UNIVERSIDADE FEDERAL DO CEARÁ

CENTRO DE TECNOLOGIA

DEPARTAMENTO DE ENGENHARIA ELÉTRICA

CURSO DE GRADUAÇÃO EM ENGENHARIA ELÉTRICA

YANA SOARES DE PAULA

ESTUDO E DESENVOLVIMENTO DE UM AMBIENTE DE SIMULAÇÃO PARA APLICAÇÕES DE REDES ELÉTRICAS INTELIGENTES UTILIZANDO O SOFTWARE PARA CO-SIMULAÇÃO MOSAIK 3.0

FORTALEZA

2022

YANA SOARES DE PAULA

ESTUDO E DESENVOLVIMENTO DE UM AMBIENTE DE SIMULAÇÃO PARA APLICAÇÕES DE REDES ELÉTRICAS INTELIGENTES UTILIZANDO O SOFTWARE PARA CO-SIMULAÇÃO MOSAIK 3.0

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia Elétrica do Centro de Tecnologia da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia Elétrica.

Orientador: Prof. Dr. Lucas Silveira Melo

FORTALEZA

2022

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

P349e Paula, Yana Soares de.
Estudo e desenvolvimento de um ambiente de simulação para aplicações de redes elétricas inteligentes utilizando o software para co-simulação mosaik 3.0 / Yana Soares de Paula. – 2022.
102 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Centro de Tecnologia, Curso de Engenharia Elétrica, Fortaleza, 2022.
Orientação: Prof. Dr. Lucas Silveira Melo.

1. Co-simulação. 2. Sistemas Multiagentes. 3. Redes Elétricas Inteligentes. 4. Mosaik. I. Título.
CDD 621.3

YANA SOARES DE PAULA

ESTUDO E DESENVOLVIMENTO DE UM AMBIENTE DE SIMULAÇÃO PARA APLICAÇÕES DE REDES ELÉTRICAS INTELIGENTES UTILIZANDO O SOFTWARE PARA CO-SIMULAÇÃO MOSAIK 3.0

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia Elétrica do Centro de Tecnologia da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia Elétrica.

Aprovada em:

BANCA EXAMINADORA

Prof. Dr. Lucas Silveira Melo (Orientador)
Universidade Federal do Ceará (UFC)

Prof. PhD. Ruth Pastôra Saraiva Leão
Universidade Federal do Ceará (UFC)

Eng. Nathanael Duque Gadelha
Universidade Federal do Ceará (UFC)

À minha família, pelo amor e apoio incondicionais que sempre me dedicaram - obrigada por possibilitarem que eu pudesse conquistar os meus sonhos. E a todos os educadores que compartilharam seus conhecimentos para meu desenvolvimento acadêmico, profissional e moral.

AGRADECIMENTOS

À minha família, por todo o apoio e amor imensuráveis. Agradecimentos especiais à minha mãe, por sua dedicação e carinho incondicionais.

A todos os professores que compartilharam seus conhecimentos e experiências para que eu pudesse me tornar a profissional que me tornei. Especialmente, ao Prof. Dr. Lucas Silveira Melo por toda a paciência e empatia prestadas nos últimos anos enquanto meu orientador acadêmico e professor.

Aos amigos que tornaram memorável a minha experiência universitária: Bruno Faustino, Davi Gomes, Felipe Gaspar, Giovanni Confalonieri, Isabela Horacio, Joelia Cavalcante, Luiz Alberto Freire, Messias Gutemberg, Nathalia Costa, Pedro Andreazza, Pedro Igor, Suzane Carvalho, entre tantos outros.

Agradeço, em particular, à minha amiga Annalyanne Pereira: por todas as horas a fio investidas estudando lado a lado na biblioteca da Física, pelas palavras de suporte quando mais precisei e por estar comigo desde o início até o fim dessa jornada. Eu não teria conseguido sem seu apoio.

Aos funcionários administrativos e prestadores de serviços da UFC que ampararam minha formação acadêmica de maneira excepcional. Cito, nominalmente, Adely Ribeiro, pelo auxílio prestado desde o início de minha vida universitária. E agradecimentos especiais aos funcionários da Cantina da Química que tornaram, diariamente, minhas visitas à UFC mais agradáveis.

Ao Doutorando em Engenharia Elétrica, Ednardo Moreira Rodrigues, e seu assistente, Alan Batista de Oliveira, aluno de graduação em Engenharia Elétrica, pela adequação do *template* utilizado neste trabalho para que o mesmo ficasse de acordo com as normas da biblioteca da Universidade Federal do Ceará (UFC).

Ao Programa de Educação Tutorial (PET) por ter sido o ambiente no qual pude progredir imensamente enquanto indivíduo. As experiências proporcionadas pelo programa e as pessoas com quem convivi por tantos anos foram os maiores responsáveis por meu amadurecimento pessoal e profissional. Agradecimentos especiais ao Prof. Dr. René Bascopé por sua dedicação para manter o PET Engenharia Elétrica como ambiente de excelência há tantos anos.

Finalmente, à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) por ter possibilitado a expansão de minha progressão acadêmica por meio do programa BRAFITEC.

“Em todas as lágrimas há uma esperança.“

(Simone de Beauvoir)

RESUMO

Nas últimas décadas, o conceito de Redes Elétricas Inteligentes (REI) ou *Smart Grids* tem proposto perspectivas que estão revolucionando a forma como a sociedade administra, representa e analisa Sistemas Elétricos de Potência (SEP). As REI propõem o aprimoramento de SEP por meio da aplicação direta de conceitos de Tecnologia da Informação (TI) em sua estrutura. Dentre as soluções de TI que possibilitam o aperfeiçoamento da representação, simulação e análise de REIs estão os sistemas de co-simulação. Estes sistemas são capazes de integrar diferentes simuladores ao sincronizá-los em um mesmo tempo de execução, possibilitando a conexão e a comunicação entre esses elementos de uma maneira simples e modular. Tais simuladores podem ser modelos matemáticos, *softwares* que representam dispositivos reais em uma rede elétrica ou Sistemas Multiagentes (SMA) que administram uma REI, por exemplo. O sistema de co-simulação Mosaik é uma solução *open-source* que foi desenvolvida focada, especificamente, para a análise de REIs e que fornece várias ferramentas interessantes para sua representação. Em 2021, uma nova versão do Mosaik foi lançada, contendo diversos aprimoramentos que possibilitam melhora de performance e ferramentas dirigidas ao tratamento de eventos discretos em co-simulação. Para simulações já desenvolvidas com a versão anterior do Mosaik, a atualização propõe uma simples adaptação na sua API sem grandes alterações na estrutura do sistema. Diante do exposto, o objetivo deste trabalho foca em verificar o impacto da adaptação de sistemas já desenvolvidos em Mosaik para a atualização, além de descrever e explorar as novas capacidades propostas pela nova versão do Mosaik 3. O estudo de caso desenvolvido neste projeto baseia-se na integração do SMA PADE com o sistema de co-simulação Mosaik, desenvolvida por (MELO *et al.*, 2020) no laboratório Grupo de Redes Elétricas (GREI) da Universidade Federal do Ceará (UFC). Na metodologia aplicada deste trabalho, adaptou-se o código-fonte da integração PADE/Mosaik para a nova versão do Mosaik 3, buscando confirmar a facilidade da adaptação à nova versão proposta pelo sistema. A partir deste projeto, comprovou-se a simplicidade em adaptar o sistema de versões anteriores do Mosaik à nova versão, tal qual proposto pela atualização.

Palavras-chave: Co-simulação. Sistemas Multiagentes. Redes Elétricas Inteligentes. Mosaik.

ABSTRACT

In the last decades, Smart Grids has proposed perspectives that are revolutionizing the way society manages, represents and analyzes Electric Power Systems (EPS). Smart Grids propose the improvement of EPS through application of Information Technology (IT) concepts in its structure. Co-simulation systems are among the IT solutions that make it possible to improve the representation, simulation and analysis of Smart Grids. These systems are able to integrate different simulators by synchronizing them in the same runtime, enabling the connection and communication between these elements in a simple and modular way. Such simulators can be mathematical models, softwares that represent real devices in an electrical network or Multiagent Systems (MAS) that manage an electrical grid, for example. Mosaik is an open-source co-simulation system solution that was developed focused specifically on Smart Grids analysis. It provides several interesting tools for its grid representation. In 2021, a new version of Mosaik was released, containing several enhancements and tools that allows improved performance and solutions aimed at handling discrete events. For simulations already developed with the previous version of Mosaik, the update proposes a simple adaptation in its API without major changes in the system's structure. Given the above, this work's objective focuses on verifying the impact of adapting systems already developed in Mosaik for the new update. Furthermore, it aims to describe and explore the new capabilities proposed by the new version of Mosaik 3. The case study developed in this project is based on a Mosaik integration with the MAS PADE, developed by (MELO *et al.*, 2020) at Grupo de Redes Elétricas (GREI) laboratory at Federal University of Ceará (UFC). In the methodology applied in this work, the PADE/Mosaik integration source code was adapted to the new version of Mosaik 3, seeking to confirm the Mosaik's ease of adaptation proposed by the update. In this project, it was proved the simplicity of adapting the system of Mosaik's previous versions to the new update, as proposed by the Mosaik 3.

Keywords: Co-simulation systems. Multi Agent Systems. Smart Grids. Mosaik.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS-FONTE

LISTA DE ABREVIATURAS E SIGLAS

AC	Atos Comunicativos
ACL	*Agent Communication Language*
FIPA	*Foundation for Intelligent Physical Agents*
GREI	Grupo de Redes Elétricas Inteligentes
IA	Inteligência Artificial
IED	*Intelligent Electronic Devices*
KQML	*Knowledge Query and Manipulation Language*
PADE	*Python Agent DEvelopment*
PI	Protocolos de Interação
REI	Redes Elétricas Inteligentes
SEP	Sistemas Elétricos de Potência
SMA	Sistemas Multiagentes
TIC	Tecnologia da Informação e Comunicação
UFC	Universidade Federal do Ceará
UML	*Unified Modeling Language*

SUMÁRIO

1 INTRODUÇÃO

Nas últimas décadas, novos conceitos e inovações tecnológicas transformaram a forma como a energia elétrica é produzida, distribuída e mantida na sociedade. Dentre as inovações que estão moldando a área da Engenharia Elétrica, o conceito de REI oferece soluções e perspectivas que estão revolucionando a forma como a sociedade administra a energia elétrica. O conceito de REI propõe o aprimoramento da rede elétrica atual, aplicando conceitos de Tecnologia da Informação e Comunicação (TIC) em sua estrutura, o que permite e incentiva a descentralização da produção de energia elétrica, assim como legitima novas estratégias de fiscalização, manutenção e atendimento de eventos que possam interferir ou influenciar a rede elétrica (PALENSKY; KUPZOG, 2013).

Esta noção permite melhorias diversas na gestão de energia elétrica em tempo real, o que possibilita respostas rápidas para eventos que possam afetar uma rede elétrica e seus consumidores. Nesse contexto, conceitos de Inteligência Artificial (IA) têm sido amplamente pesquisados e aplicados na área a fim de possibilitar análises e recuperações de interferências na rede elétrica com maior eficiência - que, de outra forma, seriam dispendiosas e demoradas para solucionar (SAMPAIO, 2017).

Considerando-se o contexto apresentado, a integração de soluções de TIC em sistemas e ambientes reais, gera uma demanda considerável de criação de simulações cada vez mais próximas da realidade por elas representadas. De fato, o desenvolvimento de sistemas computacionais que espelham autenticamente o ambiente analisado possibilitam estudo de sistemas reais de maneira mais eficiente - o que possibilita gestão de sistemas em tempo real e análises de soluções mais eficazes para problemas.

Diante do apresentado, inúmeros simuladores de sistemas eletrônicos ou reais são desenvolvidos para variadas soluções em diversas áreas de aplicação. Normalmente, tais simuladores - que nada mais são que programas ou modelos de situações reais - podem ser definidos como representações de sistemas digitais ou reais (BANDYOPADHYAY; BHATTACHARYA, 2014). A depender do tipo de representação de um simulador, ele pode ter sua execução definida em “tempo contínuo” ou “tempo discreto”. Por vezes, uma sincronização inadequada de execução de simuladores pode ocasionar erros nas trocas de comunicações entre eles. Portanto, para evitar esse problema, a criação, implementação e manutenção de simuladores deve levar em consideração o tempo de execução de seus elementos, a fim de ocasionar a sincronização de ações realizadas pelos diferentes simuladores ou modelos.

Sistemas de co-simulação são, justamente, os sistemas responsáveis por sincronizar a execução e a comunicação de diferentes simuladores de uma maneira holística (STEINBRINK *et al.*, 2017). Essa capacidade de harmonização de tempos e de comunicação entre simuladores permite uma integração adequada que acarreta análises convenientes de simulações complexas em uma maneira modular e distribuída. Assim, a co-simulação permite uma integração entre diferentes simuladores já existentes, equivalendo a um elemento de junção entre simuladores que otimiza a simulação de sistemas ciber-físicos e propicia uma visão geral da interação de simuladores diferentes.

No contexto apresentado, o sistema de co-simulação Mosaik (OFENLOCH *et al.*, 2022) é uma interessante solução *open-source* baseada em Python que foi desenvolvida levando-se em consideração o domínio de REI. O Mosaik provê, portanto, diversas ferramentas para visualização e análise de sistemas no contexto de *Smart Grids* - o que torna-o interessante ferramenta de estudos na área da Engenharia Elétrica.

Dentre os simuladores que podem ser associados a um sistema de co-simulação, destaca-se a utilização de Sistemas Multiagentes (SMA). Um SMA pode ser definido como um sistema no qual entidades autônomas (agentes) interagem entre si e com seu ambiente para atingir objetivos específicos (DEMAZEAU, 1995). A dinamicidade proposta por esses tipos de sistema permite sua aplicação em inúmeras áreas de tecnologia e são especialmente interessantes para aplicações em sistemas distribuídos. De fato, considerando-se as capacidades de implementação de SMA em estruturas modulares, autônomas e dinâmicas, identifica-se uma ampla gama de aplicações de tais sistemas em quaisquer circunstâncias que requiram solução de problemas de alta complexidade e que integram sistemas reais com áreas de TIC. Diante disso, fica claro, como descrito, inicialmente, que os SMAs tornam-se soluções proeminentes na área de REIs, já que esse domínio possui como uma de suas principais consequências a descentralização de Sistemas Elétricos de Potência (SEP) - conjuntura ideal para a aplicação de SMAs.

No entanto, para funcionar adequadamente, é necessário que o SMA esteja de acordo com regras bem-definidas de normas e protocolos capazes de realizar a interação adequada entre seus elementos. Partindo desse ponto, surge a importância da análise de conceitos desenvolvidos e mantidos pela *Foundation for Intelligent Physical Agents* (FIPA) - fundação de grande destaque na determinação de normas para a comunicação de agentes inteligentes.

É nesse contexto que surge uma solução de SMA desenvolvida pelo Grupo de Redes Elétricas Inteligentes (GREI) na Universidade Federal do Ceará (UFC): o *Python Agent*

DEvelopment (PADE) (GREI-UFC, 2019). O PADE é um *framework* desenvolvido em Python de *software* livre que utiliza como base os protocolos de comunicação de agentes das normas FIPA e que pode ser aplicado em diversas áreas de simulação e automação.

Assim, considerando-se os conteúdos apresentados, vale evidenciar o potencial em utilizar os *softwares* Mosaik e PADE em conjunto. De fato, para as demandas e contextos de Engenharia Elétrica, a utilização de um SMA em integração com um sistema de co-simulação focado em REIs presta imensa potencialidade de aplicação em várias soluções identificadas na área.

1.1 Visão Geral do Problema

A utilização de sistemas de co-simulação capacita sua aplicação em diversos elementos de Engenharia Elétrica - mais precisamente, na área de REI. A integração de SMAs para a gestão dos simuladores em processos de co-simulação também é essencial para análises, gestão e representações verídicas de sistemas reais.

Dentre os tipos de simuladores que podem ser associados em sistemas de co-simulação, descreve-se os que possuem sua execução em "tempo contínuo" e aqueles que possuem execuções em "tempo discreto". Normalmente, simuladores ditos de "tempo contínuo" estão associados a leituras de variáveis reais e que associam-se ao tempo contínuo - por exemplo: *Intelligent Electronic Devices* (IED) que realizam a leitura de tensões e correntes na rede elétrica. Já simuladores de "tempo discreto" normalmente não possuem a necessidade de serem executados em um tempo contínuo, sendo acionados apenas em momentos necessários para troca de dados ou eventos - por exemplo: um agente inteligente é acionado quando um simulador de leitura de tensões ou correntes lê um valor específico que exija alguma ação para controle do valor identificado.

Partindo-se das diferenças de execução identificadas entre esses dois tipos de execução, torna-se interessante a implementação de sistemas de co-simulação que otimizem ao máximo a performance dos sub-simuladores a depender de sua classificação. A identificação clara da classificação de um simulador e a sua aplicação correta em um sistema de co-simulação que realize o tratamento das execuções de seus sub-simuladores a partir de seus tipos pode gerar otimizações de performance e manutenção do sistema simulado.

A versão 3.0 do sistema de co-simulação Mosaik propõe, justamente, novas alternativas para o tratamento de simuladores a partir de sua classificação. Na atualização, novas

ferramentas e otimizações de performance baseadas na classificação de simuladores a partir de seu tipo de execução foram implementadas (OFENLOCH *et al.*, 2022). Além disso, a atualização do Mosaik também propõe uma fácil adaptação de sistemas que já executem com a versão anterior do sistema para o Mosaik 3.0 (SCHERFKE, 2018).

Diante disso, é válido destacar que, atualmente, ainda não há material descritivo e detalhado sobre as novas capacidades do Mosaik 3.0 em língua portuguesa. Ademais, considera-se interessante a verificação da simplicidade de adaptação de versões anteriores proposta pela atualização.

A partir do exposto, cabe evidenciar que o SMA PADE, desenvolvido pelo GREI da Universidade Federal do Ceará ainda não possui adaptação aos novos recursos dispostos pela nova versão do Mosaik 3.0. Ademais, a estrutura de integração PADE/Mosaik disposta precisou ser desenvolvida com certa complexidade a fim de tratar situações de comunicação cíclica no sistema - que poderiam ser otimizados com as novas funcionalidades do Mosaik 3.0.

1.2 Justificativa

Considerando-se a pertinência do uso de sistemas de co-simulação na área de Engenharia Elétrica para criação de simulações eficazes e representações categóricas de situações reais, fica claro o interesse de aplicações do sistema de co-simulação Mosaik em desenvolvimentos diversos para a área.

Partindo-se dessa perspectiva, uma compreensão completa e detalhada das modificações propostas pela nova versão do Mosaik 3.0 pode causar um impacto positivo em implementações de simulações voltadas à área da Engenharia Elétrica. Efetivamente, as inovações propostas pela nova versão do Mosaik 3.0 podem ser implementadas de forma a aperfeiçoar desenvolvimentos de simulações já existentes com uma otimização de performance e de manutenção de simuladores e suas interações com outros sistemas.

Mais precisamente, partindo-se do SMA PADE desenvolvido para análise de REIs, torna-se pertinente a sua adaptação para que ele possa beneficiar-se das alterações propostas pela versão Mosaik 3.0. Como já existe uma integração entre os sistemas citados, o ajuste do PADE para a nova versão do sistema Mosaik capacita um sistema já consagrado para aplicabilidades mais atualizadas que podem potencializar seu desempenho.

Portanto, fica evidente a relevância de realizar um estudo preciso dos conteúdos associados aos ajustes propostos pelo Mosaik 3.0 - justificando-se a realização desse trabalho

enquanto manual para exploração dos assuntos abordados e base para estudos futuros pelo GREI. Ademais, a aplicação e análise dos conceitos propostos no *framework* PADE também demonstram notável pertinência para averiguação das vantagens e desempenho das novas características sugeridas pelo Mosaik 3.0.

1.3 Objetivos

Considerando-se a relevância dos conceitos de co-simulação para aplicações diversas no campo da Engenharia Elétrica, fica evidente a necessidade de explorá-los para fins de pesquisa e de implementações reais. Diante disso, este documento pretende examinar os conceitos associados ao assunto, assim como explorar desenvolvimentos e possíveis cenários de uso para implementação e análise dos conceitos aqui descritos - com foco em soluções para tratamento de simuladores que lidem com eventos discretos e outros conceitos identificados no contexto de co-simulação.

Para tal, os objetivos gerais e específicos deste projeto são descritos nas seções a seguir.

1.3.1 Objetivos Gerais

Avaliar, atualizar e desenvolver um ambiente de co-simulação usando a ferramenta Mosaik 3.0 e o simulador PADE.

1.3.2 Objetivos Específicos

- Compreender os conceitos associados à atualização do sistema de co-simulação Mosaik 3.0;
- Comparar as diferenças entre o sistema Mosaik 2 e Mosaik 3.0;
- Realizar cenários de uso que integrem as capacidades do sistema multi-agentes PADE com as atualizações propostas pelo sistema de co-simulação Mosaik 3.0;
- Implementar atualizações do Mosaik 3.0 no *framework* PADE;

1.4 Metodologia

Diante dos conceitos apresentados neste documento, este projeto desenvolverá, a partir dos conteúdos identificados, um ambiente de testes que represente as inovações sugeridas

pelo sistema de co-simulação Mosaik 3.0 em integração com o *framework* de agentes inteligentes PADE.

Para a metodologia deste projeto, foi percorrida a seguinte estrutura:

- Estudo dos conceitos associados a co-simulação e a sistemas multi-agentes;
- Compreensão das atualizações propostas pelo sistema de co-simulação Mosaik 3.0;
- Análise das diferenças identificadas pelos exemplos disponibilizados pelas versões 2 e 3 do sistema de co-simulação Mosaik;
- Adaptação do *framework* PADE à nova versão do Mosaik 3.0;
- Análise dos resultados obtidos;

1.5 Estrutura do documento

Este documento está arquitetado em cinco capítulos, tal qual descrito a seguir.

O Capítulo 1 trata da contextualização e definição geral dos assuntos associados ao trabalho, além dos objetivos, compreensão do problema e respectiva justificativa, assim como a metodologia aplicada no desenvolvimento do projeto.

O Capítulo 2 trata da fundamentação teórica acerca do trabalho e, portanto, expõe conteúdos detalhados associados a sistemas de co-simulação e sistemas multiagentes. Neste capítulo, são esmiuçados todos os conteúdos teóricos associados ao trabalho, assim como a descrição do sistema de co-simulação Mosaik e do *framework* PADE. Também descrevem-se as diferenças primordiais entre as diferentes versões do sistema Mosaik e as justificativas e contextualizações das atualizações dispostas na versão Mosaik 3.0.

O Capítulo 3 identifica o estudo de caso desenvolvido para descrever os assuntos teóricos apresentados no capítulo precedente. Neste capítulo são apresentados as análises e o desenvolvimento realizados para implementação dos assuntos apresentados no trabalho.

Finalmente, o Capítulo 4 discorre sobre os resultados obtidos do desenvolvimento, dissertando sobre seus impactos e êxitos no que trata dos objetivos apresentados inicialmente. Este capítulo também conclui o trabalho, resumindo a resolução do documento e identificando possíveis melhorias e trabalhos futuros associados à continuação do desenvolvimento deste trabalho.

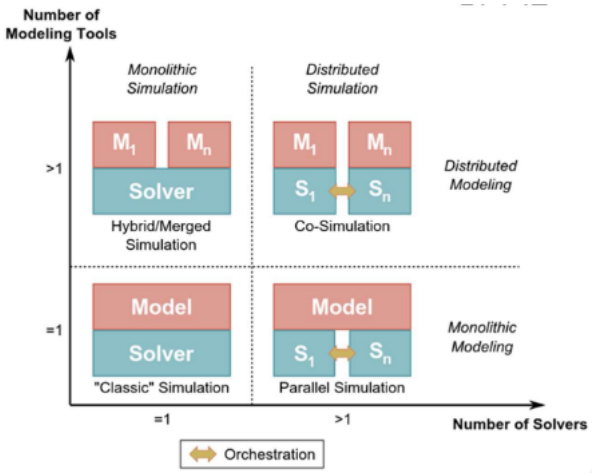
2 CO-SIMULAÇÃO E SISTEMAS MULTIAGENTES

2.1 Sistemas de Co-simulação

Considerando a complexidade atual dos desafios de engenharia em vários domínios, interoperabilidade, modularidade, flexibilidade, dinamicidade, confiabilidade e escalabilidade são aspectos fundamentais para modelar problemas do mundo real com eficiência. A tecnologia possibilitou o desenvolvimento de soluções complexas e detalhadas que modelam eventos do mundo real e preveem comportamentos por meio de simulações de computador. Baseadas em modelos matemáticos de ocorrências físicas, essas simulações ajudam a preparar, planejar e testar circunstâncias que têm perspectivas importantes no desenvolvimento e análise tecnológica (JALOTE, 2008).

Infelizmente, muitos softwares de simulação possuem propriedade intelectual, o que significa que cada um possui protocolos e sistemas privados individuais que, normalmente, não permitem a interoperabilidade com outras aplicações. A fim de resolver este problema e alcançar uma maior qualidade na interação entre diferentes sistemas de simulação, a co-simulação tem sido amplamente aplicada. Um algoritmo de co-simulação coordena a sincronização e a interação de diferentes sub-simuladores independentes para fornecer uma simulação global de todos os componentes. A co-simulação é a chave para integrar sistemas diferentes e incompatíveis em um modelo holístico que otimiza a consolidação de diversas interações, resultados e análises (STEINBRINK *et al.*, 2017).

Figura 1 – Comparativo de tipos de integrações de sistemas. A depender da configuração, pode-se ter vários modelos "M" (M_1 a M_n) ou vários solucionadores "S" (S_1 a S_n).



Fonte: (OFENLOCH *et al.*, 2022).

Pensando nisso, a co-simulação fornece um sistema flexível, modular e integrado capaz de combinar diferentes domínios de engenharia na análise de eventos físicos e digitais. A Figura 1 apresenta um comparativo entre a solução de co-simulação e outros modelos, destacando a sua arquitetura modular e distribuída. Usando um modelo de "caixa preta" mestre-escravo de seus sub-simuladores, os algoritmos de co-simulação protegem sistemas com propriedade intelectual (STEINBRINK *et al.*, 2017). Isso significa que o sistema de co-simulação não identifica como uma sub-simulação funciona - em vez disso, ele apenas adapta e usa seus resultados para permitir a interatividade com os resultados de outros sub-simuladores, sincronizando suas execuções. Assim, os resultados de cada sistema independente podem ser usados na simulação global para aprimorar a compreensão dos impactos entre cada submodelo autônomo.

A Figura 2 mostra uma representação da estrutura de co-simulação aplicada a uma REI, onde é possível observar simuladores que representam o consumo de uma casa (azul), um sistema de placas fotovoltaicas (verde), um sistema de análise de fluxo (cinza) e uma ferramenta de análise e monitoramento (amarelo).

Figura 2 – Estrutura modular de sistema de co-simulação aplicada ao contexto de REI.

Fonte: (SCHERFKE, 2018)

2.1.1 Mosaik

Mosaik é um sistema de co-simulação *open-source* baseado em Python que se concentra no domínio da Engenharia Elétrica - mais precisamente, em *Smart Grids*. Ele funciona como integrador (`mosaik-core`) de uma estrutura modular que incorpora diferentes entidades (`mosaik-components`). O Mosaik também fornece um conjunto de ferramentas que facilitam a visualização e análise de simulações (`mosaik-tools`), assim como diferentes APIs que permitem a sincronização, a comunicação e a interação entre suas entidades (SCHERFKE, 2018).

É interessante compreender alguns termos essenciais na sintaxe do Mosaik - e que

são utilizadas ao longo desse documento. [1] Alguns destes termos principais são definidos a seguir:

- **Cenário**: descrição do sistema simulado. Inclui todos os elementos e relações que consistem um determinado sistema.
- **Modelo:** é uma representação simplificada de um objeto ou sistema no mundo real. Possibilita a análise de aspectos relevantes do sistema ou objeto representado.
- **Entidades:** identifica a instância de um modelo na simulação Mosaik. Entidades podem ser conectadas para permitir fluxo de dados entre elas.
- **Simulador:** equivale a um programa que controla modelos de um tipo específico ou age como uma interface para ferramentas/*softwares* externos (ex: pandapower, HDF5, etc...). Usualmente representa uma unidade modular conectada na co-simulação.
- ***Step* ou passo:** representa uma unidade de tempo discreta (inteiro) de execução do Mosaik. Uma unidade corresponde a uma quantidade específica de segundos definida no cenário (via `time_resolution`).
- **Fluxo de dados:** é a troca de informações (dados, atributos) realizada entre dois simuladores dada a partir da conexão entre duas entidades distintas.

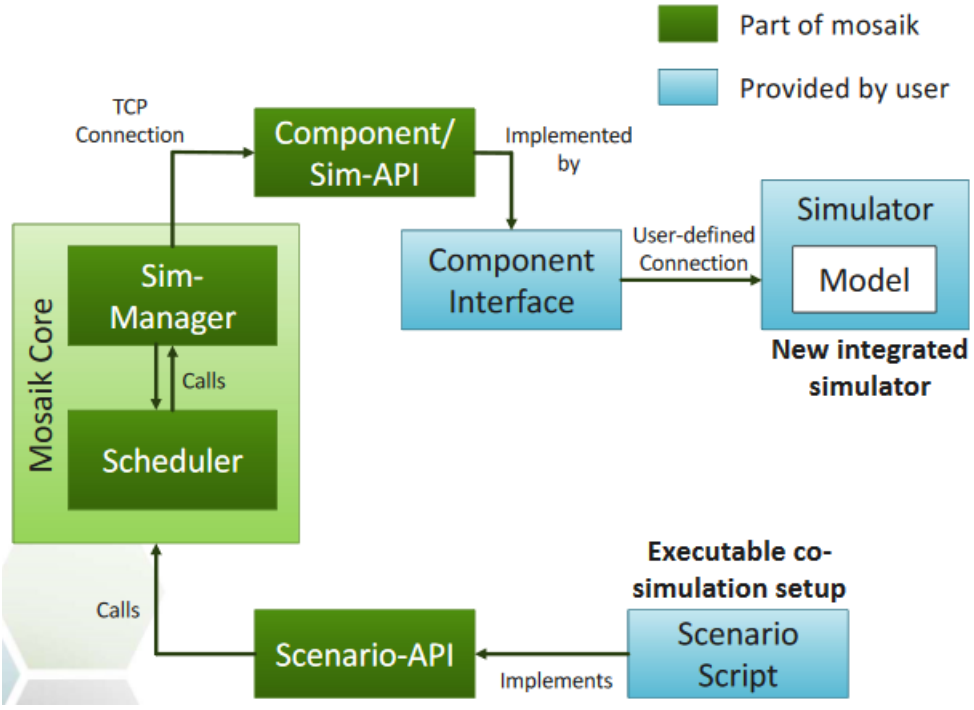
O Mosaik é constituído de quatro componentes principais que implementam os diferentes aspectos de uma estrutura de co-simulação - conforme mostrado na Figura 3. Tais componentes são definidos abaixo:

1. **Sim API:** fornece um protocolo de comunicação que permite trocas de mensagens entre diferentes simuladores e o Mosaik. Esta API tem duas variantes: uma API de *baixo-nível* e uma API *de alto-nível*.
2. **Scenario API:** cria um cenário de simulação (um ambiente) que configura entidades (simuladores). Esta API é responsável por instanciar modelos, inicializar simuladores, conectar entidades e estabelecer interações e fluxos de dados entre as entidades.
3. **SimManager:** lida com os processos do simulador e se comunica com eles. Ele é responsável por iniciar novos processos de simuladores, conectar-se aos já em execução ou executar simuladores ditos *"in-process"* - o que reduz a memória necessária e a sobrecarga de mensagens na rede.
4. **Scheduler:** coordena simulações de tipo evento discreto por meio da biblioteca Python SimPy. É responsável pelo tratamento de interações de tipos *tempo contínuo* e de *eventos*

[1] O glossário completo dos termos do Mosaik está disponível neste link: <https://mosaik.readthedocs.io/en/2.6.1/glossary.html?highlight=next_step#glossary>.

discretos - a depender das dependências ou tipos das entidades analisadas.

Figura 3 – Componentes principais de composição do Mosaik.



Fonte: (OFENLOCH *et al.*, 2022)

2.1.1.1 Manual de Implementação

Como apresentado na Figura 3, para realizar a integração de um simulador com o Mosaik é necessária a configuração de três elementos principais pelo usuário:

1. **Simulador/Modelo -** Classe de integração que implementa o modelo/*software*;
2. **Interface do componente (Sim API) -** Criação da interface de comunicação;
3. **Script do Cenário (Scenario API) -** Definição do ambiente executável do Mosaik.

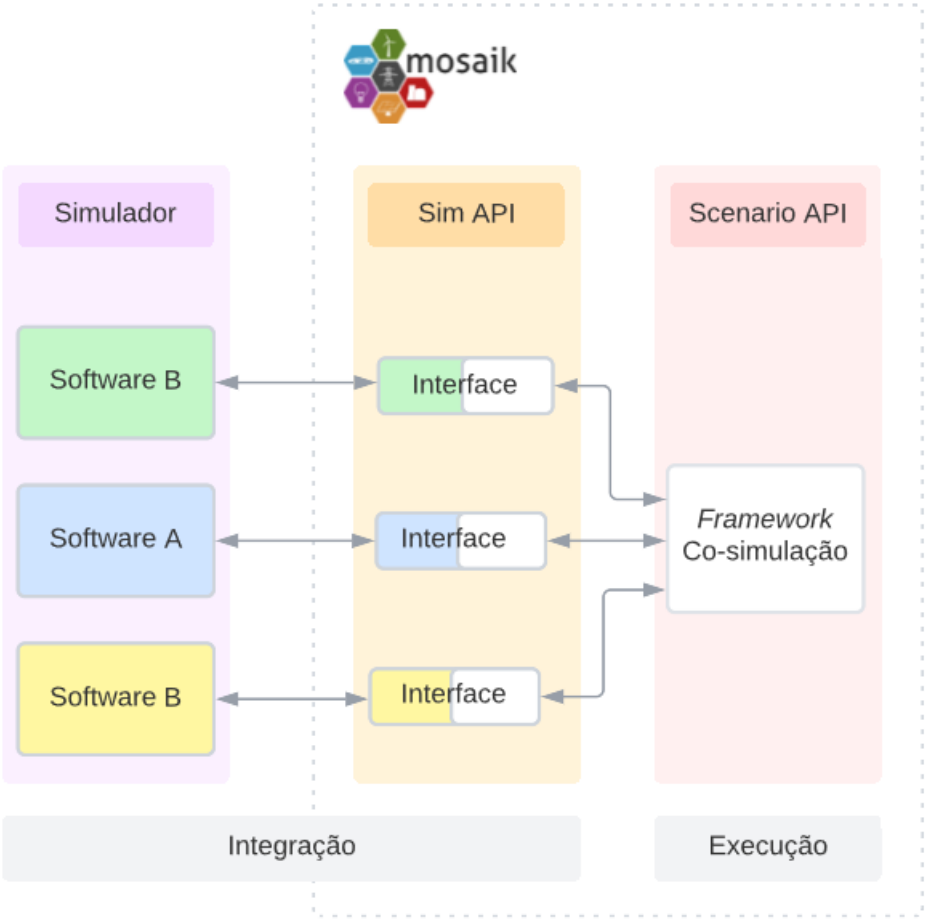
Os primeiros dois elementos da lista estão associados ao(s) *arquivo(s) de integração* de uma entidade no Mosaik. O simulador é a classe que implementa o modelo/*software* em análise. Essa classe pode implementar o modelo em um mesmo arquivo ou a partir de um outro arquivo. O Sim API é a estrutura de interface necessária para a comunicação entre o Mosaik e o simulador associado. É nele em que as definições de métodos de comunicação e as especificações (parâmetros, atributos, etc...) do simulador a ser integrado são definidas.

O terceiro componente da lista identifica o *arquivo de execução* da integração, ou seja, o Scenario API identifica o ambiente (cenário) de execução do próprio Mosaik. É nesse elemento em que todos os simuladores são inicializados e conectados entre si para início da

execução do Mosaik.

A Figura 4 apresenta graficamente os elementos implementados pelo usuário descritos na estrutura do Mosaik. Para realizar uma integração de simulador com Mosaik é obrigatória uma compreensão clara desses três elementos para estruturação da interface e ambiente de funcionamento do sistema de co-simulação. As seções a seguir descrevem em detalhes tais elementos.

Figura 4 – Elementos implementados pelo usuário para uma integração Mosaik.



Fonte: O próprio autor.

Para o contexto deste projeto, foi criado um repositório Github [2] que possui um exemplo executável simples, que pode ser utilizado como base para uma integração. Os códigos-fonte principais do repositório citado estão identificados no apêndice A. No Código-fonte 16, `example_model.py`, é possível ver o componente que determina o simulador/modelo da integração Mosaik. O Código-fonte 17, `simulator_mosaik.py`, representa a integração no Sim

[2] Repositório Github disponível em: <https://github.com/grei-ufc/tcc-yana-mosaik-examples>. É interessante comparar o código com as informações descritas em cada uma das seções a seguir.

API. Já o Código-fonte 18, `demo.py`, identifica o Scenario API do Mosaik.

2.1.1.1.1 Simulador/Modelo

Esse componente possui o simulador (modelo/*software*) que será integrado ao Mosaik. Partindo-se da documentação oficial do Mosaik (SCHERFKE, 2018), são apresentadas duas maneiras de compor uma entidade no Mosaik - a depender do contexto do *software*. Uma das opções para realizar essa integração é por meio de instância direta do modelo na interface de comunicação (Sim API) do Mosaik.

A segunda forma de realizar esse componente é criando uma classe integradora de simulador que instancia o modelo - e é esta classe integradora quem será chamada pelo Sim API.

Os códigos-fonte identificados no apêndice B permitem um comparativo entre as duas opções citadas, identificando sua influência no Sim API.

Primeiro consideraremos a opção de instanciamento direto do modelo na interface de comunicação do Mosaik. No Código-fonte 19, `model.py`, apenas o modelo a ser integrado no Mosaik é definido. No Código-fonte 20, `sim_api.py`, a classe `Model` é chamada diretamente no arquivo, onde são criadas as suas instâncias no próprio Sim API.

A segunda opção de implementação, que envolve a criação de um simulador de integração é apresentada a seguir. No Código-fonte 21, o arquivo `simulator.py`, define tanto o modelo na classe `Model`, quanto a classe integradora `Simulator` - que é a responsável pela criação de instâncias do modelo. Neste caso, o arquivo de interface `sim_api2.py` chamará esta classe integradora `Simulator` - o Código-fonte 22 representa essa opção.

Qualquer uma dessas opções pode ser utilizada para integração com o Mosaik. Basta identificar qual a sua necessidade e compreender que o objetivo principal desse componente é possibilitar a criação de instâncias no Sim API do Mosaik. Não há uma determinação pré-definida de quando uma opção deve ser utilizada em detrimento da outra. Portanto, esta decisão é determinada pelo próprio usuário que implementa a integração.

2.1.1.1.2 Interface do componente (Sim API)

Como descrito anteriormente, o Sim API é o elemento responsável pela comunicação entre Mosaik e simuladores - o equivalente a uma interface de comunicação (Figura 4). O Sim API é construído em um arquivo Python (extensão `.py`) e possui uma estrutura bem definida a

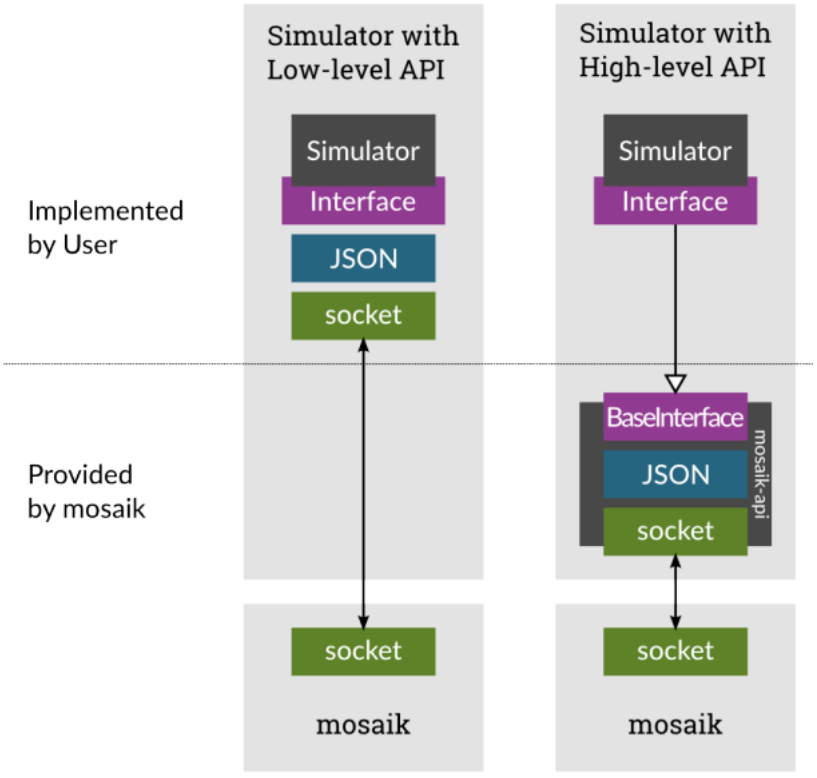
ser seguida para seu funcionamento adequado[3]. Nesta seção, será apresentado um resumo de como construir essa integração.

Para realizar a construção do Sim API e permitir a definição e comunicação de um simulador com o Mosaik, deve-se seguir os três passos principais abaixo:

1. Definição do tipo do Sim API (*alto-nível* ou *baixo-nível*).
2. Construção do dicionário de metadados do simulador.
3. Definição dos métodos de comunicação do Sim API para um simulador.

A primeira etapa para estruturar o Sim API de um simulador é definir se o simulador em questão pode se comunicar com o Mosaik por meio do Sim API de *alto-nível* ou do Sim API de *baixo-nível*. A Figura 5 resume a diferença entre esses tipos de simuladores.

Figura 5 – Diferenças entre Sim API de "baixo-nível" e de "alto-nível".



Fonte: (SCHERFKE, 2018)

O tipo de API de *baixo-nível* requer que o usuário implemente a comunicação de um simulador com o Mosaik através de mensagens codificadas em JSON em trocas TCP/IP via *sockets* de rede. Esse tipo de implementação é necessário quando um simulador possui um processo de *"loop"* diferente do Mosaik em seu núcleo ou quando um simulador não é

[3] É possível ler a documentação oficial do Mosaik sobre essa API em: <https://mosaik.readthedocs.io/en/latest/mosaik-api/index.html>.

implementado em Python ou Java. Essa implementação é mais complexa do que o API de *alto-nível*, já que ele exige que todas as estruturas de trocas de mensagens sejam desenvolvidas integralmente em JSON pelo usuário. [4].

Já o API de *alto-nível* fornece uma API compatível com Python e Java que facilita a comunicação entre simuladores e Mosaik. Esse tipo oferece métodos de comunicação "reutilizáveis" de uma classe disponibilizada pelo API (`mosaik_api.Simulator`) de maneira simples. Para utilizá-la, basta implementar uma subclasse da classe disponibilizada e implementar alguns métodos de comunicação que descrevem o que deve ser feito pelo simulador. Os métodos básicos que devem ser implementados são: `init()`, `create()`, `step()` e `get_data()`.

Após selecionado o tipo da Sim API adequada ao simulador em análise, pode-se seguir para a sua próxima etapa de construção da integração: a definição do dicionário de metadados do simulador. Os metadados são um dicionário Python que identifica todas as informações associadas ao simulador a ser integrado. Esse é o primeiro elemento a construir no arquivo de integração do Sim API. Nele, são definidas todas as informações sobre o simulador em análise para que o Mosaik possa reconhecê-lo, como: o tipo de execução do simulador; o nome dos modelos presentes no arquivo; quais seu parâmetros, atributos; tipo de visibilidade das informações; etc... O Código-fonte 1 apresenta um exemplo de estrutura desse elemento. Nos comentários do Código-fonte 17, `simulator_mosaik.py`, há uma descrição detalhada de cada um dos elementos disponíveis no componente de metadados.

```
META = {
    'api_version': '3.x.y',                     # Versao utilizada do Mosaik
    'type': 'time-based'|'event-based|hybrid',  # Tipo de execucao do simulador
    'models': {                                 # Modelos definidos nesse arquivo
        'ExampleModel': {
            'public': True|False,               # Define se um modelo pode ser instanciado
                                                # pelo usuario ou nao

            'params': ['param_1', ...],         # Parametros que podem ser passados
                                                # ao modelo em sua criacao

            'attrs': ['attr_1', ...],           # Atributos que podem ser acessados no
                                                #  modelo (leitura ou escrita)

            'any_inputs': True|False,           # Se {True}, qualquer atributo
                                                # pode ser associado a esse modelo

            'trigger': ['attr_x', ...]          # Atributos externos que causam um passo do
```

[4] Uma explicação detalhada sobre como implementar o API de baixo-nível está disponível na documentação oficial Mosaik: <https://mosaik.readthedocs.io/en/latest/mosaik-api/low-level.html>

```
19                                                  # simulador
20              'non-persistent': ['attr_y', ...]    # Atributos validos apenas por um passo de
21                                                  # execucao
22          },
23      },
24      'extra_methods': [                          # Lista adicional de metodos que o
25                                                  # simulador  pode prover alem das chamadas
26                                                  # API padrao
27          'method_example_1',
28      ]
29  }
```

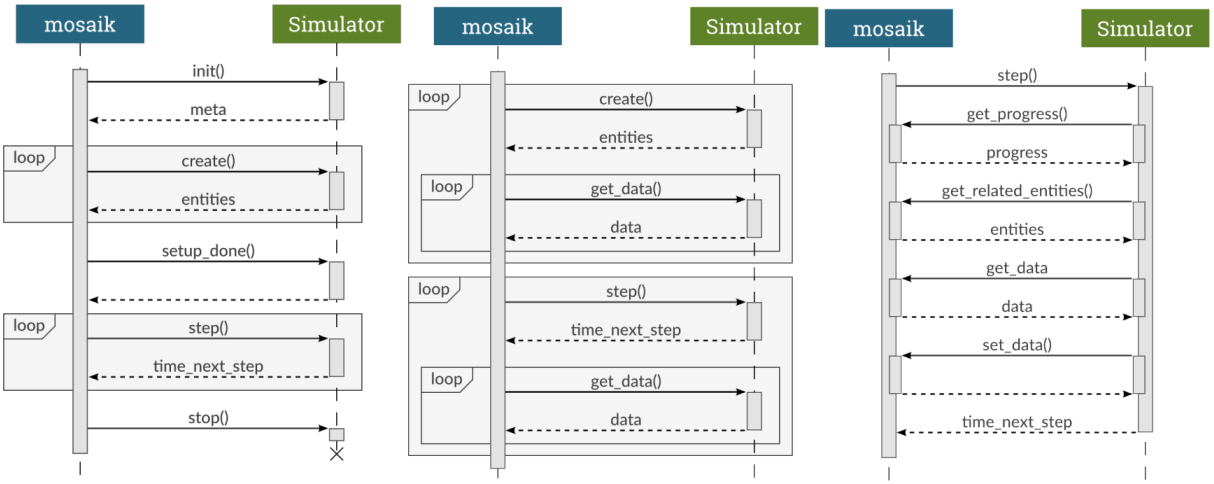
Código-fonte 1 – Exemplo de metadados do Sim API.

Após definição dos metadados do simulador, segue-se para a próxima etapa de construção da integração: o desenvolvimento dos métodos de comunicação do simulador com o Mosaik. A Figura 6 apresenta alguns diagramas de sequências com exemplos que identificam trocas de comunicação entre o Mosaik e um simulador e os métodos utilizados para tais transações. A depender do objetivo do simulador implementado, alguns dos métodos apresentados podem ou não ser utilizados [5]. Uma breve explicação dos métodos citados é apresentada abaixo:

- `init()` - o Mosaik inicia a troca de comunicação com um simulador. A partir desse método, o simulador retorna o dicionário de metadados (`meta`) que o descreve.
- `create()` - o Mosaik pode usar este método para instanciar modelos de um simulador.
- `setup_done()` - quando este método é chamado, um cenário Mosaik foi totalmente definido e configurado.
- `step()` - marca um passo de simulação ou *step* (definido pelo tempo de um simulador) no Mosaik. É neste método que cálculos, funções ou outras ações do simulador são definidas e executadas. Após execução do `step()`, o simulador pode retornar o valor da próxima execução via `next_step`, se necessário.
- `stop()` - é chamado quando o Mosaik solicita que os simuladores sejam desligados.
- `get_data()` - este método pode ser usado pelo Mosaik para acessar dados de um simulador e disponibilizá-los para outros simuladores. O simulador retorna um dicionário `data` com as informações necessárias.
- Solicitações assíncronas também são permitidas usando os métodos `get_progress()`, `get_related_entities()`, `get_data()` e `set_data()`.

[5] Uma explicação detalhada dos métodos do Sim API está disponível na documentação oficial Mosaik em <https://mosaik.readthedocs.io/en/latest/mosaik-api/low-level.html?highlight=init()#api-calls>.

Figura 6 – Alguns dos métodos disponíveis para comunicação entre simuladores no Mosaik.



Fonte: (SCHERFKE, 2018)

O Código-fonte 2 mostra um resumo[6] da estrutura-base para a integração de um simulador em um único arquivo. Essa estrutura é o fundamento necessário para realizar a interface de comunicação entre qualquer modelo/simulador com o Mosaik. A descrição das partes que compõem esta estrutura estão indicadas na Tabela 1.

Tabela 1 – Resumo de estrutura de integração Mosaik.

Integração Mosaik				
Tipo	**Nº**	**Elemento**	**Descrição**	**Observação**
Simulador/ Modelo	1	Classe do Modelo	O modelo/*software* que será integrado.	A depender da situação, o modelo pode ser instanciado diretamente no Sim API. Ou ser instanciado por um simulador de integração.
	2	Classe do Simulador	O simulador de integração do modelo.	
Sim API	3	Tipo da API	Determina se o API será do tipo "alto-nível" ou "baixo-nível".	-
	4	Metadados	É um dicionário que descreve o simulador para que o Mosaik possa reconhecê-lo.	
	5	Interface de comunicação	Classe que define os métodos de comunicação que permitem a troca de dados entre a entidade e o Mosaik.	Os métodos básicos para uma integração com Mosaik são: init(), create(), step(). Outros métodos estão identificados na Figura 6.

Fonte: O próprio autor.

[6] O resumo apresentado está disponível no repositório Github: <https://github.com/grei-ufc/tcc-yana-mosaik-examples/blob/main/base-model/random_simulator.py>.

```
# sim_api.py
"""
  Este codigo apresenta um resumo de todos os elementos necessarios para realizar
  uma integracao entre um simulador e o Mosaik.

"""
# 0. TIPO DO SIM API
import random
import mosaik_api    # importa API de alto-nivel

"""
  1. METADDOS DO SIMULADOR

"""
META = {  # Possui informacoes que definem o simulador para o Mosaik
  'models': {
    'RandomModel': {
      'public': True,
      'params': [],
      'attrs': ['val'],
    }
  }
}

"""
  2. MODELO

"""
class Model:
  def __init__(self):
    self.val = random.random()  # Este modelo cria valores randomicos

  def step(self):
    self.val += random.random()  # A cada passo, soma um novo valor randomico ao valor
    inicial

"""
  3. SIMULADOR

"""
class Simulator(object): # Serve como classe integradora para instanciar o modelo
  def __init__(self):
    self.models = []
    self.data = []

  def add_model(self):
    model = Model()   # Instancia os modelos
    self.models.append(model)
```

```
48      self.data.append([])
49
50    def step(self):
51      for i, model in enumerate(self.models):
52        model.step()
53        self.data[i].append(model.val) # adiciona o valor de *val* ao modelo *model* no
        atributo *data* do objeto *Simulator*
54
55  """
56    4. INTERFACE DE COMUNICACAO (SIM API)
57
58  """
59  class RandomSim(mosaik_api.Simulator):  # Herda classe abstrata de alto-nivel do Mosaik
60    def __init__(self): # Inicializa o simulador criado
61      super().__init__(META)
62      self.simulator = Simulator()
63      self.eid_prefix = 'RanModel'
64      self.entities = {}
65
66    def init(self, sid, eid_prefix = None): # Inicializacao enviada ao Mosaik (metadados do
        simulador)
67      if eid_prefix is not None:
68        self.eid_prefix = eid_prefix
69      return self.meta
70
71    def create(self, num, model): # Inicializa *num* instancias do modelo *model* para o API
        do Mosaik
72      next_eid = len(self.entities)
73      entities = []
74
75      for i in range(next_eid, next_eid + num):
76        eid = "%s%d" % (self.eid_prefix, i)
77        self.simulator.add_model()
78        self.entities[eid] = i
79        entities.append({"eid": eid, "type": model})
80      return entities
81
82    def step(self, time, inputs): # Obtem dados para o Mosaik
83      for eid, attrs in inputs.items():
84        for attr, values in attrs.items():
85          model_idx = self.entities[eid]
86
87      self.simulator.step()
88      return time + 60 # Tamanho do passo (step) eh  de um minuto
89
90    def get_data(self, outputs): # Permite que outros simuladores obtenham informacao do ```
        Model``
91      models = self.simulator.models
92      data = {}
```

```
93      for eid, attrs in outputs.items():
94        model_idx = self.entities[eid]
95        data[eid] = {}
96        for attr in attrs:
97          if attr not in self.meta['models']['RandomModel']['attrs']:
98            raise ValueError("Unknown output attribute: %s" % attr)
99
100         data[eid][attr] = getattr(models[model_idx], attr)
101     return data
```

Código-fonte 2 – Exemplo da estrutura necessária para uma integração Mosaik.

2.1.1.1.3 Script do Cenário (Scenario API)

O ambiente de execução do Mosaik é a API Scenario. Este elemento é responsável por iniciar entidades, instanciar modelos ou simuladores, conectar instâncias de diferentes entidades e permitir o fluxo de dados entre eles.

É possível definir a estrutura do Scenario API em 5 etapas:

1. Configuração do Scenario API;
2. Inicialização do simuladores;
3. Criação de instâncias dos modelos dos simuladores;
4. Conexão das entidades;
5. Execução da simulação.

O Código-fonte 3 mostra um exemplo da estrutura básica do Scenario API. A Tabela 2 resume e descreve cada um dos componentes constituintes dessa estrutura [7]. A seguir cada etapa é explicada brevemente.

A primeira etapa, - configuração do cenário -, requer a importação do API `mosaik` e a definição de um dicionário, que determina as informações referentes aos simuladores utilizados no cenário. Esse dicionário contém o nome dos simuladores utilizados na API e como eles podem ser executados. Ele é, então, passado como parâmetro no construtor `mosaik.World()` para a criação de um "mundo" - que representa o ambiente de execução do Mosaik.

Em seguida, é necessário inicializar os simuladores - que foram descritos no dicionário de configuração anteriormente definido. Para inicializar uma entidade, utiliza-se o método `World.start()`, passando-se o nome do simulador identificado no dicionário de configuração.

[7] Para mais detalhes sobre a estrutura do Scenario API, acesse a documentação oficial do Mosaik: <https://mosaik.readthedocs.io/en/latest/scenario-definition.html>.

A terceira etapa cria as instâncias dos simuladores inicializados na etapa anterior. Para tal, pode-se instanciar o modelo diretamente ou utilizar-se o método `create()`. Para instanciar o modelo diretamente, utiliza-se a variável da inicialização do modelo feita com `World.start()`. A partir dessa variável, instancia-se o modelo desejado. Alternativamente, utiliza-se o método `create()`, que pode criar várias instâncias simultaneamente. É importante destacar que conexões de fluxos de dados ditos "cíclicos" merecem especial atenção na hora da conexão. Essa configuração será descrita na seção sobre Mosaik 3.0.

Após a criação de instâncias dos simuladores, é realizada a conexão entre eles, por meio do método `connect()`. Esse método é utilizado para realizar troca de informações entre instâncias de entidades diferentes. No método `connect()` são, portanto, apresentados quais simuladores serão conectados e quais atributos são trocados entre eles.

Finalmente, por meio do método `world.run()`, o cenário é executado e a execução do sistema de co-simulação incia-se.

Tabela 2 – Resumo da estrutura de execução Mosaik.

Execução Mosaik				
Tipo	Nº	Elemento	Descrição	Observação
Scenario API	1	Configuração do cenário	Dicionário que identifica como os simuladores podem ser iniciados.	-
	2	Inicialização dos simuladores	Uma variável recebe a inicialização dos simuladores.	O método start() cria um ModelFactory que permite instanciar as entidades por instância direta do modelo ou por meio do método create().
	3	Criação de instâncias	Os simuladores iniciados têm suas instâncias criadas, a partir da variável criada na inicialização.	
	4	Conexão das entidades	As entidades são conectadas, o que permite o fluxo de dados entre os simuladores.	Pode-se utilizar os métodos connect(), connect_many_to_one() ou connect_randomly().
	5	Execução da simulação	Identifica o início da co-simulação e troca de dados entre as entidades.	Pode-se definir até quanto tempo a co-simulação será executada.

Fonte: O próprio autor.

```
# scenario_api.py
'''
Representa o Scenario API do Mosaik.
'''
import mosaik
import mosaik.util

"""
    1. CONFIGURACAO DO CENARIO
"""
SIM_CONFIG = {  # Determina os modelos utilizados no cenario
    'ExampleSim': {
        'python': 'simulator_mosaik:ExampleSim',
    },
    'ExampleCtrl': {
```

```
16          'python': 'controller:Controller',
17      },
18      'Collector': {
19          'cmd': '%(python)s collector.py %(addr)s',
20      },
21  }
22  END = 10  # 10 seconds
23
24  # Criacao do Mundo
25  world = mosaik.World(SIM_CONFIG)
26
27  """
28      2. INICIALIZACAO DOS SIMULADORES
29  """
30  examplesim = world.start('ExampleSim', eid_prefix='Model_')
31  examplectrl = world.start('ExampleCtrl')
32  collector = world.start('Collector')
33
34  """
35      3. CRIACAO DAS INSTANCIAS DOS MODELOS
36  """
37  monitor = collector.Monitor() # criacao de instancia direta
38
39  models = [examplesim.ExampleModel(init_val=i) for i in range(-2, 3, 2)] # criacao da
        instancia direta em um for
40
41  agents = examplectrl.Agent.create(len(models)) # criacao de instancia via create()
42
43  """
44      4. CONEXAO DAS ENTIDADES
45  """
46  for model, agent in zip(models, agents):
47      world.connect(model, agent, ('val', 'val_in'))
48      world.connect(agent, model, 'delta', weak=True)
49  mosaik.util.connect_many_to_one(world, models, monitor, 'val', 'delta')
50  mosaik.util.connect_many_to_one(world, agents, monitor, 'delta')
51
52  """
53      5. EXECUCAO DA SIMULACAO
54  """
55  world.run(until=END) # Identifica o inicio da execucao Mosaik
```

Código-fonte 3 – Exemplo da estrutura necessária para execução do Scenario API.

2.1.2 Mosaik 3.0

Em 2021, uma nova versão do Mosaik foi lançada, a fim de otimizar o tratamento de interações de simuladores que executam por eventos/tempo discretos - como, por exemplo, sistemas de comunicação que interagem com sistemas ciber-físicos. Essa atualização fez-se necessária, pois integrações desse tipo não eram suportadas adequadamente pelo Mosaik - gerando problemas de performance e de adaptação dos sistemas à interface.

A versão 3.0 do Mosaik trata, portanto, os problemas identificados anteriormente no Mosaik para esses casos de uso, suportando, assim, cenários híbridos sem afetar drasticamente as definições de interface já existentes no Mosaik. As seções a seguir identificam conceitos essenciais para a compreensão de execução de simuladores no Mosaik 3.0. Em seguida, na seção "Melhorias implementadas" (seção 2.1.2.4) são descritos individualmente as novas ferramentas aplicadas no Mosaik 3.0 em comparação com o Mosaik 2.

2.1.2.1 Paradigmas de tempo

Para compreender melhor as mudanças associadas à nova versão Mosaik, é importante destacar quais são os cenários que foram ajustados para esta atualização. Enquanto sistema de co-simulação, o Mosaik rastreia o tempo de execução de cada simulador individualmente a fim de permitir a sincronização entre as simulações de seus componentes. É necessário, portanto, compreender a diferença entre simuladores que executam em tempo contínuo e simuladores que funcionam por eventos ou tempo discreto para permitir a sincronização correta do Mosaik. Existem diferentes interpretações para essas classificações, mas a documentação oficial[8] do Mosaik determina tais categorias como descrito a seguir.

De acordo com o Mosaik, execuções por *tempo contínuo* são definidas como simulações associadas ao contexto do mundo físico, ou seja, sistemas que funcionam no tempo corrente do nosso cotidiano. Por exemplo, no contexto de REI, a saída de potência ativa de uma placa fotovoltaica a cada instante de tempo equivaleria a uma simulação de tempo contínuo.

Já simuladores classificados por tempo ou evento discreto estão, usualmente, relacionados a elementos do mundo digital. Simulações do tipo *evento discreto* são determinados como simuladores que são acionados a partir da criação de um evento ou pela disponibilidade de dados ao simulador em questão. Já simuladores de *tempo discreto* identificam simulações que

[8] Para mais detalhes sobre as classificações propostas pelo Mosaik, acesse: <https://mosaik.readthedocs.io/en/latest/scheduler.html>.

só são executados por unidades inteiras de tempo, “pulando” entre essas unidades, ao invés de considerar todo o intervalo real de tempo.

O Mosaik considera ainda uma terceira classificação de simuladores, definida como *híbrida*: reservada para simuladores que possam ter comportamentos mistos entre os dois tipos definidos anteriormente.

É interessante destacar que as classificações apresentadas pelo Mosaik possuem diferentes interpretações de implementação e, a depender do contexto, um mesmo simulador pode ser considerado com um tipo diferente a depender de seu cenário de uso.

A Figura 7 apresenta uma representação visual da diferença de execução de um simulador em tempo contínuo e em tempo/evento discreto. Como observado na imagem, o gráfico (a) apresenta um simulador em tempo contínuo. Nessa classificação de simuladores, todos os simuladores são iniciados no tempo $t = 0$ do Mosaik. Ao realizar um `step()`, o Mosaik passa o tempo atual (t_{now}) para o simulador. A partir disso, é necessário que o próprio simulador identifique no retorno do seu método `step()` qual o próximo tempo de execução para o simulador (t_{next}). Isso permite que o tempo de execução (*step*) de um simulador de tempo contínuo possa ser variável a depender da implementação.

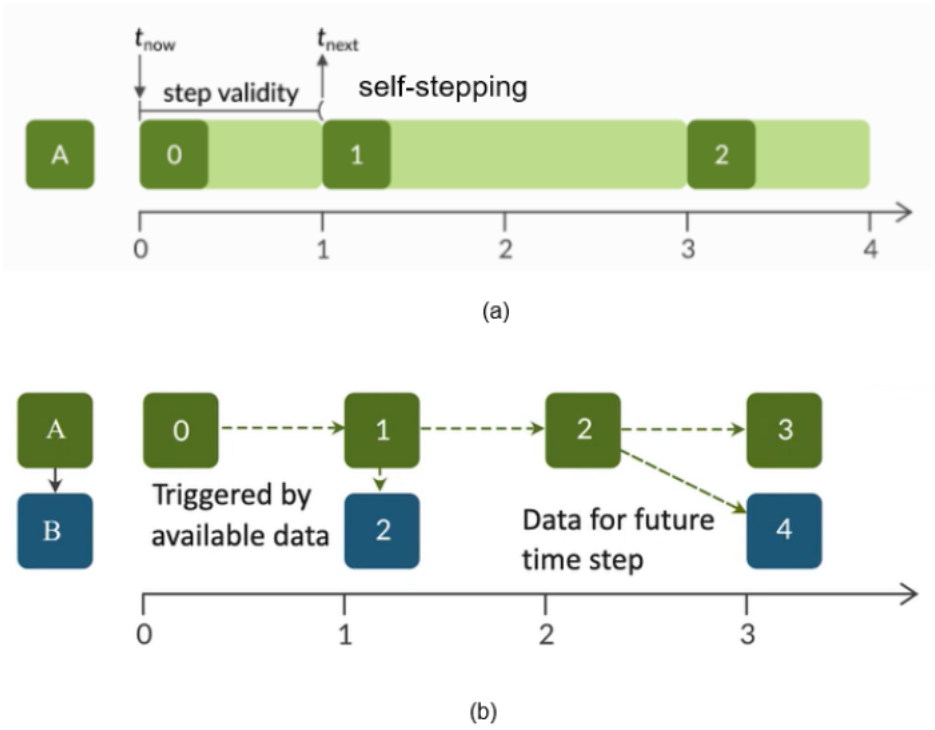
O gráfico (b) identifica um simulador baseado em eventos (simulador B). Como apresentado, o simulador só é executado quando um evento é acionado por outro simulador ou pelo próprio simulador a partir de especificações definidas no seu método `step()`. Os dados desse tipo de simulador são ditos “não-persistentes” e só são válidos o *step* atual. Esse tipo de simulador não se inicia no tempo $t = 0$ do Mosaik, mas apenas a partir do ponto em que há eventos que o acionem ou por meio do método `World.set_initial_event()` definido no Scenario API.

2.1.2.2 Fluxo de dados

É preciso ficar atento quando cria-se um fluxo de dados no Mosaik - ou seja, quando realiza-se conexão entre entidades -, pois um simulador somente consegue realizar um `step()` se houver tempo suficiente para que todos os dados recebidos por ele (*inputs*) sejam computados. A Figura 8 representa essa situação, na qual o simulador B só pode acionar o seu próximo *step* quando o tempo de step de A ($t_{next(A)}$) tiver sido igual ou superior ao tempo de computação dos dados em B ($i+1$). Enquanto $t_{next(A)}$ for inferior a $i+1$, B não pode realizar um *step*.

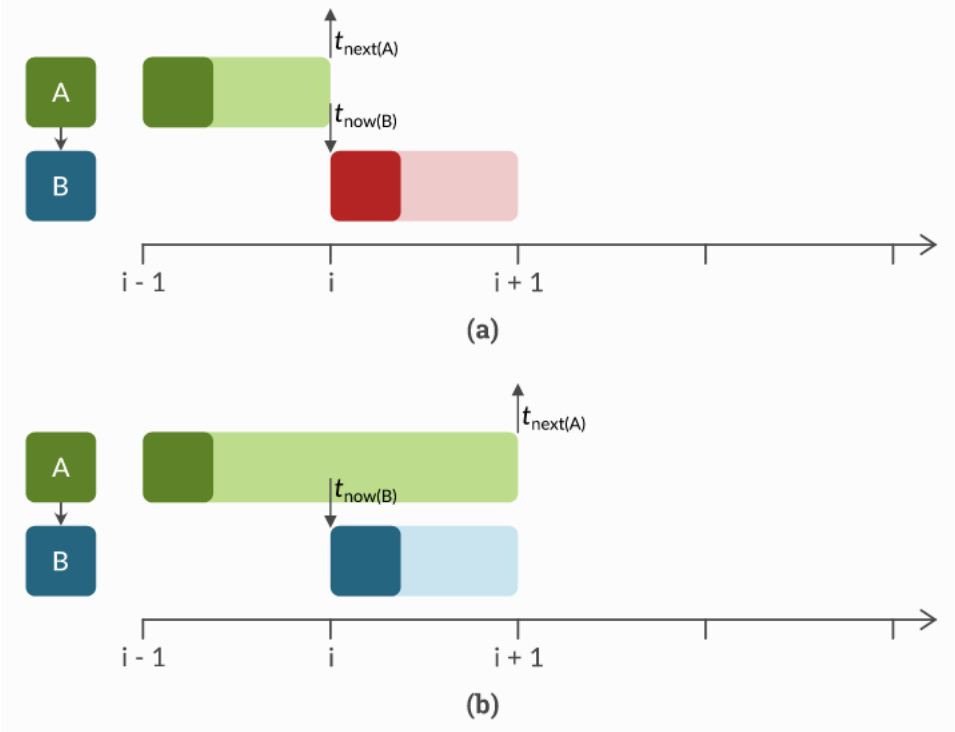
Considerando-se essa estrutura de fluxo de dados, um simulador de tempo/evento

Figura 7 – Paradigma de tempo e execução Mosaik: classificação de simuladores.



Fonte: (OFENLOCH *et al.*, 2022).

Figura 8 – Lógica de fluxo de dados entre simuladores no Mosaik.



Fonte: (SCHERFKE, 2018).

discreto identifica até onde pode avançar internamente na sua execução (tempo $i + 1$) por meio do parâmetro `max_advance`, identificado em seu método `step()`. Esse argumento garante que nenhum *step* será executado até que esse limite de tempo seja atingido. Para simuladores de

tempo contínuo, o valor de `max_advance` é sempre igual ao final da execução da simulação - que eles mesmo definem via o retorno de sua próxima execução, t_{next}, no retorno do método `step()`.

Após a determinação das conexões e dos tempos de execução dos `step()` de cada simulador, o Mosaik identifica qual será a ordem de fluxo dos dados. Os simuladores não trocam suas informações diretamente - é o Mosaik quem define isso por meio das conexões de entidades e fluxo de dados gerenciado pelo sistema de co-simulação. O Mosaik, portanto, gera chamadas de API `get_data()` para realizar essa troca de informações.

A seção a seguir identifica algumas particularidades de tipos de fluxos de dados que devem ter especial atenção na hora de sua implementação.

2.1.2.3 Conexões Cíclicas

Por vezes, alguns cenários exigem a criação de fluxos de dados ditos "cíclicos" - em que os dados de um simulador dependem dos dados de outro. Num contexto de REI, um exemplo seria quando um controlador (C) precisa dos dados de um agente inteligente (E) para limitar seus valores. A Figura 9 representa casos deste exemplo.

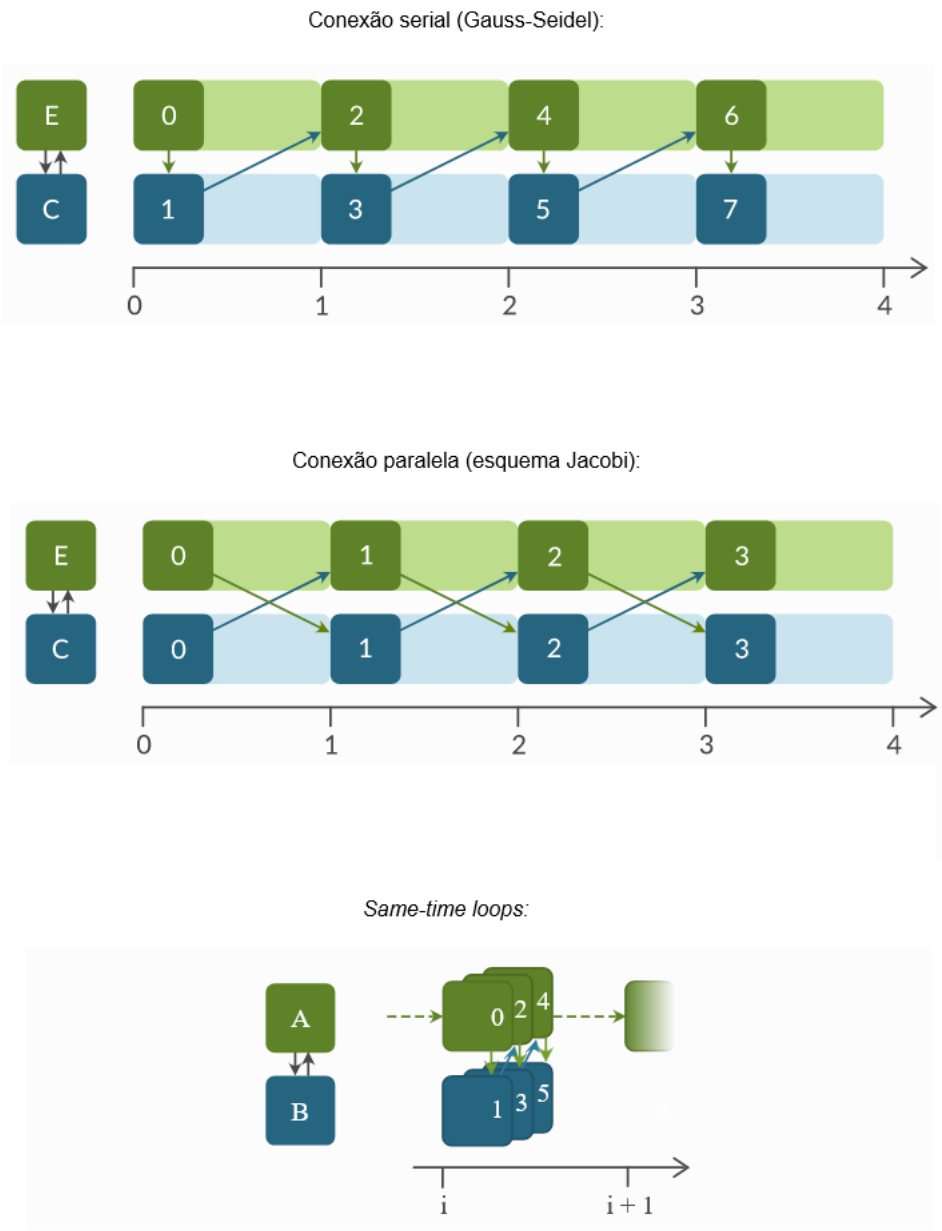
Infelizmente, o Mosaik não trata esse tipo de cenário automaticamente. Ou seja, não seria possível a execução de uma conexão cíclica como identificada diretamente no Código-fonte 4. Existem, no entanto, quatro alternativas para tratar esse tipo de situação: conexões seriais, conexões paralelas, conexões assíncronas e *same-time loops*.

```
1  # scenario_api-conexao_direta
2  world.connect(E, C)
3  world.connect(C, E)
```

Código-fonte 4 – Conexão cíclica direta: impossível no Mosaik.

A solução chamada *conexão serial* (ou Gauss-Seidel), apresenta-se como identificada na Figura 9 e trata casos que utilizem simuladores de tempo contínuo. Para aplicar essa solução, executa-se primeiramente o simulador E e utiliza-se a *flag* `time_shifted=True` em uma das conexões. Isso permite que o Mosaik compreenda que um simulador depende do *output* de outro simulador para sua execução. Além da indicação da *flag*, é necessário prover também um valor inicial (`initial_data`) utilizado na minha primeira execução do *step*. O Código-fonte 5 apresenta como seria feita a conexão nesse caso.

Figura 9 – Soluções para fluxo de dados cíclicos de simuladores.



Fonte: (SCHERFKE, 2018).

```
# scenario_api-conexao_serial
world.connect(E, C)
world.connect(C, E, ("c_out", "a_in"), time_shifted=True, initial_data={"c_out": 0})
```

Código-fonte 5 – Conexão cíclica serial.

A *conexão paralela* (ou esquema Jacobi) é acionada quando as conexões de ambos simuladores possuem a *tag* `time_shifted` e o valor inicial (`initial_data`) acionados. Esta

situação também está indicada na Figura 9. Nessa situação, os simuladores são executados em paralelo. Só é possível gerar esse tipo de conexão se os simuladores não forem executados na chamada execução *"in-process"* do Scheduler Mosaik.

```
# scenario_api-conexao_paralela
world.connect(E, C, ("a_in", "c_out"), time_shifted=True, initial_data={"a_in": 0})
world.connect(C, E, ("c_out", "a_in"), time_shifted=True, initial_data={"c_out": 0})
```

Código-fonte 6 – Conexão cíclica paralela.

Outra solução para tratar conexões cíclicas co-dependentes é a utilização de *conexões assíncronas*. Esse tipo de conexão é realizado por meio da *flag* `async_requests=True`, que aciona conexões assíncronas. As trocas de mensagens nas conexões assíncronas são determinadas no Sim API do simulador. Para este exemplo, utilizaria-se o método assíncrono `set_data()` no método `step()` do simulador C para realizar a troca de informações do controlador ao agente. A vantagem de utilização de chamadas assíncronas como neste caso é que o envio dos dados do controlador C ao agente E não seria obrigatoriamente realizado a cada *step*, mas poderia seguir uma lógica alternativa - implementada no método `step()`.

Finalmente, a última conexão possível - e a mais interessante no contexto de simuladores discretos - é o sistema de *same-time loops* ("iteraçãos instantâneas", em tradução livre). Esse fluxo de dados cíclico permite que várias trocas de informação aconteçam em um mesmo intervalo de tempo para simuladores de tipo evento discreto. Normalmente, esse caso de uso é utilizado no contexto de sistemas de comunicação - que, usualmente, possuem uma grandeza de tempo de execução significativamente menor do que simuladores que representam sistemas físicos. Como apresentado na Figura 9, essa configuração representa múltiplas trocas de informação em um mesmo intervalo de tempo, devido à sua capacidade de "instantaneidade" das trocas de dados por possuir uma grandeza pequena de execução nessa comunicação.

Para que a conexão de tipo *same-time loops* seja funcional, são necessárias três configurações:

1. As conexões entre os simuladores deve ser definida como `weak`;
2. Deve ser retornado o tempo do passo atual, `time`, no dicionário de dados, `data`, do método `get_data()`;
3. Para sair do ciclo, os valores do dicionário de `get_data()` devem ser esvaziados por um dos simuladores do ciclo.

Diante disso, a primeira etapa requer que uma das conexões entre os simuladores seja definida como `weak=True` no Scenario API. A seguir, os atributos trocados na conexão cíclica devem ser transferidos por meio do método `get_data()`, indicando-se o tempo atual no retorno do *step*. Para sair do ciclo é necessário que os atributos da conexão sejam omitidos no dicionário de retorno do `get_data()` do outro simulador; ou que um novo valor de tempo no *step* seja retornado. A documentação oficial do Mosaik destrincha essa situação em um tutorial[9] . O Código-fonte 7 identifica a conexão utilizada entre os simuladores para obter o efeito de *same-time loops* no tutorial citado - essa lógica é estendida para o exemplo de conexão da Figura 9.

```
# scenario_api-connexoes
world.connect(E, C)
world.connect(C, E, weak=True)

# sim_api-controlador    (C)
 def get_data(self, outputs):
    data = {}
    for agent_eid, attrs in outputs.items():
        for attr in attrs:
            if attr != 'delta_out':
                raise ValueError('Unknown output attribute "%s"' % attr)
            if agent_eid in self.data:
                data['time'] = self.time     # tempo atual do step eh permutado no
                                             # metodo get_data()
                data.setdefault(agent_eid, {})[attr] = self.data[agent_eid][attr]
    return data

# sim_api-agent      (E)
def step(self, time, inputs, max_advance):
    self.time = time
    data = {}
    for agent_eid, attrs in inputs.items():
        delta_dict = attrs.get('delta', {})
        if len(delta_dict) > 0:      # se houver 'delta_out' disponiveis pelo
                                     # controlador, esvazia o dicionario de dados para
                                     # finalizar o same-time loop
            data[agent_eid] = {'delta': list(delta_dict.values())[0]}
            continue
```

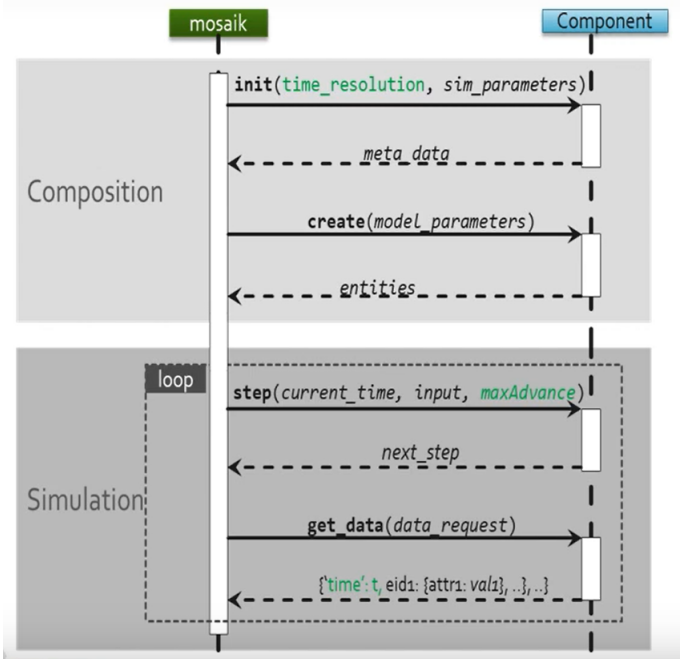
Código-fonte 7 – Conexão de *same-time loops*.

[9] Tutorial disponível em: <https://mosaik.readthedocs.io/en/latest/tutorials/sametimeloops.html>.

2.1.2.4 Melhorias implementadas

Esta seção detalhará as justificativas das modificações individuais realizadas na nova versão do Mosaik 3.0, definindo um breve comparativo entre a implementação para o Mosaik 2 e o Mosaik 3[10].

Figura 10 – Exemplo de métodos de comunicação utilizados entre o Mosaik e um simulador. Os novos parâmetros da atualização Mosaik 3.0 estão destacados na cor verde.



Fonte: (OFENLOCH *et al.*, 2022).

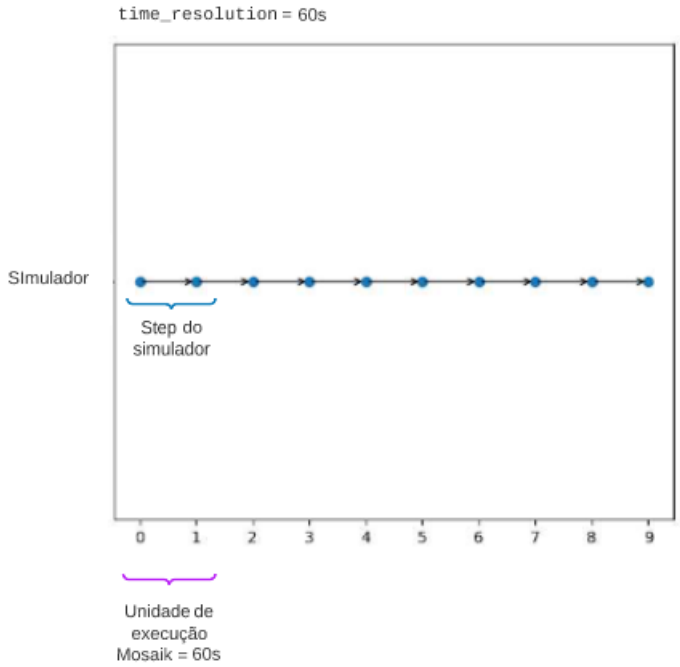
2.1.2.4.1 Tempo de resolução Mosaik

Uma das atualizações realizadas pela nova versão do Mosaik associa-se à capacidade de simuladores adaptarem-se a depender do passo de tempo de execução do Mosaik. *Time resolution* é a determinação da unidade de tempo utilizada na execução do Mosaik. O Mosaik utiliza valores inteiros para representar o seu tempo de execução. Esse valor pode ser identificado em milissegundos, segundo, minutos ou outra unidade de tempo. Por exemplo, um `time_resolution` de 15 minutos identificaria que cada passo de execução do Mosaik é executado no tempo de 15 minutos. A Figura 11 apresenta um exemplo gráfico dessa situação.

No Mosaik 2, havia a possibilidade de definição de diferentes `time_resolution` de forma implícita dentro de cada simulador. No entanto, os simuladores nunca recebiam essa

[10] As explicações abaixo são baseadas na apresentação disponível em <https://www.youtube.com/watch?v=Hlb6pGlyFxE&ab_channel=ERIGrid2.0-H2020ResearchInfrastructureProject>.

Figura 11 – Representação gráfica de um *step* do Mosaik.



informação diretamente do Mosaik, para identificar qual o tempo de execução atual do Mosaik. Com o Mosaik 3.0, essa definição agora é explícita e tratada diretamente pelo Mosaik. Com isso, o Mosaik comunica aos simuladores qual o passo de tempo do Mosaik, o que permite uma adaptação dos simuladores em relação ao valor reportado - por exemplo, um controlador que precisa realizar uma pausa para realizar cálculos em determinado tempo t_x do Mosaik receberá a informação desse tempo e poderá realizar as execuções necessárias.

2.1.2.4.2 Parâmetros dos metadados

Dentre as alterações da nova versão do Mosaik, a implementação de tratamentos para simuladores que executam baseados em tempo discreto ou evento discreto é a mais impactante em relação à sua versão anterior. Na versão 2 do Mosaik não havia uma diferenciação entre simuladores de tempo discreto ou contínuo. O Mosaik considerava que todos os simuladores executavam em tempo contínuo e, portanto, o tratamento de execução de simuladores por eventos ou por tempo discreto deveria ser realizada internamente pelos simuladores - ocasionando problemas de performance. Nessa perspectiva, duas novas alterações associadas à capacidade de execução por disponibilidade de dados de outros simuladores foram adicionadas: adição de novas características no dicionário de metadados de simuladores e alteração do retorno do método `step()`.

A primeira alteração, associada ao dicionário de configuração de simuladores (ex-

plicado na seção 2.1.1.1.2), agora dispõe de novos parâmetros que tratam simuladores de evento/tempo discretos pelo próprio sistema de co-simulação. Como visto na Figura 12, o novo dicionário de metadados possui três novos parâmetros: `trigger`, `non-persistent` e `type`. Esses parâmetros recebem uma lista de atributos.

O parâmetro `trigger` identifica que um simulador deve ser executado assim que houver novos dados disponíveis desse atributo. Já o parâmetro `non-persistent` identifica que o atributo em questão só é válido por um passo de execução (*step*). O parâmetro `type` determina o tipo de simulador utilizado de maneira mais direta - baseando-se nos dois parâmetros anteriores. Com o parâmetro `type`, é possível identificar de maneira fácil os atributos - sem a necessidade de definir individualmente os atributos nas listas `trigger` e `non-persistent`. A definição de cada `type` é descrita abaixo:

- `time-based` - simulador tradicional do Mosaik 2; simulador performa a própria execução de passo (*self-stepping*) e possui dados de tipo persistente (válidos para todos os passos de execução); é o equivalente a não setar nenhum atributo nos parâmetros `trigger` e `non-persistent`.
- `event-based` - todos os atributos especificados na lista de metadados (`attrs`) são acionados a partir de novos dados dos atributos e são não-persistentes (válidos apenas para um passo de execução); é o equivalente a setar todos os atributos nos parâmetros `trigger` e `non-persistent`.
- `hybrid` - os tipos dos atributos serão definidos baseados nos parâmetros `trigger` e `non-persistent`; podem existir atributos persistentes ou não/acionados por novos dados ou não.

Figura 12 – Comparação dos metadados das versões Mosaik.

Mosaik 2

```
{
    'api_version': 'x.y',
    'models': {
        'ModelName': {
            'public': True|False,
            'params': ['param_1', ...],
            'attrs': ['attr_1', ...],
            'any_inputs': True|False,
        },
        ...
    },
    'extra_methods': [
        'do_cool_stuff',
        'set_static_data'
    ]
}
```

Mosaik 3

```
{
    'api_version': 'x.y',
    'type': 'time-based'|'event-based'|'hybrid',
    'models': {
        'ModelName': {
            'public': True|False,
            'params': ['param_1', ...],
            'attrs': ['attr_1', ...],
            'any_inputs': True|False,
            'trigger': ['attr_1', ...],
            'non-persistent': ['attr_2', ...],
        },
        ...
    },
    'extra_methods': [
        'do_cool_stuff',
        'set_static_data'
    ]
}
```

Fonte: (SCHERFKE, 2018).

2.1.2.4.3 Retorno do método step()

Ainda associado à capacidade de um simulador executar um passo partindo da disponibilidade de dados de outra entidade, foi realizada uma alteração no método `step()`. No Mosaik 2, a configuração do método `step()` exigia, obrigatoriamente, o retorno do argumento `next_step` - que indicaria ao Mosaik o novo tempo de simulação em que o simulador deveria ser executado novamente. Lembre-se que, no Mosaik 2, os simuladores são equivalentes ao tipo `time-based` da versão 3.0 do Mosaik.

No Mosaik 3.0, esse argumento agora é opcional, a fim de tratar os simuladores acionados por eventos discretos. Isso foi feito porque simuladores acionados por eventos discretos não possuem a necessidade de identificar o tempo em que serão executados novamente - já que o que define sua execução é o recebimento de novos dados. Essa nova configuração permite, portanto, a execução de simuladores de evento discreto apenas quando há disponibilidade de dados recebidos por outros simuladores. Ou seja, a sequência de execução dos simuladores não é mais tratada pelos próprios simuladores com o uso do método `step()`, mas, sim, tratada pelo tipo do simulador executado e pela disponibilidade de dados de outros simuladores.

Para aplicações práticas, deve-se levar em consideração que, caso um simulador seja identificado como tipo `time-based`, ele deverá retornar no método `step()` o tempo de seu próximo passo de execução, `next_step`. No entanto, caso um simulador seja definido como `event-based`, o retorno do `next_step` no método `step()` é opcional e depende da aplicação necessária ao simulador.

2.1.2.4.4 Retorno do método get_data()

No Mosaik 2, por muitas vezes, simuladores são executados desnecessariamente a fim de receber os dados de outros simulador. No contexto de simuladores discretos no Mosaik 2, essa situação causa problemas de performance devido a essas execuções desnecessárias.

Como visto anteriormente, o tratamento de tipos de simulador pelo Mosaik 3, evita tais problemas de performance, ao possibilitar o tratamento desses tipos de simulador. Dentre as soluções que possibilitam essa melhoria, está a adição do parâmetro `time` no método `get_data()`. Esse parâmetro permite que o retorno de dados específicos só seja realizado em um tempo futuro `time`. Essa característica evita execuções de *steps* desnecessários de alguns simuladores discretos. A Figura 7 (b) identifica essa configuração, na qual o simulador A, em

seu passo 2, aciona a disponibilização de dados para o simulador B apenas em uma execução futura (*step* 3 do simulador A).

A Figura 10 indica onde esse parâmetro está identificado na troca de mensagens do Mosaik.

2.1.2.4.5 Parâmetro max_advance

A fim de tratar os casos de uso de simuladores de evento/tempo discreto, foi adicionado um novo parâmetro, `max_advance`, no método `step()`. Esse parâmetro indica para o simulador quanto ele pode avançar no tempo sem causar um erro de causalidade - ou seja, ele poderá executar até o tempo `max_advance` sem interrupções de outros simuladores ou execução de passo (a não ser que um *loop* dependente seja executado antes de atingir o valor de `max_advance`).

O Mosaik **deduz o valor** do `max_advance` a partir da topologia da simulação e do progresso dos simuladores envolvidos. Para simuladores de tipo `time-based` ou `hybrid` com atributos no `trigger`, o valor do parâmetro `max_advance` sempre equivale ao tempo do final da simulação (`entil=END`) Mosaik. Devido a isso, os simuladores desses tipos podem perder informações caso seus *steps* sejam maiores que os de seus valores de entrada.

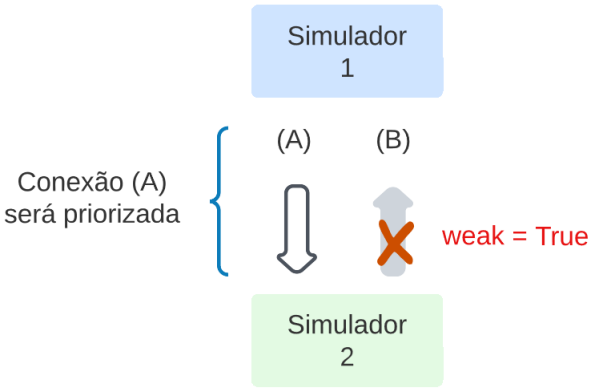
2.1.2.4.6 Parâmetro weak

Considerando-se o contexto de uso de simuladores de tempo ou evento discreto em *same-time loops*, a conexão entre essas entidades, quando são consideradas cíclicas, exige a utilização do parâmetro `weak` em um dos seus métodos `connect()`. Esse parâmetro identifica qual das conexões será priorizada entre as entidades conectadas. Ou seja, caso os dois simuladores tenham um passo no mesmo tempo, o `weak` identificará qual simulador será executado primeiro. A imagem 13 identifica uma conexão cíclica e o efeito do uso do parâmetro. O Código-fonte 8 identifica como seria utilizada uma conexão desse tipo entre simuladores de eventos discreto.

```
1    # scenario_api-weak_att
2    world.connect(E, C)
3    world.connect(C, E, weak=True) # essa conexao sera priorizada
```

Código-fonte 8 – Conexão de same-time loops - parâmetro weak.

Figura 13 – Efeito da utilização do parâmetro weak.

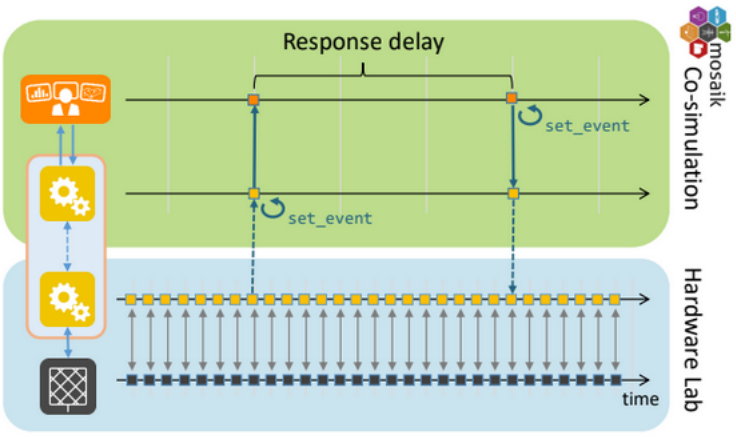


Fonte: O próprio autor.

2.1.2.4.7 Eventos externos

Finalmente, a última adição importante do Mosaik 3.0 em relação ao seu predecessor é a capacidade de integração de eventos externos por meio do método assíncrono `set_event(time_step)`. Esse método possibilita o tratamento de casos de uso denominados *"Human-in-the-loop"* - no qual um acionador externo, como uma pessoa, por exemplo, aciona algum evento que interaja com um sistema. A Figura 14 apresenta um exemplo dessa situação, na qual há interação entre um laboratório real e uma simulação no Mosaik.

Figura 14 – Exemplo de tratamento de eventos externos no Mosaik 3.0.



Fonte: (OFENLOCH *et al.*, 2022).

2.1.2.5 Comparação de exemplos

Para compreender de maneira mais prática as melhorias propostas pela nova versão do Mosaik 3, foi realizado um estudo comparativo entre os códigos dos tutoriais disponibilizados

pelas duas versões na documentação oficial do Mosaik.

A versão 2 e a versão 3 do Mosaik disponibilizam dois tutoriais equivalentes que realizam uma integração Mosaik. No entanto, há algumas mudanças mínimas entre os tutoriais, devido às atualizações da nova versão do sistema de co-simulação.

O apêndice C identifica o comparativo de cada arquivo dos tutoriais, apresentando as principais diferenças entre os códigos para o Mosaik 2 e o Mosaik 3.0[11].

2.1.2.6 Manual de Adaptação Mosaik 3.0

Caso exista um projeto já desenvolvido com Mosaik 2 e haja interesse em explorar as melhorias propostas pelo Mosaik 3, a adaptação entre as versões é bem simples. Basicamente, apenas 3 passos básicos devem ser seguidos para que um simulador seja executado na nova versão:

1. **Definição do tipo de simulador -** assim como descrito na seção 2.1.2.4.2, o parâmetro `type` do dicionário de metadados que define um simulador deve ser definido. Se necessário, os parâmetros `trigger` e `non-persistent` também podem ser preenchidos.
2. **Determinação do parâmetro** `time_resolution` **-** deve ser identificado o `time_resolution` no cenário (Scenario API) da simulação Mosaik. O valor-padrão desse parâmetro é setado para `1.`.
3. **Configuração do parâmetro** `max_advance` **-** no método `step()` é necessário adicionar o parâmetro `max_advance`, que apresenta quão distante um simulador pode avançar no tempo sem gerar um erro de causalidade. Vale destacar que essa adição é apenas para adequação à nova versão Mosaik - no entanto, o valor do parâmetro não pode ser definido explicitamente pelo usuário, já que é deduzido pelo próprio Mosaik.

É importante destacar que, a depender do simulador implementado e da lógica associada à execução do mesmo, pode haver a necessidade de alterar mais alguma configuração dentre as definidas na seção de Melhorias (seção 2.1.2.4).

2.1.3 Sumário de Migração ao Mosaik 3.0

Esta seção identifica, em uma única tabela, um resumo de todas as informações apresentadas no capítulo para fácil consulta durante a adaptação ou a construção de uma integração

[11] Os códigos executáveis dos tutoriais estão identificados no repositório: <https://github.com/grei-ufc/tcc-yana-mosaik-examples/tree/main/tutorials>.

com o sistema de co-simulação Mosaik. A Figura 15 identifica os passos necessários para a integração, execução ou adaptação de um simulador no Mosaik.

Figura 15 – Resumo de implementação Mosaik.

Integração Mosaik				
Tipo	**Nº**	**Elemento**	**Descrição**	**Observação**
Simulador/Modelo	1	Classe do Modelo	O modelo/software que será integrado.	A depender da situação, o modelo pode ser instanciado diretamente no Sim API. Ou ser instanciado por um simulador de integração.
	2	Classe do Simulador	O simulador de integração do modelo.	
Sim API	3	Tipos da API	Determina se o API será do tipo "alto-nível"ou "baixo-nível".	-
	4	Metadados	É um dicionário que descreve o simulador para que o Mosaik possa reconhecê-lo.	
	5	Interface de comunicação	Classe que define os métodos de comunicação que permitem a troca de dados entre a entidade e o Mosaik.	Os métodos básicos para uma integração com Mosaik são: `init()`, `create()`, `step()`.
Execução Mosaik				
Tipo	**Nº**	**Elemento**	**Descrição**	**Observação**
Scenario API	1	Configuração do cenário	Dicionário que identifica como os simuladores podem ser iniciados.	-
	2	Inicialização dos simuladores	Uma variável recebe a inicialização dos simuladores.	O método `start()` cria um `ModelFactory` que permite instanciar as entidades por instância direta da calsse do modelo ou por meio do método `create()`.
	3	Criação de instâncias	Os simuladores iniciados têm suas instâncias criadas, a partir da variável criada na inicialização.	
	4	Conexão das entidades	As entidades são conectadas, o que permite o fluxo de dados entre os simuladores.	Pode-se utilizar os métodos `connect()`, `connect_many_to_one()` ou `connect_randomly()`.
	5	Execução da simulação	Identifica o início da co-simulação e troca de dados entre as entidades.	Pode-se definir até quanto tempo a co-simulação será executada por meio do parâmetro `until`.
Atualização Mosaik 3.0				
Tipo	**Nº**	**Elemento**	**Descrição**	**Método/Dicionário - API**
Obrigatórios	1	Parâmetro `'type'`	É necessário definir o tipo do simulador entre `'event-based'`, `'time-based'` ou `'hybrid'`.	`Metadados` - Sim API
	2	Parâmetro `'time_resolution'`	Deve-se adicionar o parâmetro no cenário. O seu valor padrão é 1. Ele define a unidade de tempo inteira de simulação do Mosaik.	`Mosaik.start()` - Scenario API
	3	Parâmetro `'max_advance'`	Deve-se adicionar o parâmetro no método `step()` do simulador de integração. Identifica o tempo que um simulador pode avançar no tempo sem ser interrompido.	`step()` - Sim API
Outros	a	Parâmetro `'non-persistent'`	Define atributos de um simulador como "não-persistente", ou seja, só serão válidos para um passo de execução.	`Metadados` - Sim API
	b	Parâmetro `'trigger'`	Define atributos que acionam um passo de execução assim que houver novos dados disponíveis deles.	`Metadados` - Sim API
	c	Retorno opcional no método `step()`	O retorno do método `step()` agora é opcional, a fim de tratar simuladores de tipo discreto (que não possuem necessidade de especificar o próximo tempo de execução).	`step()` - Sim API
	d	Retorno do parâmetro `'time'` no método `get_data()`	Agora, pode ser retornado o parâmetro `'time'` no dicionário de retorno do método `get_data()`. Isso permite identificar execução de passos específicas tratadas em *steps* futuros.	`get_data()` - Sim API
	e	Parâmetro `'weak'`	Utilizado para determinar qual das conexões de simuladores de evento/tempo discretos é conectada primeiro.	`connect()` - Scenario API
	f	Eventos externos	É possível tratar eventos externos por meio do método `set_data()` por meio de comunicação assíncrona.	`set_data()` - Sim API

Fonte: O próprio autor.

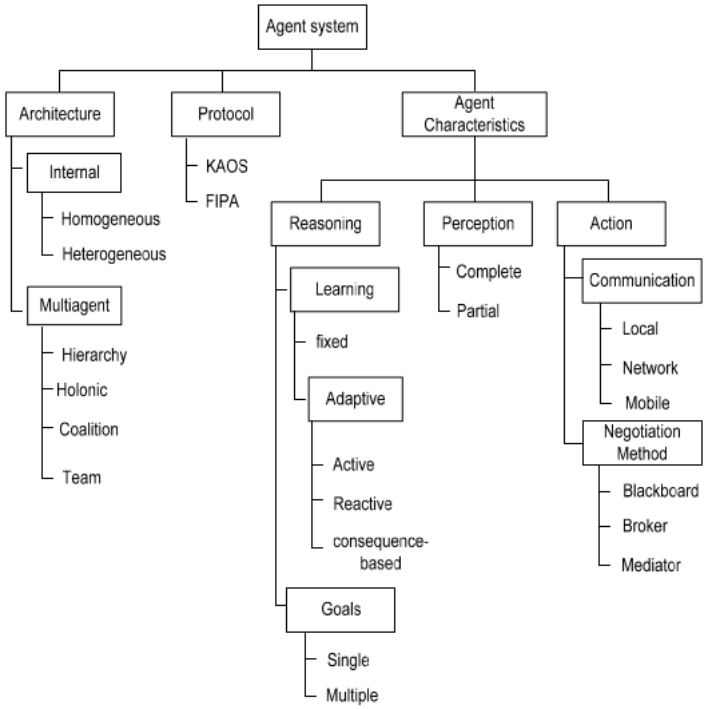
2.2 Sistemas Multiagentes (SMA)

Sistemas Multiagentes (SMA) são, resumidamente, sistemas distribuídos de entidades autônomas que interagem entre si e com seu ambiente para atingir objetivos específicos. A principal vantagem de usar SMA é sua capacidade de adaptação em ambientes dinâmicos para solução de problemas distribuídos (POSLAD, 2007). Devido às suas características modulares, descentralizadas e dinâmicas, o SMA tem sido amplamente aplicado em vários domínios: Redes de Computadores, Robótica, Logística, *Smart Grids*, etc...

Geralmente, uma estrutura SMA é determinada por um *ambiente*, seus *agentes* e suas *interações* que funcionam como uma única *organização* (BOISSIER *et al.*, 2020) (DEMAZEAU, 1995) . Como o SMA é um assunto extenso ainda em desenvolvimento, sua classificação é ampla e pode variar em muitos aspectos, como mostra a Figura 16.

Neste contexto, um *ambiente* é definido como um meio compartilhado que permite aos agentes existirem e realizar suas atividades. É importante destacar que um ambiente não se limita a um meio virtual, pois ele também pode ser considerado físico ou misto - por exemplo, uma rede de distribuição elétrica.

Figura 16 – Classificação de SMA de acordo com (BALAJI; SRINIVASAN, 2010)



Fonte: (BALAJI; SRINIVASAN, 2010).

Agentes têm muitas definições que podem diferir ligeiramente umas das outras.

Embora não haja uma definição única para esse assunto, os agentes podem ser bem definidos como entidades autônomas orientadas a objetivos que reagem a eventos com outros agentes e seu ambiente. Cada agente é autônomo em seu próprio ambiente, ou seja, atua individualmente e não depende, primordialmente, de outras entidades para executar suas tarefas. Assim, eles são adaptativos e responsáveis por sua própria tomada de decisão sem auxílio de usuários humanos. Se os agentes não podem concluir uma tarefa ou objetivo sozinhos, eles podem interagir, comunicar e trocar serviços - cooperativa ou competitivamente - com outros agentes independentes ou com o ambiente para atingir seus próprios objetivos.

Interações entre agentes são a base da dinamicidade e interoperabilidade em um SMA. São protocolos que permitem a comunicação entre os agentes, o SMA e outros serviços ou softwares. Assim, para se comunicar e se entender efetivamente, os agentes SMA devem ter o mesmo protocolo de comunicação e uma ontologia comum que especifiquem as propriedades, características e relações nas quais eles estão inseridos. Algumas linguagens SMA padronizadas que definem interações são a *Agent Communication Language* (ACL) do *Foundation for Intelligent Physical Agents* (FIPA) e a *Knowledge Query and Manipulation Language* (KQML).

Uma *organização* é a abstração de estrutura coletiva de todo o SMA que, conjuntamente, define e atinge o objetivo geral de um sistema. Em outras palavras, a organização é responsável pelas atividades gerais de coordenação e regulação do objetivo principal de um SMA.

Considerando o domínio de *Smart Grids*, um exemplo de aplicação SMA é proposto em (MELO *et al.*, 2019). Um SMA pode ser aplicado em uma rede de distribuição de energia para recompor redes elétricas afetadas por eventos do ambiente externo. Nesse cenário, os agentes seriam responsáveis pela comunicação e tomada de decisão sobre o sistema de restauração, a fim de isolar ou alimentar determinadas partes da rede de maneira otimizada.

2.2.1 Protocolos de comunicação e Interoperabilidade

Normalmente, como explicado em (POSLAD, 2007), os sistemas multiagentes precisam ser suportados por uma infraestrutura de computador distribuída genérica ou um conjunto de serviços de *middleware*, que exige que os agentes não apenas interajam uns com os outros, mas também com serviços heterogêneos não-agentes. Isso destaca a importância da interoperabilidade na comunicação de SMA.

De acordo com (HILPISCH *et al.*, 2009), um protocolo de comunicação é “um

sistema de regras que permite que duas ou mais entidades de um sistema de comunicações transmitam informações por meio de qualquer tipo de variação de uma grandeza física. O protocolo define as regras, sintaxe, semântica, sincronização de comunicação e possíveis métodos de recuperação de erros. Os protocolos podem ser implementados por *hardware*, *software* ou uma combinação de ambos". Ou seja, protocolos de comunicação habilitam a comunicação entre diferentes sistemas, descrevendo toda a estrutura necessária para que haja uma comunicação adequada entre os elementos comunicantes.

Por vezes, empresas criam *softwares* com Propriedade Intelectual (PI) que utilizam sistemas e protocolos de comunicação próprios que impossibilitam a integração de equipamentos, componentes e *softwares* de outras empresas. Essa situação pode gerar problemas na aquisição de dispositivos de marcas distintas, já que, caso não haja tratamento de interoperabilidade para dispositivos diferentes, pode haver impacto no investimento de equipamentos e na atualização de sistemas de legado. Esse problema pode ocorrer até mesmo com equipamentos ou *softwares* de uma mesma marca, caso não haja o tratamento de interoperabilidade entre componentes antigos e novos.

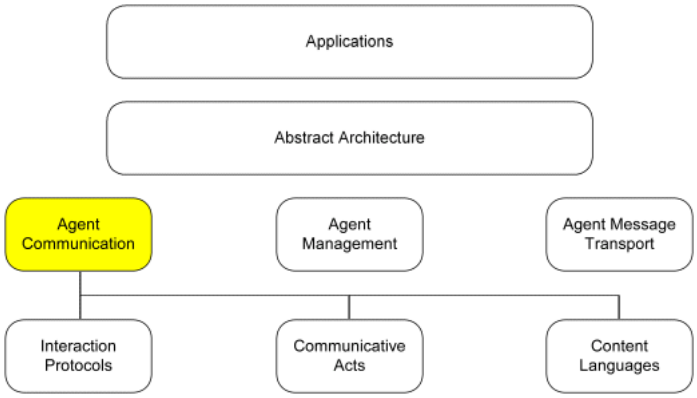
Nesse contexto, a *interoperabilidade* é definida como "a capacidade de dois ou mais componentes de *software* para cooperar apesar de suas diferenças de linguagem, interface e plataforma de execução" (WEGNER, 1996). Portanto, a interoperabilidade define a integração de sistemas, não apenas na troca de dados, mas também na consideração de todos os fatores que permeiam a execução dos componentes que se interconectam, como plataformas de *hardware*, sistemas de *software* e componentes intermediários que possam conectá-los. Para tal, é interessante a existência de normas bem definidas e "globais" que possam permitir a interoperabilidade e prevenir o bloqueio de comunicação devido ao uso exclusivo de protocolos de comunicação com Propriedade Intelectual (PI) que não trabalhem de maneira interoperável.

Diante do apresentado, a organização de padrões *Foundation for Intelligent Physical Agents* (FIPA) foi desenvolvida com foco em permitir a comunicação otimizada no contexto da interoperabilidade em sistemas multiagentes. Considerando que diferentes aplicações de SMA podem utilizar *frameworks* variados para estruturar a comunicação entre seus agentes, as especificações do FIPA propõem protocolos de comunicação padronizados que facilitam e normalizam a troca de informações e serviços entre agentes heterogêneos.

2.2.1.1 *Foundation for Intelligent Physical Agents (FIPA)*

As principais propriedades que definem a estrutura do FIPA são três componentes: *Abstract Architecture* (arquitetura do agente), *Agent Management* (gerenciamento do agente) e *Agent Communication* (comunicação do agente) (DALE, 2005). A Figura 17 indica a estrutura do FIPA e seus diferentes componentes.

Figura 17 – Árvore da estrutura do FIPA.



Fonte: (DALE, 2005)

O componente *Abstract Architecture* do FIPA é uma arquitetura padronizada que abstrai os principais aspectos do FIPA para não perder mecanismos essenciais por meio de revisões incrementais dos elementos centrais do FIPA. Ele permite a interoperabilidade dos *gateways* de transporte ao longo do tempo e especifica em um nível abstrato como os agentes se localizam e se comunicam registrando-se e trocando mensagens. É a arquitetura que define os elementos obrigatórios no desenvolvimento de um sistema baseado no FIPA. A Tabela 3 indica elementos obrigatórios que definem esse componente.

Tabela 3 – Elementos obrigatórios para desenvolvimento de um sistema baseado no FIPA.

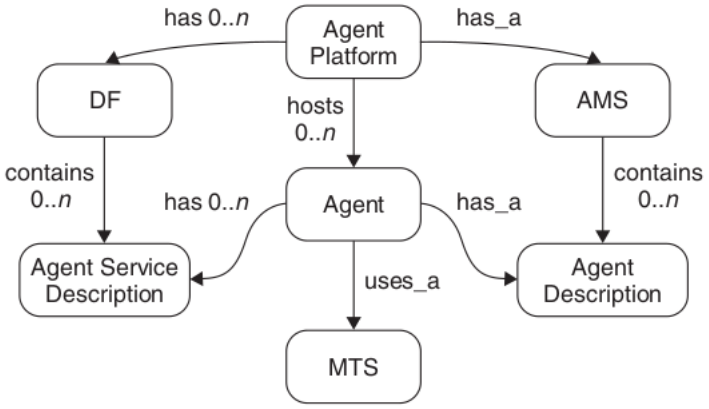
Elemento	**Descrição**
Mensagens de Agentes	A principal maneira de comunicação entre agentes. Baseado na estrutura *key-value* do FIPA-ACL.
Serviço de Transporte de Mensagens	Determina como enviar e receber mensagens de transporte entre agentes.
Diretório de Serviços de Agentes	Um repositório compartilhado de informações no qual os agentes publicam e procuram dados de entrada.
Diretório de Serviços	Um repositório compartilhado no qual agentes podem identificar serviços.

Fonte: Adaptado de (DALE, 2005).

O componente *Agent Management* do FIPA define uma estrutura de referência que permite que os agentes existam, operem e sejam gerenciados. Este modelo é mostrado na

Figura 18 e explicado na Tabela 4.

Figura 18 – Modelo de gestão de agentes FIPA.



Fonte: (BELLIFEMINE *et al.*, 2007).

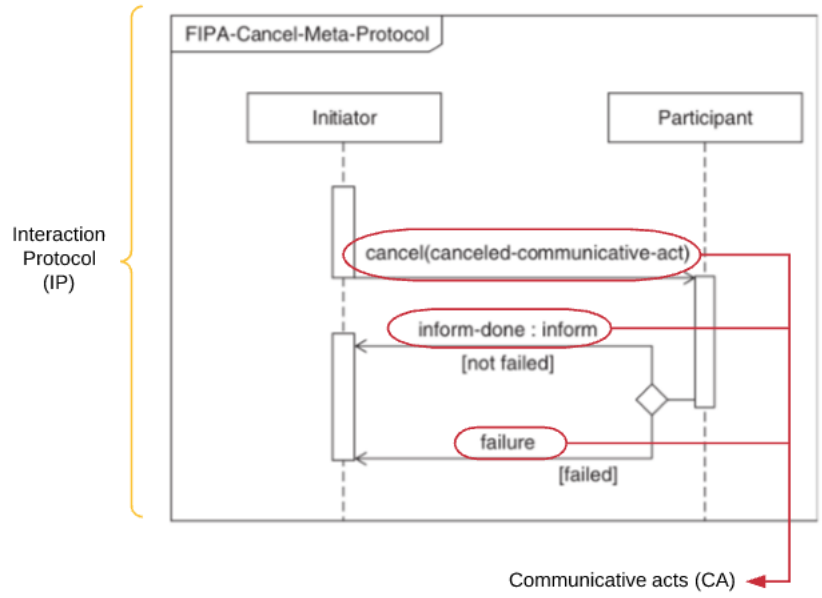
Tabela 4 – Modelo do agente de administração da FIPA.

Elemento	**Descrição**
Agent Platform (AP)	Estrutura ciber-física na qual os agentes são implementados.
Agent	Processo computacional que oferece serviços em uma AP.
Directory Facilitator (DF)	Componente opcional de uma AP que provê informações pertinentes para outros agentes.
Agent Management System (AMS)	Componente AP obrigatório responsável pela gestão de agentes e operações.
Message Transport Service (MTS)	Serviço que transporta mensagens ACL entre agentes.

Fonte: O próprio autor.

Conforme explicado anteriormente, para interagir e se comunicar, um SMA precisa de especificações que possibilitem e detalhem seu intercâmbio e respostas correspondentes. O componente *Agent Communication* do FIPA cobre esse escopo. Para isso, o padrão de ontologia ACL foi proposto pelo FIPA. Baseado na teoria dos atos de fala - que define que todo discurso está totalmente vinculado a uma ação -, o FIPA-ACL possui 22 atos comunicativos e 11 protocolos de interação que elaboram o intercâmbio de mensagens em um sistema multiagente. Os Atos Comunicativos (AC) definem os efeitos/respostas da comunicação entre os agentes emissor e receptor (BELLIFEMINE *et al.*, 2007), enquanto os Protocolos de Interação (PI) definem o processo necessário para alcançar um determinado tipo de comunicação. A Figura 19 mostra um protocolo de interação e os atos comunicativos que estão vinculados a ele. O anexo A indica uma lista de todos os atos comunicativos existentes.

Figura 19 – Exemplo de um Protocolo de Interação e os atos comunicativos inseridos nele.



Fonte: (DALE, 2005)

Em suma, todas as interações possíveis entre os agentes e seus resultados subsequentes são detalhadamente definidos nas especificações ACL do FIPA. Com base na pilha OSI ou TCP/IP, a estrutura de comunicação do FIPA pode ser separada em uma pilha de sete camadas, conforme mostrado na Tabela 5.

Tabela 5 – Camadas de comunicação da FIPA.

Camada	**Tipo**	**Definição**
1	Transporte	Protocolo de transporte, que pode utilizar HTTP, WAP ou IIOP.
2	Codificação	Habilita que mensagens sejam representadas em estruturas de dados de alto-nível, como XML, String, etc.
3	Mensagem	Possui as informações da mensagem: conteúdo remetente, destinatário, tipo da mensagem e respostas *time-out*.
4	Ontologia	Mensagens de referência em aplicações específicas de modelos conceituais ou ontologias.
5	Expressão de conteúdo	Inclui fórmulas lógicas ou operadores algébricos nas mensagens.
6	Atos de comunicação (AC)	Classifica mensagens em termos de ações ou performance.
7	Protocolo de Interação (PI)	Normalmente, mensagens não são isoladas. Essa camada relaciona-se, portanto, com a sequência de interações de comunicação de um intercâmbio de comunicação.

Fonte: (BELLIFEMINE *et al.*, 2007)

Conforme explicado anteriormente, os Protocolos de Interação (PI) são processos que representam o passo-a-passo de um tipo específico de interação em um SMA. Essas interações são representadas por diagramas de sequência *Unified Modeling Language* (UML) que especificam

as trocas entre receptor e emissor. A Figura 20 mostra algumas das especificações de PI mais importantes para o escopo deste projeto. As explicações estão detalhadas abaixo[12]:

- **FIPA Request Interaction Protocol Specification (SC00026):** permite que um agente (iniciador) solicite um serviço ou ação de outro agente (participante). A parte participante decidirá se concorda ou recusa a solicitação requisitada. Se aceito, pode ou não informar o iniciador sobre os detalhes do resultado da ação executada.
- **FIPA Contract Net Interaction Protocol Specification (SC00029):** permite que um agente (iniciador) gerencie tarefas que podem ser executadas por um ou mais agentes (participantes). Ele inicia uma "negociação" entre os agentes participantes que propõem uma solução para o agente iniciador - muito parecido com uma competição para ganhar uma licitação. Essa interação propõe uma comunicação multilateral entre os agentes.
- **FIPA Subscribe Interaction Protocol Specification (SC00035):** este PI faz um agente (iniciador) "seguir" as atualizações de outro agente (participante) em relação a uma ação ou objeto. Quando a ação/objeto inscrito do participante for alterado, o iniciador será informado sobre essas modificações.
- **FIPA Communicative Act Library Specification - Cancel Meta Protocol (SC00037):** em qualquer ponto de uma interação, o *Cancel Meta Protocol* pode ocorrer para indicar que um agente iniciador não deseja mais prosseguir com a interação em andamento. Se efetivamente executada, a interação atual será cancelada.

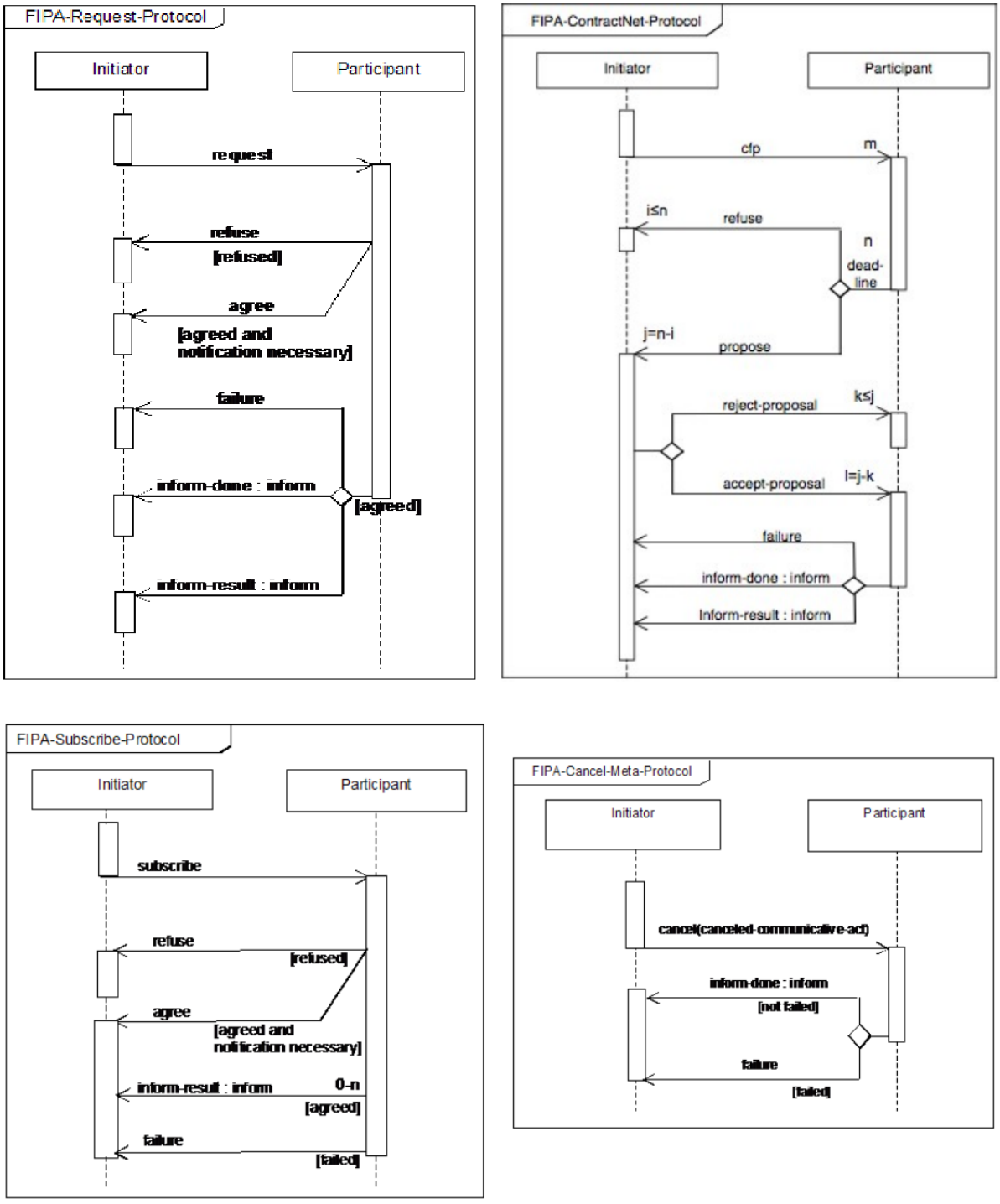
2.2.2 Python Agent DEvelopment (PADE)

No contexto das redes inteligentes modernas, muitas soluções diferentes têm sido propostas para o gerenciamento e análise de sistemas de energia. Considerando sistemas de restauração em redes de distribuição elétrica, o uso de SMA pode recompor de forma eficiente redes elétricas afetadas por eventos do ambiente externo. Pensando nisso, o Python Agent DEvelopment (PADE) (MELO *et al.*, 2019), um *framework open-source*, foi desenvolvido para facilitar a implementação de soluções SMA no domínio de SEP, utilizando a linguagem de programação Python e o modelo FIPA como base de comunicação.

Considerando a certa novidade das aplicações SMA nas últimas décadas, o PADE propõe uma solução para a escassez de plataformas SMA específicas no domínio da Engenharia

[12] Mais informações sobre essas e outras especificações estão disponíveis site oficial do FIPA, disponível no link: <http://www.fipa.org/repository/ips.php3>.

Figura 20 – Alguns exemplos de protocolos de interação do FIPA.



Fonte: (DALE, 2005)

Elétrica. Para isso, o PADE também fornece um *plugin* para integração com o Mosaik.

PADE é baseado na biblioteca Twisted[13]: um mecanismo de rede orientado a eventos escrito em Python. Esta biblioteca permite ao PADE determinar protocolos personalizados,

[13] A documentação oficial do Twisted está disponível em: <https://twistedmatrix.com/trac/wiki/Documentation>.

possibilitando a implementação do padrão FIPA. O PADE é construído com base no protocolo FIPA, sendo capaz de suportar as suas interações mais comuns FIPA: FIPA-Request, FIPA-Contract-Net e FIPA-Subscribe.

2.2.2.1 Exemplo

O Código-fonte 9 mostra um exemplo de execução do PADE e seus principais componentes[14].

```
# agent_example_1.py
# A simple hello agent in PADE!

from pade.misc.utility import display_message, start_loop
from pade.core.agent import Agent
from pade.acl.aid import AID
from sys import argv

class AgenteHelloWorld(Agent):
    def __init__(self, aid):
        super(AgenteHelloWorld, self).__init__(aid=aid)
        display_message(self.aid.localname, 'Hello World!')


if __name__ == '__main__':
    agents_per_process = 3
    c = 0
    agents = list()
    for i in range(agents_per_process):
        port = int(argv[1]) + c
        agent_name = 'agent_hello_{}@localhost:{}'.format(port, port)
        agente_hello = AgenteHelloWorld(AID(name=agent_name))
        agents.append(agente_hello)
        c += 1000

    start_loop(agents)
```

Código-fonte 9 – Estrutura básica de execução do *framework* PADE.

[14] Para acessar o tutorial completo de como o PADE funciona e suas principais ferramentas, consulte sua documentação no link: <https://pade.readthedocs.io/en/latest/>.

2.3 Integração PADE/Mosaik

O PADE possui uma integração Mosaik que permite comunicação de baixo-nível com o *software* de co-simulação. Apesar de PADE ser um *framework* Python, o tempo de execução de PADE e Mosaik são diferentes, o que requer um adaptador em estrutura de baixo-nível entre esses dois elementos. Essa adaptação já está desenvolvida e permite que os agentes PADE sejam reconhecidos no Mosaik simplesmente identificando os métodos equivalentes que normalmente são usados em uma API Mosaik de alto-nível. Isso significa que, simplesmente importando o *driver* de integração do PADE/Mosaik, é possível utilizar métodos equivalentes aos da API Mosaik de alto nível.

A Figura 21 compara as semelhanças entre a API de alto nível do Mosaik e o módulo PADE/Mosaik. Como apresentado, eles funcionam de forma semelhante - a diferença é que, de fato, os métodos que são importados na integração PADE/Mosaik foram todos desenvolvidos em baixo-nível, mas são apresentados como os métodos de alto nível normalmente encontrados no `mosaik-api`.

Figura 21 – Diferenças entre a *interface* de alto-nível do Mosaik API e o *driver* PADE/Mosaik

High-level Mosaik API

```python
import mosaik_api

example_meta = {
    'type': 'time-based',
    'models' {
        'A': {
            'public': True,
            'params': ['init_val'],
            'attrs': ['val_out', 'dummy_out'],
        },
    }
}

class ExampleSim(mosaik_api.Simulator):
    def __init__(self):
        super().__init__(example_meta)

    sim_name = 'ExampleSimulation'

    def init(self, sid):
        # Initializes the simulator
        return self.meta

    def create(self, num, model):
        # Creates entities
        return entities

    def step(self, time, inputs):
        # What will be executed in each step_size
        return time + self.step_size

    # Continue implementation...
```

PADE/Mosaik module

```python
import pade
from pade.drivers.mosaik_driver import MosaikCon
# The driver that integrates Mosaik and PADE in a low-level API

example_meta = {
    'models' {
        'Agent_X': {
            'public': True,
            'params': ['init_val'],
            'attrs': ['val_out'],
        },
    }
}

class AgentSim(MosaikCon):
    def __init__(self, agent):
        super(MosaikSim, self).__init__(example_meta, agent)

    sim_name = 'AgentSimulation'

    def init(self, sid):
        # Initializes the simulator
        return self.meta

    def create(self, num, model):
        # Creates entities
        return entities

    def step(self, time, inputs):
        # What will be executed in each step_size
        return time + self.step_size

    # Continue implementation...
```

Fonte: O próprio autor.

3 ESTUDO DE CASO

Para o estudo de caso, focou-se na análise e implementação da adaptação de sistemas já desenvolvidos com Mosaik para a nova versão 3.0. Para tal, o tutorial oficial Mosaik que aplica as novas funcionalidades da atualização foi estudado em detalhes. Ademais, partindo-se do exemplo de integração PADE/Mosaik desenvolvido por (MELO *et al.*, 2020), foram realizadas alterações no código-fonte a fim de atualizá-lo ao novo Mosaik 3.0.

Portanto, para este estudo de caso foram realizadas as duas análises identificadas a seguir:

- **Estudo de tutorial Mosaik 3.0 -** foram identificados em detalhes o novo tutorial proposto pela documentação Mosaik que identifica as novas funções e ferramentas da atualização Mosaik 3.0;
- **Atualização da demo PADE/Mosaik -** foram adicionados os novos parâmetros e atributos que determinam a atualização do Mosaik 2 para Mosaik 3.0 na integração PADE/Mosaik;

O estudo do tutorial proposto pela documentação oficial do Mosaik 3.0 é explicado em detalhes a seguir. Logo após, o código-fonte original da demonstração PADE/Mosaik e as alterações realizadas para este projeto são investigados.

3.1 Estudo de tutorial Mosaik 3.0

O sistema de co-simulação Mosaik 3.0 propõe um tutorial que exemplifica como realizar integrações em seu sistema, assim como explora as novas funcionalidades de uso de simuladores de eventos discretos. Esta seção explica em detalhes este tutorial, especificando as funções do novo Mosaik 3.0 na prática. O tutorial completo proposto pela nova versão do Mosaik está disponível na documentação oficial Mosaik[1].

Uma versão comentada e executável do tutorial estudado está disponível no repositório Github criado para este projeto[2]. Nas subseções a seguir, o tutorial executável será detalhado, baseando-se em cada um dos seus arquivos principais.

O tutorial em análise é estruturado em seis arquivos, tal qual resumidos abaixo:

- `example_model.py` - possui o modelo implementado pelo tutorial. Este modelo está descrito em um arquivo único e é instanciado diretamente na interface do Sim API do

[1] Disponível em: <https://mosaik.readthedocs.io/en/latest/tutorials/index.html>.

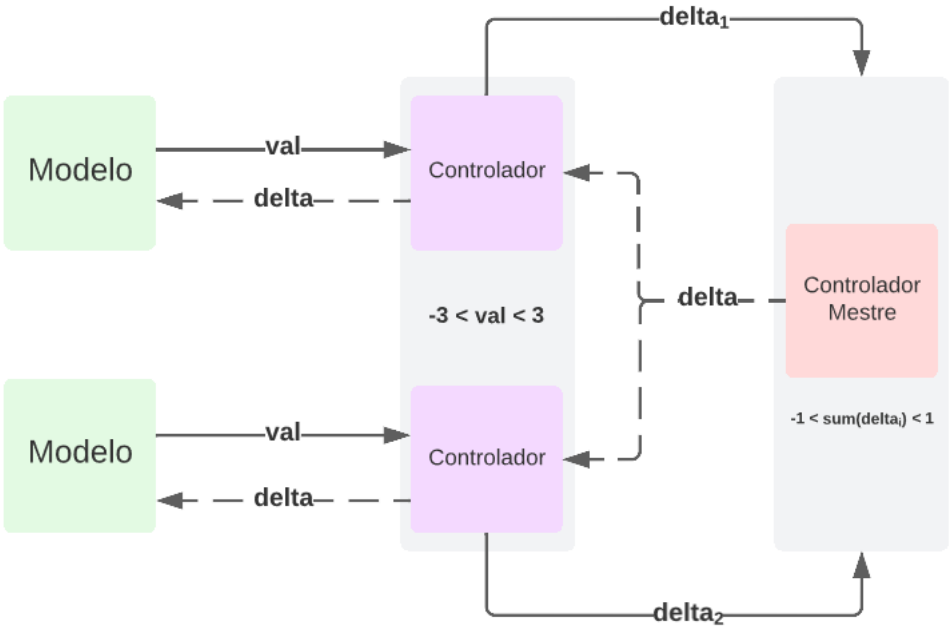
[2] Tutorial disponível em: <https://github.com/grei-ufc/tcc-yana-mosaik-examples/tree/main/tutorials/Mosaik%203/demo3>.

modelo.

- `simulator_mosaik.py` - identifica a interface de comunicação (Sim API) do modelo do tutorial com o Mosaik.
- `controller_demo_3.py` - representa um controlador com o objetivo de limitar os valores de um dos atributos (`val`) do modelo implementado.
- `controller_master.py` - controlador-mestre que gere os valores limitados pelos controladores do tutorial.
- `demo_3.py` - identifica o Scenario API do tutorial, ou seja, o arquivo de execução do Mosaik.
- `collector.py` - printa os valores da simulação na tela ao final da execução da co-simulação Mosaik. Esse arquivo não será analisado em detalhes nesse estudo.

Considerando-se o funcionamento geral do tutorial, o seu objetivo associa-se à capacidade dos simuladores citados comunicarem-se entre si e controlarem os valores do modelo implementado.

Figura 22 – Representação simplificada dos elementos do tutorial.



Fonte: O próprio autor.

A Figura 22 identifica graficamente a estrutura do tutorial, em que instâncias do modelo são conectados a controladores individuais, que, por sua vez, são conectados a um único controlador-mestre. Os modelos possuem dois atributos: `val` e `delta`. A cada iteração, um valor `delta` é adicionado ao valor `val` no modelo. Cada instância do modelo é gerido

por um controlador que limita o valor de `val` para estar sempre no intervalo $[-3,3]$. Todos os controladores são administrados por um controlador-mestre, que define se a soma de todos os `delta` obtidos pelos controladores estão no intervalo $(-1,1)$. Caso a soma dos valores de `delta` não esteja nesse intervalo, os controladores recebem o valor de `delta = 0` para que o intervalo seja respeitado.

O tutorial possibilita a identificação prática das melhorias propostas pela nova versão do Mosaik - nominalmente, o uso dos parâmetros `type`, `time_resolution`, `max_advance` e `weak`. Esses parâmetros, assim como a conexão de tipo *same-time loops*, serão apresentados na prática nos códigos deste estudo. Cada um dos elementos principais que estruturam o tutorial são apresentados em detalhes nas subseções a seguir. Ao final das explicações dos códigos, é identificado como realizar a execução do exemplo.

3.1.1 Modelo/Simulador

O tutorial proposto pelo Mosaik possui por base um modelo simples. Ele possui três elementos principais: o parâmetro `init_val` e os atributos `delta` e `val`. Partindo-se de um valor opcional inicial, `init_val`, adiciona-se, a cada iteração, um valor de `delta` ao valor atual `val`. A Figura 23 apresenta graficamente essa lógica.

Algoritmo 1: Modelo do tutorial Mosaik 3.0

Entrada: $val_0 = init\_val$

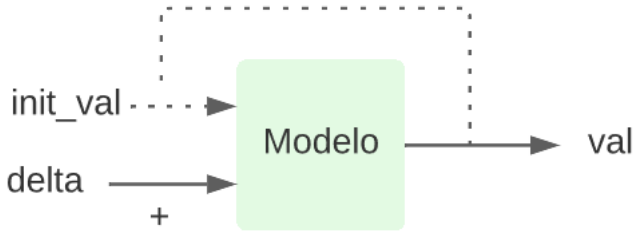
início

$val_i = val_{(i-1)} + delta,\ para\ i \in N, i > 0, delta \in Z$

fim

Saída: *val*

Figura 23 – Representação gráfica do modelo do tutorial Mosaik 3.0.



Fonte: Adaptado de (SCHERFKE, 2018).

Baseando-se nos conceitos da seção 2.1.1.1.1, este tutorial implementa o modelo

apresentado em um arquivo único: `example_model.py`. Na estrutura utilizada, o modelo é implementado sem utilização de um simulador intermediário que o instancie.

O Código-fonte 10 revela o arquivo `example_model.py`, que apresenta o modelo descrito. Já o Código-fonte 11 identifica o arquivo `simulator_mosaik.py`, que representa o Sim API da interface de comunicação do modelo com o Mosaik.

```
1  # example_model.py
2  """
3  Modelo do tutorial. Instanciado diretamente no Sim API.
4  """
5
6  class Model:
7      """Modelo simples que adiciona o valor *val* a um valor *delta* a cada passo de
       execucao.
8      Opcionalmente, eh possivel setar um valor inicial *init_val*. Por padrao, ele eh
       identificado como 0.
9      """
10     def __init__(self, init_val=0):
11         self.val = init_val
12         self.delta = 1
13
14     def step(self):
15         """Adiciona *delta* a *val* em um passo de execucao."""
16         self.val += self.delta
```

Código-fonte 10 – Modelo do tutorial Mosaik 3.0.

É interessante notar alguns pontos associados à atualização do Mosaik no arquivo `simulator_mosaik.py` (Código-fonte 11). Inicialmente, é possível identificar a determinação do simulador como de tipo `time-based` (linha 9). Essa configuração exige, portanto, que o retorno do método `step()` identifique o tempo do próximo passo de execução (*step*), `next_step`, do simulador - ver seção 2.1.2.4.3. Nesse exemplo, o valor de `next_step = time + 1` (linha 60).

A seguir, o simulador identifica uma limitação para o valor do `time_resolution` do simulador (linha 27). Especificamente, é exigido que o tempo de resolução do Mosaik seja igual a "`1.`" (unidade de tempo `float` de execução do Mosaik). O tempo de resolução, como visto anteriormente, identifica como traduzir o tempo de execução para tempo de simulação do Mosaik.

Finalmente, na linha 39 do Código-fonte 11, observa-se que o modelo é instanciado diretamente no Sim API. Logo, a implementação do modelo é realizada tal qual apresentada no

apêndice B.1.

```
# simulator_mosaik.py
"""
Interface de comunicacao (Sim API) do modelo do tutorial com o Mosaik.
"""
import mosaik_api
import example_model

META = {
    'type': 'time-based', # O simulador executa com tempo continuo!
    'models': {
        'ExampleModel': {
            'public': True,
            'params': ['init_val'],
            'attrs': ['delta', 'val'],
        },
    },
}

class ExampleSim(mosaik_api.Simulator):
    def __init__(self):
        super().__init__(META)
        self.eid_prefix = 'Model_'
        self.entities = {}
        self.time = 0

    def init(self, sid, time_resolution, eid_prefix=None):
        if float(time_resolution) != 1.: # Limita o time_resolution do Mosaik para 1.
            raise ValueError('ExampleSim only supports time_resolution=1., but'
                             ' %s was set.' % time_resolution)
        if eid_prefix is not None:
            self.eid_prefix = eid_prefix
        return self.meta

    def create(self, num, model, init_val):
        next_eid = len(self.entities)
        entities = []

        for i in range(next_eid, next_eid + num):
            model_instance = example_model.Model(init_val) # O modelo eh instanciado
    DIRETAMENTE no Sim API!
            eid = '%s%d' % (self.eid_prefix, i)
            self.entities[eid] = model_instance
            entities.append({'eid': eid, 'type': model})

        return entities

    def step(self, time, inputs, max_advance):
        print('[simulator_mosaik] max_advance: {}'.format(max_advance))
```

```
48          self.time = time
49
50          # Checa disponibilidade de novo delta e realiza step para cada instancia de modelo
51          for eid, model_instance in self.entities.items():
52              if eid in inputs:
53                  attrs = inputs[eid]
54                  for attr, values in attrs.items():
55                      new_delta = sum(values.values())
56                  model_instance.delta = new_delta
57
58              model_instance.step()
59
60          return time + 1  # Simuladores de tipo 'time-based' devem retornar um valor de
        next_step!
61
62      def get_data(self, outputs):
63          data = {}
64          for eid, attrs in outputs.items():
65              model = self.entities[eid]
66              data['time'] = self.time
67              data[eid] = {}
68              for attr in attrs:
69                  if attr not in self.meta['models']['ExampleModel']['attrs']:
70                      raise ValueError('Unknown output attribute: %s' % attr)
71
72                  # Obtem model.val ou model.delta:
73                  data[eid][attr] = getattr(model, attr)
74
75          return data
```

Código-fonte 11 – Sim API do modelo do tutorial Mosaik 3.0.

3.1.2 Controlador

O simulador do controlador é responsável por ajustar os valores de `delta` da instância do modelo. O controlador realiza a leitura do atributo `val` de uma instância do modelo e verifica se ele se encontra no intervalo `[-3, 3]`. Caso o valor de `val` supere 3, o valor de `delta = -1`; caso `val` torne-se inferior a -3, então, `delta = 1`. Essa lógica permite, portanto, que `val` sempre esteja no intervalo desejado. O Código-fonte 12 apresenta a estrutura deste simulador, no arquivo `controller.py`.

A seguir, são detalhados as configurações relacionadas à nova atualização Mosaik 3.0 identificadas no código. Inicialmente, é possível identificar que o controlador é definido como

sendo do tipo `event-based`. Isso possibilita que ele seja acionado a partir de eventos, ou seja, dados disponíveis por outro simulador. Considerando-se o tipo `event-based` deste simulador, o retorno do método `step()` é definido como `None` - já que, pela seção 2.1.2.4.3, o retorno do próximo de tempo execução, `next_step`, é opcional para simuladores de tipo `event-based`.

A seguir, é interessante apresentar que o controlador é implementado com o conceito de *same-time loops*, em comunicação com o seu controlador-mestre. Nessa perspectiva, o método `step()` deve ser analisado em detalhes. Como apresentado na seção 2.1.2.3, são necessárias três etapas para a configuração de uma conexão do tipo *same-time loop*. No código apresentado, vemos a fase de "limpeza" dos dados do dicionário de dados (`data`) pelo controlador (linha 49). Isso significa que, caso haja novos dados disponíveis para o atributo `delta` pelo controlador-mestre, então, o controlador receberá esses valores enviados pelo controlador-mestre.

Em seguida, ainda considerando configurações de *same-time loops*, é possível ver que o atributo `time` (linha 80) é identificado no método `get_data()`, assim como é exigido nesse tipo de conexão.

```
1  # controller.py
2  """
3  Um modelo de controle simples.
4  Verifica se o valor *val_in* recebido de uma instancia de modelo esta entre [-3, 3].
5  """
6  import mosaik_api
7
8  META = {
9      'type': 'event-based',  # Necessario para same-time loop
10     'models': {
11         'Agent': {
12             'public': True,
13             'params': [],
14             'attrs': ['val_in', 'delta'],
15         },
16     },
17 }
18
19 class Controller(mosaik_api.Simulator):
20     def __init__(self):
21         super().__init__(META)
22         self.agents = []
23         self.data = {}
24         self.time = 0
25
26     def create(self, num, model):
27         n_agents = len(self.agents)
28         entities = []
```

```
29          for i in range(n_agents, n_agents + num):
30              eid = 'Agent_%d' % i
31              self.agents.append(eid)
32              entities.append({'eid': eid, 'type': model})
33
34          return entities
35
36      def step(self, time, inputs, max_advance):
37          print('[controller_demo_3] max_advance: {}'.format(max_advance))
38          self.time = time
39          data = {}
40          for agent_eid, attrs in inputs.items():
41              '''
42                  Verifica se ha valores disponiveis de *delta* por evento.
43                  Caso haja dados disponiveis pelo controlador-mestre,
44                  o controlador nao calcula o *delta* e apenas recebe
45                  o valor disponivel do mestre
46              '''
47              delta_dict = attrs.get('delta', {})
48              if len(delta_dict) > 0:
49                  data[agent_eid] = {'delta': list(delta_dict.values())[0]}  # Esvazia
    dicionario *data* para same-time loop
50                  continue
51
52              values_dict = attrs.get('val_in', {})
53              if len(values_dict) != 1:
54                  raise RuntimeError('Only one ingoing connection allowed per '
55                                     'agent, but "%s" has %i.'
56                                     % (agent_eid, len(values_dict)))
57              value = list(values_dict.values())[0]
58
59              # Limita os valores de *val* dentro do intervalo [-3, 3]
60              if value >= 3:
61                  delta = -1
62              elif value <= -3:
63                  delta = 1
64              else:
65                  continue
66
67              data[agent_eid] = {'delta': delta}
68
69          self.data = data
70
71          return None
72
73      def get_data(self, outputs):
74          data = {}
75          for agent_eid, attrs in outputs.items():
76              for attr in attrs:
```

```
77                 if attr != 'delta':
78                     raise ValueError('Unknown output attribute "%s"' % attr)
79                 if agent_eid in self.data:
80                     data['time'] = self.time         # Necessario para same-time loop
81                     data.setdefault(agent_eid, {})[attr] = self.data[agent_eid][attr]
82
83         return data
```

Código-fonte 12 – Implementação de controlador do tutorial Mosaik 3.0.

3.1.3 Controlador-Mestre

O controlador-mestre é o elemento responsável por gerir todos os controladores disponíveis no sistema. Como apresentado anteriormente, esse elemento administra os valores de `delta` disponíveis/setados pelos controladores. Para tal, o controlador-mestre realiza a soma de todos os valores de `delta` dos controladores e verifica se essa soma está no intervalo $(-1, 1)$. Caso a soma dos valores de `delta` não pertença a esse intervalo, então, o controlador-mestre seta o valor de `delta = 0` para todos os controladores. O Código-fonte 13 identifica a implementação dessa estrutura lógica.

Tratando-se das configurações do Mosaik 3.0, vemos, novamente que o tipo do simulador é definido como `event-based`. Portanto, o método `step()` do controlador-mestre também não identifica um retorno de próximo tempo de execução - assim como o simulador dos controladores.

Considerando-se a conexão *same-time loops*, é possível observar mais uma das configurações exigidas na seção 2.1.2.3: o retorno do atributo `time` no método `get_data()` do simulador.

```
1  # controller_master.py
2  """
3  Controlador-mestre utilizando same-time loop.
4  Calcula a soma dos valores de *delta* dos controladores e verifica se
5  esta no limite (-1, 1). Caso nao esteja, seta todos os *delta* para 0.
6  """
7  import mosaik_api
8
9  META = {
10     'type': 'event-based',       # Necessario para same-time loop
11     'models': {
12         'Agent': {
13             'public': True,
```

```
14              'params': [],
15              'attrs': ['delta_in', 'delta_out'],
16          },
17      },
18  }
19
20  class Controller(mosaik_api.Simulator):
21      def __init__(self):
22          super().__init__(META)
23          self.agents = []
24          self.data = {}
25          self.cache = {} # Utiliza cache para armazenar valores de *delta* anteriores
26          self.time = 0
27
28      def create(self, num, model):
29          n_agents = len(self.agents)
30          entities = []
31          for i in range(n_agents, n_agents + num):
32              eid = 'Agent_%d' % i
33              self.agents.append(eid)
34              entities.append({'eid': eid, 'type': model})
35
36          return entities
37
38      def step(self, time, inputs, max_advance):
39          print('[controller_master] max_advance: {}'.format(max_advance))
40          self.time = time
41          data = {}
42          for agent_eid, attrs in inputs.items():
43              values_dict = attrs.get('delta_in', {})
44              for key, value in values_dict.items():
45                  self.cache[key] = value
46
47          if sum(self.cache.values()) < -1:
48              data[agent_eid] = {'delta_out': 0}   # Seta *delta* a ser recebido pelos
    controladores
49
50          self.data = data
51
52          return None
53
54      def get_data(self, outputs):
55          data = {}
56          for agent_eid, attrs in outputs.items():
57              for attr in attrs:
58                  if attr != 'delta_out':
59                      raise ValueError('Unknown output attribute "%s"' % attr)
60                  if agent_eid in self.data:
61                      data['time'] = self.time    # Necessario para same-time loop
```

```
62                      data.setdefault(agent_eid, {})[attr] = self.data[agent_eid][attr]
63
64          return data
```

Código-fonte 13 – Implementação de controlador-mestre do tutorial Mosaik 3.0.

3.1.4 Cenário

O Scenario API do tutorial segue a mesma estrutura-base apresentada na seção 2.1.1.1.3. A representação gráfica das conexões realizadas no sistema está identificada na Figura 24. O código do cenário é identificado no Código-fonte 14.

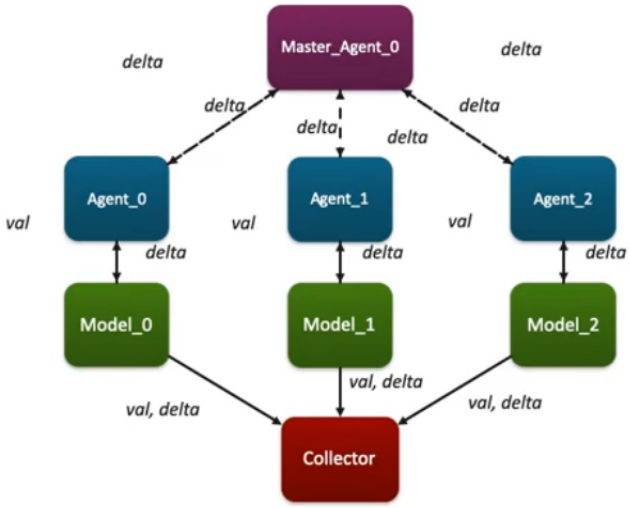
É interessante destacar que a última configuração necessária para funcionamento da conexão de tipo *same-time loops* é identificada na conexão de tipo `weak = True` entre os controladores e o controlador-mestre (linha 45). É durante esse tipo de conexão que o Mosaik compreende que a conexão controladores/controlador-mestre é priorizada em relação à conexão controlador-mestre/controladores (ver seção 2.1.2.4.6). Isso evita uma troca de informações num ciclo (*loop*) infinito.

```
1  # demo_3.py
2  import mosaik
3  import mosaik.util
4
5  # Configuracao do cenario
6  SIM_CONFIG = {
7      'ExampleSim': {
8          'python': 'simulator_mosaik:ExampleSim',
9      },
10     'ExampleCtrl': {
11         'python': 'controller_demo_3:Controller',
12     },
13     'ExampleMasterCtrl': {
14         'python': 'controller_master:Controller',
15     },
16     'Collector': {
17         'cmd': '%(python)s collector.py %(addr)s',
18     },
19 }
20 END = 6  # 10 seconds
21
22 # Criacao do mundo
23 world = mosaik.World(SIM_CONFIG)
24
```

```
25 # Inicializacao dos simuladores
26 examplesim = world.start('ExampleSim', eid_prefix='Model_')
27 examplectrl = world.start('ExampleCtrl')
28 examplemasterctrl = world.start('ExampleMasterCtrl')
29 collector = world.start('Collector')
30
31 # Criacao das instancias dos simuladores
32 models = [examplesim.ExampleModel(init_val=i) for i in (-2, 0, -2)]
33 agents = examplectrl.Agent.create(len(models))
34 print('examplemasterctrl: {}'.format(examplemasterctrl))
35 master_agent = examplemasterctrl.Agent.create(1)
36 monitor = collector.Monitor()
37
38 # Conexao das entidades
39 for model, agent in zip(models, agents):
40     world.connect(model, agent, ('val', 'val_in'))
41     world.connect(agent, model, 'delta', weak=True)
42
43 for agent in agents:
44     world.connect(agent, master_agent[0], ('delta', 'delta_in'))
45     world.connect(master_agent[0], agent, ('delta_out', 'delta'), weak=True) # Necessario
       para same-time-loops
46
47 mosaik.util.connect_many_to_one(world, models, monitor, 'val', 'delta')
48 mosaik.util.connect_many_to_one(world, agents, monitor, 'delta')
49 world.connect(master_agent[0], monitor, 'delta_out')
50
51 # Execucao da simulacao
52 world.run(until=END)
```

Código-fonte 14 – Implementação de Scenario API do tutorial Mosaik 3.0.

Figura 24 – Estrutura de conexão de elementos do tutorial Mosaik 3.0.



Fonte: (OFENLOCH *et al.*, 2022).

3.1.5 Execução e análise

Finalmente, nesta seção, apresenta-se como realizar a execução do exemplo. Além disso, são explicados os resultados da execução e o funcionamento do parâmetro `max_advance`. Um código executável desse tutorial foi construído para este projeto e salvo em repositório Github do GREI[3]. A Figura 25 apresenta o documento `README.md` do repositório citado, que identifica o passo-a-passo para execução do tutorial.

Realizada a execução do tutorial, temos como resultado a tela apresentada no Código-fonte 15. A partir dessa saída, é possível compreender melhor o funcionamento de uma conexão de tipo *same-time loop* e o impacto do parâmetro `max_advance`.

[3] Código disponível em <https://github.com/grei-ufc/tcc-yana-mosaik-examples/tree/main/tutorials/Mosaik%203/demo3>.

Figura 25 – Etapas para execução do tutorial Mosaik 3.0.

README.md

Tutorial Mosaik 3.0

Para execução deste tutorial, é importante ter o pacote `virutalenv` ou `conda` instalados em seu computador para criação de um ambiente virtual isolado. Este tutorial baseará suas etapas na utilização do pacote `conda`.

Para instalar o pacote `conda`, siga o passo-a-passo identificado nesse link.

Abaixo, seguem as etapas para executar adequadamente esse tutorial em sua máquina.

Execução

Com o pacote `conda` corretamente instalado em sua máquina, siga os passos a seguir para criação do ambiente virtual:

1. Crie um novo ambiente, com Python 3.7. Siga as instruções e confirme a criação do ambiente.

```
conda create --name mosaik3 python=3.7
```

2. Acesse o ambiente recém-criado.

```
conda activate mosaik3
```

3. Clone este repositório em uma pasta da sua máquina.

```
git clone https://github.com/yanaspaula/mosaik-examples.git
```

4. Instale o pacote do Mosaik 3.0 no ambiente.

```
pip install mosaik
```

5. Acesse a pasta do tutorial e execute o exemplo.

```
python3 demo_3.py
```

Fonte: O próprio autor.

```
$ python3 demo_3.py
Starting "ExampleSim" as "ExampleSim-0" ...
Starting "ExampleCtrl" as "ExampleCtrl-0" ...
Starting "ExampleMasterCtrl" as "ExampleMasterCtrl-0" ...
Starting "Collector" as "Collector-0" ...
INFO:mosaik_api:Starting Collector ...
----- Parametro max_advance -----
Starting simulation.
[simulator_mosaik] max_advance: 6
[controller_demo_3] max_advance: 0
[simulator_mosaik] max_advance: 6
[controller_demo_3] max_advance: 1
[simulator_mosaik] max_advance: 6
[controller_demo_3] max_advance: 2
[controller_master] max_advance: 2
[simulator_mosaik] max_advance: 6
[controller_demo_3] max_advance: 3
[simulator_mosaik] max_advance: 6
[controller_demo_3] max_advance: 4
```

```
[controller_master] max_advance: 4
[controller_demo_3] max_advance: 4
[controller_master] max_advance: 4
[simulator_mosaik] max_advance: 6
[controller_demo_3] max_advance: 6
[controller_master] max_advance: 6
[controller_demo_3] max_advance: 6
[controller_master] max_advance: 6
Simulation finished successfully.
----- Dados do controller.py -----
Collected data:
- ExampleCtrl-0.Agent_0:
  - delta: {4: 0, 5: 0}
- ExampleCtrl-0.Agent_1:
  - delta: {2: -1, 4: 0, 5: 0}
- ExampleCtrl-0.Agent_2:
  - delta: {4: 0, 5: 0}
- ExampleMasterCtrl-0.Agent_0:
  - delta_out: {4: 0, 5: 0}
- ExampleSim-0.Model_0:
  - delta: {0: 1, 1: 1, 2: 1, 3: 1, 4: 1, 5: 0}
  - val: {0: -1, 1: 0, 2: 1, 3: 2, 4: 3, 5: 3}
- ExampleSim-0.Model_1:
  - delta: {0: 1, 1: 1, 2: 1, 3: -1, 4: -1, 5: 0}
  - val: {0: 1, 1: 2, 2: 3, 3: 2, 4: 1, 5: 1}
- ExampleSim-0.Model_2:
  - delta: {0: 1, 1: 1, 2: 1, 3: 1, 4: 1, 5: 0}
  - val: {0: -1, 1: 0, 2: 1, 3: 2, 4: 3, 5: 3}
```

Código-fonte 15 – Saída da execução do tutorial Mosaik 3.0.

Na seção “Parâmetro max_advance” do Código-fonte 15, é possível ver o funcionamento deste novo parâmetro na prática. Considerando-se a estrutura implementada, sabe-se que a conexão entre controladores e controlador-mestre é definida como sendo do tipo *same-time loops*. Partindo-se dos dados observados na tela, tem-se que o valor do `max_advance` do simulador do modelo, `simulator_mosaik.py`, por ser de tipo `time-based`, equivale ao valor do tempo final final de execução do Mosaik, `END = 6`, tal qual descrito na seção 2.1.2.4.5.

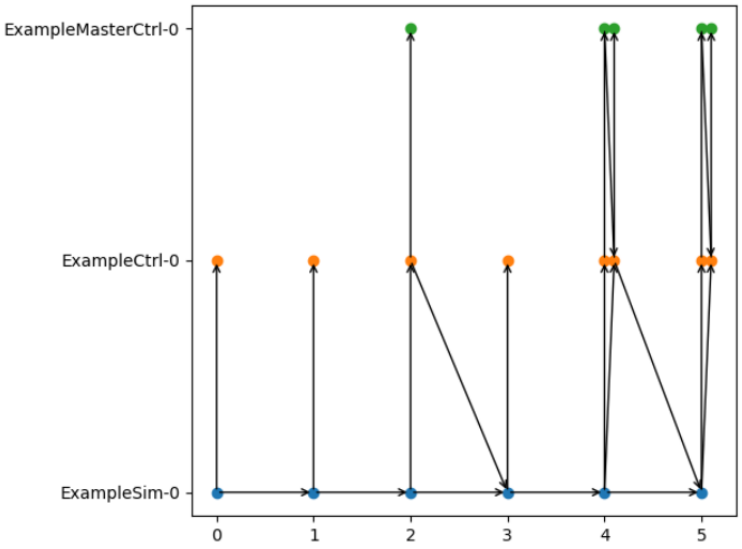
Portanto, o `max_advance` de `simulator_mosaik.py` sempre será igual a 6 - o que permite que o simulador avance, no máximo, até esse valor de *step*. Já os valores de `max_advance` printados a cada `step()` do controlador-mestre inicia em 0 e vai até 6 - que é, justamente, o valor máximo em que ele pode avançar no tempo, determinado pelo fim da execução Mosaik.

Na tela, é possível ver ainda que, a cada passo de execução (*step*), os valores de `delta` dos controladores e os valores de `delta` e `val` das instâncias do modelo. Na seção “Dados do controller.py” é possível observar em qual *step* os valores de `delta` dos controladores são modificados. Para o “Agent_1”, o valor de `delta` é alterado para -1 no tempo 2. A partir do *step* 4, o `delta` de todos os agentes é setado para 0 - ou seja, a lógica de controle do controlador-mestre é acionada.

Nos primeiros *steps*, os controladores não disponibilizam dados de `delta`, pois os valores de `val` de suas instâncias de modelos encontram-se dentro do intervalo $[-3,3]$. A partir do *step* 2, o “Agent_1”, seta o `delta` de seu modelo, `Model_1`, já que ele atinge o valor de `val = 3` e precisa ser limitado. Como “Agent_1” disponibiliza um valor de `delta = -1` - que está fora do intervalo $(-1,1)$, no *step* 4 o controlador-mestre é acionado e realiza a troca de dados com os controladores, a fim de setá-los como 0 - o que é feito, finalmente, no *step* 5 (fim da simulação).

A Figura 26 representa uma representação gráfica das trocas de dados existentes no tutorial.

Figura 26 – Representação gráfico do tutorial Mosaik 3.0.



Fonte: (SCHERFKE, 2018).

3.2 Demonstração PADE/Mosaik

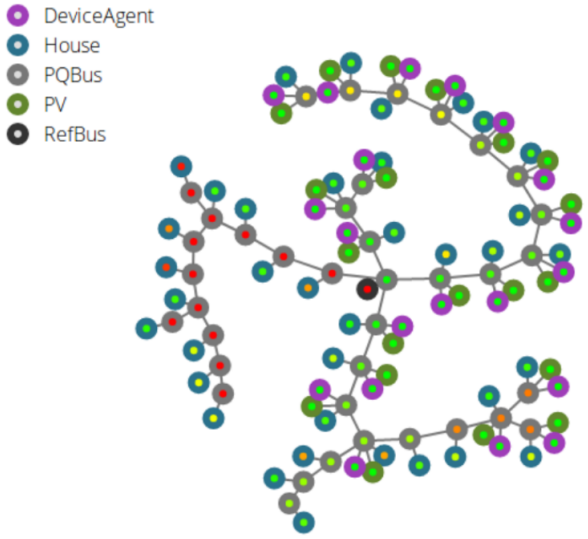
Nesse estudo é analisado se a proposta de adaptação de sistemas que utilizam Mosaik para a nova atualização são, efetivamente, simples, tal qual apresentada no estado de arte deste

trabalho (seção 2.1.2.6). O código utilizado como fonte para este desenvolvimento foi criado para demonstração de uma aplicação do SMA PADE integrado ao sistema de co-simulação Mosaik (MELO *et al.*, 2020).

Utilizando-se de simuladores disponíveis pelos criadores do Mosaik e pelo sistema multi-agente PADE, o código integra adequadamente vários simuladores, que representam uma REI. A representação da REI desta demonstração pode ser vista na Figura 27 por meio de uma interface *web* que representa cada um dos simuladores participantes de maneira visual em nós. Nela, estão representados 5 simuladores: a representação do consumo de uma residência, identificada pelo simulador `House` (em azul); o simulador de placas fotovoltaicas, `PV` (em verde), associadas às residências; agentes PADE, denominados `DeviceAgent` (em roxo), associados às placas fotovoltaicas; o simulador do barramento de conexão, `PQBus` (em cinza), entre os simuladores; e, finalmente, o barramento de referência da rede, `RefBus` (em preto).

Ademais, no centro de cada nó pode-se ver variações de cores que representam o valor da tensão nos pontos apresentados. Nessa perspectiva, as variações de cores entre vermelho-laranja-amarelo-verde representam um espectro que varia entre valores de tensão muito baixos (em vermelho) - o que identificaria uma falha no sistema - até valores de tensão adequados (em verde).

Figura 27 – Representação gráfica da rede de simuladores da demonstração PADE/Mosaik.



Fonte: (MELO *et al.*, 2020)

Resumidamente, na REI simulada da Figura 27, estão dispostas representações de casas associadas a placas fotovoltaicas - que, por sua vez, estão conectadas a agentes PADE. Os agentes PADE, denominados “DeviceAgents” realizam a leitura dos valores de potência ativa

dispostas nas placas fotovoltaicas e aplicam uma função simples de redução dos valores lidos para 10% do seu valor original.

Ao selecionar os elementos dispostos na representação *web* de simulação desta REI, é possível visualizar os valores simulados de cada tipo de simulador em um gráfico visual. A Figura 28 apresenta graficamente os valores obtidos em cada um dos simuladores. O primeiro gráfico, `HouseholdSim-0.House_31`, identifica o consumo simulado de uma residência - simulador `House` (em azul) da Figura 27. O segundo gráfico, `CSV-0.PV_1`, identifica a leitura da potência ativa, P, em *watts* (W), de uma placa fotovoltaica conectada à residência - (simulador `PV` da Figura 27. O terceiro gráfico, `DeviceAgentSimnode_d11-0.DeviceAgent_-node_d11`, identifica a leitura obtida pelo agente PADE associado à placa fotovoltaica - simulador `DeviceAgent` da Figura 27. O agente PADE aplica no valor lido uma função de 10% dos valores lidos pela placa fotovoltaica. Portanto, destaca-se uma diferença de escala entre os valores apresentados do segundo e terceiro gráficos, que identifica, justamente, a função de tratamento implementada pelos agentes PADE. Já o quarto gráfico, `PyPower-0.0-node_a3`, apresenta a leitura de tensão, U, em *volts* (V), do simulador de análise de fluxo - simulador `PQBus` da Figura 27.

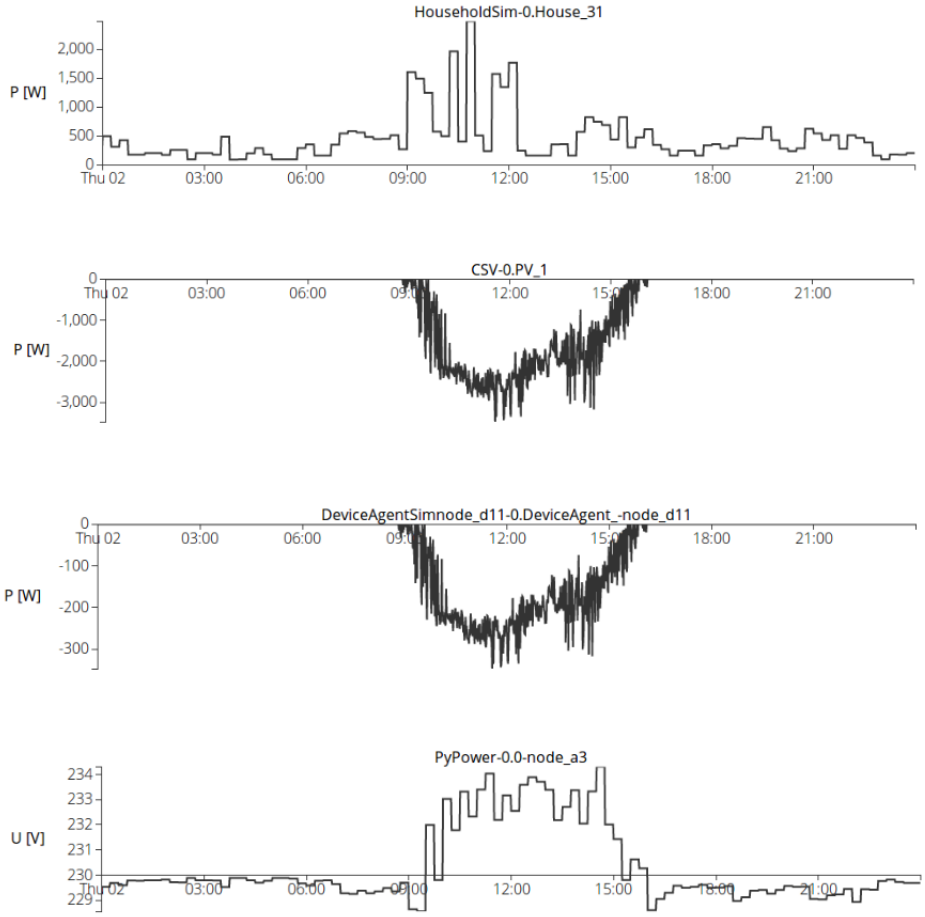
Os principais arquivos analisados e alterados a partir da demonstração original deste projeto estão identificados na Tabela 6. O diretório dos arquivos utilizados estão identificados na Figura 29.

Tabela 6 – Arquivos alterados para a implementação do estudo de caso.

Nome do Arquivo	**Descrição**	**Equivalente Mosaik**
laucher.py	Arquivo de execução da demonstração. Inicializa a execução dos agentes PADE e do arquivo demo.py.	-
demo.py	Define as configurações dos simuladores e do cenário. Conecta os simuladores e inicia o Mosaik.	Cenário Mosaik
device_agent.py	Define os métodos de comunicação do agente PADE “DeviceAgent” com o Mosaik.	Sim API
mosaik_driver.py	Possui a interface de integração dos agentes PADE com o Mosaik.	-

A integração entre os agentes PADE com o sistema de co-simulação Mosaik é desenvolvida, principalmente, nos códigos `mosaik_driver` e `device_agent`. O código `mosaik_driver` está definido na pasta/repositório do PADE e apresenta a estrutura-base de implementação da interface PADE-Mosaik. Já o código `device_agent` herda a interface de `mosaik_driver` e define os métodos da estrutura Sim API do Mosaik - que possui os métodos necessários à comunicação no sistema de co-simulação Mosaik.

Figura 28 – Visualização dos valores dos simuladores da demonstração.



Fonte: (MELO *et al.*, 2020)

O código original da integração PADE/Mosaik foi desenvolvido por (MELO *et al.*, 2020)[4]. Seu código é, por sua vez, uma adaptação da demonstração desenvolvida por (SCHERFKE, 2018)[5].

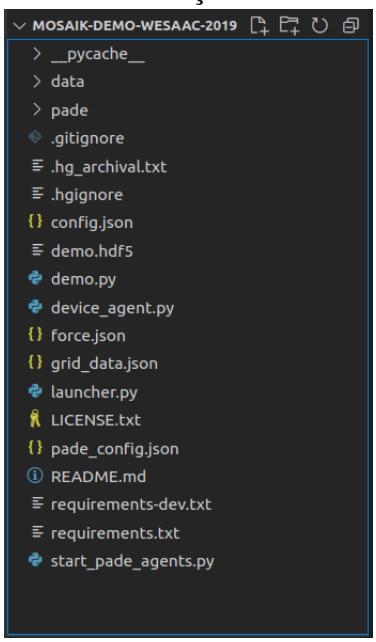
3.2.1 Atualização para Mosaik 3.0

Partindo-se do conteúdo detalhado na seção 2.1.2.4, foram aplicadas as adaptações necessárias à utilização do Mosaik 3.0 na demonstração analisada. Tais alterações são descritas e apresentadas a seguir:

- **Atualização dos requisitos -** o arquivo `requirements.txt`, que identifica as versões de bibliotecas Python necessárias à execução da demonstração PADE/Mosaik, foram atualizadas. Essa alteração foi necessária para que as versões dos pacotes utilizados estejam de acordo com a nova versão Mosaik 3.0. A Figura 30 apresenta a comparação

[4] Código da integração PADE/Mosaik disponível em: <https://github.com/grei-ufc/mosaik-demo-wesaac-2019>.
[5] Código da demonstração Mosaik disponível em: <https://gitlab.com/mosaik/examples/mosaik-demo>.

Figura 29 – Diretório de arquivos da demonstração PADE/Mosaik original.



entre o arquivo original e o atualizado;

- **Adição do parâmetro** `max_advance` **-** foi adicionado o novo parâmetro `max_advance` no método `step()` da definição dos simuladores de agentes PADE para conformidade com a nova versão Mosaik. Para tal, o arquivo `mosaik_driver.py`, pertencente à pasta `pade` teve incluída o tratamento do parâmetro `max_advance`. O parâmetro também foi adicionado no arquivo `device_agentes.py` - já que ele herda os métodos do arquivo de integração `mosaik_driver.py`;
- **Adição do parâmetro** `time_resolution` **-** este parâmetro também foi adicionado para adaptação para o Mosaik 3.0, no método `init()` nos arquivos `mosaik_driver.py` e `device_agentes.py`. Este parâmetro identifica o tempo de resolução do Mosaik para simuladores;
- **Adição do tipo de simulador no dicionário de metadados -** foram aplicados os tipos de simuladores (`time-based` ou `event-based`) nos simuladores utilizados pela demonstração - de acordo com os conceitos propostos pela nova versão do Mosaik.
- **Adição da versão do API do Mosaik 3.0 -** finalmente, nos dicionários de metadados que descrevem os simuladores conectados pelo Mosaik, foi determinada a nova versão do API Mosaik 3 como a necessária à execução dos simuladores. Essa alteração é necessária para identificar qual versão do Mosaik é implementada nos simuladores.

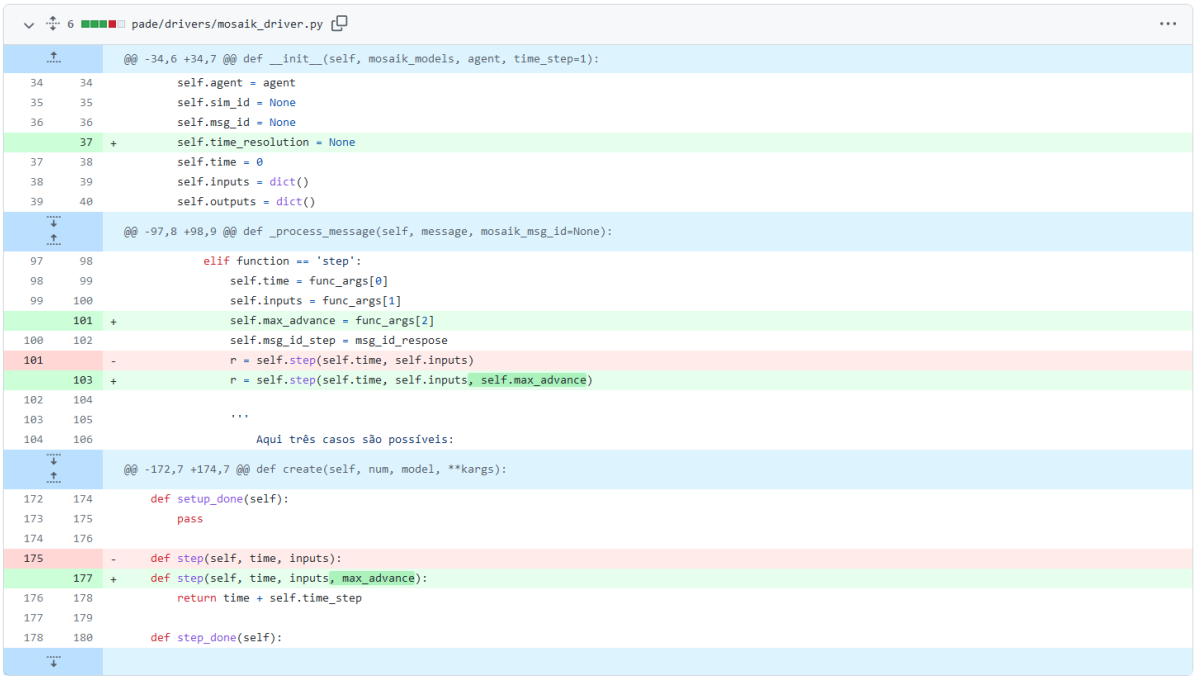
A Figura 31 e 32 identificam as modificações realizadas nos arquivos `mosaik_driver.py` e `device_agents.py`, referente às adições dos parâmetros `time_resolution` e `max_advance`.

Figura 30 – Comparação entre as versões original e adaptada dos arquivos requirements.txt.

Fonte: O próprio autor.

Os códigos do PADE[6] adaptado ao Mosaik 3.0 e da integração PADE/Mosaik[7] atualizada encontram-se no repositório do GREI.

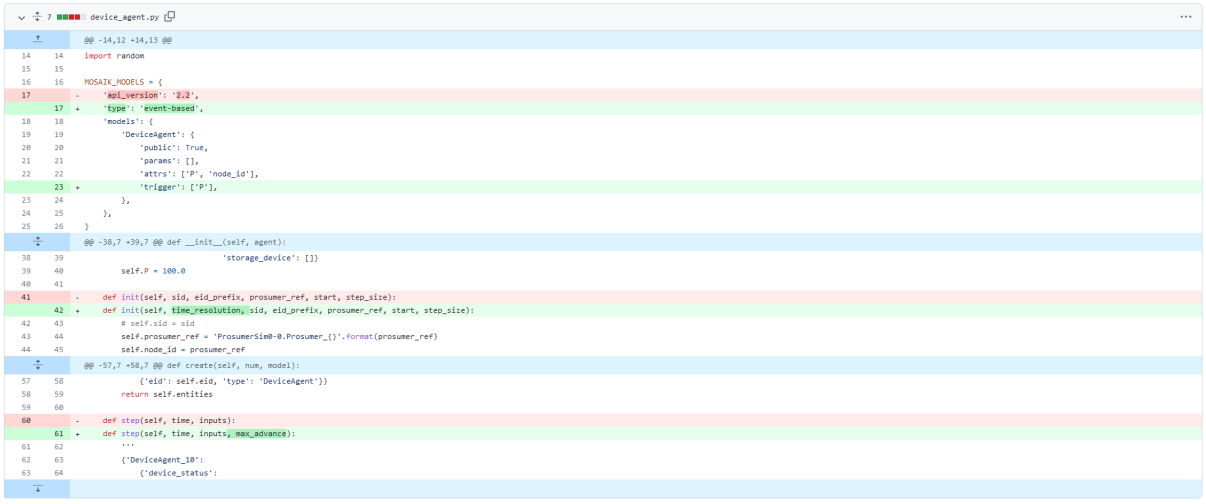
Figura 31 – Alterações realizadas no arquivo mosaik_driver.py para atualização da integração PADE/Mosaik.



[6] PADE atualizado para Mosaik 3.0 disponível em <https://github.com/grei-ufc/pade>.

[7] Integração PADE/Mosaik atualizada para Mosaik 3.0 disponível em <https://github.com/grei-ufc/mosaik-demo-wesaac-2019>.

Figura 32 – Alterações realizadas no arquivo device_agents.py para atualização da integração PADE/Mosaik.



device_agent.py

```
@@ -14,12 +14,13 @@
import random

MOSAIK_MODELS = {
-    'api_version': '2.2',
+    'type': 'event-based',
    'models': {
        'DeviceAgent': {
            'public': True,
            'params': [],
            'attrs': ['P', 'node_id'],
+            'trigger': ['P'],
        },
    },
}
@@ -38,7 +39,7 @@ def __init__(self, agent):
                            'storage_device': []}
        self.P = 100.0

-    def init(self, sid, eid_prefix, prosumer_ref, start, step_size):
+    def init(self, time_resolution, sid, eid_prefix, prosumer_ref, start, step_size):
        # self.sid = sid
        self.prosumer_ref = 'ProsumerSim0-0.Prosumer_{}'.format(prosumer_ref)
        self.node_id = prosumer_ref
@@ -57,7 +58,7 @@ def create(self, num, model):
            {'eid': self.eid, 'type': 'DeviceAgent'})
        return self.entities

-    def step(self, time, inputs):
+    def step(self, time, inputs, max_advance):
        ...
        {'DeviceAgent_10':
            {'device_status':
```

4 RESULTADOS OBTIDOS E CONCLUSÕES

A partir do estudo de caso, pode-se obter considerações importantes em relação à nova atualização do Mosaik 3.0.

4.1 Tutorial Mosaik 3.0

Inicialmente, baseando-se no tutorial Mosaik 3.0, fica evidente a implementação prática dos conteúdos apresentados na fundamentação teórica deste trabalho.

Com o código apresentado em detalhes, foi possível compreender como realizar a implementação dos conteúdos abordados, especificamente, identificando-se o uso dos parâmetros `type`, `time_resolution`, `max_advance` e `weak`. Também foi possível, evidentemente, compreender a estrutura de implementação de conexões do tipo *same-time loop* em implementação prática e detalhada.

Esse material facilita possíveis adaptações, extensões e implementações que tenham a necessidade de utilizar as novas ferramentas propostas pelo Mosaik 3.0.

4.2 Atualização PADE/Mosaik

Considerando-se as alterações realizadas na integração PADE/Mosaik, pode-se afirmar com clareza que as etapas propostas para adaptação de versões antigas para a nova atualização do Mosaik 3.0 são, efetivamente, simples de serem realizadas - tal qual propostas na seção 2.1.2.6. A adaptação do código da demonstração PADE/Mosaik foi realizada com alteração de poucos elementos de uma maneira significativamente fácil.

Esse resultado evidencia a capacidade do Mosaik 3.0 em otimizar sistemas já desenvolvidos com este sistema de co-simulação de uma maneira prática e simples. Levando-se em consideração o significativo aumento de performance proposto pelo Mosaik para as novas ferramentas que tratam eventos/tempos discretos em integração com execuções de tempo contínuo, fica explícita a notável vantagem em utilizar o sistema Mosaik 3.0 em novos sistemas de co-simulação ou em adaptar sistemas que já utilizem o sistema com a nova versão.

4.3 Conclusão e trabalhos futuros

Diante dos conteúdos e resultados apresentados, evidencia-se a pertinência da compreensão e da implementação dos assuntos referentes à nova atualização do sistema de co-simulação Mosaik. Considerando-se, ainda, a sua capacidade de integração com diversos simuladores e, mais especificamente, com Sistemas Multi-agentes (SMA), é possível afirmar um potencial imenso de casos de uso que integrem essas duas ferramentas no domínio de Redes Elétricas Inteligentes (REI).

É possível afirmar que os objetivos deste trabalho foram atingidos, sobretudo, no que refere-se à sua capacidade de dispor conteúdos para a compreensão dos conceitos de sistemas de co-simulação e SMA. Mais precisamente, no que trata-se do sistema de co-simulação Mosaik, é possível afirmar que este trabalho prova-se como uma referência teórica válida e completa para servir como base para trabalhos futuros propostos pelo laboratório Grupo de Redes Elétricas Inteligentes (GREI) e para o corpo discente em geral.

Finalmente, foi possível comprovar a pertinência das novas ferramentas propostas pela versão 3.0 do Mosaik, sobretudo no que refere-se ao seu tratamento de simuladores que possuam configurações de eventos ou tempo discreto. A nova versão do Mosaik possibilita, portanto, um tratamento eficiente de diversas ocorrências de casos reais que possuam a comunicação e conexão de sistemas discretos com sistemas de tempo contínuo. Constatou-se também um ponto crucial na publicação da nova versão Mosaik: a facilidade de adaptação de sistemas já desenvolvidos com a sua versão antiga sem aumento considerável da complexidade necessária a esse ajuste. Essas capacidades do sistema prova sua relevância de aplicabilidade em soluções de REI e sua interessante habilidade de interoperabilidade.

Partindo-se dos conceitos apresentados e dos resultados obtidos, propõe-se para trabalhos futuros uma aplicação de maior complexidade nos códigos apresentados. Baseando-se no conteúdo presente neste trabalho, é possível estender a adaptação implementada na atualização da integração PADE/Mosaik para casos mais complexos e próximos da realidade de um Sistema Elétrico de Potência (SEP), visando solucionar casos de uso semelhantes a condições reais. Nesse contexto, seria interessante a aplicação dos conceitos dispostos neste trabalho em uma integração do sistema PADE/Mosaik com um sistema de comunicação a fim de simular uma representação mais próxima da realidade no que trata-se da comunicação de agentes inteligentes em um SMA. Essa implementação poderia, portanto, representar mais fielmente uma rede de comunicações e o tempo necessário para trocas de informações entre agente em uma SEP.

REFERÊNCIAS

BALAJI, P.; SRINIVASAN, D. An introduction to multi-agent systems. In: **Innovations in multi-agent systems and applications-1**. [S.l.]: Springer, 2010. p. 1–27.

BANDYOPADHYAY, S.; BHATTACHARYA, R. **Discrete and continuous simulation: theory and practice**. [S.l.]: CRC Press, 2014.

BELLIFEMINE, F. L.; CAIRE, G.; GREENWOOD, D. **Developing multi-agent systems with JADE**. [S.l.]: John Wiley & Sons, 2007. v. 7.

BOISSIER, O.; BORDINI, R.; HUBNER, J.; RICCI, A. **Multi-Agent Oriented Programming: Programming Multi-Agent Systems Using JaCaMo**. MIT Press, 2020. (Intelligent Robotics and Autonomous Agents). ISBN 9780262044578. Disponível em: <https://books.google.com.br/books?id=71T6DwAAQBAJ>.

DALE, J. **The Foundation for Intelligent Physical Agents**. 2005. Disponível em: <http://fipa.org/>.

DEMAZEAU, Y. From interactions to collective behaviour in agent-based systems. In: CITESEER. **In: Proceedings of the 1st. European Conference on Cognitive Science. Saint-Malo**. [S.l.], 1995.

GREI-UFC. Pade documentation. **Access in February**, 2019. Disponível em: <https://pade.readthedocs.io/en/latest/>.

HILPISCH, R.; DUCHSCHER, R.; SEEL, M.; KIRK, H. P. S. **Wireless communication protocol**. 2009. Disponível em: <https://worldwide.espacenet.com/patent/search?q=pn\%3DUS7529565>.

JALOTE, P. **A concise introduction to software engineering**. [S.l.]: Springer Science & Business Media, 2008.

MELO, L. S.; SAMPAIO, R. F.; LEÃO, R. P. S.; BARROSO, G. C.; BEZERRA, J. R. Python-based multi-agent platform for application on power grids. **International Transactions on Electrical Energy Systems**, Wiley Online Library, v. 29, n. 6, p. e12012, 2019.

MELO, L. S.; SARAIVA, F.; LEÃO, R.; SAMPAIO, R. F.; BARROSO, G. C. Mosaik and pade: Multiagents and co-simulation for smart grids modeling. **Revista de Informática Teórica e Aplicada**, v. 27, n. 2, p. 107–115, 2020.

OFENLOCH, A.; SCHWARZ, J. S.; TOLK, D.; BRANDT, T.; EILERS, R.; RAMIREZ, R.; RAUB, T.; LEHNHOFF, S. Mosaik 3.0: Combining time-stepped and discrete event simulation. In: IEEE. **2022 Open Source Modelling and Simulation of Energy Systems (OSMSES)**. [S.l.], 2022. p. 1–5.

PALENSKY, P.; KUPZOG, F. Smart grids. **Annual Review of Environment and Resources**, Annual Reviews, v. 38, n. 1, p. 201–226, 2013.

POSLAD, S. Specifying protocols for multi-agent systems interaction. **ACM Transactions on Autonomous and Adaptive Systems (TAAS)**, ACM New York, NY, USA, v. 2, n. 4, p. 15–es, 2007.

SAMPAIO, R. F. Sistema de automação distribuído: uma abordagem baseada em multiagente aplicada a sistemas de distribuição de energia elétrica em média tensão. 2017.

SCHERFKE, S. Mosaik documentation. **Access in February**, 2018. Disponível em: <mosaik.readthedocs.io/>.

STEINBRINK, C.; LEHNHOFF, S.; ROHJANS, S.; STRASSER, T. I.; WIDL, E.; MOYO, C.; LAUSS, G.; LEHFUSS, F.; FASCHANG, M.; PALENSKY, P. *et al.* Simulation-based validation of smart grids–status quo and future research trends. In: SPRINGER. **International conference on industrial applications of holonic and multi-agent systems**. [S.l.], 2017. p. 171–185.

WEGNER, P. Interoperability. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 28, n. 1, p. 285–287, 1996.

APÊNDICE A – MODELO DE INTEGRAÇÃO MOSAIK

O anexo B identifica o código-fonte disponível em <https://github.com/yanaspaula/mosaik-example>. Utilize-os como base para compreender as descrições da seção 2.1.1.1.

Os códigos dispostos no repositório são adaptações dos tutoriais disponibilizados pela documentação oficial do Mosaik, disponível em: <https://mosaik.readthedocs.io/en/latest/tutorials/index.html>.

```
# example_model.py
"""
Esse arquivo possui um exemplo de um modelo simples.
O modelo incrementa seu valor *val* com um valor *delta* a cada passo.
Opcionalmente, voce pode definir o valor inicial no parametro *init_val*,
Por padrao, *init_val* eh definido como 0.
"""

class Model:
    def __init__(self, init_val=0):
        self.val = init_val
        self.delta = 1

    def step(self):
        """Performa um passo de simulacao (step) adicionando *delta* a *val*."""
        self.val += self.delta
```

Código-fonte 16 – Modelo implementado.

```
# simulator_mosaik.py
"""
Componente de interface (Sim API) do Mosaik.
Eh aqui onde os metodos de comunicacao do Mosaik sao definidos para o modelo.
Para uma explicacao detalhada dos metodos do Sim API, acesse: https://mosaik.readthedocs.io
    /en/latest/mosaik-api/low-level.html?highlight=init()#api-calls
"""
import mosaik_api        # Sim API de alto-nivel
import example_model     # Modelo a ser integrado

META = {
    'api_version': '3.0.2',             # Versao utilizada do Mosaik
    'type': 'time-based',               # Como o simulador eh avancado no tempo
    'models': {                         # Modelos definidos nesse arquivo
        'ExampleModel': {
            'public': True,             # Define se um modelo pode ser instanciado pelo
    usuario ou nao
            'params': ['init_val'],     # Parametros que podem ser passados ao modelo em
    sua criacao
```

```
17              'attrs': ['delta', 'val'],  # Atributos que podem ser acessados no modelo (
    leitura ou escrita)
18              # 'any_inputs': True|False, # Se {True}, qualquer atributo pode ser associado a
     esse modelo
19              # 'trigger': ['attr_1', ...]
20          },
21      },
22      # 'extra_methods': [                # Lista opcional de metodos que o simulador pode
    prover em adicao as chamadas API padrao
23      #     'method_example_1',
24      # ]
25  }
26
27
28  class ExampleSim(mosaik_api.Simulator): # Herda classe abstrata do API de alto-nivel
29      def __init__(self):
30          super().__init__(META)
31          self.eid_prefix = 'Model_'
32          self.entities = {}  # Mapeia EIDs (identificacao das entidades) as instancias/
    entidades
33          self.time = 0
34
35      """
36          Inicia a troca de comunicacao entre o simulador e o Mosaik.
37      """
38      def init(self, sid, time_resolution, eid_prefix=None):
39          if float(time_resolution) != 1.:
40              raise ValueError('ExampleSim only supports time_resolution=1, but'
41                               ' %s was set.' % time_resolution)
42          if eid_prefix is not None:
43              self.eid_prefix = eid_prefix
44          return self.meta
45
46      """
47          Cria instancias de um simulador.
48      """
49      def create(self, num, model, init_val):
50          next_eid = len(self.entities)
51          entities = []
52          for i in range(next_eid, next_eid + num):
53              model_instance = example_model.Model(init_val)
54              eid = '%s%d' % (self.eid_prefix, i)
55              self.entities[eid] = model_instance
56              entities.append({'eid': eid, 'type': model})
57          return entities
58
59      """
60          Marca um processo de simulacao (definido pelo tempo de um simulador) no Mosaik.
61          Eh neste metodo que calculos, funcoes ou outras acoes do simulador sao definidas e
```

```
    executadas.
    """
    def step(self, time, inputs, max_advance):
        self.time = time

        # Check for new delta and do step for each model instance:
        for eid, model_instance in self.entities.items():
            if eid in inputs:
                attrs = inputs[eid]
                for attr, values in attrs.items():
                    new_delta = sum(values.values())
                model_instance.delta = new_delta
            model_instance.step()
        return time + 1  # Step size is 1 second

    """
        Este metodo pode ser usado pelo Mosaik para acessar dados de um simulador e
    disponibiliza-los
        para outros simuladores.
    """
    def get_data(self, outputs):
        data = {}
        for eid, attrs in outputs.items():
            model = self.entities[eid]
            data['time'] = self.time
            data[eid] = {}
            for attr in attrs:
                if attr not in self.meta['models']['ExampleModel']['attrs']:
                    raise ValueError('Unknown output attribute: %s' % attr)

                # Get model.val or model.delta:
                data[eid][attr] = getattr(model, attr)
        return data

def main():
    return mosaik_api.start_simulation(ExampleSim())

if __name__ == '__main__':
    main()
```

Código-fonte 17 – Interface de comunicação - Sim API.

```
# demo.py
'''
Representa o Scenario API do Mosaik.
'''
import mosaik
import mosaik.util
```

```
7
8  # Configuracao dos modelos
9  SIM_CONFIG = {  # Determina os modelos utilizados no cenario
10     'ExampleSim': {
11         'python': 'simulator_mosaik:ExampleSim',
12     },
13     'Collector': {
14         'cmd': '%(python)s collector.py %(addr)s',
15     },
16 }
17 END = 10  # 10 seconds
18
19 # Criacao do mundo
20 world = mosaik.World(SIM_CONFIG)
21
22 # Inicializacao dos modelos
23 examplesim = world.start('ExampleSim', eid_prefix='Model_')
24 collector = world.start('Collector')
25
26 # Criacao das instancias
27 model = examplesim.ExampleModel(init_val=2)
28 monitor = collector.Monitor()
29
30 # Conexao das entidades
31 world.connect(model, monitor, 'val', 'delta')
32 more_models = examplesim.ExampleModel.create(2, init_val=3)
33 mosaik.util.connect_many_to_one(world, more_models, monitor, 'val', 'delta')
34
35 # Execucao da simulacao
36 world.run(until=END)
```

Código-fonte 18 – Ambiente de execução Mosaik - Scenario API.

APÊNDICE B – IMPLEMENTAÇÃO SIMULADORES/MODELOS NO MOSAIK

Os códigos-fonte apresentados nessa seção são versões adaptadas do tutorial Mosaik 2 disponível em <https://mosaik.readthedocs.io/en/2.6.1/tutorials/index.html>.

B.1 Modelo instanciado

Os códigos-fonte apresentados a seguir identificam a opção de implementar diretamente um modelo na interface de comunicação Mosaik (Sim API). Como pode ser observado, no arquivo `sim_api.py` (Sim API), o modelo é instanciado diretamente a partir do arquivo do modelo (`model.py`)

```python
# model.py

class Model:
    def __init__(self, init_val=0):
        self.val = init_val
        self.delta = 1

    def step(self):
        self.val += self.delta
```

Código-fonte 19 – Modelo - Implementação direta.

```python
# sim_api.py
import mosaik_api
import example_model    # Arquivo que possui o modelo a ser integrado

META = {
    'api_version': '3.0.2',
    'type': 'time-based',
    'models': {
        'ExampleModel': {
            'public': True,
            'params': ['init_val'],
            'attrs': ['delta', 'val'],
            # 'any_inputs': True|False,
            # 'trigger': ['attr_1', ...]
        },
    }
}


class ExampleSim(mosaik_api.Simulator):
```

```
21      def __init__(self):
22          super().__init__(META)
23          self.eid_prefix = 'Model_'
24          self.entities = {}
25          self.time = 0
26
27      def init(self, sid, time_resolution, eid_prefix=None):
28          if float(time_resolution) != 1.:
29              raise ValueError('ExampleSim only supports time_resolution=1, but'
30                               ' %s was set.' % time_resolution)
31          if eid_prefix is not None:
32              self.eid_prefix = eid_prefix
33          return self.meta
34
35      def create(self, num, model, init_val):
36          next_eid = len(self.entities)
37          entities = []
38          for i in range(next_eid, next_eid + num):
39              model_instance = example_model.Model(init_val) # O modelo esta sendo
        diretamente instanciado no Sim API
40              eid = '%s%d' % (self.eid_prefix, i)
41              self.entities[eid] = model_instance
42              entities.append({'eid': eid, 'type': model})
43          return entities
44
45      def step(self, time, inputs, max_advance):
46          self.time = time
47          for eid, model_instance in self.entities.items():
48              if eid in inputs:
49                  attrs = inputs[eid]
50                  for attr, values in attrs.items():
51                      new_delta = sum(values.values())
52                  model_instance.delta = new_delta
53              model_instance.step()
54          return time + 1
55
56      def get_data(self, outputs):
57          data = {}
58          for eid, attrs in outputs.items():
59              model = self.entities[eid]
60              data['time'] = self.time
61              data[eid] = {}
62              for attr in attrs:
63                  if attr not in self.meta['models']['ExampleModel']['attrs']:
64                      raise ValueError('Unknown output attribute: %s' % attr)
65                  data[eid][attr] = getattr(model, attr)
66          return data
```

Código-fonte 20 – Modelo - Interface de comunicação.

B.2 Simulador instanciado

Na segunda opção, uma classe de integração é construída para instanciar o modelo em análise. Nos códigos abaixo, a classe `simulator.py` possui a classe `Model` que determina as características do modelo e uma outra classe, chamada `Simulator` que instancia o modelo em questão. No arquivo `sim_api2.py`, a classe instanciada é `Simulator`.

```
1  # simulator.py
2
3  class Model:
4      def __init__(self, init_val=0):
5          self.val = init_val
6          self.delta = 1
7
8      def step(self):
9          self.val += self.delta
10
11
12 class Simulator(object):
13     # Simula um determinado numero de modelos "Model" e coleta alguns dados
14     def __init__(self):
15         self.models = []
16         self.data = []
17
18     def add_model(self, init_val):
19         # Cria uma instancia de "Model" com o valor inicial *init_val*
20         model = Model(init_val)
21         self.models.append(model)
22         self.data.append([])
23
24     def step(self, deltas=None):
25         """ Defina novas entradas de modelo de *deltas* para os modelos e execute uma etapa
       de simulacao.
26
27         *deltas* eh um dicionario que mapeia indices de modelo para novos valores delta
       para o modelo.
28
29         """
30         if deltas:
31             # Define novos deltas para instancias do modelo
32             for idx, delta in deltas.items():
33                 self.models[idx].delta = delta
34
35         # Da step no modelo e obtem dados
36         for i, model in enumerate(self.models):
37             model.step()
38             self.data[i].append(model.val)
```

Código-fonte 21 – Simulador - Implementação indireta.

```python
# sim_api2.py
import mosaik_api
import simulator

META = {
    'models': {
        'ExampleModel': {
            'public': True,
            'params': ['init_val'],
            'attrs': ['delta', 'val'],
        },
    },
}

class ExampleSim(mosaik_api.Simulator):
    def __init__(self):
        super().__init__(META)
        self.simulator = simulator.Simulator()  # classe Simulator eh instanciada
        self.eid_prefix = 'Model_'
        self.entities = {}

    def init(self, sid, eid_prefix=None):
        if eid_prefix is not None:
            self.eid_prefix = eid_prefix
        return self.meta

    def create(self, num, model, init_val):
        next_eid = len(self.entities)
        entities = []
        for i in range(next_eid, next_eid + num):
            eid = '%s%d' % (self.eid_prefix, i)
            self.simulator.add_model(init_val)
            self.entities[eid] = i
            entities.append({'eid': eid, 'type': model})
        return entities

    def step(self, time, inputs):
        deltas = {}
        for eid, attrs in inputs.items():
            for attr, values in attrs.items():
                model_idx = self.entities[eid]
                new_delta = sum(values.values())
                deltas[model_idx] = new_delta
        self.simulator.step(deltas)
```

```
45          return time + 60
46
47      def get_data(self, outputs):
48          models = self.simulator.models
49          data = {}
50          for eid, attrs in outputs.items():
51              model_idx = self.entities[eid]
52              data[eid] = {}
53              for attr in attrs:
54                  if attr not in self.meta['models']['ExampleModel']['attrs']:
55                      raise ValueError('Unknown output attribute: %s' % attr)
56                  data[eid][attr] = getattr(models[model_idx], attr)
57          return data
```

Código-fonte 22 – Simulador - Interface de comunicação.

APÊNDICE C – COMPARATIVO DE ARQUIVOS - TUTORIAIS MOSAIK

Este apêndice apresenta um comparativo dos arquivos dos tutoriais Mosaik disponíveis em sua documentação oficial. Os tutoriais da versão 2 do Mosaik estão disponíveis nesse link: <https://mosaik.readthedocs.io/en/2.6.1/tutorials/index.html>. Já os tutoriais da versão 3 do Mosaik encontram-se nesse endereço: <https://mosaik.readthedocs.io/en/latest/tutorials/index.html>.

No repositório <https://github.com/yanaspaula/mosaik-example/tree/main/tutorials> pode-se acessar de maneira fácil as versões executáveis dos tutoriais oficiais. Nos códigos também estão identificados alguns comentários relacionando as diferenças entre os arquivos nas duas versões do Mosaik.

Figura 33 – Comparação dos arquivos do tutorial 1 das versões 2 e 3 do Mosaik.

					Tutoriais Mosaik	
					Versão	
Arquivo	**Estrutura**	**Tipo**	**Classe**	**Método**	**Mosaik 2.0**	**Mosaik 3.0**
collector.py	Simulador	Event-based	Collector	META	-	Adicionado tipo de simulador: 'type': 'event-based'
				__init__()	Utiliza 'self.step_size = None' no método __init__()	Retirado atributo 'self.step_size = None' no método __init__()
				init()	-	Adicionado atributo 'time_resolution' no método init()
					-	Retirado atributo 'self.step_size = step_size' no método __init__()
				create()	-	-
				step()	Método step() retorna 'time + self.step_size'	Adicionado parâmetro 'max_advance' em método 'step()'
					-	Simulador não retorna nada no step(): 'return None'
				finalize()	-	-
example_model.py	Modelo	-	Model	__init__()	-	-
				step()	-	-
simulator.py			Simulator	__init()	-	**Não existe**
				add_model()	-	**Não existe**
				step()	-	**Não existe**
simulator_mosaik.py	Simulador	Time-based	ExampleSim(mosaik_api.Simulator)	META	-	Adicionado tipo de simulador: 'type': 'time-based'
				__init__()	Instanciamento do modelo realizado em 'self.simulator = simulator.Simulator()'	Adicionado atributo 'self.time() = 0'
				init()	-	Adicionado parâmetro 'time_resolution'
					-	Adicionado tratamento do novo parâmetro 'time_resolution' no método 'init()'
				create()	Utilização do método 'add_model()' para criação de instância do modelo	Instancia modelo diretamente no método 'create()'
				step()	-	Adição de parâmetro 'max_advance' no método 'step()'
					-	Adicionado atributo 'self.time = time'
				get_data()	-	Adicionado 'data['time'] = self.time' no método 'get_data()'
demo_1.py	Scenario API	-	-	-	-	world.start()' do Collector não possui mais 'step_size'

LEGENDA	
Tipo de atualização - Mosaik 3.0:	type
	time_resolution
	max_advance

Fonte: O próprio autor.

Figura 34 – Comparação dos arquivos do tutorial 2 das versões 2 e 3 do Mosaik.

					Tutoriais Mosaik	
					Versão	
Arquivo	Estrutura	Tipo	Classe	Método	Mosaik 2.0	Mosaik 3.0
collector.py	Simulador	Event-based	Collector	META	-	Adicionado tipo de execução 'type': 'event-based' nos metadados.
				__init__()	Utiliza 'self.step_size = None' no método __init__()	Retirado o atributo 'self.step_size' do método __init__().
				init()	-	Retirado atributo 'self.step_size = None' no método __init__()
					-	Adicionado atributo 'time_resolution' no método init()
				create()	-	-
				step()	Método step() retorna 'time + self.step_size'	Adicionado parâmetro 'max_advance' em método 'step()'
					-	Simulador não retorna nada no step(): 'return None'
				finalize()	-	-
controller.py	Simulador	Event-based	Controller	META	-	Adicionado tipo de execução 'type': 'event-based' nos metadados.
				__init__()		Identificação do dicionário de dados (self.data = {})no método __init__()
						Tempo inicializado em 0 por meio do atributo 'self.time = 0'
				create()	-	-
				step()	-	Adicionado parâmetro 'max_advance' em método 'step()'
					-	Especificação do atributo 'self.time = time'
					Método step() retorna tempo: 'return time + 60'	Método ste() retorna dados: 'self.data = data'
				get_data()	Não existe	Adicionado método get_data() para retorno dos dados do controlador.
example_model.py	Modelo	-	Model	__init__()	-	-
				step()	-	-
simulator.py			Simulator	__init()	-	-
				add_model()	-	-
				step()	-	-
simulator_mosaik.py	Simulador	Time-based	ExampleSim(mosaik_api.Simulator)	META	-	Adicionado tipo de simulador: 'type': 'time-based'
				__init__()	Instanciamento do modelo realizado em 'self.simulator = simulator.Simulator()'	Adicionado atributo 'self.time() = 0'
				init()	-	Adicionado parâmetro 'time_resolution'
					-	Adicionado tratamento do novo parâmetro 'time_resolution' no método 'init()'
				create()	Utilização do método 'add_model()' para criação de instância do modelo	Instancia modelo diretamente no método 'create()'
				step()	-	Adição de parâmetro 'max_advance' no método 'step()'
					-	Adicionado atributo 'self.time = time'
				get_data()	-	Adicionado 'data['time'] = self.time' no método 'get_data()'
demo_2.py	Scenario API	-	-	-	-	Novo parâmetro "weak" utilizado no método connect()

LEGENDA	
Tipo de atualização - Mosaik 3.0:	type
	time_resolution
	max_advance

Fonte: O próprio autor.

ANEXO A – ATOS DE COMUNICAÇÃO FIPA

O anexo A apresenta um resumo dos tipos de comunicação FIPA (Communicative Acts - CA) existentes.

ANEXO A - FIPA Communicative Acts (CA)

FIPA communicative act	Description
`Accept Proposal`	The action of accepting a previously submitted proposal to perform an action
`Agree`	The action of agreeing to perform some action, possibly in the future
`Cancel`	The action of one agent informing another agent that the first agent no longer has the intention that the second agent performs some action
`Call for Proposal`	The action of calling for proposals to perform a given action
`Confirm`	The sender informs the receiver that a given proposition is true, where the receiver is known to be uncertain about the proposition
`Disconfirm`	The sender informs the receiver that a given proposition is false, where the receiver is known to believe, or believe it likely that, the proposition is true
`Failure`	The action of telling another agent that an action was attempted but the attempt failed
`Inform`	The sender informs the receiver that a given proposition is true
`Inform If`	A macro action for the agent of the action to inform the recipient whether or not a proposition is true
`Inform Ref`	A macro action allowing the sender to inform the receiver of some object believed by the sender to correspond to a specific descriptor, for example a name
`Not Understood`	The sender of the act (for example, *i*) informs the receiver (for example, *j*) that it perceived that *j* performed some action, but that *i* did not understand what *j* just did. A particular common case is that *i* tells *j* that *i* did not understand the message that *j* has just sent to *i*
`Propagate`	The sender intends that the receiver treat the embedded message as sent directly to the receiver, and wants the receiver to identify the agents denoted by the given descriptor and send the received `propagate` message to them
`Propose`	The action of submitting a proposal to perform a certain action, given certain preconditions
`Proxy`	The sender wants the receiver to select target agents denoted by a given description and to send an embedded message to them
`Query If`	The action of asking another agent whether or not a given proposition is true
`Query Ref`	The action of asking another agent for the object referred to by a referential expression
`Refuse`	The action of refusing to perform a given action, and explaining the reason for the refusal
`Reject Proposal`	The action of rejecting a proposal to perform some action during a negotiation
`Request`	The sender requests the receiver to perform some action One important class of uses of the request act is to request the receiver to perform another communicative act
`Request When`	The sender wants the receiver to perform some action when some given proposition becomes true
`Request Whenever`	The sender wants the receiver to perform some action as soon as some proposition becomes true and thereafter each time the proposition becomes true again
`Subscribe`	The act of requesting a persistent intention to notify the sender of the value of a reference, and to notify again whenever the object identified by the reference changes