Universidade Federal de Campina Grande - UFCG
Centro de Engenharia Elétrica e Informática - CEEI
Unidade Academica de Engenharia Elétrica - UAEE

Matheus Vilarim Pereira dos Santos

Estudo e Aplicação da ferramenta de simulação FERAL

Campina Grande, Brasil
19 de outubro de 2021

Matheus Vilarim Pereira dos Santos

Estudo e Aplicação da ferramenta de simulação FERAL

Trabalho de Conclusão de Curso submetido à Coordenação de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande, Campus Campina Grande, como parte dos requisitos necessários para obtenção do título de Graduado em Engenharia Elétrica.

Áreas de Concentração: Simulação de Sistemas

Orientador: Prof. Dr. Edmar Candeia Gurjão

Campina Grande, Brasil
19 de outubro de 2021

Matheus Vilarim Pereira dos Santos

Estudo e Aplicação da ferramenta de simulação FERAL

Trabalho de Conclusão de Curso submetido à Coordenação de Graduação em Engenharia Elétrica da Universidade Federal de Campina Grande, Campus Campina Grande, como parte dos requisitos necessários para obtenção do título de Graduado em Engenharia Elétrica.

Aprovado em: 19/10/2021

Prof. Dr. Edmar Candeia Gurjão
Orientador

Prof. Dr. Gutemberg Gonçalves dos Santos Júnior
Avaliador

Campina Grande, Brasil
19 de outubro de 2021

Dedico esse trabalho àqueles que me deram tudo, meus pais.

Agradecimentos

Agradeço a Deus por ter me dado forças pra chegar até aqui. Agradeço aos meus pais, Veridenes Vilarim e Moisés Vilarim, por terem sido exemplos de amor, força e determinação. Muito do que sou é por causa de vocês. Assim, os dedico essa conquista.

Sou grato especialmente a minha irmã, Mayara Vilarim, que diante de todas as adversidades, me deu todo suporte, oportunidade e paciência que eu precisava para trilhar todo o caminho que me trouxe até aqui. Devo muito a você.

A minha família, entre quais destaco minhas avós, Josefa dos Santos e Maria Margarida. Ao meu avô Carlos Antônio. A estes devo todo carinho e cuidado que recebi. Agradeço a minha namorada, Maria Luana, que ao longo da confecção desse trabalho me deu todo apoio e incentivo, estando comigo nos melhores e piores dias.

Aos amigos que o curso me deu: Natan dos Santos, Taís Lima, Fabrícia Paola e Camila Machado. Obrigado pelas ajudas nas disciplinas, conversas e risadas. e os ensinamentos que vão além dos muros da universidade. Deixo também meu obrigado a todas as outras pessoas que cruzaram meu caminho ao longo do curso e me ajudaram a chegar até aqui.

Por fim, mas não menos importante, agradeço ao professor orientador Edmar Candeia Gurjão, pela paciência, compreensão, palavras de apoio e sábios ensinamentos para comigo em todas as vezes que nos encontramos no percurso da graduação e pela orientação ao longo do trabalho.

"Mude, mas comece devagar, porque a direção é mais importante que a velocidade."

Clarisse Lispector

Resumo

Devido o aumento na demanda de desenvolvimento de sistemas complexos, a virtualização dos processos e a crescente busca pela redução de custos e tempo de projeto, áreas como a engenharia virtual emergiram. Para atender a busca da indústria por uma aceleração no processo de desenvolvimento de produtos, foram concebidos sofwares que promovem ambiente de testes e prototipagem virtuais. Assim, as soluções construídas podem ser implementadas e colocadas a prova desde as primeiras etapas da cadeia de desenvolvimento. Nesse cenário, o presente trabalho tem como objetivo fazer um estudo sobre uma ferramenta com grande potencial de aplicabilidade e real capacidade de colaboração na indústria e pesquisa, o FERAL. Dessa forma, fez-se uma análise da arquitetura e componentes da ferramenta, destacando-se sua estrutura de funcionamento. Dele, conclui-se que o FERAL explora um campo pouco enfatizado por outros softwares, que é a integração entre diferentes plataformas de simulação. Destaca-se ainda que a cooperação UFCG/ Fraunhofer IESE, está desenvolvendo um módulo de 5G para a ferramenta, o que ampliará ainda mais sua área de aplicação.

Palavras-chaves: Virtualização, engenharia virtual, FERAL, modelagem e simulação.

Abstract

Due to the increasing demand for the development of complex systems, the virtualization of processes, and the growing search for cost and project time reduction, areas such as virtual engineering have emerged. To meet the industry's search for acceleration in the product development process, software was designed to promote virtual testing and prototyping environments. Thus, the built solutions can be implemented and tested from the first stages of the development chain. In this scenario, the present work aims to carry out a study on a framework with great applicability potential and a real capacity for collaboration in industry and research, FERAL. Thus an analysis of the tool's architecture and components was carried out, highlighting its operating structure. From it, it's concluded that FERAL has a promising future as a modeling and simulation tool. Since it explores a field little emphasized by other software, which is the integration between different simulation platforms. It is also noteworthy that the UFCG/Fraunhofer IESE cooperation is developing a 5G module for the tool, which will further expand its application area.

Key-words: Virtualization, virtual engineering, FERAL, modeling and simulation.

Lista de ilustrações

Lista de abreviaturas e siglas

FERAL	Focused Evaluation on Requirements and Architecture Level
IESE	Institute For Experimental Software Engineering
UFCG	Universidade Federal de Campina Grande
HiL	Hardware in the Loop
vHiL	Virtual Hardware in the Loop
CAN	Controller Area Network
MOCC	Model of Computation and Communication
SuT	System under Test
ECU	Electronic Control Unit
DT	Discrete Time
DE	Descrete Event
CT	Continuous Time
ABS	Anti-lock Braking System
USB	Universal Serial Bus
RPM	Rotações por minuto
ISO	International Organization for Standardization
OSI	Open System Interconnection
ID	Identity number
RPM	Rotações por minuto
RPM	Rotações por minuto
TT	Time Triggered
FMU	Functional Mock-Up Unit
MAC	Media Access Control

Sumário

1 Introdução

Atualmente, estão surgindo novos ecossistemas digitais que combinam produtos e processos existentes por meio de software e dados.

Um problema nesse cenário, é a integração de componentes e sistemas, que ocorre constantemente e em alguns casos precise ser realizada em tempo de execução. Em particular, os sistemas que tomam decisões para a segurança de forma automatizada e dos quais, em casos extremos, vidas humanas estão envolvidas, devem ser validados e testados de uma forma exaustiva. Isso inclui, por exemplo, plataformas automotivas que são capazes de receber e integrar dinamicamente as funções de dirigibilidade.

Com a Indústria 4.0, a produção industrial está enfrentando problemas quando se trata de integrar ativamente novos dispositivos e processos em uma planta existente. Na área das comunicações, a implantação da nova geração de tecnologias para redes móveis (5G) também representa um desafio, visto que o 5G fará uso de boa parte da atual estrutura existente.

A solução FERAL do Fraunhofer IESE é uma plataforma de simulação com vários componentes. Eles permitem a integração de cenários complexos e heterogêneos em um ambiente de teste, além da verificação sistemática de propriedades em um espaço virtual protegido com a ajuda de gêmeos digitais. A solução cria protótipos virtuais por meio do acoplamento de modelos e simuladores, código existente e plataformas de hardware virtual. Isso permite que o impacto das decisões seja revisado em um estágio inicial, dessa forma, reduzindo custos e economizando tempo.

A Universidade Federal de Campina Grande estabeleceu parceria com o Fraunhofer-IESE e recebeu a licença de uso do FERAL na instituição.

1.1 Objetivo

1.1.1 Objetivo Geral

Estudar o funcionamento da ferramenta FERAL do Fraunhofer IESE, para implementação de um modelo de aplicação, compreendendo sua estrutura de comunicação com outros softwares, o que possibilitará a composição de um ambiente de desenvolvimento de engenharia virtual completo.

1.1.2 Objetivos Específicos

- Contextualizar o uso da ferramenta FERAL;
- Contextualizar sua aplicabilidade na indústria;
- Descrever as principais vantagens que vêm com o uso do FERAL;
- Expor os principais aspectos de funcionamento e a arquitetura do software;
- Apresentar os problemas principais da tecnologia e o que está sendo pesquisado atualmente.

1.2 Estrutura do Trabalho

O capítulo 2 é reservado para explicação de conceitos necessários para o entendimento e análise da ferramenta de simulação, como *Hardware in the Loop*, *Virtual Hardware in the Loop* e modelo de computação e comunicação. Também são abordados conceitos relacionados à redes CAN, os quais são fundamentais para a compreensão do exemplo de aplicação do FERAL.

No capítulo 3 é feita uma explicação sobre os conceitos gerais presentes no núcleo do FERAL, destancando também a estrutura que permite a rápida integração dos modelos de simulação.

No capítulo 4 há uma exposição mais aprofundada da arquitetura do FERAL. Primeiro, é explicado o núcleo da ferramenta. Logo em seguida, é mostrada a hierarquia dos componentes e interface.

No capítulo 5 é apresentada uma aplicação, onde vê-se um exemplo de simulação com o software.

No capítulo 6 estão as conclusões.

2 Fundamentação Teórica

2.1 Modelagem e Simulação

Modelos e simulações são representações simplificadas de objetos, sistemas ou fenômenos mais complexos.

Além de permitir a experimentação quando os testes do mundo real são proibitivos, modelos e simulações baseados em computação muitas vezes permitem experimentações muito mais rápidas e mais baratas. O tempo necessário para simulações é impactado pelo nível de detalhe e qualidade dos modelos utilizados (1). Por exemplo, modelar a propagação de uma doença por uma simulação de computador, a qual mostra os humanos como pontos se movendo aleatoriamente na tela e espalhando doenças se um ponto doente chegar a 5 pixels de um ponto não-doente, é uma simulação muito rápida porque usa um modelo muito simplificado comparado com um que reflete o comportamento humano real (comer, dormir, usar veículos, elevadores, etc). Felizmente, como os modelos são feitos em software, testes rápidos e extensivos de modelos permitem que eles sejam alterados para refletir melhor os objetos e fenômenos que estão sendo modelados simplesmente reprogramando o software.

A palavra simulação é derivada do Latim simulare , que significa fingir. A simulação de um sistema é a execução do modelado elaborado. Nesse processo, o modelo matemático pode ser reconfigurado e experimentado com as mudanças que influenciarão o estado da saída (2).

Nesse contexto, o FERAL é uma plataforma para a rápida criação de ferramentas de simulação e avaliação virtual. Seu núcleo foi projetado para acoplar simuladores e modelos de simulação com diferentes modelos de computação e comunicação (MOCC).

Dessa forma podemos qualificar o FERAL como uma plataforma de simulação e engenharia virtual. Essa plataforma se destina a realizar soluções de simulação personalizadas, por exemplo, para avaliação de desempenho de barramento, simulação holística de sistemas ou teste de qualidade em ambiente virtual. A ferramenta é realizada como biblioteca com componentes que são instanciados e conectados através de um script de simulação. Esta abordagem é comparável ao SystemC, que também é realizado como biblioteca que é instanciada através de um compilador C.

O FERAL está escrito em Java. Isso reduz o código de baixo nível que precisa ser produzido e, por outro lado, baseia-se em uma linguagem de programação limpa e moderna que atingiu um desempenho comparável aos compiladores C++ modernos.

2.1.1 Engenharia Virtual

O desenvolvimento industrial atual é desafiado por uma complexidade crescente de requisitos de produto e processo, enquanto a redução drástica do tempo de colocação no mercado é vista como um dos principais fatores competitivos (3). Recentemente, estratégias estão sendo estabelecidas para melhorar significativamente o processo geral de desenvolvimento usando menos testes em construções físicas em favor de várias verificações de montagem, diagnósticos, simulação e análise de risco em modelos digitais. Apesar da prática bem-sucedida da engenharia digital, há evidências de que colocar sistemas avançados de TI em processos de desenvolvimento não pode, por si só, levar a uma mudança quantitativa real para gerenciar a complexidade e atingir um nível decisivamente novo de desempenho de processo. Como “os problemas significativos que enfrentamos não podem ser resolvidos no mesmo nível de pensamento que tínhamos quando os criamos” (Albert Einstein), uma nova metodologia de engenharia é necessária, compreendendo melhorias tecnológicas e de negócios significativas em produtos, processos e serviços.

Isso leva à ideia de engenharia virtual, que se refere a uma gama de atividades científicas, tecnológicas, organizacionais e de negócios usando ferramentas e métodos avançados de informação e comunicação com foco principal na integração de processos e sistemas, visualização imersiva e interação "homem-máquina". Em particular, a engenharia virtual significa que as atividades de projeto e validação ocorrem de forma colaborativa, a fim de testar protótipos de produtos, apoiar a tomada de decisões e permitir a otimização contínua do produto em parcerias interdisciplinares e entre empresas. Isso causa uma importante redefinição do processo geral de desenvolvimento do produto para apoiar a coordenação, avaliação e concretização dos resultados de engenharia de todos os parceiros envolvidos com o suporte de construções virtuais. As visões da engenharia virtual são a integração de poderosos sistemas de informação e ferramentas em diferentes tarefas, desde a geração e gerenciamento de dados até o compartilhamento e comunicação de dados (visão do sistema), rede de processos e atividades ao longo de todo o ciclo de vida do produto, permitindo validação e otimização contínua do produto (visão do processo).

Embora ainda haja uma grande quantidade de trabalho de pesquisa e desenvolvimento a ser feito, o desenvolvimento industrial já está fazendo uso de conquistas recentes em engenharia de sistemas, gerenciamento de ciclo de vida de produto, tecnologias de realidade virtual, bem como descobertas sobre comunicação entre empresas e culturas diferentes a fim de perceber os benefícios da engenharia virtual em curto prazo e se posicionar para estender esses benefícios em um futuro próximo.

2.2 Hardware in the Loop

O conceito de *Hardware in the Loop* consiste de usar um modelo de simulação do processo em conjunto com o hardware real (4). Em HiL, a planta e os sensores de realimentação são simulados enquanto as leis de controle e processamento de sinal (como filtros) são executados no hardware alvo, seja um controlador ou um PC executando leis de controle em tempo real (5).

2.2.1 Virtual Hardware in the Loop

Como apresentado, o teste de HiL consiste em um ambiente físico que se assemelha ao contexto do sistema. Normalmente esta é uma mistura de hardware real e simulações em tempo real que fornecem um ambiente de teste em loop fechado para o sistema em teste (SuT). O SuT é integrado como dispositivo físico que consiste tanto no hardware alvo quanto no software.

O FERAL foi desenvolvido para permitir testes virtuais de HiL (vHiL). O teste virtual de HiL substitui componentes do mundo real com simulações. Pode, portanto, ser usado muito antes em processos de desenvolvimento, e pode refletir o ambiente de testes físicos com diferentes níveis de precisão. Portanto, permite testes básicos de arquitetura em estágios iniciais de desenvolvimento, onde as decisões de arquitetura são tomadas. Ao simular os impactos dessas decisões, os projetistas podem reunir evidências que aumentam a confiança nos conceitos da arquitetura. Além disso, testes básicos de integração podem ser realizados em estágios iniciais de desenvolvimento que previnem defeitos e erros em estágios posteriores de desenvolvimento. Além disso, é muito mais barato replicar ambientes vHiL; desta forma, o precioso tempo de HiL pode ser substituído por testes vHiL que melhoram os testes do produto.

A Figura 1 ilustra uma configuração comum de teste vHiL. Um sistema em teste (SuT) consiste em uma comunicação simulada de barramento, um conjunto de unidades de controle eletrônico simuladas (ECUs) e uma simulação de barramento de teste que cobre todos os componentes do sistema que estão conectados ao barramento no sistema do mundo real, mas não explicitamente modelados no sistema virtual. Outras instâncias do sistema que são possivelmente simuladas em níveis muito mais baixos de detalhes são conectadas através de uma rede WiFi.

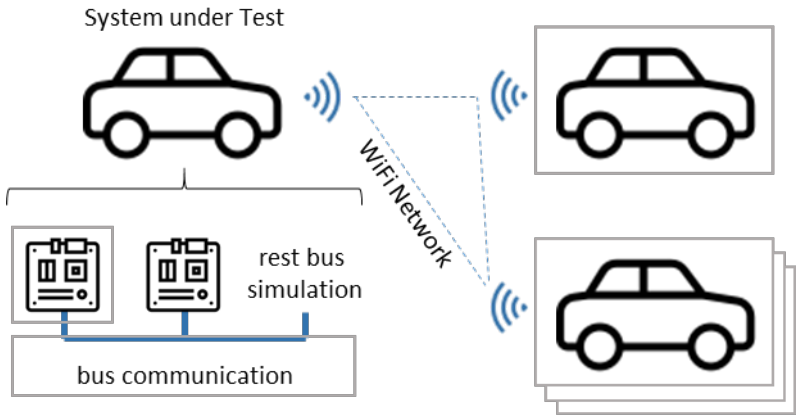

Figura 1 – Exemplo de ambiente vHil. Fonte: Documentação FERAL.

A Figura 2 ilustra uma visão de componente no ambiente vHiL da Figura 1. O SuT é simulado por uma simulação detalhada do sistema que contém modelos de simulação explícitos para ECUs selecionadas, bem como um barramento simulado da CAN (*Controller Area Network*). Uma simulação de rede detalhada conecta um sistema adicional, enquanto uma simulação de rede conecta simulações adicionais do sistema.

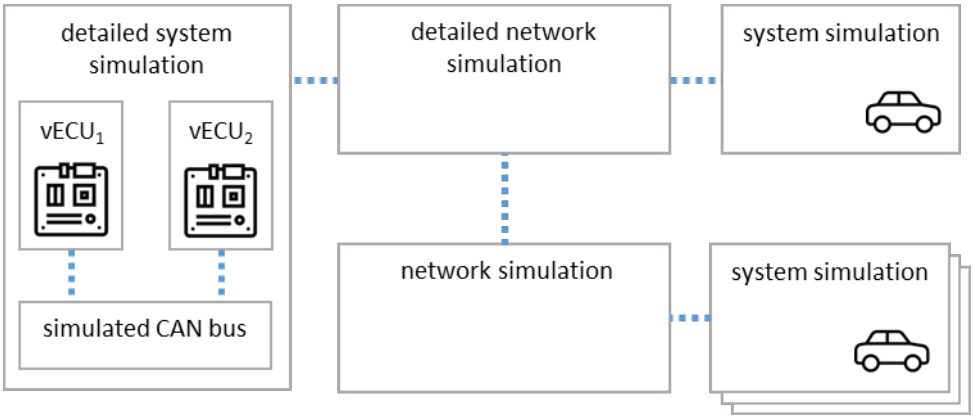

Figura 2 – Componentos o exemplo do ambiente vHil. Fonte: Documentação FERAL.

Essa é uma ilustração de um problema básico de vHiL e simulações complexas de sistema e arquitetura em geral. Os simuladores são especializados, e representam apenas uma fração do sistema do mundo real. Um simulador de comunicação, por exemplo, implementa um modelo de simulação para um tipo de rede, mas não fornece modelos de simulação para outros aspectos do sistema.

O FERAL foi projetada como uma plataforma para permitir o rápido desenvolvimento de soluções de simulação sob medida. Ele define um núcleo que suporta o acoplamento de modelos de simulação, bem como modelos de simulação que permitem o desenvolvimento de protótipos de sistemas virtuais em diferentes níveis de abstração.

Especialmente ao projetar ecossistemas inteligentes, sistemas que participarão com ecossistemas inteligentes e sistemas abertos e adaptativos, as plataformas HiL existentes

não serão mais suficientes como plataformas de desenvolvimento e teste. A virtualização desses complexos sistemas do mundo real é um dos principais objetivos do FERAL.

2.3 Modelo de computação e comunicação

Simuladores e modelos de simulação funcionam em conformidade com seus Modelos de Computação e Comunicação. Estes controlam o comportamento e a semântica da comunicação. MOCCs comuns para simulação incluem tempo discreto, evento discreto e semântica de tempo contínuo (Figura 3).

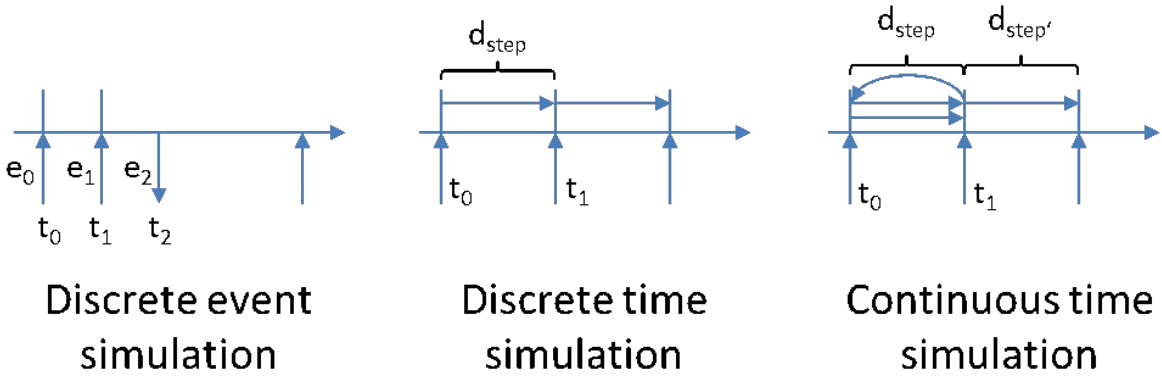

Figura 3 – Modelos de computação e comunicação. Fonte: Documentação FERAL.

Os MOCCs de tempo discreto (DT) dividem a execução em etapas de tempo discreto de tamanho constante ou variável. Eles suportam, por exemplo, a implementação de modelos físicos discretizados e algoritmos de controle. As portas de entrada sob controle de um DT MOCC armazenam um valor que pode mudar entre as etapas de tempo, mas que é mantido constante durante as etapas de tempo de uma simulação. As portas de saída comunicam os resultados da simulação no final de uma etapa de tempo. O MOCC executa componentes controlados em qualquer ordem. A duração de uma etapa de tempo reflete a granularidade da simulação; etapas de tempo mais curtas resultam em maior precisão, mas também exigem cálculos mais frequentes e mais tempo de computação. Etapas de tempo maiores requerem menos cálculos, mas geram um erro de discretização maior.

Modelos de simulação de eventos discretos (DE) implementam a comunicação baseada em eventos e o processamento de eventos de simulação. Os eventos podem ocorrer a qualquer momento e não estão limitados a intervalos de tempo. Sua execução é ordenada com base nos tempos de expiração. Isso garante que o tempo de simulação em modelos de simulação discretos aumente continuamente, mas requer sobrecarga para a classificação necessária do evento. Os componentes de simulação recebem eventos por meio de suas portas de entrada. Modelos de simulação de eventos discretos são usados, por exemplo, para a simulação de conceitos de software e para sistemas de comunicação.

A simulação de tempo contínuo (CT) se aproxima do comportamento contínuo da física do mundo real. Mudanças em elementos individuais têm impactos imediatos nos elementos dependentes. Por exemplo, quando uma mola é estendida, ela imediatamente aplica força em ambas as extremidades. Assemelhar-se a esse comportamento em uma simulação é complicado, pois a discretização necessária da simulação antes de resolvê-la resulta em um erro de simulação. Cada simulação de CT é controlada por um solucionador que avalia um componente de simulação após o outro em intervalos de tempo discretos. Depois de simular um componente, ele copia os valores de saída para as entradas de componentes dependentes. Os solucionadores controlam os erros de simulação iterando a mesma etapa de simulação até que o erro de simulação caia abaixo de um limite aceitável.

2.3.1 Acoplamento de modelos de computação e comunicação

O FERAL foi desenvolvido com o objetivo de unir modelos de simulação individuais e focados em uma simulação holística. O núcleo do FERAL, portanto, se concentra em fornecer uma estrutura para implementar e acoplar Modelos de Computação e Comunicação (MOCC) que definem como uma simulação é executada e como ela se comunica. A abordagem de acoplamento MOCC é derivada da abordagem de Ptolomeu (6) que primeiro propagou o acoplamento hierárquico de MOCC para permitir modelos de execução heterogêneos.

A integração de modelos de execução heterogêneos é a base do acoplamento do simulador. Hoje, os simuladores se concentram na simulação de um ou poucos aspectos de um sistema. À medida que os sistemas ficam mais complexos, eles interagem de uma forma mais complexa com seu ambiente. Ecossistemas de software, por exemplo, consistem em vários sistemas que são executados em plataformas de hardware, se comunicam entre si e implementam algoritmos complexos para interagir com a física e entre si. A simulação de um sistema tão complexo requer a cobertura de todos os aspectos relevantes do ecossistema. Um único simulador não é capaz de fazer isso. Portanto, o acoplamento do simulador torna-se necessário para o desenvolvimento de soluções de simulação que possibilitem, por exemplo, o teste virtual de implementações, a prototipagem virtual de algoritmos e a avaliação virtual de conceitos de arquitetura.

O sistema de acoplamento permite a integração de simuladores, modelos de simulação e outros comportamentos implementados em uma simulação semanticamente integrada. Isso é necessário para superar as limitações dos simuladores existentes que são especializados e, portanto, se concentram em um número limitado de efeitos apenas.

A Figura 4 ilustra o acoplamento do simulador que produz um protótipo virtual para uma função antibloqueio de freio para sistemas automotivos. Ele mostra o componente de software e as dependências de dados entre eles. Para criar um protótipo virtual simples do sistema, todos os componentes de software devem ser vinculados a uma simula-

ção holística. Os quatro componentes de software ilustrados foram criados com Simulink, a linguagem de programação C, ou são fornecidos como FMU. Eles implementam quatro funções de um sistema de freio antitravamento (ABS): RunWSS implementa o sensor de patinação das rodas, RunBS o sensor de freio, RunABS o controlador ABS e RunBA o atuador do freio. Os componentes do software são vinculados a um modelo de ambiente que simula as reações do sistema e fecha o loop. A avaliação do comportamento funcional do sistema requer a execução integrada de todos esses componentes e, portanto, a integração de seus modelos de computação e comunicação (MOCC).

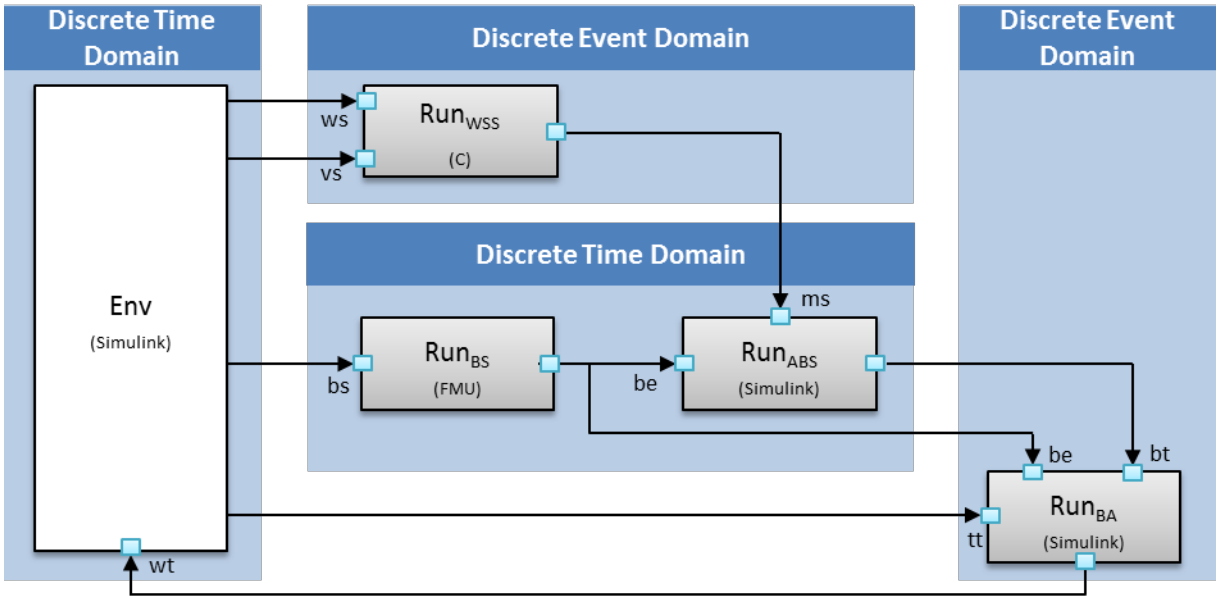

Figura 4 – Componentes de um Sistema de Frenagem Antibloqueio (ABS). Fonte: Documentação FERAL.

A Figura 4 ilustra vários componentes executáveis que fazem parte do sistema em teste ou do ambiente simulado (componente Env). Cada componente faz parte de um domínio que define o MOCC para este componente. Do ponto de vista de uma simulação, todos esses componentes são componentes executáveis (*runnables*) que precisam ser executados de acordo com seu MOCC para fornecer resultados significativos. Um MOCC define quando um executável é executado e como ele se comunica. O exemplo acima usa MOCCs de Tempo Discreto e Evento Discreto para criar domínios de simulação que controlam a execução de runnables contidos. Os domínios de tempo discreto dividem a execução de executáveis controlados em intervalos de tempo discretos. Os domínios de eventos discretos geram eventos para mensagens de chegada e temporizadores. Os executáveis controlados são executados de acordo com esses eventos.

2.4 *Controller Area Network* (CAN)

O barramento CAN foi desenvolvido pela BOSCH como um sistema de transmissão de mensagens multimestre que especifica uma taxa de sinalização máxima de 1 megabit por segundo (bps) (7). Ao contrário de uma rede tradicional como USB ou Ethernet, o CAN não envia grandes blocos de dados ponto a ponto do nó A ao nó B sob a supervisão de um barramento mestre central (8). Em uma rede CAN, muitas mensagens curtas como temperatura ou RPM são transmitidas para toda a rede, o que fornece consistência de dados em cada nó do sistema.

CAN é um barramento de comunicação serial definido pela Organização Internacional de Padronização (ISO) originalmente desenvolvido para a indústria automotiva para substituir o complexo chicote de fiação por um barramento de dois fios (9). A especificação exige alta imunidade a interferências elétricas e a capacidade de autodiagnóstico e reparo de erros de dados. Esses recursos levaram à popularidade do CAN em uma variedade de indústrias, incluindo automação predial, médica e manufatura.

O protocolo de comunicação CAN, ISO-11898: 2003 (10), descreve como a informação é passada entre dispositivos em uma rede e está em conformidade com o modelo de Interconexão de Sistemas Abertos (OSI) que é definido em termos de camadas. A comunicação real entre dispositivos conectados pelo meio físico é definida pela camada física do modelo. A arquitetura ISO 11898 define as duas camadas mais baixas do modelo OSI/ISO de sete camadas como a camada de enlace de dados e a camada física.

Os dispositivos CAN enviam dados através da rede CAN em pacotes chamados quadros. Um quadro CAN consiste nas seguintes seções.

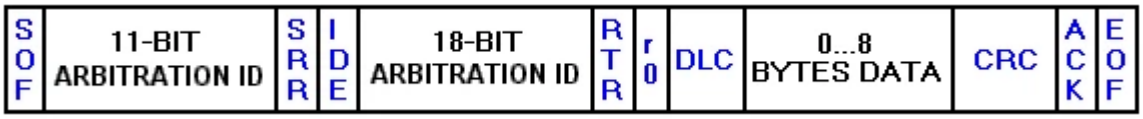

Figura 5 – Formato padrão de quadros CAN. Fonte: National Instruments.

- CAN Frame - uma transmissão CAN inteira: ID de arbitragem, bytes de dados, bit de reconhecimento, e assim por diante. Os quadros também são referidos como mensagens;
- Bit SOF (start-of-frame) – indica o início de uma mensagem com um bit dominante (0 lógico);
- ID de arbitragem – identifica a mensagem e indica a prioridade da mensagem. Os quadros vêm em dois formatos - padrão, que usa um ID de arbitragem de 11 bits, e estendido, que usa um ID de arbitragem de 29 bits;

- Bit IDE (extensão do identificador) – permite diferenciação entre quadros padrão e estendidos;
- Bit RTR (solicitação de transmissão remota) – serve para diferenciar um quadro remoto de um quadro de dados. Um bit RTR (0 lógico) dominante indica um quadro de dados. Um bit RTR recessivo (1 lógico) indica um quadro remoto;
- DLC (código de comprimento de dados) – indica o número de bytes que o campo de dados contém;
- Data Field – contém 0 a 8 bytes de dados;
- CRC (verificação de redundância cíclica) – contém código de verificação de redundância cíclica de 15 bits e um bit delimitador recessivo. O campo CRC é usado para detecção de erros;
- Slot ACK (ACKnowledgement) – qualquer controlador CAN que recebe corretamente a mensagem envia um bit ACK no final da mensagem. O nó de transmissão verifica a presença do bit ACK no barramento e tenta novamente a transmissão se nenhum reconhecimento for detectado.

2.4.1 Como funciona a comunicação da CAN

Como dito anteriormente, o CAN é uma rede peer-to-peer. Isso significa que não há nenhum mestre que controle quando nós individuais têm acesso a dados de leitura e gravação no barramento CAN. Quando um nó CAN está pronto para transmitir dados, ele verifica se o barramento está ocupado e, em seguida, simplesmente grava um quadro CAN na rede. Os quadros CAN transmitidos não contêm endereços do nó transmissor ou de qualquer um dos nós receptores pretendidos. Em vez disso, um ID de arbitragem único em toda a rede rotula o quadro. Todos os nós da rede CAN recebem o quadro CAN e, dependendo do ID de arbitragem desse quadro transmitido, cada nó CAN na rede decide se aceita o quadro.

Se vários nós tentarem transmitir uma mensagem para o barramento CAN ao mesmo tempo, o nó com a maior prioridade (ID de arbitragem mais baixa) recebe automaticamente acesso ao barramento (11). Os nós de menor prioridade devem esperar até que o barramento fique disponível antes de tentar transmitir novamente. Desta forma, você pode implementar redes CAN para garantir a comunicação determinística entre os nós CAN.

3 Arquitetura do FERAL

3.1 Estrutura Modular e funcionamento dos componentes

A lógica do núcleo FERAL é fornecer um *core* de simulação que permite a integração rápida de modelos de simulação, ou seja, Modelos de Computação e Comunicação (MOCC). Com base neste núcleo, vários aplicativos de simulação podem ser criados. FERAL cria uma árvore de componentes (veja a Figura 6) que começa no componente raiz. Os componentes intermediários contêm componentes subsequentes, como diretores que definem os Modelos de Computação e Comunicação (MOCC).

Os componentes folha são componentes de simulação, esses contêm portas que são os terminais para comunicação entre componentes (12). A execução de cada componente de simulação é controlada pelo diretor principal mais próximo, o qual também controla outros diretores.

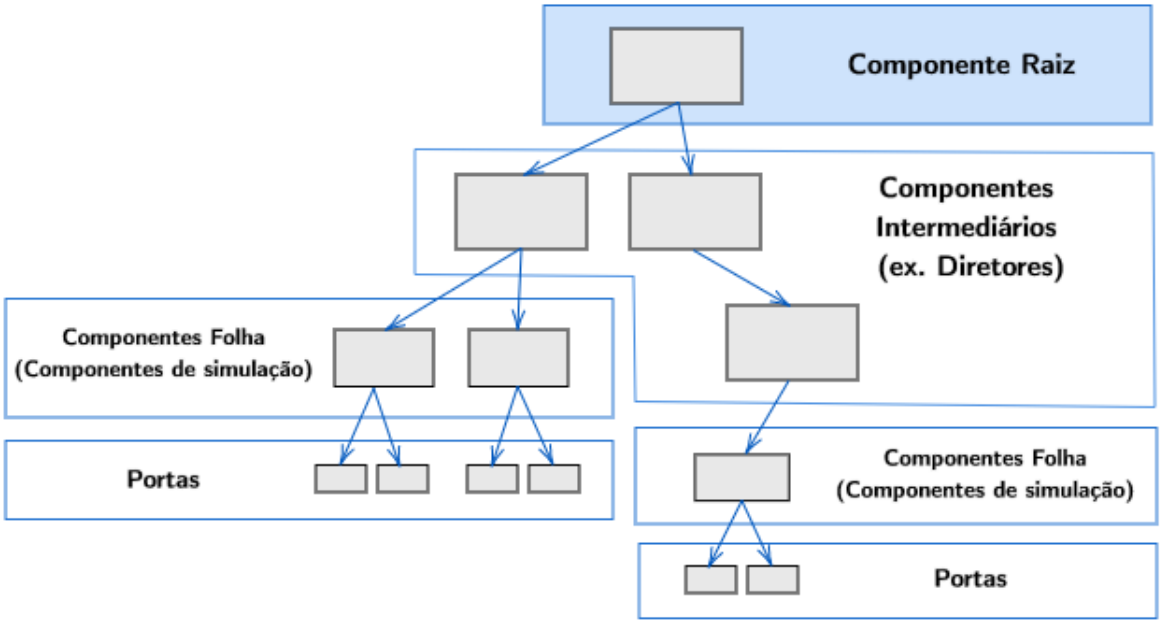

Figura 6 – Árvore de componentes do FERAL. Fonte: Autoral.

A Figura 7 ilustra os principais componentes de uma simulação do FERAL. Essa consiste em um cenário e um domínio raiz. O cenário define aspectos gerais da simulação, por exemplo, como ele se comunica com o usuário e se sua execução é controlada remotamente. O domínio raiz é um componente executável pela ferramenta, normalmente, o MOCC de nível superior da simulação. Um MOCC define a semântica de execução e comunicação de componentes controlados.

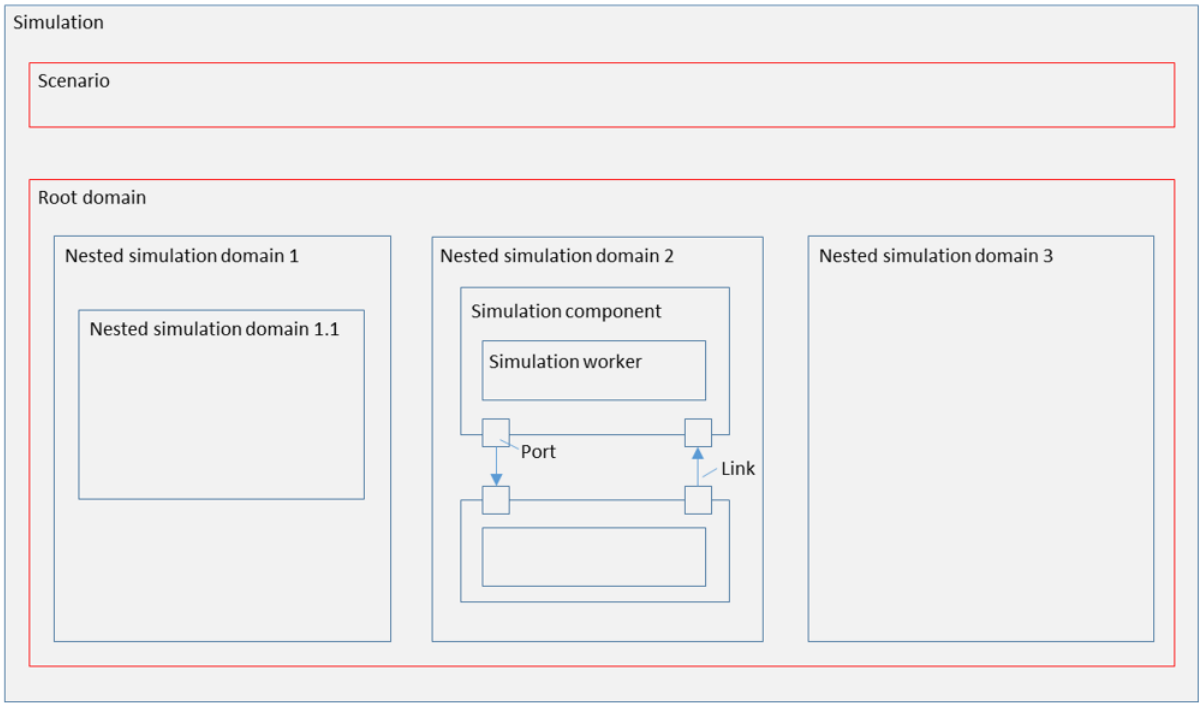

Figura 7 – Principais componentes de uma simulação do FERAL. Fonte: Documentação FERAL.

Como simulações complexas não podem ser realizadas com um único MOCC, o FERAL suporta o acoplamento hierárquico de MOCCs (12). Os domínios de simulação aninhados definem os MOCCs que estão contidos em seu domínio principal. Qualquer número de MOCCs pode ser aninhado. A execução de um MOCC aninhado é controlada pelo domínio peincipal. Se um domínio de simulação for colocado, por exemplo, dentro de um domínio de tempo real que alinha os tempos de simulação de componentes contidos ao horário do relógio de parede, o domínio de simulação aninhado será sincronizado ao tempo do relógio da melhor maneira possível.

Os componentes de simulação do FERAL implementam a conexão entre um MOCC e os trabalhadores de simulação. Ambos os componentes de simulação e trabalhadores de simulação são geralmente adaptados a um MOCC específico. Trabalhadores de simulação integram modelos de comportamento em uma simulação do FERAL. O software define classes de trabalho para a maioria dos MOCCs que permitem a implementação de modelos de simulação de acordo com a semântica desse MOCC.

A maioria dos trabalhadores usa o componente de simulação genérico de seu MOCC para simplificar a integração. O comportamento padrão da implementação do componente de simulação MOCC é suficiente para a maioria dos casos. No entanto, usar o componente de simulação como uma ponte explícita entre o MOCC e um trabalhador de simulação permite a customização dessa interação, criando um componente de simulação personalizado.

Os simuladores e comportamentos existentes são integrados ao FERAL pela subclasse de uma classe de trabalhadores. Os trabalhadores de simulação interagem com o componente de simulação para acessar as funções do MOCC, dessa forma, a execução e comunicação de um trabalhador de simulação está em conformidade com o MOCC.

O FERAL realiza comunicação entre componentes de simulação através de portas (12). São definidas a porta de entrada e saída, bem como as portas bidirecionais que permitem a comunicação em ambas as direções. Os links conectam a saída às portas de entrada e transportam mensagens de comunicação. Toda a comunicação da ferramenta entre as portas é feita por meio de mensagens de comunicação explícitas e tipadas.

O seguinte cenário de exemplo básico, ilustra a estrutura de um exemplo de simulação do FERAL e os componentes principais que estão envolvidos na simulação. A simulação de exemplo é uma simulação de tempo discreto com um componente produtor (SampleTTProducer) que gera mensagens repetidamente e um componente consumidor (SampleTTConsumer) que recebe mensagens. Um link conecta a porta de saída do componente produtor à porta de entrada do componente consumidor.

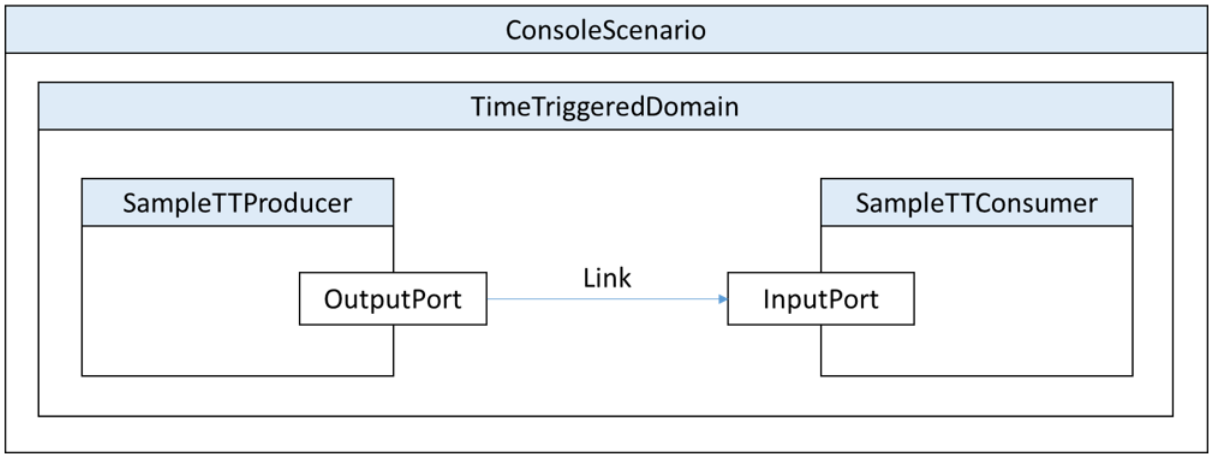

Figura 8 – Exemplo de estrutura de cenário. Fonte: Documentação FERAL.

O diagrama de instância a seguir descreve a estrutura do componente dinâmico que é criada pelo cenário, bem como os principais relacionamentos entre os componentes:

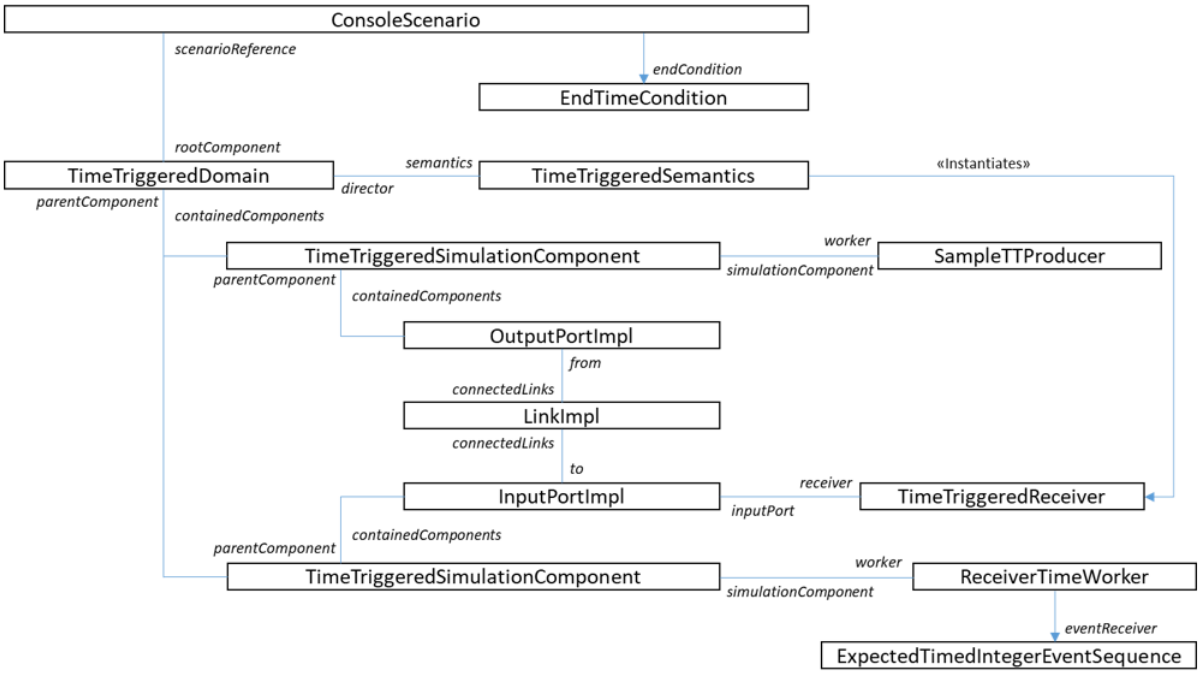

Figura 9 – Estrutura de componentes resultado do cenário. Fonte: Documentação FERAL.

Os seguintes componentes estão envolvidos no cenário de simulação de exemplo descrito acima:

- **ConsoleScenario:** Especialização da classe genérica *Scenario* descrita acima. O cenário é a classe raiz de toda simulação. Ele define o componente raiz, bem como o comportamento básico que inicializa e executa a simulação. O cenário do console é responsável pelo envio de mensagens de depuração para o console.
- **EndTimeCondition:** EndTimeCondition é uma *EndCondition* especializada. O *EndCondition* de um cenário de simulação define quando uma simulação termina. O EndTimeCondition termina a simulação em um tempo de simulação definido. No exemplo, isso ocorre 5 segundos após o início da simulação.
- **TimeTriggeredDomain:** O domínio disparado por tempo é um diretor do FERAL que instancia a semântica disparada por tempo discreto. Um componente diretor é genérico, ele não implementa diretamente o MOCC. As especializações geralmente não alteram o componente do diretor em si, mas instanciam a semântica que implementa o MOCC.
- **TimeTriggeredSemantics:** a classe de semântica acionada por tempo implementa o MOCC de tempo discreto. Um MOCC define como os componentes contidos executam e se comunicam. Portanto, define a estratégia para executar os componentes contidos e define o receptor que é atribuído às portas de entrada dos componentes

contidos. Ele também define o comportamento da porta de entrada e, portanto, o comportamento de comunicação de todos os componentes controlados pelo diretor e sua semântica.

- **TimeTriggeredSimulationComponent:** os componentes de simulação são a ponte entre a semântica genérica do FERAL e as interfaces para o acoplamento MOCC e o comportamento específico do MOCC exigido pelos trabalhadores. Um componente de simulação, portanto, implementa uma ponte entre um tipo específico de trabalhador de simulação e o diretor do FERAL. Os componentes de simulação acionados por tempo definem a atuação dos componentes de simulação que implementam um comportamento de tempo discreto.

- **SampleTTProducer:** O produtor de amostras TT é um trabalhador de simulação disparado por tempo discreto. Ele implementa um comportamento de tempo discreto que gera mensagens periodicamente e as transmite através da porta de saída de seu componente de simulação. O trabalhador de simulação acionado por tempo interage com um componente de simulação acionado por tempo.

- **ReceiverTimeWorker:** O trabalhador de tempo do receptor é um trabalhador de simulação disparado por tempo discreto. Ele recebe mensagens e as encaminha para o *ExpectedTimedIntegerEventSequence* atribuído, que compara as mensagens recebidas e os horários de recebimento com um conjunto de eventos esperado.

- **OutputPortImpl:** porta de saída que transmite mensagens de saída por um link conectado.

- **LinkImpl:** Um link que conecta uma porta de saída a uma porta de entrada.

- **InputPortImpl:** portas de entrada recebem mensagens de links conectados. O comportamento exato desta porta de entrada é definido pelo ReceiverTimeWorker que implementa o comportamento do receptor em conformidade com o tempo discreto MOCC.

- **ReceiverTimeWorker:** implementa o comportamento do receptor de acordo com o tempo discreto MOCC.

4 Núcleo do FERAL

4.1 Arquitetura do núcleo

A Figura 10 ilustra uma visão mais detalhada dos componentes e interfaces do FERAL que também destaca os principais conceitos do FERAL. Conforme descrito anteriormente, um diretor do FERAL cria um domínio de simulação cuja semântica está em conformidade com um MOCC específico (12). Todos os componentes definidos são controlados por este MOCC, independentemente do componente controlado ser um componente de simulação ou um diretor. O MOCC usa uma interface específica *RunnableComponent* que todos os componentes executáveis, implementam para esse fim. Cada diretor implementa uma interface específica que permite que os componentes de simulação aninhados e os diretores interajam com o diretor que os inclui. Ele define funções diferentes para diretores aninhados e componentes de simulação aninhados para permitir um controle mais refinado da semântica de execução para cada tipo de componente.

Um diretor e seu MOCC controlam a comunicação atribuindo o elemento receptor da porta de entrada às portas de entrada, controlando assim o comportamento dessas (12). A implementação do MOCC também controla quando uma porta de saída encaminha mensagens em espera. Esse controle é executado por meio de portas de entrada definidas e interfaces de porta de saída que são implementadas por todos os tipos de portas de entrada/saída.

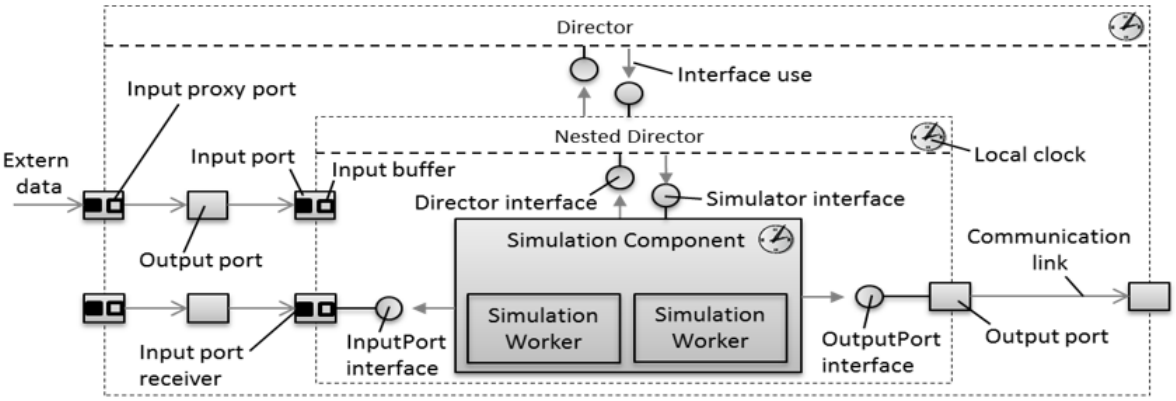

Figura 10 – Componentes e Interfaces do FERAL. Fonte: Documentação FERAL.

Um componente de simulação está em conformidade com um MOCC específico. Portanto, precisa ser controlado por este MOCC ou por uma especialização do mesmo. Os componentes de simulação, portanto, podem atribuir premissas sobre seu MOCC de controle e usar interfaces especializadas para interagir com ele.

Conforme mostrado na Figura 10, o componente de simulação não implementa diretamente o comportamento. Em vez disso, eles fornecem uma interface para trabalhadores de simulação. Estes integram ou implementam o comportamento na simulação. Isso permite uma separação clara de responsabilidades: os componentes de simulação implementam a ponte para o MOCC, os trabalhadores de simulação fornecem a interface sintática para modelos de simulação ou implementam o comportamento. Dessa forma, enquanto os trabalhadores de simulação realizam essa interface sintática, o MOCC realiza a interface semântica. Eles integram, por exemplo, Simulink, código C ou unidades de simulação funcional (FMU), mas também simuladores existentes e seus modelos de simulação.

A comunicação entre os componentes é feita por meio de links que conectam as portas de saída às portas de entrada. Esses links podem cruzar as fronteiras de um domínio de simulação. O acoplamento de modelos de comunicação entre domínios é realizado por meio de portas proxy de entrada. As portas proxy de entrada são usadas quando os dados recebidos são trocados entre componentes controlados por placas diferentes e, portanto, em domínios diferentes. Semelhante às portas de entrada regulares, as portas proxy de entrada contêm um receptor que é definido pelo diretor de controle. As portas proxy de entrada garantem a semântica de comunicação correta ao transmitir mensagens entre MOCCs. Os MOCCs de fechamento definem quando uma mensagem é encaminhada das portas de saída para os receptores, por exemplo, para uma entrada ou uma porta proxy de entrada. Os MOCCs delimitadores definem quando as mensagens das portas do proxy de entrada são encaminhadas para as portas de entrada reais dos componentes de simulação.

Os componentes de simulação são preparados para serem usados em diferentes domínios. Isso pode ser viável se um componente de simulação implementar um comportamento muito genérico, por exemplo, fornecendo um valor constante em uma de suas portas de saída. Esses componentes de simulação sofrem com o fato de que o comportamento da porta de entrada será vinculado ao MOCC do domínio de simulação em que são colocados e, portanto, não é previsível (12). Devido a isso, os componentes de simulação têm um mecanismo para ajustar o comportamento das portas de entrada. Enquanto os receptores de porta de entrada realizam a interface das portas de entrada para o MOCC, os buffers de entrada realizam a interface para o trabalhador. Portanto, um componente de simulação pode decidir, por exemplo, se um acesso de leitura deve remover o valor de uma porta de entrada ou se o valor deve permanecer por toda a chamada. Os buffers de entrada, entretanto, podem adaptar o comportamento da porta de entrada apenas nos limites definidos pelo receptor. Assim, eles não podem alterar a semântica do MOCC delimitador, por exemplo, recebendo mensagens que ainda não foram entregues ao componente.

Para evitar erros, a execução de todos os modelos de simulação precisa ser sincroni-

zada. A sincronização de tempo de modelos de simulação em todos os domínios e MOCCs é obtida aninhando modelos de simulação em uma estrutura de árvore. Cada componente de simulação e cada diretor, portanto, mantém um tempo de simulação. Com base na hierarquia MOCC, o tempo de simulação de dois componentes FERAL pode, portanto, ser diferente.

A mesma interface de simulação é implementada por todos os componentes executáveis, ou seja, diretores aninhados e componentes de simulação. A semântica básica da interface é derivada de Ptolomeu (6). Essa interface permite seu aninhamento hierárquico e permite a implementação de MOCCs para controlar componentes e diretores que estejam nos tais. Ela define cinco operações básicas: *preInitialize* e *initialize* configuram um componente antes de ser executado pela primeira vez; *preInitialize* é chamado para cada entidade para habilitar a configuração local. As operações *preFire*, *fire* e *postFire* controlam o disparo de um componente executável. A operação *preFire* determina se a entidade pode ser executada (disparada) em um determinado momento. A operação de disparo executa um componente. A operação *postFire* altera o estado externamente visível de um componente após dispará-lo, transmitindo mensagens das portas de saída para as portas de entrada por meio de links entre os componentes conectados.

5 Aplicação

5.1 Visão geral da simulação de uma rede no FERAL

Na simulação de uma rede no FERAL, uma interface de dispositivo define uma pilha de comunicação que consiste em um *Coder*, uma *Queue* e um MAC que implementa o comportamento do controlador específico da rede. O *Coder* se comunica por meio da porta do aplicativo, criando *LogicNetworkFrames* para quadros recebidos da rede simulada. Para CAN, ele lida, por exemplo, com a simulação de *bit stuffing*. A *Queue* define o comportamento de enfileiramento da rede. Nesse caso ele define, por exemplo, se os quadros em espera com a mesma ID de rede se sobrescrevem ou se são filas. Ele também define a estratégia de enfileiramento e se os quadros com IDs específicos ganham prioridade sobre outros quadros. Já O componente MAC implementa o comportamento específico do controlador, que geralmente é a função de controle de acesso ao meio, mas também inclui tratamento de erros e, possivelmente, funções de tratamento de colisões (12).

5.2 Implementação de uma rede CAN no FERAL

Contruiu-se uma simulação para exemplificar o uso da ferramenta. Dessa forma, foram utilizadas bibliotecas do FERAL de forma a compor um modelo para um barramento de rede do tipo CAN.

A aplicação simula todos os protocolos e camadas presentes em uma *Controller Area Netwok*. No cenário escolhido a rede possui um barramento de 100 Kbits/s. O código em Java utilizado para implementação da simulação pode ser encontrado no ANEXO A do presente trabalho.

Abaixo está o resultado obtido na aplicação, onde mensagens de teste foram enviadas por um transmissor (Tx) e recebidas por um receptor (Rx).

Figura 11 – Teste de simulação da rede CAN. Fonte: Autoral.

Na Figura 11, observa-se que o comportamento de referência esperado foi obedecido. Visto que pacotes de mensagem chegaram a cada 10 ms, e os pacotes recebidos obedeceram a sequencia de eventos estabelecida em código. O correto envio e recebimento dos dados, sem o comprometimento da informação, indica o funcionamento adequado da rede e dos modelos instanciados.

6 Considerações Finais

Considerando o crescente desenvolvimento de sistemas complexos, a virtualização dos processos de desenvolvimento e a crescente demanda por redução de tempo e custos de projeto, o presente trabalho teve como objetivo fazer um estudo sobre uma ferramenta com grande potencial de desenvolvimento e promissora capacidade de colaboração na indústria e pesquisa, o FERAL.

Percebeu-se que o FERAL tem a capacidade de construir modelos de simulação deveras complexos. Para tal, faz uso da sua interface de integração entre diversos simuladores, somando assim, a especificidade da área de aplicação de cada software aos cenários implementados no FERAL.

Além disso, a possibilidade de construir modelos tão elaborados da realidade, corrobora com a atual têndencia da indústria de virtualizar a construção e teste de protótipos, o que reduz tempo e custo no desenvovimento de produtos.

Porém, para atingir seus plenos benefícios a ferramenta requer alto nível de compreensão do seu funcionamento, assim, sendo necessário treinamento contínuo e aprofundado da mão de obra que será especializada na operação do software.

Por fim, conclui-se que o FERAL tem um futuro promissor como ferramenta de modelagem e simuação. Pois, explora um campo pouco enfatizado por outros softwares, que é a integração entre diferentes plataformas de simulação. Seu potêncial máximo de funcionamento ainda está por vir, visto que o mesmo ainda está em sua fase de desenvolvimento. A UFCG atualmente colabora com o *Fraunhofer IESE* para desenvolver um modelo do 5G para o FERAL. O estudo de novas aplicações que podem construir novos caminhos para além da engenharia tradicional também devem ser considerados.

Referências

1 CHATTERJEE, D.; DEORIO, A.; BERTACCO, V. Chapter 23 - high-performance gate-level simulation with gp-gpus. In: HWU, W. mei W. (Ed.). *GPU Computing Gems Emerald Edition*. Boston: Morgan Kaufmann, 2011, (Applications of GPU Computing Series). p. 343–364. ISBN 978-0-12-384988-5. Disponível em: <https://www.sciencedirect.com/science/article/pii/B9780123849885000231>. 3

2 PRADO, D. S. do. *Teoria das Filas e da Simulação*. 2ª ed. Belo Horizonte, MG: INDG Tecnologia e Serviços LTDA, 2004. 3

3 OVTCHAROVA, J. Virtual engineering: Principles, methods and applications. In: . [S.l.: s.n.], 2010. 4

4 GREGA, W. Hardware-in-the-loop simulation and its application in control education. In: . [S.l.: s.n.], 1999. v. 2, p. 12B6/7 – 12B612 vol.2. ISBN 0-7803-5643-8. 5

5 ELLIS, G. Chapter 13 - model development and verification. In: ELLIS, G. (Ed.). *Control System Design Guide (Fourth Edition)*. Fourth edition. Boston: Butterworth-Heinemann, 2012. p. 261–282. ISBN 978-0-12-385920-4. Disponível em: <https://www.sciencedirect.com/science/article/pii/B9780123859204000138>. 5

6 JANNECK, J. et al. Disciplining heterogeneity – the ptolemy approach. In: . [S.l.: s.n.], 2001. 8, 19

7 CAN in Automation. *CAN knowledge*. 2021. Disponível em: <https://www.can-cia.org/can-knowledge/>. Acesso em: 05 de outubro de 2021. 10

8 ETSCHBERGER, K.; HOFMANN, R.; STOLBERG, J. Controller area network: Basics, protocols, chips and applications. 09 2021. 10

9 AZZEH, A. Can control system for an electric vehicle. 10 2021. 10

10 STANDARDIZATION, I. O. for. *Road vehicles — Controller area network (CAN) — Part 1: Data link layer and physical signalling*. ISO 11898-1:2003. Vernier, Geneva, Switzerland: International Organization for Standardization, 2003. Disponível em: <https://www.iso.org/standard/33422.html>. 10

11 PREZ, J.; REORDA, M. S. Dependability analysis of can networks:. 09 2003. 11

12 IESE, F. I. for E. S. E. *FERAL Documentation*. 16.06.2021. Fraunhofer-Platz 1, 67663 Kaiserslautern, Germany: MidiaWiki, 2021. 12, 13, 14, 17, 18, 20

Anexos

ANEXO A – Códigos

Abaixo está o código em Java do exemplo proposto.

```
1  package de.fraunhofer.iese.feral.ext.smd.comnet.wired.regression.
       scripts;
2
3
4  import static org.junit.Assert.assertTrue;
5
6  import org.junit.Test;
7
8  import de.fraunhofer.iese.feral.ext.smd.comnet.wired.nic.can.
       network.CANNetworkNG;
9  import de.fraunhofer.iese.feral.ext.smd.comnet.wired.nic.can.
       network.CANNetworkParameter.CANBitstuffingMode;
10 import de.fraunhofer.iese.feral.ext.smd.comnet.wired.util.tools.
       SenderEventWorker;
11 import de.fraunhofer.iese.feral.krnl.scf.core.framework.
       components.link.BiLinkImpl;
12 import de.fraunhofer.iese.feral.krnl.scf.core.framework.director.
       coredirector.Director;
13 import de.fraunhofer.iese.feral.krnl.scf.core.framework.scenario.
       Scenario;
14 import de.fraunhofer.iese.feral.krnl.scf.core.framework.time.
       SimulationDuration;
15 import de.fraunhofer.iese.feral.krnl.scf.core.framework.time.
       SimulationTime;
16 import de.fraunhofer.iese.feral.krnl.scf.framework.director.
       eventdirector.simpleapi.EventTriggeredDomain;
17 import de.fraunhofer.iese.feral.krnl.scf.framework.scenario.
       ConsoleScenario;
18 import de.fraunhofer.iese.feral.krnl.scf.framework.util.test.
       regression.receiver.ReceiverEventWorker;
19 import de.fraunhofer.iese.feral.krnl.scf.framework.util.test.
       regression.sequence.ExpectedEventSequence;
20 import de.fraunhofer.iese.feral.krnl.scf.framework.util.test.
       regression.sequence.ExpectedTimedIntegerEventSequence;
21
22
```

```
23 /**
24  * Test case: Activity behavior
25  *
26  * @author kuhn
27  *
28  */
29 public class Script_CAN_100kBit_1Byte {
30
31
32     /**
33      * Run test case
34      */
35     @Test
36     public void test() {
37         ////////////////////////////////////////////////
38         // Create simulation scenario and root director
39         // - Create scenario with defined ending time
40         Scenario simulationScenario = new ConsoleScenario(
               SimulationTime.MS(55));
41         // - Create discrete event director as root component
               with basic component topology
42         Director director = new EventTriggeredDomain(null,
               SimulationDuration.NS(500));
43         // - Add director as root component to director
44         simulationScenario.setRootComponent(director);
45
46
47         ////////////////////////////////////////////////
48         // Checked event sequence
49         ExpectedEventSequence expectedEventSequence = new
               ExpectedTimedIntegerEventSequence(simulationScenario);
50
51
52         ////////////////////////////////////////////////
53         // Regression test setup
54
55         // Discrete event applications for maximum accuracy
56         // - Create CAN sender
57         SenderEventWorker canSender = new SenderEventWorker(
               director, 0, SimulationDuration.MS(10));
58         // - Create CAN receiver
```

```
59          ReceiverEventWorker canReceiver = new ReceiverEventWorker
               (director, expectedEventSequence);
60
61          // Create CAN Network with 2 nodes
62          CANNetworkNG network = new CANNetworkNG(director, 2);
63          // - Setup network speed - 100kBit
64          network.setBitRate(100000);
65          network.setBitStuffingMode(CANBitstuffingMode.NONE);
66          // - Setup communication interfaces, workers communicate
               via message type "63", it is encoded to two bytes
               length
67          network.getTopology().getInterface(0).addTxMessageType("
               63", "app1", 1);
68          network.getTopology().getInterface(1).addRxMessageType("
               63", "app1", 1);
69          // - Setup links from/to network
70          new BiLinkImpl(canSender, "comm", network.getTopology().
               getInterface(0), "app1");
71          new BiLinkImpl(canReceiver, "comm", network.getTopology()
               .getInterface(1), "app1");
72
73
74
75          ////////////////////////////////////////////////
76          // Expected reference behavior
77          // Message queues at 10ms, 520 s  tx time without bit
               stuffing (standard frame, incl. EOF) - Minimum IFS to
               following frame is 3 Bits
78          expectedEventSequence.setExpectedEventSequence(new long
               [][] {
79              { 105200001, 01},
80              { 205200001, 11},
81              { 305200001, 21},
82              { 405200001, 31},
83              { 505200001, 41},
84          });
85
86          ////////////////////////////////////////////////
87          // Start simulator
88          simulationScenario.run();
89          assertTrue(expectedEventSequence.check());
90      }
```

```
91 }
```