Universidade de São Paulo
Instituto de Matemática e Estatística
Bacharelado em Ciência da Computação

Rule-Based Godot

um arcabouço de Sistemas Baseados em Regra para um motor de jogos

Rodrigo Volpe Battistin

Monografia Final

mac 499 — Trabalho de Formatura Supervisionado

Supervisor: Dr. Wilson Kazuo Mizutani

São Paulo
2023

Resumo

Rodrigo Volpe Battistin. ***Rule-Based Godot*: *um arcabouço de Sistemas Baseados em Regra para um motor de jogos***. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2023.

Dentre inúmeras aplicações de Inteligência Artificial, a sua presença em jogos eletrônicos é uma das mais perceptíveis para o público geral. As personagens automatizadas são responsáveis por dar vida a uma grande variedade de *videogames*, demonstrando como é relevante a integração de IA com o desenvolvimento de jogos. Neste trabalho foi criado um *plugin* para o motor de jogos *Godot* que funciona como um arcabouço *software* livre de Sistemas Baseados em Regras. O principal objetivo do projeto é fornecer uma ferramenta que facilite a implementação de comportamentos automatizados para personagens em jogos eletrônicos. Outra meta é permitir que pessoas sem muita experiência em programação consigam definir comportamentos interessantes. Para isso, primeiro foi feito um estudo de material bibliográfico sobre essa técnica de Inteligência Artificial. Em adição, fizemos um aprofundamento na tecnologia do motor de jogos escolhido e uma pesquisa por jogos que utilizam sistemas desse tipo. A seguir, foi definido um processo de desenvolvimento inspirado em metodologias ágeis, com iterações de duas semanas e entregas parciais. Ao final, foi produzido um *framework* que disponibiliza: a estrutura completa de execução de regras, um subsistema de pareamento de variáveis, interfaces gráfica e textual e vários módulos prontos e reutilizáveis. O complemento foi feito para a *Godot Engine 4.1+* e procura se integrar com o ambiente de desenvolvimento da maneira menos aparente e invasiva possível. Além disso, ele foi concebido para ser facilmente extensível, permitindo a adição de novos recursos e funcionalidades que se moldem a usos específicos. Em suma, desenvolvedores de jogos podem usar o *plugin* criado para definir comportamentos semelhantes aos encontrados nos jogos de referência. Ademais, a aplicação demonstra um uso expressivo de um método de IA incomum na atualidade, mas que costumava ser amplamente utilizado como um tipo de sistema especialista.

Palavras-chave: Sistemas Baseados em Regras. Motor de jogos. Inteligência Artificial. Jogos digitais. Padrões de projeto. *Godot Engine*.

Abstract

Rodrigo Volpe Battistin. **Rule-Based Godot: *a Rule-Based System framework for a game engine***. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2023.

Amongst Artificial Intelligence's several applications, one of the most perceptible by the public is in video games. Automated characters are responsible for giving life to many types of games, demonstrating the relevance of integrating AI with game development. In this work, we report the creation of a plugin for Godot game engine that functions as a Rule-Based System framework. The main objective of this project is to provide a tool that facilitates the implementation of automated behaviour for characters in video games. Another goal is to allow people with little programming experience to define compelling behaviours. For these purposes, the first step was studying bibliographical material on the Artificial Intelligence technique. In addition, we have diven into the game engine's technology and sought out games that use this type of system. Then, a development process inspired by agile methodology was established, with two-week iterations and partial releases. Another relevant factor was the adoption of free software principles, such as providing the source code and using a software license. In the end, the produced framework contains the complete rule-executing structure, a subsystem for handling variables, graphical and textual interfaces, and several reusable modules. The addon is compatible with Godot Engine 4.1+ and tries to integrate itself into the development environment in the least apparent and intrusive way possible. Furthermore, the system's design makes it very extensible, allowing the addition of new resources and features meant for specific uses. In summary, game developers can use the plugin to define similar behaviours to those found in the reference games. Moreover, the application demonstrates an expressive use of an AI method that is uncommon nowadays but used to be the base of specialist systems.

Keywords: Rule-Based Systems. Game engines. Artificial Intelligence. Digital games. Design patterns. Godot Engine.

Lista de abreviaturas

IA	Inteligência Artificial
SBR	Sistema(s) Baseado(s) em Conhecimento
ASP	*Answer Set Programming*
LRU	*Least Recently Used*
FOSS	*Free and Open-Source Software*
JSON	*JavaScript Object Notation*
POO	Programação Orientada a Objetos
UML	*Unified Modeling Language*
MVC	*Model-View-Controller*
GUI	*Graphical User Interface*
UX	*User experience*
API	*Application Programming Interface*
TUI	*Text-based User Interface*
DAG	*Directed Acyclic Graph*

Lista de figuras

Lista de tabelas

Lista de programas

Sumário

Introdução

Dentre as inúmeras aplicações de Inteligência Artificial (IA) como uma ferramenta, inserida em sistemas de *software* de propósitos variados, o seu uso em jogos eletrônicos foi provavelmente uma das mais perceptíveis para o público leigo por muitas décadas. Antes do advento de *chatbots*[5] que auxiliam na geração de texto e muitas outras tarefas, as pessoas já consumiam *videogames* com entidades que atuam sozinhas. Além disso, as comunidades de jogos conhecem e estudam a inteligência por trás dessa automação desde que ela surgiu.

Nesse contexto, a IA é mais comumente usada para a criação de personagens que agem sem o controle de um jogador (chamadas de *bots*[6], no linguajar da área). Elas devem se movimentar, tomar decisões e executar ações de forma autônoma, transmitindo a impressão de que são seres reais dentro do mundo virtual do jogo. Esses agentes passaram por um grande avanço ao longo dos anos, desde os *goombas* que apenas andam de um lado ao outro em *Super Mario Bros.* (*Nintendo*, 1985), até *bots* que competem como jogadores profissionais de *Valorant* (*Riot Games*, 2020).

O papel dessas personagens é proporcionar interações entre a(s) pessoa(s) jogando o jogo e o ambiente criado pelo computador, podendo trazer desafios, com a intenção de aumentar a dificuldade da experiência, ou auxílios, que buscam diminuí-la. Vemos que a primeira opção é a mais comum, o que é refletido na ampla utilização de IA para controlar inimigos do jogador, em uma grande variedade de jogos. Porém, a mesma tecnologia pode ser facilmente aplicada para automatizar os aliados, como será visto nos jogos de referência.

Tendo isso, neste trabalho será desenvolvido um arcabouço (*framework*) que auxilia na implementação de Sistemas Baseados em Regras (doravante **SBR**) em um motor de jogos. O principal propósito é facilitar o trabalho de desenvolvedores que desejam utilizar esse modelo de Inteligência Artificial em seus projetos, o que será feito por meio do fornecimento de uma estrutura básica de execução, além de alguns blocos de construção reutilizáveis e uma sintaxe expressiva para codificar comportamentos.

Mesmo não sendo uma técnica comum em desenvolvimento de jogos hoje em dia, exemplos de uso nos levam a crer que ela pode ser muito prática para desenvolvedores, trazendo vantagens na montagem e expansão de comportamentos. Dessa forma, aspiramos

[5] Saiba mais sobre *chatbots* em: https://www.oracle.com/br/chatbots/what-is-a-chatbot/

[6] Para saber mais sobre a definição de *bot* no contexto da indústria de jogos, visite: https://www.techopedia.com/definition/10459/bot-software-robot

demonstrar que há benefícios em explorar essa tecnologia e, mais ainda, em torná-la acessível.

Motivação

Quando procuramos aplicações de Sistemas Baseados em Regras na indústria de jogos, percebemos que eles não são uma ferramenta muito comum de desenvolvimento. Um dos maiores motivos desse desuso é sua má reputação como sistemas ineficientes e de difícil implementação, além de estarem um tanto ultrapassados, visto que são usados há quase 30 anos nessa área (Millington e Funge, 2009).

Mesmo com esses pontos contrários, ao se estudar esse tipo de sistema é possível perceber que o que ele perde em complexidade de implantação, ele ganha em poder de expressão e facilidade de uso, por meio da sua abordagem declarativa (Bourg e Seemann, 2004). O uso de regras com uma estrutura simples de "se-então" é, ao mesmo tempo, natural ao raciocínio humano e prático para a execução de uma máquina, trazendo vantagens com relação a outros modelos de Inteligência Artificial usados em jogos eletrônicos, que necessitam de uma abstração muito maior.

Além disso, encontramos casos interessantes nos quais esses sistemas são utilizados para dar ao jogador a liberdade de programar o comportamento de suas personagens, quando esses não são controlados por ele. Esse tipo de uso nos revela que a declaração de regras é considerada intuitiva o suficiente para ser deixada nas mãos de um público que, de maneira geral, não possui experiência em programação. Vale apontar que a interface gráfica fornecida tem grande papel nisso, pois se ela não existisse, esses usuários provavelmente teriam dificuldades em interagir com o sistema.

Diante disso, temos indícios de que a dificuldade de implementação não é transferida para a usabilidade desses sistemas, já que manipular as declarações é complexo no nível de desenvolvimento, mas apenas as descrever é considerado fácil para o usuário. Ademais, é possível concluir que se um jogador comum consegue montar um comportamento interessante para seu aliado autônomo, os *game designers* especializados nessa área conseguiriam usar o mesmo sistema para construir a inteligência de muitas outras personagens do jogo.

Um exemplo desse tipo de aplicação está na série de jogos *Dragon Age*, especificamente em *Dragon Age: Origins* (*BioWare*, 2009) e *Dragon Age II* (*BioWare*, 2011), nos quais há o menu de "Táticas", que pode ser visualizado na Figura 1. Outro caso de uso é o sistema de *gambits* em *Final Fantasy XII* (*Square Enix*, 2006) e em seu *remake Final Fantasy XII: The Zodiac Age* (*Square Enix*, 2017), o qual pode ser visto no menu da Figura 2.

Em ambos os jogos, são utilizadas regras formadas por uma condição à esquerda e a ação a ser tomada à direita, sendo que as condições são relacionadas à própria personagem, aos outros aliados, ou aos inimigos e as ações geralmente representam habilidades da

[7] Disponível em: https://dragonage.fandom.com/wiki/Tactics_(Dragon_Age_II)
Acesso em: 21 abr. 2023

[8] Disponível em: https://www.nintendoblast.com.br/2019/05/analise-final-fantasy-xii-zodiac-age.html
Acesso em: 21 abr. 2023

Figura 1: *Menu de Táticas em Dragon Age II. Fonte: Dragon Age Wiki*[7].

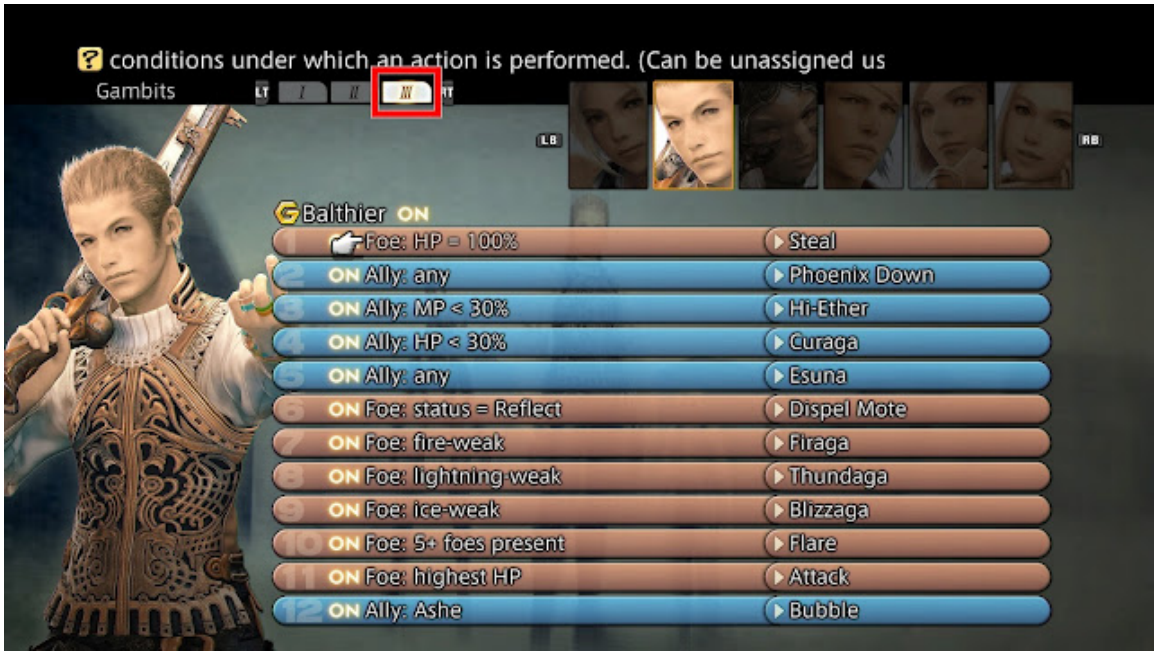

Figura 2: *Menu de Gambits em Final Fantasy XII: The Zodiac Age. Fonte: Vinícius Rutes em Nintendo Blast*[8].

personagem, ou o uso de itens. É possível perceber a modularidade que esse tipo de declaração de regras proporciona, já que cada uma delas lida apenas com uma situação

específica e, portanto, podem ser concebidas gradativamente.

Também vale apontar que o jogador possui controle sob quais regras devem dominar o comportamento, o que é expresso de uma forma simples: elas são ordenadas por prioridade, de cima para baixo em ordem decrescente, aproveitando a interface disponível. Com isso, fica fácil acrescentar novos comportamentos e depois organizá-los na estrutura já existente, o que permite que a construção seja feita em etapas, desfrutando de um processo visual e interativo.

Em suma, esses exemplos demonstram que a inclusão de um SBR em um jogo eletrônico pode trazer uma funcionalidade atraente para os jogadores, fornecendo uma maneira simples e intuitiva de criar comportamentos para personagens. Tendo isso, não há motivos para acreditar que os desenvolvedores não poderiam aproveitar essa ferramenta durante o desenvolvimento do jogo em si, tendo os mesmos benefícios.

Portanto, podemos defender que seria útil haver mais recursos que diminuam a complexidade de implementação desses sistemas, permitindo que desenvolvedores aproveitassem suas vantagens sem terem de construir uma estrutura do zero.

Objetivos do trabalho

Com isso, o principal objetivo do trabalho é criar uma ferramenta que forneça aos desenvolvedores de jogos o alicerce de um SBR, facilitando sua implementação em projetos específicos. Isso será feito por meio de um arcabouço (ou *framework*), que disponibilizará: a estrutura de dados para representar as regras; o objeto que representa o Sistema Baseado em Regras; um subsistema de inferência e unificação, já incluso no sistema principal (mais sobre isso na Seção 1.2.3) e alguma forma de integrar essas funcionalidades ao ambiente de desenvolvimento.

Além dessa estrutura básica, é desejável fornecer alguns módulos reutilizáveis que têm grande chance de serem usados em qualquer tipo de jogo, o que irá adiantar o trabalho dos desenvolvedores, além de servir como modelo para novas funcionalidades que eles desejem acrescentar. Isso está ligado à outra meta: facilitar a expansão do *framework* pelo desenvolvedor usuário, garantindo flexibilidade e adaptabilidade do sistema a qualquer projeto.

Outro ponto importante é que *videogames* costumam ser desenvolvidos por equipes multidisciplinares e como a definição da inteligência de personagens costuma ser responsabilidade de um *game designer*, alguém que não possui necessariamente experiência em programação, é relevante que a ferramenta possua uma interface gráfica na qual a edição das regras seja fácil. Os exemplos sugerem que esse tipo de interface permite que pessoas sem conhecimento técnico construam comportamentos interessantes, algo que ajudará muito o trabalho da equipe desenvolvedora.

Ademais, há o desejo de tornar a tecnologia de SBR o mais acessível e transparente possível, então será disponibilizado o código-fonte do arcabouço para outras pessoas poderem estudá-lo e entender como foi implementado. Isso exigirá uma licença de *software* que garanta liberdade de acesso e disseminação do programa. Há uma grande variedade de opções de licença, mas seu uso é extremamente dependente das tecnologias escolhidas

para o projeto, portanto, daremos preferência para a licença mais aberta possível que segue os conformes das aplicações utilizadas.

O principal fator que influenciou essa decisão foi o Movimento *Software* Livre[9], o qual acredita que o compartilhamento de código é benéfico não só para o desenvolvimento de aplicações, mas também para gerar e transmitir conhecimento entre a comunidade de desenvolvedores. Isso também teve impacto na escolha das ferramentas usadas no projeto, priorizando opções de código aberto sempre que possível.

Por fim, além dos objetivos práticos há também o estudo de conhecimentos teóricos. Em especial, será analisada uma técnica de IA pouco abordada em outros contextos, algo que irá permear todo o trabalho e espera-se que traga experiência relevante para a formação de um especialista em Inteligência Artificial. A outra vertente teórica-prática a ser estudada é o processo de construção de código para desenvolvimento de jogos, o que será feito conforme a metodologia explicada a seguir e permitirá um contato mais próximo com o sistema de um motor de jogos.

Trabalhos relacionados: *Think Engine*

Quando procuramos por trabalhos existentes que integram alguma forma de IA declarativa com desenvolvimento de jogos, vemos que não existem tantos exemplos. Uma referência encontrada que foi julgada relevante foi o pacote ***Think Engine*** (Angilica *et al.*, 2022) para o motor de jogos *Unity*[10], o qual integra módulos de Programação de Conjunto Resposta (*Answer Set Programming*, ou *ASP*) diretamente no Editor.

Podemos definir *ASP* como uma forma de programação lógica declarativa na qual reduzimos problemas de busca à computação de modelos estáveis, que também são chamados de conjuntos de resposta, ou *Answer Sets* (Lifschitz, 2019). A principal diferença para um Sistema Baseado em Regras é a natureza puramente lógica da definição de Programação de Conjunto Resposta, visto que não é um tipo de sistema a ser executado, mas sim uma forma de expressar problemas abstratos.

Os detalhes do funcionamento de *Answer Set Programming* estão fora do escopo desta monografia, mas é importante saber que o *Think Engine* precisa utilizar um *solver* lógico especializado nesse tipo de programação. A estrutura completa do pacote para a *Unity* pode ser visualizada na Figura 3, na qual vemos a presença do *ASP Solver* na camada de raciocínio, a mais interna do sistema.

Olhando mais atentamente para a arquitetura, na Figura 3, vemos que existem dois tipos de "cérebros", que seriam os objetos programados para representar o modelo de IA: um que utiliza planejamento e outro que apenas recebe estímulos e aciona atuadores. Ao comparar ambos com a proposta de um Sistema Baseado em Regras, vemos que o cérebro reativo é o que mais se aproxima, visto que ele lida diretamente com o mundo virtual.

Prosseguindo, podemos notar que a estrutura em várias camadas do *Think Engine* é bem mais complexa do que é pretendido neste trabalho, em especial, a inclusão de uma

[9] Saiba mais sobre *software* livre em: https://www.gnu.org/philosophy/free-software-intro.pt-br.html

[10] Site oficial da Unity: https://unity.com/pt

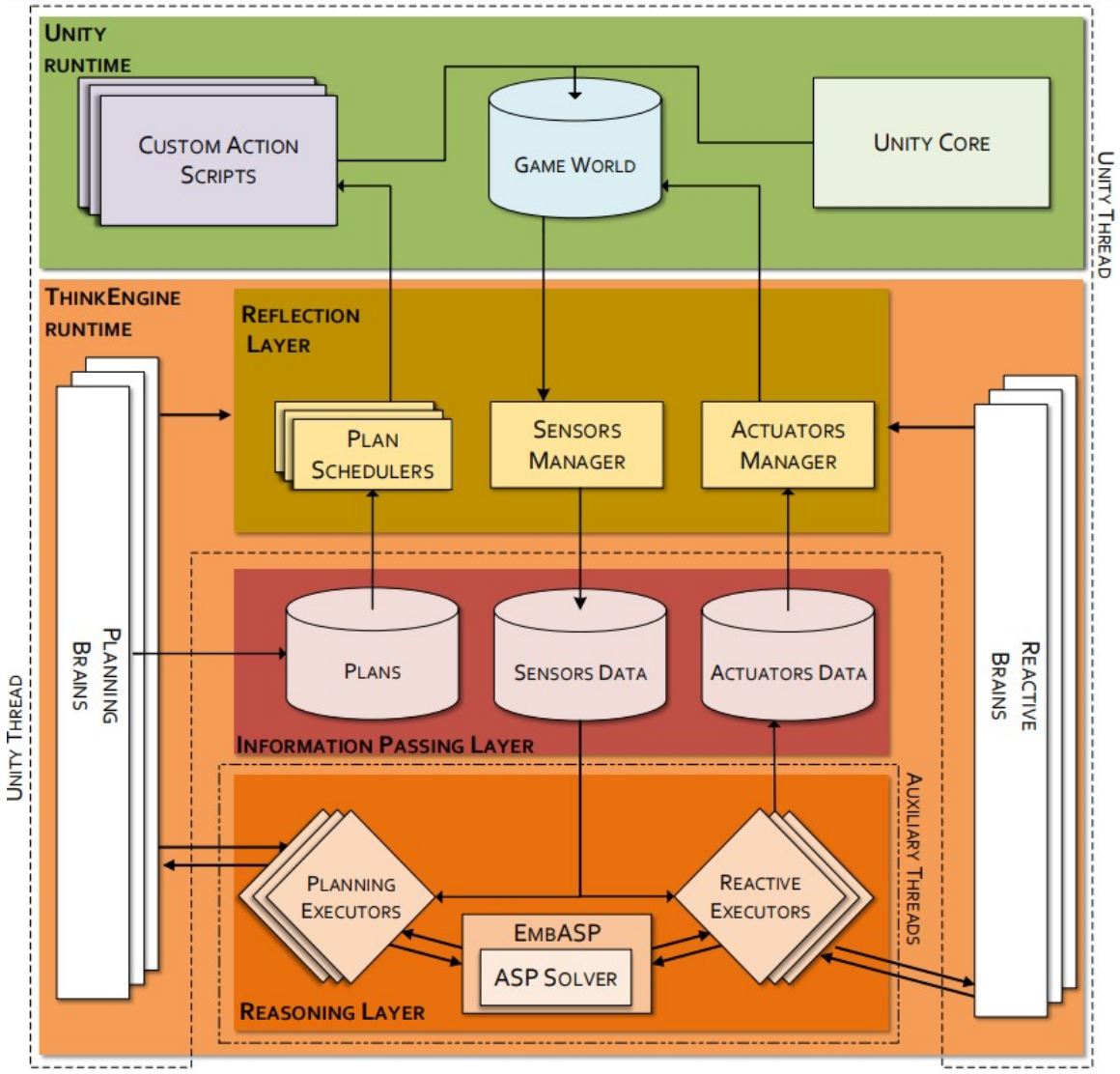

Figura 3: *Arquitetura interna do Think Engine (em inglês). Fonte: ANGILICA et al., 2022.*

camada intermediária para passagem de informação não se mostra necessária quando a estrutura de dados das regras contém tudo que é necessário. Além disso, o papel de sensores e atuadores pode ser abstraído para as partes "se" e "então" das regras que iremos usar, respectivamente, sem a necessidade de lidarmos com componentes externos.

Dessa forma, fica claro que o sistema presente no pacote para *Unity* possui uma complexidade de implementação ainda maior do que um Sistema Baseado em Regras, tendo em vista a inclusão de um módulo de planejamento, o gerenciamento de componentes extras como sensores e atuadores e a necessidade de um *solver* de programação lógica. Portanto, julgando esse sistema conforme os nossos objetivos, o desejo de fornecer uma ferramenta compreensível e extensível pelo desenvolvedor usuário estaria prejudicado.

Ademais, a facilidade de uso por pessoas não programadoras não é uma funcionalidade alcançada pelo *Think Engine*, já que para especificar o comportamento de um cérebro é preciso criar um arquivo `.asp` e declarar regras lógicas em uma sintaxe parecida com

Prolog (Angilica *et al.*, 2022). Por fim, o fato da *Unity* não ser de código aberto fere nosso princípio de *software* livre, o que nos leva a procurar outras opções de motores de jogo que sejam *open-source*.

Metodologia

Portanto, para cumprir os objetivos destacados, o projeto a ser desenvolvido é a construção de um *plugin* para a ***Godot Engine***[11], um motor de jogos de código aberto, que implemente um SBR genérico e com interface gráfica. Esse complemento irá adicionar ao motor de jogos os componentes mencionados anteriormente, por meio de novos tipos de nós e cenas (mais sobre esses conceitos na Seção 1.3.1) e da inclusão de *scripts* especiais, assim construindo um arcabouço que poderá ser usado em vários jogos.

Dentre os motivos da escolha pela *Godot*, podemos citar que seu repositório[12] é o número um no *GitHub* dentre *engines open-source*, sendo talvez a opção de código livre mais conhecida atualmente. Em decorrência dessa escolha, o projeto irá possuir a mesma licença *MIT*[13], para manter compatibilidade e garantir o princípio de código livre adotado, além de ser necessário haver uma licença para submeter o *plugin* na biblioteca oficial de recursos, a *Godot Asset Library*[14], algo que será uma meta para a etapa final do trabalho.

O código será construído e disponibilizado em um repositório aberto ao público do *GitHub*[15], que está referenciado no Apêndice A. Todas as versões do sistema mencionadas neste texto estarão etiquetadas no repositório; além disso, a documentação de uso da versão final será incluída no documento *README* e os diagramas de classe com a arquitetura de alto nível do sistema serão disponibilizados.

Porém, antes de se começar a escrever código será feita uma pesquisa por material acadêmico e exemplos práticos da aplicação de Sistemas Baseados em Regras em jogos, fazendo assim um levantamento bibliográfico de referências e inspirações relevantes. Esse estudo irá se intercalar com o desenvolvimento do *software* em etapas posteriores do trabalho, mas é essencial haver um entendimento básico do assunto antes de se começar a programar.

Tendo uma boa noção da estrutura do sistema, será iniciado o desenvolvimento do *plugin* seguindo um modelo inspirado em métodos ágeis, com iterações de duas semanas e entregas parciais. As versões intermediárias mais relevantes são: *Alpha*, na qual há usabilidade mínima, o que no nosso caso já irá incluir uma interface gráfica simples; *Beta*, na qual todas as funcionalidades já estão inclusas, mesmo que com problemas; e versão de produção 1.0.0, na qual o sistema está estável e teoricamente pronto para ser lançado na *Asset Library*.

Ademais, ao longo do desenvolvimento será criado um ambiente de testes, semelhante

[11] Site oficial do motor de jogos *Godot*: https://godotengine.org/

[12] Acesse o repositório da *Godot* em: https://github.com/godotengine/godot

[13] Licença MIT disponível em: https://opensource.org/license/mit/

[14] Visite a *Godot Asset Library* em: https://godotengine.org/asset-library/asset

[15] Acesse o repositório do projeto diretamente em: https://github.com/rvbatt/rule-based-godot

a um *videogame* muito simples de uma cena, para emular o uso de um usuário do complemento, ou seja, um desenvolvedor de jogos que utiliza *Godot*. Serão realizados testes de unidade nos componentes individuais e testes de integração, nos quais será feita a simulação de uma pessoa comum jogando a cena criada, com o intuito de analisar se o comportamento das entidades automatizadas é interessante no contexto de um jogo. Esse tipo de teste será subjetivo e irá contar com a experiência prévia do desenvolvedor com inúmeros jogos eletrônicos.

Concluindo, a métrica adotada para avaliar o desempenho do sistema será comparar o poder expressivo disponível com o que foi exibido nos exemplos de motivação, ou seja, queremos que ao final do projeto seja possível descrever comportamentos semelhantes aos expressos em *Dragon Age II* (*BioWare*, 2011) ou *Final Fantasy XII: The Zodiac Age* (*Square Enix*, 2017).

Com o entendimento geral do que este trabalho trata, passaremos agora a explicar as etapas de seu desenvolvimento. No Capítulo 1, faremos um estudo bibliográfico dos conceitos relevantes e apresentaremos as tecnologias que nos permitirão desenvolver o projeto. Já no Capítulo 2, explicaremos nossa proposta de *software*, elicitando seus requisitos e o que deve ser cumprido para solucionar os problemas apontados. O Capítulo 3 consistirá em uma série de relatos sobre o processo de desenvolvimento, com as dificuldades encontradas e decisões tomadas. No Capítulo 4 será feita uma apresentação da versão final do sistema, mostrando sua arquitetura e como o usuário utiliza as funcionalidades prontas. Por fim, no Capítulo 5 daremos nossas conclusões sobre o trabalho, relatando se ele cumpriu seus objetivos ou não, e indicando possíveis trabalhos futuros.

Capítulo 1

Fundamentação teórica e tecnológica

Começaremos agora com o estudo mais aprofundado dos conteúdos teóricos e ferramentas tecnológicas disponíveis para a elaboração do projeto. Antes de detalhar o tipo de sistema escolhido, será dada uma visão geral da área de IA em jogos digitais, com um breve histórico de seus avanços. A seguir, explicaremos os Sistemas Baseados em Regra, do funcionamento básico a um subsistema que aumenta consideravelmente seu poder expressivo. Por fim, exploraremos o ferramental fornecido pelo motor de jogos escolhido para o projeto, dando ênfase aos itens que serão úteis para o desenvolvimento do *software*.

1.1 Breve histórico da IA em jogos

Para contextualizar a área de estudo deste trabalho, vamos passar rapidamente por algumas das técnicas usadas ao longo da história dos *videogames* para reger o comportamento de personagens autônomas, o que não representará uma lista completa, mas pontuará avanços relevantes a este trabalho. Para cada tecnologia citada, daremos um jogo como exemplo, não necessariamente o pioneiro em seu uso, mas aquele que teve maior impacto em sua disseminação, segundo os autores Millington e Funge (2009).

Se olharmos historicamente, talvez o primeiro jogo no qual temos inimigos com algum nível de inteligência atuando contra o jogador é *Pac-Man* (*Bandai Namco*, 1980). Nele, os fantasmas se movimentando pelo nível parecem perseguir o jogador quando esse está perto, mas se o *Pac-Man* come uma cereja e se torna uma ameaça, os inimigos fogem.

A aparente estratégia dos fantasmas foi programada com uma **Máquina de Estados Finita**, uma técnica extremamente simples e comum, usada até hoje na indústria de jogos. O modelo apresenta ótimo desempenho para comportamentos simples, com poucos estados bem definidos, mas assim que tentamos aumentar a complexidade encontramos um problema: um crescimento significativo no número de estados e exponencial na quantidade de transições entre eles.

Na década seguinte, enquanto os jogos continuavam utilizando basicamente a mesma tecnologia de máquina de estados, outra maneira de melhorar a atuação das personagens controladas pelo computador foi introduzida em *Goldeneye 007* (*Rare Ltd.*, 1997): um sistema de percepção. Com ele, os inimigos viam seus colegas e percebiam se foram mortos, o que alterava seu comportamento, assim adicionando uma nova camada à inteligência.

Além disso, nos anos 90 houve a introdução de um nível de planejamento que se encontra abstratamente acima da tomada de decisão individual, formado pela chamada Estratégia de grupo. Agora, podemos encontrar esquadrões de inimigos que possuem um comportamento conjunto, no qual todos os indivíduos tomam decisões com base na conduta do grupo, algo que ficou mais evidente em jogos de estratégia em tempo real (*RTS*) como *Warhammer: Dark Omen* (*Mindscape*, 1998).

Chegando aos anos 2000, temos o lançamento de um dos primeiros jogos nos quais a IA é o ponto central da jogabilidade: *The Sims* (*Maxis*, 2000). Na simulação de mundo real apresentada pelo jogo, a experiência do jogador se baseia quase exclusivamente em observar o comportamento automático das personagens vivendo suas vidas virtuais. Ademais, o mecanismo de interação do jogador é indireto, não há como controlar diretamente os movimentos de seu avatar, somente selecionar ações e esperá-lo executá-las.

A técnica de inteligência artificial empregada pelos *Sims* é chamada de Comportamento Orientado a Metas (***Goal-Oriented Behavior***), um termo genérico que engloba qualquer método no qual primeiro são definidos objetivos e depois as ações são escolhidas de maneira a cumpri-los. Essa modelagem "de trás para frente" é benéfica em certos contextos, mas pode ser prejudicada por um número muito grande de opções, principalmente quando é preciso encadear várias ações, considerando os efeitos colaterais e o tempo necessário para executar cada uma.

Apenas quatro anos depois, tivemos a popularização de outra modelagem de IA que dominou a indústria de jogos, trazendo uma estrutura de dados que incorpora um comportamento de forma modular e hierárquica. As ***Behavior Trees*** utilizadas em *Halo 2* (*Bungie Software*, 2004), responsáveis pelo comportamento versátil dos alienígenas inimigos, foram as primeiras a serem descritas em detalhe e inspiraram muitos jogos que viriam depois.

Uma grande vantagem da estrutura das árvores de comportamento é ser de fácil visualização e abstração, algo que permite que *designers* sem conhecimento de programação montem as ações de uma personagem de forma relativamente fácil. Porém, esse modelo possui limitações, principalmente quando tenta representar estados convencionais, ou quando as personagens devem reagir a estímulos externos, algo simples que eleva o nível da inteligência (Millington e Funge, 2009).

Finalmente, chegamos ao estado atual da IA em jogos, no qual há inúmeras técnicas sendo empregadas e uma grande diversidade de abordagens. Nos últimos anos, observamos o crescimento de Aprendizado de Máquina nessa área, trazendo a possibilidade de ajustar as ações das personagens conforme interagem com o jogador, o que está fora do escopo deste trabalho e pode ser estudado mais a fundo no artigo de Lu e Li (2022).

Em suma, muitas técnicas foram desenvolvidas na indústria de jogos e houve avanços significativos desde os primórdios das máquinas de estado. O tipo de sistema que estuda-

remos mais a fundo neste trabalho é apenas um de muitos, mas que se destaca por sua utilidade fora do contexto de *games* e pelo manuseio de lógica declarativa, uma ferramenta cuja simplicidade de entendimento esconde um grande poder expressivo.

1.2 Sistemas Baseados em Regras

Um Sistema Baseado em Regras (SBR) é um tipo de sistema especialista, o que é definido na área de Inteligência Artificial como um *software* que visa emular a inteligência de um especialista em um determinado domínio, por meio da representação e utilização de conhecimentos fornecidos manualmente pelos projetistas, ou inferidos pelo sistema. Embora não seja a abordagem mais popular em desenvolvimento de jogos atualmente, sistemas especialistas são utilizados há quase três décadas nessa área (Millington e Funge, 2009).

No contexto deste trabalho, consideramos um SBR que está inserido em um mundo virtual dinâmico com o propósito de tomar decisões em tempo real acerca da atualização desse mundo, o que será expresso tanto por ações de personagens não controladas por jogadores, quanto por mudanças no próprio ambiente. Essa modelagem mais ampla nos permitirá abstrair o funcionamento do mundo e utilizar a mesma abordagem declarativa para expressar desde pequenas atualizações no estado interno de um agente, até uma configuração global de salvamento do jogo.

Em outras aplicações de sistemas especialistas, é possível defini-los como caixas-pretas que apenas respondem perguntas, mas em nosso caso, queremos um comportamento mais parecido com um sistema de controle, que interage diretamente com o mundo e não somente recebe uma requisição e devolve uma próxima ação a ser realizada.

1.2.1 Estrutura e funcionamento básico

A estrutura básica de um SBR é dada por: uma seção de memória contendo fatos relevantes; um conjunto de regras "se-então" que operam sobre esses fatos e um algoritmo de resolução de conflitos entre regras. Em aplicações concretas, os fatos podem ser armazenados em um banco de dados interno ou obtidos diretamente da memória de trabalho disponível ao sistema, e o algoritmo de resolução pode ser encapsulado em um objeto chamado **Árbitro**.

As regras possuem dois componentes: a **condição** (parte "se"), que representa um predicado formado por afirmações acerca dos fatos disponíveis ao sistema, conectadas por operadores *booleanos*, e um conjunto de **ações** (parte "então"), que expressam mudanças a serem aplicadas no mundo no qual o sistema está inserido, ou alterações aos próprios fatos do sistema. Dizemos que uma condição, ou a regra associada, é **satisfeita** quando seu predicado é verdadeiro e **acionar** uma regra, ou uma ação, significa realizar todas as modificações descritas na parte "então".

Como os fatos utilizados pelas regras são mutáveis, o valor verdade das condições está em constante mudança. Dessa forma, para verificarmos se uma condição é verdadeira, ou satisfeita, é preciso fazer um **pareamento** entre cada afirmação presente e os fatos disponíveis naquele instante de execução, que podem ser completamente diferentes no

momento seguinte. Para facilitar a nomenclatura, cada afirmação atômica de uma condição será também chamada de "pareamento", no sentido de uma associação entre dois valores que precisam ser pareados, comparados.

Vale destacar que o processo de verificação de uma condição pode se tornar complexo, especialmente quando adicionamos conexões entre os pareamentos, o que é feito por meio das **variáveis**. Isso será discutido na Seção 1.2.3.

Entrando na parte de funcionamento, o método de inferência mais comumente utilizado em SBR é o "Encadeamento para Frente" (***Forward Chaining***), no qual partimos dos dados e aplicamos as regras válidas, gerando os resultados determinados por essas regras. O outro principal método, menos aplicado nesse contexto, é o "Encadeamento para Trás" (*Backward Chaining*), no qual partimos de um fato desejado e tentamos encontrar um conjunto de regras que quando acionadas geram esse objetivo.

Neste trabalho trataremos apenas de *Forward Chaining*, por ser o método mais comum e menos complexo de ser implementado, mas existem alguns exemplos de uso de *Backward Chaning* para jogos que valem ser estudados, como o presente em Bourg e Seemann, 2004. Vale apontar que esse método seria classificado como *Goal-Oriented Behavior*, algo que foi mencionado na Seção 1.1 como a técnica empregada no jogo *The Sims*.

Tendo isso, um ciclo básico de funcionamento do sistema consiste em:

1. Realizar uma varredura periódica do conjunto de regras e fatos disponíveis;
2. Coletar todas as regras satisfeitas;
3. Pedir para o Árbitro escolher uma delas;
4. Acionar a escolhida.

Uma variação possível é continuar acionando as regras aplicáveis até não haver mais nenhuma, algo que é utilizado quando o sistema tem a intenção de corrigir inconsistências ou manter os dados atualizados, e as regras descrevem como ir de um estado instável para um estável. Para se obter isso, basta repetir os passos 3 e 4, desde que o Árbitro seja projetado para parar a execução quando não há mais nenhuma regra satisfeita e que a regra acionada seja sempre removida do conjunto coletado.

A estratégia empregada pelo Árbitro para escolher qual regra deve ser acionada — ou acionada primeiro, no caso de acionarmos todas as possíveis — pode variar. Essas estratégias podem ser chamadas de "resolução de conflito" na literatura (Bourg e Seemann, 2004), pois evitam inconsistências que poderiam surgir se todas as regras satisfeitas fossem aplicadas sem uma ordem específica, mas neste trabalho empregaremos o termo **arbitragem de regras**, por ser mais intuitivo.

Para ilustrar o funcionamento básico do sistema, daremos um exemplo de como as regras podem ser representadas em uma sintaxe declarativa simples e como elas se aplicam em uma base de dados.

1.2.2 Exemplo simples

Considere um jogo semelhante ao *Dragon Age II* (*BioWare*, 2011), no qual controlamos uma personagem Protagonista e temos uma aliada Curandeira, que é controlada pelo computador. Suponha que, em vez da interface gráfica mostrada na Figura 1, temos um editor de texto para definir o comportamento da Curandeira, por meio da declaração de regras em uma sintaxe simplificada. Assim, tome as seguintes regras, declaradas no Programa 1.1:

Programa 1.1 Regras simples para Curandeira.

```
1    ▷ Primeira regra
2   SE Vida da Curandeira < 25%
3   ENTAO Curandeira toma poção de cura
4
5    ▷ Segunda regra
6   SE Curandeira medrosa E Protagonista defendendo Curandeira
7   ENTAO Curandeira corajosa, REMOVE Curandeira medrosa
8
9    ▷ Terceira regra
10  SE Vida da Protagonista < 50% OU Protagonista pedindo cura
11  ENTAO Curandeira anda até Protagonista, Curandeira cura Protagonista
```

Agora, considere que no momento em que fazemos a verificação das condições, os fatos presentes em nosso sistema são:

```
1   Vida da Curandeira = 85%
2   Curandeira medrosa
3   Vida da Protagonista = 65%
4   Protagonista pedindo cura
5   Protagonista defendendo Curandeira
```

Com isso, podemos facilmente ver que a primeira regra não é satisfeita, enquanto a segunda e terceira regras são, portanto, as duas últimas serão dadas para o Árbitro. Suponha que, para a arbitragem escolhida, as regras são declaradas em ordem de prioridade, de cima para baixo em ordem decrescente, e o Árbitro escolhe sempre a regra satisfeita de maior prioridade, ou seja, a que foi declarada mais acima.

Dessa forma, será selecionada a segunda regra, que possui ações que alteram os fatos do sistema, trocando o estado da Curandeira para “corajosa” e removendo o anterior de “medrosa”. Consequentemente, o conjunto de dados do sistema após acionamento ficará assim:

```
1   Vida da Curandeira = 85%
2   Vida da Protagonista = 65%
3   Protagonista pedindo cura
4   Protagonista defendendo Curandeira
5   Curandeira corajosa
```

Como podemos ver, a estrutura básica de um SBR já proporciona alta flexibilidade no controle de um mundo virtual, porém, como foi mencionado anteriormente, há um tipo de recurso que quando adicionado aumenta consideravelmente o poder declarativo das regras: unificação de variáveis.

1.2.3 Unificação de variáveis

A grande ideia por trás do processo de unificação de variáveis é podermos escrever regras sobre entidades genéricas — em vez de especificarmos um mesmo comportamento para todos os objetos de um mesmo tipo, por exemplo, o que rapidamente se torna inviável em ambientes com uma abundância de objetos. No momento em que escrevemos uma regra com uma variável, não estamos pensando em uma atribuição específica, mas sim abstraindo um grupo ou uma classe de entidades.

Sabendo disso, uma **variável**, ou ***wild card***, é um identificador para uma entidade coringa, uma referência para um objeto que só é atribuída em tempo de execução. As variáveis possuem nomes únicos no escopo da regra na qual são utilizadas, pois a ideia é garantir que todas as suas ocorrências referenciem um único objeto ou valor, por meio do processo de **unificação**. Assim, cria-se uma conexão entre os pareamentos que utilizam um mesmo nome, aumentando o poder expressivo da condição, além de possibilitar que as ações usem o mesmo objeto pareado.

Quando uma variável é utilizada na condição, isso indica que é preciso procurar um **vínculo** que a torne satisfeita, correspondendo a um objeto ou valor que pode ser utilizado no lugar de todas as instâncias dessa variável, assim possibilitando uma substituição. A busca deve ser restrita a um escopo definido pelo projeto do sistema, e a impossibilidade de encontrar uma substituição válida implica que a condição é considerada falsa.

Portanto, a condição só será satisfeita se houver um conjunto de vínculos tal que, para cada vínculo entre uma variável e um valor, todo predicado que utiliza essa variável pode ser satisfeito ao substituí-la pelo valor associado. Para diferenciarmos variáveis e valores de cadeias de caracteres, vamos adotar uma convenção utilizada no livro de Millington e Funge (2009), na qual adicionamos um **ponto de interrogação no começo de uma variável** para demarcá-la. Assim, *?exemplo* representa uma variável, mas *exemplo* é um valor constante.

Usando essas substituições, ações podem acessar os valores pareados por meio dos nomes das variáveis, proporcionando uma transmissão de dados entre as duas partes de uma mesma regra; vale ressaltar que variáveis de mesmo nome em regras diferentes não são unificadas. Essa passagem de informação é feita sem a necessidade de camadas extras no sistema, apenas utilizando a estrutura existente e a funcionalidade de unificação de variáveis.

Assim, para ilustrar o uso de regras com variáveis e o processo de unificação, iremos modificar e expandir o exemplo dado na Seção 1.2.2, de forma a acomodar um time com mais personagens:

Programa 1.2 Regras com variáveis para Curandeira.

```
1    ▷ Terceira regra, versão genérica
2  SE ?Aliada éaliada E (Vida da ?Aliada < 50% OU ?Aliada pedindo cura)
3  ENTAO Curandeira anda até ?Aliada, Curandeira cura ?Aliada
4
5    ▷ Quarta regra
6  SE ?Aliada éaliada E ?Aliada defendendo Curandeira
7      E ?Aliada sendo atacada por ?Inimigo E Curandeira corajosa
8  ENTAO Curandeira ataca ?Inimigo
```

Com essas alterações, a terceira regra, que antes só era satisfeita quando a Protagonista estava com a saúde debilitada ou pedindo cura, agora passa a acomodar qualquer aliada em uma dessas situações, o que significa que a Curandeira irá perceber quando qualquer pessoa do seu time estiver precisando de ajuda e tentará ir até ela. Repare que a mesma variável é utilizada nas ações, o que garante que a mesma personagem que foi vinculada ao termo *?Aliada* será a que receberá a cura.

Já para a quarta regra, temos que uma situação específica precisa ocorrer: uma aliada qualquer *A* está defendo a Curandeira, simultaneamente está sendo atacada por um inimigo *I* e, além disso, a Curandeira está se sentindo corajosa. Somente nessa situação que a regra deve ser acionada, fazendo a Curandeira atacar *I*, assim ajudando sua defensora.

Prosseguindo, considere os seguintes fatos no momento da verificação:

```
1  Vida da Protagonista = 65%
2  Vida da Curandeira = 85%
3  Curandeira corajosa
4  Vida da Guerreira = 50%
5  Guerreira defendendo Curandeira
6  Vida da Feiticeira = 25%
7  Feiticeira sendo atacada por Zumbi
```

Na versão genérica da terceira regra, o sistema de unificação irá procurar uma substituição para a variável *?Aliada* e descobrirá que a Feiticeira é a única que satisfaz a condição. Dessa forma, essa regra é satisfeita com o vínculo associado entre *?Aliada* e *Feiticeira*.

Para a quarta regra, o sistema percebe que a Guerreira está defendendo a Curandeira e que a Feiticeira é a única sendo atacada, porém, como a condição exige que seja a mesma personagem (por se tratar da mesma variável), ela não é satisfeita nesse momento. Se a segunda aparição de *?Aliada* fosse substituída por uma nova variável, digamos *?Aliada2*, então a regra seria satisfeita com os vínculos entre *?Aliada* e *Guerreira*, *Aliada2* e *Feiticeira*, *?Inimigo* e *Zumbi*. Portanto, nesse exemplo a Curandeira irá andar até a Feiticeira e a curar.

1.2.4 Desafios de implementação

Agora, vemos que existem três principais desafios ao se implementar um SBR: **(i)** como representar a estrutura de dados que compõe as regras; **(ii)** como avaliar se os fatos

satisfazem a condição de alguma regra, o que é especialmente complexo quando temos o subsistema de unificação de variáveis e **(iii)** como fazer a arbitragem entre as regras satisfeitas, para escolher qual delas será acionada.

Dentre esses itens, o terceiro se mostra o mais fácil, pois os algoritmos de arbitragem são abstratos e só possuem dois requisitos: a entrada consiste em um conjunto de regras satisfeitas e a saída deve ser uma dessas regras, que foi escolhida para ser acionada. O algoritmo mais simples possível é aquele que devolve a primeira regra aplicável (***First Applicable***), assumindo que as regras foram declaradas em ordem de prioridade e, portanto, a primeira satisfeita deve ser a acionada. Esse foi o método utilizado no exemplo da Seção 1.2.2.

Porém, um problema dessa arbitragem mais ingênua é que podem ocorrer situações nas quais a mesma regra é acionada repetidamente, pois ela é a primeira na lista de verificação e suas ações não alteram os fatos do sistema de uma forma que invalida sua condição. Há outros algoritmos mais complexos que evitam esse problema de ciclagem guardando um histórico das regras acionadas, ou então introduzindo um elemento de aleatoriedade.

Uma estratégia comum que evita os problemas do *First Applicable* é a chamada ***Least Recently Used (LRU)***, que mantém uma fila das regras acionadas e com isso procura sempre escolher aquela que foi usada menos recentemente. Na primeira execução, pode-se utilizar a primeira regra aplicável, mas após isso o método verifica em seu histórico se alguma das regras satisfeitas ainda não foi acionada, ou qual delas foi invocada há mais tempo.

Sobre o item **(i)**, representação das regras, vemos que a condição é um componente com grande potencial para ser otimizado, pois a lógica *booleana* que compõe sua estrutura fornece uma organização intuitiva: os pareamentos sendo objetos atômicos e os operadores *booleanos* atuando como conectores entre eles, nos levando a pensar em uma estrutura de **árvore**, na qual pareamentos são folhas e operadores são nós internos.

Já as ações podem ser pensadas como pedaços de código a serem executados, em sua forma mais simples. Em programação funcional, seria possível guardar uma referência para uma função e então chamá-la, passando como argumento os vínculos criados durante a verificação da condição. Caso esse artifício não esteja disponível, é possível parametrizar as mudanças que devem ser aplicadas de alguma forma.

Por fim, para a avaliação do valor verdade de uma condição com variáveis, podemos intuir que é necessário percorrer a estrutura de dados enquanto guardamos os vínculos entre as variáveis e suas atribuições. Assim, a maneira como a estrutura de regras é organizada e a ordem na qual a percorremos terão grande impacto no processo de avaliação da condição.

1.2.5 Vantagens e desvantagens

Superando-se os desafios mencionados, as principais vantagens de se usar regras declarativas para definir a IA em um jogo são a transparência, praticidade e velocidade com as quais é possível definir o comportamento de uma entidade. Escrever as regras segue uma lógica muito próxima de como as descreveríamos em linguagem natural (“se esta

condição for verdade, então estas ações ocorrem") e sua estrutura modular nos permite realizar modificações manuais de forma simples, mudando completamente a estratégia de uma personagem com apenas algumas alterações em suas regras (Angilica *et al.*, 2022; Bourg e Seemann, 2004).

Além disso, quando comparamos essa estratégia com outros modelos de tomada de decisão empregados no desenvolvimento de jogos, vemos que ela apresenta um poder expressivo, generalidade e escalabilidade que são difíceis de se equiparar. Por comparação, no modelo de Máquina de Estados, mencionado na Seção 1.1, representar um comportamento complexo exige um número demasiado de estados, o que por sua vez implica em uma quantidade ainda maior de transições entre eles, enquanto em SBR, uma única regra pode representar desde uma situação muito específica até um contexto amplamente genérico, com o uso de variáveis.

Outra comparação relevante é com as *Behavior Trees*, sendo que as duas abordagens dão foco para partes distintas do processo de tomada de decisão: as árvores de comportamento se concentram na execução de ações, enquanto os Sistemas Baseados em Regras têm como cerne as condições necessárias para se acionar uma ação. Dessa forma, vemos que os dois se complementam até certo ponto, e a princípio não há impedimentos técnicos que nos impeçam de uni-los, usando *Behavior Trees* no lugar das ações acionadas pelas regras.

Para ilustrar a diferença de complexidade entre as abordagens, iremos escolher um comportamento simples e implementá-lo de três modos: com uma Máquina de Estados, uma *Behavior Tree* e um conjunto de regras. Assim, considere uma personagem que precisa chegar a um cômodo passando por uma porta; as Figuras 1.1 e 1.2 e o Programa 1.3 demonstram cada modelagem possível, na ordem mencionada anteriormente.

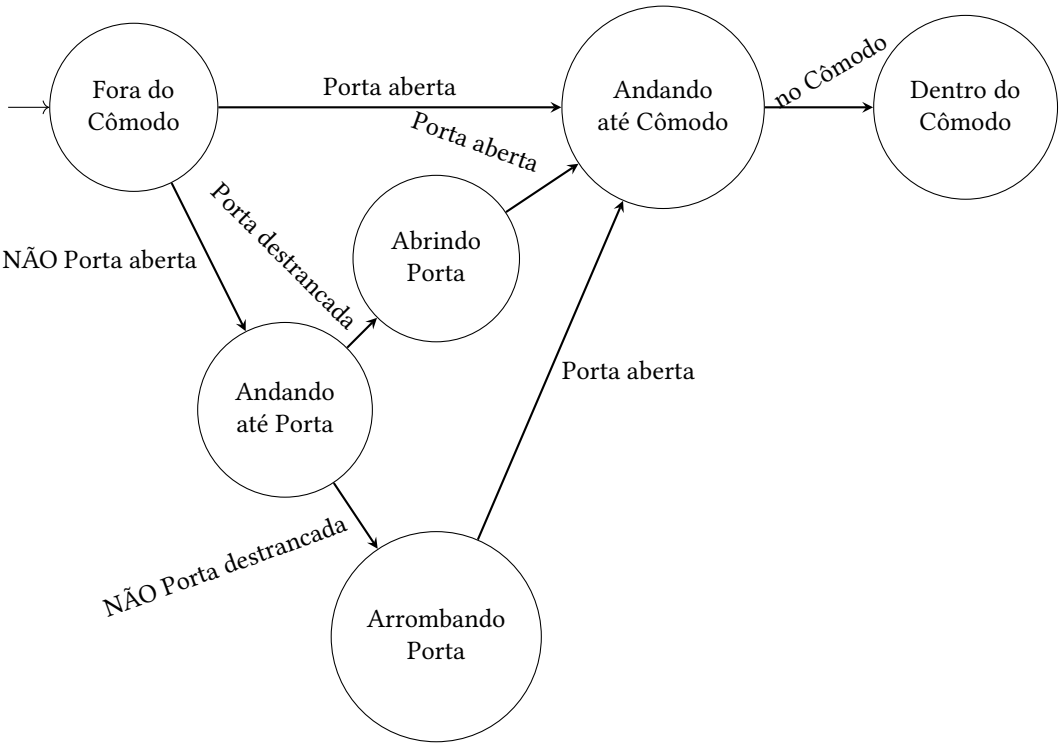

Figura 1.1: *Máquina de Estados para comportamento do cômodo com porta.*

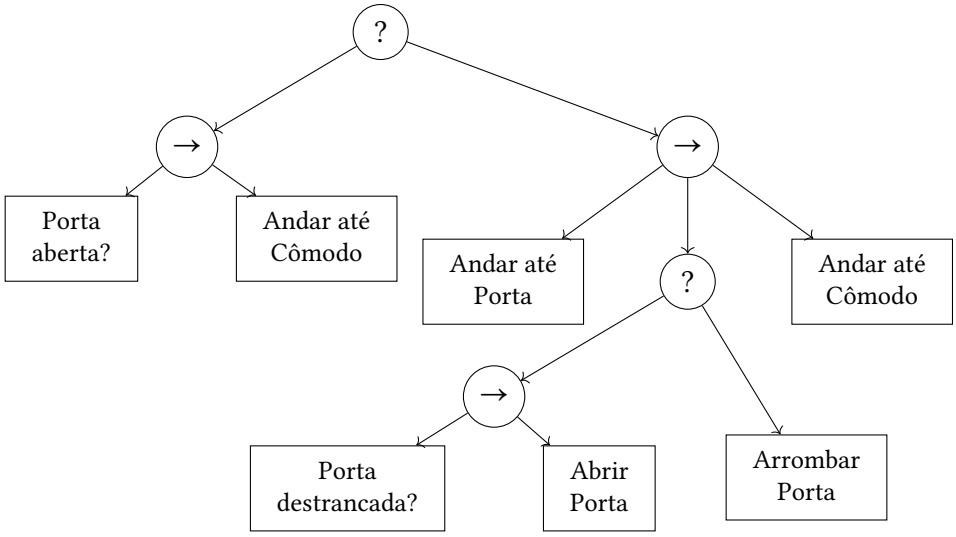

Figura 1.2: *Behavior Tree para comportamento do cômodo com porta.*

Programa 1.3 Conjunto de regras para comportamento do cômodo com porta.

```
1    ▷ Primeira regra
2   SE Porta aberta
3   ENTAO Andar até Cômodo
4
5    ▷ Segunda regra
6   SE NAO Porta aberta E Porta destrancada
7   ENTAO Andar até Porta, Abrir Porta, Andar até Cômodo
8
9    ▷ Terceira regra
10  SE NAO Porta aberta E NAO Porta destrancada
11  ENTAO Andar até Porta, Arrombar Porta, Andar até Cômodo
```

Comparando todos os métodos, notamos que o baseado em regras é o mais modular, contendo uma regra isolada para cada possibilidade da porta (aberta, fechada e trancada, apenas fechada), enquanto os outros precisam incorporar todos os três casos em uma estrutura unificada. Além disso, podemos perceber que a forma textual declarativa é mais fácil de ser entendida por alguém que desconhece técnicas de IA, enquanto os grafos que representam Máquinas de Estado e *Behavior Trees* precisam de um certo nível de conhecimento para serem lidos.

Porém, os SBR também apresentam uma desvantagem relevante: sua dificuldade de implementação, algo que gerou uma reputação negativa na indústria de jogos e desmotiva desenvolvedores a usarem essa técnica (Millington e Funge, 2009). Sua complexidade está associada aos desafios explicados na Seção 1.2.4, que trazem a necessidade de um alto grau de abstração e da manipulação interna de formalismos declarativos.

Outro ponto contrário a esse tipo de sistema é seu desempenho, o qual é afetado principalmente pelo processo de unificação. Quando o conjunto de regras alcança um certo tamanho, o tempo necessário para processar cada condição, fazendo os vínculos

necessários, pode prejudicar a execução do jogo em si, principalmente se o SBR estiver gerenciando um aspecto básico de usabilidade.

Uma forma de mitigar o último problema é utilizando agrupamentos de regras que podem ser ligados e desligados, assim delimitando subconjuntos que codificam um aspecto específico do comportamento e que só precisam ser analisados quando esse aspecto é aplicável (Millington e Funge, 2009). Porém, mesmo com essa possível solução, o problema de desempenho continua uma desvantagem relevante, especialmente para jogos que exigem reações rápidas do jogador.

Unindo os problemas de ineficiência e dificuldade de implementação, conseguimos compreender o porquê dessa abordagem ser incomum no desenvolvimento de jogos na atualidade. Assim, como dito nos objetivos do trabalho, desejamos mitigar o segundo problema por meio de nossa implementação genérica, o que diminuirá consideravelmente essa desvantagem para desenvolvedores que usam o motor de jogos escolhido, o qual será tratado a seguir.

1.3 Motor de jogos *Godot*

Godot Engine é um motor de jogos de código aberto e livre (*FOSS*), disponibilizado gratuitamente sob a licença *MIT*[1] e que conta com uma grande comunidade de desenvolvedores e usuários. Algumas de suas características mais relevantes é o uso de uma linguagem própria, chamada *gdscript* e o fato de nenhum componente precisar ser compilado, incluindo o próprio Editor e os complementos criados por usuários.

Quando nos referimos ao motor de jogos, geralmente estamos falando do Editor da *Godot*, a aplicação que permite trabalhar nos projetos, criar cenas (mais sobre isso na Seção 1.3.1) e escrever os *scripts* em seu editor de texto embutido. Esse é o aplicativo com o qual os desenvolvedores têm contato e é nele que o *plugin* que compõe este trabalho será desenvolvido.

Agora, como essa foi a tecnologia escolhida para implementarmos este trabalho, será preciso entender um pouco de como o motor de jogos funciona e quais as ferramentas disponíveis ao desenvolvedor que utiliza a *Godot game engine*. Todas as informações apresentadas a seguir foram extraídas da documentação oficial da versão 4.1 (Linietsky *et al.*, 2014).

1.3.1 Nós, cenas e grupos

Os **nós** são os componentes atômicos de um jogo, os blocos de montar que quando unidos formam toda a estrutura do projeto. Existem dezenas de tipos diferentes, de corpos rígidos a *sprites*, câmeras e tocadores de som, além dos tipos básicos, os quais são apenas entidades vazias, como `Node2D` e `Control` (interface gráfica). Algo relevante para os desenvolvedores de *plugins* é que a Godot permite criar novos tipos de nós customizados.

Outra característica importante é que esses blocos podem ser organizados em uma estrutura de árvore, algo que passa a ser chamado de **cena**. Todos os elementos de alto

[1] Licença disponível em: https://github.com/godotengine/godot/blob/master/LICENSE.txt

nível que compõem o jogo serão cenas, desde as personagens até os níveis, sendo que é possível salvar cenas no sistema de arquivos e adicioná-las a outras cenas como se fossem um único nó, assim criando uma composição recursiva.

Um exemplo de uma cena simples se encontra na Figura 1.3, na qual observamos a estrutura de árvore à esquerda e o editor de cenas à direita, com as setas vermelhas conectando cada nó ao elemento visual gerado por ele.

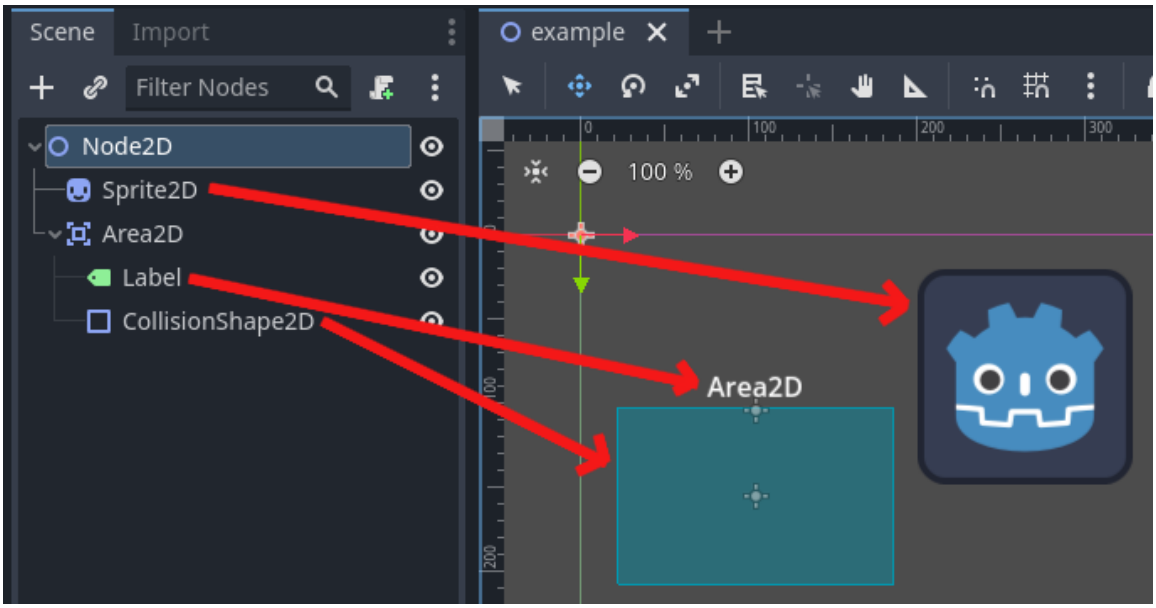

Figura 1.3: *Exemplo de uma cena simples na Godot 4.1. As setas em vermelho levam da representação do nó na árvore da cena (Scene) até o elemento gráfico que ele gera na tela. Esses elementos são visíveis tanto para o desenvolvedor, durante a edição do projeto, quanto para o jogador, durante a execução do jogo.*

Além disso, os nós podem ser organizados em **Grupos**, de maneira semelhante ao uso de *tags* em outros *software*, o que permite filtrar a árvore da cena e acessar apenas os nós que possuem a etiqueta desejada. Um nó pode estar em múltiplos grupos simultaneamente, e por meio do acesso à raiz da árvore, utilizando a função `get_tree()`, é fácil executar buscas em grupos específicos por meio de código.

1.3.2 Recursos

Outro componente básico são os **recursos**, que não acrescentam funcionalidades tão visíveis, mas servem como contêineres de dados para os nós. A principal diferença entre os dois é que os recursos não aparecem na árvore da cena, em vez disso eles são gerenciados como propriedades de um nó e atuam mais como estruturas de dados do que como objetos físicos no mundo virtual.

Ademais, a Godot também possibilita declarar novos tipos de recurso, assim permitindo que desenvolvedores declarem estruturas de dados novas, mas que possuem suporte nativo. Além disso, é possível determinar novos formatos de arquivo para salvar e carregar recursos do disco, além dos padrões `.tres` (recurso textual) e `.res` (binário). Isso é realizado com

novas subclasses de `ResourceFormatSaver` e `ResourceFormatLoader`, que podem ser programadas para entender *XML*, *JSON*, textos simples, entre outros.

Vale apontar que existe uma grande semelhança entre os recursos e os *ScriptableObjects* do motor de jogos *Unity*, uma das opções mais utilizadas por desenvolvedores de jogos.

1.3.3 Inspetor e painéis do Editor

Quando estamos editando um projeto, a interface gráfica com a qual interagimos é composta de ***docks*** ("docas", em tradução livre), que podem ser movidas de lugar e **painéis**, os quais são fixados à parte inferior do Editor. As docas adicionam funcionalidades mais gerais, potencialmente úteis em qualquer momento, como o explorador de arquivos, editor de cenas e menu de importação de recursos; já os painéis costumam fornecer menus úteis em um contexto específico, como o editor de animações para nós do tipo `AnimationPlayer` ou os resultados de busca, um painel que só aparece quando estamos com o editor de texto aberto.

A Figura 1.4 demonstra a interface padrão do Editor, com legendas em inglês de cada doca importante e a indicação do painel inferior.

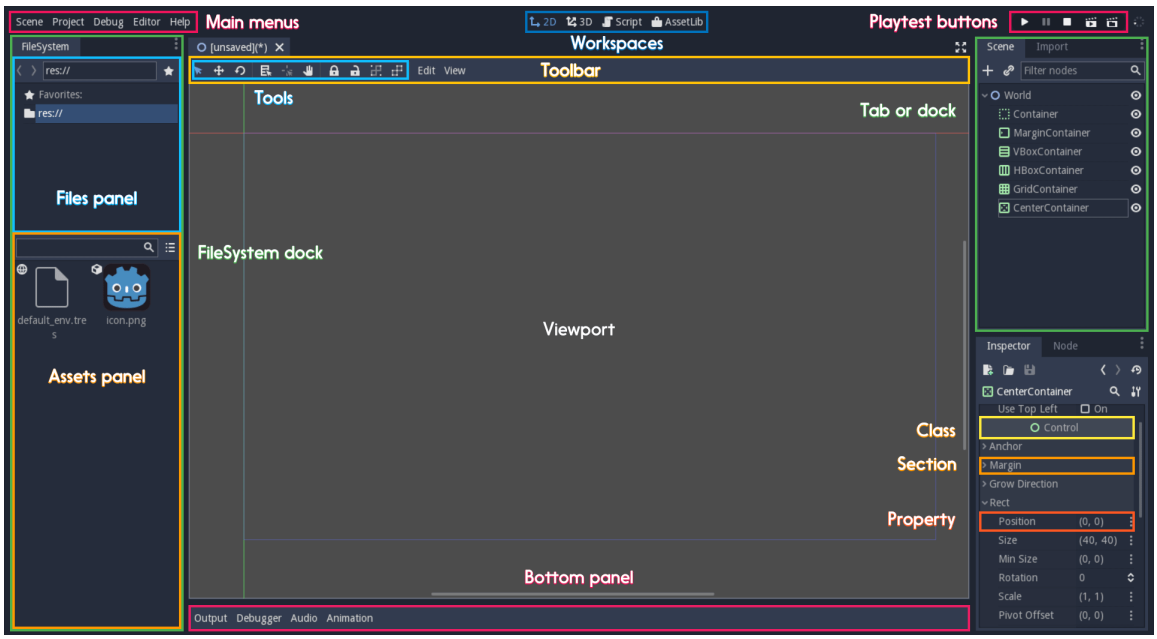

Figura 1.4: *Interface do Editor da Godot e seu vocabulário (em inglês). Cada parte relevante está etiquetada e demarcada por uma cor. Fonte: Godot Docs*[2].

Uma das docas mais importantes é o **Inspetor**, no qual aparecem as propriedades exportadas (serializadas) do nó atualmente selecionado, separadas em categorias, baseadas na hierarquia de classes, e em agrupamentos, que podem ser abertos ou fechados durante

[2] Disponível em: https://docs.godotengine.org/en/4.1/contributing/documentation/docs_writing_guidelines.html
Acesso em: 16 ago. 2023

a edição, por conveniência. Uma funcionalidade muito útil desse menus são os *widgets* de edição de propriedades para tipos nativos, que possuem usabilidade intuitiva; um exemplo se encontra na Figura 1.5.

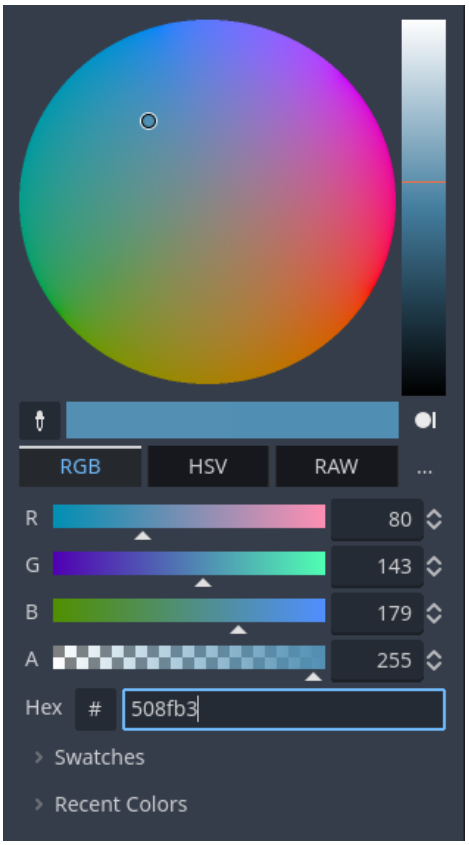

Figura 1.5: *Widget de seleção de cor presente no Editor da Godot 4.1.1. Repare na quantidade de opções, da seleção visual até o uso do código hexadecimal da cor.*

Vale destacar que o Inspetor fornece uma interface gráfica nativa para editar recursos, já que eles são considerados propriedades de nós Além disso, existe uma classe específica de *plugins* para essa doca, a qual permite modificar a maneira como cada propriedade é gerenciada e adicionar submenus personalizados.

1.3.4 Sistema de *plugins*

Uma ferramenta muito abrangente do motor de jogos é o sistema de *plugins*, que permite incluir novas funcionalidades ao Editor, como novos nós, painéis e *scripts* de ferramentas, tudo de forma transparente ao usuário. A maneira mais comum de se instalar esses complementos é por meio da loja oficial, *Godot Asset Library*[3], mas, na prática, tudo que se precisa fazer para utilizar um *plugin* é adicionar seu diretório em `res://addons/` e então ligá-lo no Editor.

A estrutura de um *addon* (termo também usado na comunidade) pode ser composta completamente por *scripts* escritos na linguagem nativa, *gdscript*, e por cenas como qualquer

[3] Acesse a biblioteca de *assets* em: https://godotengine.org/asset-library/asset

outra, que poderiam ser criadas em um projeto comum. Dessa forma, tanto a implementação quanto o entendimento se tornam fáceis e intuitivos para um usuário padrão.

Um *plugin* é formado por um arquivo de configuração e um *script* principal, que deve herdar da classe `EditorPlugin` e estar marcado com a anotação `@tool`, indicando que esse arquivo será executado no Editor. A partir desse *script*, é possível adicionar outros complementos de utilidades específicas, como o já mencionado `InspectorPlugin`, assim como *addons* de depuração, conversão de recursos, entre outros.

Um exemplo que demonstra a extensão das capacidades do sistema de *plugins* é o *Dialogic*[4], um complemento que permite criar personagens, diálogos e cenas para exibir conversas de maneira fácil. A Figura 1.6 mostra sua tela principal, a qual é integrada no Editor da *Godot* e permite a manipulação de arquivos especiais por meio de uma interface gráfica, criando linhas do tempo que então são transformadas em cenas.

[4] Repositório oficial do Dialogic: https://github.com/coppolaemilio/dialogic

[5] Disponível em: https://github.com/coppolaemilio/dialogic/blob/dialogic-1/addons/dialogic/Documentation/Content/Tutorials/Images/TimelineEditor_Example.PNG
Acesso em: 28 set. 2023

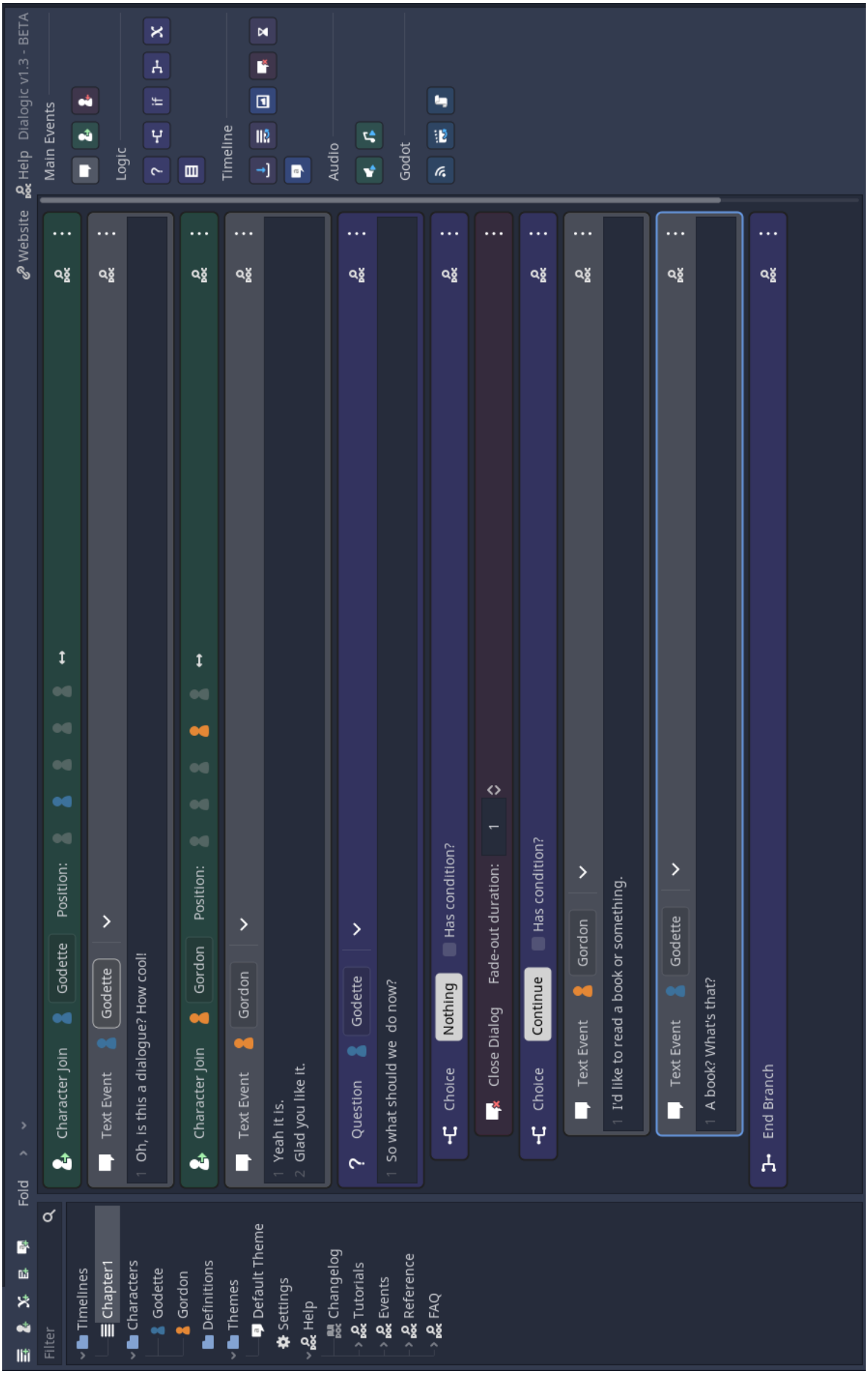

Figura 1.6: *Exemplo de linha do tempo em Dialogic 1.0. Fonte: repositório oficial*[5].

Capítulo 2

Proposta de solução

Como dito anteriormente, a proposta deste trabalho é desenvolver um *plugin* para o motor de jogos *Godot* (ou *addon*, como também é dito na comunidade) que implemente um SBR genérico e extensível, de forma a facilitar a montagem de comportamentos para personagens automatizadas. Tendo a base teórica para entender os mecanismos do sistema, trataremos agora das questões de implementação, detalhando como a nossa solução proposta irá resolver os problemas apontados.

Primeiro, faremos uma análise de requisitos nas Seções 2.1 e 2.2, listando as funcionalidades que a aplicação deve conter e as questões técnicas que devem ser consideradas. A seguir, daremos as linhas gerais da arquitetura de alto nível, explicando os componentes necessários e sua organização. Por fim, levantaremos os pontos que mais afetam a experiência do usuário com o *software* e tentaremos encontrar formas de atender às suas necessidades.

A Tabela 2.1 inclusa neste capítulo contém a listagem de todos os requisitos levantados para o projeto, os quais serão discutidos nas seções a seguir. Ela está separada por tipos e inclui o código de cada requisito, um identificador que será utilizado para referenciar cada requisito específico.

2.1 Requisitos funcionais

A principal funcionalidade do sistema é a declaração de **regras** que interajam com o ambiente do jogo, fazendo verificações relevantes acerca do mundo virtual e descrevendo ações que o alterem de alguma maneira significativa. Apesar disso, por se tratar de uma ferramenta para desenvolvedores, os dados que são pareados não precisam ser visíveis ao(s) jogador(es), assim como os resultados das ações não precisam ter efeitos necessariamente perceptíveis.

Como exemplo de uma situação na qual uma regra opera com dados ocultos ao jogador, suponha que o jogo possui uma área invisível que detecta quando uma personagem entra em um local proibido. Quando isso ocorre, é aumentada a probabilidade de se encontrar um inimigo. Tanto a área quanto o valor da probabilidade são desconhecidos pelo jogador, mas essas informações são relevantes ao desenvolvedor e, portanto, devem acessíveis.

Tipo	Código	Descrição
Funcionais	F1	Declarar regras com acesso a propriedades, métodos, sinais e relações como hierarquia e distância
	F2	Aplicar álgebra *booleana* nas condições
	F3	Ordenar ações
	F4	Alterar o método de arbitragem de regras
	F5	Configurar opções de iteração automática
	F6	Criar novos pareamentos, árbitros e ações de maneira fácil
	F7	Utilizar variáveis
	F8	Editar regras por meio de uma interface gráfica
Técnicos	T1	Compatível com versão *4.1+* da *Godot Engine*
	T2	Compatível com *gd-plug*
	T3	Uso de *GDScript*
	T4	Não deve ser necessário remanejar diretórios
	T5	Não deve ser necessário alterar estrutura de cenas
	T6	Não deve ser necessário modificar configurações do projeto
	T7	Não deve ser necessário compilar nenhuma parte do código
UX	X1	Integração inaparente com Editor nativo
	X2	Uso de *widgets* nativos para atribuição de propriedades
	X3	Cenas de demonstração para servir de exemplo
	X4	Boa documentação, incluindo *API* e passo-a-passo para criação de novas classes

Tabela 2.1: *Lista de todos os requisitos do software, incluindo funcionais, técnicos e de experiência de usuário. Os valores da* ***Código*** *serão usados para referenciar cada requisito específico.*

Para isso, a declaração de regras deve ser conceituada no ambiente do Editor da *Godot*, por meio dos elementos disponíveis ao desenvolvedor durante a edição de um projeto. Isso deve incluir as propriedades, os métodos e sinais[1] dos nós, assim como as relações entre eles que sejam relevantes para a elaboração de cenas, como a posição na hierarquia da árvore, ou a distância entre dois corpos. A condição de uma regra deve poder fazer pareamentos com esses elementos e as ações devem conseguir alterá-los.

Sobre as **condições**, é necessário que elas possibilitem verificações complexas que envolvam mais de um elemento, pois, na maioria das vezes, várias entidades estão envolvidas em um único evento de um *videogame*, algo que queremos controlar utilizando as regras. Um exemplo simples que demonstra isso são jogos de esportes de time, como futebol ou basquete, nos quais a posição de todos os jogadores em campo pode influenciar o comportamento das personagens automatizadas. Nesse contexto, algo como um passe de bola pode envolver praticamente todas as entidades presentes na cena do jogo.

Assim, é preciso haver uma estrutura que permita conectar pareamentos atômicos por meio de operações lógicas, de forma que seja possível compor uma condição de complexidade arbitrária utilizando verificações pontuais. Como estamos tratando de álgebra

[1] Saiba mais sobre os sinais da *Godot* em: https://docs.godotengine.org/en/4.1/getting_started/step_by_step/signals.html

booleana, precisamos ter, no mínimo, os conectores que a definem: *E*, *OU* e *NÃO*. Seria possível incluir mais conectores (como *OU* exclusivo), mas tendo os três fundamentais, sabemos que é possível construir qualquer outro (Boole, 1854). Portanto, decidimos manter uma implementação enxuta.

Agora, com relação às **ações**, elas têm o requisito básico de representar uma mudança a ser realizada no mundo do jogo, e de ser possível aplicá-la, gerando os efeitos esperados. Além disso, a única questão sensível em sua implementação é a ordem com que ocorrem, pois podem existir dependências entre elas, ou seja, uma ação necessitar de algo que só é obtido ao final de outra. Um exemplo é uma regra acionada quando uma personagem deve passar por uma porta, com as ações: andar até a porta, abri-la e atravessá-la; podemos ver que é necessário seguir essa sequência na ordem em que foi declarada. Desse modo, precisamos garantir a ordem de execução de alguma forma.

Além disso, sabemos que escolher qual regra será acionada em cada iteração tem grande impacto no funcionamento do sistema, principalmente para evitarmos o problema de ciclagem discutido na Seção 1.2.4. Assim, um fator que precisa ser ajustável é a **arbitragem**, o método de escolher dentre as regras satisfeitas qual deve ter suas ações disparadas. Adotaremos a *First Applicable* como padrão, por ser a mais simples de implementar, sendo que, em decorrência disso, decidimos incorporar a sua convenção para a listagem das regras: elas são declaradas em ordem de prioridade, da mais prioritária para a menos.

Outra configuração importante é a frequência com a qual o SBR deve iterar sob suas regras, pois isso varia conforme o tipo de jogo: em jogos de turno, podemos esperar até o jogador tomar uma decisão para verificar as regras, enquanto em jogos de tiro em tempo real é preciso que o mundo seja atualizado a cada *frame*. Portanto, seria conveniente fornecer uma maneira de definir quando o sistema deve iterar, apenas em gatilhos específicos ou de forma automática.

Olhando para os componentes mencionados até então, vemos que existem três tipos de recurso que geram variedade para a capacidade funcional do sistema: pareamentos, ações e métodos de arbitragem. Sabendo disso, um requisito importante para o *plugin* é proporcionar ao desenvolvedor usuário a habilidade de criar novas instâncias desses recursos. Isso porque nosso sistema poderá ser utilizado em diferentes jogos eletrônicos, cada um com sua lógica própria e aspectos únicos que precisam ser verificados ou modificados, sendo que as regras devem ter capacidade de modelar isso.

Consequentemente, o sistema implementado deve ser flexível e facilmente extensível pelo usuário, trazendo opções que permitam e facilitem a adição de novos pareamentos, ações e métodos de arbitragem. Para auxiliar nesse processo, é desejado fornecer alguns exemplos para cada tipo de recurso, os quais servirão de modelo e ainda poderão ser utilizados como ferramentas prontas.

Outra funcionalidade que é extremamente poderosa e desejamos incluir no *framework* é o sistema de unificação, o qual permitirá o uso de **variáveis** na construção das regras. A justificativa para a inclusão desse subsistema complexo é a possível abundância de entidades em um *videogame*, algo que ocorre, por exemplo, em jogos do gênero *Bullet Hell*[2],

[2] Saiba mais sobre o gênero *Bullet Hell* em: https://en.wikipedia.org/wiki/Bullet_hell

ou *Hack and Slash*[3] com muitos inimigos. Nesses tipos de jogos, é possível haver centenas de objetos sendo criados e destruídos em um segundo, logo, provavelmente modelaremos regras que se aplicam a uma categoria de entidades e não a um objeto constante, algo que necessita do poder expressivo das variáveis (como explicado na Seção 1.2.3).

Uma consequência da adição de variáveis é que o SBR precisará procurar entidades que atendam à condição de cada regra — em vez de apenas testar alguém fixo —, mas como foi dito anteriormente, é possível haver centenas de objetos para serem verificados, o que prejudicaria gravemente o desempenho do sistema. Dessa forma, é preciso haver uma maneira de filtrar possíveis candidatos para cada regra, algo que pode ser feito usando os Grupos de nós, discutidos na Seção 1.3.1.

Por fim, como a métrica do projeto é fornecer funcionalidades parecidas com as presentes nos jogos que serviram de motivação, é interessante que o arcabouço conte com **interface gráfica**, para permitir a visualização do conjunto de regras sendo editado. A principal motivação para essa *feature* é que a elaboração da inteligência artificial de personagens costuma ser responsabilidade de um *game designer*, alguém que não possui necessariamente experiência em programação, portanto, uma ferramenta gráfica facilitaria sua edição e permitiria a criação de comportamentos interessantes, como os dos exemplos.

2.2 Requisitos técnicos

O *plugin* será desenvolvido para a versão *4.1.+* do motor de jogos *Godot*, sendo que queremos garantir compatibilidade com qualquer versão seguinte dentro do mesmo lançamento. Ele usará o sistema de *plugins* explicado na Seção 1.3.4 e respeitará as especificações explicadas na documentação oficial (Linietsky *et al.*, 2014), seguindo os tutoriais sobre como construir e configurar complementos para a *Godot*.

Além disso, ele será feito para ser compatível com o ***gd-plug***[4], um gerenciador de *plugins* minimalista disponível na *Godot Asset Library*[5] que permite a instalação automática e o controle de versionamento de complementos para *Godot*. A razão para incluirmos suporte a esse gerenciador vem do nosso objetivo de tornar a tecnologia de SBR o mais acessível possível, o que agora também significa facilitar a instalação do *plugin*, principalmente de uma forma que seja natural a desenvolvedores que utilizam a *engine*.

Dessa forma, serão seguidos os passos necessários para garantir que a interação com o *gd-plug* seja o mais fácil possível: hospedar o código em um repositório aberto do *GitHub*; ter uma pasta `addons/` na raiz do repositório e colocar todo o necessário para o funcionamento do *plugin* em um subdiretório da pasta mencionada. Seguindo esses simples passos, nosso complemento poderá ser instalado e gerenciado de maneira prática e automática, por meio de outro *plugin* já conhecido na comunidade.

Prosseguindo, para o desenvolvimento do código foi escolhida a linguagem própria

[3] Saiba mais sobre o gênreo *Hack and Slash* em: https://pt.wikipedia.org/wiki/Hack_and_slash

[4] Acesse o repositório do *gd-plug* em: https://github.com/imjp94/gd-plug

[5] Acesse a biblioteca de recursos da *Godot* em: https://godotengine.org/asset-library/asset

do motor de jogos, ***GDScript*** (Linietsky *et al.*, 2014), mesmo havendo suporte para *C#*. Um dos motivos para essa escolha é que a linguagem nativa apresenta vários fatores que a tornam mais prática, tanto para a criação do *addon* quanto para sua utilização pelo desenvolvedor usuário, sendo alguns deles: uma sintaxe simples, semelhante à *Python*; tipagem estática opcional, de forma que é possível escolher entre *duck typing* e tipagem estática; açúcares sintáticos específicos do *Godot*, como `$Node` para `get_node("Node")`, ou `^"path"` para `NodePath("path")`.

Outra razão crítica é que o suporte para *C#* só está presente em uma versão alternativa da *Godot Engine*, chamada de *Mono*, a qual contém o ambiente de execução *.NET*[6]. Tanto a versão normal do Editor quanto a *Mono* apresentam suporte para *GDScript*, mas para utilizar códigos na outra linguagem é preciso ter a segunda opção, a qual é mais pesada e contém muitas funcionalidades que não serão usadas na maioria dos projetos. Portanto, usar *C#* iria impor aos usuários do complemento que instalassem a versão alternativa do motos de jogos, algo que consideramos inconveniente.

Ainda sobre inconveniência de uso, um dos requisitos do *plugin* é ser o menos invasivo possível ao projeto do usuário, o que significa que para ter um funcionamento correto, **não** deve ser necessário: **(i)** remanejar a hierarquia de pastas; **(ii)** alterar a estrutura de cenas, ou **(iii)** modificar as configurações do projeto. Em consequência, deverá haver uma forma de buscar automaticamente as adições feitas pelo usuário, sem depender de um diretório específico, e a posição do nó do sistema na cena deve ser arbitrária, de forma que ele possa controlar qualquer nó independentemente de estar em uma sub-cena instanciada, ou na árvore principal (mais sobre nós e cenas na Seção 1.3.1).

Em contrapartida, será considerado aceitável para um bom funcionamento que o desenvolvedor usuário necessite: **(i)** configurar manualmente gatilhos customizados para o acionamento do sistema; **(ii)** salvar e carregar recursos do disco quando quiser reutilizá-los; **(iii)** seguir um modelo para a implementação de novas classes e **(iv)** reiniciar o *plugin* ao acrescentar um novo recurso. Os dois primeiros itens se aplicam a um usuário comum, que usará o arcabouço como um sistema caixa-preta[7], enquanto os últimos dois se referem a pessoas que desejam estender suas funcionalidades.

Por fim, não deve ser necessário compilar nenhum componente do sistema em uma linguagem de programação à parte, nem instalar *software* adicionais, de forma que não haja requisitos para seu uso além do próprio motor de jogos. Em resumo, não queremos ter dependências de programas externos, pois assim temos a segurança de que nenhuma mudança em uma interface externa fará o sistema parar de funcionar, sendo o único potencial risco uma mudança na versão da própria *Godot Engine*.

2.3 Arquitetura e estruturas de dados

Sabendo as funcionalidades que precisam ser inclusas e as tecnologias que serão utilizadas, conseguimos elaborar a arquitetura de alto nível do *framework*, especificando

[6] Saiba mais sobre a estrutura *.NET* no site oficial: https://dotnet.microsoft.com/pt-br/

[7] Sistema definido apenas em termos de suas entradas e saídas, sem especificação de sua implementação interna. Leia mais em: https://pt.wikipedia.org/wiki/Caixa_preta_(teoria_dos_sistemas)

como cada elemento definido na Seção 1.2 será representado e como os requisitos serão cumpridos. Um fator importante que irá influenciar na arquitetura e que não foi mencionado até então é o fato da *Godot* ser Orientada a Objetos, o que significa que o sistema será organizado em classes, com heranças e composições.

Dessa forma, serão utilizadas técnicas de modelagem típicas de Programação Orientada a Objetos (POO), em particular, os padrões de *design* de *software* orientado a objetos propostos inicialmente no livro de Gamma *et al.* (1994). Para simplificar sua menção, iremos referenciá-los como padrões do "*GoF*", acrônimo de *Gang of Four*, em referência aos quatro autores do livro original.

Dito isso, a arquitetura do sistema está condensada no diagrama de classes presente na Figura 2.1, o qual utiliza a *Unified Modeling Language* (*UML*[8]) para denotar as relações entre as classes. Em uma primeira observação, uma característica que se destaca por diferir da modelagem típica de Orientação a Objetos é a ausência de interfaces, algo que foi consequência da escolha pela linguagem *GDScript*, pois ela não oferece essa funcionalidade, porém, isso não nos impede de aplicar os padrões de projeto do *GoF*: basta substituirmos as interfaces por classes abstratas.

Observando a figura mais a fundo, vemos que no ponto de conexão entre o Editor nativo da *Godot* e nosso *plugin* há o uso do padrão arquitetural *Model-View-Controller*, ou ***MVC***, no qual há: um Modelo, o objeto que representa a estrutura de dados central do projeto e contém a lógica de aplicação; uma ou mais formas de Visualização, correspondentes aos elementos visuais apresentados ao usuário; e um Controlador, o ger enciador de entrada do usuário que faz a ponte entre as interações nas Visualizações e os dados no Modelo. Repare que cada componente é altamente coeso, tendo responsabilidades bem definidas e que não se intersectam.

No nosso caso, o Controlador será composto pelos *scripts* centrais do *plugin*, responsáveis por declarar os novos tipos de estruturas e requisitar ao Editor que a interface gráfica seja adicionada à tela de edição dos projetos. A *Graphical User Interface* (*GUI*) corresponde à forma de Visualização e o Sistema Baseado em Regras será o Modelo, o objeto principal que contém os dados e sabe como gerenciá-los.

A grande vantagem desse padrão arquitetural é o isolamento da lógica de aplicação, pois ela é gerenciada apenas no Modelo e não é afetada pela maneira com a qual o usuário está interagindo com o sistema, seja com uma interface gráfica, via terminal ou por arquivos de configuração. Dessa forma, é possível adicionar novos Visualizadores ou Controladores sem precisarmos modificar o código da lógica central, ilustrando um baixo acoplamento entre as partes do sistema e permitindo grande flexibilidade e extensibilidade (Martin, 2017).

Prosseguindo, observamos que para a arbitragem de regras será aplicado o *design pattern* conhecido como **Estratégia**, no qual uma família de algoritmos é encapsulada em uma interface, ou classe abstrata, no nosso caso, de modo que cada subclasse possui uma implementação concreta e intercambiável do algoritmo (Gamma *et al.*, 1994). Com isso, teremos os árbitros como objetos que definem a estratégia de arbitragem para o SBR,

[8] Leia mais sobre *UML* em: https://pt.wikipedia.org/wiki/UML

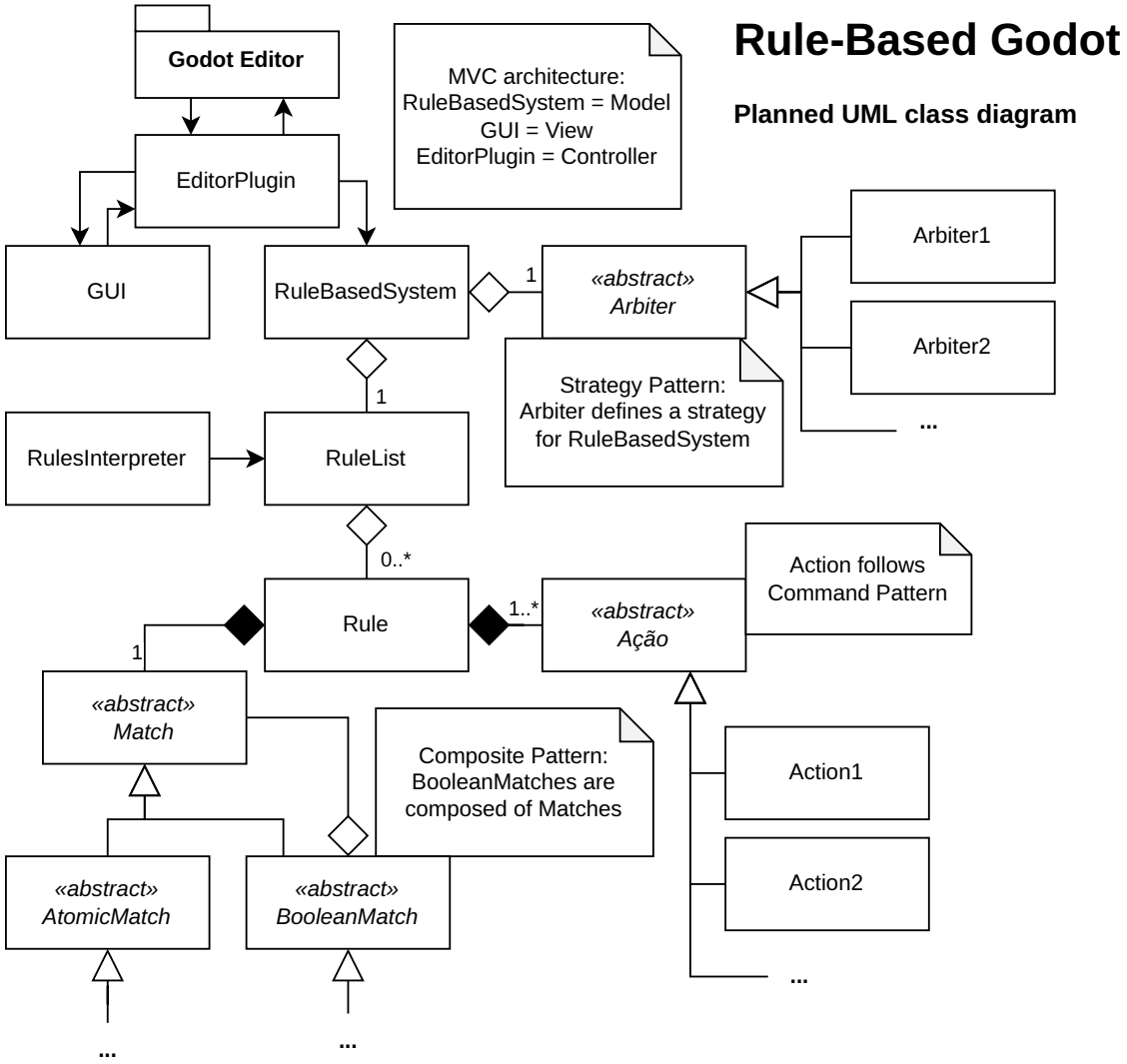

Figura 2.1: *Diagrama de classes planejadas para o Rule-Based Godot plugin (em inglês). Na parte superior esquerda, vemos o uso do padrão arquitetural MVC, implementado por meio das classes* `RuleBasedSystem` *(Modelo),* `GUI` *(Visualização) e* `EditorPlugin` *(Controlador). À direita, vemos que a classe de árbitro define uma Estratégia para o Sistema Baseado em Regras, seguindo o padrão do GoF (*Gamma et al., 1994*) de mesmo nome. No canto inferior direito temos as ações, as quais seguem o padrão Comando, também de GoF. Por fim, no canto inferior esquerdo notamos que os pareamentos podem se organizar em uma estrutura de árvore, com* `AtomicMatches` *sendo elementos unitários e* `BooleanMatches`*, elementos compostos. Isso segue o padrão Composite, de GoF.*

sendo possível alternar entre métodos simplesmente trocando o objeto.

Ademais, o sistema irá possuir uma lista ordenada de regras, organizada em ordem decrescente de prioridade, a qual será representada pela classe `RuleList`. O motivo de haver uma classe para isso, em vez de um simples *array*, é que a *Godot* possui uma forma nativa de persistir recursos em disco, mas isso exige que eles sejam representados por uma única classe. Assim, como queremos salvar os comportamentos e eles são definidos por mais de uma regra, precisamos centralizar a lista completa em um único recurso.

Também está prevista a criação de uma classe para gerar e interpretar as regras em um formato customizado, permitindo manipular esses recursos de uma maneira mais intuitiva dentro do nosso *addon*. Esse formato também poderá ser persistido em arquivos, mas sua abordagem será diferente do que foi dito anteriormente: em vez de representar um `Resource` da *Godot*, algo feito para ser usado pelo computador, a ideia é que ele modele as regras de uma maneira legível para um ser humano.

Agora, como um comportamento típico de uma personagem em um jogo exige várias regras, nosso formato customizado trabalha apenas com `RuleList`, o que significa que para salvar apenas uma regra será preciso criar uma lista unitária. Por fim, como dito na Seção 1.3.2, para implementar essa funcionalidade é preciso criar novas classes que herdem de superclasses abstratas próprias da *game engine*.

Olhando agora para a classe `Rule`, a unidade de informação mais importante para o sistema, vemos que ela será composta por instâncias de pareamento (ou *Match*, seguindo a nomenclatura de Millington e Funge, 2009) e por uma lista ordenada de `Actions`. Trataremos primeiro das ações, objetos que encapsulam e parametrizam uma alteração desejada ao mundo virtual do jogo, assim como o padrão do *GoF* chamado **Comando** (Gamma *et al.*, 1994), o qual, inclusive, também é conhecido como *Action*.

O fato de termos um *array* de ações é importante, pois a ordem com que aparecem nesse vetor será a mesma ordem de execução, assim garantindo que dependências entre as alterações propostas sejam respeitadas. De maneira semelhante à lista de regras, queremos que o usuário manipule a ordem dessa estrutura de dados de maneira simples, visto que isso tem grande impacto no comportamento gerado.

Agora, conforme dito na Seção 1.2.4, a condição de uma regra será composta por uma árvore de pareamentos, os quais são separados em dois tipos: compostos (predicados *booleanos*) e atômicos (chamados de *datum* no livro de Millington e Funge, 2009). Na Figura 2.1, isso é expresso por meio do padrão **Composto**, no qual podemos criar uma árvore de objetos que implementam a mesma interface, assim permitindo que o cliente interaja com a interface sem saber se está lidando com uma folha, ou com uma composição de um número arbitrário de objetos (Gamma *et al.*, 1994).

Em nossa arquitetura, o tipo abstrato *Match* representa o componente base, as subclasses de pareamentos *booleanos* serão compostos por outros *Matches* e os subtipos de *AtomicMatch* serão os componentes folha. Essa composição permite criar árvores de profundidade arbitrária, assim fornecendo todo o poder expressivo da lógica *booleana*. Um exemplo simples de como ficará a estrutura de uma condição está na Figura 2.2, na qual temos a árvore que representa: (**NÃO** *NumericMatch*) **E** (*StringMatch* **OU** *DistanceMatch* **OU** *HierarchyMatch*).

2.4 Experiência de usuário

Tendo a estrutura básica do sistema, podemos voltar nossa atenção para os usuários do produto para entender como adaptar as funcionalidades presentes e proporcionar a melhor experiência possível de usabilidade. Aqui estamos considerando que os requisitos apontados na Seção 2.1 já cobrem as necessidades básicas do usuário, portanto, nesse

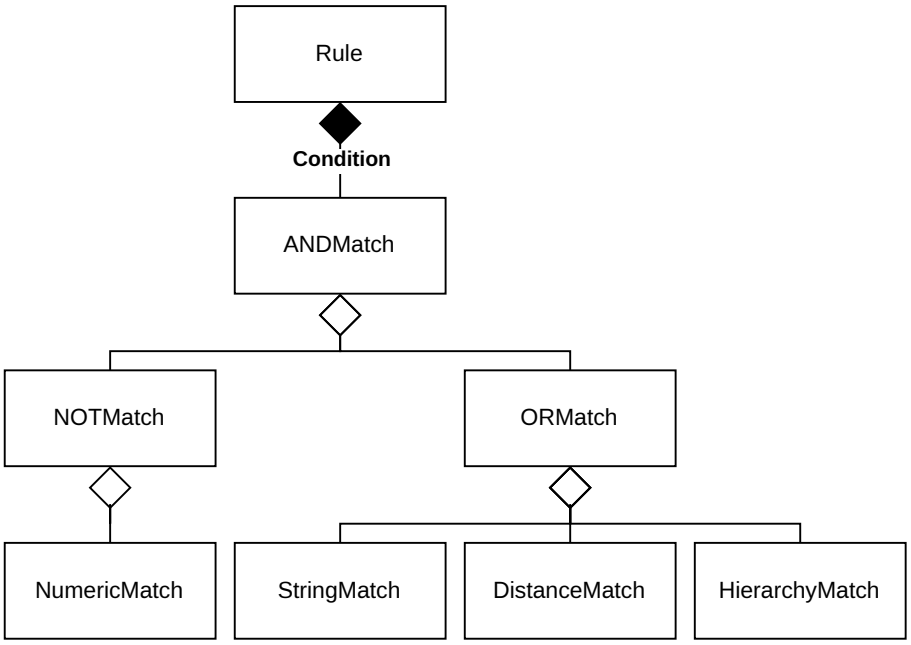

Figura 2.2: *Exemplo de condição como uma árvore de pareamentos. Os operadores booleanos (AND, NOT e OR) são os nós internos e os outros pareamentos são as folhas. Note que o* `NOTMatch` *possui apenas um filho, pois o operador NÃO é unário, enquanto* `ANDMatch` *possui dois e* `ORMatch`*, três. Esses dois últimos operadores possuem um número arbitrário de entradas (*Boole, 1854*).*

momento estamos apenas polindo as formas de interação com o sistema, com um foco maior em *User Experience* (*UX*).

O público-alvo do *plugin* será desenvolvedores de jogos utilizando a *Godot game engine*, portanto, podemos considerar que a usabilidade do sistema deve ser compatível e complementar à experiência de usuário já presente no Editor nativo. Em especial, devemos tentar integrar as novas funcionalidades da maneira menos aparente possível, utilizando os recursos que já são disponibilizados normalmente.

Para começar, a *GUI* deve utilizar os menus e painéis habituais, descritos na Seção 1.3.3, além de manter o mesmo estilo visual do resto do Editor, de tal forma que não seja imediatamente aparente se essa ferramenta é nativa, ou uma adição de um complemento. Vale lembrar que o sistema de *plugins* nos permite adicionar novas docas e novos painéis inferiores, ambos compostos por cenas comuns (vide Seção 1.3.4), logo, é fácil garantir que o *design* de nossa interface tenha o mesmo estilo familiar ao usuário.

Ainda sobre a interface gráfica, sabemos que será necessário atribuir variáveis para declarar as regras, portanto, uma funcionalidade extremamente desejável é mantermos os *widgets* para atribuir valores a variáveis de tipos nativos, visto que eles possuem uma usabilidade intuitiva e já são conhecidos pelos usuários da *Godot* (veja um exemplo na Figura 1.5). Desse modo, ao editarmos uma das novas classes em nossa interface, deverá haver uma maneira de acessar as suas variáveis por meio dos *widgets*.

Além dos aspectos visuais, é necessário que o usuário entenda a lógica por trás da declaração de regras, assim como o funcionamento básico de um SBR. Algo que pode

auxiliar muito nesse quesito são exemplos de uso, pequenas cenas que demonstrem a configuração necessária para o sistema funcionar e os efeitos que ele possui no mundo virtual. Por meio delas, será ensinado de maneira prática como usar o *framework* para criar comportamentos, além de demonstrar a jogabilidade de cenas criadas com ele.

Esses exemplos devem cobrir todos os recursos prontos que estarão disponíveis a quem baixou o complemento, servindo como uma espécie de tutorial, ou uma forma de documentação extra. As cenas da *Godot* contendo esses exemplos serão disponibilizadas no repositório do projeto[9], de forma que as pessoas possam baixá-las para aprender e testar o *software*, mas elas não estarão inclusas no lançamento da aplicação principal.

Tendo isso, poderemos dizer que experiência do usuário mais alto nível do *plugin* estará satisfeita. Consideramos que esse tipo de usuário utilizará os instrumentos do jeito que lhe foram fornecidos, como se o SBR fosse uma caixa-preta, o que significa que sua interação será restrita à configuração inicial e à declaração de regras via interface gráfica. Porém, precisamos nos preocupar também com o público que deseja estender as funcionalidades do *addon*.

Para atender às necessidades do usuário avançado, nossa *Application Programming Interface* (*API*) precisa ser estável e bem documentada, de preferência intuitiva o suficiente para não ser necessário adentrar no código-fonte toda vez que se deseja acrescentar algo. De qualquer forma, o principal artefato para auxiliar o desenvolvedor usuário será uma boa documentação[10], a qual deve mostrar os métodos e propriedades públicos de cada classe, além de como elas interagem entre si.

Ademais, é interessante que a documentação forneça o passo-a-passo para a criação de novas subclasses de pareamentos, ações e árbitros, os três tipos de recurso que especializam os comportamentos criados com o sistema. Como dito anteriormente, a versão final do projeto terá inclusos exemplos de cada tipo, mas é necessário ao menos explicá-los para que um desenvolvedor consiga entender os aspectos necessários ao funcionamento de cada classe.

Contudo, algo que deve preceder a documentação, e que devemos nos atentar durante o desenvolvimento, é planejar a estruturação das superclasses abstratas de tal forma que a implementação das subclasses concretas seja o mais simples possível. Se isso for atendido, o trabalho do usuário que deseja estender o complemento será muito mais fácil, além de auxiliar no melhoramento do próprio *plugin*, permitindo que novas funcionalidades sejam acrescentadas sem muito esforço.

Uma maneira de simplificar a criação de novos recursos é fornecer elementos pré-configurados que possam ser ativados ou desativados conforme a necessidade do desenvolvedor, o que irá aumentar a complexidade das classes-base, mas diminuirá a quantidade de código necessária para implementar algo novo. Assim, isso é algo que vale a pena ser analisado durante o desenvolvimento.

---

[9] As cenas de teste podem ser acessadas em: https://github.com/rvbatt/rule-based-godot/tree/main/test_scenes

[10] Leia a documentação em: https://github.com/rvbatt/rule-based-godot/blob/main/README.md

Capítulo 3

Desenvolvimento do sistema

Neste capítulo será feito um relato das principais etapas do processo de desenvolvimento, descrevendo o progresso do sistema ao longo de suas versões intermediárias. Essa descrição irá explicar como os requisitos levantados anteriormente foram cumpridos, dando as linhas gerais do que foi implementado, mas sem explicitar o código. Além do que já estava planejado, serão relatados os contratempos encontrados no meio do caminho, suas soluções e as novas funcionalidades que decidimos incluir, tendo em vista que a modelagem de um sistema de *software* é algo iterativo e que sofre alterações ao longo do desenvolvimento.

Talvez mais importante que os componentes implementados seja a justificativa para cada decisão tomada, visto que é possível acessar o repositório e ler tudo que foi desenvolvido, mas a motivação por trás das escolhas feitas não pode ser deduzida a partir do código. Dessa forma, sempre que houver mais de uma maneira de se implementar algo, justificaremos a que foi escolhida, assim como explicaremos o porquê de termos adicionado cada funcionalidade não prevista.

Para auxiliar na organização das resoluções feitas ao longo do projeto, iremos demarcar as **Decisões de projeto** de uma maneira especial, enumerando-as para serem facilmente referenciadas e escrevendo-as de uma forma que facilite a sua consulta posterior. Porém, ao invés de seguir a numeração habitual de teoremas e figuras, que utiliza o número do capítulo, um ponto e o índice do item dentro do capítulo, iremos descartar o primeiro identificador, pois sabemos que todas as Decisões serão declaradas somente neste capítulo.

Além disso, iremos destacar quando uma resolução é diretamente dependente de outra, por meio de uma notação do tipo: "*Decisão X.Y*", o que indica que esta é a Y-ésima Decisão que depende de algo declarado na Decisão X. Dessa forma, fica claro quais escolhas são independentes e podem ser consultadas em qualquer ordem e quais devem ser lidas em sequência.

Agora, conforme explicado na Metodologia deste trabalho, teremos as versões *Alpha* e *Beta* do projeto, porém, decidimos incluir uma Pré-*Alpha*, demarcando o momento em que temos a estrutura que define um Sistema Baseado em Regras, sem o sistema de unificação ou outros recursos adicionais. Para cada uma das versões, falaremos a etiqueta que as identifica no repositório, assim como qualquer *software* que seja requisito para

executá-la.

Por fim, vale destacar que todo o processo de desenvolvimento foi regido com base no estudo do material bibliográfico levantado. A inspiração para a implementação inicial do sistema veio principalmente de Millington e Funge (2009), com alguns detalhes que se basearam na abordagem diferente trazida por Bourg e Seemann (2004). Além disso, aplicamos os padrões de projeto de *GoF* (Gamma *et al.*, 1994), seguindo o que foi definido na Seção 2.3 e com alguns usos não previstos na fase de planejamento.

3.1 Versão Pré-*Alpha*

O primeiro passo no desenvolvimento do projeto foi a declaração do *RuleBasedSystem* como um novo tipo de nó reconhecido pela *Godot*, de forma que ele possa ser acrescentado a qualquer cena por meio da seleção no menu nativo do Editor. A Figura 3.1 ilustra a visão de um usuário acessando esse menu, no qual observamos a aparição do novo subtipo de *Node*, identificável pelo seu ícone de régua (*"ruler"*, em inglês, que é semelhante a *"rule"*, ou regra). Esse ícone foi encontrado sob a licença *CC0 1.0*[1] e depois modificado[2].

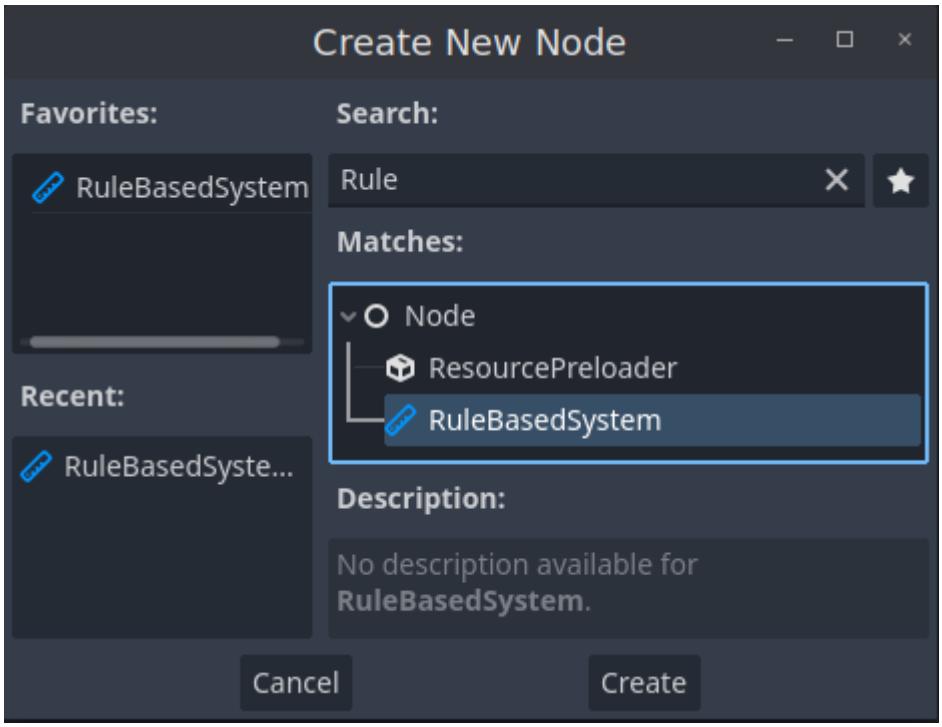

Figura 3.1: *Aparecimento do tipo RuleBasedSystem no menu de criação de nós, nativo da Godot 4.1.1. Esse tipo de nó customizado foi adicionado pelo plugin Rule-Based Godot. (versão Pré-Alpha).*

O principal motivo para definirmos o SBR como um tipo de nó é sua necessidade de acessar outros nós, para fazer os pareamentos e executar as ações, o que é feito por meio da função `get_node(path)`, implementada pela classe base `Node`. Esse acesso à árvore da

[1] Licença *Creative Commons* de domínio público, disponível em: https://creativecommons.org/publicdomain/zero/1.0/

[2] O ícone original se encontra em: https://seekicon.com/free-icon/ruler-round_1

cena é necessário para o sistema interagir com o mundo do jogo. Contudo, sabemos que ele pode ser custoso quando a árvore se torna profunda e que pode haver mudanças na hierarquia entre os nós durante a execução do jogo. Assim, devemos ter certo cuidado com os caminhos pré-definidos.

A seguir, foi preciso decidir como representar as estruturas de dados manipuladas pelo sistema, em particular, se elas seriam recursos ou novas cenas. Analisando as diferenças entre essas classes (mais detalhes na Seção 1.3.1), percebemos que o uso de sub-cenas iria gerar custos desnecessários, pois teríamos que buscar os componentes do sistema na estrutura de árvore da cena principal; além disso, seria mais difícil incluir uma interface gráfica que se integrasse ao editor de cenas.

Dessa forma, optamos por utilizar recursos para representar as classes acessórias ao Sistema Baseado em Regras, as quais serão explicitadas na Decisão 1 a seguir. A razão para o uso de recursos ser uma decisão de projeto importante é que o fluxo de trabalho habitual da *Godot* se baseia na manipulação de nós, já que eles são os blocos básicos do motor de jogos, assim, a utilização de outros tipos de classes é algo notável[3].

Decisão de projeto 1 (Uso de recursos). As classes `RuleList` e `Rule`, assim como as superclasses abstratas de árbitros, pareamentos e ações, serão todas subtipos de `Resource`, para garantir compatibilidade com a interface do Inspetor e facilitar a persistência e o carregamento de arquivos.

Conforme explicado na Seção 1.3.2, a principal vantagem do uso de recursos é que eles foram projetados para serem estruturas de dados, portanto, podemos nos aproveitar de facilidades prontas, como a opção de salvá-los e carregá-los como arquivos de forma automática, permitindo o compartilhamento de estruturas. Além disso, eles ocupam menos espaço na memória que nós e possuem compatibilidade nativa com o Inspetor, permitindo que suas propriedades sejam editadas com a interface gráfica disponível.

Dessa forma, a funcionalidade de interface gráfica, uma das mais importantes na concepção do *plugin* como um produto para usuários desenvolvedores, será resolvida basicamente com a utilização do Editor nativo. Esse aproveitamento do *Inspector* nos permitirá economizar o tempo que seria gasto montando o *design* de uma janela dedicada à declaração visual de regras, assim permitindo focar em outros pontos do sistema. Além disso, o fato de utilizarmos um menu nativo nos dá mais segurança de que ele não apresentará problemas de compatibilidade.

Portanto, registramos essa escolha como a Decisão de projeto 2, a qual toma proveito da Decisão 1, mas não é uma decorrência direta. Seria possível utilizamos recursos e não o Inspetor, ou vice-versa; por isso, é importante salientar esses dois pontos de maneira independente.

Decisão de projeto 2 (Uso do Inspetor). A interface gráfica (*GUI*) para declaração de regras será construída no Inspetor, para aproveitar as funcionalidades nativas como: *widgets* de atribuição de propriedades, agrupamentos de propriedades, sub-menus automáticos, entre outras.

---

[3] Tanto que a documentação apresenta uma página dedicada a alternativas para o uso de nós: https://docs.godotengine.org/en/4.1/tutorials/best_practices/node_alternatives.html

Assim, as regras devem ser declaradas por meio da criação e edição dos recursos no Inspetor, utilizando todas as facilidades explicadas na Seção 1.3.3. A Figura 3.2 ilustra a interface que aparece ao usuário, com marcações em amarelo que traduzem cada elemento relevante, além de apontar para os campos de edição de propriedades, em particular para um campo que abre o *widget* de seleção de cor, ilustrado na Figura 1.5.

Prosseguindo, podemos constatar pela imagem que foram implementados alguns exemplos de recursos — após criarmos as superclasses abstratas —, de tal forma que esses módulos reutilizáveis possam ser usados para demonstrar as funcionalidades do sistema. Os recursos elaborados foram:

- Pareamentos (sufixo *Match*):
  - *Booleanos* (operadores lógicos habituais):
    - `NOT`;
    - `AND`: possui número arbitrário de entradas;
    - `OR`: possui número arbitrário de entradas.
  - Atômicos:
    - `Numeric`: testa se um valor numérico, obtido a partir de uma propriedade ou chamada de método, está em um determinado intervalo;
    - `String`: testa se uma *string*, obtida a partir de uma propriedade ou chamada de método, é igual a uma constante definida;
    - `AreaDetection`: testa se há objetos em uma área específica, sendo possível determinar se queremos detectar objetos específicos, ou quaisquer.
- Ações (sufixo *Action*):
  - `SetProperty`: atribui um valor constante a uma variável;
  - `CallMethod`: chama um método, passando os argumentos definidos em um vetor.

Em contrapartida, é importante destacar que a classe `RuleList`, prevista no diagrama da Figura 2.1, ainda não foi criada nessa versão do sistema. Ela é temporariamente substituída por um *Array* de `Rules` no SBR, pois o processo de transformação de conjuntos de regras para arquivos e vice-versa será implementado depois. Além disso, não há a opção de alterar a arbitragem de regras, sendo aplicada a *First Applicable* diretamente na função de iteração do sistema.

Dessa forma, para utilizar o sistema nesse estágio do desenvolvimento é preciso: **(i)** criar um nó do tipo `RuleBasedSystem` na cena que será gerenciada; **(ii)** definir as regras, criando os recursos por meio do *Inspector* e **(iii)** chamar o método que testa as regras, no momento desejado. A versão Pré-*Alpha* do sistema pode ser acessada com a etiqueta `pre-alpha1` no repositório[4], não possuindo nenhum requisito além do Editor da *Godot* 4.1.1.

[4] Acesse a versão Pré-*Alpha* em: https://github.com/rvbatt/rule-based-godot/releases/tag/pre-alpha1

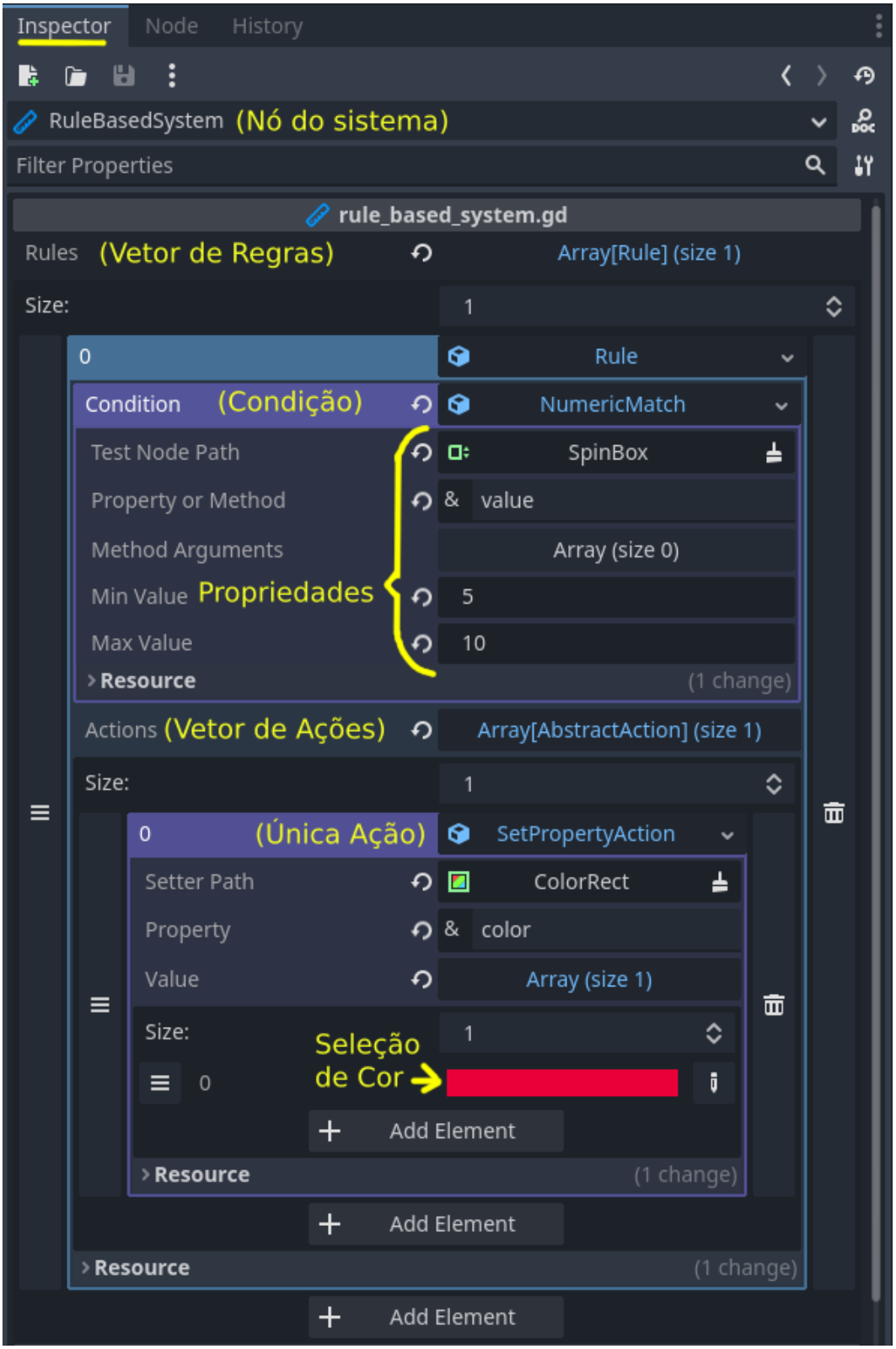

Figura 3.2: *Demonstração da interface gráfica para edição de regras (versão Pré-Alpha). Lendo as anotações de cima para baixo, vemos que a interface inteira está dentro do Inspetor, no qual está selecionado um nó do tipo* `RuleBasedSystem`*. Esse nó possui um vetor de regras como propriedade, sendo que cada regra é composta de uma condição e um vetor de ações. As propriedades da condição e das ações podem ser editadas normalmente no Inspetor, dando destaque para o campo de seleção de cor, que abre seu widget correspondente e fornece ao desenvolvedor executando o teste um feedback visual claro que a regra foi disparada corretamente.*

3.1.1 Cenas de teste

Para testar as funcionalidades presentes na primeira etapa do desenvolvimento, foram criadas algumas cenas de teste que procuram explorar, principalmente, os diferentes pareamentos. Para demonstrar as mudanças que as regras aplicam no mundo do jogo de uma forma clara e visual, foram colocados quadrados coloridos no canto das cenas (*Displays*) que mudam de cor quando uma regra é acionada.

No primeiro teste, verificamos se a lógica implementada pelos pareamentos *booleanos* segue o Teorema de *DeMorgan* (Boole, 1854), dado como Teorema 1. A condição da regra testada, ilustrada na Figura 3.3, representa a expressão lógica: ¬(¬(Line1=red) ∨ ¬(Line2=yellow)), sendo que os componentes nomeados "*LineX=color*" representam um `StringMatch` no qual o texto presente na linha X deve ser igual à palavra *color*.

Teorema 1 (DeMorgan). *Seja* $\langle A, \vee, \wedge, \neg, \bot, \top \rangle$ *uma álgebra booleana e tome* $x, y \in A$. *Então,* $\neg(x \vee y)$ *é equivalente a* $\neg x \wedge \neg y$ *e* $\neg(x \wedge y)$ *é equivalente a* $\neg x \vee \neg y$.

Figura 3.3: *Inspetor contendo condição que testa o Teorema de DeMorgan (versão Pré-Alpha). O predicado ilustrado corresponde a:* $\neg(\neg(Line1{=}red) \vee \neg(Line2{=}yellow)) = (Line1{=}red \wedge Line2{=}yellow)$.

Aplicando as regras de *DeMorgan*, essa condição é equivalente a: (Line1=red ∧ Line2=yellow), ou seja, a primeira linha deve possuir a *string* "*red*" e a segunda, "*yellow*". Quando isso ocorre, a regra é disparada e o retângulo no centro da tela é pintado de laranja, a combinação das duas cores, o que pode ser visto na Figura 3.4.

Figura 3.4: *Estado final da cena que testa o Teorema de DeMorgan (versão Pré-Alpha). A condição exigia que o texto da linha de cima fosse "red" e o da de baixo fosse "yellow", de forma a disparar a ação que altera a cor do retângulo no centro da tela para laranja.*

Ademais, para testar o pareamento de detecção de área foram criadas duas cenas, pois esse mesmo *match* foi feito para funcionar tanto com áreas 2D, quanto 3D, e não seria prático construir um experimento que juntasse elementos desses dois espaços em uma só cena. Assim, para cada teste foi montado um componente de área móvel controlado por entradas do teclado e colocou-se um objeto de cada tipo detectável pelo nó de área específico, algo que varia entre os espaços bi e tridimensionais.

Com isso, é possível mover a área até cada elemento e observar se o *display* no canto da tela é pintado com a cor correspondente, indicando que a regra correta foi acionada. A Figura 3.5 demonstra a cena 2D, na qual a área móvel está detectando um *TileMap* de cor roxa e, portanto, acionando a ação que colore o *display* com a mesma cor; já na Figura 3.6, temos a versão 3D, com a detecção de um *GridMap* (versão em três dimensões de um *TileMap*) e a coloração correspondente do quadrado no canto da tela.

O propósito desses testes era confirmar que todos os objetos detectáveis pelas funções nativas são reconhecidos pelo pareamento. Além disso, eles representam um primeiro exemplo de jogabilidade que utiliza o Sistema Baseado em Regras do *plugin*, demonstrando a integração entre uma ação comum de um jogo, como mover um personagem, e o acionamento de regras que leem essa ação e reagem a ela, conforme foram codificadas.

Para demonstrar a configuração do Inspetor nesses testes com detecção de área, observe na Figura 3.7 a regra responsável por detectar a colisão da área móvel 2D com o *TileMap* e com isso colorir o *Display* com a cor roxa. Na figura vemos que essa regra foi nomeada, por meio da edição do campo *Name* no agrupamento *Resource*, algo que é muito útil na organização da árvore de recursos no Inspetor. Essa é a regra que foi acionada no exemplo de execução da Figura 3.5.

Figura 3.5: *Cena de teste para detecção de área 2D (versão Pré-Alpha). A Área móvel destacada na imagem é controlada por teclado, de forma que quando ela colide com algum dos objetos colocados na cena, uma regra é disparada e o Display passa a ter a mesma cor do objeto detectado.*

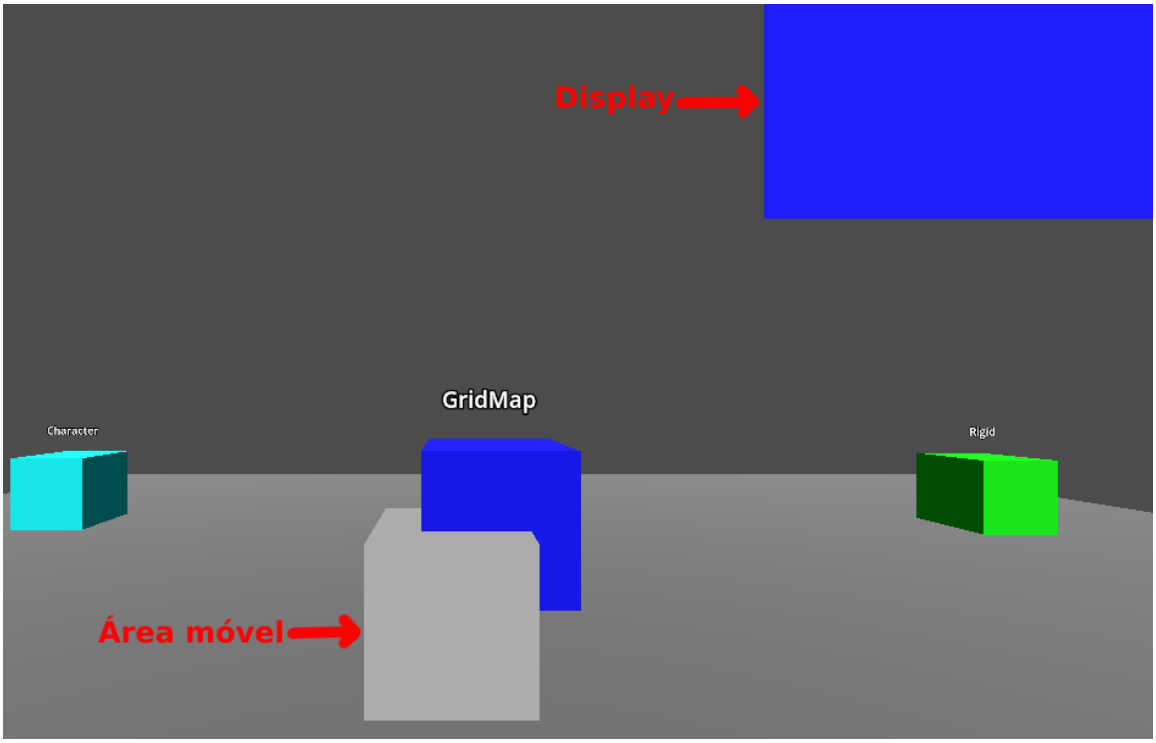

Figura 3.6: *Cena de teste para detecção de área 3D, semelhante à Figura 3.5 (versão Pré-Alpha). A Área móvel tridimensional demarcada é novamente controlada por teclado, de forma que quando ela colide com algum dos objetos em cena, a regra correspondente dispara e colore o Display no canto da tela com a mesma cor do nó detectado.*

Por fim, a configuração para o teste da verificação numérica já foi exibida na Figura 3.2, portanto, podemos observar agora as atribuições feitos no Inspetor em vez das anotações

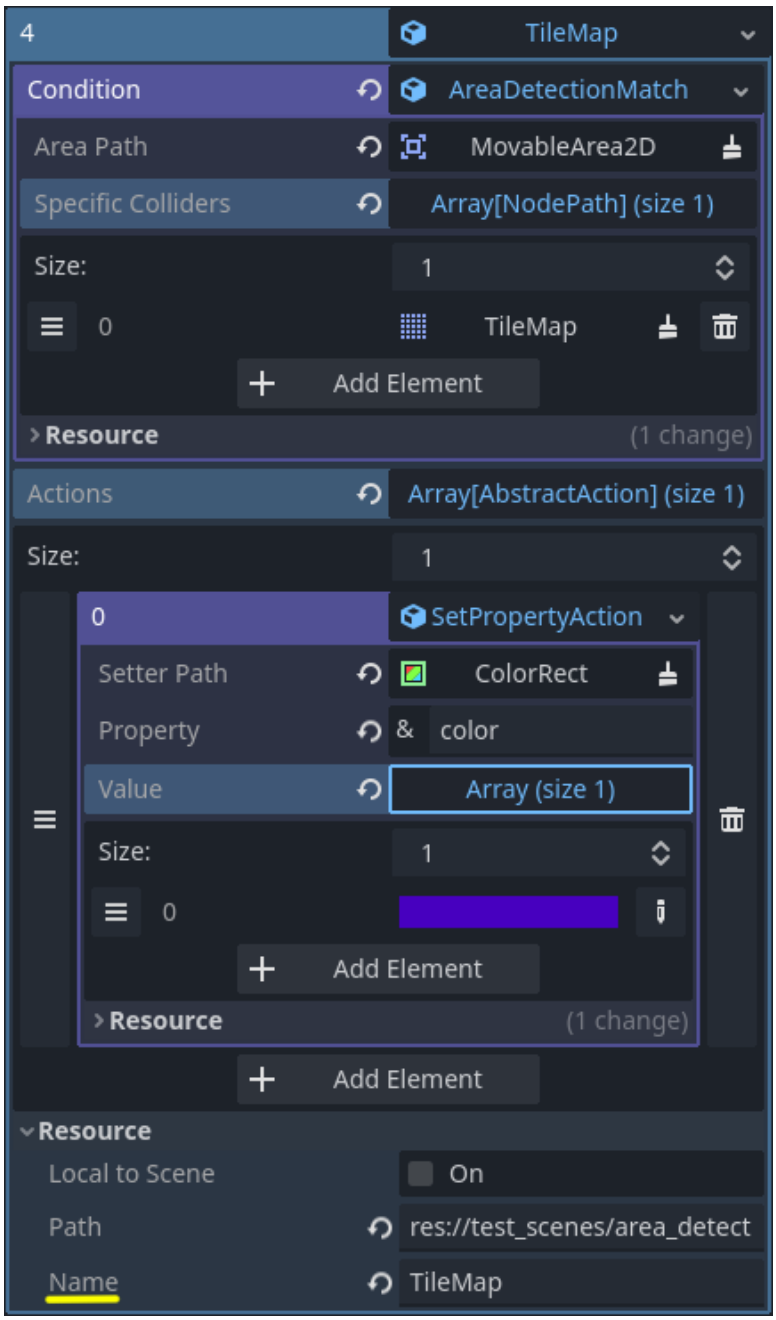

Figura 3.7: *Configuração da regra de colisão com TileMap presente no teste de detecção de área 2D, visto na Figura 3.5 (versão Pré-Alpha). A regra foi nomeada como "TileMap", por meio da propriedade* `resource_name`*. Repare que a condição é uma* `AreaDetectionMatch`*, que opera sobre a área MovableArea2D e detecta especificamente o nó do TileMap. A ação associada altera a propriedade color do ColorRect, o nó setter, atribuindo o valor igual à cor roxa.*

em amarelo. Como se trata de um tipo de pareamento muito simples, que apenas confere se um determinado número está em um intervalo, não há necessidade de ilustrar a cena, apenas reparar que a propriedade *value* do nó *SpinBox* deve pertencer a $[5, 10]$ para a condição ser satisfeita.

3.2 Versão *Alpha*

O primeiro avanço entre a versão pré e a *Alpha* foi a implementação do padrão Estratégia (Gamma *et al.*, 1994) para a arbitragem de regras, conforme estipulado na arquitetura presente na Figura 2.1. Feito isso, foi aberta a possibilidade de criarmos um exemplo de árbitro concreto, assim completando a lista dos tipos de recurso que podem ser adicionados pelo usuário, a qual recebeu as seguintes atualizações nessa versão do sistema:

- Árbitros (sufixo *Arbiter*), ambos explicados na Seção 1.2.4:
  - `FirstApplicable`;
  - `LeastRecentlyUsed`.
- Pareamento Atômico (sufixo *Match*)
  - `Distance`: testa se a distância entre a origem de dois nós está em um dado intervalo.
- Ação (sufixo *Action*):
  - `EmitSignal`: adiciona um sinal a um nó, caso ele não o possua, e então o emite.

3.2.1 Sistema de unificação

Agora, a principal funcionalidade acrescentada durante o segundo ciclo de desenvolvimento foi o **sistema de unificação**, o qual foi incorporado à execução normal de nosso SBR por meio da manutenção de uma estrutura de dados e da adaptação de certas funções. Para começar, a estrutura que contém as informações necessárias para o processo de unificação é uma tabela de símbolos, também chamada de dicionário, incluída na classe `Rule` com o nome `bindings`. Ela guarda os vínculos entre o nome de uma variável e uma lista de possíveis candidatos para seu valor.

Tendo isso, podemos tratar das etapas que ocorrem durante a execução do sistema: tudo começa quando solicitamos à classe `Rule` que verifique se sua condição é satisfeita, o que é transformado em uma chamada do método `is_satisfied(bindings)` no pareamento raiz da árvore de condições. Por meio dessa função, é passada uma referência para o dicionário alocado no objeto da regra. Caso haja conectores *booleanos* na condição, eles irão chamar o mesmo método em seus pareamentos filhos, transmitindo novamente a referência para a tabela de símbolos.

Como apenas os pareamentos atômicos precisam lidar com os vínculos, considere que a sequência de chamadas alcançou uma das folhas da árvore. Nesse momento, é preciso analisar os parâmetros do *match* que podem ser variáveis (o que nessa etapa do projeto se restringe a nós): se forem apenas valores constantes, usamos os que estão definidos e devolvemos se eles cumprem o requisito ou não (verdadeiro ou falso), mas caso algum deles esteja configurado como *wildcard*, precisamos aplicar unificação, o que se inicia verificando no dicionário os seus possíveis candidatos.

Perceba que o primeiro pareamento a usar uma variável não encontrará uma entrada no dicionário com tal nome, portanto, devemos ter uma forma padrão de selecionar a

primeira leva de candidatos, o que no nosso caso será feito olhando apenas os nós filhos do SBR. Continuando o caso em que temos pelo menos uma variável, o *match* atual irá percorrer a lista de candidatos e selecionar apenas aqueles que satisfazem seu próprio requisito, para que, ao final desse processo, a entrada correspondente no dicionário seja atualizada (ou criada) com a nova lista.

Dessa forma, estamos garantindo que toda variável chave na tabela de símbolos aponta apenas para valores que cumprem todos os pareamentos que a utilizam, considerando os observados até agora. Isso significa que após testarmos toda a condição, teremos uma lista com valores que podem ser unificados, ou seja, podem substituir todas as ocorrências de sua variável. Somente nesse momento poderemos responder à pergunta inicial sobre a regra: caso a lista de candidatos de alguma variável esteja vazia, a regra não é satisfeita, pois não há substituições que a tornem verdadeira; caso contrário, ela é satisfeita.

Algo muito importante de ser apontado é que o fato do dicionário ser uma propriedade da regra nos garante que o escopo das variáveis também é restrito à regra na qual elas aparecem. Dessa forma, é possível manter essa propriedade como privada, já que as referências necessárias são passadas por meio de chamadas de funções. Isso segue as especificações dadas na Seção 1.2.3 e nos permite usar variáveis de mesmo nome em regras diferentes, sem que elas interfiram uma na outra.

Para ilustrar o processo de unificação sendo aplicado na condição de uma regra, será dado um exemplo simples, por meio de diagramas que representam os objetos em execução. As Figuras 3.8 a 3.13 contêm o passo-a-passo da chamada da função `condition_satisfied()`, que chega à classe `Rule` e então é repassada para os pareamentos que compões sua condição. Seguindo esses passos, vemos o momento em que cada componente entra em execução (quando é pintado de vermelho) e temos o pseudocódigo da chamada de seu método (bloco *Execution*), além da indicação de referências presentes em tempo de execução (setas pontilhadas).

O resto das etapas de unificação ocorre quando uma regra é acionada, o que sempre sucede à verificação de sua condição e, portanto, acontece após os candidatos serem comprovados como substituições válidas. Ao chamarmos a função `trigger(bindings)` de cada ação no vetor da classe `Rule`, estamos novamente passando o dicionário de variáveis e valores como referência, assim permitindo acesso às atribuições feitas durante a verificação da condição.

Desta maneira, cada ação poderá verificar se algum de seus parâmetros é um *wildcard* ou não e, no caso positivo, é possível procurar no dicionário por uma substituição. Como o sistema garante que somente regras satisfeitas são acionadas, sempre haverá um valor que pode ser utilizado, portanto, conseguiremos acionar todas as ações da regra escolhida, seja com valores fixos ou com uma substituição adequada. Vale destacar que nessa versão do sistema só aplicamos a ação no primeiro candidato, mas isso será corrigido posteriormente.

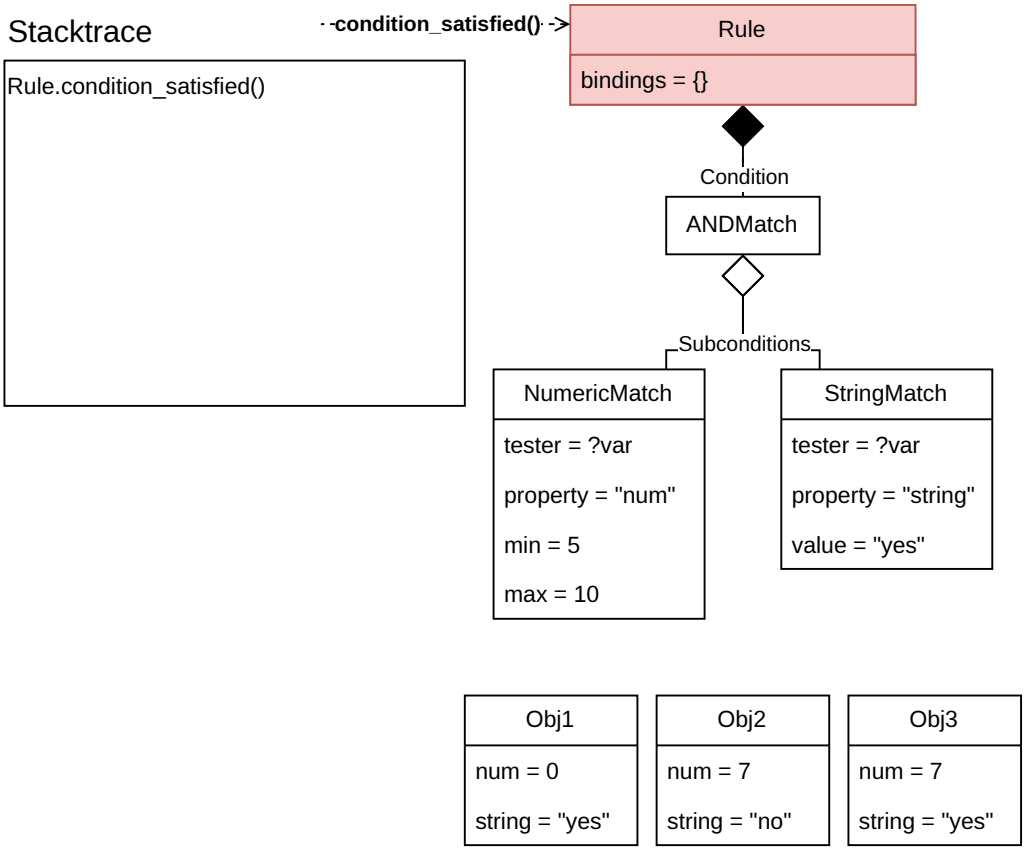

Figura 3.8: *Estado inicial do exemplo de unificação em tempo real (versão Alpha). Vemos uma instância de* `Rule` *recebendo a chamada do método* `condition_satisfied()`*. Sua condição é formada por um operador AND, que por sua vez aponta para dois pareamentos: um numérico e outro de string. Repare que ambos os pareamentos utilizam a mesma variável (?var) como o nó alvo do teste, o que significa que para a condição ser satisfeita, um mesmo objeto precisa satisfazer ambos os pareamentos. Há também três objetos em cena, numerados de* 1 *a* 3*, os quais possuem as mesmas propriedades verificadas na condição (num e string).*

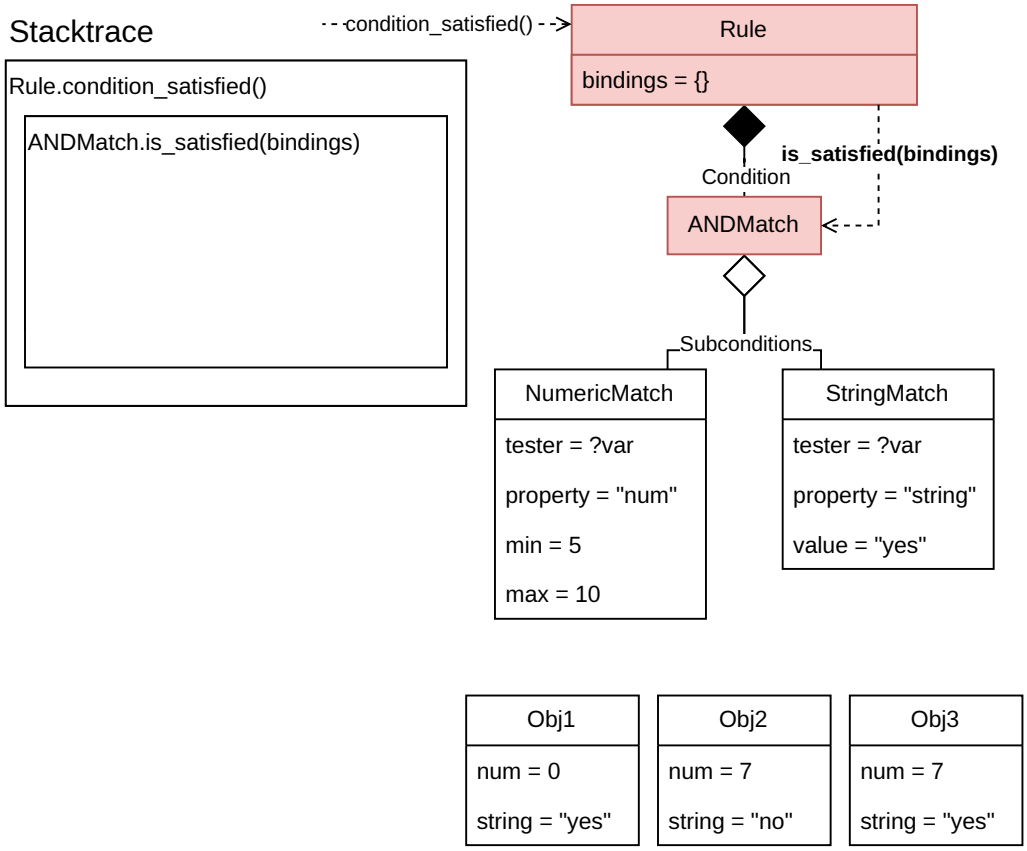

Figura 3.9: *Passo 1 do exemplo de unificação em tempo real (versão Alpha). A chamada de* `condition_satisfied()` *em* `Rule` *foi transformada em uma chamada de* `is_satisfied(bindings)` *no* `ANDMatch`*. É importante destacar que o dicionário está sendo passado por referência, o que significa que o pareamento booleano pode consultar e alterar diretamente a estrutura presente em* `Rule`*.*

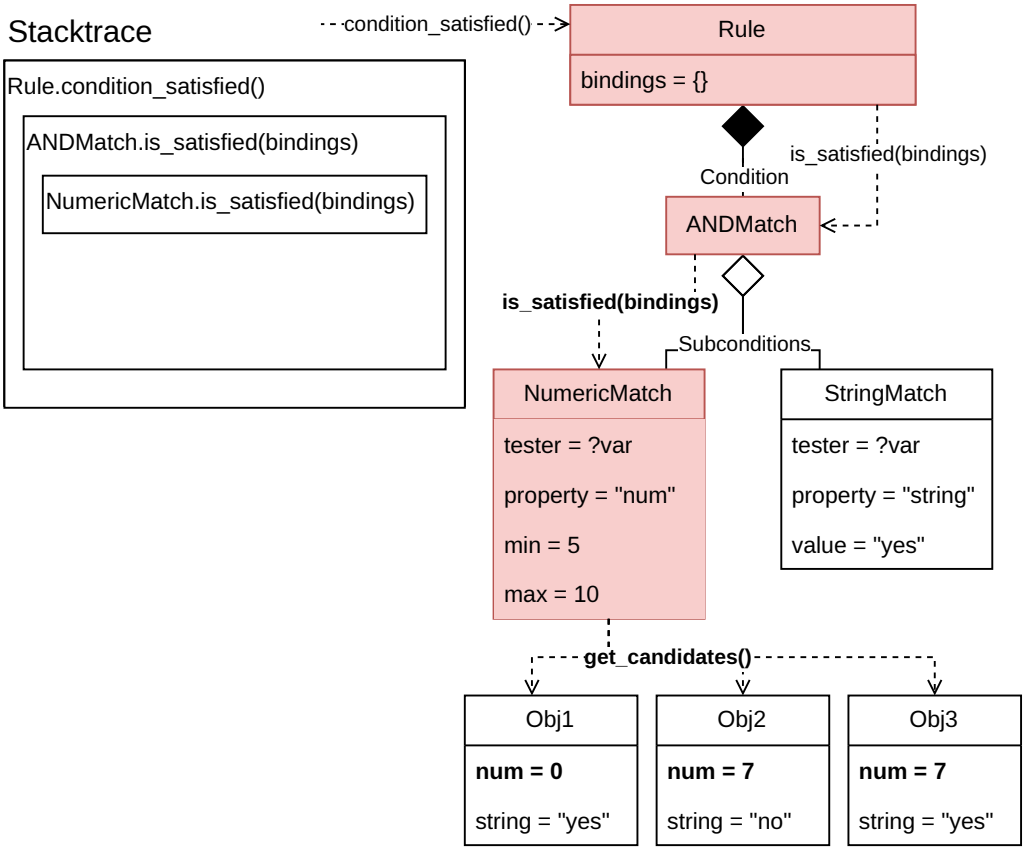

Figura 3.10: *Passo 2 do exemplo de unificação em tempo real (versão Alpha). A execução em* `ANDMatch` *cria uma chamada no objeto* `NumericMatch`*, passando novamente a referência para o dicionário* `bindings`*. Durante a execução do pareamento numérico, há uma chamada da* `get_candidates()`*, que procura por todos os objetos presentes em cena. A seguir, cada candidato é testado e somente aqueles com a propriedade num no intervalo* $[5, 10]$ *são aprovados. Assim, os objetos 2 e 3 passam no teste, portanto, é criado um vínculo entre eles e a variável ?var em* `bindings` *(veja na Figura 3.11). Como há pelo menos um objeto que satisfaz o pareamento, sua chamada irá devolver* `true`*.*

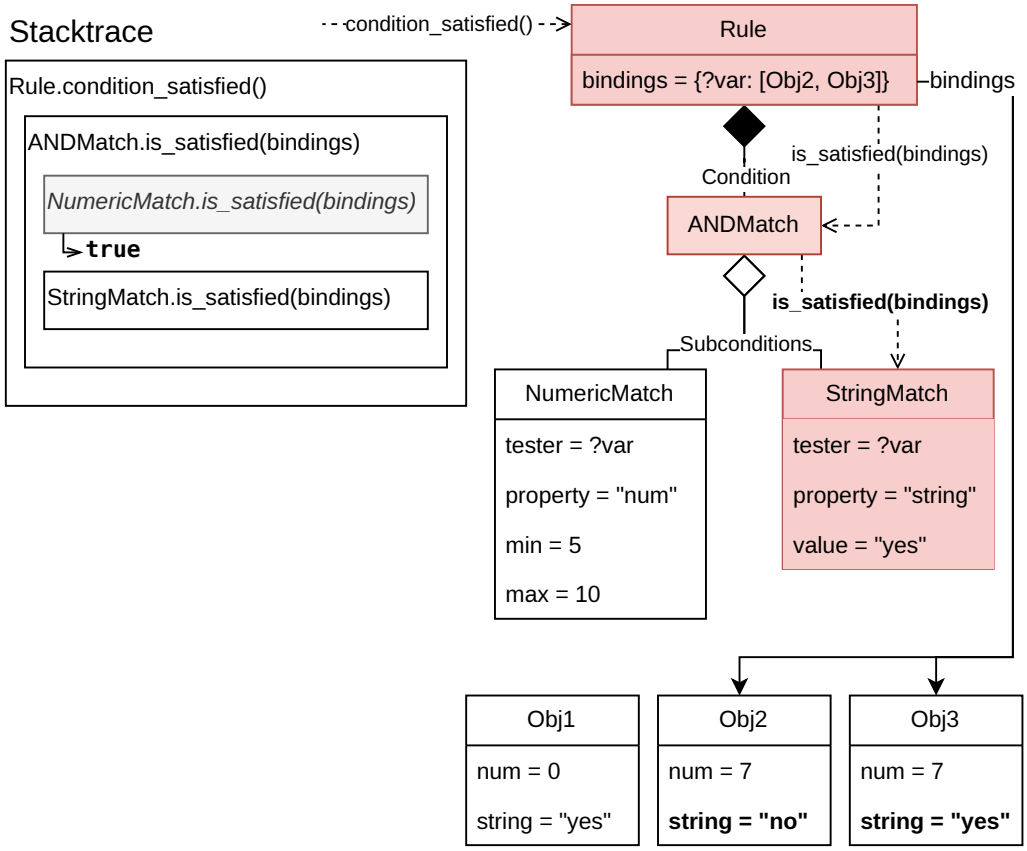

Figura 3.11: *Passo 3 do exemplo de unificação em tempo real (versão Alpha). Como* `NumericMatch` *encerrou e devolveu o valor "verdadeiro", o próximo passo da execução em* `ANDMatch` *invoca o pareamento de string. Agora, o dicionário* `bindings` *presente em* `Rule` *apresenta uma entrada para a variável ?var, portanto, os candidatos de* `StringMatch` *são adquiridos por meio dessa referência. Analisando a lista de objetos, foi verificado que a string de Obj2 não é igual a "yes", portanto, esse objeto não satisfez o pareamento em execução. Dessa forma, esse objeto é removido da entrada ?var do dicionário, pois ela deve conter apenas valores que podem substituir todas as ocorrências da variável. Porém, como o terceiro objeto passou no teste, a chamada aberta devolverá verdadeiro.*

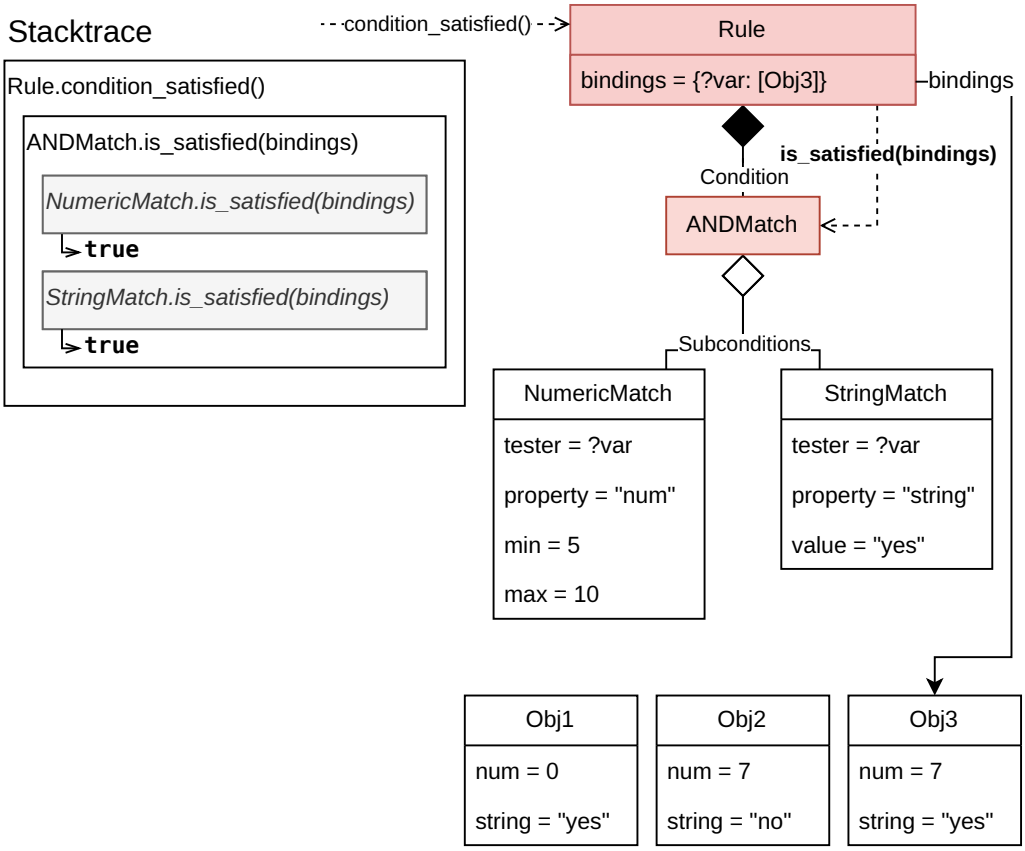

Figura 3.12: *Passo 4 do exemplo de unificação em tempo real (versão Alpha). Após as chamadas de* `Numeric` *e* `StringMatch`*, a execução à* `ANDMatch`*, no qual a única operação restante é devolver o valor de* $\texttt{true} \wedge \texttt{true} = \texttt{true}$*. Repare que o dicionário* `bindings` *na instância de* `Rule` *contém a chave ?var associada a um vetor que inclui somente o Obj3. Esse foi o resultado acumulado dos dois últimos passos: o passo 2, presente na Figura 3.10, criou a entrada com o valor [Obj2, Obj3] e o passo 3, visto na Figura 3.11, removeu o Obj3.*

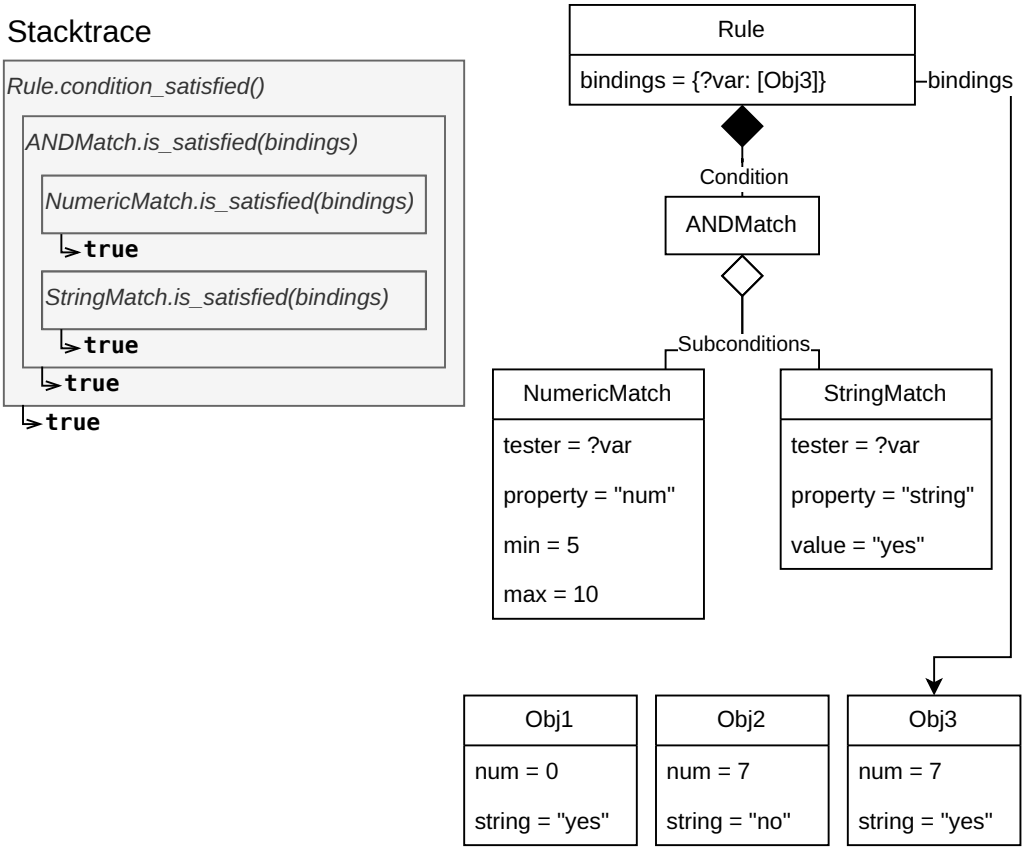

Figura 3.13: *Estado final do exemplo de unificação em tempo real (versão Alpha). Como* `ANDMatch` *devolveu* `true`*, a chamada de* `condition_satisfied()` *devolve o mesmo valor e é encerrada. Após isso, voltamos ao mesmo estado do mundo do jogo que vemos na Figura 3.8, com uma notável diferença: o dicionário* `bindings`*, no objeto* `Rule`*, agora contém uma associação entre a variável ?var e o Obj3. Essa associação está salva para poder ser utilizada pelas ações definidas na mesma regra, caso essa for escolhida para ser acionada.*

Tendo definido o processo de unificação, conseguimos notar que as subclasses de `AbstractMatch` e `AbstractAction` que desejem usar variáveis precisam lidar com vários fatores: a referência da tabela de símbolos, verificar quando um parâmetro é *wildcard* e gerenciar os candidatos. Porém, como o uso de variáveis é algo que queremos que todos os pareamentos e ações tenham suporte, o mais intuitivo é isolar esses fatores comuns nas classes abstratas e definir alguma forma dos subtipos concretos só implementarem o que é único a eles.

Para lidar com isso, foi decidido implementar o *design pattern* do *GoF* chamado de **Método *Template***, no qual definimos o esqueleto de um algoritmo em uma superclasse abstrata e delegamos alguns de seus passos em métodos abstratos, de forma que cada subclasse concreta precisa apenas definir esses passos individuais (Gamma *et al.*, 1994). Esse padrão se encaixa perfeitamente no nosso caso, nos permitindo lidar com as variáveis na parte genérica do modelo, enquanto os pareamentos e ações definem apenas os passos específicos de seu funcionamento único; portanto, optamos por tomar a Decisão de projeto 3:

Decisão de projeto 3 (*Template Method*). Todas as funções que lidam com variáveis usarão o Método *Template* (Gamma *et al.*, 1994) para isolar o seu gerenciamento da lógica principal.

Com isso, o padrão é aplicado no método `is_satisfied`, usado para gerenciar os nós testados pelos pareamentos e cujo pseudocódigo, em português, pode ser lido no Programa 3.1. Analisando esse código, vemos que a estrutura de *é_satisfeito(vínculos)* se encarrega de todas as questões derivadas do uso de variáveis, enquanto delega a lógica própria do que define se o pareamento é satisfeito ou não para o método abstrato *nó_satisfaz_pareamento* (que no sistema real tem o nome `node_satisfies_match`).

Programa 3.1 Classe `PareamentoAbstrato` com método `é_satisfeito(vínculos)`, seguindo o *Template Method*.

```
1   Classe ParamentoAbstrato
2
3   Propriedade nó_é_variável  ▷ Valor booleano
4   Propriedade identificador_nó  ▷ Identificador da variável, caso o nó seja um wildcard
5   Propriedade nó_fixado  ▷ Valor constante para o nó, caso não seja um wildcard
6
7   Metodo busque_candidatos()  ▷ Pode ser sobrescrito
8    ▷ Devolve uma lista de candidatos iniciais para o nó
9   Metodo nó_satisfaz_pareamento(nó)  ▷ Método Abstrato
10   ▷ Devolve verdadeiro se o nó satisfaz esse pareamento específico
11
12  Metodo é_satisfeito(vínculos)
13      se nó_é_variável
14          candidatos ← nulo
15          se vínculos.possui(identificador_nó)
16              candidatos ← vínculos.pegue(identificador_nó)
17          senao
18              candidatos ← busque_candidatos()
19          fim
20
21          candidatos_válidos ← Array()
22          paracada candidato em candidatos
23              se nó_satisfaz_pareamento(candidato)
24                  candidatos_válidos.anexe(candidato)
25              fim
26          fim
27
28          vínculos.defina(identificador_nó, candidatos_válidos)
29          devolva nao candidatos_válidos.está_vazio()
30      fim
31
32      devolva nó_satisfaz_pareamento(nó_fixado)
```

Agora, além do funcionamento interno do sistema de unificação, também é importante pensarmos na sua usabilidade, em como o usuário irá declarar e usar as variáveis. Seguindo

a mesma ideia do *Template Method*, decidimos isolar a manipulação das variáveis no Inspetor, o que foi feito por meio do agrupamento das propriedades de classe relacionadas a isso, equivalentes às mostradas no Programa 3.1. Além disso, como a distinção entre *wildcard* e valor constante altera quais propriedades são utilizadas, decidimos esconder dinamicamente as irrelevantes, conforme a configuração do usuário.

Na Figura 3.14 constatamos a combinação desses dois fatores: o agrupamento *Target Node* contém as propriedades relacionadas ao nó que deve ser testado por um pareamento, sendo a *Is Wildcard* a que controla o aparecimento das outras duas. Caso esteja ligada, o campo do identificador da variável é mostrado para o usuário digitar um nome, mas se for desligada, essa propriedade é substituída pelo caminho até o nó fixado. A praticidade dessa configuração dinâmica nos levou a tomar a Decisão de projeto 4:

Decisão de projeto 4 (Inspetor dinâmico). Modificaremos dinamicamente o Inspetor conforme configurações do usuário, para fornecer praticidade de uso.

Figura 3.14: *Demonstração do Inspetor dinâmico (versão Alpha). À esquerda, vemos que a propriedade Is Wildcard está ligada, fazendo aparecer o campo para definirmos o identificador da variável a ser usada. Já à direita, a propriedade foi desligada, escondendo o campo do identificador e revelando uma propriedade que representa o caminho de um nó fixado.*

Ademais, a versão *Alpha* do sistema também introduz uma nova funcionalidade que não estava inicialmente prevista: uma interface baseada em texto (*Text-based User Interface*, ou *TUI*). Trataremos a seguir dos detalhes sobre sua motivação e implementação, mas é válido ressaltar que esse evento demonstra como o planejamento de um sistema de *software* nunca será perfeito, pois não é possível prever as complicações e novas necessidades que serão descobertas durante o desenvolvimento.

3.2.2 Interface textual

Durante a elaboração desta versão do sistema, notou-se que a utilização do Inspetor para definir as regras traz algumas desvantagens, principalmente quando a árvore de condição é muito profunda, por ser preciso abrir e fechar muitos sub-menus. Sabemos que a implementação de uma *GUI* é muito importante para o *plugin*, porém, foi possível constatar por experiência própria que não é todo tipo de desenvolvedor que deseja utilizar essa forma de interação com o sistema.

Como dito na Introdução desta monografia, a interface gráfica é pensada principalmente para *game designers* que não têm conhecimento em programação. Todavia, desenvolvedores que trabalham majoritariamente com código provavelmente iriam preferir digitar as configurações que desejam, em vez de selecionar cada uma com o *mouse*. Isso porque quem passa a maior parte do tempo escrevendo código tende a digitar mais rápido, logo, uma interface que permita declarar regras por meio de palavras seria bem mais eficiente para essas pessoas.

Por conta disso, decidiu-se implementar uma interface textual, que forneça um editor de texto próprio para declaração de regras, de maneira semelhante ao que foi mostrado no exemplo da Seção 1.2.2. Isso está encapsulado na Decisão de projeto 5 a seguir:

Decisão de projeto 5 (Interface textual). O *plugin* contará com uma interface de usuário baseada em texto (*TUI*), para facilitar e acelerar a declaração de regras por usuários programadores.

O primeiro desafio técnico acarretado por essa decisão é que precisamos fazer nossas duas interfaces se comunicarem, o que significa fazer o *Inspector* receber as atualizações do editor de texto dedicado. Para resolver isso, foi preciso criar um complemento para o Inspetor, estendendo suas funcionalidades para fazer a conexão com nossa *TUI*. A *Godot* dá o nome de *InspectorPlugin* para esse tipo de adição, mesmo se tratando apenas de um *script*, na prática.

Com a parte técnica resolvida, foi preciso definir a sintaxe que será utilizada na nossa interface textual, pois isso influenciará tanto a sua implementação, quanto a usabilidade. Seria possível criar uma gramática própria, o que nos daria controle para definir a forma mais intuitiva possível de descrever as regras, nos aproximando da linguagem natural; porém, isso exigiria a criação de um interpretador, algo que está muito além do escopo deste trabalho.

Portanto, optamos por estudar linguagens já estabelecidas à procura de uma que possibilite a declaração de regras, sendo que desejamos uma sintaxe simples e, de preferência, o mais semelhante possível à usada nos exemplos. Essa procura nos levou ao formato *JSON*[5], o qual foi feito para ser natural de ler e escrever (por seres humanos), ao mesmo tempo em que é prático de gerar e interpretar (por máquinas).

Esse formato é utilizado para representar dados semi-estruturados, organizados em dicionários e vetores, ambas estruturas de dados que estão implementadas nativamente pela *Godot*. Além disso, o motor de jogos possui a classe `JSON`, um recurso feito especificamente para criar e interpretar dados nesse formato, algo que facilitará muito o processo de desenvolvimento. Em decorrência disso tudo, foi tomada a Decisão de projeto 5.1:

Decisão de projeto 5.1 (Formato *JSON*). A sintaxe da interface textual seguirá o formato *JSON*, para facilitar a interpretação das regras e permitir o uso da biblioteca nativa da *Godot*.

Em decorrência dessa decisão, a classe prevista para salvar e carregar as regras em algum formato customizado deverá lidar com arquivos `.json`. Porém, devido à forma como a *Godot* gerencia essa funcionalidade dos recursos, foi preciso implementar duas classes: `RulesFormatSaver`, que salva as regras em disco, e `RulesFormatLoader`, que as carrega. Outra adição à arquitetura foi a criação de um conversor da *string* formatada para o objeto do pareamento ou ação, o qual fica responsável por instanciar a subclasse certa e então configurar suas propriedades.

Finalizando a estruturação da sintaxe, um ponto específico que precisa ser estabelecido é como distinguir o identificador de uma variável de uma cadeia de caracteres qualquer,

[5] Leia mais sobre o formato *JSON* em: https://pt.wikipedia.org/wiki/JSON

visto que no formato *JSON* só existem os seguintes valores atômicos: números, *strings*, `true`, `false` e `null`; ou seja, precisaremos descrever o nome da variável como uma *string*. Para isso, decidimos adotar a mesma notação explicada na Seção 1.2.3: um ponto de interrogação no começo da cadeia de caracteres, o que irá nos levar à Decisão 5.2:

Decisão de projeto 5.2 (Prefixo para variável). O identificador de uma variável na interface textual será caracterizado por uma *string* cujo primeiro caractere é "?", para manter compatibilidade com a notação do livro de Millington e Funge (2009).

Com sintaxe definida, podemos implementar a *TUI*, que permitirá ao usuário digitar as regras que irão reger o comportamento de suas personagens. Isso foi feito por meio da adição de um painel inferior, conceito explicado na Seção 1.3.3, chamado de *Rules Editor*. Ele apresenta funcionalidades típicas de um editor de texto, como realce de sintaxe, *undo* e *redo*, indentação automática, entre outras.

Além disso, como cada pareamento e ação terá uma formatação específica que precisará ser seguida, resolvemos acrescentar botões que inserem um modelo de cada tipo de recurso. Dessa forma, o usuário pode escolher o que quer inserir e então substituir os campos dos valores. A adição desses modelos exigiu a elaboração de uma meta-linguagem para representar a ordem e o domínio de cada campo, seguindo as restrições da sintaxe adotada e sendo inspirada nas representações de gramáticas formais.

Vale destacar que criar essa meta-linguagem se mostrou muito mais simples do que desenvolver uma linguagem completa para representação de regras, especialmente porque não é preciso implementar um interpretador. As especificações completas dos modelos criados não serão dadas agora, pois elas passarão por alterações logo na próxima versão do sistema. Portanto, os detalhes serão explicados na Seção 3.3.1 e os resultados devem ser consultados no arquivo `README.md`, no repositório do projeto[6].

Dito isso, segue um exemplo simples de como a meta-linguagem representa o pareamento de distância entre dois nós, nesta etapa do desenvolvimento: o modelo *["Distance", min, max, "source_node", "?var|node"]* pode ser preenchido como *["Distance", 5, 10, "Mesa/Copo", "?obj"]*, assim representando o pareamento que verifica os objetos que estão à uma distância $5 \leq d \leq 10$ do Copo (cujo nó é filho da Mesa) e guarda todos os candidatos válidos na variável *obj*. Repare que a primeira *string* do vetor é um identificador para o tipo de pareamento, seguido das propriedades relevantes para essa verificação e, por fim, as opções que lidam com o uso de variáveis.

Concluindo, podemos visualizar a interface textual completa na Figura 3.15, na qual percebemos o realce de sintaxe, com as palavras-chave *Rules*, *if* e *then*, além do pareamento *Numeric* e da ação *CallMethod*. Também notamos os botões no canto inferior, em ordem da esquerda para direita: inserção de uma nova regra; inserção de um pareamento; inserção de uma ação; restaurar o texto inteiro; aplicar as regras no `RuleBasedSystem` atualmente selecionado.

A utilização do sistema nesta versão é praticamente igual à anterior, com a diferença que agora o usuário pode escolher entre criar os recursos no Inspetor, ou declarar as regras no *Rules Editor* e então apertar o botão *Apply* para aplicá-las no SBR selecionado

[6] Leia a documentação da sintaxe em: https://github.com/rvbatt/rule-based-godot/blob/main/README.md

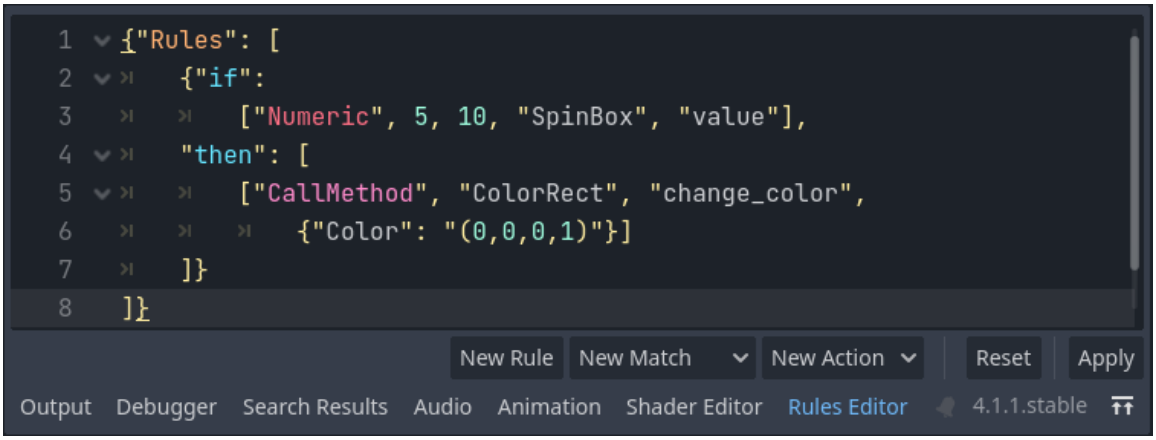

Figura 3.15: *Rules Editor, painel inferior com editor de texto para declaração de regras (versão Alpha). Palavras-chave são destacadas com uma cor diferente, como Rules em laranja ou CallMethod em rosa. Olhando para os botões na parte inferior, temos:* `New Rule`*, que insere o modelo de uma nova regra na posição do ponto de inserção de texto;* `New Match`*, que insere o modelo do tipo de pareamento escolhido no submenu pop-up;* `New Action`*, com comportamento similar ao botão anterior, mas com modelos de ações;* `Reset`*, que restaura todo o texto do editor para o esqueleto de uma regra vazia;* `Apply`*, que aplica as regras declaradas no SBR atualmente selecionado.*

no momento. Novamente, foram criadas cenas de teste para as novas funcionalidades e alguns testes anteriores foram atualizados. É possível acessar essa versão com a etiqueta `alpha`[7].

3.2.3 Cenas de teste

Os testes mais importantes acrescentados nessa versão do sistema são aqueles que comprovam o bom funcionamento do sistema de unificação, sendo que aqui serão destacados dois deles. O primeiro busca provar que a transmissão de informação entre o pareamento e a ação é bem sucedida, por meio de um pareamento que salva todos os nós que o satisfazem em uma variável e uma ação que executa um comando nos candidatos salvos nessa variável.

A declaração da regra usada nesse primeiro teste pode ser visualizada por meio das duas interfaces: com o *Rules Editor*, na Figura 3.16 e com o *Inspector*, na Figura 3.17. Interpretando qualquer uma delas, temos a seguinte definição: qualquer objeto que entra em uma certa *MovableArea2D* terá sua propriedade *color* configurada para a cor preta (representada pelo vetor $(0, 0, 0, 1)$ no código *RGBA*[8]). Esse comportamento é alcançado por meio da variável *obj_in_area*, pois, durante a verificação do pareamento, todos os objetos detectados pela área são colocados na lista apontada por essa variável e, depois, a ação aplica a atribuição de propriedade no primeiro da lista.

[7] Acesse diretamente a versão *Alpha* do *plugin* em: https://github.com/rvbatt/rule-based-godot/tree/alpha

[8] Leia mais sobre o formato *RGBA* em: https://en.wikipedia.org/wiki/RGBA_color_model

```
{"Rules": [
    {"if":
        ["AreaDetection", "MovableArea2D", "?obj_in_area"],
    "then": [
        ["SetProperty", "?obj_in_area", "color",
            {"Color": "(0,0,0,1)"}]
    ]}
]}
```

Figura 3.16: *Rules Editor com declaração textual da regra do teste de detecção de área wildcard (versão Aplha). Repare no uso da variável ?obj_in_area, que representa qualquer objeto que adentrou a MovableArea2D.*

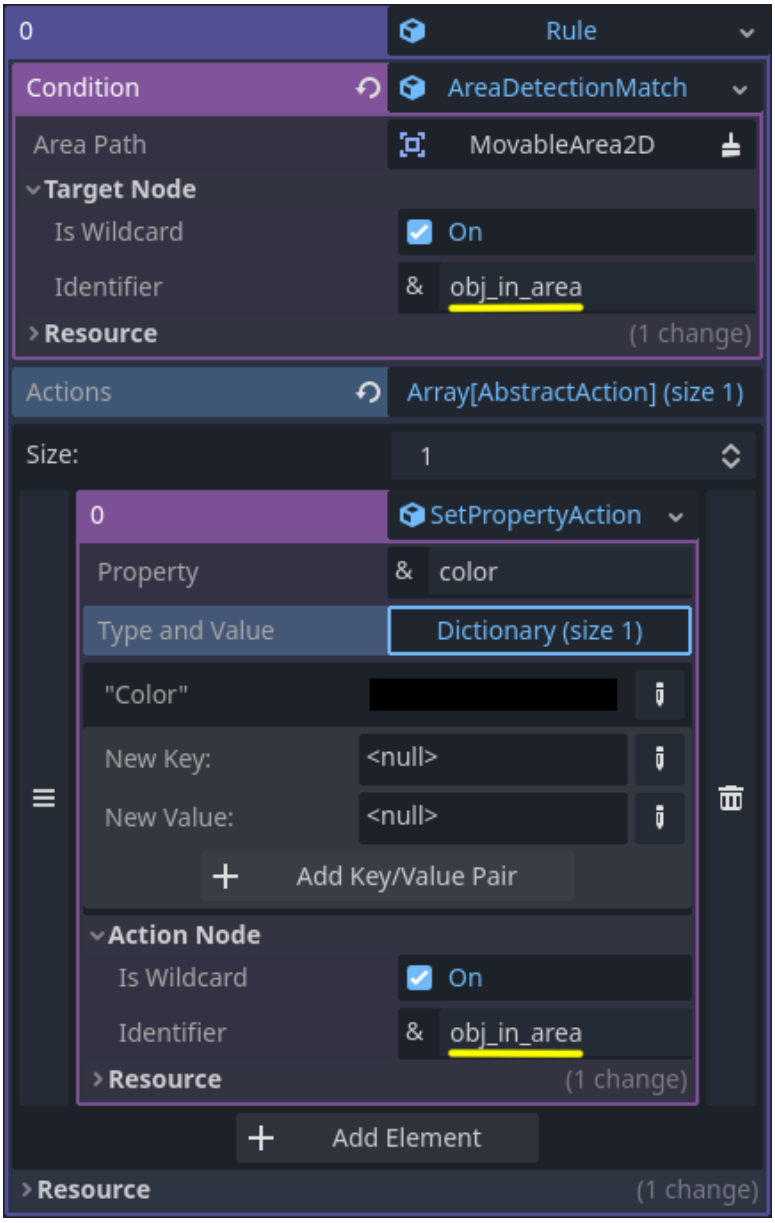

Figura 3.17: *Inspetor com regra do teste de detecção de área wildcard (versão Alpha). A regra declarada aqui de forma gráfica é equivalente à mostrada na Figura 3.16. Repare novamente no uso da variável ?obj_in_area.*

A cena em execução está demonstrada na Figura 3.18, na qual vemos a *MovableArea2D* encostando em *RigidBody* e *StaticBody*, assim acionando a regra que muda a cor do primeiro corpo detectado para preto. Vale relembrar que, nessa versão do sistema, só aplicamos a primeira substituição da variável, ou seja, a ação só é acionada no primeiro objeto que satisfez a condição. Caso a área móvel tivesse encostado somente no corpo estático, ele teria sido pintado de preto.

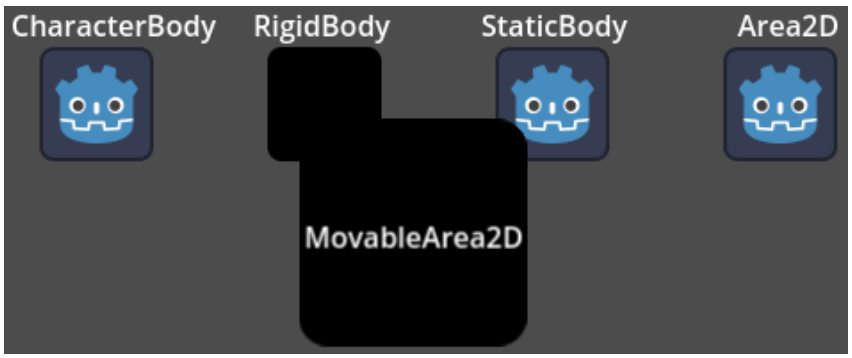

Figura 3.18: *Cena de teste para detecção de área wildcard (versão Alpha). É utilizada a regra demonstrada nas Figuras 3.16 e 3.17, de forma que a MovableArea2D detecta entidades em seu perímetro e faz a primeira que entrou nele mudar de cor para preto. Repare que RigidBody e StaticBody estão tocando a área móvel, mas como o corpo da esquerda foi o primeiro a ser detectado, ele é o único que sofre a mudança de cor.*

Prosseguindo, a segunda cena de teste trata da seleção de candidatos durante a verificação da condição, garantindo que somente aqueles que atendem a todos os pareamentos são escolhidos. Para testar isso, foram criados objetos com duas propriedades editáveis, uma numérica e outra de cadeia de caracteres, sendo definida uma condição composta no qual o mesmo objeto deve satisfazer certos requisitos para ambas as propriedades.

Os detalhes da declaração desse teste estão nas Figuras 3.19 e 3.20, nas quais vemos a atribuição dos recursos na *GUI* e sua codificação na *TUI*, respectivamente. Interpretando a regra mostrada, vemos que queremos encontrar um objeto com duas propriedades: a primeira chamada *number*, com um valor numérico no intervalo $[5, 10]$ e a segunda chamada *string*, com um valor igual a *"yes"*; os objetos que satisfazem essa condição são listados na variável *multiple*, para depois podermos aplicar duas ações neles: alterar o número para 100 e a *string* para *"Wildcard"*.

Finalmente, a execução da cena de teste está demonstrada na Figura 3.21, na qual temos três objetos demarcados por cores diferentes, cada um com um campo para a propriedade numérica e outro para a de cadeia de caracteres. A imagem da esquerda mostra a disposição inicial da cena, enquanto a da direita apresenta o resultado da aplicação das regras. Notamos que somente o objeto amarelo, o mais abaixo, foi modificado, pois somente ele atendia aos requisitos da regra.

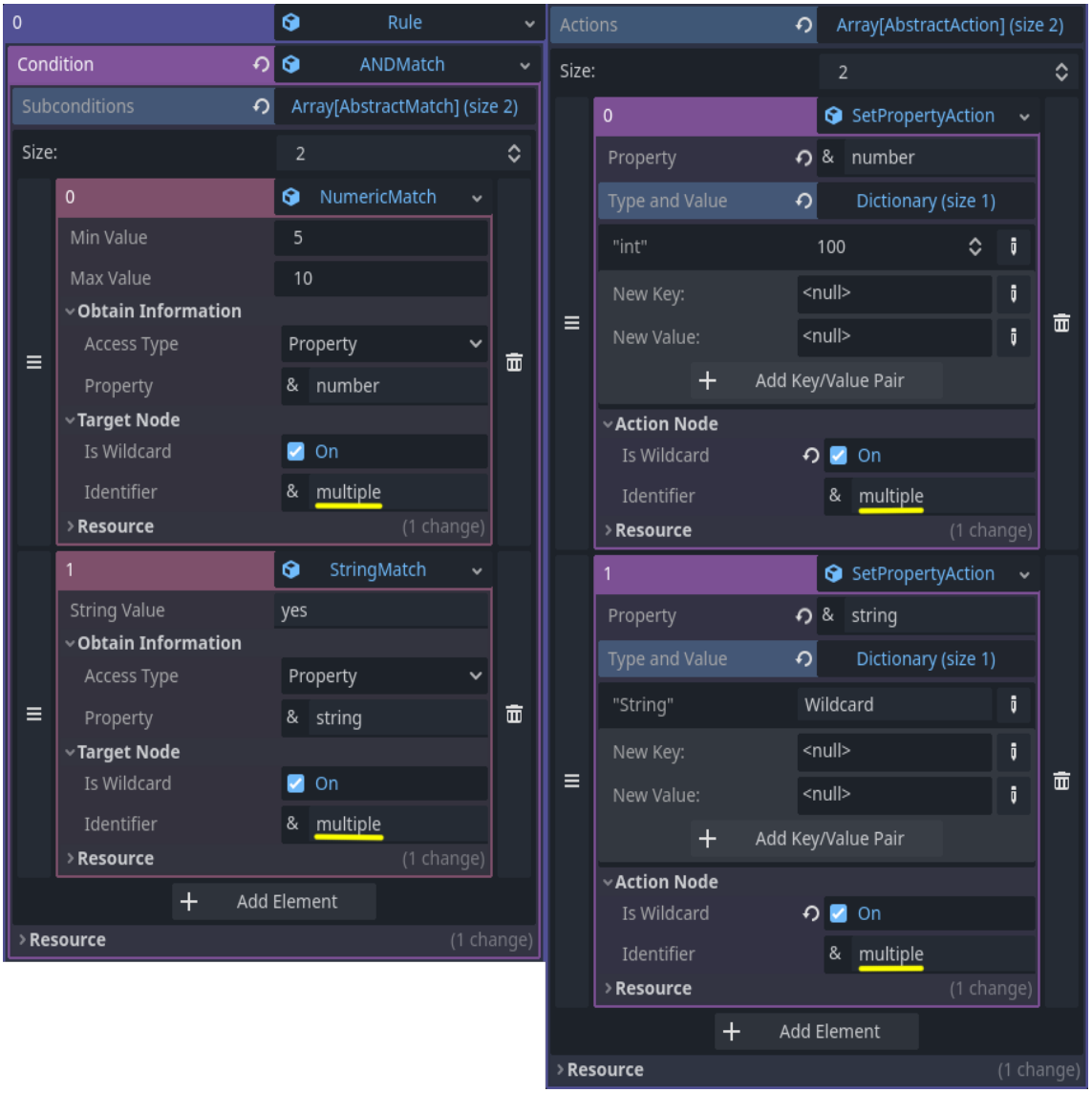

Figura 3.19: *Inspetor com regra do teste de condição múltipla com variável compartilhada (versão Alpha). Vemos que a condição é um operador E com dois pareamentos: o primeiro verifica se a propriedade number contém um valor entre* 5 *e* 10*, enquanto o segundo testa se a propriedade string é igual à "yes". Ambos os pareamentos são aplicados em um nó genérico, identificado pela variável multiple. Olhando agora as ações, vemos que são duas: uma altera number para 100 e a outra, string para Wildcard, mas o importante é que ambas são aplicadas no nó multiple, ou seja, em qualquer nó que satisfez a condição por completo.*

```
{"Rules": [
    {"if": ["AND", [
        ["Numeric", 5, 10, "?multiple", "number"],
        ["String", "yes", "?multiple", "string"]
    ]],
    "then": [
        ["SetProperty", "?multiple", "number", {"int": "100"}],
        ["SetProperty", "?multiple", "string",
            {"String": "Wildcard"}]
    ]}
]}
```

Figura 3.20: *Rules Editor com regra do teste de condição múltipla com variável compartilhada, versão textual da regra criada na interface gráfica da Figura 3.19 (versão Alpha). Repare na variável ?multiple, que aparece em todos os componentes atômicos da regra, garantindo que se uma mesma entidade satifaz os dois pareamentos, ela será alvo de ambas as ações.*

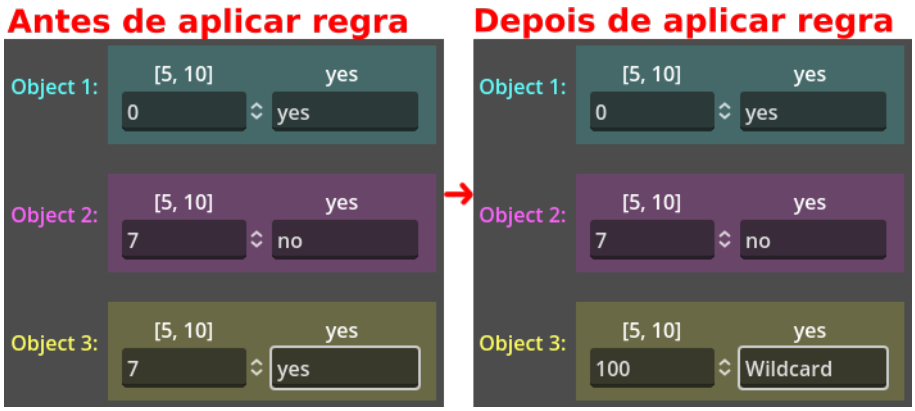

Figura 3.21: *Cena de teste para condição composta com variável compartilhada, cuja regra foi declarada nas Figuras 3.19 e 3.20 (versão Alpha). À esquerda, vemos o estado antes da apliação da regra, no qual temos três objetos: o Objeto* 1*, cujo campo numérico apresenta* 0 *e a string é igual a "yes"; o Objeto* 2*, cujo o número é* 7 *o texto é "no" e o terceiro objeto ´, com* 7 *e "yes". À direita, vemos o resultado da aplicação da regra: somente o Objeto* 3 *sofreu mudanças, pois ele era o único que satisfazia a condição completa, que exigia o número no intervalo* $[5, 10]$ *e a string igual a "yes".*

Vale apontar que nos dois outros objetos, uma das propriedades atendia o valor que era pedido, enquanto a outra não, o que comprova que nossa condição verifica ambas para eleger os candidatos válidos. O primeiro objeto (azul) possui a *string* idêntica ao valor desejado, o qual está escrito acima do campo de inserção, mas a propriedade *number* não está no intervalo necessário, o qual também está indicado acima da caixa numérica. Já para o objeto do meio, ocorre o oposto: o valor numérico é aceitável, mas a palavra inserida, não.

3.3 Versão *Beta*

Chegando à versão *Beta* do *plugin*, a primeira medida tomada foi consertar o ponto destacado na Seção 3.2.3, referente às ações invocarem suas mudanças em todos os nós válidos. Esse deve ser o comportamento padrão, pois o uso de variáveis pressupõe a modelagem de categorias de entidades, portanto, o esperado é que todos os objetos se atualizem simultaneamente, sem esperar um ciclo do sistema para cada um. Em termos abstratos, queremos aplicar todas as possíveis substituições da variável, gerando os efeitos decorrentes de cada uma.

Outra questão complementar a essa e que também precisava ser corrigida era o escopo de busca por candidatos, visto que até o momento estávamos restringindo os nós testados aos filhos do SBR, algo que viola o requisito **T4** levantado na Seção 2.2 (veja a Tabela 2.1). Da maneira como estava antes, o usuário era obrigado a colocar o nó de `RuleBasedSystem` na raiz de sua cena principal e qualquer sub-cena adicional precisaria ser instanciada abaixo na hierarquia da árvore.

Desse jeito, concluímos ser preciso delimitar o escopo de atuação do sistema de uma maneira mais dinâmica e aberta. A forma encontrada de se fazer isso foi com o uso de Grupos, explicados na Seção 1.3.1, pois eles permitem "etiquetar" nós que estejam em qualquer local da cena, não importando se foram instanciados depois, ou estão presentes em uma sub-cena desempacotada a partir do disco. Com isso, o motor de jogos consegue acessar todos os nós de um mesmo grupo como se fossem uma só entidade.

Consequentemente, foi decidido que, quando configuramos uma variável em um pareamento, deve haver uma opção para definir os Grupos de nós nos quais queremos realizar a busca por candidatos. Essa escolha foi condensada na Decisão de projeto 6 abaixo, a qual também prevê o seu uso para as ações, assim expandindo o escopo dos efeitos gerados pelo sistema.

Decisão de projeto 6 (Grupos de nós). Serão usados Grupos para determinar o escopo de variáveis que representam nós. Em pareamentos, a primeira lista de candidatos será dada pela união de todos os Grupos especificados, mas caso não haja nenhum, serão usados os filhos do SBR. Em ações, deve ser possível invocar as mudanças em todos os nós de um Grupo.

Como último ajuste planejado para essa versão, temos que ela é a primeira com compatibilidade completa com o *gd-plug*, o que implica que o diretório `addons/-rule_based_godot` está na raiz do repositório, conforme explicado na Seção 2.2. Essa foi uma mudança organizacional simples, mas necessária para cumprirmos o requisito **T2** (Tabela 2.1).

Tratando agora das funcionalidades não previstas, percebemos ao longo do desenvolvimento que seria útil para a definição de comportamentos se fosse possível recuperar o valor exato encontrado por um pareamento, quando esse verifica o pertencimento a um intervalo. Isso poderia ser utilizado como mais uma forma de transmissão de informação entre a condição e as ações de uma regra, assim permitindo uma maior conexão entre os dois componentes.

Um exemplo lúdico que demonstra como esse mecanismo pode ser útil é: suponha

que em nosso jogo uma das personagens possui um detector de metais, com uma luz que brilha mais forte quanto mais perto de um objeto metálico ela está. Para modelar isso, temos o pareamento de distância e um método que altera a luminosidade do indicador, o qual pode chamado por meio da ação `CallMethod`. Se não for possível recuperar o valor exato encontrada pelo pareamento, será necessário criar uma regra para cada intervalo de distância, definindo algo como o pseudocódigo visto no Programa 3.2:

Programa 3.2 Regras para detector de metal, sem variável de dados.

```
1    ▷ Primeira regra
2  SE (0 ≤ distancia para ?metal < 10) E ?metal é metal
3  ENTAO configura Brilho para 2
4
5    ▷ Segunda regra
6  SE (10 ≤ distancia para ?metal < 20) E ?metal é metal
7  ENTAO configura Brilho para 1
8
9    ▷ Terceira regra
10 SE (20 ≤ distancia para ?metal) E ?metal é metal
11 ENTAO configura Brilho para 0
```

Agora, com a adição de uma **variável de dados**, podemos salvar a distância exata encontrada e usá-la para configurar diretamente o brilho do detector de metais, por meio de uma expressão matemática (a qual assumimos ser interpretada antes de passarmos o argumento para a função). O Programa 3.3 demonstra essa nova modelagem, que utiliza a variável de dados e o método parametrizado para configurar o brilho:

Programa 3.3 Regras para detector de metal, com variável de dados.

```
1    ▷ Única regra
2  SE (0 ≤ distancia para ?metal = ?dist) E ?metal é metal
3  ENTAO configura Brilho para ⌈(20−?dist)/10⌉
```

Repare que conseguimos representar esse comportamento complexo com apenas uma regra, sendo que antes precisaríamos de um número de regras igual à quantidade de níveis de luminosidade (equivalentes aos intervalos de distância). Essa simplificação da representação é benéfica tanto para o usuário, que precisa escrever menos, quanto para a eficiência do sistema, que utiliza menos objetos em memória e itera sobre menos regras.

Esse exemplo trata de uma situação específica, mas podemos abstraí-lo para um contexto mais genérico, no qual temos um pareamento numérico e uma ação que depende do seu valor exato. Em suma, conseguimos intuir que a adição das variáveis de dados fornece mais poder expressivo para a declaração de regras, complementando o sistema de unificação já implementado. Tudo isso nos levou à Decisão de projeto 7:

Decisão de projeto 7 (Variáveis de dados). Pareamentos cuja verificação se baseia em dados, ou seja, podem ser reduzidos a um teste de um valor em um domínio, irão oferecer a

possibilidade de salvar o valor exato obtido em uma variável, para oferecer mais poder expressivo e simplificar a declaração de regras. As ações poderão usar esses dados por meio de uma *string* com o primeiro caractere igual a "?", seguido do nome da variável, da mesma forma como foi definido na Decisão 5.2

Prosseguindo com as adições não planejadas, foi sentida a necessidade de acrescentar alguma opção para automatizar o ciclo de execução do Sistema Baseado em Regras por haver muitos casos em que é desejada uma atualização constante do mundo do jogo, exigindo a verificação contínua das regras. Nesses casos, não faz sentido deixar para o usuário a responsabilidade de invocar uma iteração: ele teria que configurar gatilhos em todos os possíveis eventos do jogo que exigem uma atualização do sistema. Assim, acabamos tomando a Decisão de projeto 8:

Decisão de projeto 8 (Configurações de iteração). O SBR irá possuir opções que permitem a iteração automática sobre as regras, seguindo uma taxa de atualização definida pelo usuário. Também deve ser possível configurar o sistema para iterar somente quando é solicitado.

Para adicionarmos essas opções automáticas, decidimos que o objeto do sistema será convertido em um `Timer`, um tipo especial de nó que funciona como um alarme, esperando uma quantidade pré-definida de tempo para disparar um sinal. Com isso, agora é possível determinar um período de espera entre uma iteração e outra, por meio da propriedade *wait_time*, editável no Inspetor sob a categoria *Timer*. Porém, como dito na Decisão 8, o tempo de espera só deve ser seguido caso a opção correspondente estiver selecionada.

Essa configuração de cronômetro é extremamente flexível, porém, há mais um caso especial que gostaríamos de dar suporte: iterar sobre as regras a cada *frame* de execução do jogo. Essa alternativa atrela diretamente a frequência de acionamento das regras com a taxa de atualização do mundo do jogo, garantindo que todas as possíveis mudanças no ambiente serão analisadas pelo SBR assim que ocorrem, ou no máximo no quadro seguinte.

Porém, isso possui um risco associado, pois executar um procedimento custoso a cada quadro pode prejudicar o desempenho do jogo. Para mitigar esse problema, ao menos em partes, a *Godot* separa a atualização em duas funções: `_process`, que realmente atualiza o nó a cada quadro, da maneira mais frequente possível e `_physics_process`, que ocorre a uma taxa fixa de 60 vezes por segundo e lida com a física do jogo[9]. Como o acionamento de regras é algo que pode afetar o mundo virtual diretamente, inclusive os eventos de física, escolhemos implementar a iteração automática no processamento de taxa fixa.

Assim, terminamos com três opções de iteração para o `RuleBasedSystem`, as quais são selecionadas pelo usuário via Inspetor no momento de configuração da cena:

- Configurações de iteração do `RuleBasedSystem` (propriedade *Iteration Update*):
  - `Every Frame`: itera sobre as regras a uma taxa fixa de 60 vezes por segundo (segue o *Physics Fps*, que pode ser alterado nas configurações do projeto);

[9] Veja mais sobre as duas formas de processamento na *Godot game engine* em: https://docs.godotengine.org/en/4.1/tutorials/scripting/idle_and_physics_processing.html

  - `On Timer`: itera sobre as regras a cada `wait_time` segundos (seguindo a propriedade da categoria *Timer*);
  - `On Call`: itera sobre as regras somente quando o método *iterate* é chamado (pode ser conectado a um sinal, ou invocado diretamente).

Por fim, a última adição simples realizada foi a criação de mais um tipo de pareamento, o qual permite analisar a estrutura da árvore de cena e utilizar a relação de hierarquia entre os nós para verificar se uma ação deve ser tomada ou não. Esse pareamento cumpre o último ponto do nosso requisito funcional que exigia o acesso e controle das propriedades disponíveis aos desenvolvedores durante a edição de cenas (**F1** na Tabela 2.1).

- Pareamento Atômico (sufixo *Match*)
  - `Hierarchy`: verifica a relação entre dois nós na árvore da cena, analisando se um é pai (*Parent of*), filho (*Child of*) ou irmão (*Sibling of*) do outro.

Agora, um aspecto importante trabalhado para a versão *Beta* e que merece destaque especial é a extensibilidade do sistema, ou seja, a capacidade do desenvolvedor usuário de implementar novas funcionalidades para o *plugin*. A Seção 3.3.1 irá tratar das decisões tomadas para assegurar que esse processo seja o mais fácil possível.

3.3.1 Extensibilidade pelo usuário

O *plugin* foi concebido como um *software* de código aberto, o que significa que o usuário consegue acessar e modificar o código-fonte. Portanto, em termos técnicos sempre foi possível adicionar novas funcionalidades. Contudo, quando falamos em extensibilidade no contexto desse trabalho, estamos nos referindo a uma estrutura construída para possibilitar novas adições com o mínimo de trabalho possível. Portanto, essa seção irá tratar das bases criadas para possibilitar que o usuário crie novas classes com o mínimo de código possível.

Conforme dito na Seção 2.1, o desenvolvedor usuário precisa poder criar novos pareamentos, árbitros e novas ações, portanto, resolvemos investir em estruturas que facilitem a criação de novas instâncias desses componentes. O intuito é esconder a complexidade de implementação nas superclasses abstratas, de forma que as subclasses concretas precisem definir apenas o mínimo possível. Em particular, não queremos que o usuário tenha que se preocupar com o uso de variáveis e com a sintaxe *JSON*.

Primeiramente, decidimos deixar algumas funcionalidades pré-configuradas nas superclasses abstratas de pareamentos e ações, de maneira que as subclasses concretas só precisam ligar as opções correspondentes para utilizá-las. Isso introduz mais complexidade nas classes abstratas, mas como não é necessário modificá-las para criar novos recursos, é melhor lidar com toda a complexidade "por baixo dos panos", assim diminuindo o trabalho necessário para o usuário implementar algo novo. Diante disso, chegamos à Decisão de projeto 9:

Decisão de projeto 9 (Subclasses configuráveis). Subclasses implementadas pelo desenvolvedor usuário poderão ligar ou desligar funcionalidades extras, como: gerenciamento de nós *wildcard*, variáveis de dados, pré-configurações para obter um dado por meio de uma

propriedade, ou chamada de método etc. O estado de ativação das funcionalidades deve ser refletido na presença, ou ausência dos correspondentes agrupamentos de propriedades no Inspetor, seguindo a Decisão 4.

Explicando agora as configurações criadas, uma que está ativa por padrão nos dois tipos de recurso gerencia nós *wildcard*, seguindo o processo de unificação descrito na Seção 3.2.1. É possível desativá-lo caso o usuário deseje implementar um novo tipo de componente que não lida com nós. Um possível exemplo seria a verificação e alteração de uma propriedade global, como o tamanho da janela do aplicativo, algo que poderia ser feito por meio de um pareamento e uma ação que não precisam acessar nenhum nó.

Em pareamentos, outra configuração padrão é o uso de variáveis de dados, a qual assume que a verificação pode ser reduzida a um teste de inclusão, ou seja, analisar se um valor está em um domínio. O motivo disso ser o padrão é que a maioria dos elementos em jogos eletrônicos atende a essa propriedade, pois se baseiam em: um valor numérico em um intervalo (distância entre objetos, quantidade de vida, tempo decorrido), ou em uma categoria (personagem aliada ou inimiga, de quem é a vez, ataque de fogo ou de gelo).

Por fim, percebemos que a maioria dos valores testados em pareamentos provêm das propriedades do nó testado, ou do resultado da chamada de algum de seus métodos. Com isso em mente, resolvemos incluir pré-configurações que automatizam a obtenção do dado a ser testado, permitindo a declaração de uma propriedade que deve ser acessada, ou de um método que deve ser chamado, especificando seus argumentos. Vale destacar que essa configuração é dependente da anterior, portanto, só deve ser ligada caso a anterior também for.

Tendo isso, trataremos agora da maneira com a qual o usuário interage com essas configurações. Ao criar uma nova instância de pareamento atômico ou ação, é possível ligar ou desligar certas propriedades *booleanas* no construtor da classe (método `_init`), cada uma correspondente a uma das funcionalidades mencionadas. Para cada configuração ativada, um novo agrupamento de propriedades aparecerá no Inspetor, com o mesmo nome da propriedade *booleana* marcada como verdadeira.

Dessa forma, a interface gráfica só mostrará as funcionalidades que estão ativas, permitindo a edição das propriedades necessárias. Para ilustrar isso, a Figura 3.22 apresenta duas versões de uma nova classe de pareamento: na primeira (acima), todas as configurações foram ativadas, fazendo o Inspetor mostrar todos os possíveis agrupamentos; já na segunda (abaixo), somente a configuração que gerencia nós foi ligada, portanto, as outras foram removidas da nossa *GUI*.

Com isso, o usuário consegue ver quais funcionalidades estão ativas por meio da interface gráfica, não precisando ter ciência das inativas. Porém, esse poder de configuração gera uma complicação: a sintaxe utilizada na *TUI* deve comportar as múltiplas combinações de propriedades que podem ou não ser inclusas, conforme as configurações ativas. Dessa forma, é preciso haver uma representação para parâmetros opcionais, tanto em nossa meta-linguagem, quanto no processo interno de interpretação.

Todavia, conseguimos prever que o uso desses parâmetros opcionais irá introduzir mais um fator de complexidade para a declaração textual de regras. Unindo isso à formatação

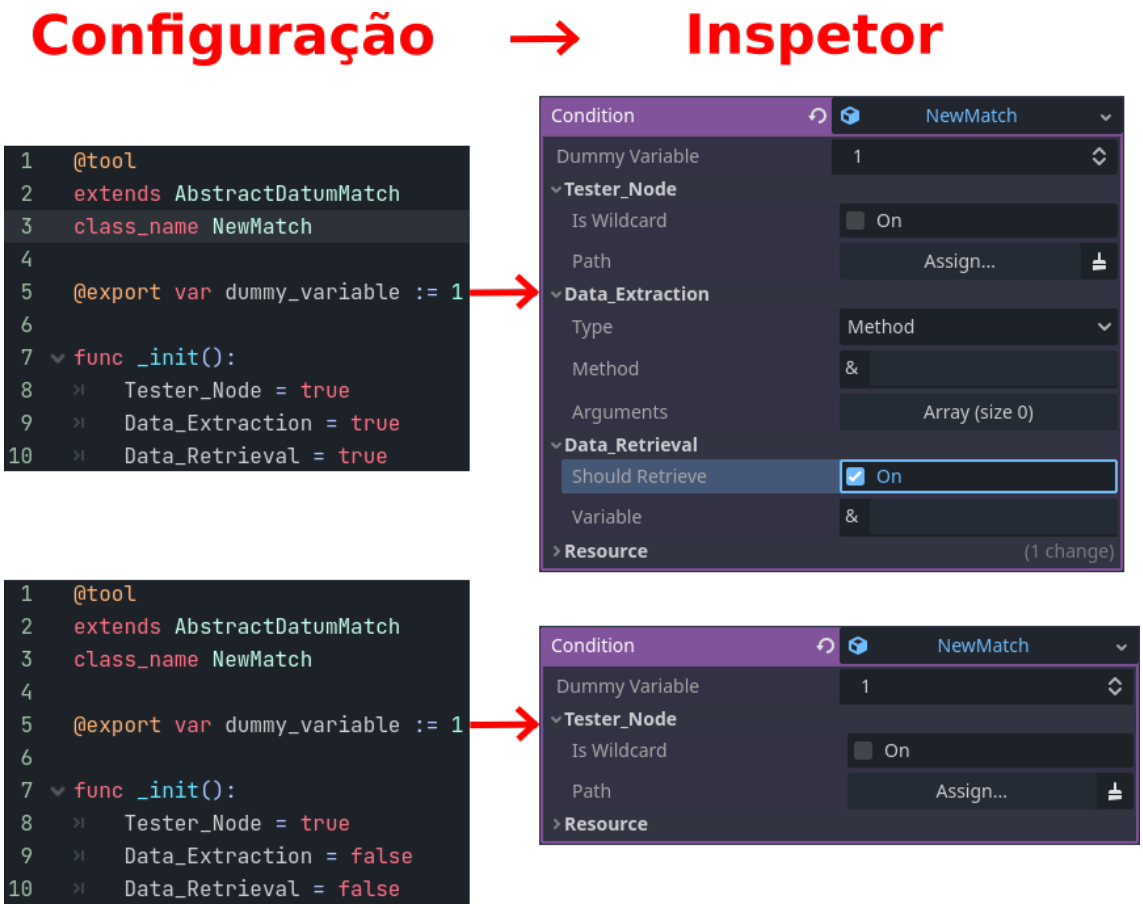

Figura 3.22: *Demonstração de como as configurações de um novo pareamento (esquerda) podem afetar o aparecimento de agrupamentos de propriedades no Inspetor (direita). As opções atribuídas como `true` no código se refletem em agrupamentos de mesmo nome na interface gráfica.*

específica de cada tipo de pareamento ou ação, concluímos ser necessário encontrar uma maneira de simplificar e padronizar a sintaxe utilizada. Essa padronização deve definir a sequência e a maneira com que representaremos os parâmetros opcionais, garantindo compatibilidade entre a meta-linguagem usada nos modelos (explicados na Seção 3.2.2) e a representação interna.

Com isso, decidimos que os recursos devem gerar e interpretar sua representação *JSON* de maneira automática, aplicando a padronização descrita acima. Com essa sintaxe automática, as classes concretas não terão nenhum trabalho relacionado à declaração textual de regras, pois tudo será resolvido de maneira uniforme nas classes abstratas, seguindo a padronização. Tendo isso, registramos a Decisão de projeto 10:

Decisão de projeto 10 (Sintaxe automática). Os modelos para a representação *JSON* de pareamentos e ações serão gerados automaticamente, demarcando campos opcionais e seguindo a mesma ordem com que as propriedades foram declaradas no *script*. Esses modelos seguirão uma meta-linguagem própria, fornecendo indicações de como transformá-los no formato *JSON* correto. Isso evita que o usuário tenha que definir a interpretação de novos recursos.

Para implementarmos essa automação, recorremos a alguns mecanismos de mais baixo

nível presentes na *Godot Engine* e na linguagem *GDScript*. Em primeiro lugar, estabelecemos que o identificador do tipo de pareamento ou ação será dado pelo nome da classe, definido com a palavra-chave `class_name`, sendo removidos os sufixos de *Match* ou *Action*. Dessa forma, supomos que o usuário irá declarar a classe com o nome que deseja utilizar na interface textual.

Ademais, a representação *JSON* inclui todas as propriedades exportadas, na ordem em que são declaradas no código. Para marcar uma propriedade como exportada, usamos a família de anotações `@export`[10], e para distinguir as privadas das exportadas, verificamos as suas *flags* de uso. A visibilidade das propriedades é algo que já estávamos manipulando para implementar o Inspetor dinâmicos, que pode ser visto na Figura 3.14.

Para salvar e carregar os valores dessas propriedades, foram criados *wrappers* para as funções nativas `var_to_str` e `str_to_var`, as quais fazem a tradução entre uma variável qualquer e a `string` formatada que a representa. Decidimos modificar a formatação nativa, por ela ser muito verborrágica, contendo aspas e espaços desnecessários e aproveitamos a oportunidade para introduzir duas notações próprias: **(i)** uma *string* iniciada com o acento circunflexo (“^”) indica um `NodePath` e **(ii)** o termo *“inf”* indica o valor infinito da representação numérica em ponto flutuante (IEEE, 2008).

Por fim, foi utilizada a referência global de classes declaradas no projeto para procurar e adicionar todos os subtipos de ações e pareamentos no menu de inserção do *Rules Editor*. Dessa forma, conseguimos anexar os modelos dos recursos criados pelo usuário desenvolvedor, por meio dos botões demonstrados na Figura 3.15. A única questão dessa funcionalidade é que o usuário precisa reiniciar o *plugin*, algo que é considerado aceitável para nossos requisitos técnicos (Seção 2.2).

A versão *Beta* do projeto pode ser acessada com a etiqueta de mesmo nome no repositório[11]. É importante destacar que o suporte ao *gd-plug* não introduz nenhuma dependência para a execução do código, logo, nosso único resuisito continua sendo o Editor da *Godot 4.1+*. Na seção seguinte, observaremos as cenas de teste elaboradas para essa versão do sistema.

3.3.2 Cenas de teste

O primeiro teste serve apenas para averiguar o funcionamento do pareamento de hierarquia criado para essa versão do sistema. Para isso, foram definidas três regras, uma para cada tipo de relação testada pelo `HierarchyMatch`: se um nó é irmão do outro, ou seja, possuem o mesmo pai; se um nó é pai direto de outro e se o primeiro nó é filho direto do segundo. A Figura 3.23 mostra como o usuário pode escolher dentre essas opções na interface gráfica. Vale apontar que é necessário haver tanto a relação de pai para filho, quanto de filho para pai, pois o pareamento define um nó fixo como a origem da relação e o seu alvo pode ser outro objeto fixado, ou uma variável.

[10] Leia mais sobre as anotações de exportação da *Godot* em: https://docs.godotengine.org/en/4.1/tutorials/scripting/gdscript/gdscript_exports.html

[11] Acesse diretamente a versão *Beta* em: https://github.com/rvbatt/rule-based-godot/tree/beta

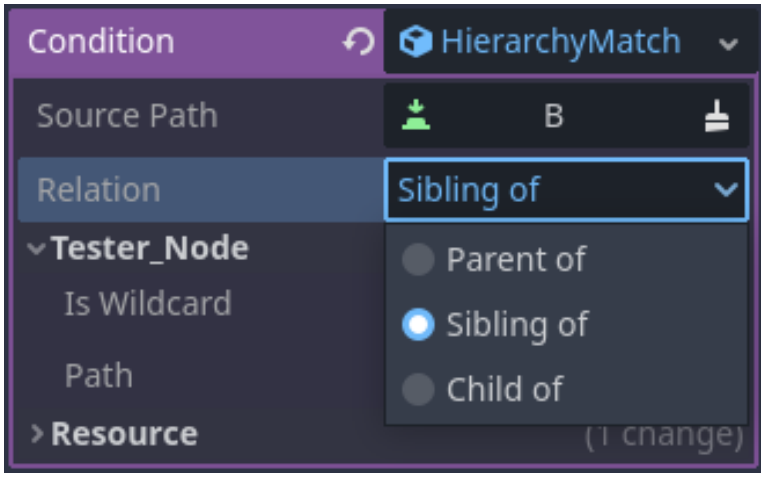

Figura 3.23: *Inspetor com opções de relação hierárquica entre dois nós dadas por* `HierarchyMatch` *(versão Beta). A ordem de leitura é: "<Source Path> é <Relation> de <Tester Node>?", ou seja, essa condição representa: "B é irmão de C?" (o nó C está configurado como o Path do Tester Node, mas o menu da relação está o ocultando).*

Como esse teste não apresenta nenhuma configuração complexa, não trabalha com variáveis e todas as regras apresentam o mesmo formato do exemplo dado na Figura 3.23, não será dada a declaração explícita das regras utilizadas. Portanto, podemos analisar diretamente a Figura 3.24, na qual temos vários possíveis estados da cena, começando pelo estado inicial no canto superior esquerdo e depois seguindo com situações que disparam apenas uma regra cada, assim permitindo verificá-las individualmente.

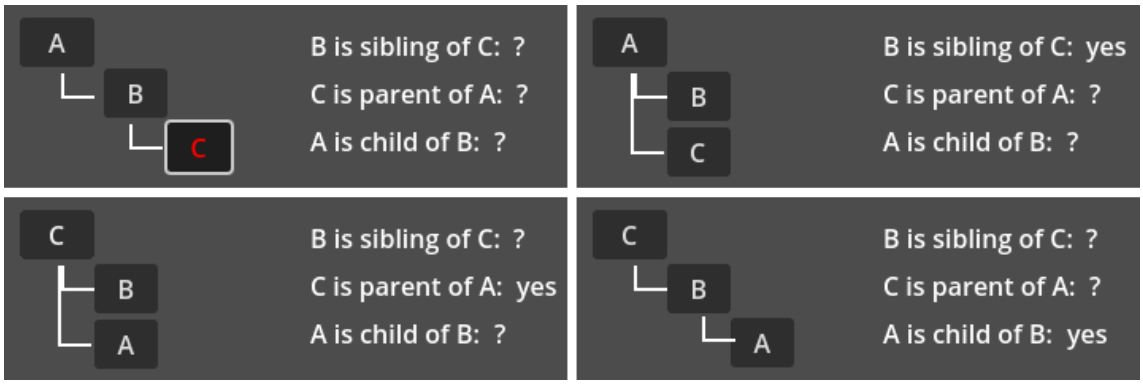

Figura 3.24: *Quatro possíveis estados da cena de teste de hierarquia de nós (versão Beta). O estado inicial, no canto superior esquerdo, não satisfaz nenhuma regra; repare que o item C está selecionado, de forma que o próximo item clicado pelo jogador será atribuído como o pai de C. O canto superior direito mostra uma configuração na qual apenas a primeira regra é satisfeita, pois o B é irmão de C, mas C não é pai de A e A não é filho de B. No canto inferior esquerdo, apenas a segunda regra é satisfeita e no inferior direito, somente a terceira. O conjunto dessas situações demonstra que todos os testes do pareamento funcionam como esperado.*

Para alterar a estrutura da árvore testada, o jogador pode selecionar qualquer um dos itens e depois clicar em outro, fazendo com que o nó selecionado se torne filho do que foi clicado em sequência. Por exemplo: no estado inicial da Figura 3.24, o bloco *C* está selecionado, o que é visível devido ao seu contorno branco e texto vermelho; se o jogador

clicar em *A*, a árvore ficará igual à presente no canto superior direito, com *C* sendo filho de *A*.

Continuando, a segunda cena montada visa testar duas funcionalidades de uma vez: o uso de Grupos e as variáveis de dados. Para isso, foi implementada uma versão do exemplo do detector de metais, representado no Programa 3.3, com algumas modificações que tornam claro o uso dos Grupos na busca por objetos metálicos e a variável de dados para salvar a distância encontrada.

Primeiramente, o brilho do sensor não será discretizado com base em intervalos de distância, pois utilizaremos diretamente o valor da distância para configurá-lo, o que significa que o brilho terá um valor contínuo. Esse tipo de comportamento não seria possível sem o salvamento do valor exato encontrado pelo pareamento, portanto, é um ótimo teste para as variáveis de dados.

Segundamente, o sensor só irá detectar objetos que pertençam do Grupo *metal*, assim garantindo que o escopo de busca é condizente com o teste. Além disso, teremos uma ação que atualiza qualquer elemento de texto pertencente ao Grupo *distance_tag* com o valor exato da distância, ilustrando uma união das duas funcionalidades testadas.

Por fim, será usada a configuração de iteração a cada *frame*, pois queremos que o brilho e o valor da distância estejam sincronizados com o que ocorre no mundo do jogo, em tempo real. Dessa forma, o jogador movendo o detector de metais por meio do teclado terá um retorno visual imediato, o que é importante em vários tipos de jogos.

A Figura 3.25 ilustra dois momentos do teste, um com o detector de metais longe do objeto metálico e outro com os dois mais próximos. A cena em si não é difícil de entender, logo, será dado mais destaque para a declaração da regra que gera esse comportamento, a qual pode ser vista nas Figuras 3.26 e 3.27. O ponto mais importante a ser observado é o uso das variáveis, em especial de *?dist*, que representa o valor da distância e aparece nos três componentes da regra: no pareamento da condição e nas duas ações.

Perceba na declaração do pareamento que a origem da medição de distância é o nó *MetalDetector*, enquanto o alvo é qualquer objeto pertencente ao Grupo *metal*, cujo nome aparece num vetor, já que podemos determinar a união de vários Grupos como o escopo de busca. Também vale ressaltar que o intervalo definido para os valores é $[0, +\infty[$, ou seja, qualquer distância será válida e irá satisfazer a condição. Algo menos relevante é a variável *?found_metal*, que serve apenas para registrar a lista dos objetos de metal detectados.

Olhando agora para as ações, a primeira se aplica em todos os nós do Grupo *distance_tag*, alterando a propriedade *text* para ser igual ao valor contido na variável *dist*. Repare que é utilizada a sintaxe com o prefixo de ponto de interrogação, conforme a Decisão de projeto 7, pois as ações precisam diferenciar *strings* constantes de identificadores de variáveis. Finalmente, a segunda ação chama o método *set_brightness* do *MetalDetector*, passando diretamente a distância armazenada na variável de dados.

Com isso, concluímos o relatório das etapas de desenvolvimento do sistema. O próximo capítulo tratará da versão final, abordando os últimos detalhes de implementação que foram inclusos desde o *Beta* e explicando o uso do *plugin*, como se fosse um manual do usuário.

Figura 3.25: *Dois momentos da cena de teste do detector de metais (versão Beta). A primeira imagem (acima) mostra um momento em que o detector de metais está longe do objeto metálico, portanto, o brilho vermelho está fraco e o letreiro indica cerca de* 94.5 *píxeis de distância. Na imagem de baixo, o detector se encontra bem próximo do metal, tornando o brilho mais intenso e fazendo o letreiro ser atualizado para quase* 60 *píxeis de distância. Note que o texto que não está no Grupo especificado pela regra não é afetado.*

```
{"Rules": [
    {"if":
        ["Distance", "?dist", "^MetalDetector", 0, "inf",
            "?found_metal", ["metal"]],
    "then": [
        ["SetProperty", ["distance_tag"], {"text": "?dist"}],
        ["CallMethod", "^MetalDetector", "set_brightness",
            ["?dist"]]
    ]}
]}
```

Figura 3.26: *Rules Editor com a declaração da regra do teste do detector de metais (versão Beta). Repare no uso da variável ?dist para salvar o valor encontrado por* `DistanceMatch` *e depois utilizá-lo nas duas ações. Outro ponto importante é o Grupo "ditance_tag", que faz o* `SetProperty` *ser aplicado em todos os nós com essa etiqueta. Veja também o uso da nova sintaxe, com "inf" representando o infinito de ponto flutuante e "^MetalDetector" simbolizando o NodePath do nó do sistema até o detector de metais.*

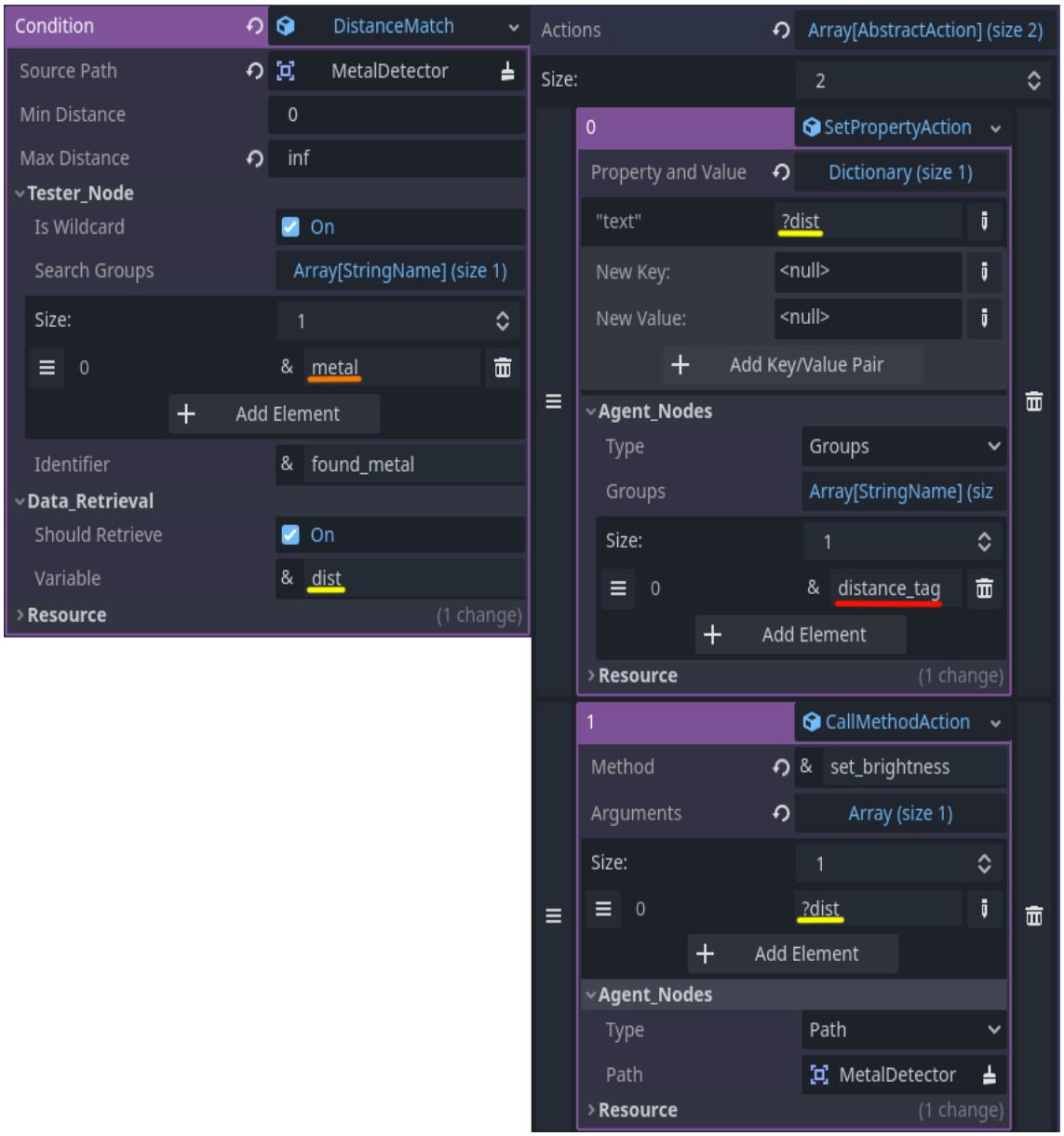

Figura 3.27: *Inspetor com declaração da regra do teste de detector de metais, idêntica à vista na Figura 3.26 (versão Beta). A condição, à esquerda, salva a distância do MetalDetector até um objeto do Grupo metal (destacado em laranja) na variável de dados ?dist (destacada em amarelo). Vemos que a primeira ação, à direita, usa o valor de ?dist para atualizar o texto de todos os nós no Grupo distance_tag (marcado em vermelho). Já a segunda ação, abaixo da primeira, também obtém o dado em ?dist e o usa para configurar o brilho do detector de metais.*

Capítulo 4

Resultados: *Rule-Based Godot*

Neste capítulo, apresentaremos o *Rule-Based Godot*, a versão final do sistema desenvolvido para este trabalho. Ele é um complemento para a *Godot Engine 4.1+*, compatível com o *Plugin Manager gd-plug*[1], disponibilizado sob a licença *MIT*[2] e feito inteiramente com a linguagem *GDScript*. Em termos práticos, o *plugin* constitui um arcabouço para Sistemas Baseados em Regra, fornecendo uma estrutura com diversas funcionalidades, como um sistema de unificação de variáveis já embutido, integração com a interface gráfica nativa e interface textual dedicada.

O *software* final é composto de: 30 arquivos, incluindo os *scripts* de código e a cena que forma o *RulesEditor*; 1486 linhas de código em *GDScript* e 210 *commits* no repositório. Para acessar o lançamento **1.0.0**, procure pela etiqueta de mesmo nome no repositório[3].

A seguir, descreveremos a arquitetura final do sistema, destacando as diferenças para o que foi planejado inicialmente, além de analisarmos se ela cumpre os objetivos definidos na Introdução e os requisitos levantados no Capítulo 2. Em sequência, falaremos sobre a usabilidade do *plugin*, explicando como instalá-lo, como declarar regras em uma cena, o uso das interfaces e a aplicação de variáveis. Também será dada uma lista de todos os *resources* que já vêm disponíveis, além de um passo-a-passo de como o usuário desenvolvedor pode criar novos.

Porém, antes de analisarmos o estado final do projeto, é preciso destacar a criação de um último pareamento atômico, `DistinctVariablesMatch`. Ele garante que é possível atribuir um valor distinto para cada variável em uma lista determinada, dessa forma, se um mesmo objeto for uma substituição válida para mais de uma variável, conseguimos assegurar que ele só será salvo como candidato de uma delas.

- Pareamento Atômico (sufixo *Match*)
  - `DistinctVariables`: assegura que todas as variáveis em uma lista determinada possuirão valores distintos quando for aplicada uma substituição.

[1] O *gd-plug* pode ser instalado na biblioteca oficial: https://godotengine.org/asset-library/asset/962

[2] Licença do projeto disponível em: https://github.com/rvbatt/rule-based-godot/blob/main/LICENSE.txt

[3] Acesse o primeiro lançamento em: https://github.com/rvbatt/rule-based-godot/releases/tag/v1.0.0

Isso é útil em condições do tipo "se mais de um objeto atende a esse requisito…", pois podemos criar dois pareamentos que testam a mesma condição, mas cada um com uma variável diferente, depois adicionar o `DistinctVariablesMatch` para garantir que as variáveis possuem valores distintos. Para testar esse novo pareamento, criamos uma cena na qual a regra só é acionada se uma área móvel estiver detectando dois objetos distintos ao mesmo tempo, o que pode ser visto na Figura 4.1.

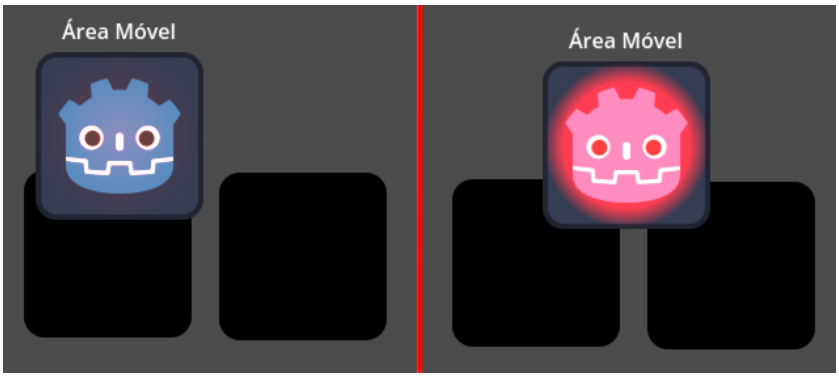

Figura 4.1: *Cena de teste para variáveis distintas (Versão 1.0.0). A condição exige que a Área Móvel esteja detectando quaisquer dois objetos distintos. À esquerda, vemos que o `DistinctVariablesMatch` assegura que o mesmo objeto não é pareado duas vezes, causando um acionamento errôneo da regra. Já à direita, temos a situação correta de acionamento: quando há dois objetos diferentes na área.*

Para modelar regras com esse novo *match*, basta colocá-lo abaixo do pareamento *booleano* que contém todas as ocorrências das variáveis que devem ser distintas. Com isso, deve-se preencher a lista de variáveis que devem ter atribuições distintas nesse escopo (não é preciso usar o prefixo "?"). A Figura 4.2 demonstra essa construção no editor textual, enquanto a Figura 4.3 ilustra como fica o Inspetor.

```
{"if": ["AND", [
    ["AreaDetection", "^MovableArea2D", "?obj1", ["collider"]],
    ["AreaDetection", "^MovableArea2D", "?obj2", ["collider"]],
    ["DistinctVariables", ["obj1", "obj2"]]
]],
```

Figura 4.2: *RulesEditor com condição de teste para variáveis distintas (Versão 1.0.0). Repare que os dois pareamentos de detecção de área são iguais, a não ser pelo nome da variável. Veja também que o `DistinctVariables` só contém a lista das variáveis que precisam ser distintas.*

4.1 Arquitetura do sistema

A arquitetura final do sistema pode ser entendida como o produto da arquitetura planejada, ilustrada na Figura 2.1, com as Decisões de projeto descritas ao longo do Capítulo 3. Isso quer dizer que a estrutura e os padrões de projeto estabelecidos no nosso planejamento foram mantidos, porém, houve a adição de alguns componentes auxiliares e a divisão de certas classes, além da especificação dos métodos e propriedades. Sabendo disso, observe a Figura 4.4 para ver todos os detalhes relevantes da implementação final.

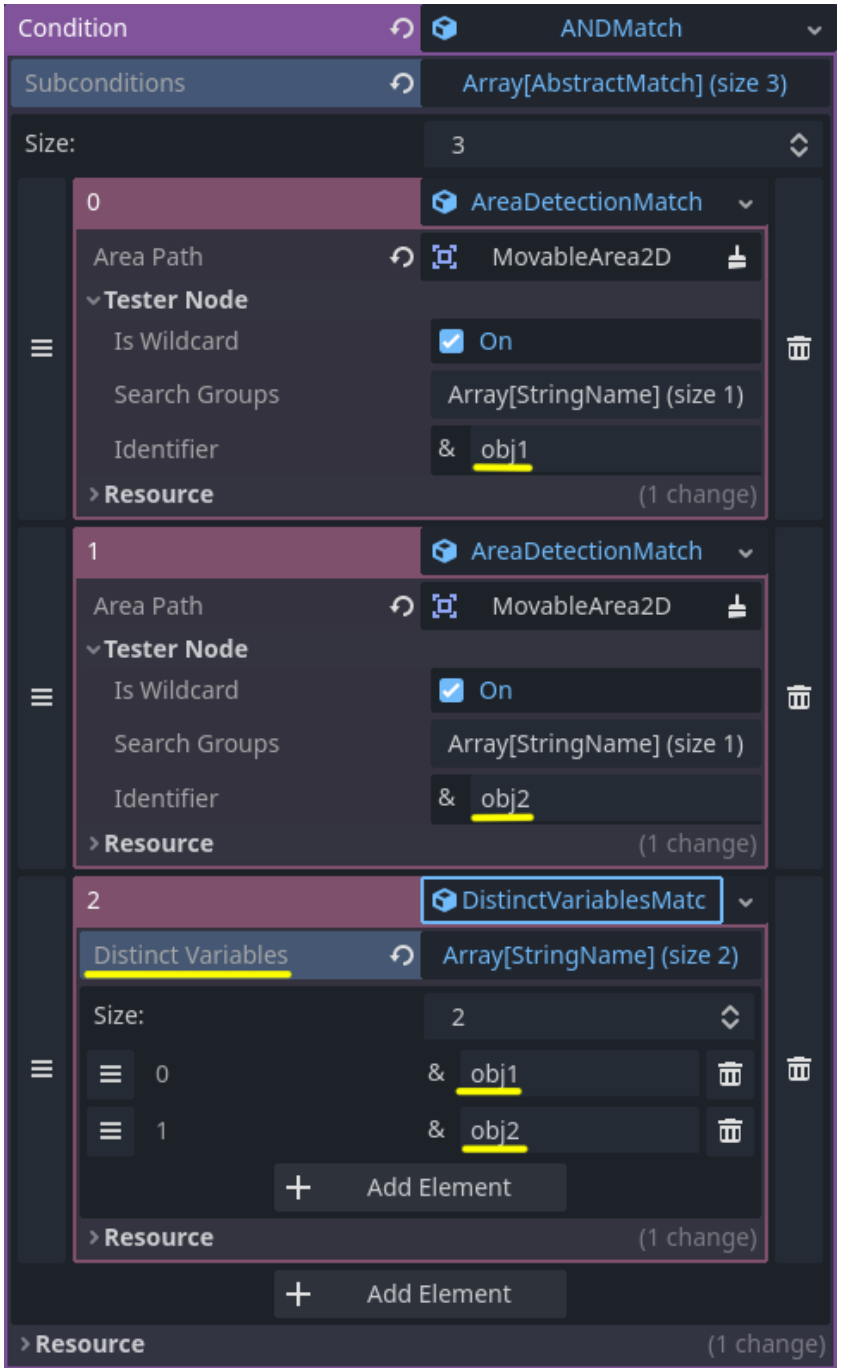

Figura 4.3: *Inspetor com condição de teste para variáveis distintas (Versão 1.0.0). Semelhante à Figura 4.2. Em amarelo, vemos destacadas as variáveis utilizadas: "obj1" e "obj2", os dois objetos distintos que devem ser detectados. Repare que nenhuma ocorrência apresenta o prefixo "?", pois todos os campos utilizados são exclusivos para o nome de variáveis.*

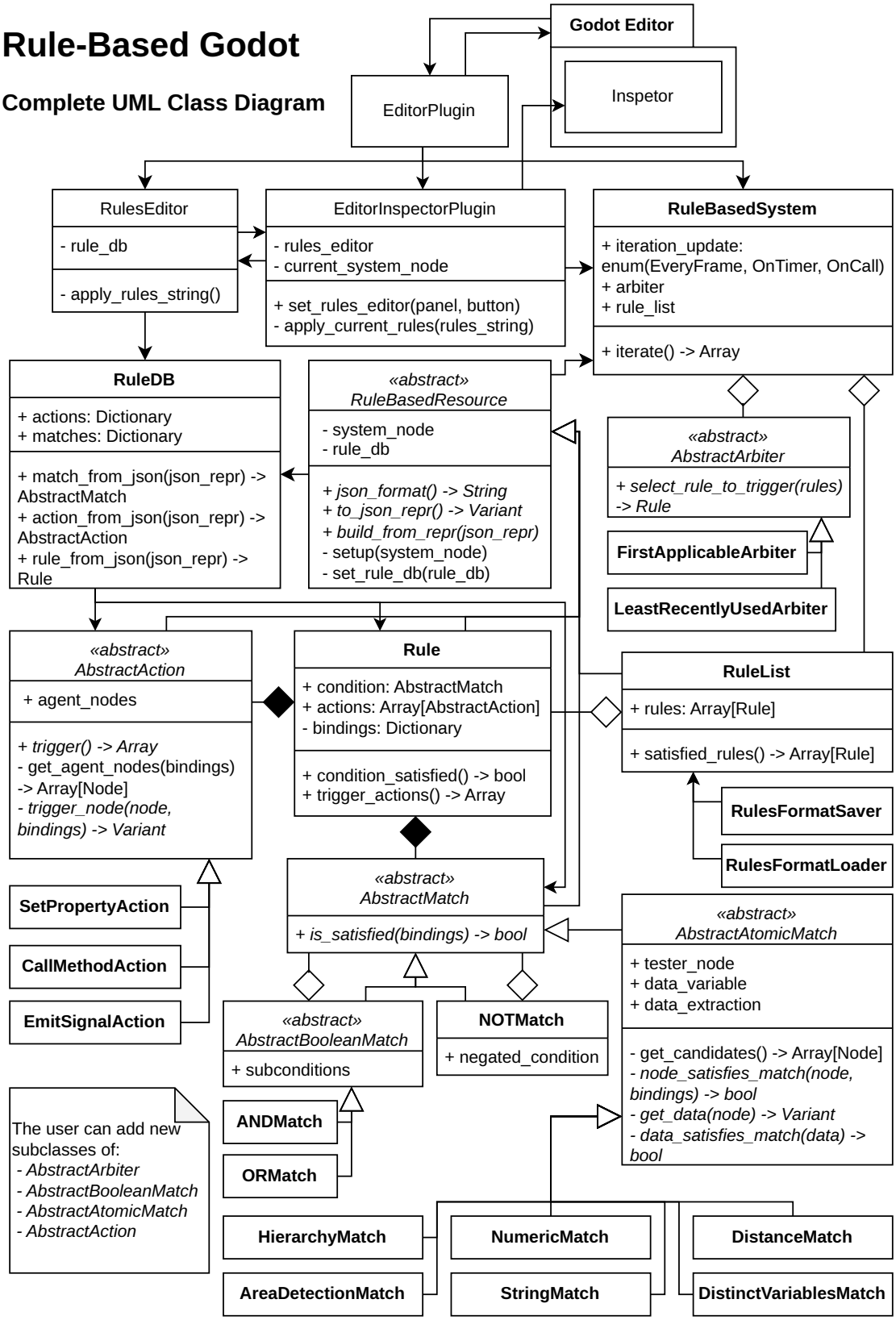

Figura 4.4: *Diagrama de classes completo da versão final do sistema (1.0.0), em inglês. Compare-o com o diagrama do que foi planejado, presente na Figura 2.1.*

O primeiro ponto de divergência está na arquitetura *MVC*, que rege a conexão do nosso *plugin* com o resto do Editor da *Godot*. Na versão final do sistema, temos o `RulesEditor` como um elemento de Visão, algo que não estava planejado e surgiu a partir da Decisão de projeto 5 (Interface textual), sendo que ele substituiu uma classe de *GUI*. O motivo de não termos um componente específico para a interface gráfica vem da Decisão de projeto 2 (Uso do Inspetor): como estamos utilizando o *Inspector*, o qual já está implementado no Editor nativo, não precisamos criar uma classe nova. Dito isso, vale apontar que o Inspetor também atua como uma Visão para o sistema, portanto, o papel definido na arquitetura planejada ainda é cumprido.

Agora, como foram criadas duas interfaces distintas para a mesma tarefa (declaração de regras), foi necessário estabelecer uma conexão entre elas, para garantir que a atualização de uma é refletida na outra. Para isso, foi necessário criarmos um `EditorInspectorPlugin`, o qual permite a atualização do Inspetor por meio de código, assim viabilizando a transmissão das configurações feitas no `RulesEditor` para a nossa interface gráfica. Dessa forma, esse novo sub-*plugin* atua como um Controlador, gerenciando as entradas do usuário e garantindo que ambas as interfaces estão sincronizadas.

Em decorrência, vemos que o `EditorPlugin` perdeu seu papel como Controlador, mas se manteve como a conexão principal do complemento com o resto do *Godot Editor*. Por outro lado, vemos que o `RuleBasedSystem` continua sendo o Modelo, conforme o previsto, servindo como detentor da estrutura de dados e gerenciador das configurações da aplicação. Sobre essas configurações, vale apontar a propriedade `iteration_update`, responsável por controlar a frequência com a qual o sistema itera sob suas regras, algo que foi incluso devido à Decisão de projeto 8 (Configurações de iteração).

Olhando agora para as duas classes auxiliares que não estavam previstas, vemos que a superclasse abstrata `RuleBasedResource` garante a aplicação da Decisão de projeto 1 (Uso de recursos), pois sendo ela um *Resource* da *Godot*, todas as suas subclasses também serão: `RuleList`, `Rule`, `AbstractMatch` e `AbstractAction`. Além disso, notamos que ela define métodos abstratos para o uso da sintaxe *JSON*, seguindo a Decisão de projeto 5.1 (Formato *JSON*).

O outro componente auxiliar é o `RuleDB`, uma espécie de banco de dados para os tipos de ações e pareamentos, cujo principal uso é a instanciação de objetos a partir de sua representação *JSON*. Com isso, sua maior contribuição para o sistema é a inclusão automática de novos recursos criados pelo desenvolvedor usuário, possibilitando sua instanciação a partir da sintaxe *JSON* automática — fruto da Decisão de projeto 10 (Sintaxe automática). Ele faz isso por meio de uma busca na referência global de todas as classes do projeto, assim garantindo que o usuário só precisa declarar um `class_name` para ter sua adição reconhecida pelo *plugin*.

Essa busca automática por novas classes também permite que nossa *TUI* esteja sempre atualizada com os modelos das ações e pareamentos adicionados. Isso demonstra que estamos seguindo nossos princípios de extensibilidade e de diminuir o máximo possível o trabalho do usuário, visto que estamos facilitando e automatizando a inclusão de seus recursos no ambiente do sistema.

Sobre as classes que estavam planejadas e acabaram divididas em duas, foi explicado

na Seção 3.2.2 que separamos as funcionalidades de salvamento e carregamento de objetos `RuleList` em duas classes: `RulesFormatSaver` e `RulesFormatLoader`. Além disso, vemos que há duas classes distintas que definem pareamentos compostos: `AbstractBooleanMatch` e `NOTMatch`, sendo que o planejado era definir apenas uma classe abstrata, da qual todos os operadores *booleanos* iriam herdar.

Decidimos separar o pareamento NÃO por representar o único operador lógico unário, o que exigiria um tratamento especial de qualquer forma. Assim, deixamos a classe `AbstractBooleanMatch` aberta para ser estendida pelo usuário, permitindo a adição de outros tipos de operadores *booleanos*, enquanto `NOTMatch` é fornecido como um *script* pronto.

Olhando agora para a classe `AbstractAtomicMatch`, percebemos a influência de várias Decisões de projeto na definição de suas propriedades e métodos. Para começar, todas as suas propriedades presentes na Figura 4.4 lidam com *flags* de configuração, um conceito que surgiu a partir da Decisão de projeto 9 (Subclasses configuráveis) e permite que suas subclasses liguem ou desliguem funcionalidades específicas.

A `tester_node` gerencia se o nó testado é fixo ou variável e, no segundo caso, abre o campo que permite definir os Grupos a serem usados como escopo de busca, seguindo a Decisão de projeto 6 (Grupos de nós). Já a `data_variable` ativa ou desativa o uso de variáveis de dados, uma nova funcionalidade introduzida a partir da Decisão de projeto 7 (Variáveis de dados). Por fim, `data_extraction` permite utilizar pré-configurações que extraem um dado de uma propriedade ou chamada de método, algo pensado para ser complementar à opção anterior.

Ademais, todos os métodos do pareamento atômico abstrato são definidos como passos de `is_satisfied(bindings)`, o qual segue o **Método *Template*** de Gamma *et al.* (1994), algo que foi implementado em decorrência da Decisão de projeto 3 (*Template Method*). Porém, os métodos não definem simplesmente uma sequência linear de passos: a configuração das variáveis determina quais métodos devem e não devem ser implementados, de maneira que o usuário pode adaptar sua nova classe às opções fornecidas.

Finalmente, observando a classe `AbstractAction`, temos que a propriedade `agent_nodes` possui um papel semelhante à `tester_node`, sendo responsável por gerenciar se os alvos das ações são nós fixos, Grupos de nós ou variáveis. Dessa forma, a propriedade atua como uma *flag* que seleciona a configuração desejada para o comportamento da ação, de maneira semelhante ao pareamento abstrato visto anteriormente.

Caso `agent_nodes` esteja ligado, o usuário só precisa implementar `trigger_node(node, bindings)`, pois esse método atua como um passo de `trigger()`, o qual também segue o ***Template Method*** (Gamma *et al.*, 1994). Isso demonstra que a criação de novas ações também foi pensada para ser o menos trabalhosa possível, algo muito importante para nosso objetivo de garantir extensibilidade do sistema.

4.1.1 Análise de objetivos e requisitos

Após descrever a arquitetura e os fatores que a levaram a ser do jeito que é, agora nos resta analisar se ela cumpre os requisitos de *software* e os objetivos do projeto. Re-

capitulando o que foi dito no Capítulo 2, o principal objetivo do trabalho era fornecer um arcabouço que facilite a implementação de um Sistema Baseado em Regras em um motor de jogos. Analisando o *plugin* entregue, podemos afirmar que isso é cumprido, pois fornecemos a estrutura completa para execução de regras, com um subsistema de unificação de variáveis já incluso, tudo integrado ao motor de jogos *Godot*.

Outro objetivo era fornecer alguns módulos reutilizáveis que pudessem ser utilizados diretamente por quem instalou o programa, o que foi feito por meio dos recursos prontos disponibilizados. Além disso, era desejado que o usuário pudesse expandir esses módulos de maneira fácil e prática, algo que serviu de motivação para várias Decisões de projeto: 3 (*Template Method*), 9 (Subclasses configuráveis) e 10 (Sintaxe automática), todas elas com efeitos perceptíveis na versão final do sistema.

A última meta com impactos na arquitetura era que o *software* fosse fácil de ser utilizado e compreendido, em especial para pessoas sem conhecimento técnico de programação. Para atender a esses desejos, a principal funcionalidade implementada foi a interface gráfica, a qual permite declarar e editar regras visualmente, sendo possível observar seu progresso por meio das diversas imagens demonstradas ao longo do Capítulo 3.

Por outro lado, havia também a intenção de fornecer uma interface semelhante às encontradas nos jogos *Dragon Age II* (*BioWare*, 2011) e *Final Fantasy XII: The Zodiac Age* (*Square Enix*, 2017), que podem ser vistas nas Figuras 1 e 2, respectivamente. Sabendo isso, notamos que a interface baseada em texto se aproxima mais desses exemplos do que a gráfica, pois permite a declaração mais direta das regras, portanto, podemos entender que o uso conjunto de nossas duas interfaces cumpre os dois objetivos propostos.

Ainda sobre a *TUI*, nota-se que, apesar de ser pensada para programadores, a sintaxe utilizada não é muito diferente do que está presente nos exemplos lúdicos da Seção 1.2. Adicionalmente, como temos os botões que inserem os modelos de cada pareamento e ação, a pessoa descrevendo as regras não precisa saber a formatação *JSON* de cor, algo que diminui consideravelmente a curva de aprendizado necessária para o uso dessa interface.

Concluindo os objetivos abstratos do projeto, trataremos agora dos requisitos do sistema de *software*, na ordem em que aparecem na Tabela 2.1. Para começar, é possível criar regras com acesso a praticamente todos os elementos da *Godot Engine*, desde propriedades (`Numeric` e `StringMatch`, `SetPropertyAction`) e métodos (`Numeric` e `StringMatch`, `CallMethodAction`), sinais (`EmitSignalAction`), até relações específicas entre nós (`Hierarchy`, `Distance` e `AreaDetectionMatch`). Com isso, o requisito **F1** está cumprido.

Vale apontar que, além das funcionalidades prontas, o usuário pode criar novos recursos que testem ou acionem qualquer aplicação do motor de jogos, basta seguir a interface definida para os métodos. Dessa maneira, é possível definir regras mais exóticas, que manipulem configurações próprias do Editor, mexam com arquivos no disco, ou acessem a *internet*, por exemplo. As únicas exigências são que o usuário crie instâncias de `AbstractAtomicMatch`, implementando `is_satisfied(bindinds)` e de `AbstractAction`, implementando `trigger(bindings)`.

Continuando, vemos que a estrutura da regra cumpre seus dois requisitos: a condição é

composta de uma árvore com operadores *booleanos* e pareamentos atômicos (**F2**), e as ações são organizadas em um vetor, na ordem em que devem ser executadas (**F3**). Também fica claro que a definição do SBR cumpre mais dois requisitos: os árbitros definem estratégias de arbitragem de regra (**F4**) e a propriedade *Iteration Update* permite configurar iterações automáticas ou manuais do sistema (**F5**).

Quanto à possibilidade de criar novos componentes (**F6**), temos que isso foi amplamente discutido ao longo da Seção 3.3.1 e será novamente abordado na Seção 4.2.4. O mesmo se aplica ao uso de variáveis (**F7**), algo que foi trabalhado e expandido a partir da versão *Alpha* do *plugin*, com a explicação completa da implementação do sistema de unificação na Seção 3.2.1 e a introdução das variáveis de dados na Seção 3.3.

O último requisito funcional diz respeito à interface gráfica (**F8**), algo que foi implementado no começo do desenvolvimento do *plugin*, sendo demonstrado continuamente por meio das cenas de teste. Em suma, é possível perceber que os requisitos funcionais são os mais perceptíveis na arquitetura do sistema, visto que definem as funcionalidades básicas que precisavam ser implementadas o quanto antes.

Olhando agora para os requisitos técnicos da Tabela 2.1, vemos que eles representam detalhes de implementação menos notáveis a partir da estrutura de classes, logo, exigem mais atenção. Primeiramente, para garantir compatibilidade com futuras atualizações da *Godot Engine 4.1+* (**T1**), foi tomado o cuidado de não se usar nenhuma classe marcada como experimental ou obsoleta. Além disso, foi seguida a documentação oficial dessa versão (Linietsky *et al.*, 2014), obedecendo aos padrões sugeridos de como implementar um *plugin* e outros componentes.

Semelhantemente, a compatibilidade com o *gd-plug* (**T2**) foi assegurada através da leitura de seu manual e do cumprimento de seus requisitos de uso. Por fim, o último tópico satisfeito por meio da consulta de documentação externa foi o uso da linguagem *GDScript* (**T3**). Vale destacar que, por ser a linguagem oficial da *Godot*, o editor de texto embutido no Editor da *engine* apresenta documentação e preenchimento automático.

Agora, um requisito que exigiu esforço adicional foi garantir que não é preciso alterar a estrutura de diretórios para se utilizar o *plugin* (**T4**). Para isso, foi necessário descobrir uma forma de buscar os novos arquivos feitos pelo usuário sem saber sua localização, o que foi feito recorrendo à lista global de todas as classes implementadas no projeto. Dessa forma, o `RuleDB` consegue acesso ao caminho de todos os *scripts* relevantes ao sistema, salvando-os em seu "banco de dados" para serem consultados quando necessário.

Outro requisito que demandou trabalho dedicado foi a proibição de se depender da hierarquia da cena (**T5**), algo que afeta principalmente a busca por nós que podem substituir uma variável. A maneira encontrada de resolver esse problema foi com o uso de Grupos (conforme descrito na Decisão de projeto 6), os quais permitem que as regras tenham acesso a qualquer nó da cena atual, independentemente de sua posição na árvore, contanto que ele esteja etiquetado corretamente.

Por outro lado, o requisito que diz que não devemos depender de configurações do projeto (**T6**) não foi difícil de ser cumprido, pois nenhuma funcionalidade consulta ou altera uma configuração específica. Igualmente, não foi necessário dar atenção especial ao requisito que proíbe a compilação do código (**T7**), pois o uso da linguagem nativa e a falta

de bibliotecas externas assegura que não há o que ser compilado. Em particular, o *plugin* foi projetado para não ter dependência de nenhum *software* adicional.

Enfim, chegamos aos requisitos de experiência de usuário (*UX*). Sobre a integração inaparente com o Editor nativo (**X1**), sabemos que isso diz respeito basicamente às interfaces de usuário, portanto, devemos analisar se elas se destacam ou se mesclam ao Editor. Para a interface gráfica, vemos que a escolha pelo Inspetor (Decisão de projeto 2) praticamente oculta a implementação de nossas funcionalidades, pois elas se baseiam em complementar os campos que já estão presentes nativamente.

Em compensação, o *Rules Editor* é uma adição perceptível ao usuário, por acrescentar um novo tipo de painel inferior. Porém, devido ao uso do nó de editor de código (`CodeEdit`) e das mesmas cores de destacamento de sintaxe, notamos que ele é praticamente idêntico ao editor de texto embutido, o que o faz ter uma aparência familiar aos usuários da *Godot*. Além disso, os botões presentes seguem a formatação padrão e não há nenhum ícone que o faça destoar do resto da interface nativa.

Sobre o uso dos *widgets* para propriedades de tipos nativos (**X2**), temos que isso é decorrência imediata do uso do Inspetor, portanto, esse requisito é facilmente cumprido. Outra exigência atendida prontamente são os exemplos de uso (**X3**), pois a elaboração de cenas de teste ao longo do desenvolvimento já cumpre essa função — podemos entender que os testes realizados demonstram as funcionalidades e, ao mesmo tempo, as testam.

Por último, sobre a escrita de uma boa documentação (**X4**), aponta-se sua presença no arquivo `README.md` (em inglês), na raiz do repositório[4]. Nele, consta a *API* das classes principais, o passo-a-passo para a elaboração de novas classes e mais outras informações. Para facilitar a leitura do arquivo em inglês, a Seção 4.2 trará as informações essenciais da documentação, além de complementá-la com explicações detalhadas do estado final do sistema.

4.2 Manual do usuário

Nesta seção são dadas todas as informações relevantes a um usuário do *plugin*, incluindo as instruções de como utilizá-lo, desde sua instalação até a adição de novos recursos, além da listagem do conteúdo extra presente no repositório. O intuito é mostrar a documentação feita para o programa, mas de uma forma que explicite as funcionalidades presentes na versão final do sistema.

Primeiramente, falaremos sobre os diretórios presentes no repositório, pois nem todos fazem parte da aplicação em si, havendo outros tipos de conteúdo que podem ser úteis ao usuário. O principal recurso extra são as cenas de teste, contidas no diretório `test_scenes`, as quais foram utilizadas ao longo do desenvolvimento e mencionadas através do Capítulo 3. Elas podem ser consideradas uma demonstração das capacidades do sistema e, portanto, servem como exemplo para as pessoas que desejam desenvolver jogos com o *plugin*.

Sabendo disso, a melhor forma do usuário experimentar as funcionalidades do sistema antes de aplicá-las em seu projeto é por meio das cenas de teste. Portanto, recomendamos

[4] Leia o *README* em: https://github.com/rvbatt/rule-based-godot/blob/main/README.md

que a pasta `test_scenes` seja baixada em adição aos itens essenciais do *software*, algo que ocorre quando o repositório é clonado, mas precisa ser feito manualmente ao se utilizar o instalador do *gd-plug* (mais sobre isso na Seção 4.2.1).

Ademais, o repositório contém os diagramas mostrados ao longo dessa monografia: a arquitetura planejada (Figura 2.1), o exemplo de uma condição representada por uma árvore de pareamentos (Figura 2.2), a execução de uma condição composta com um variável compartilhada (Figuras 3.8 a 3.13) e a arquitetura final (Figura 4.4). Eles servem como uma documentação extra e podem ajudar desenvolvedores que desejam estudar SBR mais a fundo.

Por fim, temos um diretório com *templates* para as classes abstratas de: pareamentos *booleanos*, pareamentos atômicos, ações e árbitros. Esses modelos servem como artifício para garantir que desenvolvedores usuários consigam estender o *plugin* da maneira mais simples possível, algo que será abordado na Seção 4.2.4. Com isso, os aspectos relevantes do repositório foram cobertos, logo, podemos partir para o manual de instalação do *plugin*.

4.2.1 Instalação do *plugin*

Há duas formas de se instalar o *plugin Rule-Based Godot*:

- **Por meio do *gd-plug***. Supondo que o gerenciador de *plugins* já está instalado (instruções se encontram em seu repositório[5]), é preciso:

  1. Adicionar o arquivo `plug.gd`, ilustrado no Programa 4.1, na raiz do projeto da *Godot*. Caso ele já exista, basta adicionar as linhas *4* e *5* na função `_plugging()`.

Programa 4.1 Arquivo `plug.gd`, com configuração recomendada para a instalação do *addon* via o gerenciador *gd-plug*.

```
1  extends "res://addons/gd-plug/plug.gd"
2
3  func _plugging():
4      plug("rvbatt/rule-based-godot", {"include": ["addons/rule_based_godot/",
5                  "test_scenes/", "script_templates/"]})
```

  É possível aplicar configurações adicionais, como instalar uma versão específica, ou atualizar a aplicação automaticamente quando um novo *commit* é feito no repositório. Para mais informações sobre isso, veja a documentação de *gd-plug*.

  2. Ainda na raiz do projeto, executar o seguinte comando no terminal:

     ```
     godot --no-window -s plug.gd install
     ```

  3. No Editor da *Godot*, entrar no menu de *Project*, clicar em *Project Settings*, ir na aba de *Plugins* e ligar o *Rule-Based Godot*, acionando o campo *Enabled*. A Figura 4.5 demonstra a ativação desse campo.

[5] Veja as instruções de instalação do *gd-plug* em: https://github.com/imjp94/gd-plug

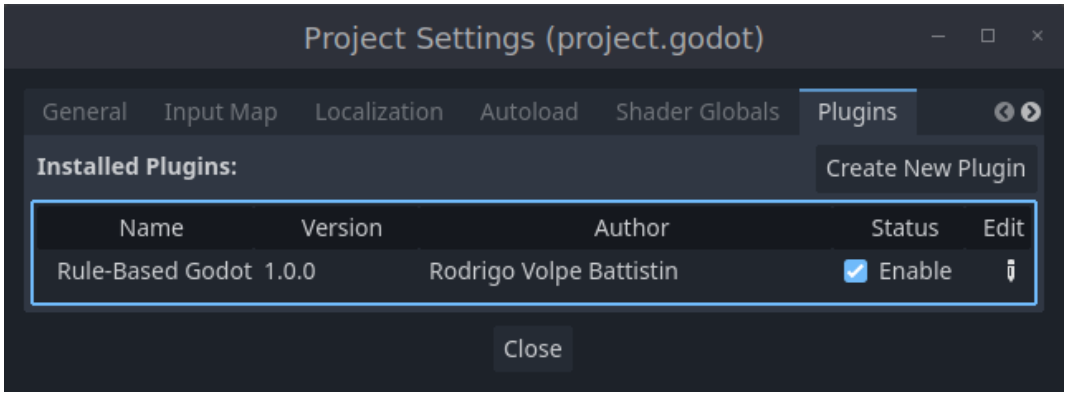

Figura 4.5: *Aba de Plugins do Project Settings, com Rule-Based Godot ativo (versão 1.0.0).*

- **Acessando diretamente o repositório**[6]. Para isso, você deve:

  1. Baixar, ou clonar o repositório. Recomenda-se utilizar a última versão presente no *main branch*. Se for baixado, ele virá em um arquivo `.zip` e precisa ser descompactado.
  2. Copiar o diretório `addons/rule_based_godot` para a raiz do projeto da *Godot*. Caso a pasta `addons` já exista, copie apenas o subdiretório e coloque-o na pasta.
  3. (Opcional) Copie os diretórios `test_scenes` e `script_templates` para o projeto. Se a pasta de *templates* já exisitir, copie apenas os subdiretórios.
  4. Abrir o projeto no Editor da *Godot* e seguir os mesmos passos do item 3 da instalação via *gd-plug*.

Não foi possível adicionar o *plugin* na *Godot Asset Library* até a entrega desta monografia, mas isso está planejado para um futuro próximo. Após a submissão na biblioteca oficial de recursos, a instalação poderá ser feita diretamente no Editor, por meio do *workspace AssetLib*: basta procurar por "*Rule-Based Godot*", instalar todos os arquivos presentes e depois acionar o *plugin*, por meio do mesmo menu demonstrado na Figura 4.5.

4.2.2 Adição de regras em uma cena

Para configurar o Sistema Basedo em Regra na cena desejada, siga os seguintes passos, utilizando a Figura 4.6 como referência:

1. Adicione um nó do tipo `RuleBasedSystem` à cena, por meio do menu ilustrado na Figura 3.1;
2. Configure a frequência de iteração do sistema, editando a propriedade *Iteration Update* no Inspetor. As opções são:
   - `Every frame`: itera todo *frame* de física. **Tome cuidado ao utilizar essa opção**, pois um conjunto muito grande de regras pode prejudicar o desempenho do jogo;

[6] Acesse o repositório em: https://github.com/rvbatt/rule-based-godot

- `On Timer`: itera a cada *wait_time* segundos, propriedade presente na categoria *Timer*;
- `On Call`: só itera quando a função `iterate()` é chamada. É possível conectá-la a sinais[7] para ter maior controle.

3. Adicione um árbitro, na propriedade *Arbiter*. **Não escolha o *AbstractArbiter***, pois essa é a classe abstrata.

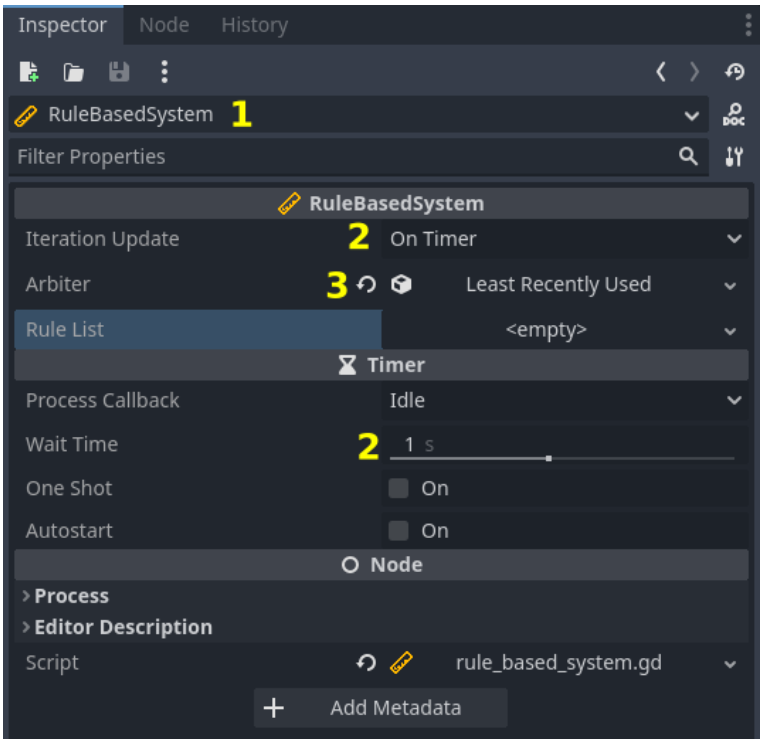

Figura 4.6: *Inspetor marcado com as configurações necessárias de um Sistema Baseado em Regras (versão 1.0.0). Os números em amarelo seguem os passos enumerados: no 1, criamos e selecionamos um nó do tipo `RuleBasedSystem`; no 2, configuramos a iteração automática do sistema a cada 1s; no 3, adicionamos um árbitro que implementa o método Least Recently Used.*

Agora, é preciso escolher se será usada a interface gráfica ou textual para declarar as regras. Se foi escolhida a interface gráfica, continue olhando para o *Inspector* (utilize a Figura 4.7 como guia) e prossiga da seguinte forma:

4. Crie um novo `RuleList` na propriedade de mesmo nome e comece a adicionar regras no vetor, quantas forem necessárias;
5. Para cada regra, crie seus componentes, **nunca escolhendo as opções que começam com *Abstract***:
    i Crie a condição. Se você escolher um pareamento *booleano*, crie suas subcondições repetindo esse passo. Se você escolher um pareamento atômico, edite suas propriedades, não se esquecendo de expandir os agrupamentos;

[7] Veja como utilizar sinais em: https://docs.godotengine.org/en/4.1/getting_started/step_by_step/signals.html

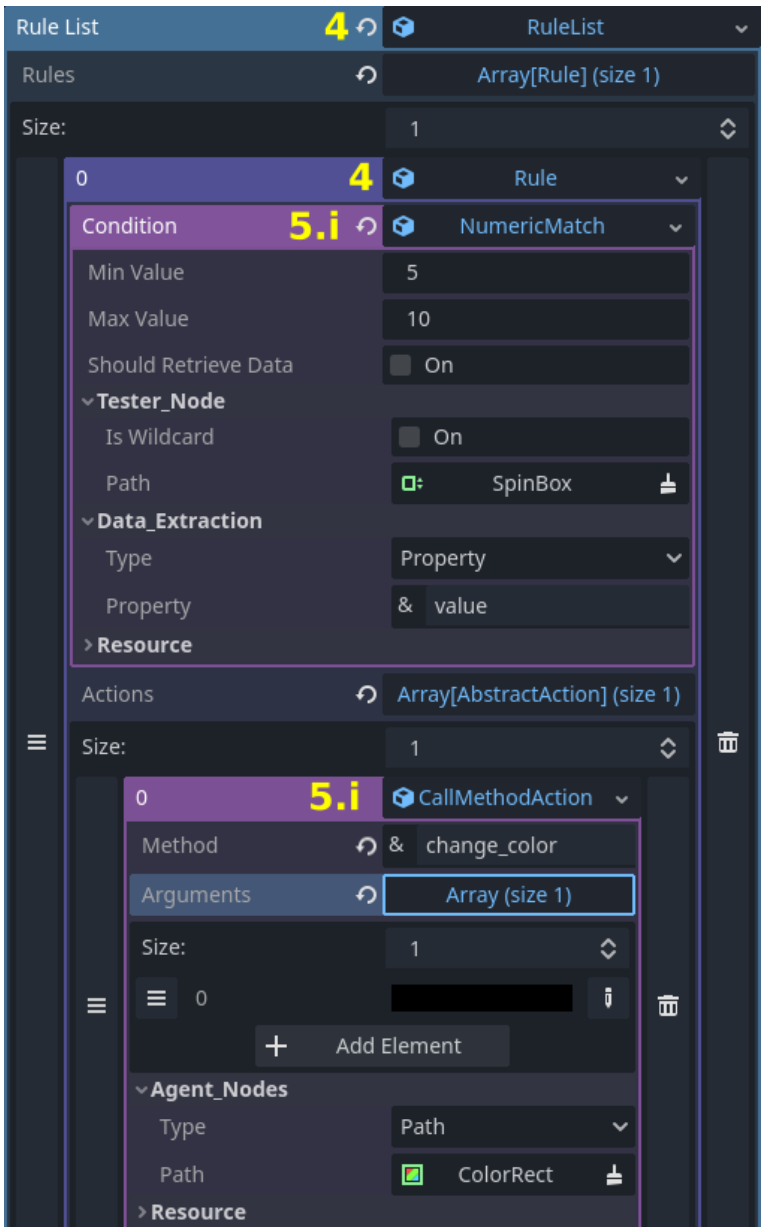

Figura 4.7: *Inspetor marcado com a declaração de regras via interface gráfica (versão 1.0.0). Os números em amarelo seguem os passos enumerados: no 4, criamos a* `RuleList` *e instanciamos uma regra; no 5.i, definimos a condição como um* `NumericMatch`*, preenchendo todas as suas propriedades; no 5.ii, adicionamos uma* `CallMethodAction` *no vetor de ações e atribuímos suas propriedades.*

ii Adicione uma ação no vetor, depois edite suas propriedades, incluindo as que estão ocultas nos agrupamentos. Repita esse passo quantas vezes for necessário.

6. (Opcional) Salve os componentes que deseja reutilizar em um arquivo `.tres`, basta clicar na seta para baixo ao lado de seu nome (∨) e depois em *Save*. É possível salvar a lista inteira, uma regra, uma condição (árvore de pareamentos) ou uma ação.

Por outro lado, caso deseje utilizar a interface textual, volte sua atenção para o *Rules Editor* e siga os seguintes passos, observando as Figuras 4.8, 4.9 e 4.10:

4. Abra o painel inferior *Rules Editor* e use o botão de `Reset` para restaurar o texto;

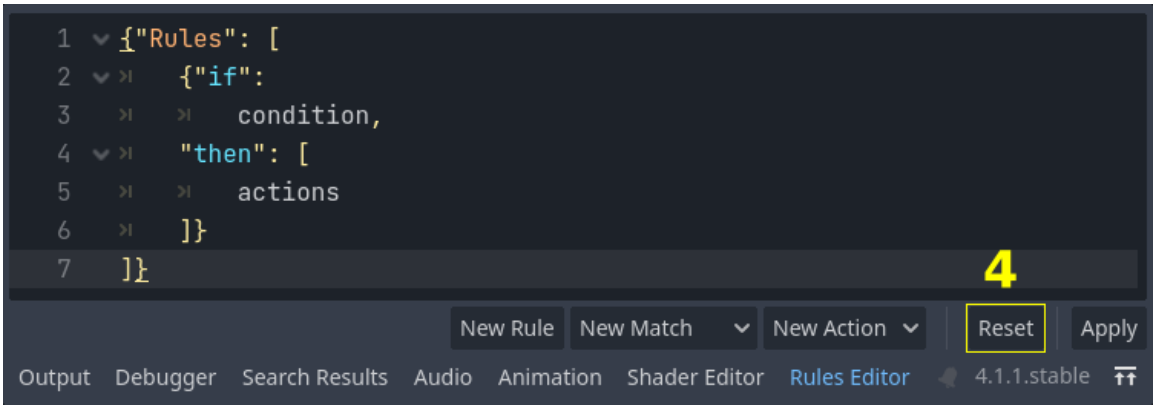

Figura 4.8: *Rules Editor com texto padrão, no estágio inicial de declaração de regras (versão 1.0.0). A marcação em amarelo indica que apertamos o botão de Reset no passo 4, o que faz o editor de texto aparecer desta maneira.*

5. Adicione novas regras com o botão `New Rule`, tomando cuidado com o ponto de inserção de texto, pois o modelo será inserido exatamente na mesma posição;
6. Para cada regra, substitua os *placeholders* da condição e do vetor de ações, utilizando os botões correspondentes para inserir os modelos. Tome sempre cuidado com a posição do cursor:
    i Apague a palavra "*condition*", mantenha o cursor na posição onde ela estava e clique no botão `New Match`. Escolha uma das opções do menu que surgiu. O formato *JSON* desse pareamento será inserido no editor;
    ii Siga a sintaxe explicada na documentação[8] e substitua os campos. Se você escolheu um pareamento *booleano*, haverá outra palavra "*condition(s)*", assim, repita o passo i. Se você escolheu um pareamento atômico, apenas substitua as propriedades;
    iii Apague "*actions*" e clique no botão `New Action`, depois selecione uma das opções. O modelo dessa ação será inserido onde estava o ponto de inserção;
    iv Substitua os *placeholders* das propriedades pelos valores que deseja, seguindo a sintaxe;
    v Repita os passos iii (sem a remoção da palavra "*actions*") e iv para adicionar mais ações, quantas forem necessárias.
7. Após declarar todas as regras, clique no botão `Apply` para aplicá-las no `RuleBasedSystem` atualmente selecionado. Caso a sintaxe *JSON* esteja incorreta, um erro aparecerá no *Output* e a operação será cancelada;
8. (Opcional) Salve o texto criado em um arquivo `.json`, caso deseje reutilizar alguma parte.

[8] Veja a explicação da sintaxe no *README* do repositório: https://github.com/rvbatt/rule-based-godot

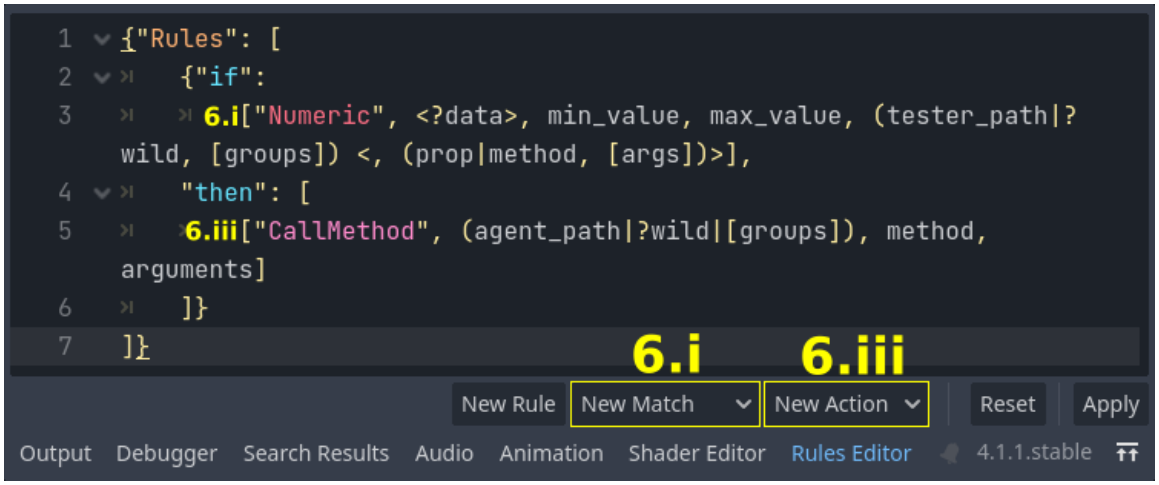

Figura 4.9: *Rules Editor com modelos de `NumericMatch` e `CallMethodAction` (versão 1.0.0). As marcações em amarelo indicam os passos associados a cada aspecto da imagem: no 6.i, utilizamos o botão `New Match` e escolhemos a opção "Numeric" para inserir o modelo presente na linha 3 do código; no 6.iii, clicamos em `New Action` e depois selecionamos o "CallMethod", assim adicionando o modelo da linha 5. Repare que não estão mostrados os efeito dos passos 6.ii e 6.iv, pois assim podemos visualizar os dois modelos de uma vez. Os passos pulados aparecem na Figura 4.10.*

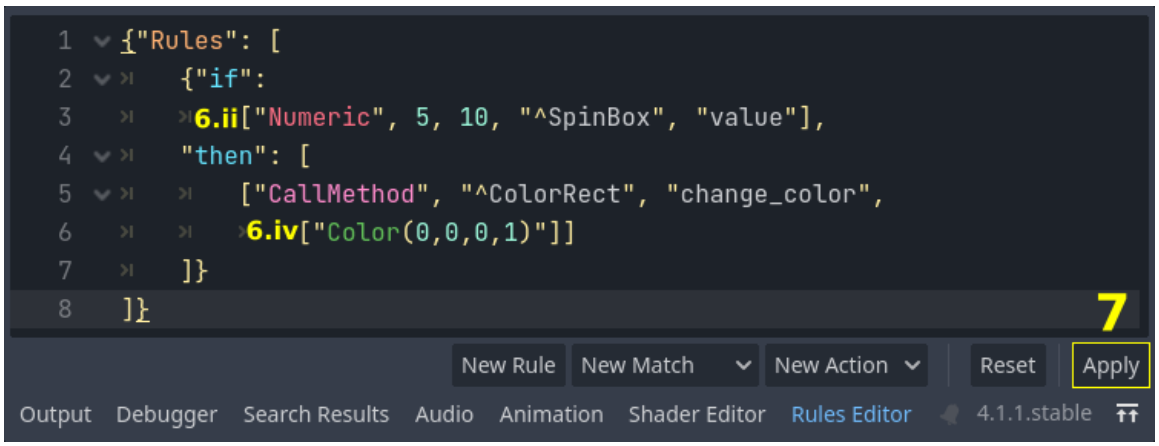

Figura 4.10: *Rules Editor preenchido com pareamento numérico e ação de chamada de método (versão 1.0.0). As marcações indicam em qual passo tratamos de cada item: no 6.ii, preenchemos o modelo de `NumericMatch`; no 6.iv, o de `CallMethodAction`; por fim, no 7 clicamos no botão `Apply` para aplicar as regras no sistema.*

Após isso, o sistema estará configurado, logo, resta apenas salvar a cena e executá-la. Repare que foi dado o passo-a-passo para a adição de regras a partir do zero, porém, seria possível pular algumas etapas caso tivéssemos componentes já salvos. Para carregar recursos no Inspetor, basta clicar na seta para baixo ao lado do campo de edição da propriedade, depois em *Load*, ou *Quick Load*. Já no editor de texto, é possível copiar arquivos `.json` ou regras declaradas em outros locais.

Vale destacar que as Figuras 4.6 a 4.10 servem também para ilustrar a aparência final das interfaces, assim fornecendo uma visão geral de como o usuário interage com o sistema. As escolhas dos tipos de ação e pareamento foram ocultadas, por ser algo que pôde ser

observado ao longo dos testes do Capítulo 3. Contudo, um ponto importante que não pode deixar de ser abordado é o uso de variáveis, algo que será visto na Seção 4.2.3.

4.2.3 Uso de variáveis

Há dois tipos de variáveis que podem ser utilizadas na declaração de uma regra: nós *wildcard*, que passam pelo processo de unificação, e variáveis de dados, que não. Ambos utilizam a mesma notação na interface textual: são expressos por um ponto de interrogação ("?") seguido de um identificador, para ser possível diferenciar as variáveis de termos constantes, conforme foi explicado na Seção 1.2.3.

Outro ponto em comum entre os dois tipos é que seus identificadores são restritos ao escopo da regra, ou seja, é possível usar o mesmo nome para variáveis em regras distintas sem que elas interajam entre si. Por exemplo: se temos uma regra que define quando um inimigo deve pular e outra para quando ele deve correr, ambas podem utilizar uma variável chamada *?inimigo* sem ter o risco de uma mesma personagem sair correndo e pulando ao mesmo tempo.

Em contrapartida, a interpretação do que cada tipo de variável representa é muito diferente, o que faz com a utilização de cada uma também seja. Portanto, explicaremos um por vez:

◇ **Nós *wildcard***

Devem ser pensados como as entidades controladas pela regra. Quando o mesmo nome aparece mais de uma vez na condição, isso implica que se trata da mesma entidade, ou seja, se a variável aparece em três pareamentos, o mesmo ser ou objeto precisa satisfazer os três. Além disso, se uma ação é aplicada em um nó *wildcard*, isso significa que todas as entidades que atendem à condição irão executar o que é descrito.

Dessa forma, regras modeladas com esse tipo de variável atuam sob categorias de entidades, servindo como definições genéricas para o comportamento de um tipo de personagem, objeto, ambiente, etc. Essa é a mesma definição dada na Seção 1.2.3, portanto, os nós *wildcard* passam pelo processo de unificação, garantindo que todas as suas ocorrências possam ser substituídas por um mesmo elemento.

Para utilizar esse tipo de variável, atente-se aos agrupamentos *Tester Node*, nos pareamentos e *Agent Nodes*, nas ações. Nos pareamentos, é preciso ligar o campo *Is Wildcard*, o que faz surgir as propriedades *Search Groups* e *Identifier*; a primeira serve para definir a lista dos Grupos de nós que devem ser buscados, enquanto a segunda representa o nome da variável. **Não utilize** o prefixo "?" no preenchimento do identificador, ele é implícito.

Já nas ações, é preciso configurar o tipo (*Type*) dos agentes para *Wildcard*, o que novamente faz surgir a propriedade *Identifier*. De maneira semelhante ao que foi visto com os pareamentos, não se deve iniciar a *string* do identificador com um ponto de interrogação. Enfim, a Figura 4.11 ilustra o uso desses dois conjuntos de propriedades.

◇ **Variáveis de dados**

Representam uma informação, um valor que determina se o pareamento é satisfeito ou não. Devem ser pensadas como uma forma de guardar o dado mais importante do

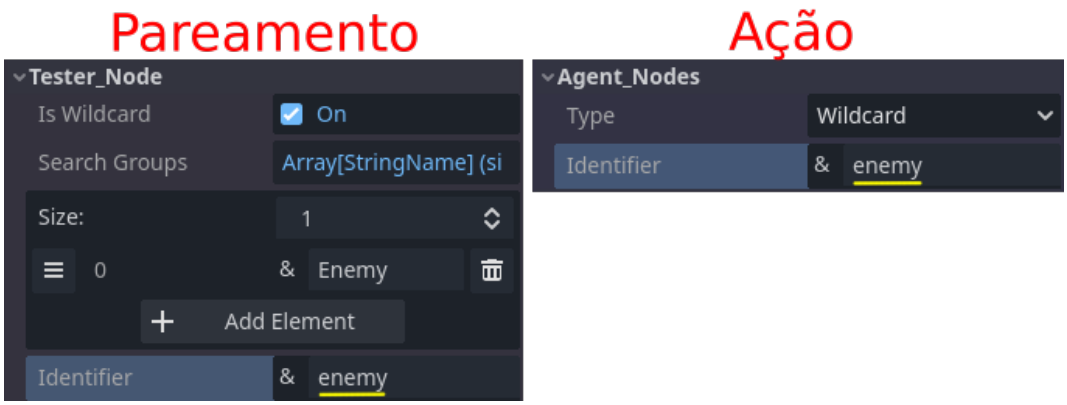

Figura 4.11: *Inspetor com configuração de uso para uma variável do tipo nó wildcard. Repare que o pareamento e a ação utilizam a mesma variável ?enemy, o que significa que ela será unificada para garantir que todos os inimigos que satisfazem a condição executarão a ação.*

pareamento, e não como a entidade que é testada. Ao contrário dos nós *wildcard*, um mesmo identificador não deve ser utilizado mais de uma vez na mesma condição, pois a informação presente nessa variável será sobrescrita.

Dessa maneira, se uma regra possui dois pareamentos que utilizam uma variável de dados chamada *?dado*, ao final da verificação da condição teremos guardado a informação de somente um deles (o que foi testado por último), enquanto o valor do outro terá sido perdido. O motivo disso ocorrer é que não aplicamos unificação nesse tipo de variável, apenas salvamos diretamente o valor dado.

Ademais, as ações podem acessar o dado guardado, por meio do seu identificador, e utilizá-lo como argumento de um método, valor de uma propriedade, ou argumento de sinal. Dessa forma, é possível definir chamadas de funções com argumentos dinâmicos, recuperados pela regra durante a execução do jogo, ou alterar continuamente uma propriedade de um objeto.

Para utilizar esse tipo de variável, é preciso ativar o campo *Should Retrieve Data* de um pareamento e preencher *Data Variable* com o identificador que será usado (sem começar por ponto de interrogação). Agora, **nas ações é preciso utilizar o prefixo "?"**, pois ele irá substituir um argumento e, portanto, precisamos ser capazes de diferenciar um termo fixo de uma variável de dados. Se o identificador for usado sem o prefixo correto, ele será entendido como uma *string* constante, o que fará a regra não funcionar como o esperado. A Figura 4.12 demonstra a configuração do Inspetor para o uso correto.

4.2.4 Recursos disponíveis e como criar novos

A Tabela 4.1 contém a lista de todos os módulos reutilizáveis disponíveis na versão final do *plugin*. Ela contém o tipo, o identificador (nome único dentre recursos do mesmo tipo) e uma breve descrição do que cada um representa. Para ações e pareamentos, o identificador é o primeiro argumento do *array* de representação *JSON*.

Caso o desenvolvedor usuário deseje implementar novos tipos de árbitros, pareamentos ou ações, é altamente recomendado que ele baixe os *templates* dessas classes, presentes no

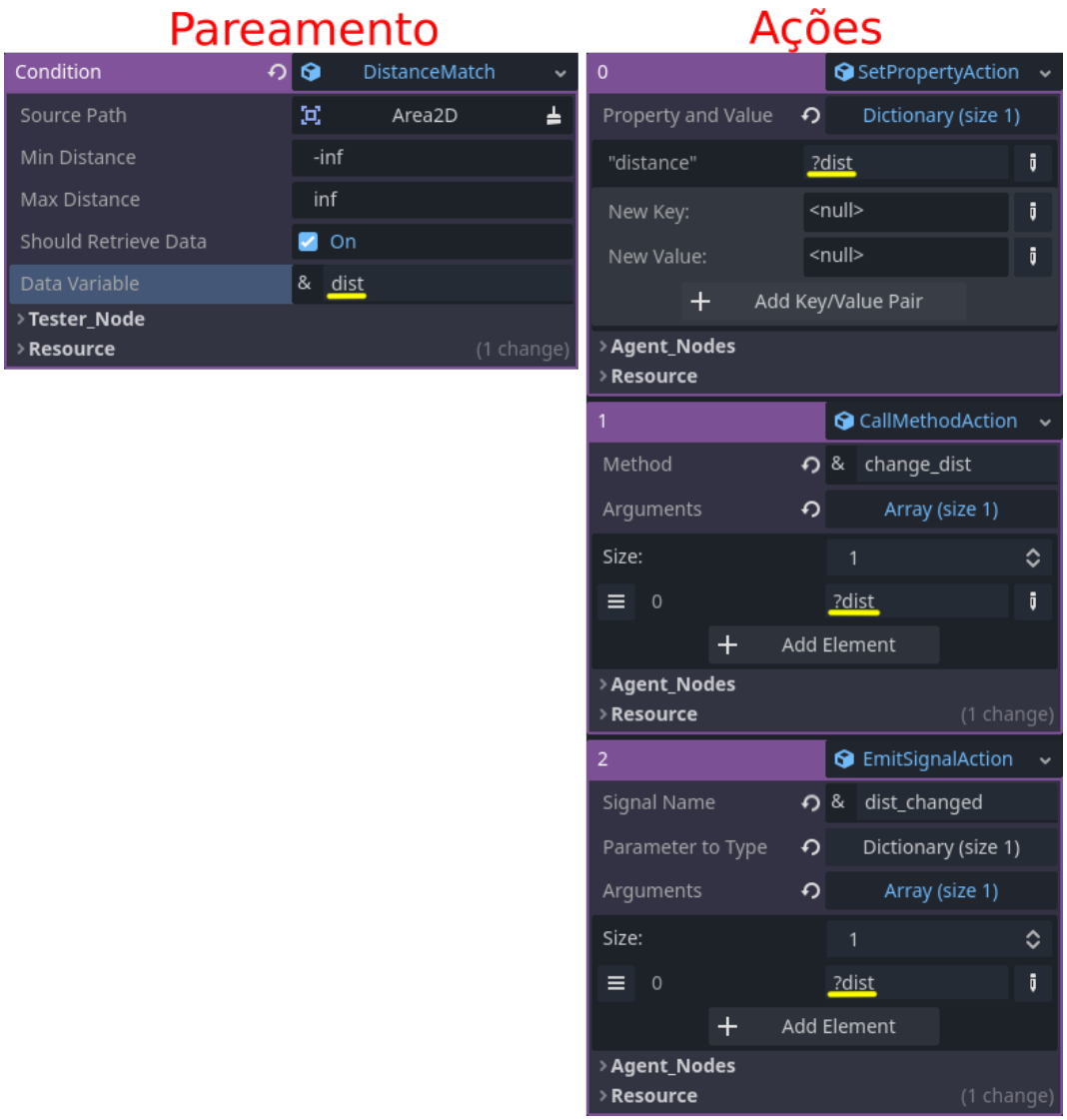

Figura 4.12: *Inspetor com configuração de uso para uma variável de dados. Repare que o pareamento e as três ações utilizam a mesma variável ?dist, o que significa que o valor salvo a partir do pareamento de distância será utilizado em todas as ações.*

repositório[9]. Esses modelos contêm comentários explicativos do que é necessário em cada classe, com as definições das funções que precisam e podem ser implementadas, além de indicações de como cada *flag* de configuração afeta o seu funcionamento.

O intuito de fornecer esses arquivos é integrá-los a uma funcionalidade nativa do Editor da *Godot*: o uso automático de *templates* na criação de novos *scripts*[10]. A Figura 4.13 demonstra como o usuário pode escolher um modelo quando está criando um novo arquivo de *script*; com essa opção, o arquivo criado será uma cópia do seu *template*, economizando a escrita de várias linhas de código.

[9] Baixe os *templates* em: https://github.com/rvbatt/rule-based-godot/tree/main/script_templates

[10] Leia mais sobre *script templates* na *Godot* em: https://docs.godotengine.org/en/4.1/tutorials/scripting/creating_script_templates.html

Tipo (sufixo)	Identificador	Descrição
Árbitro (*Arbiter*)	`FirstApplicable`	seleciona a primeira regra satisfeita, assumindo ordenação por prioridade
	`LeastRecentlyUsed`	seleciona a regra satisfeita que foi disparada há mais tempo
Pareamento *booleano* (*Match*)	`NOT`	operador NÃO
	`AND`	operador E, com número arbitrário de entradas
	`OR`	operador OU, com número arbitrário de entradas
Pareamento atômico (*Match*)	`Numeric`	testa se um valor numérico, obtido a partir uma propriedade ou chamada de método, está em um determinado intervalo
	`String`	testa se uma *string*, obtida a partir de uma propriedade ou chamada de método, é igual a uma constante
	`Hierarchy`	testa se dois nós respeitam uma das relações: primeiro é pai (*Parent of*) do segundo, primeiro é filho (*Child of*) do segundo, ou são irmãos (*Sibling of*)
	`Distance`	testa se a distância entre a origem de dois nós está em um dado intervalo
	`AreaDetection`	testa se há objetos (específicos ou quaisquer) em uma área
	`DistinctVariables`	assegura que certas variáveis possuem valores distintos quando é aplicada uma substituição
Ação (*Action)*	`SetProperty`	atribui um valor a uma variável de um nó
	`CallMethod`	chama um método de um nó, passando os argumentos definidos em um vetor
	`EmitSignal`	adiciona um sinal a um nó, caso ele não o possua, e o emite

Tabela 4.1: *Lista de todos os recursos prontos presentes na versão 1.0.0 do plugin.*

Para o Editor reconhecer os modelos, é preciso que eles estejam em uma pasta chamada `script_templates` localizada na raiz do projeto, da mesma maneira que no repositório. A estrutura de subdiretórios precisa ter um formato específico, por isso recomenda-se baixar a pasta inteira e depois movê-la para a raiz do projeto, ou copiar todo o seu conteúdo, caso esse diretório já esteja sendo usado. Vale apontar que é possível alterar o caminho padrão da pasta de *templates* por meio da propriedade `editor/script/templates_search_path`, presente no menu de *Project Settings*.

Não discutiremos os detalhes técnicos de cada *template* — caso o leitor deseje, os arquivos podem ser lidos no Apêndice B —, porém, serão feitos alguns apontamentos de como a arquitetura tem influência na construção de novas classes. Em suma, as duas Decisões de projeto que ficam evidentes ao desenvolvedor usuário são: a 3 (*Template Method*) e a 9 (Subclasses configuráveis).

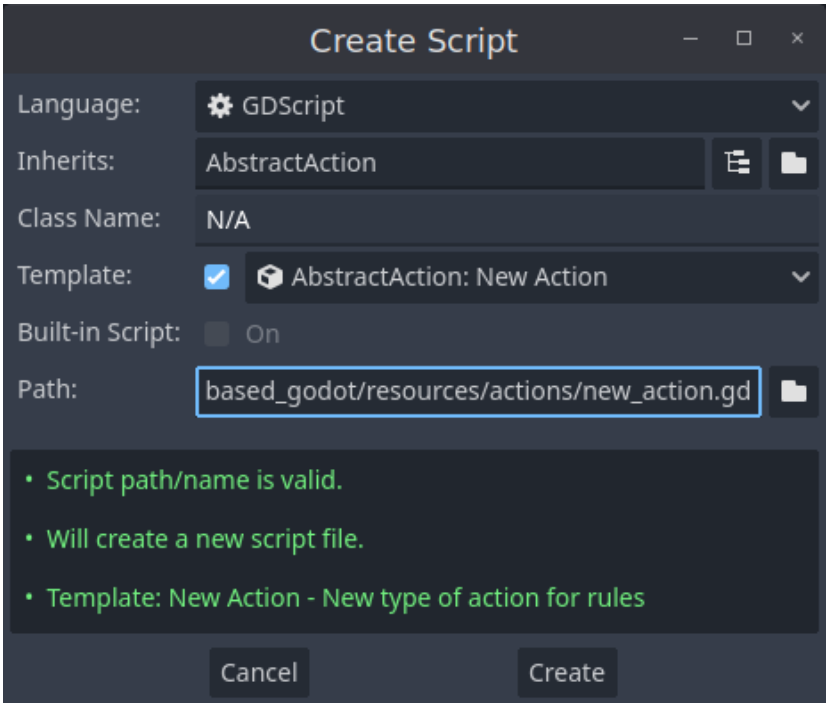

Figura 4.13: *Menu de criação de scripts, com uso de template para novo tipo de* `AbstractAction` *(versão 1.0.0).*

Elas são perceptíveis devido às *flags* de configuração, ativadas ou desativadas nos construtores, e das funções auxiliares, responsáveis por implementar o **Método *Template*** (Gamma *et al.*, 1994). Como cada combinação de configurações ativas exige a implementação de um conjunto específico de métodos, o usuário deve primeiro escolher quais funcionalidades deseja e, a partir delas, analisar quais métodos deve elaborar.

A classe na qual isso é mais aparente é *AbstractAtomicMatch*, cuja relação de configurações e métodos pode ser vista na Tabela 4.2. Podemos notar que a aplicação das *flags* especializa o pareamento: primeiro delimitando que ele testa nós, depois que a verificação se baseia em dados e, por fim, que serão usados métodos pré-definidos para se obter o dado.

Flags ativas	Métodos que **devem** ser implementados	Métodos que *podem* ser implementados
	`is_satisfied(bindings)`	
Tester Node	`node_satisfies_match(node, bindings)`	`get_candidates()`
Tester Node, *Data Based Node*	`get_data(node)`, `data_satisfies_match(data)`	`get_candidates()`
Tester Node, *Data Based Node*, *Get Node Data Preset*	`data_satisfies_match(data)`	`get_candidates()`

Tabela 4.2: *Relação entre flags de configuração e funçõs que devem ou podem ser implementadas em uma subclasse de AbstractAtomicMatch (versão 1.0.0).*

Capítulo 5

Conclusões

Neste trabalho foi elaborado um arcabouço que facilita a implantação de um Sistema Baseado em Regras em projetos de desenvolvimento de jogos com a *Godot Engine*, um motor de jogos de código aberto. Sendo um modelo de Inteligência Artificial, SBR permitem definir comportamentos de *bots* por meio de regras "se-então", de tal forma que essas personagens conseguem atuar sozinhas no mundo virtual e assim proporcionar interações com os jogadores.

Como motivação e métrica para o desenvolvimento dessa aplicação, temos exemplos de jogos que fornecem ao jogador a possibilidade de declarar regras para personagens aliadas automatizadas. Jogos como *Dragon Age II* (*BioWare*, 2011) e *Final Fantasy XII: The Zodiac Age* (*Square Enix*, 2017) fornecem interfaces gráficas nas quais o usuário pode selecionar a condição e a ação de cada regra, além de ordená-las com base em prioridade.

Sabendo disso e avaliando o produto final, percebemos que ele fornece todos os blocos de construção necessários para replicar os comportamentos presentes nos jogos supracitados. Tomando a Figura 1 como base, podemos argumentar que todas as regras apresentadas podem ser implementadas na última versão do sistema, utilizando somente os recursos disponibilizados (Tabela 4.1). A única questão nebulosa é o comando dado como ação da regra 4: "utilizar a condição atual na próxima tática", o que, no contexto do jogo, significa aplicar o operador *AND* entre essa condição e a seguinte, construindo uma condição composta para a ação dada na segunda tática.

Dessa forma, podemos considerar a união das duas últimas linhas como a tática 4 e partir para a análise de uma possível implementação da lista de regras. Em todos os casos, supomos que existe um método que dispara a habilidade descrita como ação, de forma que o uso de `CallMethodAction` é suficiente para esse componente das regras. Tratando agora das condições, podemos dizer que:

1. Para a condição de haver pelo menos dois inimigos cercando a personagem, a parte mais complexa é garantir que a condição encontrou dois inimigos distintos, evitando que uma mesma entidade seja pareada duas vezes. Uma forma de resolver isso é com três pareamentos conectados em um mesmo operador E: dois pareamentos de distância, tais que eles utilizam variáveis diferentes como alvo e se restringem ao Grupo de Inimigos; e um pareamento de variáveis distintas, o qual irá assegurar que

os dois pareamentos de distância encontraram valores distintos e, portanto, existem realmente dois inimigos diferentes próximos da personagem.

2. Para verificar a quantidade de vida da personagem, um pareamento numérico pode facilmente testar a propriedade da entidade fixa.

3. Para testar se uma aliada qualquer está com menos da metade da vida, podemos usar um pareamento numérico cujo alvo é um nó *wildcard* e que só realiza buscas no Grupo das Aliadas.

4. Para a condição composta, devemos analisar cada componente do E separadamente: podemos usar pareamentos de *string* para encontrar um inimigo que possui a personagem como alvo e que está no estado "Atacando com arma branca". Esse pareamento deve buscar no Grupo Inimigos por qualquer nó que satisfaça tais exigências, salvando os candidatos válidos em uma variável.

   Tendo isso, usamos o conector *booleano* e definimos o segundo pareamento: para testar se o mesmo inimigo está na faixa de curta distância, declaramos um pareamento de distância que se aplica ao Grupo Inimigos e utiliza o mesmo identificador de variável. Dessa forma, a entidade alvo será unificada e a regra só será acionada se um mesmo *bot* estiver atacando nossa personagem com uma arma branca e em curta distância.

Portanto, concluímos ser possível declarar regras semelhantes às de *Dragon Age II* (*BioWare*, 2011), sendo necessário somente mais algumas configurações para termos um comportamento análogo ao das personagens do jogo. Considerando uma implementação em *Godot*, o `RuleBasedSystem` estaria na mesma cena da personagem sendo controlada, portanto, bastaria selecionar o árbitro que usa o método *First Applicable* e configurar a iteração do sistema para todo *frame* de física (60 vezes por segundo). Com isso, teríamos uma aliada que age por conta própria, seguindo as mesmas táticas expressas na Figura 1 e que está sincronizada com a taxa de atualização do mundo do jogo.

Para implementarmos algo semelhante aos *gambits* de *Final Fantasy XII: The Zodiac Age* (*Square Enix*, 2017), ilustrados na Figura 2, o processo seria o mesmo, não havendo nenhuma característica especial que nos leve a crer que não seria possível replicar as regras desse jogo. Dessa forma, vemos que nossa métrica para o desempenho do projeto é satisfeita, ao demonstrarmos que o *plugin* apresenta poder expressivo o suficiente para montar comportamentos semelhantes aos presentes nos jogos de referência.

Além da métrica comparativa para o poder expressivo do sistema, tínhamos alguns objetivos amplos para o trabalho, os quais foram tratados, em sua maioria, na Seção 4.1.1. Os objetivos que não foram abordados nessa Seção são aqueles que não dizem respeito ao conteúdo do *software* em si, mas de aspectos relacionados ao projeto de desenvolvimento: a adoção de princípios de código livre e o estudo de conteúdos teóricos.

Começando pelos ideais de código aberto, foi definido que o *plugin* teria a mesma licença MIT da *Godot Engine* e que o projeto inteiro seria desenvolvido em um repositório aberto ao público do *GitHub*[1]. Como o código-fonte se encontra nesse local de acesso público e temos

[1] O repositório pode ser acessado no Apêndice A, ou diretamente em: https://github.com/rvbatt/rule-based-godot

a licença protegendo-o, vemos que o programa segue os quatro princípios de *software* livre definidos pela *Free Software Foundation*[2]: **(0)** o programa pode ser executado da maneira que o usuário quiser; **(1)** o usuário pode estudar e alterar o código-fonte; **(2)** é permitido redistribuir cópias da aplicação; **(3)** é permitido redistribuir cópias modificadas.

Dessa forma, fica claro que o *Rule-Based Godot* pode ser considerado um *software* de código aberto; unindo isso ao fato de ser gratuito para baixar e instalar, podemos afirmar que ele se enquadra na categoria de *FOSS*, assim como o motor de jogos *Godot*. Como dito na Introdução, isso vai ao encontro do nosso objetivo de tornar a tecnologia de SBR o mais transparente e acessível possível, permitindo que qualquer desenvolvedor acesse, estude, modifique e distribue uma implementação compreensível desse modelo de IA, sem precisar pagar por isso.

Também vale destacar o uso de aplicações de código aberto para o desenvolvimento deste trabalho, como a própria *Godot Engine*, *draw.io*[3] para construção de diagramas e *Krita*[4] para edição de imagens. Além disso, foi utilizado um ícone de régua[5] criado pelo artista gráfico Alexandru Stoica e disponibilizado sob uma licença de domínio público. O uso desses recursos ilustra a importância de se atentar às licenças durante o desenvolvimento de um sistema de *software*.

Ademais, é preciso apontar que o desenvolvimento de um SBR envolveu a procura e estudo de literatura sobre o assunto, principalmente durante as etapas iniciais, nas quais houve o entendimento dos conceitos básicos e a busca por exemplos de uso. Esses fatores são os menos aparentes no resultado entregue — quando comparados com os outros objetivos e, principalmente, com os requisitos levantados —, porém, talvez eles sejam os mais importantes para o contexto desse projeto como um Trabalho de Conclusão de Curso.

Como dito na Introdução, a técnica de IA escolhida não é muito popular, especialmente em meio ao fortalecimento do aprendizado de máquina nos últimos anos. Em decorrência disso, não há muitos trabalhos ou fontes para serem consultados, especialmente com foco em desenvolvimento de jogos. Portanto, as referências levantadas nesta monografia podem servir de ponto de partida para aqueles que desejam estudar mais sobre o assunto e o sistema criado pode servir de exemplo de uso.

Por fim, serão discutidas algumas possibilidades de trabalhos futuros que podem expandir o que foi criado no arcabouço *Rule-Based Godot*. Será mostrada uma possível melhoria estrutural, que consiste na troca de uma estrutura de dados e na incorporação de um algoritmo conhecido na indústria de IA e já utilizado em outros sistemas do tipo. Além disso, citaremos alguns recursos extras que poderiam ser adicionados.

---

[2] Leia mais sobre *software* livre em: https://www.gnu.org/philosophy/free-sw.en.html

[3] Site oficial do *draw.io*: https://www.drawio.com/

[4] Site oficial do *Krita*: https://krita.org/

[5] Ícone disponível em: https://seekicon.com/free-icon/ruler-round_1

5.1 Trabalhos futuros

A primeira grande possível melhoria para o sistema seria a incorporação do algoritmo *Rete* (Forgy, 1982), um método padrão na indústria de IA para pareamento de regras em um banco de dados. Esse não é o algoritmo mais eficiente conhecido atualmente, porém suas variações ainda são a base para a implementação de muitos sistemas de grande porte, tanto SBR, quanto outros sistemas especialistas (Millington e Funge, 2009). Diante disso, faremos uma rápida análise do que define esse método e das mudanças necessárias para aplicá-lo no *plugin*.

Primeiramente, o *Rete* exige que todas as regras estejam representadas em uma única estrutura de dados: um grafo acíclico dirigido (*Directed Acyclic Graph*, ou *DAG*). Para isso, ele utiliza três tipos de nós: **(i)** predicados individuais, equivalentes aos nossos pareamentos atômicos; **(ii)** nós de junção, que implementam operadores *booleanos* e **(iii)** nós que representam as regras em si, nos quais estão declaradas as ações. Um exemplo dessa estrutura se encontra na Figura 5.1, na qual está definido o mesmo comportamento usado para comparar diferentes métodos de IA na Seção 1.2.5.

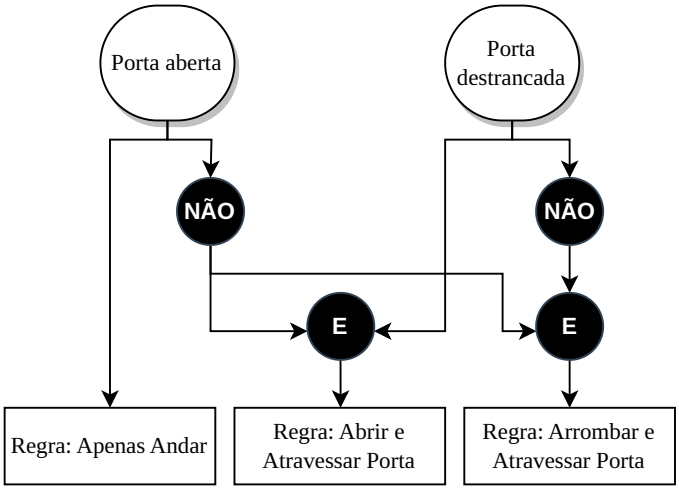

Figura 5.1: *Estrutura Rete para comportamento do cômodo com porta. Nós redondos e claros representam pareamentos atômicos, nós escuros são operadores booleanos e retângulos representam as regras. Repare que temos a mesma lógica descrita no Programa 1.3, mas com o reaproveitamento dos pareamentos e o ocultamento das ações, que agora estão implícitas nos nós de* `Rule`.

Observe na imagem que cada predicado individual só aparece uma vez (Porta aberta e Porta destrancada), mesmo com ambos sendo utilizados em mais de uma regra. Isso demonstra a principal vantagem desse algoritmo: o reaproveitamento de pareamentos atômicos; se a condição de duas regras utiliza a mesma verificação, não precisamos duplicar o objeto que a representa, podemos fazer as duas regras apontarem para o mesmo. Por comparação, na implementação atual de *Rule-Based Godot* é necessário instanciar um *resource* idêntico em cada regra.

Além da economia de memória, esse reaproveitamento gera um aumento na eficiência do sistema, pois todas as regras são verificadas ao mesmo tempo, através da estrutura unificada, e cada pareamento só é executado uma vez. No algoritmo *Rete*, uma iteração do

sistema consiste em uma varredura do grafo, começando pelos predicados em seu topo e descendo até as regras, os nós terminais. Dessa forma, conseguimos garantir que o tempo de iteração está limitado pela complexidade do *DAG*.

Ademais, vale destacar que o *Rete* permite a manipulação de variáveis, portanto, não haveria nenhuma perda de expressividade caso o sistema fosse remodelado usando esse algoritmo. A maneira com a qual ele lida com variáveis é muito parecida com a que implementamos: os vínculos entre o nome da variável e a lista de todas as substituições válidas são passados de nó em nó, sendo feitas as remoções necessárias conforme a especificação de cada um. Porém, em vez de passarmos uma referência ao dicionário de vínculos, precisamos passar cópias, pois não é mais possível manter esses dados no objeto da regra e garantir que os pareamentos de sua condição são exclusivos, ou seja, só precisam referenciar uma regra.

Com isso, para entender o processo de unificação nessa estrutura, observe as Figuras 5.2 a 5.6, nas quais repetimos o exemplo dado na Seção 3.2.1. Ao longo dessa execução, fica evidente uma desvantagem do algoritmo de *Rete*: pareamentos testam objetos que não precisariam, pois não têm acesso aos vínculos dos testes anterior e, portanto, não podem utilizá-los como lista de potenciais candidatos. Porém, também é possível perceber que essa implementação alternativa possui uma execução semelhante e gera os mesmos resultados.

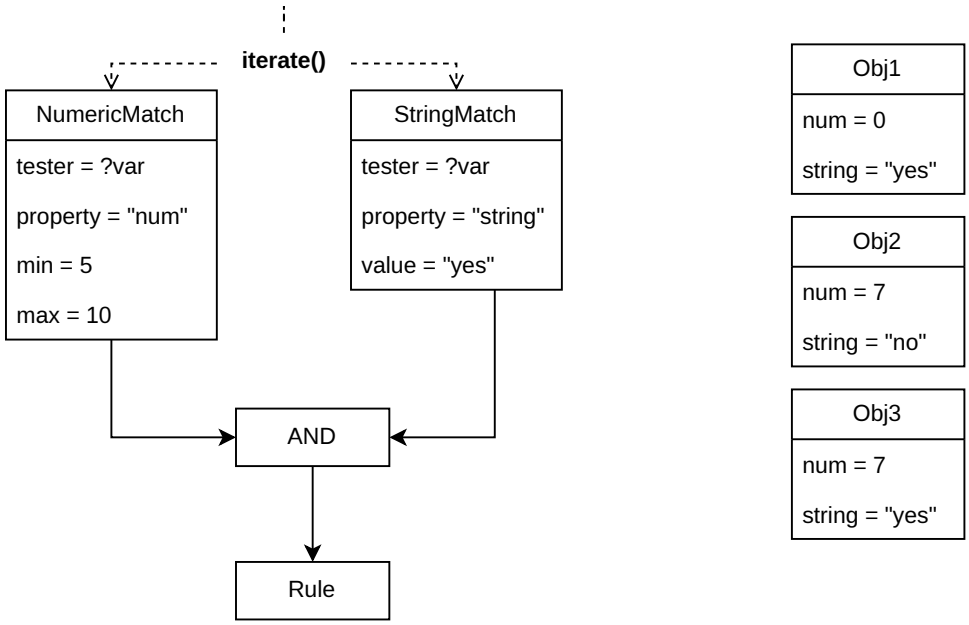

Figura 5.2: *Estado inicial do exemplo de unificação com Rete, equivalente à Figura 3.8. Num primeiro momento, a chamada da função* `iterate()` *alcança todos os nós no topo do DAG. Repare que a condição da regra está representada de cima para baixo, com os pareamentos atômicos apontando para o operador booleano que os une. O objeto* `Rule` *agora contêm todas as ações a serem executadas pela regra.*

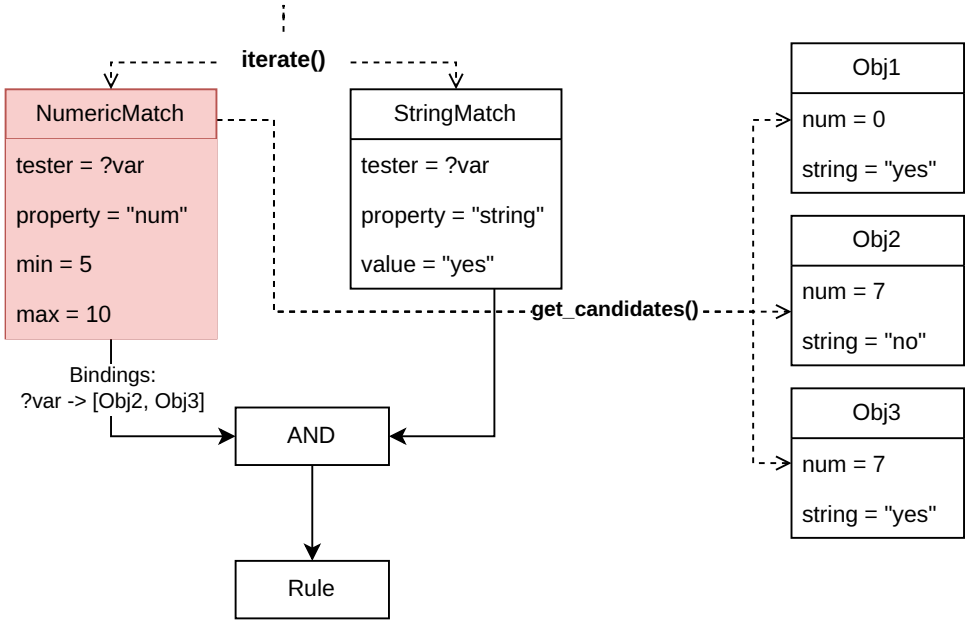

Figura 5.3: *Passo 1 do exemplo de unificação com Rete, equivalente à Figura 3.10. Temos a execução do `NumericMatch`: primeiro é feita uma busca de todos os candidatos válidos (`get_candidates()`); depois, verifica-se um por um, salvando os que satisfazem o pareamento em uma lista; por fim, salvamos a associação entre a variável que representa o nó wildcard (?var) e a lista dos objetos que são substituições válidas. Essa associação é transmitida por meio do dicionário Bindings.*

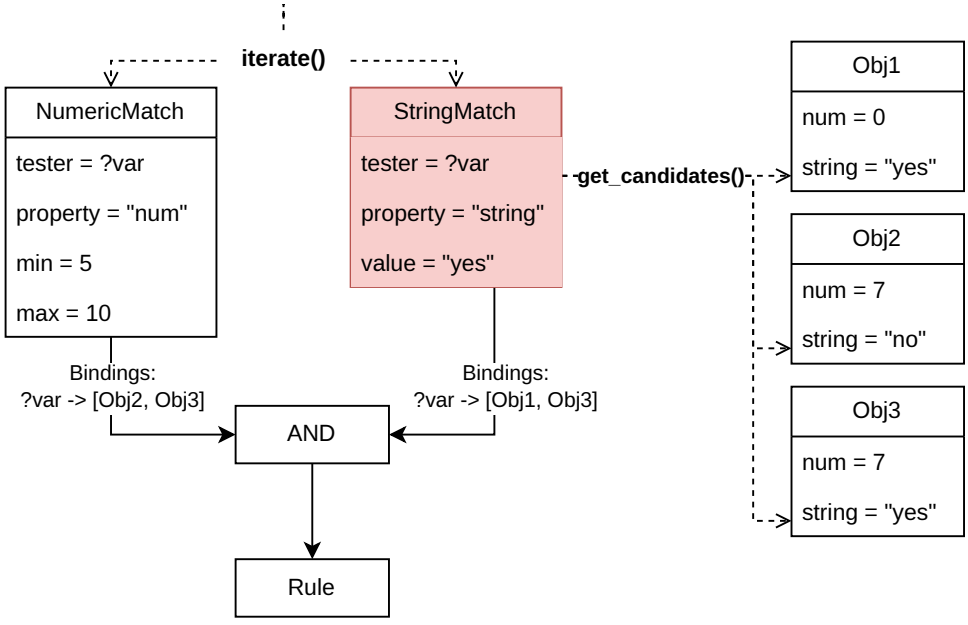

Figura 5.4: *Passo 2 do exemplo de unificação com Rete, equivalente à Figura 3.11. Constatamos a execução de `StringMatch`, repetindo o mesmo processo do passo anterior. Repare que, diferentemente da Figura 3.11, não restringimos os candidatos aos nós válidos para o pareamento anterior, pois não há acesso a essa lista. Com isso, notamos que o Obj1 foi testado e salvo em Bindings, algo que não ocorreu na primeira versão desse exemplo.*

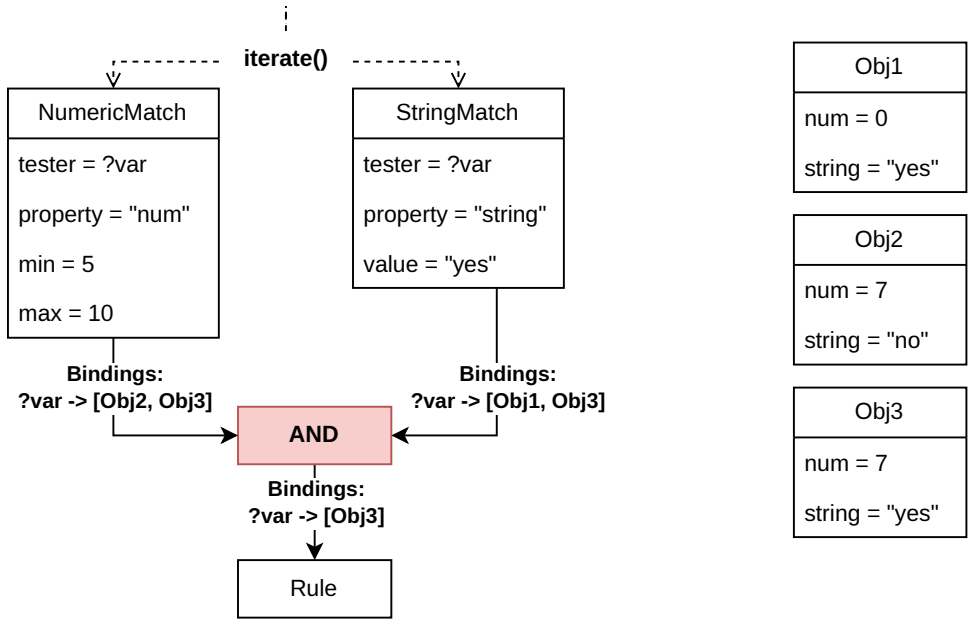

Figura 5.5: *Passo 3 do exemplo de unificação com Rete, equivalente à Figura 3.12. Vemos a execução do operador E, o qual recebe os vínculos dos dois pareamentos e realiza uma filtragem: somente elementos que aparecem em ambas as listas são mantidos como substituições válidas de ?var. Assim, somente o Obj3 aparece no dicionário transmitido pelo* `ANDMatch`.

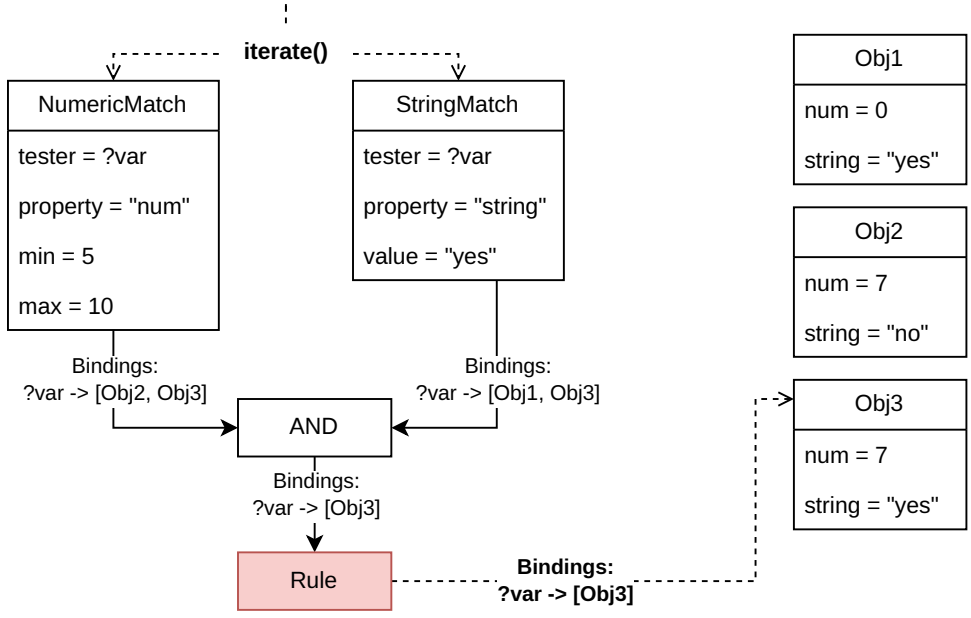

Figura 5.6: *Passo 4 do exemplo de unificação com Rete, que representa a execução das ações associadas à regra e, portanto, não estava incluso no exemplo de unificação dado nas Figuras 3.8 a 3.13. O objeto* `Rule` *aplica a substituição de ?var por Obj3 e depois executa as ações da regra, encerrando a chamada da função* `iterate()`*. Em casos reais, haveria a atuação de um árbitro para selecionar qual regra deve ser acionada, mas como nesse exemplo só há uma, isso foi omitido.*

Além do *Rete*, uma adição interessante ao sistema seriam regras para gerenciar subconjuntos de regras, permitindo que elas fossem agrupadas e esses grupos fossem ligados ou desligados conforme a necessidade. Isso permitiria a criação de sub-comportamentos, módulos que podem ser reutilizados e compostos para definir a inteligência de uma personagem, de maneira semelhante aos estados de uma Máquina de Estados.

Além do auxílio na modelagem de comportamentos, o agrupamento de regras é uma possível solução para o problema de desempenho encontrado em sistemas com muitas definições. Como é possível desligar um número arbitrário de regras, podemos garantir que somente as aplicáveis estarão ligadas, assim economizando o tempo e o processamento que seriam gastos em condições que sabemos serem insatisfeitas.

Passando para adições específicas ao *plugin* da *Godot* que poderiam ser feitas futuramente, listamos a criação de uma interface gráfica própria, ao invés de se aproveitar uma doca já existente. Isso exigiria a adaptação de várias funcionalidades nativas do Inspetor, principalmente dos *widgets* de atribuição de propriedades, sendo eles o principal impeditivo para a elaboração de uma *GUI* dedicada neste projeto.

Porém, entende-se que uma interface específica para a declaração de regras poderia trazer facilidades ao usuário que o Inspetor não consegue, como a edição da árvore (ou grafo, no caso do *Rete*) de condições por meio do manuseio de nós e suas conexões. Unindo essa ideia com a implementação do algoritmo mencionado anteriormente, poderia haver uma interface gráfica para arrastar e ligar os nós que formam o *Rete*, assim facilitando o entendimento dessa estrutura complexa.

Por fim, poderiam ser disponibilizados pareamentos e ações que lidam com elementos externos às cenas, como: manipulação de arquivos, permitindo ao usuário consultar e alterar dados diversos, e alteração de configurações do projeto da *Godot*. Esses recursos podem ser facilmente implementados pelo usuário, por meio dos *templates* fornecidos, porém, seria vantajoso apresentá-los como módulos prontos e já testados.

Apêndice A

Repositório do projeto

O repositório que contém todo o desenvolvimento do código do projeto pode ser acessado em: https://github.com/rvbatt/rule-based-godot

As versões do sistema mencionadas no texto estão etiquetadas e podem ser visualizadas nos seguintes *links*:

- Pré-*alpha*: https://github.com/rvbatt/rule-based-godot/releases/tag/pre-alpha1
- *Alpha*: https://github.com/rvbatt/rule-based-godot/releases/tag/alpha
- *Beta*: https://github.com/rvbatt/rule-based-godot/releases/tag/beta
- **v1.0.0**: https://github.com/rvbatt/rule-based-godot/releases/tag/v1.0.0

A documentação de uso do *plugin* se encontra no documento `README.md`, na raiz do repositório.

Apêndice B

Script Templates

Programa B.1 *Template* para novo tipo de `AbstractArbiter`

```
1   # meta-name: New Arbiter
2   # meta-description: New type of arbiter to select which rule to trigger
3   # meta-default: true
4
5   # There is no need to declare a new class_name,
6   # you can just create a .tres resource and Quick Load it
7   #class_name NewArbiter
8   extends AbstractArbiter
9
10  # MUST be implemented
11  func select_rule_to_trigger(satisfied_rules: Array[Rule]) -> Rule:
12      ## Returns one of the satisfied_rules
13      push_error("Abstract Method")
14      return null
```

Programa B.2 *Template* para novo tipo de `AbstractBooleanMatch`

```
1   # meta-name: New Boolean Match
2   # meta-description: New type of multi-input boolean match for rules
3   # meta-default: true
4
5   # MUST have a class_name. Recommended use of the suffix 'Match'
6   #class_name Match
7   extends AbstractBooleanMatch
8
9   # Inherited varible: subconditions (Array[AbstractMatch])
10
11  # MUST be implemented, using the subconditions
12  func is_satisfied(bindings: Dictionary) -> bool:
13      push_error("Abstract method")
14      return false
```

Programa B.3 *Template* para novo tipo de `AbstractAtomicMatch`

```
# meta-name: New Atomic Match
# meta-description: New type of atomic match for rules
# meta-default: true

@tool # Atomic matches MUST be @tool
# MUST have a class_name. Recommended use of the suffix 'Match'
#class_name Match
extends AbstractAtomicMatch

# Export variables and give them a default value
@export var num_var := 1.0 # Use float instead of int
@export_node_path() var path_var := ^""
var _node_var: Node # Keep an internal reference to nodes

func _init():
    # Should the match check some Node?
    Tester_Node = true

    # Requires Tester_Node
    # Is the check based on some data from the node?
    Data_Based_Node = true

    # Requires Data_Based_Node
    # Is the data extracted from a property or method call?
    Get_Node_Data_Preset = false

    # When using nodes that may change position in the SceneTree,
    # save the reference to the original node
    _preset_node_path("path_var", "_node_var")

    # Connect signal when match is created
    #_pre_connect("_node_var", "signal", receiving_func)

# Should ONLY be implemented IF Tester_Node = false
#func is_satisfied(bindings: Dictionary) -> bool:
# return false

# Can be optionally implemented if Tester_Node = true
#func _get_candidates() -> Array[Node]:
# ## Returns all the nodes that may satisfy the match
# var candidates: Array[Node] = []
# return candidates

# NEEDS to be implemeted IF Tester_Node = true and Data_Based_Node = false
#func _node_satisfies_match(tester_node: Node, bindings: Dictionary) -> bool:
# return false

# NEEDS to be implemented IF Data_Based_Node = true
func _data_satisfies_match(data: Variant) -> bool:
    push_error("Abstract Method")
    return false

```

cont ⟶

⟶ *cont*

```
53  # NEEDS to be implemented IF Data_Based_Node = true and Get_Node_Data_Preset
        = false
54  func _get_data(tester_node: Node) -> Variant:
55      push_error("Abstract Method")
56      return null
```

Programa B.4 *Template* para novo tipo de `AbstractAction`

```
1   # meta-name: New Action
2   # meta-description: New type of action for rules
3   # meta-default: true
4
5   @tool # Actions MUST be @tool
6   # MUST have a class_name. Recommended use of the suffix 'Action'
7   #class_name Action
8   extends AbstractAction
9
10  # Export variables and give them a default value
11  @export var num_var := 1.0 # Use float instead of int
12  var _signal_var := "signal_name"
13  # This is a dictionary with the name of the signal parameters as keys and
14  # an arbitrary element with the same type as the corresponding argument as
        value
15  var _parameter_to_type: Dictionary# = {number_param: 1, string_param: ""}
16
17  func _init():
18      # Does the action trigger some Node(s)?
19      Agent_Nodes = true
20
21      # Adds a signal if Agent_Nodes = true and (agent_)type = Path
22      _pre_add_signal("_signal_var", "_parameter_to_type")
23
24  # Should ONLY be implemented IF Agent_Nodes = false
25  #func trigger(bindings: Dictionary) -> Array:
26  # ## Returns some result of triggering rhis action
27  # return []
28
29  # Can be optionally implemented if Agent_Nodes = true
30  #func _get_agent_nodes(bindings: Dictionary) -> Array:
31  # ## Returns all the nodes that will perform the action
32  # ## consider checking agent_type
33  # return []
34
35  # NEEDS to be implemeted IF Agent_Nodes = true
36  func _trigger_node(agent: Node, bindings: Dictionary) -> Variant:
37      ## Returns some result of triggering this node
38      ## Should check bindings for data variables, e.g.:
39      # if data_var is String and data_var.begins_with('?'):
40      #  data_var = bindings.get(data_var.trim_prefix('?'))
41      push_error("Abstract Method")
42      return null
```

Referências

[Angilica *et al.* 2022] Denise Angilica, Giovambattista Ianni e Francesco Pacenza. "Declarative ai design in unity using answer set programming". In: *IEEE Conference on Games (CoG)*. 2022 (citado nas pgs. 5–7, 17).

[Boole 1854] George Boole. *An investigation of the laws of thought. on which are founded the mathematical theories of logic and probabilities*. Walton e Maberly, 1854 (citado nas pgs. vii, 27, 33, 40).

[Bourg e Seemann 2004] David M. Bourg e Glenn Seemann. *AI for Game Developers*. O'Reilly Media, Inc., jul. de 2004 (citado nas pgs. 2, 12, 17, 36).

[Forgy 1982] Charles L. Forgy. "Rete. A fast algorithm for the many pattern/many object pattern match problem". *Artificial Intelligence* 19.1 (1982), pp. 17–37 (citado na pg. 96).

[Gamma *et al.* 1994] Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994 (citado nas pgs. vii, 30–32, 36, 44, 51, 52, 78, 92).

[IEEE 2008] IEEE. "Ieee-754, standard for floating-point arithmetic". *IEEE Std 754-2008* (jan. de 2008), pp. 1–58 (citado na pg. 67).

[Lifschitz 2019] Vladimir Lifschitz. *Answer Set Programming*. Springer, 2019 (citado na pg. 5).

[Linietsky *et al.* 2014] Juan Linietsky, Ariel Manzur e a comunidade Godot. *Godot Engine 4.1 documentation in English*. 2014. URL: https://docs.godotengine.org/en/4.1/ (acesso em 03/11/2023) (citado nas pgs. 19, 28, 29, 80).

[Lu e Li 2022] Yunlong Lu e Wenxin Li. "Techniques and paradigms in modern game ai systems". *Algorithms* 15 (ago. de 2022) (citado na pg. 10).

[Martin 2017] Robert C. Martin. *Clean Architecture. A Craftsman's Guide to Software Structure and Design*. 1ª ed. Pearson, set. de 2017 (citado na pg. 30).

[Millington e Funge 2009] Ian Millington e John Funge. *Artificial Intelligence for Games*. 2ª ed. Morgan Kaufmann Publishers, jan. de 2009 (citado nas pgs. 2, 9–11, 14, 18, 19, 32, 36, 55, 96).