UNIVERSIDADE
FEDERAL RURAL
DE PERNAMBUCO

Daniel José Freire de Araújo

Verificação de Refinamento em Diagramas de Sequência com Estruturas de Controle

Recife

2019

Daniel José Freire de Araújo

Verificação de Refinamento em Diagramas de Sequência com Estruturas de Controle

Monografia apresentada ao Curso de Bacharelado em Ciências da Computação da Universidade Federal Rural de Pernambuco, como requisito parcial para obtenção do título de Bacharel em Ciências da Computação.

Universidade Federal Rural de Pernambuco – UFRPE

Departamento de Computação

Curso de Bacharelado em Ciências da Computação

Orientador: Lucas Albertins de Lima

Recife

2019

MINISTÉRIO DA EDUCAÇÃO E DO DESPORTO
UNIVERSIDADE FEDERAL RURAL DE PERNAMBUCO (UFRPE)
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

http://www.bcc.ufrpe.br

Ficha de Aprovação do Trabalho de Conclusão de Curso

Trabalho defendido por **DANIEL JOSÉ FREIRE DE ARAÚJO** às 10:00 do dia 06 de dezembro de 2019, no Auditório do Departamento de Computação - DC – Sala 07, como requisito para conclusão do curso de Bacharelado em Ciência da Computação da Universidade Federal Rural de Pernambuco, intitulado " **Verificação de Refinamento em Diagramas de Sequência com Estruturas de Controle**", orientado por Lucas Albertins de Lima e aprovado pela seguinte banca examinadora:

Lucas Albertins de Lima
DC/UFRPE

Sidney de Carvalho Nogueira
DC/UFRPE

Dados Internacionais de Catalogação na Publicação
Universidade Federal Rural de Pernambuco
Sistema Integrado de Bibliotecas
Gerada automaticamente, mediante os dados fornecidos pelo(a) autor(a)

A663v Araújo, Daniel José Freire de
Verificação de Refinamento em Diagramas de Sequência com Estruturas de Controle / Daniel José Freire de Araújo. - 2019.
70 f. : il.

Orientador: Lucas Albertins de Lima.
Inclui referências.

Trabalho de Conclusão de Curso (Graduação) - Universidade Federal Rural de Pernambuco, Bacharelado em Ciência da Computação, Recife, 2019.

1. UML. 2. Diagrama de Sequência. 3. Refinamento. 4. Estruturas de Controle. I. Lima, Lucas Albertins de, orient. II. Título

CDD 004

Agradecimentos

Agradeço primeiramente a meus pais pelo amor, apoio e incentivo nas horas difíceis, de desânimo e cansaço.

Ao meu orientador, que sempre mostrou confiança em mim. Obrigado pelo convite para iniciação científica, pela orientação e dedicação à elaboração deste trabalho.

A esta universidade, seu corpo docente, direção e administração que oportunizaram a janela que hoje vislumbro.

Aos meus amigos, companheiros de trabalhos e irmãos na amizade que fizeram parte da minha formação e que vão continuar presentes em minha vida com certeza.

A todos que direta ou indiretamente fizeram parte da minha formação, o meu muito obrigado.

“A persistência é o caminho do êxito.”
(Charles Chaplin)

Resumo

A linguagem UML oferece diversos tipos de diagramas para modelagem de sistemas, entre os principais diagramas comportamentais está o diagrama de sequência. O diagrama de sequência pode ser utilizado para modelar casos de uso de sistemas de forma simples e visual. Contudo, a linguagem UML como um todo apresenta modelos informais que possuem verificações baseadas em experiência humana. Este trabalho é a continuação de um linha de pesquisa que tem o objetivo de formalizar diagramas de sequência UML e realizar verificações de refinamento entre diagramas. Aqui é proposta a versão inicial de uma ferramenta capaz de traduzir diagramas de sequência UML para uma especificação formal CSP e realizar a verificação de refinamentos através do verificador FDR4. O ponto diferencial deste trabalho é o processo de formalização de fragmentos combinados que representam estruturas de controle no contexto de diagramas de sequência, aqui serão abordados os fragmentos *option, alternative, parallel e loop*.

Palavras-chave: UML, Diagrama de Sequência, Formal, CSP, FDR, Refinamento,Estruturas de Controle ,Fragmento Combinado.

Abstract

The UML language offers several types of diagrams for system modeling, among the main behavioral diagrams is the sequence diagram. The sequence diagram can be used to model system use cases simply and visually. However, the UML language as a whole presents informal models that can only be verified by human experience. This paper refers to the continuation of a research line that aims to formalize UML sequence diagrams and perform refinement checks between diagrams. Here we propose an initial version of a tool capable of translating UML sequence diagrams into CSP and performing a refinement check using the FDR4 verifier. The differential point of this work is the process of formalization of combined fragments that represent control structures in sequence diagrams, here we will cover fragments such as *option,alternative, parallel and loop*.

Keywords: UML, Sequence Diagram, Formal, CSP, FDR, Refinement,Control Structure ,Combined Fragment

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

UML	Unified Modeling Language
CSP	Communicating Sequential Processes
OCL	Object Constraint Language
APAC	Agência Pernambucana de Águas e clima
SIRHPE	Sistema de Informação para Recursos Hídricos de Pernambuco
PCD	Plataforma de Coleta de Dados
CEMADEN	Centro Nacional de Monitoramento e Alertas de Desastres Naturais
INPE	Instituto Nacional de Pesquisas Espaciais
CPTEC	Centro de Previsão de Tempo e Estudos Climáticos

Sumário

1 Introdução

Este trabalho se trata da extensão do projeto de iniciação científica intitulado ”Desenvolvimento de um Arcabouço para Verificação de Refinamento de Modelos UML Comportamentais”.

1.1 Contexto da Pesquisa

Um projeto em etapas (*stepwise design*) de um sistema inicia de um modelo abstrato e evolui para um modelo concreto enquanto os conceitos do sistema amadurecem (KHENDEK; BOURDUAS; VINCENT, 2001). Essa tarefa de projetar é comumente realizada através da produção de uma série de modelos de projeto onde cada modelo é um refinamento de um anterior. Durante o processo tradicional de desenvolvimento de software, é comum que se construa uma série de diagramas que representam visualmente a arquitetura ou funcionamento de alguma parte do sistema (SOMMERVILLE, 2011). Nesse contexto, um refinamento entre diagramas representa que um diagrama mais recente preserva propriedades de um diagrama construído previamente, mas adiciona novas características.

A linguagem UML (UML, 2015) é bastante conhecida como um padrão para se criar modelos de projeto de sistemas, sejam eles de hardware ou de software, ela possui diagramas que possibilitam a modelagem tanto da parte estrutural como também a parte comportamental de sistemas. Dentre os diagramas comportamentais, o diagrama de sequência é comumente utilizado em projetos de software, onde tem o propósito de descrever interações entre diversas entidades do sistema. A Figura 1 ilustra um diagrama de sequência simples.

A imagem ilustra um diagrama com dois objetos, x da classe A e y da classe B. Esses objetos trocam mensagens entre si, tais mensagens são representadas pelas setas e seus respectivos rótulos, m0 e m1. As características do diagrama de sequência e seus componentes serão melhor exploradas no capítulo de Fundamentação Teórica.

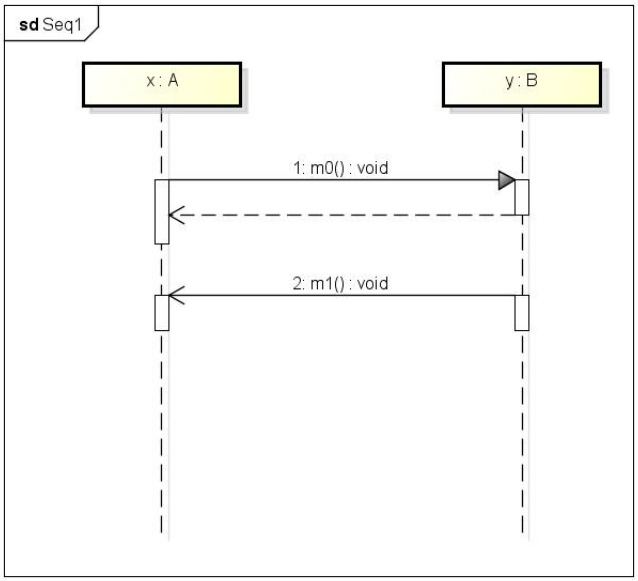

Figura 1 – Diagrama de sequência simples

1.2 Problema da pesquisa

Modelos de projeto podem ter semânticas informais ou formais, quando falamos de modelos formais no contexto de software é comum que sejam citados um ou mais métodos formais. “Métodos formais utilizados no desenvolvimento de sistema de computadores são técnicas matemáticas para descrever propriedades de um sistema. Tais métodos formais provém frameworks nos quais as pessoas podem especificar, desenvolver e verificar sistemas de forma sistemática e não ad hoc”(WING, 1990) tradução nossa.

Modelos informais, como os que usam a linguagem UML (KENT; EVANS; RUMPE, 1999), são populares pois permitem representar ideias de forma simples e visual, com poucas regras e limitações preestabelecidas. Tanto nos processos tradicionais quanto nas metodologias ágeis como *Scrum* (SCHWABER; BEEDLE, 2002) e *Extreme Programming* (BECK, 2000), ainda é comum se utilizar diferentes tipos de diagramas UML em projetos de software, seja de forma obrigatória ao processo, visando representar a estrutura de elementos de um sistema, ou de forma complementar para compartilhar o entendimento de um projeto dentro de uma equipe de desenvolvedores.

Os modelos formais por sua vez, são mais complexos devido à necessidade de entendimento e manipulação de conceitos matemáticos e semânticas formais (WINSKEL, 1993). Esses tipos de modelos não possuem uma aplicação tão comum quanto os modelos informais, e por isso são geralmente utilizados em contextos onde são obrigatórios, como por exemplo em sistemas críticos (STOREY, 1996). De forma geral, os modelos formais são mais seguros pois garantem corretude por prova matemática, além de possuírem diversas regras preestabelecidas que evitam ambiguidades e di-

vergências semânticas. Outro ponto característico de modelos formais é a existência de ferramental para realizar diversos tipos de validações entre seus modelos de forma automatizada.

A verificação de refinamento é um exemplo de verificação que pode ser realizada em modelos formais. Uma verificação de refinamento entre modelos deve verificar se as propriedades definidas em um modelo abstrato estão presentes em um modelo mais concreto, independentemente do modelo concreto possuir novas propriedades. No contexto de diagramas UML, uma verificação de refinamento entre diagramas iria verificar se as propriedades definidas em um diagrama de mais alto nível se mantém em um diagrama de mais baixo nível e com mais detalhes.

Quando um modelo de projeto tem uma semântica informal como no UML, cabe ao projetista julgar se o modelo concreto é um refinamento de um modelo abstrato. O projetista se baseia somente na sua intuição e experiência, as quais podem levar a erros devido ao fator humano. Em projetos de sistemas críticos, onde um erro pode levar a situações desastrosas como grande perda financeira ou perda de vidas humanas, são utilizados modelos com semânticas formais, uma vez que esse tipo de modelo pode ser analisado de forma automática através de ferramentas como verificadores de modelos (BAIER; KATOEN, 2008). Uma inconsistência não detectada nas etapas iniciais de um projeto de software pode resultar em um alto custo para sua correção.

Tendo esses pontos em vista, como poderíamos unir a praticidade de uso dos modelos informais com a segurança e o ferramental presentes em modelos formais para verificar refinamentos em diagramas de sequência UML?

1.3 Justificativa

Inicialmente vendida como uma linguagem para especificação de modelos de software orientado a objeto (UML, 2015), hoje a UML é difundida como uma linguagem diagramática de propósito geral extremamente abrangente, com uma série de diagramas para diferentes fins. Entretanto, tal abrangência dificulta o nível de validações que podemos realizar nesses modelos devido a sua semântica informal, a qual é descrita textualmente em linguagem natural.

Com essas características em mente, planejamos realizar uma conversão de UML para a linguagem CSP (HOARE, 1985), com foco nos diagramas de sequência. Ao descrever a semântica de diagramas da linguagem UML usando uma notação como CSP estamos definindo uma semântica formal para um modelo informal. Dessa forma, além de poder definir modelos não ambíguos, podemos usar o poder ferramental de uma linguagem formal para realizar raciocínio lógico automatizado nos modelos.

A notação CSP fornece uma álgebra de processo para especificação de sistemas concorrentes, ela possui uma semântica formal não ambígua e mecanismos de cálculo de refinamento. Além disso possui uma série de operadores que facilitam a definição de uma semântica composicional para os elementos da UML. Possui também modelos semânticos compatíveis com a semântica de diagramas de sequência, a qual é baseada em sequência de eventos, também conhecida como *traces*. Acima de tudo isto possui raciocínio automatizado através da ferramenta FDR (GIBSON-ROBINSON et al., 2014).

O foco deste trabalho é utilizar verificação de refinamento presente no CSP para melhorar o processo de detecção de inconsistências durante a modelagem de diagramas de sequência de um sistema. Para tanto, vislumbramos adaptar semânticas formais definidas anteriormente em (LIMA; IYODA; SAMPAIO, 2016).

Tal trabalho serve como base para a criação de uma estratégia de tradução de diagramas de sequência para a notação CSP. Usando a ferramenta FDR conseguimos verificar refinamentos de tais especificações baseando-se nas noções de refinamento de diagrama de sequência definidas em (LIMA; IYODA; SAMPAIO, 2016). O cenário ideal é unir os dois mundos, onde o projetista pode focar no nível UML e utilizar uma ferramenta formal para realizar verificações de refinamento.

1.4 Objetivos

1.4.1 Objetivo Geral

Expandir o arcabouço de verificação de refinamento e tradução de diagramas de sequência visando o desenvolvimento de uma ferramenta com foco em minimizar o esforço com validações neste tipo de diagrama e permitir a detecção de falhas de forma antecipada, assim, reduzindo custos.

1.4.2 Objetivos Específicos

1. Extender o módulo de tradução de diagramas de sequência desenvolvido em trabalhos anteriores visando dar a suporte a tradução de fragmentos combinados.

2. Extender o módulo de verificação de refinamentos de diagramas de sequência desenvolvido em trabalhos anteriores visando melhorar a rastreabilidade de falhas de refinamento.

3. Desenvolver uma ferramenta no formato de plugin da plataforma Astah (CHANGE-VISION, 2018), capaz de verificar refinamentos entre diagramas de sequência.

1.5 Metodologia

Como citado anteriormente nos objetivos, o desenvolvimento desta pesquisa está atrelado ao desenvolvimento de uma ferramenta no formato de plugin para a plataforma Astah. Todas as etapas de desenvolvimento da ferramenta serão realizadas com a linguagem Java (GOSLING; HOLMES; ARNOLD, 2005), utilizando como base os diagramas criados na plataforma Astah. Este projeto de pesquisa será desenvolvido seguindo as seguintes etapas:

1. **Adaptar semântica CSP para tradução de diagramas de sequência:** Nesta pesquisa adaptaremos uma semântica formal definida em trabalhos anteriores, essas adaptações têm o intuito de facilitar a utilização da semântica no contexto de uma ferramenta.
2. **Desenvolver um tradutor UML $\rightarrow$ CSP:** Essa é a primeira etapa do desenvolvimento da ferramenta, aqui transformaremos um diagrama de sequência UML em uma especificação CSP com a semântica definida na etapa anterior.
3. **Integrar verificador FDR à ferramenta:** Após a tradução dos diagramas poderemos utilizar as funcionalidades do FDR para realizar verificações de refinamento nas especificações CSP geradas.
4. **Desenvolver um tradutor de contraexemplos:** Em casos de falha de refinamento, a ferramenta FDR retorna um contraexemplo que identifica onde houve a falha. Para exibir a falha de forma intuitiva precisamos transformá-la em um diagrama de sequência e destacar o elemento que causou a inconsistência.
5. **Integração da ferramenta como plugin Astah:** Integrar todos os elementos citados anteriormente em uma ferramenta única, no formato de plugin da plataforma Astah.
6. **Realização de estudos de caso:** Validar a ferramenta criada com diferentes estudos de caso.

1.6 Estrutura do Trabalho

Além desse capítulo introdutório, este trabalho apresenta mais quatro capítulos. O Capítulo 2 apresenta o embasamento teórico necessário para o desenvolvimento deste trabalho, nele são contemplados os principais conceitos relacionados a diagramas de sequência e à linguagem CSP.

O Capítulo 3 apresenta os resultados obtidos na pesquisa, exemplificando todo o processo de tradução e seus resultados.

O Capítulo 4 apresenta um estudo de caso da aplicação da ferramenta desenvolvida neste trabalho, nele são contemplados diversos exemplos do uso da ferramenta no contexto de um projeto real.

Por fim, o Capítulo 5 contém a conclusão do trabalho, nele são apresentadas todas as contribuições, limitações e dificuldades do trabalho, além de comentar trabalhos relacionados e possíveis trabalhos futuros na área de pesquisa.

2 Fundamentação Teórica

Esse capítulo tem o intuito de apresentar conceitos fundamentais ao desenvolvimento deste trabalho. As seções a seguir irão fornecer o embasamento teórico necessário para o entendimento de diagramas de sequência UML e da linguagem CSP.

A primeira seção aborda os diagramas de sequência, nela serão contemplados todos os elementos de diagramas utilizados neste trabalho. Todos os exemplos serão ilustrados com diagramas construídos na plataforma Astah, que é a plataforma alvo da ferramenta desenvolvida neste trabalho.

A segunda seção aborda a linguagem CSP com foco em seus operadores e modelo de traces, que é o modelo utilizado para verificações. Nela serão contemplados todos os operadores utilizados neste trabalho.

2.1 Diagramas de Sequência

A linguagem UML possui uma série de diagramas de diferentes tipos e funções, entre os tipos de diagrama podemos identificar duas grandes categorias: diagramas estruturais e diagramas comportamentais. Nesta pesquisa iremos focar em diagramas de sequência, um dos principais diagramas comportamentais.

Os diagramas comportamentais, assim como sugere sua classificação, tem foco em representar o comportamento de um sistema através da representação de interações. O diagrama de sequência representa a interação entre entidades de um sistema durante sua execução, essas interações são representadas através de trocas de mensagem ao longo do tempo.

A seguir apresentaremos os elementos básicos que compõem um diagrama de sequência. Após a introdução dos elementos básicos iremos apresentar elementos mais complexos que são o foco deste trabalho.

2.1.1 Linhas de vida

As linhas de vida ou *lifelines* representam as entidades presentes em um diagrama de sequência e suas interações ao longo do tempo. Tais interações são representadas ao longo do eixo vertical de uma linha de vida, esse eixo representa a passagem de tempo, onde o ponto mais alto do eixo representa o inicio da interação e a base do eixo representa o fim da interação. Dependendo do contexto que o diagrama representa, uma linha de vida pode representar diferentes elementos. Considerando o

contexto de um sistema de software, uma linha de vida pode ser utilizada para representar as entidades do sistema, tal como classes e suas respectivas instâncias, além disso também é possível representar elementos mais complexos como um sistema externo ou até mesmo atores do processo representado. Na Figura 2 destacamos as linhas de vida presentes em um diagrama de sequência simples.

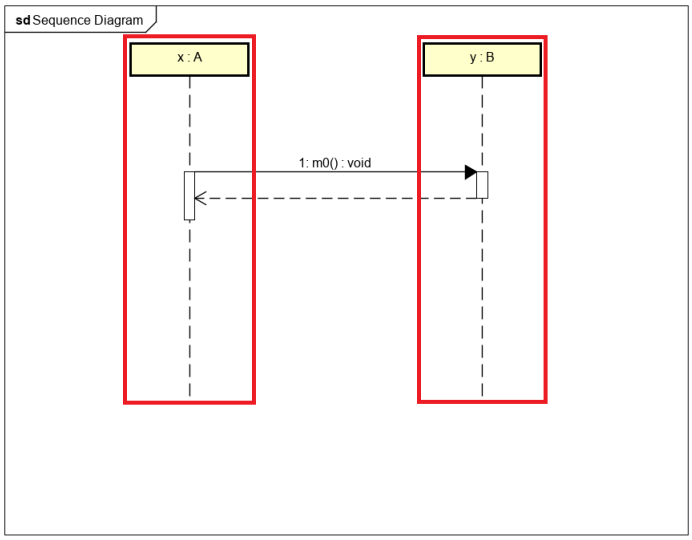

Figura 2 – Linhas de vida de um diagrama de sequência

De forma geral podemos afirmar que as linhas de vida representam as entidades que se comunicam no modelo em questão.

2.1.2 Mensagens

As mensagens do diagrama de sequência representam a comunicação entre as suas entidades. Toda mensagem deve ligar uma linha de vida a outra ou a ela mesma, e a sua disposição na vertical representa a ordem que irá acontecer. A ordem do envio de mensagens é definida de cima para baixo, sendo a mensagem mais acima a primeira que será enviada. Uma mensagem pode ser dividida em dois eventos: envio e recebimento.

No contexto de um sistema de software as mensagens podem representar a chamadas de função em diferentes entidades do sistema. Um dos conceitos mais importantes da mensagem é o seu significado semântico. As mensagens em diagrama de sequência podem ser representadas de duas formas :

- **Mensagem Síncrona:** Esse tipo de mensagem representa a comunicação síncrona entre linhas de vida, ou seja, em uma mensagem síncrona a linha de vida que envia a mensagem espera uma resposta para continuar sua execução.Esse tipo de mensagem é comumente utilizada para representar chamadas de méto-

dos que possuem algum retorno. O retorno de uma mensagem síncrona é representada na forma de uma mensagem de resposta.Em um diagrama de sequência a mensagem síncrona é representada por uma seta com ponta preenchida, enquanto que a sua mensagem de retorno é representada por uma seta tracejada.

- **Mensagem Assíncrona:** Diferente da mensagem síncrona, esse tipo de mensagem não precisa de uma resposta, uma linha de vida que comunica uma mensagem síncrona continua sua execução logo após o envio da mensagem. Esse tipo de mensagem é comumente utilizada para representar comunicação de sinais. Em um diagrama de sequência a mensagem assíncrona é representada por uma seta simples.

A Figura 3 destaca os diferentes tipos de mensagens presentes em um diagrama de sequência. A mensagem *m0* mais acima é síncrona e possui uma mensagem de retorno, enquanto que a mensagem *m1* é assíncrona.

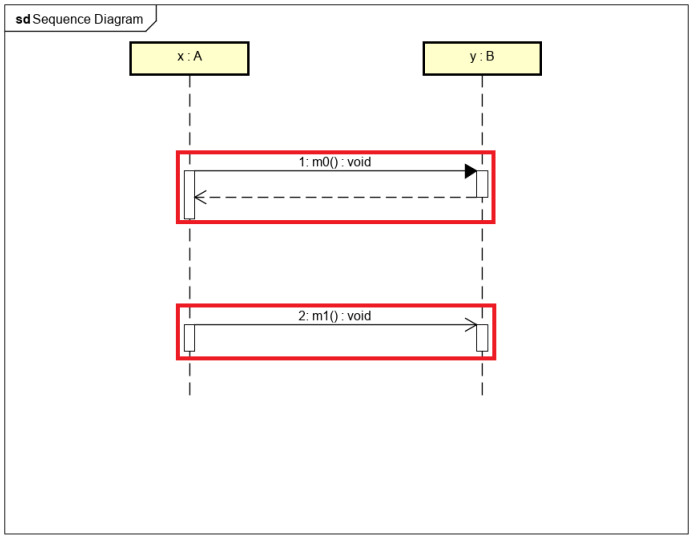

Figura 3 – Mensagens em um diagrama de sequência

2.1.3 Fragmentos Combinados

Os fragmentos combinados de um diagrama de sequência são uma série de operadores mais complexos que possuem diferentes significados semânticos. Com fragmentos combinados é possível representar operações presentes em linguagens de programação como laços de repetição, operações de escolha e até mesmo paralelismo.

Um fragmento combinado é definido por um operador de interação (*interaction Operator*) e pelos seus respectivos operandos (*interaction Operand*). O operador de

interação define qual o significado semântico do fragmento, define se o fragmento representa uma operação de loop, escolha, paralelismo, dentre outros. Cada fragmento possui um ou mais operandos, esses operandos são contêineres que agrupam interações que representam um cenário presente no fragmento.

A linguagem UML possui uma grande quantidade de fragmentos combinados para diagramas de sequência, neste trabalho iremos contemplar apenas os fragmentos relacionados a estruturas de controle, ou seja, fragmentos que alteram diretamente o fluxo de um diagrama de sequência.

2.1.3.1 Alternative - alt

O fragmento **alt** representa uma operação de escolha ou alternativas de comportamento. Nesse fragmento cada comportamento possível é representado por um operando de interação, tais operandos estão relacionados a uma guarda que representa a condição necessária para que o operando seja executado, em um **alt** apenas um operando pode ser escolhido. O **alt** possui significado semântico similar ao operador *if/else* presente em linguagens de programação.

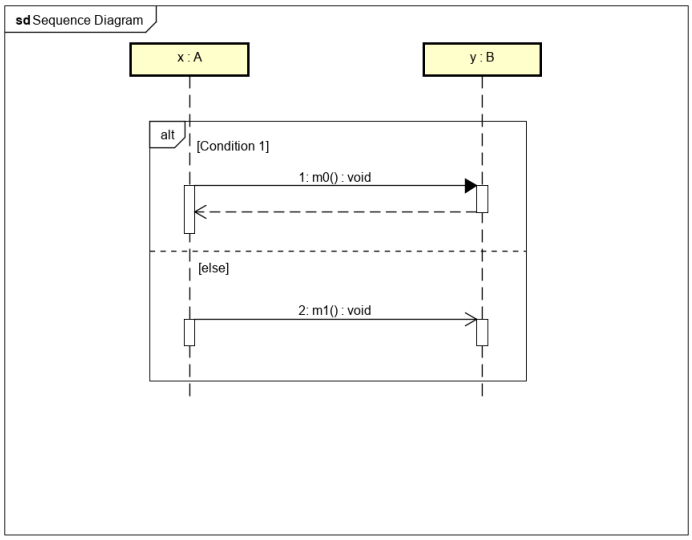

Figura 4 – Fragmento alt

A Figura 4 representa um fragmento **alt** com apenas dois operandos, sendo um deles relacionado à guarda *else*. Um operando com guarda *else* só será executado caso todos os operandos definidos anteriormente tiverem guardas avaliadas como falso, além disso, caso um operando possua uma guarda vazia a mesma será sempre avaliada como verdadeira.

2.1.3.2 Option - opt

O fragmento **opt** é muito parecido com o **alt**, o que os diferencia é a quantidade alternativas de comportamentos, enquanto o alt deve conter dois ou mais operandos, o **opt** possui apenas um. O **opt** possui o mesmo significado semântico que um operador *if* presente em linguagens de programação, e no contexto do UML possui o mesmo significado semântico que um **alt** com apenas um operando não vazio. A Figura 5 representa um fragmento **opt** que executa a mensagem *m0* caso sua guarda seja avaliada como verdadeira.

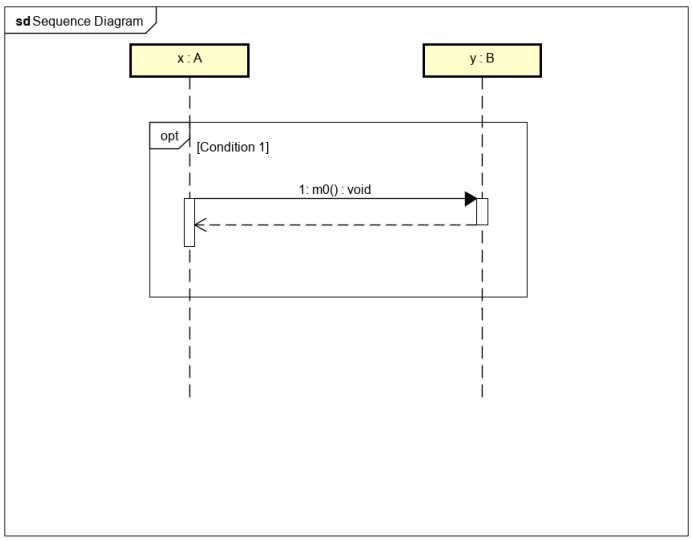

Figura 5 – Fragmento opt

2.1.3.3 Parallel - par

O fragmento *parallel* representa a execução paralela de comportamentos. Nesse fragmento cada operando representa um fluxo do paralelismo,ou seja, todos os operandos então em paralelo e podem ser executados em qualquer ordem. Apesar da ordem de execução dos operandos ser variável, o conteúdo de cada operando ainda deve seguir a ordem de execução que foi definida. O **par** pode ser utilizado para representar uma série de operações independentes que devem ser realizadas mas que não se restrigem a uma ordem de execução específica.

A Figura 6 representa um fragmento **par** com três operandos, cada um desses operandos será executado independentemente de forma paralela.

2.1.3.4 Loop - loop

O fragmento *loop* representa um loop de repetição. Esse fragmento possui apenas um operando que será repetido um número de vezes. Uma das peculiaridades do

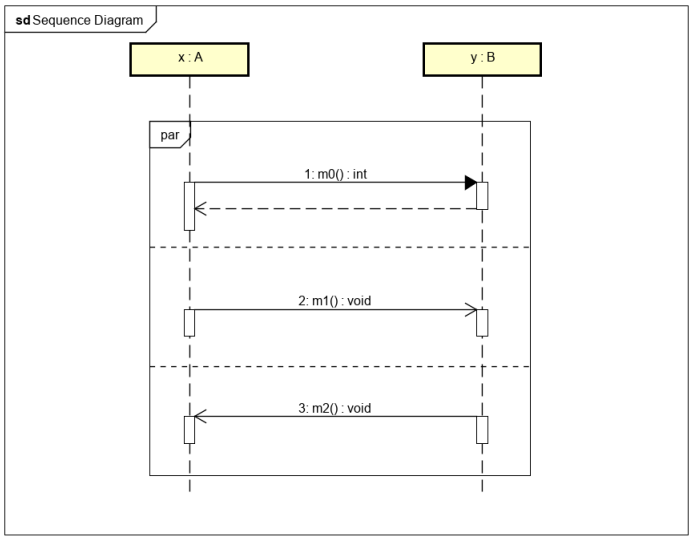

Figura 6 – Framento par

loop é a forma de limitar seu número de iterações, segundo a (UML, 2015), os limites do loop podem ser definidos da seguinte forma:

$$loop(min - int[, max - int])$$

Onde *min-int* representa um limite inferior de iterações e *max-int* representa o limite superior de iterações. Dada essa definição, é possível identificar três formatos distintos de loop:

- **Loop sem limite definido:** Caso não sejam especificados *min-int* e *max-int*, o loop será considerado como infinito. No diagrama ele será representado como $loop$ sem nenhum parêntese ou parâmetro.
- **Loop com apenas um limite definido:** Caso o loop só possua um limite definido, apenas o limite inferior será considerado, ou seja, o limite vai representar o número exato de repetições do loop. No diagrama esse tipo de loop será representado como $loop(x)$ onde *x* é um número inteiro.
- **Loop com dois limites definidos:** Caso o loop possua limite inferior e superior definidos ele será repetido pelo menos o número de vezes definido no limite inferior, e no máximo o número de vezes definido no limite superior. No diagrama esse tipo de loop vai ser representado por $loop(x, y)$ onde *x* e *y* são números inteiros.

A Figura 7 ilustra um diagrama com os diferentes tipos de fragmento *loop*.

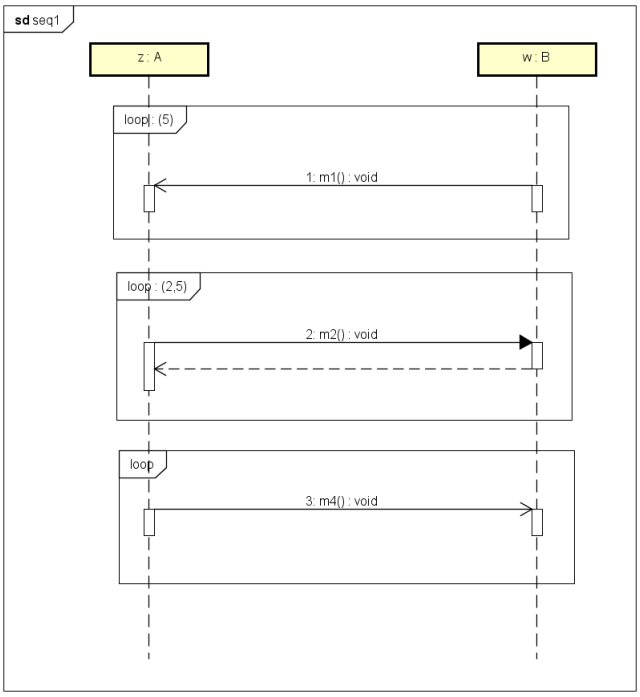

Figura 7 – Diagrama com fragmentos loop

2.2 CSP

A linguagem CSP é uma álgebra de processos comumente utilizada para representar sistemas constituídos por componentes que interagem entre si. CSP também pode ser utilizada para representar sistemas concorrentes devido a sua variedade de operadores e os tipos de verificação possíveis de realizar, tais verificações permitem identificar comportamentos como *deadlock*, não-determinismo e *livelock*. Além dessas características, o ferramental existente para a linguagem foi um dos pontos importantes para a sua escolha como domínio semântico para diagramas de sequência. O verificador de modelos FDR é a principal ferramenta da linguagem CSP, com ela é possível verificar diversos tipos de refinamento através de uma interface iterativa. A seguir serão apresentadas as principais características da linguagem CSP.

2.2.1 Processos e eventos

Um processo CSP é a unidade básica de descrição de comportamento, ele é definido em termos de eventos ou de outros processos. Eventos por sua vez representam alguma ação existente no processo, como por exemplo a comunicação de algum dado. Processos também podem conter diversos operadores do CSP. Tais operadores possuem diferentes semânticas que podem representar desde operações de escolha até paralelismo de processos.

A linguagem apresenta dois processos básicos, *SKIP* e *STOP*. O processo de

SKIP representa um término com sucesso, é um processo que comunica o evento (tick) e termina. Por sua vez o processo *STOP* representa o término abrupto de um processo,ou seja, um *deadlock* canônico.

$$ProcessA = event1 \rightarrow SKIP$$

$$ProcessB = event1 \rightarrow STOP$$

O exemplo acima possui dois processos: *ProcessA* e *ProcessB*,e ambos os processos comunicam um evento *event1*. A principal difereça entre esses processos são suas terminações, o primeiro processo termina com sucesso (*SKIP*) enquanto que o segundo entra em *deadlock* (*STOP*). Por convenção os eventos são representados por letras minúsculas e os processos por letras maiúsculas.

2.2.2 Tipos de dados e Canais

Um canal define um conjunto de eventos que podem ser realizado pelos processos. É possível definir um canal que comunica diversos tipos de valores. Os tipos de dados presentes no CSP são similares aos encontrados em linguagem de programação. A linguagem possui os tipos primitivos básico como inteiros, caracteres e booleanos. Além dos tipos primitivos também é possível definir conjuntos, sequências (listas), tuplas e tipos algébricos. Esses últimos podem compor diversos outros tipos em um só.

$$datatype\ COM = s \mid r$$

$$channel\ MSG1 : COM$$

$$channel\ MSG2 : COM.COM$$

Neste examplo é definido um tipo algébrico *COM* que é composto pelos elementos ”s”e ”r”. A seguir é definido um canal *MSG1* que comunica *COM*,ou seja, comunica os valores ”s”e ”r”. Os canais podem comunicar a composição de *datatypes*, então neste caso o canal *MSG2* pode comunicar qualquer combinação de dois elementos do *datatype COM*.

Além disso, os canais também podem ser utilizados como mecanismos de sincronização entre processos. Na próxima sub-seção deste capítulo serão visto os operadores de paralelismo presentes em CSP, lá serão exemplificados os diferentes caso de uso de canais para sincronização de processos. Linguagens de programação modernas como Go (DONOVAN; KERNIGHAN, 2015) utilizam estrutura de canais similares para sincronização em situações de paralelismo.

2.2.3 Operadores

Aqui iremos descrever apenas os operadores utilizados ao longo deste trabalho.

2.2.3.1 Operador prefixo

O operador prefixo é representado por "$\rightarrow$ ". Esse operador é utilizado para modelar comportamento sequencial entre processos e eventos.

$$S = a \rightarrow P$$

Neste exemplo o processo *S* espera indefinidamente pelo evento "*a*" e depois se comporta como o processo *P*.

2.2.3.2 Escolha externa

O operador de escolha externa é representado por " [ ] " ou por "□ ". Esse operador é utilizado para representar a decisão do ambiente. Quando dois processos são colocados em escolha externa, apenas um deles será executado enquanto o outro será ignorado.

$$S = (s \rightarrow P)\ [\,]\ (r \rightarrow Q)$$

Neste exemplo apenas um dos lados do processo *S* será executado. A primeira opção é comunicar o evento *s* e então seguir para o processo *P*, ou comunicar o evento *r* e seguir para o processo *Q*. A Figura 8 ilustra o processo *S* de forma mais intuitiva.

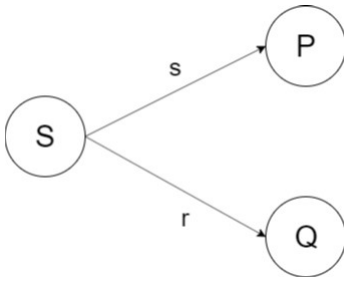

Figura 8 – Exemplo de escolha externa

2.2.3.3 Composição sequencial

O operador de composição sequencial é representado por " ; ". Esse operador é utilizado para compor processos de forma sequêncial. Ao utilizar o operador de composição sequencial entre dois processos estamos sinalizando que o segundo processo será executado assim que o primeiro terminar com sucesso, ou seja, assim que o primeiro processo se comportar como SKIP.

$$S = P \; ; \; Q$$

Neste exemplo o processo *S* irá iniciar o processo *P*, assim que *P* terminar sua execução com sucesso o processo *Q* será iniciado.

2.2.3.4 Entrelaçamento

O operador de entrelaçamento ou *interleaving*, é representado por " $|||$ ". Esse operador é utilizado para realizar a composição paralela de processos. Compor dois processos independentes com *interleaving* significa que esses processo irão executar paralelamente sem sincronizar em nenhum de seus eventos.

$$S = P \; ||| \; Q$$

Neste exemplo o processo *S* irá iniciar os processos *P* e *Q*, que serão executados de forma independente sem nenhuma sincronização de eventos.

2.2.3.5 Composição paralela generalizada

O operador de composição paralela generalizada é representado por "$[|X|]$" onde X representa um conjunto de eventos. Compor dois processos com esse operador significa que esses processos irão executar paralelamente sincronizando nos eventos do conjunto X.

$$S = P \; [\; |\{s, x, r\}| \;] \; Q$$

Neste exemplo o processo *S* inicia os processo *P* e *Q* de forma paralela. Esses processos serão executados paralelamente até que um deles tenha que comunicar algum evento do conjunto de sincronização, nesse caso o conjunto $\{s, x, r\}$. Ao atingir um evento de sincronização o processo irá parar e esperar que o outro chegue no mesmo evento, após a sincronização os eventos podem continuar paralelamente até atingir outro evento de sincronização. Eventos fora do conjunto de sincronização são livres para serem comunicados.

2.2.3.6 Composição paralela alfabetizada

O operador de composição paralela alfabetizada é representado por "$[X \parallel Y]$" onde X e Y representam conjuntos de eventos. Compor dois processos com esse operador significa que esses processos irão executar paralelamente sincronizando nos eventos comuns de X e Y, qualquer evento fora desses conjuntos será bloqueado.

$$S = P\ [\ \{|s, x|\}\ \|\ \{|x, r|\}]\ Q$$

Neste exemplo o processo *S* inicia os processos *P* e *Q* de forma paralela. Esses processos comunicam os conjuntos de evento {s,x} e {x,r} respectivamente. Devido à composição paralela alfabetizada, esses processos irão sincronizar na interseção dos seus conjuntos de evento, neste caso irão se sincronizar apenas no evento *x*.

2.2.3.7 Esconder

O operador de esconder ou *hiding* é representado por " \". Esse operador tem a funcionalidade de esconder um conjunto de eventos de um processo, os eventos escondidos não poderão ser observados e não serão considerados em nenhuma verificação.

$$S = (a \rightarrow b \rightarrow c \rightarrow SKIP)\ \backslash\{b, c\}$$

Neste exemplo o processo *S* tem os eventos *b* e *c* escondidos, tais eventos não poderão ser utilizados em nenhuma verificação com o processo *S*.

2.2.4 Modelo Semânticos e Traces

Em CSP, um modelo semântico se refere a uma representação abstrata do comportamento dos processos. Existem diversos modelos semânticos em CSP mas neste trabalho será o usado o modelo de *trace*. Um *trace* nada mais é do que uma sequência de eventos presentes em um processo. Dado um processo *P*, *Traces(P)* representa o modelo de traces desse processo, ou seja, representa o conjunto com todas as combinações possíveis de eventos presentes nesse processo.

$$P = x \rightarrow y \rightarrow z \rightarrow STOP$$

$$Traces(P) = \{<>, < x >, < x, y >, < x, y, z >\}$$

Como pode ser visto no exemplo acima, o modelo de *traces* é representado por um conjunto de possíveis eventos de um processo, esse foi um dos principais motivos que influenciaram a utilização da linguagem CSP e especificamente do modelo de *traces* para este trabalho. Quando observamos um diagrama de sequência podemos facilmente identificá-lo como um processo. Suas mensagens representam eventos que devem ser executados em ordem e existem elementos que podem realizar alterações no seu fluxo, esse tipo de comportamento pode ser facilmente modelado no contexto de *traces*.

Os modelos semânticos possibilitam os conceitos de **igualdade** e **refinamento**. Considerando o modelo de *traces*, dado dois processos *P* e *Q* a noção de **igualdade** entre processo é dada por:

$$P \equiv_{\mathsf{T}} Q$$

$$Traces(P) = Traces(Q)$$

Que pode ser interpretado como:

- *P* tem os mesmo traces que *Q*.
- *P* é igual a Q no modelo semântico de traces.

Por sua vez, a noção de **refinamento** é dada por:

$$P \sqsubseteq_{\mathsf{T}} Q$$

$$Traces(Q) \subseteq Traces(P)$$

Que pode ser interpretada como:

- *P* é refinado por Q no modelo semântico de traces.
- *Q* refina *P* no modelo semântico de traces.

As verificações de refinamento e igualdade nem sempre serão verdadeiros para quaisquer processos *P* e *Q*. Em casos de falha é possível identificar ao menos um contraexemplo que invalida a verificação. No contexto da verificação de refinamento, um contraexemplo é um *trace* que não cumpre as definições estabelecidas anteriormente, ou seja, não cumpre $Traces(Q) \subseteq Traces(P)$.

3 Verificação de Refinamento de Diagramas de Sequência

Este capítulo apresenta os detalhes da elaboração e implementação de uma solução para verificação de refinamento de diagramas de sequência UML considerando operadores de controle, aqui serão apresentadas todas as etapas do processo de formalização e verificação de refinamento.

Neste trabalho serão utilizadas as noções de refinamento introduzidas em (LIMA; IYODA; SAMPAIO, 2016). Nesse trabalho são definidos elementos semânticos importantes para tradução de UML para CSP, utilizaremos a semântica e noções de refinamento desse trabalho relacionado como base ao longo do processo de tradução e verificação, realizando adaptações quando necessário.

3.1 Visão Geral do Processo de Verificação de Refinamento

Assim como foi dito nos capítulos anteriores, a proposta deste trabalho é construir uma ferramenta capaz de realizar verificação de refinamento em diagramas de sequência com suporte a operadores de controle. Tendo isso em mente, o processo de verificação pode ser resumido nas seguintes etapas:

1. **Criar diagramas de sequência:** Os diagramas de sequência são a base de todo o processo, visto que eles serão o alvo da verificação de refinamentos. A ferramenta Astah-UML (CHANGE-VISION, 2018) foi escolhida como plataforma alvo do projeto devido à sua grande comunidade, boa usabilidade, versão gratuita para estudantes e ao seu suporte a plugins e API Java.

2. **Traduzir diagrama de sequência para CSP:** Uma vez que temos uma ferramenta para criar diagramas de sequência agora podemos elaborar uma estratégia de tradução para a linguagem CSP. Para realizar a tradução dos diagramas foi desenvolvido um tradutor na linguagem Java, visto que com Java podemos acessar a API do Astah e consequentemente conseguimos manipular seus diagramas fora do editor da plataforma.

3. **Verificar Refinamento:** Ao final da etapa de tradução será gerada uma especificação CSP que representa os diagramas que foram traduzidos. Agora que temos uma especificação CSP devidamente criada podemos utilizar a ferramenta FDR

(GIBSON-ROBINSON et al., 2014) para realizar uma verificação de refinamento automatizada.

4. **Detecção de Contraexemplo:** Em caso de falha na verificação de refinamento precisamos identificar qual parte da especificação CSP gerada na etapa de tradução está impossibilitando o refinamento. Para isso utilizamos o FDR para identificar qual o *trace* da especificação causou a falha, esse *trace* é denominado contraexemplo.

5. **Rastreabilidade de Contraexemplo:** Após identificar o contraexemplo na especificação CSP devemos rastrear o evento até a sua implementação nos diagramas de sequência da etapa 1. Tendo identificado o elemento do diagrama relacionado com o *trace* de contraexemplo podemos destacá-lo para que o usuário o identifique facilmente.

A Figura 9 ilustra as etapas do processo de verificação de refinamento.As seções a seguir descrevem cada etapa do processo de verificação de refinamento em detalhes.

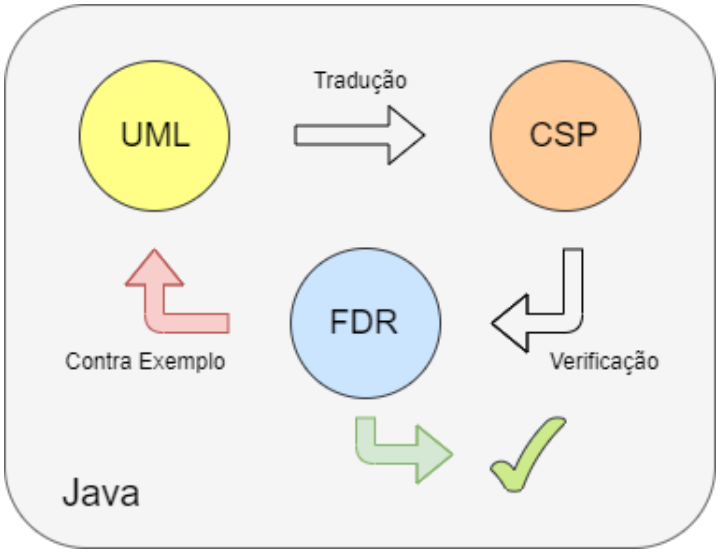

Figura 9 – Visão Geral do Processo de Verificação de Refinamento

3.2 Tradução

A etapa de tradução é a base da verificação de refinamento, nessa parte do processo é dada como entrada dois diagramas de sequência que vão gerar duas especificação CSP como saída. Para realizar a etapa de tradução foi desenvolvido um módulo tradutor em Java que utiliza a API da ferramenta Astah para acessar as propriedades dos diagramas de sequência, uma vez que temos acesso a elementos do

diagrama podemos utilizar a semântica de tradução definida em (LIMA; IYODA; SAMPAIO, 2016) para elaborar uma estratégia de tradução.

Ao longo da tradução de um diagrama são construídos diversos processos CSP que são compostos paralelamente para formar um processo único que representa todo o diagrama de sequência. Esses processos menores são referentes a elementos do diagrama como as linhas de vida e processos auxiliares para representar o ambiente de trocas de mensagem (*MessagesBuffer*).

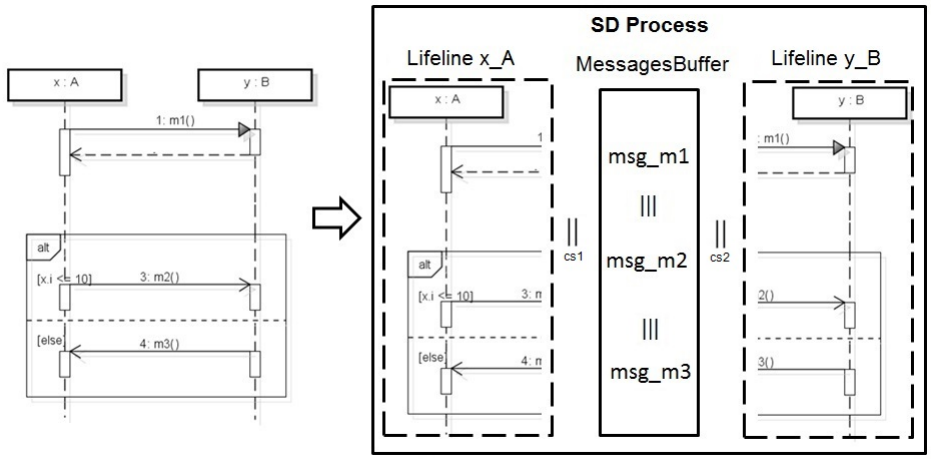

Figura 10 – Exemplo do *MessagesBuffer*

A Figura 10 ilustra a utilização de paralelismo na tradução de diagramas de sequência para CSP. Nela podemos observar a presença do *MessagesBuffer*, um processo utilizado para sincronizar a ordem de execução das mensagens, mais detalhes sobre a tradução de mensagens e processos auxiliares como o *MessagesBuffer* serão vistas ao longo do Capítulo.

A seguir vamos apresentar um diagrama de sequência simples e demonstrar todas as etapas do processo de tradução. Para que o tradutor funcione corretamente o diagrama a ser traduzido deve cumprir os seguintes pré-requisitos:

- Todas as linhas de vida devem estar devidamente nomeadas.
- Todas as linhas de vida devem representar instâncias de uma classe previamente definida no projeto do diagrama.
- Todas as mensagens entre linhas de vida devem estar devidamento nomeadas.
- Todas as mensagens entre linhas de vida devem referenciar operações presentes nas classes definidas no projeto do diagrama.
- Todas as mensagens síncronas devem possuir mensagem de retorno.

Essas restrições são importantes para garantir um diagrama de sequência bem formado. Tendo em vista esses requisitos, podemos utilizar o diagrama da Figura 11 como exemplo para a tradução.

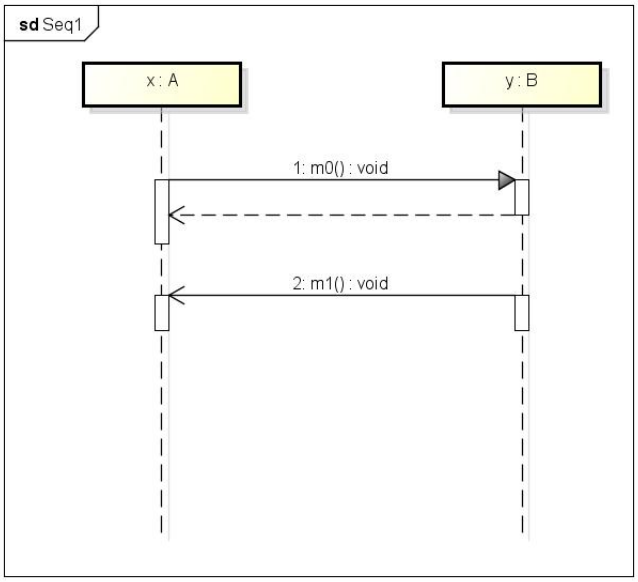

Figura 11 – Diagrama de sequência para tradução

O diagrama da Figura 11 contém apenas duas linhas de vidas representadas por x e y, essas linhas de vida são instância das classes A e B respectivamente. O diagrama possui três mensagens, sendo uma mensagem síncrona *m0*, sua mensagem de retorno e uma mensagem assíncrona *m1*. Todos os elementos do diagrama estão devidamente nomeados e, apesar de não ser possível visualizar na imagem, estão associados a uma classe ou operação definida no projeto do diagrama, sendo assim é possível realizar a tradução.

A tradução de diagramas para CSP segue as seguintes etapas:

1. Definição de Tipos de dados
2. Definição de canais de comunicação
3. Construção dos processos de linhas de vida e mensagem
4. Tradução de fragmentos combinados
5. Paralelismo e sincronização de mensagens

3.2.1 Definição de tipos de dados

Durante essa etapa o tradutor analisa todas as linhas de vida do diagrama, verificando a quantidade de linhas de vida e suas respectivas classes para que sejam

definidos tipos e subtipos de dados CSP. Como foi visto no Capítulo 2, esses tipos de dados estão associados aos canais de comunicação e aos eventos que eles comunicam. Enquanto verifica todas as linhas de vida do diagrama, o tradutor também verifica as mensagens para que sejam atribuídas aos tipos de dados CSP de suas respectivas linhas de vida e para verificar quais os tipos de parâmetros comunicados no diagrama. A Figura 12 destaca os elementos do diagrama de sequência que são utilizados na definição de tipos de dados.

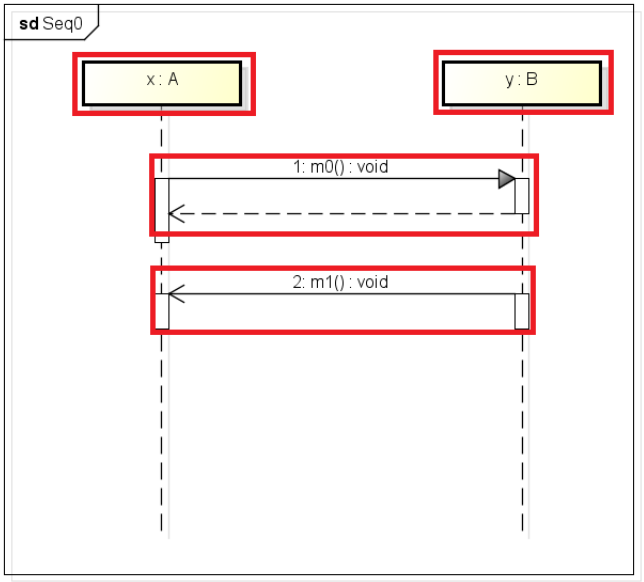

Figura 12 – Elementos utilizados na definição de tipos de dados

A definição de dados do diagrama em questão é traduzido no trecho da especificação CSP ilustrada pela Figura 13.

```
datatype COM = s | r
datatype ID = lf1id|lf2id
datatype ID_SD = sd1id
datatype MSG = m1|m0_I|m0_O
subtype A_SIG = m1
subtype B_OPS = m0_I | m0_O
get_id(m1) = m1
get_id(m0_I) = m0_I
get_id(m0_O) = m0_O
```

Figura 13 – Tipos de dados traduzidos para CSP

Na Figura 13 inicialmente são criados tipos identificadores e modificadores de mensagem. O tipo *COM* é utilizado para representar se uma mensagem está sendo

enviada ou recebida, onde o envio é representado por *"s"* e o recebimento é representado por *"r"*. Os tipos *ID* e *ID_SD* provém identificadores únicos para linhas de vida e diagrama de sequência, *"lf1id"* representa a primeira linha de vida e *"lf2id"* representa a segunda, *"sd1id"* por sua vez identifica o diagrama de sequência como todo.

O diagrama exemplo possui três mensagens, o tipo *MSG* é utilizado para identificá-las. Neste caso como uma das mensagens é síncrona, são criadas dois identificadores, um para representar a mensagem enviada (*m0_I*), e outro para identificar a mensagem de retorno (*m0_O*). As mensagem assíncronas são identificadas pelo nome da mensagem no diagrama de sequência, nesse caso apenas a mensagem *m1* é assíncrona.

Durante a identificação dos tipos de mensagens também são criados subtipos que agrupam as mensagem como operações ou sinais. No contexto da tradução de diagramas consideramos as mensagens síncronas como operações e as mensagens assíncronas como sinais, por isso são criados os subtipos como *A_SIG* e *B_OPS*. O subtipo *A_SIG* contém todos os identificadores de mensagens assíncronas enviadas para linhas de vida que são instâncias da classe *A*, o subtipo *B_OPS* por sua vez contém todos os identificadores relacionados a mensagens síncronas enviadas para linhas de vida que são instâncias da classe *B*.

Ao final das definições de tipos também são criadas funções auxiliares como *"get_id"* que retornam os identificadores das mensagens do diagrama.

3.2.2 Definição de canais de comunicação

Uma vez que os tipos de dados necessários foram definidos, o tradutor pode começar a etapa de definição de canais. Como foi visto no Capítulo 2, os canais em CSP são utilizados para comunicar eventos, tais eventos podem comunicar diferentes dados, no caso dos diagramas de sequência os eventos irão representar a troca de mensagens entre linhas de vida, e os dados que serão comunicados são uma composição dos diferentes tipos de dados traduzidos na etapa anterior. A Figura 14 destaca os elementos do diagrama de sequência que são utilizados na definição dos canais de comunicação.

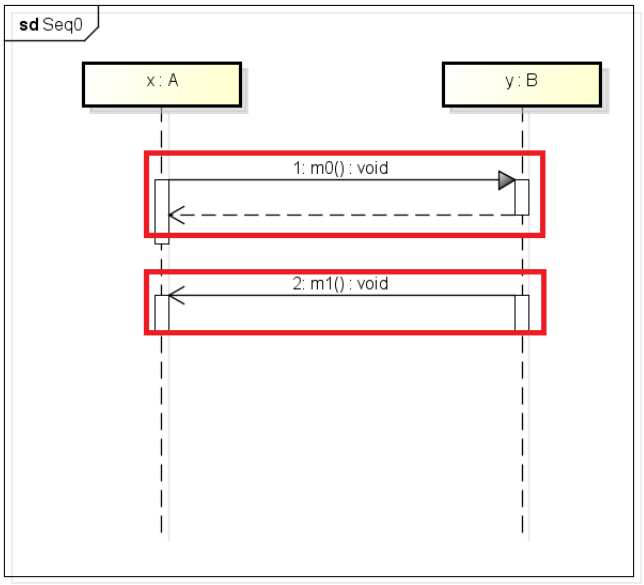

Figura 14 – Elementos utilizados na definição de canais

Como se pode ver na Figura 15, os canais são compostos por uma composição dos tipos de dados definidos anteriormente. O canal *"A_mSIG"* por exemplo vai comunicar todos os eventos relacionados a mensagem assíncronas enviadas para linhas de vida que são instâncias da classe *A*.

```
channel beginInteraction,endInteraction
channel A_mSIG: COM.ID.ID.A_SIG
channel B_mOP: COM.ID.ID.B_OPS
```

Figura 15 – Canais CSP

Quanto ao formato do evento que será enviado, podemos ver que ele obedece o padrão *"COM.ID.ID.A_SIG"*, onde *"COM"* representa se a mensagem está sendo enviada ou recebida, os campos *"ID"* representam as linhas de vida que estão enviando e recebendo a mensagem, respectivamente, e *A_SIG* representa os identificadores das mensagens assíncronas que são enviadas para A. Um evento válido no canal *A_SIG* poderia ter o seguinte formato : $A\_mSIG.s.lf1id.lf2id.m1$.

Além dos canais relacionados às mensagens do diagrama de sequência, também são criados os canais "*beginInteraction*"e "*endInteraction*"que serão utilizados posteriormente nas etapas de paralelismo e sincronização para representar o início e fim do fluxo de mensagens, respectivamente. Em diagramas mais complexos também são necessários outros canais auxiliares, como por exemplo para representar guardas em fragmentos combinados.

3.2.3 Construção dos processos de linhas de vida e mensagem

Após a construção dos canais de comunicação, o próximo passo do tradutor é criar os processos que vão representar as linhas de vida e as mensagens do diagrama. Para isso o tradutor começa a avaliar as linhas de vida individualmente, percorrendo-as verticalmente de cima para baixo, traduzindo todas as mensagens e fragmentos. Os processos criados são constituídos de eventos que representam as mensagens da linha de vida, todos os eventos são definidos na mesma ordem em que acontecem no diagrama de sequência. A Figura 16 destaca os elementos do diagrama de sequência que são utilizados na definição dos processos de linha de vida.

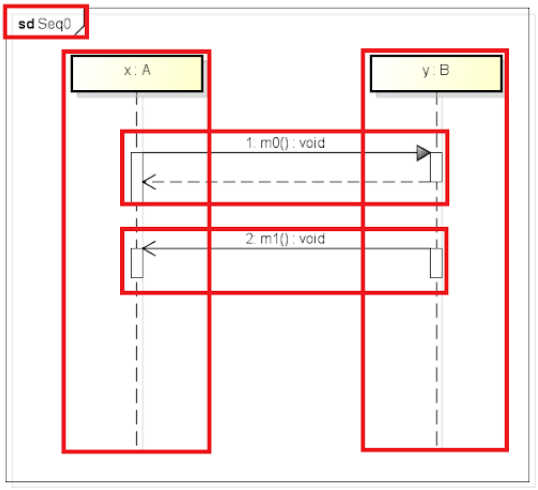

Figura 16 – Elementos utilizado na definição dos processos de linha de vida

A Figura 17 ilustra a especificação CSP gerada para as linhas de vida do diagrama em questão.

```
Seq0_x_A(sd_id,lf1_id,lf2_id) =(B_mOP.s!lf1_id!lf2_id.m0_I ->
SKIP);(B_mOP.r!lf2_id!lf1_id?out:{x | x <-B_OPS,(get_id(x) == m0_O)}
-> SKIP);(A_mSIG.r!lf2_id!lf1_id?signal:{x | x <- A_SIG,(get_id(x)
== m1)} -> SKIP)

Seq0_y_B(sd_id,lf1_id,lf2_id) =(B_mOP.r!lf1_id!lf2_id?oper:{x | x <-
B_OPS,(get_id(x) == m0_I)} -> SKIP);(B_mOP.s!lf2_id!lf1_id.m0_O ->
SKIP);(A_mSIG.s!lf2_id!lf1_id.m1 -> SKIP)
```

Figura 17 – Processos de linhas de vida em CSP

Os processos das linhas de vida são identificados pelo nome do diagrama, nome da instância da linha de vida e pela classe da linha de vida, no exemplo utilizado

o diagrama é identificado por *“Seq0*”, as instâncias das linhas de vida são identificadas por *“x”* e *“y”* e as classes por *“A”* e *”B”*. Cada processo de linha de vida contém todos os eventos de envio e recebimento de mensagens na mesma ordem descrita no diagrama.

Tomando como exemplo o processo *”Seq0_x_A”*, podemos observar que esse processo recebe três parâmetros. Um dos parâmetros é o identificador do diagrama de sequência e os outros dois representam os identificadores das linhas de vida que interagem ao longo do processo, nesse caso como as mensagens comunicadas por esse processo só estão relacionadas a duas linhas de vida , só foram necessários dois parâmetros. A utilização de parâmetros permite a execução de mais de uma instância do mesmo diagrama na especificação CSP gerada.

A tradução das mensagens é feita através da composição de canais de comunicação, assim como visto na Seção 3.2.2. Ainda considerando o processo *”Seq0_x_A”*, podemos observar que nele são comunicados uma série de eventos de mensagens. Inicialmente é comunicada a mensagem *m0* e o recebimento de sua respectiva mensagem de retorno, seguido do recebimento da mensagem assíncrona *m1*.

Observando o último campo do primeiro evento de mensagem podemos ver o identificador ”*m0_I*”, a terminação ”*_I*” representa que o evento é referente a uma mensagem síncrona, neste caso a mensagem ”*m0*” do diagrama exemplo. Similarmente, um dos eventos do mesmo processo possui um identificador com terminação ”*_0*”, essa terminação também está presente na tradução de mensagens síncronas mas é utilizado apenas em eventos relacionados a mensagens de retorno, nesse caso o retorno da mensagem ”*m0*”. As mensagens assíncronas não possuem nenhuma peculiaridade e são traduzidas sem nenhuma modificação em seus identificadores assim como exemplificado nas Seções anteriores.

Além de definir processos para as linhas de vida, essa etapa da tradução também cria um processo para cada mensagem do diagrama, esses processos são criados para que possam ser utilizados nas etapas de paralelismo e sincronização. A Figura 18 representa os processos CSP de todas as mensagens do diagrama exemplo.

Os nomes dos processos das mensagens possuem identificadores das instâncias e classes das linhas de vida que comunicam a mensagem, assim como o nome do diagrama.Considerando o processo *”Seq0_x_A_y_B_m0”* da Figura 18 temos *”Seq0”* como identificador do diagrama, *”x_A”* como identificador da linha de vida origem da mensagem, *”y_B”* como identificador na linha de vida destino da mensagem e “m0” como identificador da mensagem. Ainda observando a Figura 18 podemos identificar que cada processo possui parâmetros similares aos processos de linha de vida, e que eles comunicam os eventos de envio e recebimento de sua mensagem associada.

```
Seq0_x_A_y_B_m0(sd_id,lf1_id,lf2_id) =B_mOP.s.lf1_id.lf2_id?x:{x |
x<-B_OPS,get_id(x) == m0_I} -> B_mOP.r.lf1_id.lf2_id!x ->
Seq1_x_A_y_B_m0(sd_id,lf1_id,lf2_id)

Seq0_y_B_x_A_m1(sd_id,lf2_id,lf1_id) = A_mSIG.s.lf2_id.lf1_id?x:{x |
x<-A_SIG,get_id(x) == m1} -> A_mSIG.r.lf2_id.lf1_id!x ->
Seq1_y_B_x_A_m1(sd_id,lf2_id,lf1_id)

Seq0_y_B_x_A_m0_r(sd_id,lf2_id,lf1_id) = B_mOP.s.lf2_id.lf1_id?x:{x
| x<-B_OPS,get_id(x) == m0_O} -> B_mOP.r.lf2_id.lf1_id!x ->
Seq1_y_B_x_A_m0_r(sd_id,lf2_id,lf1_id)
```

Figura 18 – Processos de mensagens em CSP

Diferentemente dos processos de linha de vida que terminavam com um *SKIP*, os processos da mensagem reiniciam após a comunicação de seus eventos, isso acontece porque esses processos só serão utilizados como auxiliares na etapa de paralelismo e sincronização.

Esses processos de mensagem serão utilizados posteriormente na etapa de paralelismo e sincronização, mais especificamente no processo *MessagesBuffer*. Esse processo auxiliar realiza a composição paralela dos processos de mensagens do diagrama e tem o intuito de possibilitar a execução ordenada dos eventos que compõem os processos de mensagens.

A Figura 19 representa o processo *MesssagesBuffer* gerado para o diagrama exemplo da Figura 16. Nesse processo podemos observar a composição paralela dos processos de mensagens através do operador de entrelaçamento (*interleaving*), o uso desse operador implica no paralelismo completo dos processos associados.

```
Seq0_MessagesBuffer(sd_id,lf1_id,lf2_id) =
(Seq0_x_A_y_B_m0(sd_id,lf1_id,lf2_id)|||
Seq0_y_B_x_A_m0_r(sd_id,lf2_id,lf1_id)|||
Seq0_y_B_x_A_m1(sd_id,lf2_id,lf1_id))/\endInteraction ->
SKIP
```

Figura 19 – Processo *MessagesBuffer*

3.3 Tradução de Fragmentos Combinados

Ainda dentro da etapa de tradução de linhas de vida e mensagens está a tradução de fragmentos combinados. Alguns diagramas mais complexos possuem frag-

mentos combinados, esses elementos do diagrama são complexos o suficiente para precisarem de módulos tradutores específicos. O estado atual da ferramenta dá suporte a quatro fragmentos combinados, sendo eles: *alt*, *opt*, *par* e *loop*.

3.3.1 Tradução do fragmento alt

Como foi citado no Capítulo 2, o fragmento *alt* representa uma escolha para diferentes alternativas de comportamento. A Figura 20 ilustra um diagrama com um fragmento *alt*, no exemplo em questão o fragmento possui apenas duas alternativas, ou seja, dois operandos e duas guardas. O desafio em traduzir o fragmento *alt* está na avaliação das guardas e na construção de um processo que represente diferentes possibilidades de fluxo.

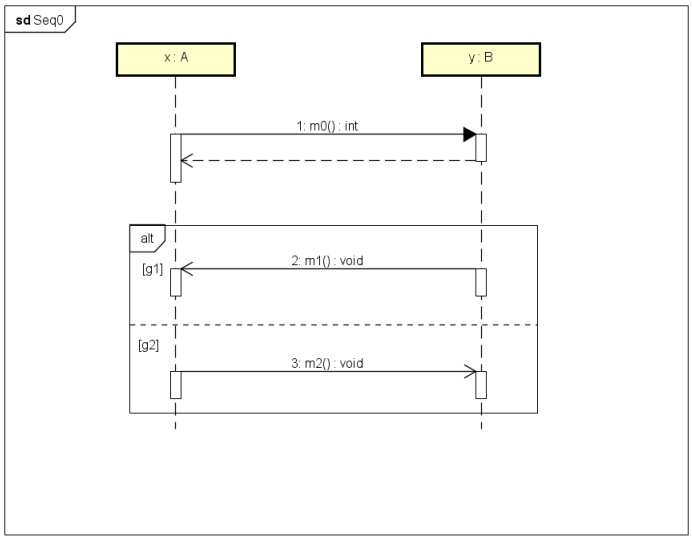

Figura 20 – Diagrama com fragmento *alt*

O primeiro passo para possibilitar a tradução do *alt* é definir canais extras que representam as avaliações das guardas presentes no fragmento, para o exemplo em questão o seguinte canal será criado:

$$channel\ alt1 : Bool.Bool$$

As guardas só podem ser avaliadas como *true* ou *false*, para isso são criados canais para cada *alt* e nele são adicionados um número de parametros booleanos equivalente ao número de guardas presentes no fragmento. Dessa forma é possível representar as diferentes possibilidades de fluxo dentro de um *alt*. No estado atual da ferramenta as guardas são definidas de forma abstrata utilizando apenas um identificador genérico ao invés de uma condição real. Isso acontece pois a ferramenta ainda não possui integração com uma semântica de definição de classes e parâmetros, essa é uma melhoria prevista para trabalhos futuros.

Após a definição dos canais auxiliares é possível construir um processo que represente o fragmento *alt*, para isso é utilizado o operador de escolha externa do CSP. A Figura 21 ilustra um processo gerado após a tradução do fragmento *alt* considerando a linha de vida da classe *A* presente na Figura 20. Na definição do processos são criados fluxos de execução correspondentes a cada um dos operandos presentes no fragmento, esses fluxos são compostos com o operador de escolha externa permitindo que apenas um seja executado.

```
alt1?g1?g2 -> (g1 & (A_mSIG.r!lf2_id!lf1_id?signal:{x | x <-
A_SIG,(get_id(x) == m1)} -> SKIP)
[]
g2 & (B_mSIG.s!lf1_id!lf2_id.m2 -> SKIP)
[]
(g1 == false and g2 == false) & SKIP)
```

Figura 21 – Tradução do fragmento *alt*

Na Figura 21 podemos observar as duas guardas *g1* e *g2* que foram utilizadas no diagrama exemplo, contudo essa não é a única forma de utilizar guardas no fragmento *alt*. Caso um operando não possua guarda definida, a mesma será implicitamente avaliada como verdadeira, outra forma de criar um operando que sempre terá uma guarda verdadeira é atribuindo o valor *else* à guarda.

Na prática, a tradução de um fragmento vai ser inserida no processo de cada linha de vida que é cruzada pelo mesmo, então a tradução das linhas de vida do exemplo em questão será representado pelo trecho ilustrado na Figura 22.

3.3.2 Tradução do fragmento opt

O fragmento *opt* possui um comportamento similar ao *alt*, porém se diferencia na quantidade de operandos, essa similaridade permite que a tradução do *opt* siga a mesma lógica do *alt*. A Figura 23 representa um diagrama de sequência com um fragmento *opt*.

Assim como na tradução do fragmento *alt*, é necessária a criação de canais extras para representar a avaliação da guarda do único operando do fragmento *opt*. Para o exemplo em questão será criado um canal simples :

$$channel\ opt1 : Bool$$

```
Seq0_x_A(sd_id,lf1_id,lf2_id) =(B_mOP.s!lf1_id!lf2_id.m0_I
-> SKIP);(B_mOP.r!lf2_id!lf1_id?out:{x | x
<-B_OPS,(get_id(x) == m0_O)} -> SKIP);
alt1?g1?g2 -> (g1 & (A_mSIG.r!lf2_id!lf1_id?signal:{x | x <-
A_SIG,(get_id(x) == m1)} -> SKIP)
[]
g2 & (B_mSIG.s!lf1_id!lf2_id.m2 -> SKIP)
[]
SKIP)

Seq0_y_B(sd_id,lf1_id,lf2_id)=(B_mOP.r!lf1_id!lf2_id?oper:{x
| x <- B_OPS,(get_id(x) == m0_I)} ->
SKIP);(B_mOP.s!lf2_id!lf1_id.m0_O -> SKIP);
alt1?g1?g2 -> (g1 & (A_mSIG.s!lf2_id!lf1_id.m1 -> SKIP)
[]
g2 & (B_mSIG.r!lf1_id!lf2_id?signal:{x | x <-
B_SIG,(get_id(x) == m2)} -> SKIP)
[]
SKIP)
```

Figura 22 – Tradução de linhas de vida com fragmento *alt*

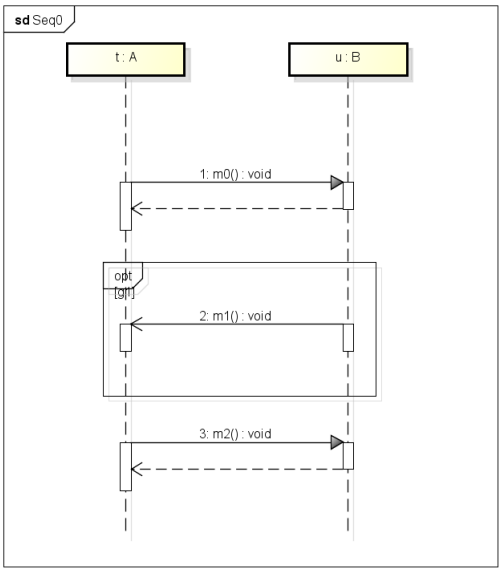

Figura 23 – Diagrama com fragmento *opt*

Diferentemente do *alt* que pode possuir inúmeros operandos, o fragmento *opt* sempre irá possuir apenas um operando, e isso se reflete no seu canal que possui apenas um parâmetro booleano. Outra similaridade em relação ao fragmento *alt* é o processo gerado após a tradução. O fragmento *opt* também requer a utilização do operador de escolha externa, mas o utiliza em apenas uma verificação simples de guarda, assim como mostra a Figura 24.

A tradução das linhas de vida incluindo a tradução do *opt* é representada pela Figura 25.

```
opt1?g1 -> (g1 & (A_mSIG.r!lf2_id!lf1_id?signal:{x | x <-
A_SIG,(get_id(x) == m1)} -> SKIP)[](g1 == false) & SKIP)
```

Figura 24 – Tradução do fragmento *opt*

```
Seq0_t_A(sd_id,lf1_id,lf2_id) =(B_mOP.s!lf1_id!lf2_id.m0_I
-> SKIP);(B_mOP.r!lf2_id!lf1_id?out:{x | x
<-B_OPS,(get_id(x) == m0_O)} -> SKIP);
opt1?g1 -> (g1 & (A_mSIG.r!lf2_id!lf1_id?signal:{x | x <-
A_SIG,(get_id(x) == m1)} -> SKIP)[]SKIP);
(B_mOP.s!lf1_id!lf2_id.m2_I ->
SKIP);(B_mOP.r!lf2_id!lf1_id?out:{x | x <-B_OPS,(get_id(x)
== m2_O)} -> SKIP)

Seq0_u_B(sd_id,lf1_id,lf2_id)=
(B_mOP.r!lf1_id!lf2_id?oper:{x | x <- B_OPS,(get_id(x) ==
m0_I)} -> SKIP);(B_mOP.s!lf2_id!lf1_id.m0_O -> SKIP);
opt1?g1 -> (g1 & (A_mSIG.s!lf2_id!lf1_id.m1 ->
SKIP)[]SKIP);(B_mOP.r!lf1_id!lf2_id?oper:{x | x <-
B_OPS,(get_id(x) == m2_I)} ->
SKIP);(B_mOP.s!lf2_id!lf1_id.m2_O -> SKIP)
```

Figura 25 – Tradução de linhas de vida com o fragmento *opt*

3.3.3 Tradução do fragmento par

O fragmento *par* representa a composição paralela de um conjunto de mensagens. A Figura 26 representa um diagrama de sequência com o fragmento *par*, no exemplos em questão apenas duas mensagens estão dentro do fragmento.

Uma das características do fragmento *par* é a ausência de guardas em seus operandos, assim removendo a necessidade de canais auxiliares. A tradução do fragmento *par* consiste na composição paralela de mensagens através do operador de entrelaçamento (*interleaving*), representado por "|||". A Figura 27 representa a tradução do fragmento *par* do exemplo em questão quando é considerada a linha de vida da classe *A*.

A tradução das linhas de vidas incluindo a tradução do *par* é representada pela Figura 28.

3.3.4 Tradução do fragmento loop

Dentre os fragmentos suportados pela ferramenta, o *loop* é o mais complexo do ponto de vista do processo de tradução. A sua complexidade se dá pelo compor-

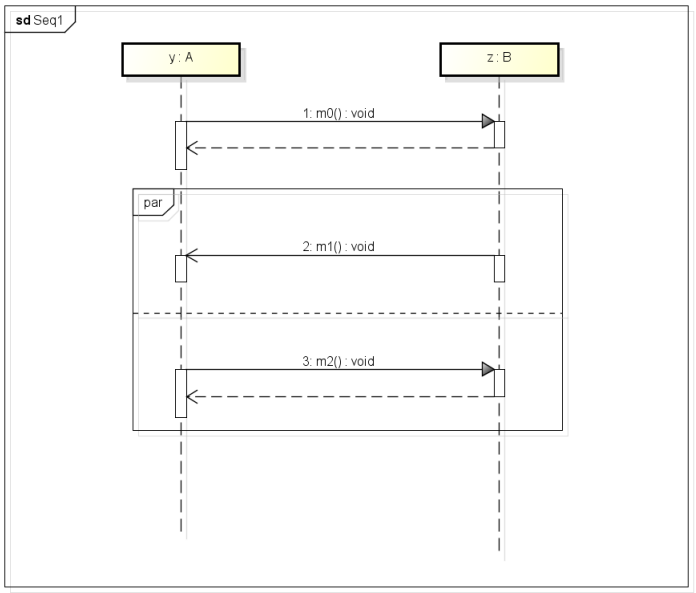

Figura 26 – Tradução do fragmento *par*

```
(A_mSIG.r!lf2_id!lf1_id?signal:{x | x <- A_SIG,(get_id(x) ==
m1)} -> SKIP
|||
(B_mOP.s!lf1_id!lf2_id.m2_I -> B_mOP.r!lf2_id!lf1_id?out:{x
| x <-B_OPS,(get_id(x) == m2_O)} -> SKIP))
```

Figura 27 – Diagrama com fragmento *par*

```
Seq1_y_A(sd_id,lf1_id,lf2_id) =(B_mOP.s!lf1_id!lf2_id.m0_I
-> SKIP);(B_mOP.r!lf2_id!lf1_id?out:{x | x
<-B_OPS,(get_id(x) == m0_O)} -> SKIP);
(A_mSIG.r!lf2_id!lf1_id?signal:{x | x <- A_SIG,(get_id(x) ==
m1)} -> SKIP|||(B_mOP.s!lf1_id!lf2_id.m2_I ->
B_mOP.r!lf2_id!lf1_id?out:{x | x <-B_OPS,(get_id(x) ==
m2_O)} -> SKIP))

Seq1_z_B(sd_id,lf1_id,lf2_id)=(B_mOP.r!lf1_id!lf2_id?oper:{x
| x <- B_OPS,(get_id(x) == m0_I)} ->
SKIP);(B_mOP.s!lf2_id!lf1_id.m0_O -> SKIP);
(A_mSIG.s!lf2_id!lf1_id.m1 ->
SKIP|||(B_mOP.r!lf1_id!lf2_id?oper:{x | x <-
B_OPS,(get_id(x) == m2_I)} -> B_mOP.s!lf2_id!lf1_id.m2_O ->
SKIP))
```

Figura 28 – Tradução de linhas de vida com o fragmento *par*

tamento repetitivo e pela variedade de casos possíveis. Assim como citado na Seção 2.1.3.4 do Capítulo 2, o loop pode possuir três formatos segundo a (UML, 2015):

- **Loop sem limite definido**
- **Loop com apenas um limite definido**
- **Loop com dois limites definidos**

Para cobrir todos os casos de tradução vamos considerar o diagrama de sequência da Figura 29.

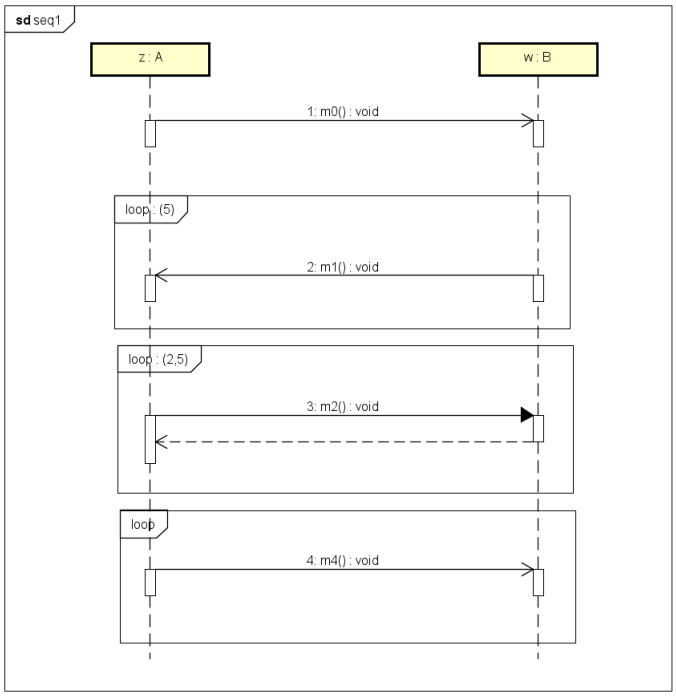

Figura 29 – Diagrama de sequência com todos os casos de *loop*

Considerando o primeiro *loop* do exemplo, podemos identificar que se encaixa no caso de **loop com apenas um limite definido**, neste caso o limite são 5 repetições. Por definição, cada iteração do *loop* deve executar todas mensagens presentes no interior do fragmento, nesse caso a mensagem assíncrona *m1*, o desafio da tradução do loop está em representar todas as iterações em todos os casos possíveis de *loop*.

Para representar o comportamento do *loop* é necessária a criação de processos auxiliares na etapa de tradução. Esses processos auxiliares servem para representar o comportamento de cada fragmento *loop* de cada linha de vida. A Figura 30 ilustra o processo criado para o primeiro *loop* do exemplo.

Note que este processo utiliza uma abordagem baseada em recursão, onde o processo será chamado recursivamente com parâmetros diferentes até atingir a condição de parada e passar a se comportar como *SKIP*.

Agora considerando o segundo fragmento *loop* do exemplo podemos identificar que ele se encaixa no caso de **loop com dois limites definidos**, neste caso os valores

```
seq1_LOOP1_z_A(iter, bound,lf2_id,lf1_id)=if(iter<=bound)
then A_mSIG.r!lf2_id!lf1_id?signal:{x | x <-
A_SIG,(get_id(x) == m1)} ->
seq1_LOOP1_z_A(iter+1,bound,lf2_id,lf1_id) else SKIP
```

Figura 30 – Processo auxiliar do *loop* com um limite

2 e 5 representam respectivamente o limite inferior e superior de iterações. Por definição, esse caso de *loop* deve executar ao menos o número de iterações definidas pelo limite inferior, após realizar o mínimo de repetições o loop pode continuar até o limite superior ou acabar. Para representar esse comportamento foi necessária a crição de dois processos auxiliares:

```
seq1_LOOP2_z_A(iter,bound,maxbound,lf1_id,lf2_id)=
if(iter<=bound) then B_mOP.s!lf1_id!lf2_id.m2_I ->
B_mOP.r!lf2_id!lf1_id?out:{x | x <-B_OPS,(get_id(x) ==
m2_O)} ->
seq1_LOOP2_z_A(iter+1,bound,maxbound,lf1_id,lf2_id) else
seq1_LOOP2_z_Aaux(iter,bound,maxbound,lf1_id,lf2_id)

seq1_LOOP2_z_Aaux(iter,bound,maxbound,lf1_id,lf2_id)=if(iter
<=maxbound) then B_mOP.s!lf1_id!lf2_id.m2_I ->
B_mOP.r!lf2_id!lf1_id?out:{x | x <-B_OPS,(get_id(x) ==
m2_O)} -> seq1_LOOP2_z_Aaux(iter+1,
bound,maxbound,lf1_id,lf2_id) [] SKIP else SKIP
```

Figura 31 – Processo auxiliar do *loop* com dois limites

Note que neste caso o loop também é utilizado de forma recursiva mas muda o seu comportamento após a realização do número mínimo de iterações. Para representar a possibilidade de saída do *loop* foi utilizado o operador de escolha. Adicionalmente a linguagem UML possui um fragmento denominado *break* que pode ser utilizado para quebrar a execução de outro fragmento, devido a limitações técnicas a ferramenta desenvolvida ainda não possui suporte ao fragmento *break*.

Agora para o último caso do fragmento *loop*, o **loop sem limite definido** ou loop infinito. Dentre os três casos esse é o mais simples, só é necessária a criação de um processo simples que comunica o conteúdo do *loop* e reinicia indefinidamente.

Após a definição de todos esses processos auxiliares é possível utilizá-los nos processos de linhas de vida do diagrama, assim como mostra a Figura 33.

```
seq1_LOOP3_z_A(lf1_id,lf2_id)=B_mSIG.s!lf1_id!lf2_id.m4 ->
seq1_LOOP3_z_A(lf1_id,lf2_id)
```

Figura 32 – Processo auxiliar do *loop* infinito

```
seq1_z_A(sd_id,lf1_id,lf2_id)=(B_mSIG.s!lf1_id!lf2_id.m0 ->
SKIP);seq1_LOOP1_z_A(0,5,lf2_id,lf1_id);seq1_LOOP2_z_A(0,2,5
,lf1_id,lf2_id);seq1_LOOP3_z_A(lf1_id,lf2_id)

seq1_w_B(sd_id,lf1_id,lf2_id)=
(B_mSIG.r!lf1_id!lf2_id?signal:{x | x <- B_SIG,(get_id(x) ==
m0)} -> SKIP);seq1_LOOP1_w_B(0,5,lf2_id,lf1_id);
seq1_LOOP2_w_B(0,2,5,lf1_id,lf2_id);
seq1_LOOP3_w_B(lf1_id,lf2_id)
```

Figura 33 – Tradução de linhas de vida com *loop*

3.4 Paralelismo e Sincronização

A última etapa da tradução consiste na criação dos processos que irão representar a execução dos diagramas de sequência, tais processos são construídos através da composição paralela dos processos construídos nas etapas anteriores. Os dois processos auxiliares criados nessa etapa são o *MessagesBuffer* que foi mostrado na Seção 3.2.3 e o *SeqParallel*, esses processos representam a composição paralela das mensagens e das linhas de vida dos diagramas de sequência, respectivamente.

Diferente do *MesssagesBuffer*, o processo *SeqParallel* utiliza paralelismo alfabetizado para compor os processos das linhas de vida paralelamente. A Figura 34 ilustra o processo *SeqParallel* do diagrama exemplo.Nesse processo podemos observar a composição paralela alfabetizada de dois processos de linha de vida e a sincronização nos eventos de mensagem.

```
Seq0Parallel(sd_id,lf1_id,lf2_id) =
Seq0_x_A(sd_id,lf1_id,lf2_id)[ {|B_mOP.s.lf1_id.lf2_id.m0_I,
B_mOP.r.lf2_id.lf1_id.m0_O, A_mSIG.r.lf2_id.lf1_id.m1|} ||
{|B_mOP.r.lf1_id.lf2_id.m0_I, B_mOP.s.lf2_id.lf1_id.m0_O,
A_mSIG.s.lf2_id.lf1_id.m1|} ]Seq0_u_B(sd_id,lf1_id,lf2_id)
```

Figura 34 – Processo *SeqParallel*

Agora que os eventos auxiliares foram criados, o tradutor gera o processo que representa o diagrama de sequência como um todo.

```
SD_Seq0(sd_id,lf1_id,lf2_id) = beginInteraction
->((Seq0Parallel(sd_id,lf1_id,lf2_id); endInteraction ->
SKIP)[|{|B_mOP.s.lf1_id.lf2_id.m0_I,B_mOP.r.lf1_id.lf2_id.m0
_I,B_mOP.s.lf2_id.lf1_id.m0_O,B_mOP.r.lf2_id.lf1_id.m0_O,
A_mSIG.s.lf2_id.lf1_id.m1,A_mSIG.r.lf2_id.lf1_id.m1,endInter
action|}|]Seq0_MessagesBuffer(sd_id,lf1_id,lf2_id))
```

Figura 35 – Processo final do diagrama exemplo

O processo *SD_Seq0* descrito na Figura 35 representa a tradução final do diagrama de exemplo para um processo CSP. Esse processo é formado através da composição paralela generalizada dos processos auxiliares *MessagesBuffer* e *SeqParallel*. Essa abordagem permite formar um processo que representa fielmente o diagrama de sequência, comunicando suas mensagens na ordem em que foram definidas no diagrama original. Em um caso normal de utilização da ferramenta, dois processos *SD_Seq* seriam construídos, um para cada diagrama presente na verificação de refinamento.

3.5 Verificação de Refinamento

Após a tradução dos diagramas para uma especificação CSP é possível utilizar a ferramenta FDR para realizar a verificação de refinamento. O FDR suporta os seguintes tipos de verificação de refinamento baseado em diferentes modelos semânticos:

- Trace
- Falhas
- Falhas e Divergências

Como foi dito nos Capítulos 1 e 2, este trabalho aborda a verificação de refinamentos no modelo semântico de *traces*.

Em CSP, a verificação de refinamento no nível de *traces* avalia se o conjunto de *traces* de um processo mais concreto está contido no conjunto de *traces* de um outro processo mais abstrato. No entanto, no nível de UML tal refinamento não pode ser considerado pois o diagrama evolui de um nível abstrato para um nível mais concreto, ou seja, são adicionados novos elementos ao *trace* inicial. Nesta situação novos eventos podem surgir para adicionar detalhes nos modelos mais abstratos. Sendo assim, foi criada uma adaptação do refinamento de CSP para a realidade de diagrama

de sequência usando noções de refinamento definidas em (LIMA; IYODA; SAMPAIO, 2016).

Para realizar a verificação de refinamentos é necessária a utilização dos seguintes elementos do CSP e da ferramenta FDR:

1. **Asserção** representada pela palavra reservada "*assert*". Essa funcionalidade da ferramenta FDR permite realizar verificações como refinamento por trace.
2. **Operador *hide*** do CSP, representado por " $\setminus$ ". Esse operador será utilizado para esconder as novas mensagens que surgiram no diagrama mais concreto.
3. **Operador de refinamento de traces**, representado por $\sqsubseteq_{\mathsf{T}}$. Esse operador é utilizado em conjunto com a asserção para informar ao FDR que a verificação de refinamento de traces deve ser avaliada.

A verificação de refinamento de traces em CSP pode ser representada por:

$$P \sqsubseteq_{\mathsf{T}} Q$$

Que pode ser lido como "*P* é refinado por *Q* no modelo de traces" ou "*Q* refina *P* no modelo de traces", onde *P* e *Q* representam processos. Para realizar uma verificação de refinamento através da ferramenta FDR utilizaremos comandos no seguinte formato:

$$assert\ P\ [T=\ Q \setminus \{hiddenEvents\}$$

Onde *"[T="* é equivalente ao operador " $\sqsubseteq_{\mathsf{T}}$ " e representa a verificação de refinamento de traces, e "$\{hiddenEvents\}$" representa o conjunto de eventos do processo *Q* que serão escondidos, esse conjunto pode estar relacionado a qualquer um dos processos da verificação, assim como será mostrado a seguir.

A etapa de verificação de refinamento é onde o usuário da ferramenta terá ação. A Figura 36 representa o painel de verificação de refinamento da ferramenta, nele o usuário pode escolher os diagramas do projeto que serão utilizados na verificação de refinamento e pode escolher qual o tipo de verificação de refinamento que será utilizada.

Para a ferramenta foram consideradas duas noções de refinamento, as quais serão explicadas em seguida.

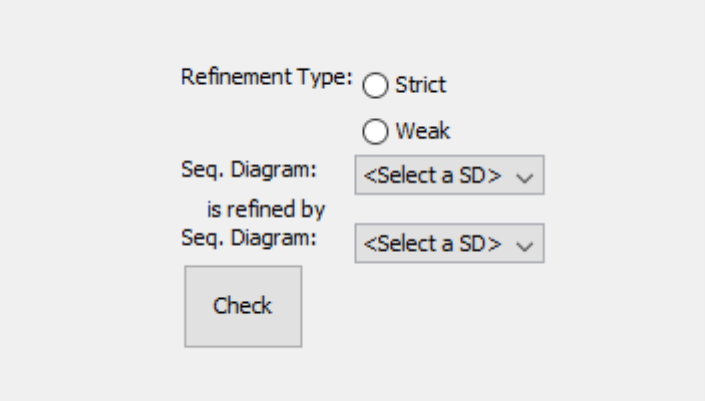

Figura 36 – Configuração de refinamento da ferramenta

3.5.1 Weak Refinement

No refinamento *weak* é verificado se os *traces* presentes no diagrama mais abstrato estão presentes no *trace* do diagrama mais concreto. No entanto, os novos eventos que surgirem no diagrama mais concreto não são considerados. Nessa noção de refinamento, o diagrama mais concreto pode adicionar novos *traces* usando os eventos presentes em ambos os diagramas, no entanto, os *traces* existentes no diagrama mais abstratos devem ser preservados. Podemos imaginar a aplicação desse refinamento na modelagem de um caso de uso onde inicialmente é considerado apenas o fluxo principal, e em seguida uma nova versão é gerada adicionando também eventos relacionados a fluxos alternativos ou de exceção. O *weak refinement* pode ser descrito da seguinte forma:

$$assert\ t(SD2) \setminus diff(Events(t(SD2)), Events(t(SD1)))\ \sqsubseteq_{\mathsf{T}}\ t(SD1)$$

Onde $t(seq)$ representa o processo CSP do diagrama *seq* e *diff(Events(t(SD2)), Events(t(SD1)))* representa o conjunto de eventos que estão presentes no processo *SD2* mas não em *SD1*.Esta asserção verifica se o conjunto de traces da especificação *SD1* está dentro do conjunto de traces da implementação *SD2*, desconsiderando possíveis novos eventos de *SD2*.

3.5.2 Strict Refinement

O *Strict refinement* é mais rigoroso se comparado ao *Weak refinement*, ele é utilizado para verificar a equivalência entre especificações desconsiderando novos eventos surgidos na implementação. Dado um caso de uso completo com diversos fluxos (fluxo principal, fluxo alternativo e fluxo excepcional), o *Strict refinement* pode ser utilizado para verificar a equivalência entre uma especificação inicial do caso de uso e uma

implementação que adiciona novos eventos. Assim como no caso anterior, é possível verificar o *Strict refinement* da seguinte forma:

$$assert\ t(SD1)\ \sqsubseteq_{\mathsf{T}}\ t(SD2) \setminus diff(Events(t(SD2)), Events(t(SD1)))$$

$$assert\ t(SD2) \setminus diff(Events(t(SD2)), Events(t(SD1)))\ \sqsubseteq_{\mathsf{T}}\ t(SD1)$$

Note que nesse tipo de verificação são realizadas duas asserções, isso é necessário pois a relação de equivalência entre processos só existe caso o conjunto de traces dos processos sejam exatamente iguais. Note também que uma das asserções verificadas é a mesma presente no *weak refinement*, ou seja, sempre que o *strict refinement* for validado, o *weak refinement* também será, contudo a relação contrária não é verdadeira.

3.6 Contraexemplo e Rastreabilidade

A última funcionalidade da ferramenta é a rastreabilidade de contraexemplos. Ao realizar uma verificação de refinamentos pela ferramenta FDR a asserção pode resultar em falha, nesse caso o FDR identifica um contraexemplo em termos da sequência de eventos que levaram a falha no refinamento. Uma das funcionalidades da ferramenta é exibir as inconsistências presentes nos diagramas, para isso foi construído um módulo capaz de extrair contraexemplos do FDR e transmití-los para os diagramas originais.

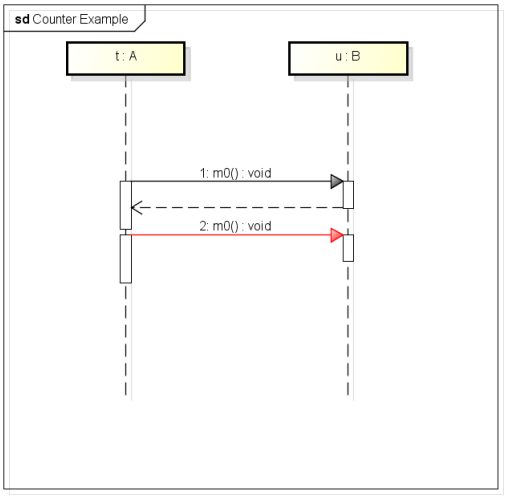

Figura 37 – Rastreabilidade de contraexemplo na ferramenta

A Figura 37 ilustra um diagrama com um contraexemplo. Quando é identificado um contraexemplo na verificação de refinamento, a ferramenta irá extrair o evento que

invalida o refinamento e traduzí-lo de volta para a semântica de diagrama de sequência via engenharia reversa. Após a tradução reversa, a ferramenta vai identificar o diagrama de sequência e a mensagem que causou a falha de refinamento e vai destacá-la com a cor vermelha. Caso o usuário corrija o problema e refaça a verificação de refinamento, o contraexemplo irá desaparacer e o refinamento será dado como verdadeiro.

4 Avaliação

Este capítulo tem o intuito de demonstrar a utilização da ferramenta desenvolvida em um estudo de caso. Para essa avaliação foram escolhidos diferentes casos de uso relacionados à Agência Pernambucana de Águas e Clima (APAC). A APAC, juntamente com outras instituições, realiza monitoramento hidrometereológico em todo o estado de Pernambuco. Para realizar esse monitoramento a APAC conta com sistemas de coleta de dados através de plataformas de sensores espalhadas por todo o estado. A APAC também conta com um software para manter o cadastro e controle dos seus sensores, se trata do Sistema de Informação para Recursos Hídricos de Pernambuco (SIRHPE).

Ao longo deste capítulo utilizaremos a ferramenta desenvolvida para realizar verificações de refinamento em casos de uso do sistema SIRHPE e em algumas etapas do processo de coleta de dados da APAC.

4.1 Coletores de dados da APAC

A APAC realiza a coleta de dados hidrometeorológicos através de scripts que fazem análise e processamento de dados brutos oriundos de Plataformas de Coleta de dados (PCDs). Essas plataformas ficam espalhadas por todo o estado e se comunicam com diversos servidores de instituições que não se limitam à APAC.

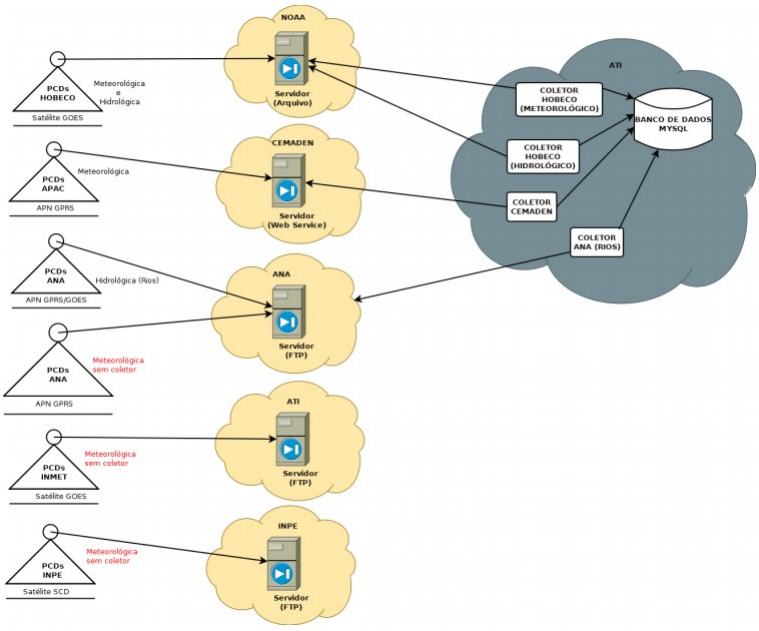

Figura 38 – Arquitetura de coleta de dados da APAC

Como podemos ver na Figura 38, diferentes atores estão envolvidos no processo de coleta de dados. O primeiro deles é a plataforma de coleta que obtêm dados do ambiente através de sensores. Essa plataformas possuem *Data Loggers* que armazenam os dados coletados até que sejam enviados para servidores FTP(*File Transfer Protocol*)(POSTEL; REYNOLDS, 1985) via telemetria. O envio de dados é feito em intervalos previamente definidos e que variam de PCD para PCD, além disso também existem PCDs que realizam transmissão de dados via satélite.

Após coletar o dados do ambiente, a maioria dos PCDs irá enviar seus dados para servidores FTP intermediários, esses servidores são propriedade de diversas instituições como CEMADEN, CPTEC e INPE. Os servidores servem como intermédio para armazenar os dados brutos enviados pelos PCDs até que um coletor especializado recolha os dados.

Os coletores são os atores finais do processo de coleta, eles são scripts hospedados nos servidores da Agência Estadual de Tecnologia da Informação do estado (ATI). Esses coletores são compostos por tarefas agendadas que irão acessar os dados dos servidores FTP intermediários, processar os dados e armazená-los em bases de dados MySQL (DUBOIS; BY-WIDENIUS, 1999). Note que conforme a Figura 38, podemos identificar diferentes tipos de coletores que são especializados em algum tipo de dado, isso possibilita que coletores extraiam dados diferentes do mesmo servidor intermediário.

Agora que foram estabelecidas todas as entidades do processo de coleta, podemos elaborar cenários para a aplicação da ferramenta de verificação de refinamento. As subseções a seguir irão apresentar dois possíveis cenários onde a utilização de verificação de refinamento poderia ser útil para garantir a evolução do sistema.

4.1.1 Caso 1: Processo de um Coletor

Como foi dito anteriormente, os coletores são *scripts* que executam tarefas em tempos pré-determinados, mais especificamente um agendamento *Crontab*(GROUP, 2018). A implementação dos coletores garante que os mesmos entrarão em contato com os servidores múltiplas vezes ao longo de um dia, para representar isso em um diagrama de sequência podemos utilizar o fragmento *loop*.

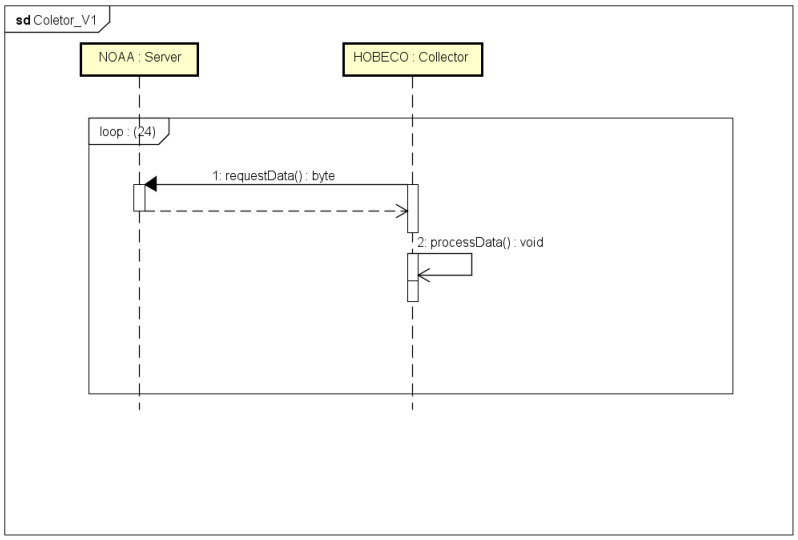

Figura 39 – Modelagem inicial da coleta de dados

A Figura 39 ilustra uma ideia inicial do funcionamento de um coletor. Com esse diagrama conseguimos representar uma instância de um coletor que busca dados em uma instância de servidor, para o exemplo vamos supor que o coletor faz 24 requisições ao servidor no intervalo de um dia, podemos representar isso pelo fragmento *loop* com o limite superior de interações em 24. Contudo, essa implementação ainda não reflete o funcionamento esperado do coletor, conforme a Figura 38 podemos observar que após processar os dados internamente, os coletores enviam os dados coletados para uma base de dados MySQL. Para atingir esta restrição basta adicionar uma nova entidade e uma nova mensagem ao diagrama conforme a Figura 40.

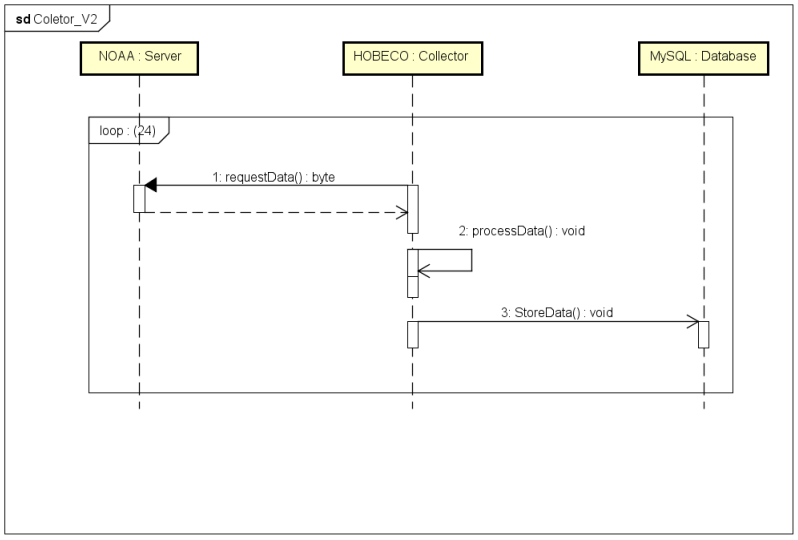

Figura 40 – Modelagem atualizada da coleta de dados

Agora temos uma versão aprimorada do diagrama construído inicialmente, com isso podemos utilizar a ferramenta desenvolvida para verificar se o novo diagrama é

um refinamento do primeiro. Ao executar verificação de refinament *weak* recebemos a confirmação de que o novo diagrama é um refinamento do anterior. Isso acontece pois o refinamento *weak* verifica apenas se o novo diagrama ainda contém os mesmo traces do diagrama inicial.

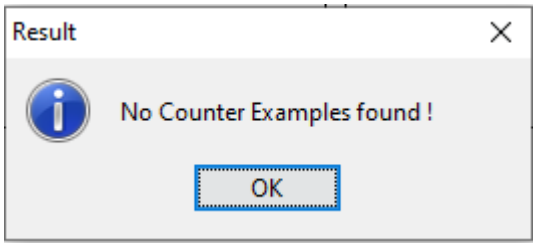

Figura 41 – Confirmação da verificação de refinamento

4.1.2 Caso 2: Processo de múltiplos Coletores

Agora vamos considerar um caso mais amplo, incluindo alguns dos múltiplos coletores e servidores intermediários apresentados na Figura 38. Ao considerar múltiplos coletores e servidores precisamos levar em conta que todas essas entidades são independentes e podem trabalhar de forma simultânea. Para representar este comportamento em um diagrama de sequência podemos utilizar o fragmento *par*, conforme a Figura 42.

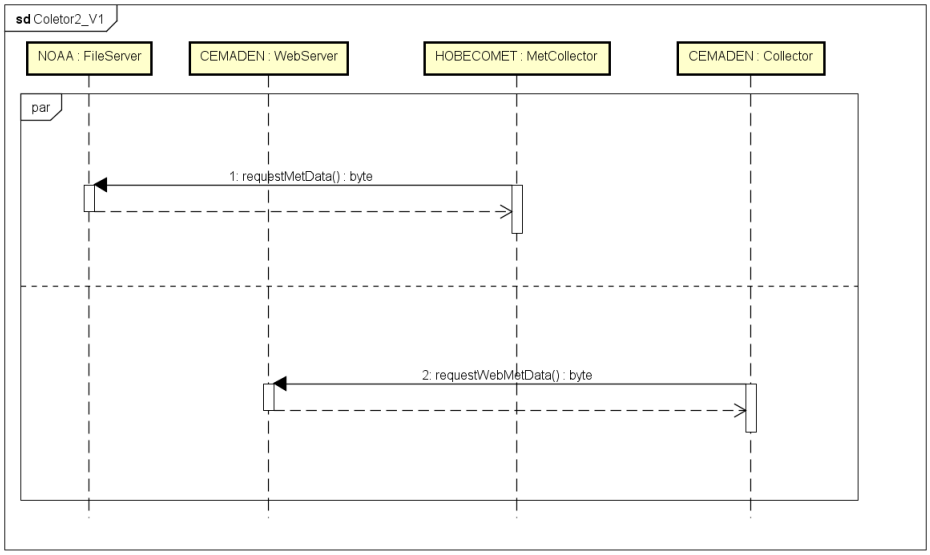

Figura 42 – Modelagem inicial da coleta de dados por múltiplos coletores

Nesta figura podemos ver dois coletores trabalhando simultaneamente fazendo requisições para seus respectivos servidores associados. Agora vamos supor a adição de um novo coletor ao sistema, consequentemente a modelagem do sistema seria atualizada para representar a nova entidade, conforme mostra a Figura 43.

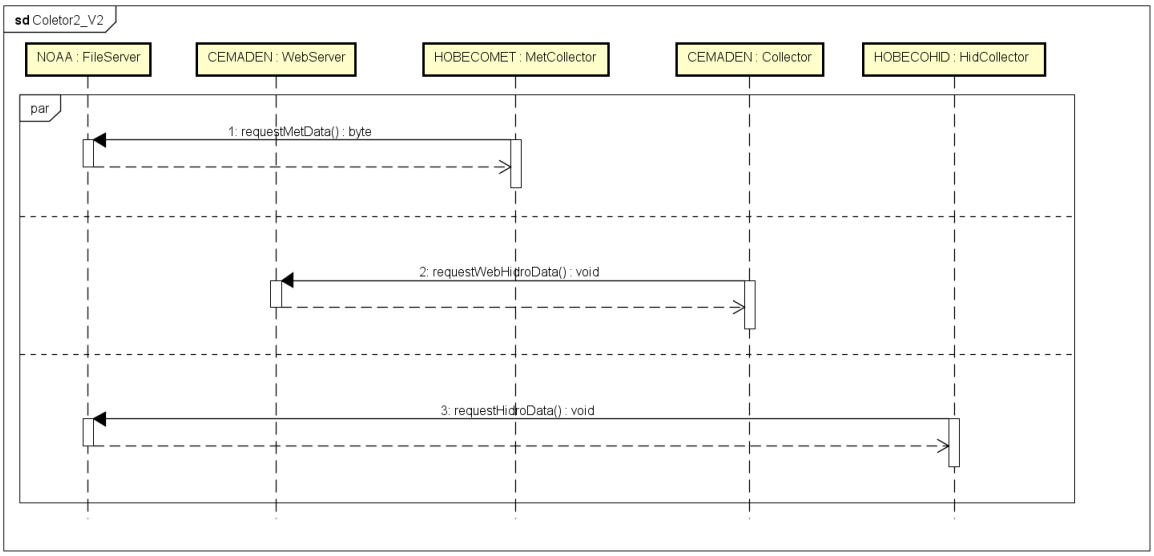

Figura 43 – Modelagem atualizada da coleta de dados por múltiplos coletores

Agora temos um diagrama que inclui um novo coletor, mas note que há uma diferença nessa modelagem se comparada a anterior. Na primeira versão do sistema o coletor CEMADEN requisitava dados metereológicos ao servidor web CEMADEN através da chamada *requestWebMetData*. Na nova versão esse relacionamento foi modificado, agora o coletar CEMADEN está requisitando os dados hidrológicos do servidor através da chamada *requestWebHidroData*. Ao realizar a verificação de refinamento *weak* nos dois diagramas criados seria identificado um contraexemplo conforme ilustra a Figura 44.

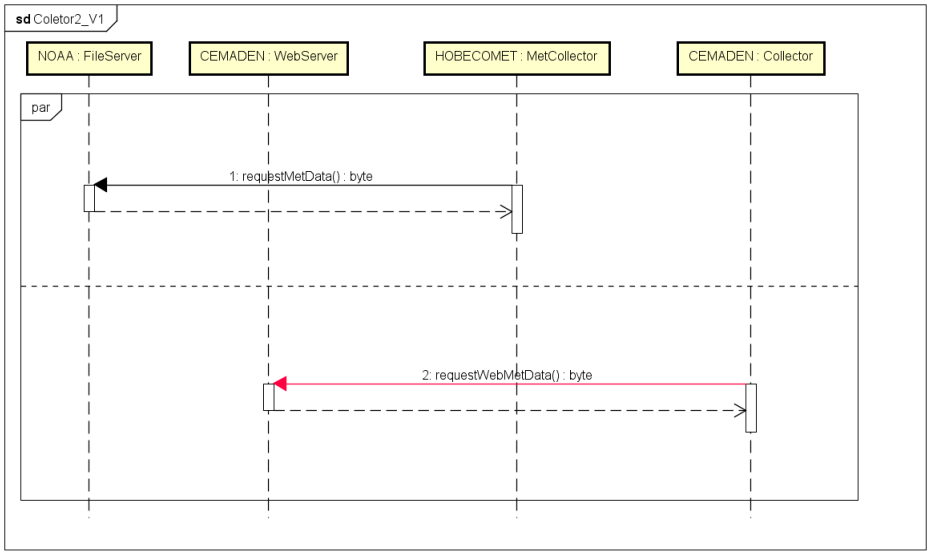

Figura 44 – Contraexemplo da modelagem de múltiplos coletores

O novo diagrama não possui mais a coleta de dados metereológicos do servidor CEMADEN, isso causa uma inconsistência na verificação de refinamento, visto que o novo diagrama não contém todos os traces do diagrama inicial. Para que o refinamento

ocorra é necessário modificar o novo diagrama de forma a manter todos as interações anteriores, nesse caso basta modificar o método de coleta do CEMADEN.

4.2 SIRHPE

O Sistema de Informação para Recursos Hídricos de Pernambuco (SIRHPE) é o sistema utilizado no gerenciamento do sistema de coleta de dados hidrometeorológicos da APAC. O sistema oferece uma interface gráfica onde é possível gerenciar componentes importantes do sistema de monitoramento, tais quais sensores, pluviômetros, PCDs e outros equipamentos de monitoramento meteorológico.

Além disso, o sistema também possibilita o gerenciamento do cadastro de todos os pontos de monitoramento ao redor do estado. O sistema permine gerenciar dados de de instituições, seções esporádicas em rios e barragens, estações de monitoramento e reservatórios destinados a coleta de dados metereológicos.

Neste estudo de caso utilizaremos a verificação de refinamento em diagramas de sequência elaborados conforme os casos de uso descritos na documentação de especificação do SIRHPE.

4.2.1 Caso 1: Adicionar Pluviômetro

Na documentação de especificação do SIRHPE, o fluxo para a adição de um pluviômetro no sistema é descrito da seguinte maneira:

1. O usuário poderá acessar a tela de inclusão de pluviômetro através do menu Administrativo > Monitoramento > Pluviômetro > Incluir ou acessar a tela de consulta através do menu Administrativo > Monitoramento > Pluviômetro > Incluir;
2. O usuário prossegue com a operação de inclusão

A descrição do caso de uso é acompanhada de uma imagem do protótipo da ferramenta, ilustrada pela Figura 45.

A descrição do caso de uso ainda é bastante vaga e não possui detalhes sobre os componentes do sistema que serão requisitados durante a operação de inclusão de um novo pluviômetro. Apesar da falta de detalhes podemos elaborar uma ideia inicial de como a funcionalidade poderia ser implementada. Para isso vamos considerar o caso mais simples: uma classe responsável por tratar eventos da interface, uma classe responsável pelas regras de negócio do caso de uso e uma classe que representa a conexão com um banco de dados. Para representar esse caso de uso vamos utilizar três linhas de vida que trocam mensagens e estão dispostas como ilustra a Figura 46.

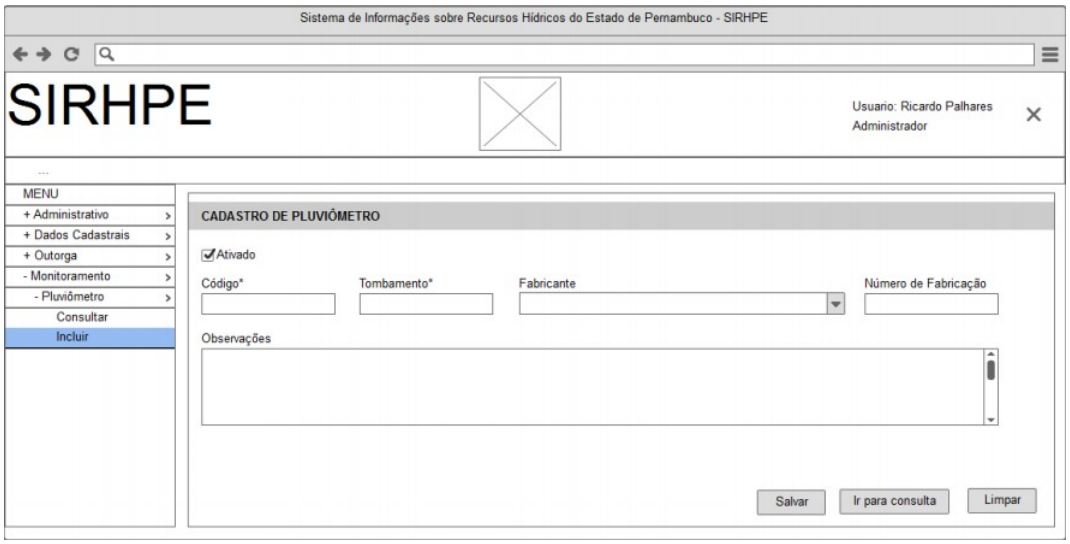

Figura 45 – Protótipo de interface para adição de pluviômetro no SIRHPE

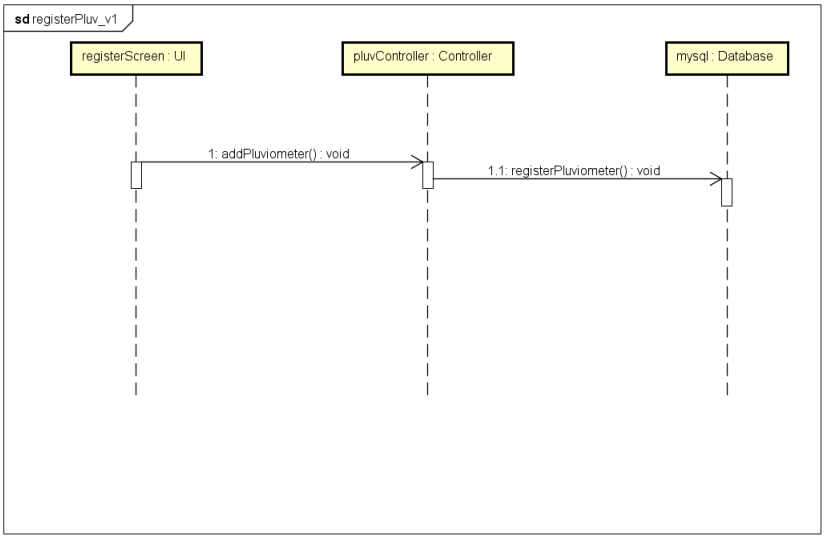

Figura 46 – Modelagem inicial da adição de novo pluviômetro

Até o momento o modelo criado cumpre as especificações do caso de uso, porém o caso de uso ainda é muito limitado e possui algumas brechas como a possibilidade de registrar o mesmo pluviômetro mais de uma vez. Com isso em mente é feita uma atualização no caso de uso que adiciona a seguinte regra de negócio :

- O sistema não deverá permitir a inclusão de mais de um pluviômetro com o mesmo código e tombamento.

Com essa atualização o modelo desenvolvido anteriormente não representa mais o caso de uso e precisa ser atualizado. A nova restrição implica na necessidade de uma verificação no momento de cadastro, em caso de falha da verificação o fluxo de cadastro deve ser desviado para evitar pluviômetros com identificadores repetidos na

base de dados. Esse desvio de fluxo pode ser considerado como um fluxo alternativo, em diagramas de sequência a melhor forma de representar fluxos alternativos é através dos fragmentos *alt* e *opt*. Nesse caso podemos utilizar o *opt* para fazer uma verificação antes de permitir o cadastro no banco de dados, assim como mostra a Figura 47.

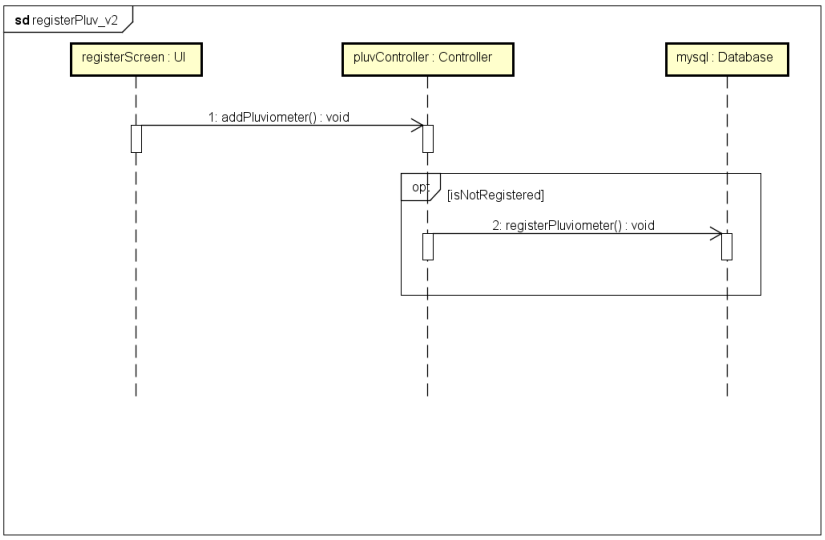

Figura 47 – Modelagem atualizada da adição de novo pluviômetro

Ao realizar a verificação de refinamento *weak* entre os dois modelos criados, identificamos que a mudança é de fato um refinamento, visto que o modelo mais atual permite todos os traces do modelo inicial e adiciona um novo trace possível. Caso seja realizado o refinamento *strict* o mesmo não será válido uma vez que os dois diagramas não são equivalentes.

4.2.2 Caso 2: Associar Periférico à PCD

Na documentação de especificação do SIRHPE o fluxo de associação de um periférico à PCD é descrito da seguinte forma:

1. Para adicionar periféricos à PCD, o usuário acessará o popup de consultas de periféricos através do botão ”Consultar Periféricos”;
2. No popup, o usuário poderá restringir sua consulta preenchendo quais campos desejar;
3. O usuário selecionará um ou mais periféricos e deverá clicar no botão “Adicionar Selecionados”;
4. Ao adicionar os periféricos selecionados o sistema deverá persistir em banco de dados a associação;
5. Para cada periférico adicionado o sistema deverá incluir um registro no histórico;

6. Quando finalizar a consulta e adição dos periféricos desejados, o usuário deverá clicar no botão “Fechar” para retornar à tela de associação dos periféricos à PCD;

Vamos novamente elaborar uma estrutura de classes que poderia ser utilizada neste caso de uso, vamos considerar somente as etapas 3 a 6, visto que as duas primeiras etapas podem ser extraídas em um caso de uso separado relacionado à consulta de periféricos. De forma similar ao Caso 1 iremos criar uma classe para representar a interface, uma classe para representar um controlador responsável pelas regras de negócio e uma classe de conexão com banco de dados. O diagrama de sequência do caso será composto por três linhas de vida conforme a Figura 48.

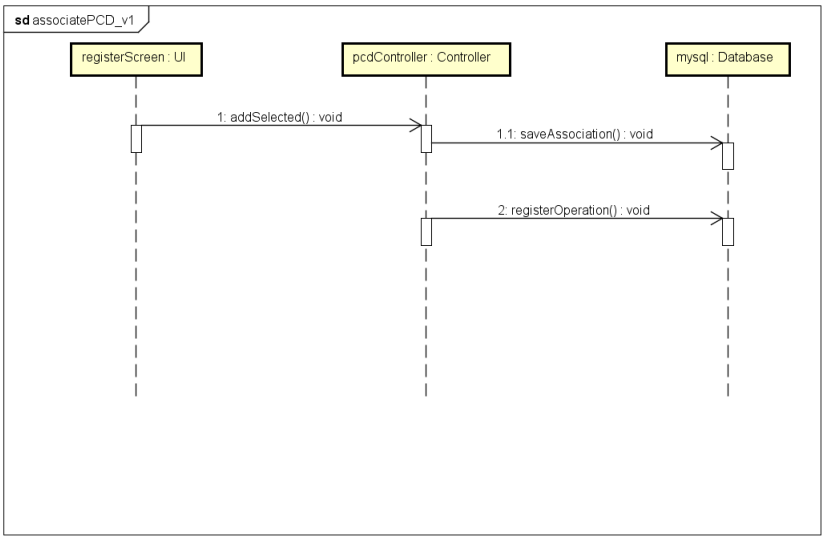

Figura 48 – Modelagem inicial da associação de periférico à PCD

Novamente temos uma modelagem que cobre a descrição do caso de uso mas ainda pode ser evoluída. Assim como no Caso 1 foi decidido adicionar mais uma restrição de negócio ao caso de uso, desta vez a restrição diz o seguinte :

- Se, ao mesmo tempo, dois ou mais usuários consultarem o mesmo periférico, apenas o primeiro que associá-los a uma PCD conseguirá concluir o processo. Os demais visualizarão uma mensagem de erro informando que o periférico, de determinado número de série, já está associado a uma PCD. Caberá ao usuário desmarcar o citado periférico e repetir a ação;

Assim como no Caso 1, para cumprir a nova restrição nas regras de negócios será necessária uma atualização no diagrama de sequência criado. Desta vez foi adicionado um fluxo excepcional para tratar o caso de associar um periférico já associado, para representar esse fluxo no diagrama podemos utilizar o operador *alt* com dois operandos. O diagrama atualizado poderia ser construído conforme a Figura 49.

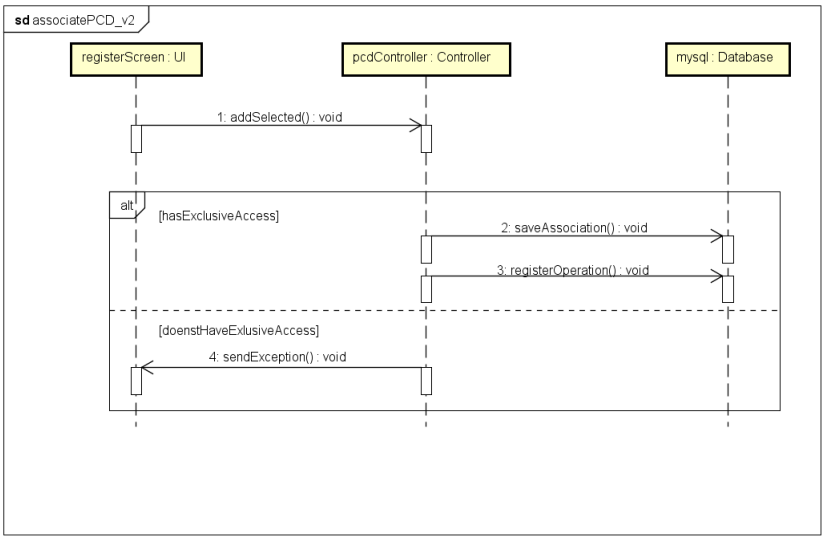

Figura 49 – Modelagem atualizada da associação de periféricos à PCD

Como podemos observar, cada operando representa um fluxo possível do caso de uso. Ao verificar o refinamento *weak* entre os dois diagramas podemos confirmar que o novo diagrama é de fato um refinamento do primeiro.

Após todos esses exemplos podemos concluir que a verificação de refinamento em diagramas de sequência é relevante no contexto de modelagem incremental de casos de uso. O refinamento *weak* se mostrou útil em todos os casos, validando modificações e identificando erros humanos. O refinamento *strict* por sua vez não possui muita aplicação nesse contexto, porém seria útil em casos onde uma funcionalidade do sistema deve ser completamente remodelada em um ponto de vista estrutural mas deve manter o mesmo funcionamento.

5 Conclusão

Este capítulo sumariza as contribuições deste trabalho, bem como suas limitações. Além disso também serão sugeridos possíveis trabalhos futuros que podem ser realizados à partir dessas contribuições.

5.1 Contribuições

Na linha de pesquisa de formalização de modelos UML já foram propostas diversas abordagens de aplicações de métodos formais para diferentes modelos UML, contudo, poucos trabalhos propõe uma abordagem prática. Neste trabalho foi proposta uma abordagem *hands-on* para o tema, com isso obtemos resultados que contribuem para a ideia da viabilidade da formalização de modelos UML.

A construção de uma ferramenta verificadora de refinamento no formato de *plugin* mostra que é possível utilizar semânticas formais de UML em softwares de modelagem bem estabelecidos como o Astah. Esse tipo de abordagem produz resultados mais palpáveis,visto que esse tipo de ferramenta pode ser utilizada em aplicações do mundo real.

A principal contribuição deste trabalho foi a adição do suporte à tradução de fragmentos combinados. Trabalhos anteriores se viam limitados pela falta de suporte a tradução desses elementos, uma vez que diagramas de sequência são comumente utilizados para representar casos de uso, que por sua vez podem possuir fluxos complexos ou até mesmo diversos desvios de fluxos.

O foco em fragmentos de controle adveio do fato de serem os fragmentos mais comumente utilizados em diagramas de sequência. A possibilidade de especificar desvios de fluxo de diagramas através da adição de fragmentos como *opt* e *alt* se mostrou especialmente relevante. Esses fragmentos possibilitam a verificação de refinamento em diagramas comumente utilizados para representar casos de uso, assim como exemplificado na Seção 2 do Capítulo 4.

Os exemplos do Capítulo 4 demonstram que a possibilidade de realizar verificações de refinamento em casos de uso representa um grande benefício, especialmente devido à forma simples e automatizada desse processo. A integração direta da ferramenta de modelagem com a tradução formal e verificador FDR permite que o usuário final da ferramenta não tenha nenhum contato com notações formais e consiga se aproveitar de diferentes verificações de refinamento, podendo identificar inconsistências em seu projeto através de contraexemplos.

5.2 Trabalhos Relacionados

Ao longo dos anos surgiram diversas iniciativas que buscam formalizar a linguagem UML. Grande parte dessas pesquisas propõem semânticas formais para diagramas UML de forma a ir além das capacidades oferecidas por linguagens de restrições de objeto como *OCL* (WARMER; KLEPPE, 2003).

Em (LIMA; DIDIER; CORNÉLIO, 2013) são abordados diagramas de atividade em SysML(HAUSE et al., 2006),um subconjunto extendido de UML. Nessa pesquisa é utilizada a notação formal CML (WOODCOCK et al., 2012) para estabelecer uma semântica formal para os diagramas. Um trabalho similar é realizado em (LIMA; IYODA; SAMPAIO, 2014) no contexto de diagramas de sequência SysML. Esses trabalhos apresentam uma semântica de formalização de diagramas que mais tarde foi expandida e adaptada para o método formal CSP em (LIMA; IYODA; SAMPAIO, 2016).

Uma das linhas de pesquisa que buscam formalizar o UML é a UML-B (SNOOK; BUTLER, 2006), uma formalização da linguagem UML para a notação B (ABRIAL; ABRIAL, 2005). Essa pesquisa desenvolve uma abordagem para formalização de diagramas de classe, nela os autores desenvolveram um mapa de tradução onde um diagrama é traduzido em um componente B. Segundo os autores, essa abordagem permite os seguintes benefícios: “A estrutura é provida pelo UML ao invés do B, a semântica é provida através de restrições nos modelos UML e a prova e refinamento é alcançada através da tradução para B”.

Existem também linhas de pesquisa que utilizam redes de petri (PETERSON, 1977) para definir semânticas formais para diagramas UML. Em (EICHNER et al., 2005) é criada uma semântica formal para representar diagramas de sequência, enquanto que em (STÖRRLE; HAUSMANN, 2005) é proposta uma abordagem similar para o contexto de diagramas de atividade. Outras linguagens de especificação como *Object-Z* (SMITH, 2012), *VHDL*(NAVABI, 1997) e Promela(HOLZMANN, 2004) também são utilizadas para criar mapeamentos formais de diagramas UML em trabalhos como (KIM; CARRINGTON, 2000), (MCUMBER; CHENG, 1999) e (MCUMBER; CHENG, 2001).

Outras abordagens buscam realizar verificações formais em diagramas UML para fins de viabilizar o desenvolvimento orientado a modelos (*Model-driven Development*) (SELIC, 2003). Esse é o caso do UMLtoCSP (CABOT; CLARISÓ; RIERA, 2007) que faz uso de programação de restrição (ROSSI; BEEK; WALSH, 2006) para verificar corretude de diagramas de classe UML anotados com restrições *OCL*.

Em (LIMA; IYODA; SAMPAIO, 2016) é definida a base das noções de refinamento de diagramas de sequência UML em termos da linguagem CSP. Nesse trabalho são definidos elementos semânticos importantes para tradução de UML para CSP. Outros pontos importantes introduzidos na pesquisa são referentes a noções

de refinamento, mais especificamente *weak refinement*, *strict refinement* e *renaming refinement*.

A tabela a seguir compara este trabalho com a maioria dos trabalhos relacionados citados.

Trabalho	**Método formal**	**Diagrama**	**Ferramenta**
Este trabalho	CSP	Diag. Sequência	Sim
(SNOOK; BUTLER, 2006)	B	Diag. Classe	Sim
(LIMA; DIDIER; CORNÉLIO, 2013)	CML	Diag. Atividade	Não
(LIMA; IYODA; SAMPAIO, 2014)	CML	Diag. Sequência	Não
(EICHNER et al., 2005)	Rede de Petri	Diag. Sequência	Não
(STÖRRLE; HAUSMANN, 2005)	Rede de Petri	Diag. Atividade	Não
(LIMA; IYODA; SAMPAIO, 2016)	CSP	Diag. Sequência	Não

Tabela 1 – Tabela de comparação de trabalhos relacionados

5.3 Limitações e trabalhos futuros

Apesar da ferramenta construída apresentar bons resultados ainda existem diversos fatores que limitaram sua construção. Podemos elencar limitações tais como:

- **Parâmetros de mensagem:** Em versões iniciais da ferramenta foi adicionado o suporte para detecção de parâmetros de mensagens dos diagramas. Essa funcionalidade se mostrou muito limitada pela dificuldade de representar todos os parâmetros possíveis na especificação CSP. Ainda nas versões iniciais da ferramenta foi removido o suporte a parâmetros de mensagem. Esse suporte poderia ser implementado através da integração de diagramas de sequência e diagramas de classe, contudo, diagramas de classe não foram abordados no escopo deste trabalho.

- **Composição de fragmentos combinados:** Uma das principais limitações do projeto é a falta de suporte à detecção de fragmentos internos a outros fragmentos combinados. Essa limitação surgiu pela dificuldade de identificar esse tipo de fragmento através da API da plataforma Astah, contudo, essa limitação não representa uma funcionalidade impossível de ser implementada em um trabalho com escopo maior. Como resultado dessa limitação não foi possível implementar fragmentos tais como o *break*, uma vez que esse fragmento é comumente utilizado dentro de fragmentos como *loop*.

- **Suporte a todos fragmentos combinados:** Neste trabalho foram abordados apenas os fragmentos combinados relacionados à estruturas de controle. Como

dito anteriormente esses são os fragmentos mais comumente utilizados em casos reais, contudo, outros fragmentos tais como *strict*, *critical*, *ignore*, *consider*, *negate* e *assert* ainda não são suportados pela ferramenta.

- **Regras de tradução formais:** O processo de tradução foi codificado de forma a suportar a API do Astah, idealmente as regras de tradução poderiam ser definidas formalmente. Desta forma o trabalho ficaria mais genérico e independente de tecnologia, além de possibilitar provas de corretude.

- **Validação limitada:** Apesar da validação da ferramenta ter sido feita considerando casos de uso reais, o cenário ideal de validação seria um estudo empírico. Desta forma poderíamos avaliar o uso da ferramenta em diferentes cenários e comparar a abordagem que utilizada a ferramenta com uma abordagem *ad hoc*.

As limitações descritas anteriormente abrem caminhos para trabalhos futuros na área. A adição do suporte a parâmetros de mensagem e composição de fragmentos combinados tem o potencial de viabilizar o uso da ferramenta em casos que apresentam diagramas mais complexos, além disso também poderia viabilizar a tradução dos fragmentos combinados que não foram abordados neste trabalho. A adição de todas essas funcionalidades à ferramenta possibilitaria uma abordagem mais elaborada para validação da ferramenta em um contexto real.

Outra possibilidade de trabalho futuro é a adaptação da tradução de diagramas. Apesar deste trabalho ter um foco em diagramas de sequência, seria possível definir regras formais de tradução visando reutilizar a mesma lógica de tradução em outros tipos de diagramas UML, tais como diagramas de atividade e diagramas de máquina de estado.

Além da possibilidade de evolução da ferramenta, este trabalho também abre portas para outras áreas de pesquisa. A linha de pesquisa de refatoração de modelos UML foi uma área considerada ao longo do desenvolvimento do trabalho. Trabalhos como (SUNYÉ et al., 2001) introduzem propostas interessantes de refatoração que poderiam ser adaptadas para métodos formais.

Referências

ABRIAL, J.-R.; ABRIAL, J.-R. *The B-book: assigning programs to meanings*. [S.l.]: Cambridge University Press, 2005. Citado na página 64.

BAIER, C.; KATOEN, J.-P. *Principles of model checking*. [S.l.]: MIT press, 2008. Citado na página 14.

BECK, K. *Extreme programming explained: embrace change*. [S.l.]: addison-wesley professional, 2000. Citado na página 13.

CABOT, J.; CLARISÓ, R.; RIERA, D. Umltocsp: a tool for the formal verification of uml/ocl models using constraint programming. In: ACM. *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. [S.l.], 2007. p. 547–548. Citado na página 64.

CHANGE-VISION. *Astah - Software Design Tools*. 2018. Disponível em: <http://astah.net/>. Citado 2 vezes nas páginas 15 e 30.

DONOVAN, A. A.; KERNIGHAN, B. W. *The Go programming language*. [S.l.]: Addison-Wesley Professional, 2015. Citado na página 25.

DUBOIS, P.; BY-WIDENIUS, M. F. *MySQL*. [S.l.]: New riders publishing, 1999. Citado na página 54.

EICHNER, C. et al. Compositional semantics for uml 2.0 sequence diagrams using petri nets. In: SPRINGER. *International SDL Forum*. [S.l.], 2005. p. 133–148. Citado 2 vezes nas páginas 64 e 65.

GIBSON-ROBINSON, T. et al. Fdr3—a modern refinement checker for csp. In: SPRINGER. *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. [S.l.], 2014. p. 187–201. Citado 2 vezes nas páginas 15 e 31.

GOSLING, J.; HOLMES, D. C.; ARNOLD, K. *The Java programming language*. [S.l.]: Addison-Wesley, 2005. Citado na página 16.

GROUP, T. O. *The Open Group Base Specifications Issue 7, 2018 edition*. 2018. Disponível em: <https://pubs.opengroup.org/onlinepubs/9699919799/>. Citado na página 54.

HAUSE, M. et al. The sysml modelling language. In: CITESEER. *Fifteenth European Systems Engineering Conference*. [S.l.], 2006. v. 9, p. 1–12. Citado na página 64.

HOARE, C. A. R. *Communicating Sequential Processes*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1985. ISBN 0-13-153271-5. Citado na página 14.

HOLZMANN, G. J. *The SPIN model checker: Primer and reference manual*. [S.l.]: Addison-Wesley Reading, 2004. v. 1003. Citado na página 64.

KENT, S.; EVANS, A.; RUMPE, B. Uml semantics faq. In: SPRINGER. *European Conference on Object-Oriented Programming*. [S.l.], 1999. p. 33–56. Citado na página 13.

KHENDEK, F.; BOURDUAS, S.; VINCENT, D. Stepwise design with message sequence charts. In: SPRINGER. *International Conference on Formal Techniques for Networked and Distributed Systems*. [S.l.], 2001. p. 19–34. Citado na página 12.

KIM, S.-K.; CARRINGTON, D. A formal mapping between uml models and object-z specifications. In: SPRINGER. *International Conference of B and Z Users*. [S.l.], 2000. p. 2–21. Citado na página 64.

LIMA, L.; DIDIER, A.; CORNÉLIO, M. A formal semantics for sysml activity diagrams. In: SPRINGER. *Brazilian Symposium on Formal Methods*. [S.l.], 2013. p. 179–194. Citado 2 vezes nas páginas 64 e 65.

LIMA, L.; IYODA, J.; SAMPAIO, A. A formal semantics for sequence diagrams and a strategy for system analysis. In: IEEE. *2014 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*. [S.l.], 2014. p. 317–324. Citado 2 vezes nas páginas 64 e 65.

LIMA, L.; IYODA, J.; SAMPAIO, A. Refinement verification of sequence diagrams using csp. In: SPRINGER. *Brazilian Symposium on Formal Methods*. [S.l.], 2016. p. 235–252. Citado 6 vezes nas páginas 15, 30, 32, 49, 64 e 65.

MCUMBER, W. E.; CHENG, B. H. Uml-based analysis of embedded systems using a mapping to vhdl. In: IEEE. *Proceedings 4th IEEE International Symposium on High-Assurance Systems Engineering*. [S.l.], 1999. p. 56–63. Citado na página 64.

MCUMBER, W. E.; CHENG, B. H. A general framework for formalizing uml with formal languages. In: IEEE COMPUTER SOCIETY. *Proceedings of the 23rd international conference on Software engineering*. [S.l.], 2001. p. 433–442. Citado na página 64.

NAVABI, Z. *VHDL: Analysis and modeling of digital systems*. [S.l.]: McGraw-Hill, Inc., 1997. Citado na página 64.

PETERSON, J. L. Petri nets. *ACM Computing Surveys (CSUR)*, ACM, v. 9, n. 3, p. 223–252, 1977. Citado na página 64.

POSTEL, J.; REYNOLDS, J. File transfer protocol. 1985. Citado na página 54.

ROSSI, F.; BEEK, P. V.; WALSH, T. *Handbook of constraint programming*. [S.l.]: Elsevier, 2006. Citado na página 64.

SCHWABER, K.; BEEDLE, M. *Agile software development with Scrum*. [S.l.]: Prentice Hall Upper Saddle River, 2002. v. 1. Citado na página 13.

SELIC, B. The pragmatics of model-driven development. *IEEE software*, IEEE, v. 20, n. 5, p. 19–25, 2003. Citado na página 64.

SMITH, G. *The Object-Z specification language*. [S.l.]: Springer Science & Business Media, 2012. v. 1. Citado na página 64.

SNOOK, C.; BUTLER, M. Uml-b: Formal modeling and design aided by uml. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, ACM, v. 15, n. 1, p. 92–122, 2006. Citado 2 vezes nas páginas 64 e 65.

SOMMERVILLE, I. *Software engineering*. [S.l.]: Addison-Wesley/Pearson, 2011. Citado na página 12.

STOREY, N. R. *Safety critical computer systems*. [S.l.]: Addison-Wesley Longman Publishing Co., Inc., 1996. Citado na página 13.

STÖRRLE, H.; HAUSMANN, J. H. Towards a formal semantics of uml 2.0 activities. *Software engineering*, v. 64, p. 117–128, 2005. Citado 2 vezes nas páginas 64 e 65.

SUNYÉ, G. et al. Refactoring uml models. In: SPRINGER. *International Conference on the Unified Modeling Language*. [S.l.], 2001. p. 134–148. Citado na página 66.

UML, O. *Unified Modeling Language TM (UML®) Version 2.5*. 2015. Citado 4 vezes nas páginas 12, 14, 23 e 44.

WARMER, J. B.; KLEPPE, A. G. *The object constraint language: getting your models ready for MDA*. [S.l.]: Addison-Wesley Professional, 2003. Citado na página 64.

WING, J. M. A specifier's introduction to formal methods. *Computer*, IEEE, v. 23, n. 9, p. 8–22, 1990. Citado na página 13.

WINSKEL, G. *The formal semantics of programming languages: an introduction*. [S.l.]: MIT press, 1993. Citado na página 13.

WOODCOCK, J. et al. Features of cml: A formal modelling language for systems of systems. In: IEEE. *2012 7th International conference on system of systems engineering (SoSE)*. [S.l.], 2012. p. 1–6. Citado na página 64.