.

Para cadastrar essas informações como domínio do planejador, os tipos de objeto e as ações são escritas em um arquivo em prolog seguindo uma notação especial, que é usado como input do módulo. Um exemplo dessa definição de domínio está na figura 2 (os detalhes não são importantes).

```
:- beginDomainDefinition(monkeyDomain).

type(place).
type(character).
type(object).

% actionSpecification(action, typeSpecs,
actionSpec(move(X, Y), [place(X), place(
actionSpec(carry(X, Y), [place(X), place
actionSpec(climb(X), [place(X)], [], [on
actionSpec(workOut(S, NewS), [], [], [st
actionSpec(win(X), [place(X)], [], [onPo

:- endDomainDefinition(monkeyDomain).
```

Figura 2 - definição do domínio do planejador

Os pré-requisitos e efeitos de cada ação são definidos nessa notação. Para que o macaco suba ou carregue as escadas, primeiro ele deve estar na mesma posição que elas. Em seguida, para que ele suba as escadas, definiu-se que ele deve ter a força mínima igual a um.

Finalmente, para executar a ação “vencer(X)”, ele e a banana devem estar na posição X, e ele deve alcançar a banana.

Após definido o domínio, define-se o problema, similarmente ao PDDL. Os objetos são definidos em um arquivo prolog, que é outro *input* do módulo, conforme a figura 3.

```
:- beginProblemDefinition(monkeyDomain).

% Type facts
place(a).
place(b).
place(c).

character(monkey).

object(stairs).
object(banana).

:- endProblemDefinition(monkeyDomain).
```

Figura 3 - definição do problema do planejador

Depois, chama-se uma “função prolog” passando a lista de fatos inicial e o objetivo desejado. Essa função retornará um plano, ou seja: uma sequência de ações que satisfazem os objetivo.

A figura 4 contém a lista de fatos inicial usada no exemplo. O macaco começa na posição “a” e o planejador deve fazê-lo ir até a posição “b”, carregar as escadas até a posição “c”, depois subir as escadas e vencer o desafio. Em qualquer momento antes de subir as escadas, ele deverá “malhar”, pois sua força inicial começa em zero, o que não é suficiente para subir as escadas.

```
facts([
  onPos(monkey, a),
  onPos(stairs, b),
  onPos(banana, c),
  onGround(monkey),
  strength(monkey, 0)
  ]).
```

Figura 4 - lista de fatos iniciais

A figura 5 contém uma chamada para a função e a figura 6 contém o plano que ela retorna. Nota-se que é o resultado esperado que foi comentado.

```
6 ?- monkey:facts(X),
planning:planAStar(monkeyDomain, X, [win], monkey:heuristic, Plan, PlanCost, _).
```

Figura 5 - chamada para a função de planejamento

```
Plan = [move(a, b), workOut(0, 1), carry(b, c), climb(c), win(c)],
```

Figura 6 - plano retornado pela função

O módulo de planning implementado neste projeto recebe uma função heurística definida pelo usuário. Essa heurística associa um custo a cada tipo de ação. A função de planejamento utiliza o algoritmo A* (FERGUSON, 2005) com a heurística recebida para calcular o plano de menor custo que satisfaz o objetivo. A título de exemplo, modifica-se o exemplo anterior para utilizar a função heurística da figura 7.

```
heuristic(actionExecution(carry(b, c),_), 15) :- !.
heuristic(actionExecution(_,_), 5).
```

Figura 7 - função heurística do planejamento

Essa heurística associa o custo 5 a qualquer tipo de ação, exceto à ação “carregar(b, c)”, que ganha custo 15. Isso significa que é mais caro para o macaco carregar a escada diretamente da posição “b” para a “c”. Se ele carregá-la de “b” para “a” e posteriormente de “a” para “c”, o custo total será menor. A função retornará o plano de menor custo - ou os planos de menor custo, caso tenha empate. A figura 8 mostra um desses resultados com a nova heurística - justamente o que se esperava.

```
Plan = [move(a, b), carry(b, a), workOut(0, 1), carry(a, c), climb(c), win(c)],
PlanCost = 30 ;
```

Figura 8 - plano e custo retornados pela função

11. Banco de Conhecimento Inicial

O usuário da biblioteca precisa primeiro especificar as características de seu gênero de história e depois especificar os elementos e características de sua história específica. Ele faz isso escrevendo dois arquivos de input em linguagem Prolog, como explicado a seguir.

11.1. Arquivo de Definição de Gênero

Nesse arquivo serão definidos os tipos de enumeração (escalares), tipos de entidade, de estado, de gatilhos e de evento, tudo conforme definido na especificação de requisitos (Seção 5). Recomenda-se reler essa seção para se entender estas definições. A título de exemplo, cria-se um arquivo para definir uma história do gênero medieval chamado “medievalSpec.pl” (a extensão “.pl” é do Prolog).

Os tipos que serão definidos neste arquivo são aqueles que já não são nativos do Gerador. Caso se deseje substituir algum tipo nativo, primeiro é necessário removê-lo do Gerador e depois se deve criar outro com o mesmo nome. Inicialmente, pode-se declarar quais tipos nativos serão removidos no arquivo “medievalSpec.pl”, caso não serão utilizados ou serão substituídos. A figura 9 mostra um exemplo de alguns tipos sendo removidos.

```
1   % >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> Genre Definition
2
3   % -------------------------------- Remove native types
4
5   removeNativeEntityType(animal).
6
7   removeNativeStateType(knowsThat).
8   removeNativeStateType(thinksThat).
9
10  removeNativeEventType(move(_,_,_)).
11
```

Figura 9 - remoção de tipos nativos do Gerador de Histórias

Apenas tipos de entidade, estado e evento podem ser removidos. A notação para cada um deles é:

- “*removeNativeEntityType(nome_do_tipo_de_entidade).*”
- “*removeNativeStateType(nome_do_tipo_de_estado).*”
- “*removeNativeEventType(nome_do_tipo_de_evento(lista_de_underlines)).*”

No caso dos eventos, a “lista_de_underlines” corresponde à assinatura do evento, trocando os argumentos por “_”.

As próximas linhas do arquivo “medievalSpec.pl” definem os tipos de enumeração. Eles podem ser simplesmente uma lista de valores ou podem conter informações extras. Por exemplo, a enumeração “velocidade” pode conter os valores [lento, médio, rápido]. Porém, dada uma velocidade, pode-se desejar saber seu valor em quilômetros por hora. Para isso, um item da enumeração pode ser associado a um valor complementar que pode ser usado mais tarde. A figura 10 mostra o exemplo de duas enumerações sendo definidas.

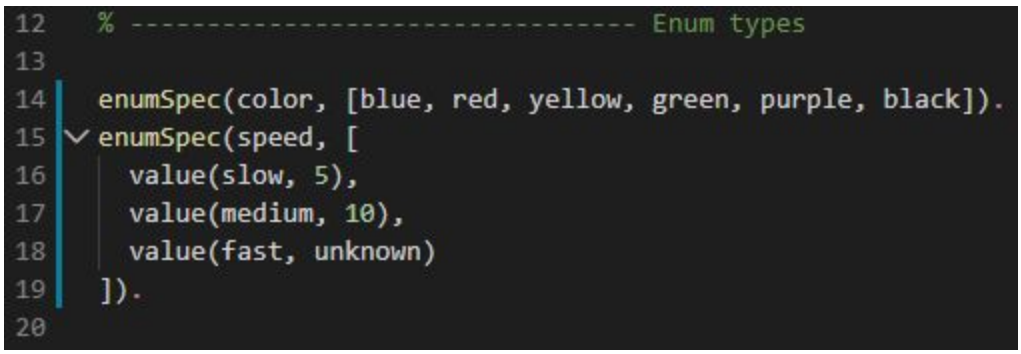

Figura 10 - definição de duas enumerações

A notação para a definição de enumerações é:

- “*enumSpec(nome_da_enumeração, [lista_de_items]).*”

Os itens e nome da enumeração obrigatoriamente devem começar com letra minúscula e conter apenas caracteres alfanuméricos.

Caso se deseje associar um valor complementar a um item da enumeração, pode-se trocar o nome do item na lista por:

- “*value(nome_do_item, valor_complementar)*”

O valor complementar pode ser um número ou outro nome que se inicia com letra minúscula, exceto “nil”, que é usado internamente para indicar que não há complemento.

O próximo passo no arquivo “medievalSpec.pl” é definir os tipos de entidade. Isso também inclui as categorias de entidade. A figura 11 exemplifica essa definição:

```
21  % ---------------------------------- Entity types
22
23  typeSpec(orc).
24
25  categorySpec(man, character).
26  categorySpec(woman, character).
27  categorySpec(prince, man).
28  categorySpec(building, place).
29
```

Figura 11 - definições de categorias de entidades

A notação para a definição de tipos de entidade é:

- "*typeSpec(nome_do_tipo).*"

A notação para a definição de categorias de entidade é:

- "*categorySpec(nome_da_categoria, superclassificação).*"

A superclassificação de uma categoria pode ser outra categoria/ tipo criado nesse arquivo ou um tipo nativo do Gerador. O nomes de tipo e categoria devem começar por letra minúscula e conter apenas caracteres alfanuméricos.

O próximo passo no arquivo "medievalSpec.pl" é definir os tipos de estados. A figura 12 contém um exemplo dessa definição.

```
30  % ------------------------------- State types
31  stateTypeSpec(builtIn, [
32    entityArg(building),
33    entityArg(place)
34  ]).
35  stateTypeSpec(isKnight, [
36    entityArg(character)
37  ]).
38  stateTypeSpec(kidnappedBy, [
39    entityArg(character),
40    entityArg(character)
41  ]).
42  stateTypeSpec(defeatedBy, [
43    entityArg(character),
44    entityArg(character)
45  ]).
46  stateTypeSpec(savedBy, [
47    entityArg(character),
48    entityArg(character)
49  ]).
50
```

Figura 12 - definição de tipos de estados

A notação para a definição de tipos de estado é:

- "*stateTypeSpec(nome_do_tipo, [lista_de_tipos_de_argumentos]).*"

Cada elemento da "lista_de_tipos_de_argumentos" deve ser uma opção dentre as seguintes notações:

- "*entityArg*" - Indica que o argumento é uma entidade.
- "*entityArg(nome_da_classificação)*" - Indica que o argumento é uma entidade que é classificada em "nome_da_classificação" (note que isso vale para todas as classificações da entidade, não apenas para a superclassificação).
- "*scalarArg*" - Indica que o argumento é um escalar (número ou enumeração).
- "*scalarArg(number)*" - Indica que o argumento é um número.
- "*scalarArg(nome_da_enumeração)*" - Indica que o argumento é um item do tipo de enumeração "nome_da_enumeração".
- "*stateArg*" - Indica que o argumento é um estado na notação de assinatura (ex: "*feliz(joana)*").

- “*stateArg(nome_do_tipo_de_estado)*” - Indica que o argumento é um estado do tipo “nome_do_tipo_de_estado” na notação de assinatura (ex: “*nome_do_tipo_de_estado(joana)*”).
- “*eventArg*” - Indica que o argumento é um evento na notação “*event(assinatura_do_evento, tempo_de_ocorrência)*” (ex: “*event(mover(cidade,campo), 5)*”).
- “*eventArg(nome_do_tipo_de_evento)*” - Indica que o argumento é um evento do tipo “nome_do_tipo_de_evento” na notação “*event(assinatua_do_evento,tempo_de_ocorrência)*” (ex: “*event(nome_do_tipo_de_evento(cidade, campo),5)*”).

Os tipos de argumentos de um tipo de estado são importantes, pois são usados pelo algoritmo na hora de processar eventos. Todos os “nomes_de_tipo” devem começar com letra minúscula e conter apenas caracteres alfanuméricos.

O próximo passo no arquivo “medievalSpec.pl” é definir os tipos de gatilhos. A figura 13 mostra um exemplo disso.

```
51  % ------------------------------ Trigger types
52  triggerTypeSpec(villainActs, passive).
53  triggerTypeSpec(heroActs, passive).
54  triggerTypeSpec(defeat, active).
55
```

Figura 13 - definições de tipos de gatilhos

A notação para a definição de tipos de gatilhos é:

- “*triggerTypeSpec(nome_do_tipo, ativo_ou_passivo)*”

A informação “ativo_ou_passivo” define se o gatilho é ativo ou passivo e deve ser uma opção dentre as notações:

- “*passive*” - para um gatilho passivo.
- “*active*” - para um gatilho ativo.

Por último, definem-se os tipos de eventos no arquivo “medievalSpec.pl”. A definição de um tipo de evento tem uma estrutura bem mais complexa, portanto será explicada com base na figura 14.

```
% -------------------------------- Event types
eventTypeSpec(
  kidnap(Kidnapper, Kidnapped, Plc),          % #1 - Assinatura do tipo de evento
  [                                           % #2 - Condições de estados
    standsIn(Kidnapper, Plc),
    standsIn(Kidnapped, Plc),
    userPersonality(_,_,_,_,Neuroticism),
  ],
  [                                           % #3 - Condições de Prolog
    entityClassification(Kidnapper, character),
    entityClassification(Kidnapped, character),
    Kidnapper \== Kidnapped,
    Neuroticism >= 0
  ],
  [villainActs],                              % #4 - Triggers dependentes
  [tick(1)],                                  % #5 - Triggers disparados
  [],                                         % #6 - Estados removidos
  [kidnappedBy(Kidnapped, Kidnapper)]         % #7 - Estados adicionados
).

```

Figura 14 - definição de tipo de evento

Pode-se observar que a notação da definição de um tipo de evento é:

- "*eventTypeSpec(#1,#2,#3,#4,#5,#6,#7).*"

Para entender cada pedaço da notação, é necessário entender abstratamente um pouco do que é uma variável Prolog. Uma variável Prolog tem um nome que começa com uma letra maiúscula e possui apenas caracteres alfanuméricos. Ela representa um valor qualquer. Por exemplo, o estado "feliz(joão)" é um estado que representa que a entidade "joão" está feliz. O estado "feliz(Personagem)" não é um estado que representa nenhuma entidade em específico: pode se referir a "joão", "maria", ou qualquer entidade que exista. A letra maiúscula no início de "Personagem" indica que essa palavra é uma variável, não uma entidade específica.

Uma variável é instanciada quando ela temporariamente assume um valor para que algum teste seja realizado. Se a condição para um evento ocorrer é "feliz(joão)", então basta que esse estado exista para que a condição seja satisfeita. Caso a condição para um evento ocorrer é "feliz(Personagem)", basta que exista algum estado como "feliz(joão)" ou "feliz(maria)". Quando o Gerador de Histórias está testando essa condição, ele temporariamente instancia a variável "Personagem" para "joão" ou "maria", ou seja, a condição "feliz(Personagem)" é temporariamente substituída por "feliz(joão)" ou "feliz(maria)".

Variáveis com nomes iguais são instanciadas simultaneamente. Se as condições para que um evento ocorra são “feliz(Personagem)” e “acordado(Personagem)”, ambas devem ser instanciadas para “maria” ou para “joão” simultaneamente. Isso quer dizer que elas compartilham valores. Então as condições só serão satisfeitas se houver uma pessoa que estiver feliz e acordada. Se apenas os estados “feliz(maria)” e “acordado(joão)” forem verdadeiros, as condições não estarão satisfeitas.

O nome “_” não começa com uma letra maiúscula, mas é uma variável. O underline é uma variável especial, chamado de “variável anônima”. Ele é uma exceção para a regra explicada no parágrafo anterior, ou seja, variáveis com o mesmo nome “_” não são instanciadas simultaneamente. Portanto, se as condições para que um evento ocorra são “feliz(_)” e “acordado(_)”, mesmo que os únicos estados verdadeiros sejam “feliz(maria)” e “acordado(joão)”, as condições serão satisfeitas.

Com base nessas informações, pode-se entender a notação da definição do tipo de evento. O termo #1 é a assinatura do tipo do evento. Trata-se do nome do tipo de evento seguido de uma lista de variáveis Prolog entre parênteses:

- “nome_do_tipo_de_evento(lista_de_variáveis_prolog)”

As variáveis dessa lista são instanciadas quando quando um evento é criado. Isso quer dizer que se o tipo de evento tem assinatura “sequestra(Personagem1, Personagem2, Lugar)”, um evento que tem esse tipo é “sequestra(maria, joão, cidade)”. A escolha de quais variáveis colocar nessa lista fica a critério do usuário, mas na dúvida, ele pode repetir todas as variáveis usadas na definição do evento na assinatura.

O termo #2 são as Condições de Estado do evento. É uma lista contendo os estados que devem ser verdadeiros para que o evento ocorra. Para que a condição seja que um estado seja falso (ou não exista), pode-se escrever o estado precedido do operador “\+”. Por exemplo, se as condições são de que joão deve estar feliz mas não deve estar acordado, a lista fica: “[feliz(joão), \+acordado(joão)]”. No exemplo apresentado, a condição para que um personagem sequestre o outro é que eles estejam no mesmo lugar (a variável “Plc” é igual para ambos). O estado “userPersonality” é um estado nativo. Ele é usado no exemplo na lista de condições para se instanciar a variável “Neuroticism”, que é usada abaixo.

O termo #3 são as Condições de Prolog do evento. É uma lista contendo comandos Prolog (como aritmética e comparação) que devem ser verdadeiros para que o evento execute. A capacidade dessa lista pode ser explorada melhor por usuários avançados. Ela pode ser utilizada para fazer comparações com variáveis instanciadas, usando operadores como “==”,

“\==”, “=<”, “>=”. Nesse exemplo, testa-se a diferença entre as variáveis instanciadas “Sequestrador” e “Sequestrado”, para garantir que o evento não ocorra com personagens iguais (caso contrário um evento como “sequestra(joão, joão, cidade)” poderia ocorrer). Também há dois tipos de notação de condições que podem ser usadas nessa lista:

- “*entityClassification(nome_da_entidade, nome_da_classificação)*” - condição em que a entidade “nome_da_entidade” deve estar classificada em “nome_da_classificação”.
- *“enumValue(nome_da_enumeração, nome_do_item, valor_complementar)*” - em que, na enumeração “nome_da_enumeração”, o item “nome_do_item” tem valor complementar igual a “valor_complementar”. Por exemplo, com a enumeração da figura 10, seria verdadeiro: “*enumValue(speed, slow, 5)*”.

As notações “entityClassification” e “enumValue” não precisam ser usadas como condições, mas podem ser usadas para instanciar variáveis.

O termo #4 é uma lista de gatilhos dos quais o evento depende. Se um dos gatilhos dessa lista é disparado, um evento desse tipo pode ocorrer. Os gatilhos na lista começam por letra minúscula e contém apenas caracteres alfanuméricos.

O termo #5 é uma lista de gatilhos disparados quando o evento executa. Eles estão na seguinte notação:

- “nome_do_gatilho(delta_tempo)”

O “delta_tempo” é quanto tempo depois que o evento executa o gatilho é disparado. Por exemplo, o termo “tick(5)” irá agendar o gatilho “tick” para ser disparado 5 unidades de tempo após o evento ocorrer. Esse valor pode ser igual a zero.

O termo #6 é uma lista de estados que são removidos (ficam falsos) após o evento executar. Toda variável usada aqui deve estar instanciada quando o evento ocorre.

O termo #7 é uma lista de estados que são adicionados (ficam verdadeiros) após o evento executar. Toda variável usada aqui deve estar instanciada quando o evento ocorre.

11.2. Arquivo de Definição de História

Nesse arquivo serão definidas as entidades, os estados da história, disparo fixo de gatilhos, eventos, trama e função heurística. A título de exemplo, cria-se um arquivo para definir uma história do gênero previamente definido (medieval) chamado "storySpec.pl”.

As primeiras coisas a serem definidas neste arquivo são as entidades da história. A figura 15 mostra um exemplo dessa definição.

```
% >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> Story definition

% ---------------------------------- Entities
entitySpec(cassandra, woman).
entitySpec(horace, prince).
entitySpec(capital, place).
entitySpec(forest, place).
entitySpec(palace, building).
entitySpec(morgarath, man).
entitySpec(crown, thing).

```

Figura 15 - definições de entidades

A notação para definição de entidades é, portanto:

- "*entitySpec(nome_da_entidade, superclassificação).*"

Os nomes das entidades e superclassificações começam por letra minúscula e possuem apenas caracteres alfanuméricos.

A próxima etapa no arquivo "storySpec.pl" é definir os estados da história. Esses são os estados da configuração inicial da história. A figura 16 tem um exemplo com isso.

```
% ---------------------------------- States
builtIn(palace, capital).
isKnight(cassandra).
standsIn(horace, palace).
standsIn(cassandra, capital).
standsIn(morgarath, forest).
standsIn(crown, capital).
isHolding(cassandra, crown).

```

Figura 16 - definições de estados

Como se pode observar, não há notação específica para os estados da história. Cada um deles segue o padrão de assinatura do nome e argumentos cadastrados na seção de tipos de estados da história, no arquivo de definição de gênero ("medievalSpec.pl"). Por essa razão, não devem existir nomes de entidades ou eventos ou tipos conflitantes com os nomes dos tipos de estados.

O próximo passo no arquivo "storySpec.pl" é definir os disparos fixos de gatilhos. Eles são agendamentos de disparos, ou seja, os gatilhos são disparados no momento determinado. A definição está exemplificada na figura 17.

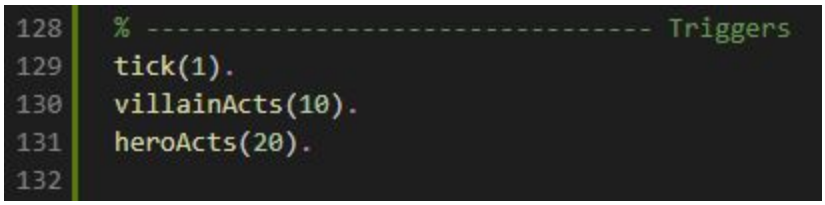

Figura 17 - definições de gatilhos

Assim como os estados da história, os disparos de gatilhos não tem notação fixa, mas dependem dos nomes dos tipos de gatilho. Cada agendamento segue o formato:

- "*nome_do_tipo_do_gatilho(tempo_de_disparo).*"

O próximo passo no arquivo “storySpec.pl” é a definição de eventos. Isso não é muito comum, porque esses eventos devem ter ocorrido antes da história começar e não alteram o estado do universo da história, porque já ocorreram. Essa definição está exemplificada na figura 18.

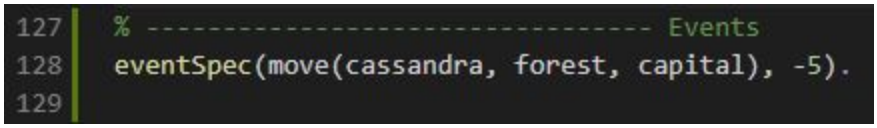

Figura 18 - definição de evento

A notação para definição de eventos é:

- “*eventSpec(assinatura_do_evento, tempo_de_ocorrência).*”

O “tempo_de_ocorrência”, nesse caso, é sempre um valor negativo.

O próximo passo é definir a trama da história. A trama guiará o Gerador com a ordem de ocorrência de eventos através do uso de um algoritmo planejador. Ela é uma sequência de objetivos que devem ser cumpridos na ordem em que são definidos. Um exemplo de definição da trama está na figura 19.

```
133  % ---------------------------------- Plot
134  plotSpec([
135    [
136      isHolding(horace, crown)
137    ],
138    [
139      kidnappedBy(horace, Villain)
140    ],
141    [
142      \+ kidnappedBy(_,_),
143      defeatedBy(Villain, _),
144      savedBy(horace, cassandra)
145    ]
146  ]).
147
```

Figura 19 - definição de trama

A notação para definição da trama é:

- "*plotSpec([lista_de_objetivos]).*"

Cada objetivo é uma lista de estados da história que devem ser verdadeiros (ou o operador "\+" pode ser usado para ele precisar ser falso), assim como no termo #2 da definição de tipos de evento do arquivo "medievalSpec.pl". As variáveis Prolog também podem ser usadas aqui. No exemplo, "*\+kidnappedBy(_,_)*" se traduz em: nenhum estado kidnappedBy deve existir, independentemente dos argumentos.

A trama também tem um detalhe importante: quando um objetivo dela é alcançado, todas as variáveis Prolog instanciadas comprometem-se a manter o valor instanciado. Tomando o segundo objetivo do exemplo: "*kidnappedBy(horace, Villain)*" exige que o personagem "horace" seja sequestrado por qualquer outra pessoa (representado pela variável "Villain"). Se o Gerador de histórias alcançar esse objetivo com "*kidnappedBy(horace, morgarath)*", então "Villain" é permanentemente instanciado para "morgarath". Ao tentar cumprir o próximo objetivo "*defeatedBy(Villain, _)*", o Gerador tentará satisfazer diretamente "*defeatedBy(morgarath, _)*". Na prática, a variável deixa de ser variável e passa a ser um valor fixo.

Por último, deve-se definir no “storySpec.pl” a função heurística usada pelo Processador de Eventos para planejar o andamento da história. Essa definição pode se tornar complexa sem maiores conhecimentos em Prolog, mas há uma receita básica que se pode seguir para

alcançar bons resultados. Um exemplo de implementação um pouco complexo está demonstrado na figura 20.

```
41  % ---------------------------------- Heuristic predicate
42
43  heuristicPredicateSpec(actionExecution(move(_,_,_),_), 10) :-
44    !.
45  heuristicPredicateSpec(actionExecution(carry(_,_,_,_),_), Cost) :-
46    random(R), Cost is R*10, !.
47  heuristicPredicateSpec(actionExecution(_,States), 15) :-
48    member(isHolding(cassandra, crown), States), !.
49  heuristicPredicateSpec(_, 50).
50
```

Figura 20 - definição de função heurística

A função heurística é uma única, mas seu comportamento é determinado utilizando a notação "*heuristicPredicateSpec*" múltiplas vezes. A ordem dessas linhas é importante.

Até agora, todas as definições demonstradas estavam no formato de termo do prolog - um nome em letra minúscula, seguido de informações entre parênteses, seguido de um ponto "." final. Por exemplo: "*entitySpec(cassandra, woman)*". A notação da função heurística pode estar nesse formato, mas também pode existir no formato de cláusula prolog, quando possui o operador ":-" antes do ponto "." final. Por exemplo: "*heuristicPredicateSpec(_,1) :- !.*".

A função heurística associa cada evento da história a um custo. Considere a última linha do exemplo da figura 20: "*heuristicPredicateSpec(_, 50).*". Esse termo associa o custo 50 a qualquer evento da história (por isso a variável anônima "_" é usada). Agora analise o termo "*heuristicPredicateSpec(actionExecution(move(_,_,_),_), 10).*". Ele associa o custo 10 a qualquer evento do tipo "move", ou seja, o evento em que um personagem se move de um lugar para o outro. De maneira geral, a notação para associação do evento a um custo é:

- "*heuristicPredicateSpec(actionExecution(assinatura_do_evento, lista_de_estados_obtidos), custo_do_evento).*"

Se a "assinatura_do_evento" usar variáveis, ela vai se encaixar em diversos eventos. No último exemplo, a assinatura "*move(_,_,_)*" serviu para representar qualquer evento do tipo "move". A assinatura "move(_,_,cidade)" serviria para representar qualquer evento em que uma pessoa se move para a cidade, e assim por diante. A "lista_de_estados_obtidos" é uma lista de prolog que contém todos os estados da história após o evento ser executado. Ela também pode

ser usada para calcular o custo do evento, para usuários avançados que saibam fazê-lo em Prolog. A notação com variáveis anônimas "*heuristicPredicateSpec(actionExecution(_,_), custo_do_evento).*" equivale à notação "*heuristicPredicateSpec(_, custo_do_evento).*". Ela serve para associar um custo fixo a qualquer evento.

Para fazer uma função heurística básica, pode-se seguir os passos a seguir, sem a necessidade de conhecer Prolog:

1. Defina um custo básico para todos os eventos:

   heuristicPredicateSpec(_, 50).
2. Defina uma exceção. Por exemplo, todos os eventos custam 50, exceto os que são do tipo "move", que custam 10. Defina a exceção em uma linha acima do custo básico, adicionando ":- !." ao final do termo. Exemplo:

   heuristicPredicateSpec(actionExecution(move(_,_,_),_), 10) :- !.

   heuristicPredicateSpec(_,50).
3. Se houver mais exceções, volte ao passo 2 e as defina. Deixe sempre as linhas mais específicas em cima, pois são as condições que são testadas primeiro. Isso quer dizer que se um evento não se enquadra na primeira linha "*move(_,_,_)*", ele cai no *fallback* da linha de baixo, que associa o custo 50 (a última linha não precisa de ":- !").

12. Gerador de Histórias

12.1. Considerações

12.1.1. História Coerente

Para que a história gerada seja coerente, é desejável que algumas informações dos estados tenham consistência interna. Se o estado "conhece" tem dois argumentos que são personagens, em algum momento da história, o estado conhece("roberto", "roberto") poderia passar a existir. É claro que "roberto" conhece "roberto". Além de redundante, a informação é confusa e pode atrapalhar o usuário da biblioteca, há diversos tipos de problemas como esse. O próprio usuário pode adicionar dois estados conflitantes - como "dormindo(cláudia) e acordada(cláudia)". Seria possível criar uma maneira de registrar limitações desse tipo na biblioteca, mas grande parte delas ficaria a cargo de criação do usuário. É melhor, portanto, garantir a consistência nos momentos de criar e mudar os estados da história. Fica a cargo do usuário, assim, criar eventos que não criam informações inconsistentes.

Alguns exemplos de pontos que devem ser consistentes:

foiCasadaVezes(mary, 2) → Mary foi casada duas vezes

foiCasadaVezes(mary, 1) → Mary foi casada uma vez

O estado "*foiCasadaVezes*" com "*mary*" no primeiro argumento deve existir no máximo uma vez na base de conhecimento. Para garantir consistência, essa informação só deve ser adicionada se já não existir.

conhece(mary, john) → Mary e John se conhecem

conhece(john, mary) → Mary e John se conhecem

O estado "*conhece*" com Mary e John como dois primeiros argumentos em qualquer ordem deve existir no máximo uma vez na base de Conhecimento. Para garantir consistência, essa informação só deve ser adicionada se já não existir.

A - primeiraViagemFoiFelizNaEscala10(mary, john, praia, parque, 5) → a primeira vez que Mary e John estavam viajando da praia para o parque, eles estavam felizes 5/10

B - primeiraViagemFoiFelizNaEscala10(mary, john, praia, parque, 7) → a primeira vez que Mary e John estavam viajando da praia para o parque, eles estavam felizes 7/10

C - primeiraViagemFoiFelizNaEscala10(john, mary, praia, parque, 9) → a primeira vez que John e Mary estavam viajando da praia para o parque, eles estavam felizes 9/10

D - primeiraViagemFoiFelizNaEscala10(john, mary, parque, praia, 10) → a primeira vez que Mary e John estavam viajando do parque para a praia, eles estavam felizes 10/10

Note que A, B e C são equivalentes e portanto inconsistentes, porque eles têm valores escalares diferentes. D não é equivalente a nenhum deles. Isso acontece porque [john, mary] são comutáveis (mantendo a semântica) e [parque, praia] não são comutáveis na frase.

achaQue(mary, feliz(john)) → Mary acha que John está feliz (independente disso ser verdade).

sabeQue(mary, feliz(john)) → Mary sabe que John está feliz (então isso é verdade).

No caso do predicado “*sabeQue*”, deve-se garantir que ele está atualizado com as alterações de estado.

Uma má prática é guardar informação redundante na base de conhecimento. Por exemplo, dado um modelo de personalidade, pode-se inferir que um personagem é normalmente bravo e criar o estado “*éBravo(personagem)*”. Porém, se essa informação já pode se extraída da personalidade do personagem, criar esse estado só adiciona informação redundante.

12.1.2. Utilização de Gatilhos

Os gatilhos são a força motriz do gerador de histórias. Se nenhum gatilho for disparado, nada vai acontecer. Mesmo que um evento tenha todas as suas condições satisfeitas, ele só irá ocorrer após ser ativado por um gatilho. Considere um vilão que deseja sequestrar uma pessoa. Em vários momentos da história, todas as pré-condições para o evento “sequestro” podem ser satisfeitas. Porém, ele não vai realizar o sequestro até chegar o momento “certo” do seu plano, ou simplesmente até o escritor achar que é o momento certo. Em um jogo, esse momento pode depender da posição da câmera do jogador, de suas ações, de sua localização no jogo e diversos outros fatores. Os gatilhos, dessa maneira, delegam um pouco do controle de *timing* e direcionamento da história para o usuário.

Os gatilhos passivos servem para entregar mais controle sobre a execução de eventos para o Gerador, de forma a cumprir a Trama. Quando disparados, o Gerador decide se um evento vai ocorrer ou não como consequência do disparo. Assim, podem ser disparados de forma regular para deixar o planejamento seguir. O tipo de gatilho nativo “tick” segue essa lógica, portanto os eventos nativos que dependem dele disparem outro “tick” quando são executados. Esse disparo ocorre uma unidade de tempo após a execução do evento, possivelmente criando uma cadeia de execuções em um tempos regulares.

Os gatilhos ativos servem para executar ações semi-forçadas em um tempo específico. O usuário pode utilizá-los para desencadear eventos que devem ocorrer em um momento específico ou fazer um *timer*. Os eventos também podem disparar esses gatilhos, podendo representar ocorrências que ocorrem em um período não nulo de tempo ou causar efeitos colaterais, se disparados no mesmo instante de ocorrência de um evento. Por exemplo, o gatilho ativo nativo “motion” é disparado quando um personagem se move. Se o personagem está segurando um objeto, esse gatilho obrigatoriamente dispara o evento “carrega”, que faz o

personagem carregar o objeto de um lugar para outro com ele, como efeito colateral de ter se movido.

12.2. Processador de Eventos

O Processador de Eventos é o módulo que recebe uma consulta, com o tempo atual, calcula quais eventos ocorreram desde a última consulta e os retorna, atualizando o estado do universo no processo. Ele é implementado em Prolog e unifica os conceitos de consulta, gatilhos e planejador para processar os eventos visando a Trama.

O primeiro detalhe a notar é que o Processador de Eventos nunca considera a Trama completa, mas somente seu próximo objetivo. Imagine uma trama com dois objetivos separados:

- O personagem “A” deve se encontrar com o personagem “B”;
- O personagem “A” deve se encontrar com o personagem “C”.

Pode ser que o Processador de Eventos primeiro faça o “B” andar até a posição do “A”, cumprindo o primeiro objetivo. Depois, ele pode fazer o “C” andar até a posição do “A”, cumprindo o segundo objetivo. Porém, se “B” e “C” estiverem inicialmente no mesmo lugar, é mais eficiente que o personagem “A” ande até a posição deles, cumprindo ambos os objetivos. O Processador de Eventos só sabe processar um objetivo por vez, então ele encontrará essa solução se a Trama for modelada como um único objetivo:

- O personagem “A” deve se encontrar com o personagem “B” e o personagem “A” deve se encontrar com o personagem “C”.

Quando o fim da Trama é alcançado, o Processador de Eventos para de gerar a ocorrência de novos eventos.

Para um evento ocorrer, um gatilho deve ter sido disparado no meio tempo. O Processador verifica quais são todos os eventos que possivelmente podem executar como consequência dele. No caso do gatilho ativo, ele escolhe o evento que levará ao Estado do Universo mais perto do próximo objetivo da trama dentre as opções e o executa. No caso do gatilho passivo, ele faz o mesmo, mas o evento só é executado caso o novo Estado do Universo seja mais perto do próximo objetivo da trama do que o Estado do Universo atual. O módulo planejador com a função heurística definida pelo usuário é utilizado para o cálculo destas distâncias.

Existe um detalhe importante do processador de eventos: independente do tipo de gatilho que dispara um evento, o evento só é executado caso exista uma maneira de cumprir o próximo Objetivo da Trama a partir do novo Estado do Universo. Ou seja, o Processador de Eventos jamais vai deixar um evento como “morte” do personagem principal ocorrer se isso impedir a Trama de ser realizada - mesmo que o trigger disparado seja ativo.

Por causa do funcionamento do Processador de Eventos, na hora de definir o gênero e as especificações da história deve-se ter cuidado ao definir os Tipos de Eventos e a Trama. Se a Trama tiver objetivos difíceis ou impossíveis de serem alcançados, o Processador de Eventos pode ficar preso, sem escolhas. Isso também pode acontecer se os Tipos de Eventos não forem restritivos o suficiente, ou seja, se não forem implementadas restrições semânticas de coerência como “um objeto não pode se mover sozinho” ou “um personagem não pode entregar um objeto para si mesmo”.

12.3. Definições Nativas

Alguns elementos do Gerador de História que podem ser definidos pelo usuário já possuem valores nativos. Esta seção enumera todos eles.

- Entidades nativas:

```
typeSpec(thing).
typeSpec(character).
typeSpec(place).
typeSpec(animal).
```

- Tipos de estados nativos:

```
stateTypeSpec(standsIn, [
  entityArg(character),
  entityArg(place)
]).
stateTypeSpec(standsIn, [
  entityArg(thing),
  entityArg(place)
]).
stateTypeSpec(distanceInKilometers, [
  entityArg(place),
  entityArg(place),
  scalarArg(number)
```

```
]).
stateTypeSpec(isHolding, [
  entityArg(character),
  entityArg(thing)
]).
stateTypeSpec(knowsPerson, [
  entityArg(character),
  entityArg(character)
]).
stateTypeSpec(knowsWhere, [
  entityArg(character),
  entityArg(place)
]).
stateTypeSpec(knowsThat, [
  entityArg(character),
  stateArg
]).
stateTypeSpec(knowsThat, [
  entityArg(character),
  eventArg
]).
stateTypeSpec(thinksThat, [
  entityArg(character),
  stateArg
]).
stateTypeSpec(personalityOf, [
  entityArg(character),
  scalarArg(number),
  scalarArg(number),
  scalarArg(number),
  scalarArg(number),
  scalarArg(number)
]).
stateTypeSpec(relationshipBetween, [
  entityArg(character),
  entityArg(character),
  scalarArg(number),
  scalarArg(number),
  scalarArg(number),
  scalarArg(number),
  scalarArg(number)
]).
stateTypeSpec(userPersonality, [
  scalarArg(number),
```

```
  scalarArg(number),
  scalarArg(number),
  scalarArg(number),
  scalarArg(number)
]).
```

- Tipos de gatilhos nativos:

```
triggerTypeSpec(tick, passive).
triggerTypeSpec(motion, active).
```

- Tipos de eventos nativos:

```
eventTypeSpec(
  move(Char, Plc1, Plc2),
  [standsIn(Char, Plc1)],
  [entityClassification(Char, character), Plc1 \== Plc2],
  [tick],
  [tick(1), motion(0)],
  [standsIn(Char, Plc1)],
  [standsIn(Char, Plc2)]
).
eventTypeSpec(
  carry(Char, Thing, Plc1, Plc2),
  [standsIn(Char, Plc2), standsIn(Thing, Plc1), isHolding(Char, Thing)],
  [entityClassification(Char, character), entityClassification(Thing, thing),
Plc1 \== Plc2],
  [motion],
  [motion(0)],
  [standsIn(Thing, Plc1)],
  [standsIn(Thing, Plc2)]
).
eventTypeSpec(
  give(Char1, Char2, Thing, Plc),
  [standsIn(Char1, Plc), standsIn(Char2, Plc), standsIn(Thing, Plc),
isHolding(Char1, Thing)],
  [entityClassification(Char1, character), entityClassification(Char2,
character), entityClassification(Thing, thing), Char1 \== Char2],
  [tick],
  [tick(1)],
  [isHolding(Char1, Thing)],
  [isHolding(Char2, Thing)]
).
```

12.4. Arquitetura

A figura 21 demonstra o modelo de componentes das dependências dos módulos do Prolog. O Banco de Conhecimento Interno é representado pelos módulos do “Story Generator”, com exceção do “eventProcessor.pl”. O Estado do Universo da história é representado pelo módulo “state.pl”. O módulo “index.pl” exporta vários predicados Prolog de todo o pacote, para integrá-lo com o C#. Como o Prolog é uma linguagem declarativa de lógica de primeira ordem e não possui funções, métodos ou variáveis tradicionais de linguagens imperativas, não foi montado um diagrama UML com seus módulos.

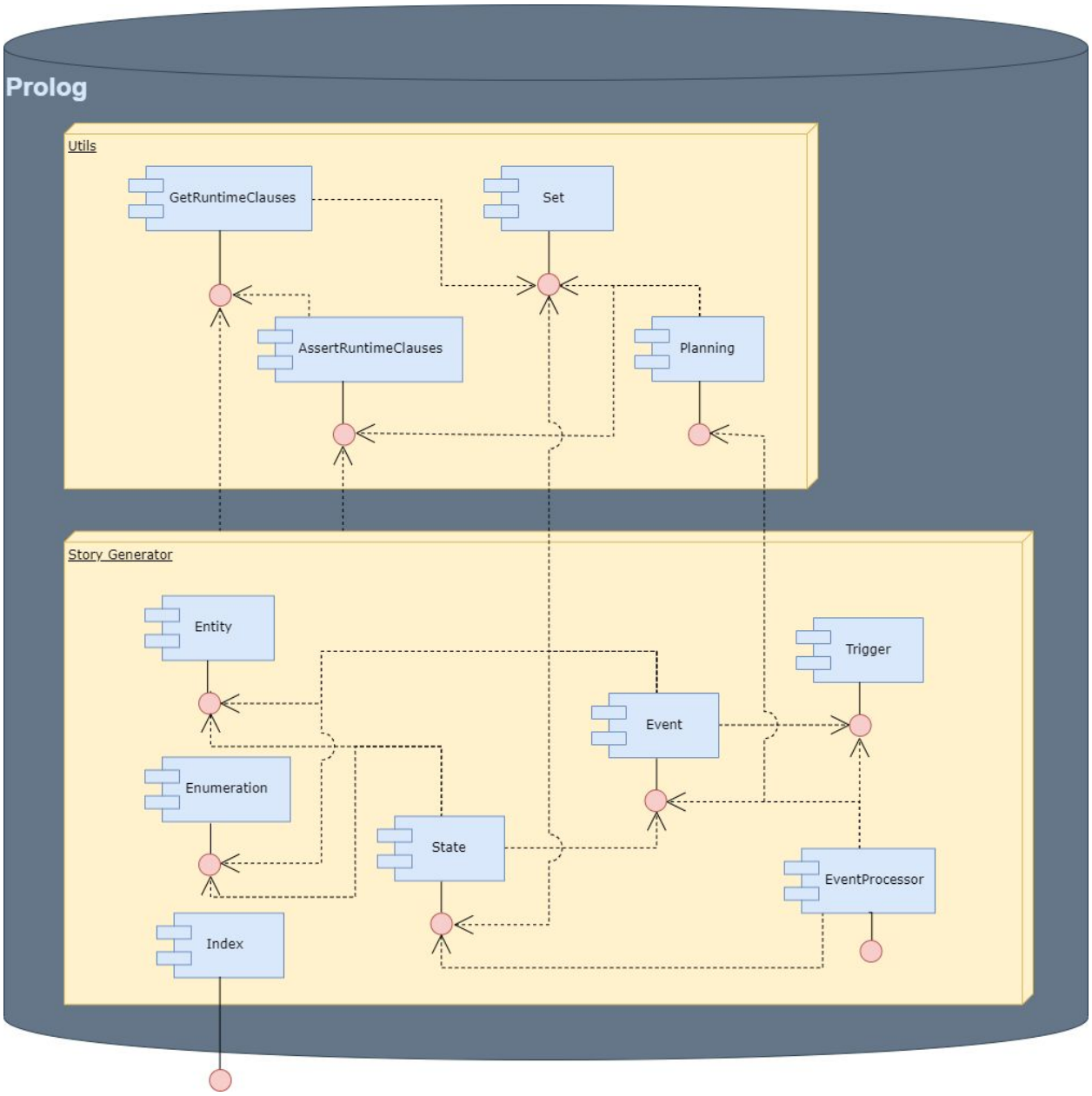

Figura 21 - diagrama de componentes do Gerador de Histórias em Prolog

12.5. Interface em C#

A interface em C# do Gerador de Histórias é basicamente um conjunto de classes imutáveis que representam os dados retornados pelo Gerador: Eventos, Gatilhos (Triggers), Estados, Entidades e Escalares. A biblioteca disponibiliza uma classe principal no padrão "*Singleton*": "*Generator*". O *Generator* faz a conexão com o Prolog com a bilblioteca SWIPICs e se comunica com a Base de Conhecimento através de *queries*. Ele age com um *proxy* para

funcionalidades do Gerador de Histórias em Prolog: disponibiliza um método para consulta, que recebe uma lista de Gatilhos, um tempo atual e retorna uma lista de Eventos ocorridos. Disponibiliza uma propriedade que retorna o Estado do Universo e outra que permite alterar o valor de personalidade do jogador. Esta última é uma tupla de 5 números representando seu modelo BigFive. Ela se traduz no Estado da História “userPersonality”, que pode ser usado como condição em eventos.

Também é disponibilizado um método que permite forçar a execução de um evento no Gerador. Isso é útil para quando um acontecimento do jogo ocorre independentemente do Gerador de Histórias e altera o Estado do Universo do jogo. Normalmente os acontecimentos do jogo disparam triggers para que o Gerador processe um evento resultante, mas pode-se inverter o controle obrigando o Gerador a processar um evento. Se o jogador conversa com um NPC (*Non Playable Character* - personagem não jogável), pode-se forçar o Gerador a executar o evento em que os dois personagens se conhecem. Deve-se ter um cuidado - o evento é forçado mesmo se suas condições não forem satisfeitas. Isso é feito entre consultas e o evento forçado ocorrerá no momento de tempo em que ocorreu a última consulta. Essa funcionalidade não deve ser abusada, pois isso tiraria o propósito da biblioteca de Gerador de Histórias, já que ela perderia o controle da história.

Existe um pequeno *mismatch* de dados entre o Prolog e o C# que ocorre se o usuário tiver definido uma entidade com o mesmo nome de um estado ou um estado com o mesmo nome de um evento e assim por diante. O C# pode interpretar e converter o tipo de forma errônea. O melhor é evitar o conflito de nomes de elementos.

As figuras 22 e 23 são respectivamente um diagrama de componentes e um diagrama de classes UML do Gerador de Histórias em C#.

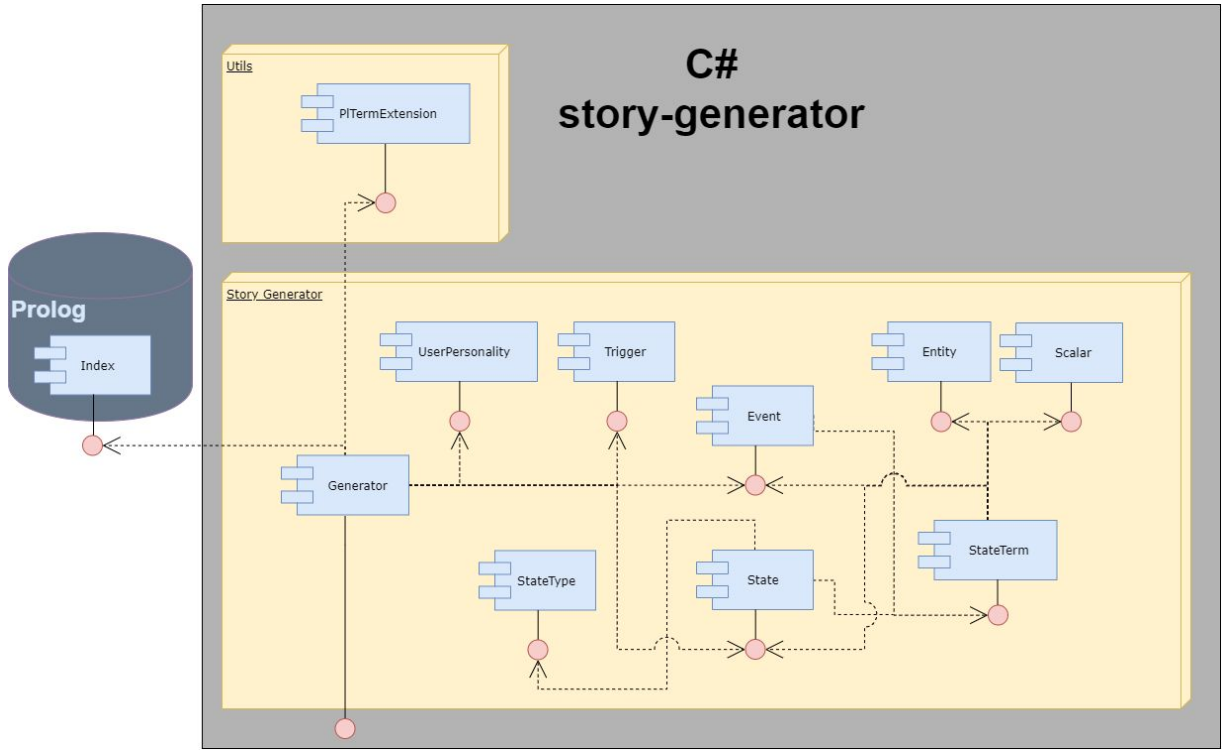

Figura 22 - diagrama de componentes do Gerador de Histórias em C#

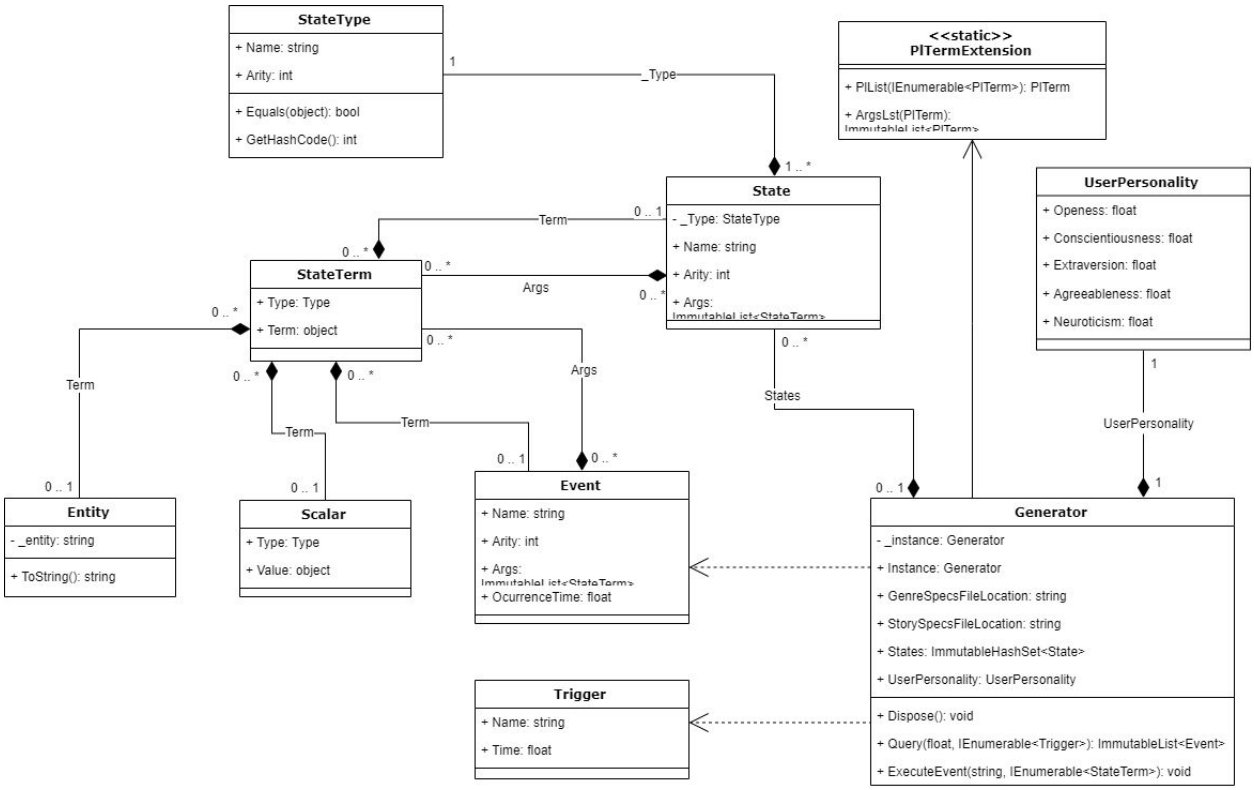

Figura 23 - diagrama de classes UML do Gerador de Histórias em C#

13. Testes Manuais do Gerador

O projeto “generator-test” foi utilizado para testes manuais do gerador de histórias. Nele, definiu-se um arquivo de especificação de gênero “medievalSpec.pl” e um arquivo de especificação de história “storySpec.pl”. Um diagrama de componentes simples é mostrado na figura 24.

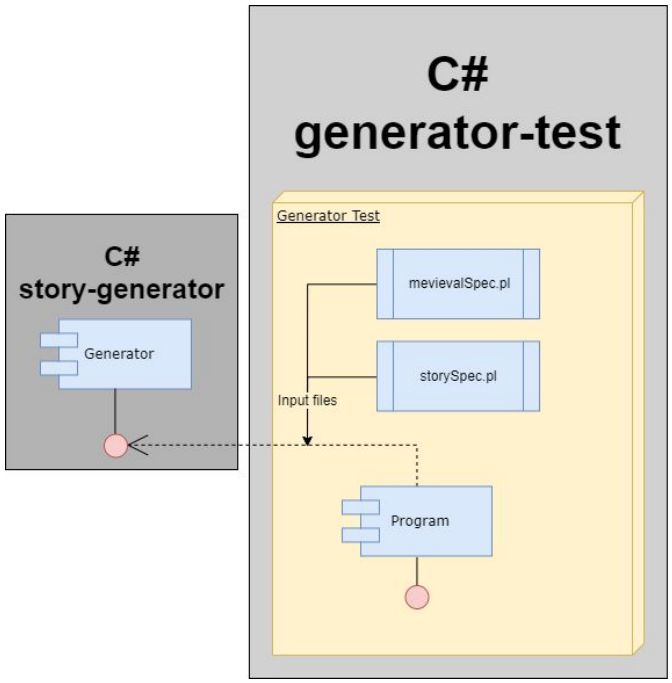

Figura 24 - diagrama de componentes do Teste do Gerador

A título de exemplo, especificou-se uma história simples, com função heurística de custo igual para todos os eventos. Três personagens foram definidos: “horace”, “morgarath” e “cassandra”. No início, cada um começa em uma localização: “horace” começa no “palace”, “morgarath” começa na “forest” e “cassandra” começa na “capital”. “cassandra” começa segurando uma “crown” (coroa). A Trama definida está na figura 25.

```
133  % ------------------------------ Plot
134  plotSpec([
135    [
136      isHolding(horace, crown)
137    ],
138    [
139      kidnappedBy(horace, Villain)
140    ],
141    [
142      \+ kidnappedBy(_,_),
143      defeatedBy(Villain, _),
144      savedBy(horace, cassandra)
145    ]
146  ]).
147
```

Figura 25 - definição da trama

Primeiro, testou-se a propriedade “UserPersonality” e uma Consulta com um trigger “tick(1)”. Os resultados estão na figura 26, com os eventos retornados impressos no Console.

```
User personality Inputed:
Openess: 0,7
Conscientiousness: 0,5
Extraversion: 0,1
Agreeableness: 0,8
Neuroticism: -0,1

tick(1) <----------------------------------------------------------
Name: move/ Arity: 3/ OcurrenceTime: 1
Args: (horace , palace , capital , )

Name: give/ Arity: 4/ OcurrenceTime: 2
Args: (cassandra , horace , crown , capital , )
```

Figura 26 - eventos executados para alcançar o primeiro objetivo da Trama

Pode-se observar que a personalidade do usuário foi lida corretamente e que o primeiro objetivo da Trama foi cumprido através do disparo de triggers passivos “tick”: Horace se moveu até a posição de Cassandra e Cassandra deu a coroa para ele.

Em seguida, outra consulta foi realizada, disparando o trigger “villainActs(5)”. A figura 27 mostra os eventos executados até satisfazer o segundo objetivo da Trama. Horace se move para a floresta (carregando a coroa junto) e é sequestrado por morgarath.

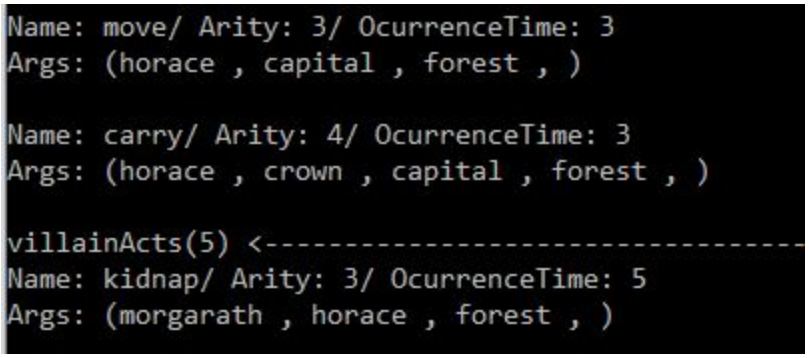

Figura 27 - eventos executados para alcançar o segundo objetivo da Trama

Por fim, o último objetivo da Trama exige que ninguém esteja sequestrado por ninguém, que o vilão Morgarath tenha sido derrotado por alguém e que Horace tenha sido salvo por Cassandra. Com o disparo do trigger "*heroActs(10)*", a sequência de eventos é retornada para satisfazer o último objetivo, como demonstrado na figura 28.

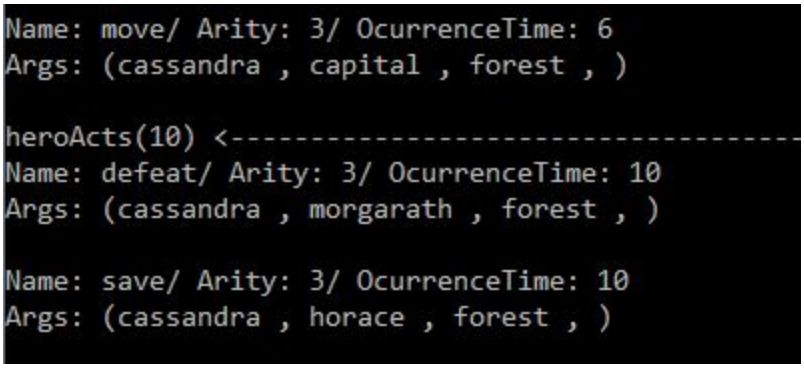

Figura 28 - eventos executados para alcançar o último objetivo da Trama

Também foi testada a propriedade "*States*" do "*Generator*". Após a execução de todos os eventos que satisfizeram a Trama, imprimiu-se o Estado do Universo da História, mostrado na figura 29.

```
States  <----------------------------------------------------------------
Name: standsIn/ Arity: 2
Args: (cassandra , forest , )

Name: standsIn/ Arity: 2
Args: (horace , forest , )

Name: builtIn/ Arity: 2
Args: (palace , capital , )

Name: standsIn/ Arity: 2
Args: (crown , forest , )

Name: defeatedBy/ Arity: 2
Args: (morgarath , cassandra , )

Name: isKnight/ Arity: 1
Args: (cassandra , )

Name: savedBy/ Arity: 2
Args: (horace , cassandra , )

Name: standsIn/ Arity: 2
Args: (morgarath , forest , )

Name: isHolding/ Arity: 2
Args: (horace , crown , )

Name: userPersonality/ Arity: 5
Args: (0,7 , 0,5 , 0,1 , 0,8 , -0,1 , )
```

Figura 29 - estado do universo da história

Pode-se notar que os estados de fato satisfizeram o último objetivo da Trama. Para um último teste manual, chamou-se a função “ExecuteEvent” para forçar um evento “move(cassandra, forest, palace)” no Gerador de Histórias. O Estado do Universo da História foi impresso no Console novamente (figura 30) e os testes se encerraram com sucesso (“cassandra” de fato mudou de localização).

```
States after forced "move(cassandra, forest, palace)" <--------------------
Name: isKnight/ Arity: 1
Args: (cassandra , )

Name: userPersonality/ Arity: 5
Args: (0,7 , 0,5 , 0,1 , 0,8 , -0,1 , )

Name: standsIn/ Arity: 2
Args: (morgarath , forest , )

Name: standsIn/ Arity: 2
Args: (horace , forest , )

Name: standsIn/ Arity: 2
Args: (cassandra , palace , )

Name: builtIn/ Arity: 2
Args: (palace , capital , )

Name: defeatedBy/ Arity: 2
Args: (morgarath , cassandra , )

Name: isHolding/ Arity: 2
Args: (horace , crown , )

Name: standsIn/ Arity: 2
Args: (crown , forest , )

Name: savedBy/ Arity: 2
Args: (horace , cassandra , )
```

Figura 30 - estado do universo da história após evento forçado

14. Considerações técnicas e dificuldades

14.1. Implementação

O SWI-Prolog oferece nativamente predicados para utilização de conjuntos. Porém eles não eram ideais para a implementação do planejador desejada, portanto um módulo sobre a estrutura de dados conjunto (“set.pl”) foi implementado.

A biblioteca SWIPlCs executa um processo único do SWI-Prolog. Portanto, para não haver nenhuma preocupação em trabalhar com múltiplas threads, a classe “Generator” foi implementada no padrão “Singleton” e o código em Prolog, com uma única Base de

Conhecimento. Isso quer dizer que a biblioteca consegue apenas fazer a geração de uma história por vez, sem conseguir fazê-lo simultaneamente.

14.2. Documentação e Testes

Em Prolog, programar de forma a ser mais flexível ou mais performático é um constante dilema. Em geral, garantir a flexibilidade foi importante para o funcionamento correto dos módulos mais complexos, como o planejador. Para garantir que a performance dos algoritmos não inviabilizasse o projeto, o código foi documentado seguindo o padrão do pacote “pldoc” (FRUEHWIRTH, 2012) e criaram-se testes de unidade com o pacote “plunit” (FRUEHWIRTH, 2012) baseados nessa documentação. Apenas os métodos públicos foram testados, conforme recomendado por SCIAMANNA (2016). A figura 31 mostra parte da documentação exportada para html.

Figura 31 - documentação do módulo “set.pl”

14.3. Normalização de Erros e Robustez

Devido à integração entre as linguagens C# e Prolog, foi muito difícil padronizar erros. Para garantir a robustez, todo input deveria ser testado, incluindo durante a comunicação entre as linguagens e a validação de tipos de entidades, eventos, etc. Por falta de tempo e

abrangência do problema, esses tipos de teste não foram implementados em toda interface de troca de dados.

Felizmente, as próprias mensagens de erro do Prolog são explicativas e a biblioteca SWIPICs redireciona-as para o ”*standard output*”. Assim, todos os erros de input do usuário da biblioteca são impressos para ele no *Console*.

15. Considerações Finais

Os modelos de dados traduzidos de Prolog para C# ficaram interessantes, podendo ser usados para realizar buscas e filtragens - espelhando o Estado do Universo da História para o Estado do Universo do jogo.

Algumas partes do código em Prolog teriam ficado mais simples em C#, como o algoritmo A* usado no planejador. Porém, a natureza declarativa do Prolog simplificou muito o modelo de dados usado no planejamento, já que a notação é parecida com a utilizada no PDDL. Dividir essa parte do código nas duas linguagens não seria uma ideia muito boa porque ocorreriam muitas chamadas entre as linguagens, o que tem um custo.

O projeto não é bem escalável do jeito que foi finalizado, pois descobriu-se durante a implementação que o algoritmo do planejador tem dificuldades de rodar com muitos objetos e ações definidos em seu domínio, causando *overflow* na pilha de execução do Prolog, mesmo na versão 64 bits. Isso ocorreu mesmo com a grande preocupação em performance e flexibilidade no módulo mais básico “set.pl”, usado pelo planejador. Os testes criados para esse módulo funcionaram muito bem como testes de regressão para otimizações realizadas no código. Se o projeto fosse refeito do início, ainda mais testes de viabilidade de implementação seriam realizados e se teria ainda mais foco em buscar alternativas para os predicados de conjunto (“set.pl”) que usassem algoritmos de hash. De qualquer jeito, percebeu-se que parte do problema de escalabilidade é inerente ao problema, e outras soluções poderiam ser buscadas - talvez fazendo ainda mais uso de funções heurísticas.

16. Referências Bibliográficas

BARTLE, R. Hearts, clubs, diamonds, spades: Players who suit MUDs. 1996. 27f.

BECKER, K. Artificial Intelligence Planning with STRIPS, A Gentle Introduction. 2015. Disponível em <http://www.primaryobjects.com/2015/11/06/artificial-intelligence-planning-with-strips-a-gentle-introduction/>. Acesso em 14 nov. 2019.

BLACKBURN, P; BOS, J; STRIEGNITZ, K. Learn Prolog Now!. 2006. Disponível em: <http://www.learnprolognow.org/lpnpage.php?pageid=online>. Acesso em: 27 out. 2019.

CAMPBELL, J. The Hero with a Thousand Faces. Commemorative edition. New Jersey: Princeton University Press, 2004. 497 p.

CHAITIN, J. Narratives and Story-Telling. Beyond Intractability. 2003. Disponível em: <https://www.beyondintractability.org/essay/narratives>. Acesso em: 14 abr. 2019.

FERGUSON, D; LIKHACHEV, M; STENTZ, A. A Guide to Heuristic-based Path Planning. School of Computer Science, Carnegie Mellon University, Pittsburgh. 2005. 10f.

FIKES, R. E; NILSSON, N, J. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. In: IJCAI, 2., 1971, London. *Anais…* California: North-Holland Publishing Company, 1971, p. 189-208

FRUEHWIRTH, T; KONINCK, L; WIELEMAKER, J. Reference Manual. SWI-Prolog. 2012. Disponível em: <https://www.swi-prolog.org/pldoc/doc_for?object=manual>. Acesso em: 27 out. 2019.

GERLACH, M.; FARB, B.; REVELLE, W.; AMARAL, L. A. N. A robust data-driven approach identifies four personality types across four large data sets. 2018. 11f.

GHALLAB, M; NAU, D; TRAVERSO, P. Automated Planning and Acting. Authors' manuscript. Cambridge University Press, 2016. 472 p.

GHALLAB, M. et al. PDDL - The Planning Domain Definition Language. Yale Center for Computational Vision and Control. 1998. 27f.

LIMA, E. S.; FEIJÓ, B.; FURTADO, A. L. Player behavior and personality modeling for interactive storytelling in games. Entertainment Computing, Vol. 28, 2018. p. 32-48, 16f.

NEO PI-R - Inventário de personalidade. Psi. 2010. Disponível em: <www.psitestes.com.br/teste-neo-pi-r-neo-ffir-27.html>. Acesso em: 14 abr. 2019.

RIBEIRO, D. Significado de Narrativa. Dicio - Dicionário online de Português. 2018. Disponível em: <www.dicio.com.br/narrativa/>. Acesso em: 14 abr. 2019.

ROE, C. Storytelling 101: The 6 Elements Behind Every Complete Narrative. POND5 BLOG. 2016. Disponível em: <https://blog.pond5.com/6477-storytelling-101-the-6-elements-of-every-complete-narrative/>. Acesso em: 14 abr. 2019.

SCIAMANNA, A. SHOULD PRIVATE METHODS BE TESTED?. ANTHONY SCIAMANNA. 2016. Disponível em: <https://anthonysciamanna.com/2016/02/14/should-private-methods-be-tested.html>. Acesso em 9 nov. 2019.