Amaury Tavares Ribeiro Júnior

Verificação de Deadlock e Não-Determinismo em Ações de SysML 2.0

Recife

2021

Amaury Tavares Ribeiro Júnior

Verificação de Deadlock e Não-Determinismo em Ações de SysML 2.0

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

Universidade Federal Rural de Pernambuco – UFRPE

Departamento de Computação

Curso de Bacharelado em Ciência da Computação

Orientador: Lucas Albertins de Lima

Recife

2021

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)

A489v Júnior, Amaury Tavares Ribeiro
Verificação de Deadlock e Não-Determinismo em Ações de SysML 2.0 / Amaury Tavares Ribeiro Júnior. - 2021.
64 f. : il.

Orientador: Lucas Albertins de Lima.
Inclui referências e apêndice(s).

Trabalho de Conclusão de Curso (Graduação) - Universidade Federal Rural de Pernambuco, Bacharelado em Ciência da Computação, Recife, 2021.

1. SysML 2.0. 2. CSP. 3. FDR. 4. deadlock. 5. não-determinismo. I. Lima, Lucas Albertins de, orient. II. Título

CDD 004

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

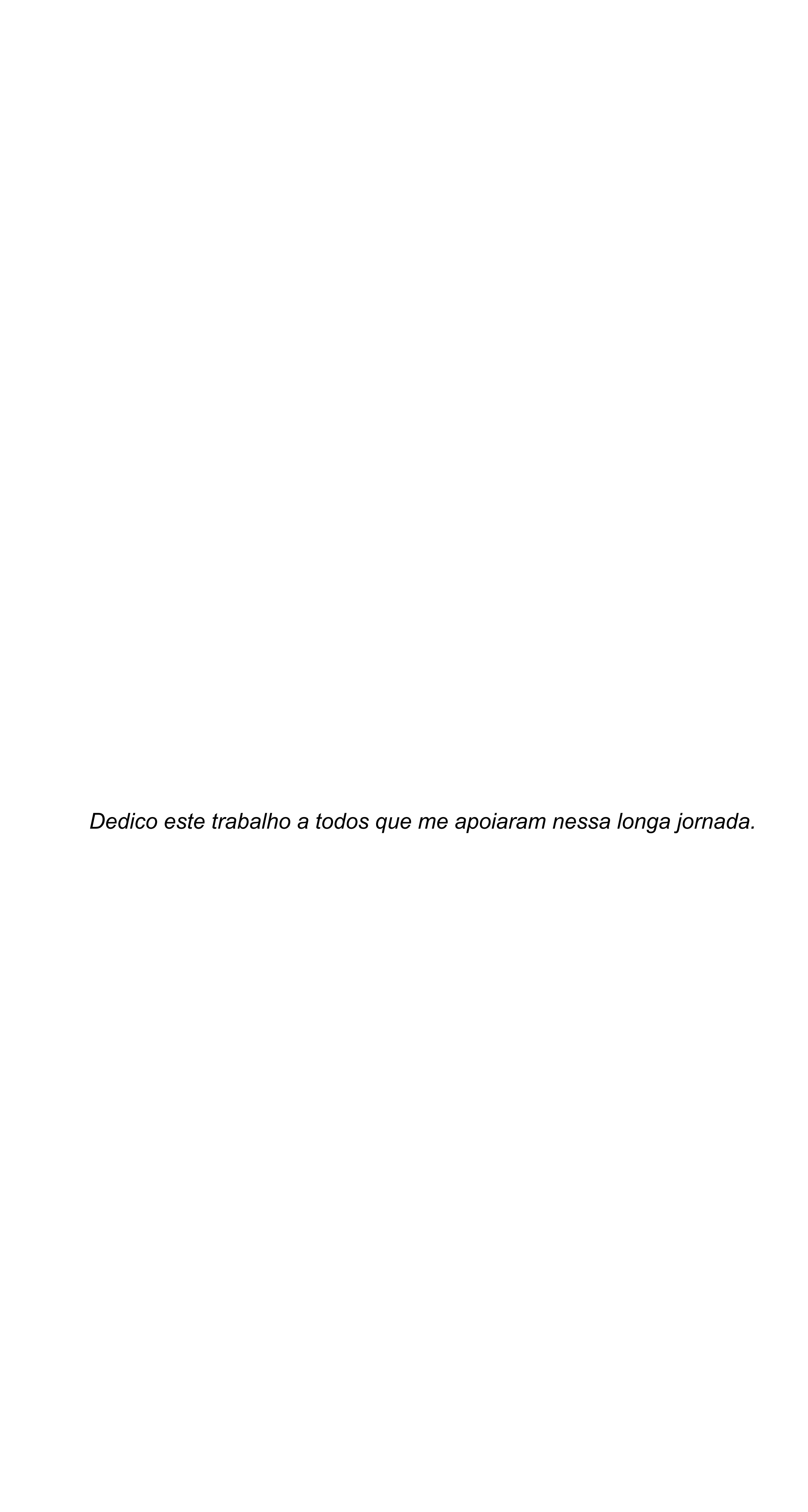
Trabalho defendido por **Amaury Tavares Ribeiro Júnior** às 8:00 horas do dia 15 de julho, via Google Meet, como requisito para conclusão do curso de Bacharelado em Ciência da Computação da Universidade Federal Rural de Pernambuco, intitulado "**Verificação de Deadlock e Não-Determinismo em Ações de SysML 2.0**", orientado pelo Prof. Lucas Albertins de Lima e aprovado pela seguinte banca examinadora:

____

Lucas Albertins de Lima
DC/UFRPE

____

Sidney de Carvalho Nogueira
DC/UFRPE



Dedico este trabalho a todos que me apoiaram nessa longa jornada.

Agradecimentos

Primeiramente, agradeço aos meus pais, por terem me proporcionado uma vida digna, confortável e repleta de ensinamentos, sendo os principais responsáveis por me fazer chegar até este momento. Sem eles, eu não teria forças para enfrentar as batalhas diárias que a vida nos submete.

Agradeço também ao meu orientador pela paciência, esforço e competência na orientação e incentivo que tornaram possível a realização deste trabalho de conclusão de curso.

Por fim, agradeço a instituição Universidade Federal Rural de Pernambuco, incluindo todos os professores e amigos, por terem sido a espinha dorsal da minha formação acadêmica, uma conquista que me fez evoluir como homem e adquirir sabedoria e conhecimentos que vou levar para resto da minha vida.

“O sucesso é uma consequência e não um objetivo.”
(Gustave Flaubert)

Resumo

A crescente complexidade dos sistemas tem levado a um esforço crescente para validá-los. Focar em iniciativas para criar ferramentas para a identificação de problemas o mais cedo possível vem sendo uma abordagem bastante desejada para minimizar custos e esforços. Alguns problemas como *deadlock* e não-determinismo podem se tornar cada vez mais difíceis de detectar devido à natureza concorrente e distribuída que os sistemas podem apresentar. A linguagem SysML 2.0 com sua notação para ações que vem sendo desenvolvida pela OMG pode ser usada para modelar comportamentos, mesmo os concorrentes, o que os torna adequados para descrever a dinâmica desses sistemas. Vários trabalhos propõem semântica formal a modelos SysML 1.0 para fins de verificação, incluindo verificação de *deadlock*. Mas nossa proposta é distinta no fato de fornecermos uma semântica formal para ações de SysML 2.0 que não apenas verifica a presença de *deadlocks*, mas também não-determinismo. Este último é geralmente negligenciado na literatura, embora possa ser considerado relevante em arquiteturas complexas de sistemas. Todo este processo de verificação é automatizado provendo também total rastreabilidade de volta para SysML 2.0 em caso de ser detectado problema no modelo. Portanto, o usuário não precisa entender ou manipular notações formais em qualquer parte do processo. Sendo assim, nossa principal contribuição é um verificador para analisar propriedades de ações de SysML 2.0, especificamente *deadlock* e não-determinismo, não exigindo nenhum conhecimento da semântica formal subjacente.

Palavras-chave: SysML 2.0, KerML, CSP, FDR, *deadlock*, não-determinismo, verificação.

Abstract

The growing complexity of systems has led to an increasing effort to validate them. Focusing on initiatives for creating tools to identify problems as early as possible has been a very desirable approach to minimize costs and efforts. Some problems like deadlock and non-determinism can become increasingly difficult to detect due to the concurrent and distributed nature that systems can present. The SysML 2.0 language has been developed by OMG. It provides notation for actions that can be used to model behaviors, even concurrent ones, which makes them suitable for describing the dynamics of these systems. Several works propose formal semantics to SysML 1.0 models for verification purposes, including deadlock verification. But our proposal is distinct in that we provide a formal semantics for SysML 2.0 actions that not only check for the presence of deadlocks, but also non-determinism. The latter is generally neglected in the literature, although it can be considered relevant in complex system architectures. This entire verification process is automated and also provides full traceability back to SysML 2.0 in case a problem is detected in the model. Therefore, the user does not need to understand or manipulate formal notations in any part of the process. Therefore, our main contribution is a checker for analyzing properties of SysML 2.0 actions, specifically deadlock and non-determinism, not requiring any knowledge on the underlying formal semantics.

Keywords: SysML 2.0, KerML, CSP, FDR, deadlock, nondeterminism, verification.

Lista de ilustrações

Figura 1 – Exemplo de deadlock em um diagrama de atividade 15
Figura 2 – Definição da ação deadlock2 na linguagem textual SysML 2.0 . . . 16
Figura 3 – Contraexemplo da ação deadlock2 na linguagem textual SysML 2.0 16
Figura 4 – Representação visual do contraexemplo da ação deadlock2 16
Figura 5 – Exemplo de não-determinismo em um diagrama de atividade 17
Figura 6 – Definição da ação nonDeterminism1 na linguagem textual SysML 2.0 17
Figura 7 – Contraexemplo da ação nonDeterminism1 na linguagem textual SysML 2.0 17
Figura 8 – Representação visual do contraexemplo da ação nonDeterminism1 17
Figura 9 – Estrutura da SysML 2.0 (OMG, 2020b) 20
Figura 10 – Mapeamento da sequência de eventos retornada pelo FDR 24
Figura 11 – Visão geral do processo do verificador 25
Figura 12 – Arquitetura do verificador 26
Figura 13 – Definição dos processos de um *node* 28
Figura 14 – Estrutura da especificação CSP apresentada em (LIMA; TAVARES; NOGUEIRA, 2020) 28
Figura 15 – Adaptadores SysML 2.0 30
Figura 16 – Especificação SysML 2.0 da ação do carregador de bateria 38
Figura 17 – Representação visual da ação do carregador de bateria 39
Figura 18 – Contraexemplo SysML 2.0 da ação do carregador de bateria 39
Figura 19 – Representação visual do contraexemplo da ação do carregador de bateria 40
Figura 20 – Modificação da especificação SysML 2.0 da ação do carregador de bateria 40
Figura 21 – Representação visual da modificação da ação do carregador de bateria 40
Figura 22 – Contraexemplo da modificação SysML 2.0 da ação do carregador de bateria 41
Figura 23 – Representação visual do contraexemplo da modificação da ação do carregador de bateria 41
Figura 24 – Especificação SysML 2.0 da ação do sistema de freios 42
Figura 25 – Representação visual da ação do sistema de freios 42
Figura 26 – Contraexemplo SysML 2.0 da ação do sistema de freios 43
Figura 27 – Representação visual do contraexemplo da ação do sistema de freios 44
Figura 28 – Representação visual da ação do sistema de vendas online 45
Figura 29 – Representação visual do contraexemplo da ação do sistema de vendas online 46

Figura 30 – Representação visual da correção da ação do sistema de vendas online 46
Figura 31 – Representação visual da ação do plano de produção de produtos . 47
Figura 32 – Representação visual do contraexemplo da ação do plano de produção de produtos 49
Figura 33 – Especificação SysML 2.0 da solução da ação do plano de produção de produtos 50
Figura 34 – Representação visual da solução da ação do plano de produção de produtos 50
Figura 35 – Especificação SysML 2.0 da ação para o sistema de vendas online 58
Figura 36 – Contraexemplo SysML 2.0 da ação para o sistema de vendas online 59
Figura 37 – Especificação SysML 2.0 da ação para o plano de produção de produtos 60
Figura 38 – Contraexemplo SysML 2.0 da ação para o plano de produção de produtos 61

Lista de tabelas

Tabela 1 – Fator de custos para correção do problema em fases do projeto de acordo com (HASKINS et al., 2004) 12
Tabela 2 – Nós da SysML representados na linguagem textual SysML 2.0 . . . 22
Tabela 3 – Traduções dos nós de objetos 31
Tabela 4 – Traduções das arestas 32
Tabela 5 – Traduções dos nós de ações 33
Tabela 6 – Traduções dos nós de controles 35

Lista de abreviaturas e siglas

UML	Unified Modeling Language
SysML	System Modeling Language
API	Application Programming Interface
CSP	Communicating Sequential Processes
FDR	Failures Divergences Refinement
LTS	Labeled Transition System
OMG	Object Management Group
INCOSE	International Council on Systems Engineering
JVM	Java Virtual Machine
CTS	Configuration Transition System
PLTL-X	Past Linear TemporalLogic
fUML	Foundational Subset for Executable UML
PCTL	Probabilistic Computation Tree Logic

Sumário

	Lista de ilustrações	**7**
1	**INTRODUÇÃO**	**12**
2	**REFERÊNCIAL TEÓRICO**	**18**
2.1	**SysML 2.0**	**18**
2.2	**CSP**	**22**
3	**VERIFICADOR DE AÇÕES SYSML 2.0**	**25**
3.1	**Visão Geral do Processo de Verificação de Ações SysML 2.0**	**25**
3.2	**Tradução de Ações SysML 2.0**	**27**
3.2.1	Xtext	28
3.2.2	Nós de Objetos	29
3.2.3	Arestas	30
3.2.4	Nós de Ações	32
3.2.5	Nós de Controles	34
3.3	**Verificação e Rastreabilidade**	**36**
4	**ESTUDOS DE CASOS**	**38**
5	**CONCLUSÃO**	**51**
5.1	**Trabalhos Relacionados**	**52**
5.2	**Trabalhos Futuros**	**53**
	REFERÊNCIAS	**55**
A	**ESPECIFICAÇÃO SYSML 2.0 DA AÇÃO PARA O SISTEMA DE VENDAS ONLINE**	**58**
B	**CONTRAEXEMPLO SYSML 2.0 DA AÇÃO PARA O SISTEMA DE VENDAS ONLINE**	**59**
C	**ESPECIFICAÇÃO SYSML 2.0 DA AÇÃO PARA O PLANO DE PRODUÇÃO DE PRODUTOS**	**60**
D	**CONTRAEXEMPLO SYSML 2.0 DA AÇÃO PARA O PLANO DE PRODUÇÃO DE PRODUTOS**	**61**

1 Introdução

O fator de custo para resolver um problema aumenta conforme os erros são identificados em fases mais tardes do projeto (HASKINS et al., 2004). Conforme apresentado na Tabela 1, o fator de custo para a correção do problema aumenta consideravelmente a medida que o projeto vai avançando nas fases, comparado com a fase de Requisitos, o custo pode ser até 8x maior na fase de *Design*, 16x maior na fase de Construção, 78x maior na fase de Testes e podendo atingir um custo de até 1500x maior na fase de Operação. Além disso, como a complexidade dos sistemas vem crescendo cada vez mais, focar o esforço em criar ferramentas para a identificação de problemas o mais cedo possível vem sendo uma abordagem bastante desejada para minimizar custos e esforços.

Fase	Fator de custo
Requisitos	1x
Design	3x até 8x
Construção	7x até 16x
Testes	21x até 78x
Operação	29x até 1500x

Tabela 1 – Fator de custos para correção do problema em fases do projeto de acordo com (HASKINS et al., 2004)

A *Unified Modeling Language* (UML) é considerada a linguagem padrão de modelagem de propósito geral que é muito utilizada na modelagem e *design* de sistemas de *software*. A UML possui vários tipos de diagramas que são divididos em 2 categorias, sendo elas diagramas comportamentais e diagramas estruturais. De acordo com (REGGIO et al., 2013), o diagrama de atividades é um dos diagramas mais utilizados. O diagrama de atividades é um diagrama comportamental que representa aspectos dinâmicos do sistema, sendo normalmente utilizados para descrever as funcionalidades e fluxos dos sistemas, incluindo mecanismos de concorrência. Ele é representado por arestas, nós de controles, nós de ações e nós de objetos. As arestas de um diagrama de atividades podem ser utilizadas tanto para controlar seu fluxo de execução quanto para a comunicação de dados entre os nós do diagrama.

Segundo (HAUSE et al., 2006), a *System Modeling Language* (SysML) é uma extensão da UML com o objetivo de prover suporte a especificação, análise, *design*, verificação e validação de sistemas. Enquanto a UML é centrada no *software*, SysML tem o foco não apenas no *software*, mas na engenharia de sistemas em geral, os quais podem ser sistemas físicos. A SysML mantém alguns diagramas da UML, como o diagrama de atividades, mas também inclui novos diagramas, como por exemplo,

diagramas de requisitos e diagramas paramétricos. De acordo com (OMG, 2017), a SysML 2.0 pretende melhorar a SysML 1.0 em termos de precisão, expressividade, interoperabilidade, consistência e integração, fornecendo uma notação textual padrão da linguagem (OMG, 2020b) e também uma *Application Programming Interface* (API) SysML 2.0 (OMG, 2020c) destinada a aumentar ainda mais a interoperabilidade, especificando alguns serviços padrões para manipular os modelos SysML 2.0.

Com o passar do tempo, os projetos estão se tornando cada vez mais amplos e heterogêneos, nos quais diversos fatores devem ser considerados, um deles é a concorrência. Embora a UML seja boa para modelagem de sistemas, ela possui apenas uma descrição em linguagem natural, não sendo possível automatizar sua análise, com isso a verificação dos modelos tem de ser realizada de forma manual, podendo trazer problemas como ambiguidade. Por outro lado, verificadores de modelo (*model checkers*) são automatizados e não possuem ambiguidade, porém requerem que o usuário tenha conhecimento sobre como manipular a notação da linguagem formal que é usada pelo verificador. O verificador de modelo usa a abordagem de analisar todas as possibilidades de uma máquina de estados, com o objetivo de validar a satisfatibilidade de uma propriedade sendo analisada (BAIER; KATOEN, 2008).

Duas das propriedades relacionada à concorrência e que são verificadas na aborgadem utilizada em (LIMA; TAVARES; NOGUEIRA, 2020) é o *deadlock* e o não-determinismo, que são dois dos problemas mais comuns em sistemas que possuem algum tipo de concorrência. De acordo com (ROSCOE; HOARE; BIRD, 1997), um sistema está em *deadlock* se nenhum progresso pode ser feito, podemos entender melhor o que é o *deadlock* com um exemplo muito conhecido chamado de Jantar dos Filósofos. No Jantar dos Filósofos 5 filósofos estão sentados em uma mesa redonda para jantar, cada filósofo tem um prato com espaguete à sua frente, cada prato possui um garfo para pegar o espaguete. O espaguete está muito escorregadio e, para que um filósofo consiga comer, será necessário utilizar dois garfos. Cada filósofo alterna entre duas tarefas: comer ou pensar. Quando um filósofo fica com fome, ele tenta pegar os garfos à sua esquerda e à sua direita, um de cada vez, independente da ordem. Caso ele consiga pegar dois garfos, ele come durante um determinado tempo e depois recoloca os garfos na mesa, em seguida ele volta a pensar. O problema do *deadlock* acontence quando cada um dos filósofos pega o primeiro garfo para comer, mas não consegue pegar o segundo garfo por não ter mais garfos disponíveis na mesa, com isso todos os filósofos acabam ficando em um estado de espera indefinida pelo segundo garfo para conseguir comer.

O não-determinismo é importante para fins de abstração, e de acordo com (KLOETZER; BELTA, 2008), um sistema é não-determinístico se a partir de um determinado estado pode-se ir para mais de um estado com a mesma entrada.

A maior parte das abordagens, como as de (BALDAN; CORRADINI; GADDUCCI, 2004), (ELMANSOURI; HAMROUCHE; CHAOUI, 2011), (ESHUIS, 2006), (ABDELHALIM et al., 2010) e (OUCHANI; MOHAMED; DEBBABI, 2014) seguem o padrão de traduzir o diagrama de atividade para uma linguaguem que seja possível de verificar propriedades e que seja equivalente ao diagrama, e em seguida usam um verificador de modelos para analisar essas propriedades. A abordagem em (LIMA; TAVARES; NOGUEIRA, 2020) traduz o diagrama de atividade para uma especificação equivalente na linguagem formal *Communicating Sequential Processes* (CSP) (HOARE, 1985), que é composto de elementos independentes, que se sincronizam e que se comunicam através de canais. Após isso é utilizado o verificador de modelos *Failures-Divergences Refinement* (FDR) (GIBSON-ROBINSON et al., 2014) para verificação de *deadlock* e não-determinismo na linguagem formal CSP. O FDR retorna uma sequência de eventos para ilustrar o caminho até o ponto onde ocorreu o *deadlock* ou não-determinismo, por fim esta sequência de eventos é traduzida para um contraexemplo em diagrama de atividades que possui sua arestas e nós percorridos pelo FDR pintados de vermelho e é retornado para o usuário. Assim, o usuário não necessita ter nenhum conhecimento sobre a linguagem formal para fazer a verificação automática de *deadlock* e não-determinismo.

Uma das limitações do trabalho apresentado em (LIMA; TAVARES; NOGUEIRA, 2020) é que a ferramenta construída está vinculada ao ambiente de modelagem UML Astah (VISION, 2019), impedindo que seja utilizada em outros ambientes e ferramentas de modelagem. Contudo, devido a linguagem SysML 2.0 possuir uma notação textual padrão que será utilizada pelos diversos ambientes, esperamos neste trabalho criar um novo verificador utilizando a abordagem previamente construída em (LIMA; TAVARES; NOGUEIRA, 2020) para utilizar a notação textual especificada na SysML 2.0 como entrada da verificação, e também construir uma rastreabilidade para gerar um contraexemplo em ações de SysML 2.0. Assim, tornando a abordagem não mais vinculada apenas ao ambiente de modelagem UML Astah, mas podendo ser integrada a qualquer ambiente e ferramenta de modelagem UML que use a notação textual especificada na SysML 2.0.

É possível observar um exemplo dessa verificação no diagrama de atividade da Figura 1, que é composto por 5 nós, sendo eles da esquerda para a direita, o primeiro um nó *initial*, o segundo um nó *action* nomeado de `act1`, o terceiro outro nó *action* nomeado de `act2`, o quarto um nó *decision* e por último um nó *final*. Este diagrama possui um *deadlock* pelo fato de que o nó *action* `act1` precisa que todas as suas arestas de entrada (sendo uma delas que vem do nó *initial* e outra que vem do nó *decision*) sejam completadas antes de completar suas arestas de saída, e neste caso é possível ver que a aresta de entrada que vem do nó *decision* só pode ser completada quando o fluxo atingir o nó *decision*, causando assim um deadlock no nó `act1`.

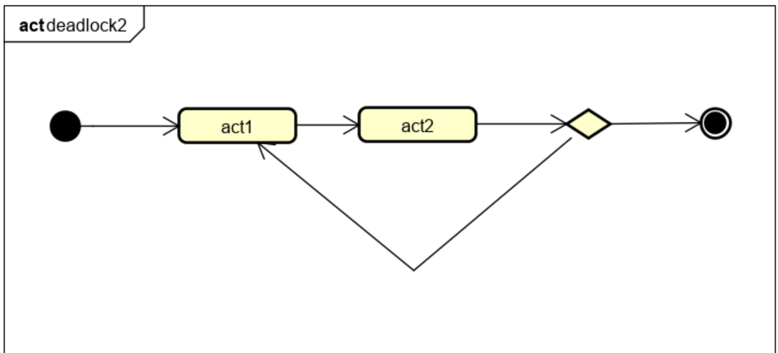


Figura 1 – Exemplo de deadlock em um diagrama de atividade

O diagrama de atividade da Figura 1 é representado na linguagem textual SysML 2.0 da forma como é apresentado na Figura 2. A linguagem textual SysML 2.0 não possui mais o conceito de atividades, mas sim ações (*actions*). O que era conhecido como atividades agora é uma definição de ação (sendo *action def* na linguagem textual), e o uso de atividades agora é representado como um *action usage* (sendo apenas a palavra *action* na linguagem textual), porém não é obrigatório uma ação possuir uma definição para ser executada.

O novo verificador construído passará a receber como entrada a ação na linguagem textual SysML 2.0, e em seguida irá traduzir o diagrama para a linguagem formal CSP e utilizar o checador de modelos FDR para verificar a propriedade de *deadlock*, após esse processo o FDR irá retornar a sequência de eventos que foi realizada até encontrar o *deadlock*, e por fim será construído um contraexemplo da ação na linguagem textual SysML 2.0, adicionando `"// [deadlock trace]"` ao final de cada nó e aresta percorrida pelo FDR. Como pode-se observar na Figura 3, foi percorrido apenas um nó e uma aresta, com isso o usuário pode identificar que o nó `act1` não foi percorrido e provavelmente é a causa do *deadlock*. Com esse contraexemplo na linguagem textual SysML 2.0 existe a possibilidade de uma ferramenta que suporte SysML 2.0 leia a informação do *deadlock* e marque no diagrama o caminho até ele, como por exemplo a Figura 4, que marca os nós e arestas percorridas com a cor vermelha.

É possível observar outro exemplo dessa verificação no diagrama de atividade da Figura 5, que é composto por 5 nós, sendo eles da esquerda para a direita, o primeiro um nó *initial*, o segundo um nó *action* nomeado de `act1`, o terceiro um nó *decision*, o quarto um nó *action* nomeado de `act2` e por último um nó *final*. Este diagrama possui um não-determinismo pelo fato de que a partir do nó *decision* é possível chegar ao nó *final* e ao nó *action* `act2` de uma forma não-determinística.

O diagrama de atividade da Figura 5 é representado na linguagem textual SysML 2.0 da forma como é apresentado na Figura 6. O novo verificador irá seguir o mesmo processo feito no exemplo do *deadlock*. Como pode-se observar na Figura 7, foi percor-

```
action def deadlock2 {
    first start;
    succession start then act1;
    action act1;
    succession act1 then act2;
    action act2;
    succession act2 then decisionNode;
    decide decisionNode;
        then act1;
        then done;
}
```

Figura 2 – Definição da ação deadlock2 na linguagem textual SysML 2.0

```
action def deadlock2 {
    first start; // [deadlock trace]
    succession start then act1; // [deadlock trace]
    action act1;
    succession act1 then act2;
    action act2;
    succession act2 then decisionNode;
    decide decisionNode;
        then act1;
        then done;
}
```

Figura 3 – Contraexemplo da ação deadlock2 na linguagem textual SysML 2.0

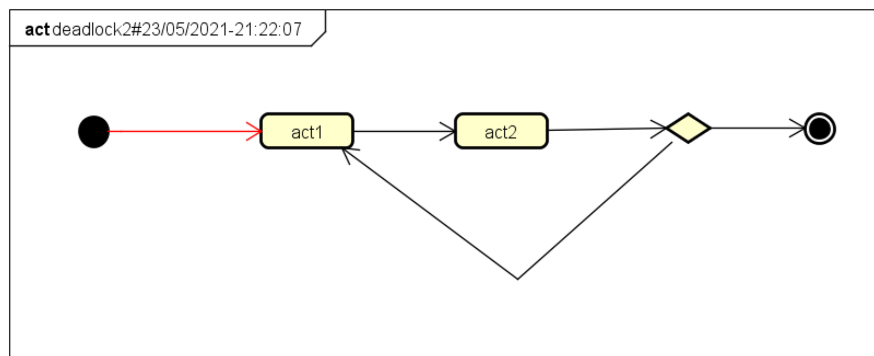


Figura 4 – Representação visual do contraexemplo da ação deadlock2

rido alguns nós e arestas, com isso o usuário pode identificar que o nó `decisionNode` foi o último percorrido e provavelmente é a causa do não-determinismo. Esse contraexemplo na linguagem textual SysML 2.0 também pode ser apresentado de uma forma visual por alguma ferramenta que suporte SysML 2.0, como por exemplo a Figura 8, que marca os nós e arestas percorridas com a cor vermelha.

O restante deste trabalho está organizado da seguinte forma. A Seção 2 apresenta o referêncial teórico. Seção 3 apresenta os detalhes d verificador construído. Seção 4 ilustra o verificador em estudos de casos. Seção 5 conclui, apresenta os trabalhos relacionados, discute limitações e descreve possíveis trabalhos futuros.

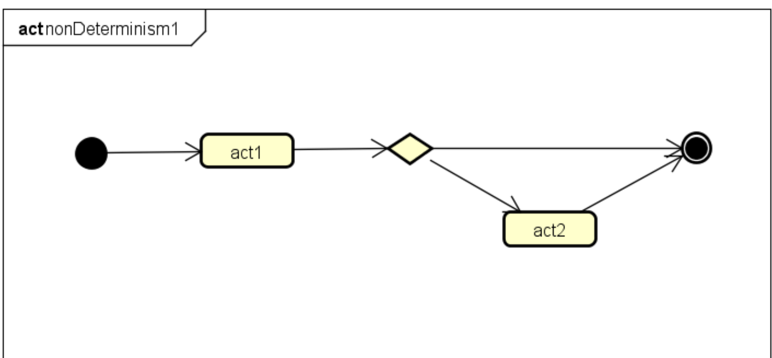


Figura 5 – Exemplo de não-determinismo em um diagrama de atividade

```
action def nonDeterminism1 {
    first start;
    succession start then act1;
    action act1;
    succession act1 then decisionNode;
    decide decisionNode;
        then act2;
        then done;
    action act2;
    succession act2 then done;
}
```

Figura 6 – Definição da ação nonDeterminism1 na linguagem textual SysML 2.0

```
action def nonDeterminism1 {
    first start; // [non-determinism trace]
    succession start then act1; // [non-determinism trace]
    action act1; // [non-determinism trace]
    succession act1 then decisionNode; // [non-determinism trace]
    decide decisionNode; // [non-determinism trace]
        then act2;
        then done;
    action act2;
    succession act2 then done;
}
```

Figura 7 – Contraexemplo da ação nonDeterminism1 na linguagem textual SysML 2.0

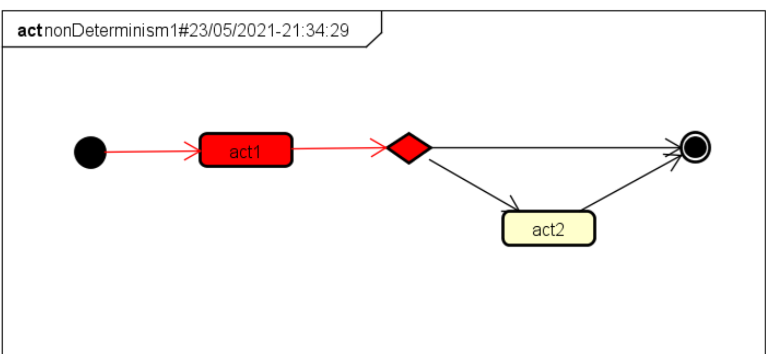


Figura 8 – Representação visual do contraexemplo da ação nonDeterminism1

2 Referêncial Teórico

Nesta Seção apresentamos o referêncial teórico necessário para entender o trabalho, na Subseção 2.1 iremos apresentar o quê é a SysML e quais as diferenças entre a versão 1.0 e 2.0, também apresentamos as vantagens de utilizá-la e os nós e arestas que são traduzidos para a linguagem textual SysML 2.0 que são utilizados neste trabalho. Além disto, na Subseção 2.2 falaremos sobre o quê é a linguagem formal CSP, mostramos alguns conceitos da linguagem e por fim falaremos o quê é e como funciona o checador de modelos FDR.

2.1 SysML 2.0

De acordo com a *Object Management Group* (OMG), a *Unified Modeling Language* (UML) é uma linguagem de modelagem visual de propósito geral que é projetada para especificar, visualizar, construir e documentar os artefatos de sistemas de *software*. A UML foi adotada pela OMG em 1997 e desde então vem sendo utilizada para modelar várias fases de um sistema de *software*, desde os primeiros contatos com o cliente até a geração do código. Segundo (HAUSE et al., 2006), a UML 2.0 veio com objetivo de resolver alguns problemas encontrados na versão 1.0 e também aprimorar alguns diagramas, como por exemplo o diagrama de estrutura composta que fornece uma arquitetura hierárquica, entretanto, originalmente permitia apenas um nível de hierarquia e não permitia a modelagem de fluxos em *links*.

A *System Modeling Language* (SysML) foi adotada pela OMG em 2006, essa adoção foi em resposta à solicitação *Request for Proposal* emitida pela OMG e a *International Council on Systems Engineering* (INCOSE) para uma versão customizada da UML 2.0 projetada para atender às necessidades específicas dos engenheiros de sistema. Com isso a SysML 1.0 (OMG, 2007) reutiliza um subconjunto de conceitos e diagramas da UML 2.0 e extende com alguns novos diagramas e construções apropriadas para modelagem de sistemas, sendo assim a SysML 1.0 é uma linguagem de modelagem visual que estende a UML 2.0 para dar suporte à especificação, análise, projeto, verificação e validação de sistemas complexos que incluem componentes de *hardware*, *software*, dados, procedimentos e instalações.

De acordo com (OMG, 2017), existem algumas limitações significativas do formalismo usado para SysML 1.0 que resultam em ambiguidades de interpretação. Por exemplo, SysML 1.0 não inclui um mapeamento formal completo entre a sintaxe concreta e a sintaxe abstrata, o que pode resultar em ambiguidade em como um diagrama SysML está em conformidade com as regras da gramática. Além disso, a semântica

do SysML 1.0 geralmente é definida em inglês, em vez de uma representação formal mais precisa, o que pode resultar em ambiguidade de significado.

Em contraste, a SysML 2.0 (OMG, 2020b) terá uma especificação mais formal de sua sintaxe abstrata, sintaxe concreta, semântica e os mapeamentos entre elas. Para maximizar a flexibilidade desta especificação, a abordagem necessária é especificar um pequeno conjunto de conceitos fundamentais e sua semântica de base usando uma semântica declarativa matemática. A semântica de base utilizada na SysML 2.0 é a KerML (OMG, 2020a) que fornece uma base comum para a criação de novas linguagens de modelagem (ou evolução das linguagens de modelagem existentes), na Figura 9 pode-se observar que a SysML 2.0 é construída como uma extensão da KerML. A sintaxe abstrata SysML 2.0 estende a sintaxe abstrata *Kernel*, fornecendo construções especializadas para sistemas de modelagem. Além disso, a biblioteca de modelos da SysML 2.0 (*Systems Library*) estende a biblioteca de modelos da *Kernel* (*Kernel Library*) para fornecer a especificação semântica para SysML 2.0. Finalmente, a SysML 2.0 fornece um conjunto adicional de bibliotecas de domínio (*Domain Libraries*) para fornecer um conjunto rico de modelos de referência em vários domínios importantes para a modelagem de sistemas (como quantidades, unidades e geometria básica). Em seguida, as bibliotecas de modelo escritas na própria SysML 2.0, baseadas na semântica de base, são usadas para estender ainda mais os conceitos da linguagem e suas semânticas associadas. Essas extensões representarão os principais conceitos de domínio da linguagem SysML 2.0. Tendo como objetivo fornecer uma interpretação sintática e semântica uniforme para a linguagem. Ou seja, um modelo SysML 2.0 deve ser interpretado de forma consistente e sujeito a uma avaliação objetiva para saber se está em conformidade com a especificação SysML 2.0, seja esta interpretação feita por um humano que interpreta uma visão do modelo, ou uma máquina que interpreta o modelo.

A vantagem de basear a semântica SysML em uma abordagem declarativa é que técnicas bem conhecidas de lógica matemática podem ser usadas para fazer deduções formais com base nas afirmações feitas em um modelo, a fim de provar coisas que são verdadeiras ou não sobre o sistema ou domínio que está sendo modelado. A semântica declarativa contrasta com a semântica operacional que especifica como um modelo é executado, de forma que os resultados da execução são avaliados para determinar como o sistema se comportará. Na Figura 9, também é possível observar as estruturas da linguagem SysML 2.0, uma das principais estruturas e a que vamos focar neste trabalho é a *Action* que é dividida em duas partes uma *ActionDefinition* e uma *ActionUsage*, uma *ActionDefinition* pode especificar o que uma ação (*action*) faz em termos de sua transformação de itens de entrada em itens de saída, e quando uma *ActionUsage* executa uma *ActionDefinition*, ela transforma seus itens de entrada em seus itens de saída de acordo com essa definição.

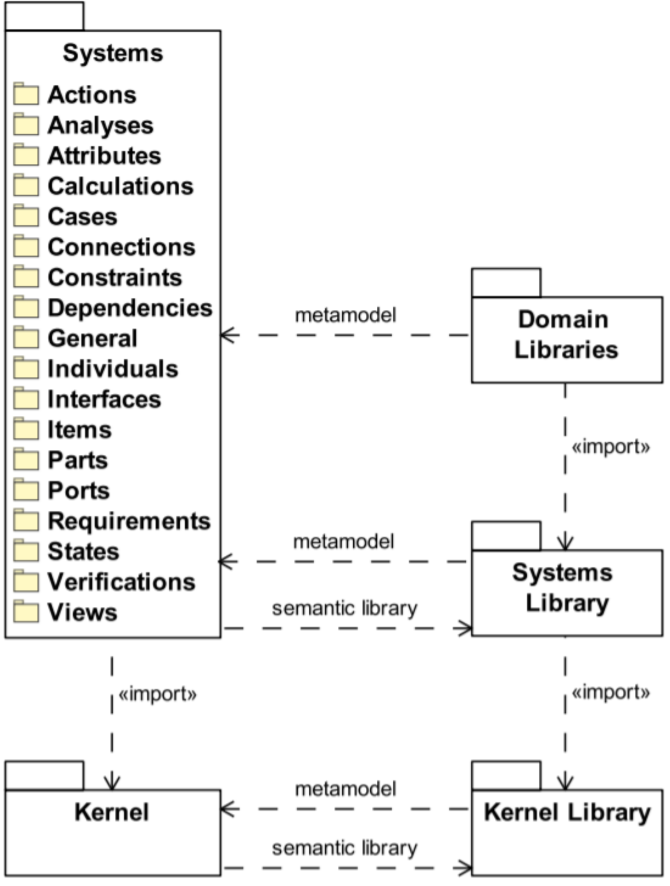


Figura 9 – Estrutura da SysML 2.0 (OMG, 2020b)

Na Tabela 2, é possível observar os principais elementos da estrutura *Action* que foram adaptados neste trabalho. Primeiro falaremos do nó *action*, que representa um nó básico de comportamento em que se espera a realização de uma determinada tarefa. Um nó *action* pode ter a especificação do seu comportamento previamente definida e depois diversas invocações podem ser realizadas a este comportamento. Em SysML 1.0, esta especificação é o que se chama de atividade e cada invocação também é conhecida como nós *call behavior*. Em SysML 2.0 só existe o conceito de ações (*actions*). Ao se utilizar a palavra chave `action def` é criado uma *ActionDefinition* e ao se utilizar a palavra chave `action` é criado uma *ActionUsage*, contudo, não é obrigatório existir uma *ActionDefinition* para fazer o uso da *ActionUsage*. No entanto, quando existe uma *action def* que depois é referenciado por um ou mais nós *action*, entendemos que estes últimos são nós de invocação, comumente conhecidos como *call behavior*. O nó *send signal* representa um nó que envia um sinal (dado) para um nó *accept event*, e é definido utilizando a palavra chave `action` seguido do nome do nó, com a adição da palavra chave `send` seguido da referência do *pin* ou *parameter* de entrada, por exemplo `signal1::x` onde o nó `signal1` tem o *pin* `x`, por fim é utilizado a palavra chave `to` seguido do nome do nó do tipo *accept event*. O nó *accept event* representa um nó que recebe um sinal (dado) enviado por um nó *send signal* e após isso

continua o seu fluxo, e é definido utilizando a palavra chave `action` seguido do nome do nó e indicando o dado que é aceito por ele com a palavra chave `accept` seguido pelo nome do dado e tipo, como por exemplo `y : Integer`, esse dado é acessado através de um *pin* de saída.

E em relação aos nós de objetos, o *pin* representa um parâmetro de entrada ou saída de uma *action*, e pode ser definido dentro de parênteses em nós *actions* utilizando a palavra chave `in` para indicar se é um *pin* de entrada ou `out` para indicar se é um *pin* de saída, seguido do nome do nó e o seu tipo, como por exemplo `in x : Integer`. Dentro do corpo da *action* é possível indicar o valor de um *pin* de saída, é necessário especificar a linguagem que será utilizada na expressão utilizando a palavra chave `language` seguido do nome da linguagem entre aspas, em seguida é possível definir a expressão da seguinte forma `/* expressão */`. O outro nó de objeto é o *parameter* que representa um parâmetro de entrada ou saída de uma definição de ação, e que pode ser definido da mesma forma que os *pins*, porém apenas sendo possível dentro de parênteses de um nó *call behavior*.

Em relação aos nós de controles, o nó *initial* representa o começo de um processo ou fluxo de trabalho de uma ação, e é definido utilizando a palavra chave `first` seguido do nome do nó. O nó *final* marca o estado final de uma ação e representa a conclusão de todos os fluxos de um processo, entretanto, já é definido na base da linguagem textual SysML 2.0 sendo chamado de *done* e segue o mesmo padrão de definição que um nó *action*. O nó *merge* representa a fusão de diferentes fluxos, podendo ter vários fluxos de entrada e apenas um fluxo de saída, e é definido utilizando a palavra chave `merge` seguido do nome do nó. O nó *decision* representa uma decisão, podendo ter vários fluxos de saída e apenas um fluxo de entrada, e é definido utilizando a palavra chave `decide` seguido do nome do nó, pode-se utilizar a palavra chave `if` seguido de uma expressão para definir uma guarda para as arestas de saída. Já as arestas de saída são definidas utilizando a palavra chave `then` ou `else` seguido do nome do nó de destino. O nó *fork* divide um único fluxo em dois ou mais fluxos simultâneos, e é definido utilizando a palavra chave `fork` seguido do nome do nó. O nó *join* combina dois ou mais fluxos simultâneos e os reintroduz em um único fluxo, e é definido utilizando a palavra chave `join` seguido do nome do nó.

Control flows mostram o fluxo de controle da atividade, uma aresta de entrada inicia a execução de um nó, e uma vez concluída a execução, o fluxo continua com as arestas de saída do nó, sendo definidas utilizando a palavra chave `succession` seguida pelo nome do nó de origem, e adicionando a palavra chave `then` seguida pelo nome do nó de destino. *Object flows*, transportam dados do nó de origem para o nó de destino, e são definidos de duas formas, a primeira sendo utilizada para transferência de dados entre nós utilizando a palavra chave `flow` seguido da referência da origem do dado, e

Nó	SysML 2.0	Nó	SysML 2.0
Initial	`first start;`	**Action Usage**	`action actionUsage;`
Final	`action done: Action :>> endShot;`	**Action Definition**	`action def ActionDefinition {}`
Merge	`merge mergeNode;`	**Send Signal**	`action sendSignal send signal1::x to acceptEvent;`
Decision	`decide decisionNode; then act2; then done;`	**Accept Event**	`action acceptEvent accept y : Integer;`
Fork	`fork forkNode;`	**Pins**	`action action1(in x : Integer, out y : Integer) { language "Alf" /* y = x; */ }`
Join	`join joinNode;`	**Parameters**	`action def ActionDefinition(in x : Integer) {}`
Control Flow	`succession act1 then decisionNode;`	**Object Flow**	`bind nonDeterminism2::y = join1;` `flow action0::y to action1::x;`

Tabela 2 – Nós da SysML representados na linguagem textual SysML 2.0

adicionando a palavra chave `to` seguida da referência do destino do dado, a segunda forma é utilizada para tranferência de dados entre um *parameter* e um nó interno da ação utilizando a palavra chave `bind` seguido por uma expressão que vincula a origem do dado com o destino do dado, como por exemplo `diagram::x = act1::y`. E por fim a definição de uma ação é simplesmente a definição de um nó *action* com nós internos.

2.2 CSP

Uma álgebra de processo como CSP (HOARE, 1985) pode ser usada para descrever sistemas compostos de componentes interativos, que são processos autônomos independentes com interfaces usadas para interagir com o ambiente. Tal formalismo fornece uma maneira de especificar explicitamente e raciocinar sobre as interações entre diferentes componentes. Além disso, fenômenos que são exclusivos do mundo concorrente, que surgem da combinação de componentes e não de componentes individuais, como *deadlock* e *livelock*, podem ser mais facilmente compreendidos

e controlados usando tal formalismo. O suporte de ferramentas é outro motivo para o sucesso de CSP na indústria e, consequentemente, para nossa escolha de utilizá-lo como notação formal. Por exemplo, FDR (GIBSON-ROBINSON et al., 2014) fornece uma análise automática do refinamento do modelo e de propriedades como *deadlock*, divergência e não-determinismo.

No CSP, um processo é a unidade básica para descrever o comportamento. É definido em termos de eventos e outros processos. A função $\alpha(P)$ retorna o alfabeto de um processo P, ou seja, os eventos que este processo pode comunicar. O processo $SKIP$ não faz nada e indica que um processo foi encerrado com sucesso. Um processo que é o *deadlock* canônico é $STOP$. Um processo $a \rightarrow P$ oferece o evento a ao ambiente e então se comporta como P. Os eventos são comunicados por canais. Os canais podem comunicar dados $a.x$ (canal a comunica x), dados de saída $a!x$ (canal a gera valor x) e dados de entrada $a?x$ (canal a recebe uma entrada em x). A composição sequencial $P1; P2$ se comporta como o processo $P1$ e, desde que termine com sucesso, $P2$ assume. A notação CSP não possui operador explícito para recursão, mas permite usar o nome do processo em sua definição. Por exemplo, $P = a \rightarrow P$ executa a e depois se comporta como P.

A escolha externa $P1 \Box P2$ oferece inicialmente eventos de ambos os processos. A ocorrência do primeiro evento resolve a escolha em favor do processo que o realiza. Já na escolha interna $P1 \sqcap P2$ o ambiente não tem controle. A composição paralela $P1 \underset{cs}{\|} P2$ sincroniza $P1$ e $P2$ nos canais do conjunto cs, onde eventos que não estão em cs ocorrem independentemente. Processos compostos em intercalar $P1 ||| P2$ executam sem sincronização. O operador de ocultação de eventos $P \setminus cs$ internaliza os eventos que estão no conjunto de canais cs, que não se tornam mais visíveis para o ambiente.

Neste trabalho, nos concentramos na detecção de *deadlock* e não-determinismo, que são propriedades verificáveis em FDR. O FDR tem como especificações de entrada o CSP $_M$, uma versão legível por máquina de CSP e as traduz para *Labeled Transition System* (LTS), uma notação baseada em máquina de estado. A abordagem adotada pelo FDR para verificar propriedades como *deadlock* e não-determinismo é baseada na análise global, onde todo o modelo é expandido e verificado exaustivamente. No entanto, o FDR possui diversos mecanismos de otimização para melhorar o desempenho de seu mecanismo de raciocínio (ROSCOE; HOARE; BIRD, 1997). Em caso do FDR detectar um *deadlock* ou não-determinismo, é retornado uma sequência de eventos que mostra o ponto onde o problema ocorreu. Por exemplo, no não-determinismo encontrado na Figura 8 é retornado pelo FDR a seguinte sequência de eventos, [`startActivity_nonDeterminism1.1` $\rightarrow$ `update_nonDeterminism1.1.1.1` $\rightarrow$ `ce_nonDeterminism1.1.1` $\rightarrow$ `event_act1_nonDeterminism1.1` $\rightarrow$ `ce_nonDetermi`

nism1.1.2]. Na figura 10, é possível identificar à quais nós e arestas pertecem cada um desses eventos, o evento startActivity_nonDeterminism1.1 é um evento de inicialização da ação, o evento update_nonDeterminism1.1.1.1 pertence ao nó *initial*, o evento ce_nonDeterminism1.1.1 pertence ao nó *initial*, nó *action* act1 e a aresta entre o nó *initial* e nó *action* act1, o evento *event_act1_nonDeterminism1.1* pertence ao nó *action* act1 e o evento ce_nonDeterminism1.1.2 pertence ao nó *action* act1, nó *decision* e a aresta entre o nó *action* act1 e o nó *decision*.

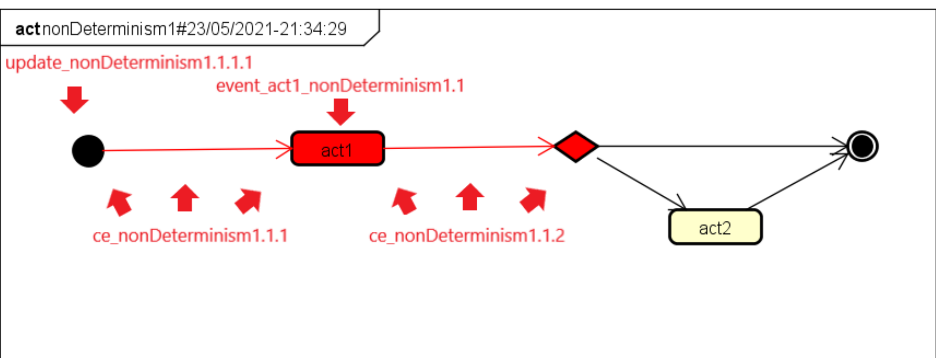


Figura 10 – Mapeamento da sequência de eventos retornada pelo FDR

3 Verificador de ações SysML 2.0

Nesta Seção apresentamos todo o processo do verificador de ações SysML 2.0 que foi desenvolvido. Na Subseção 3.1 damos uma visão geral do processo de verificação de ações SysML 2.0, na Subseção 3.2 mostramos como é feita a tradução da linguagem SysML 2.0 para a linguagem formal CSP de cada nó e aresta apresentado na Seção 2, e por fim mostramos como é feita a verificação e rastreabilidade de *deadlocks* e não-determinismos.

3.1 Visão Geral do Processo de Verificação de Ações SysML 2.0

Primeiramente vamos entender melhor como o verificador funciona. Na Figura 11 é possível observar o seu processo de uma forma simplificada.

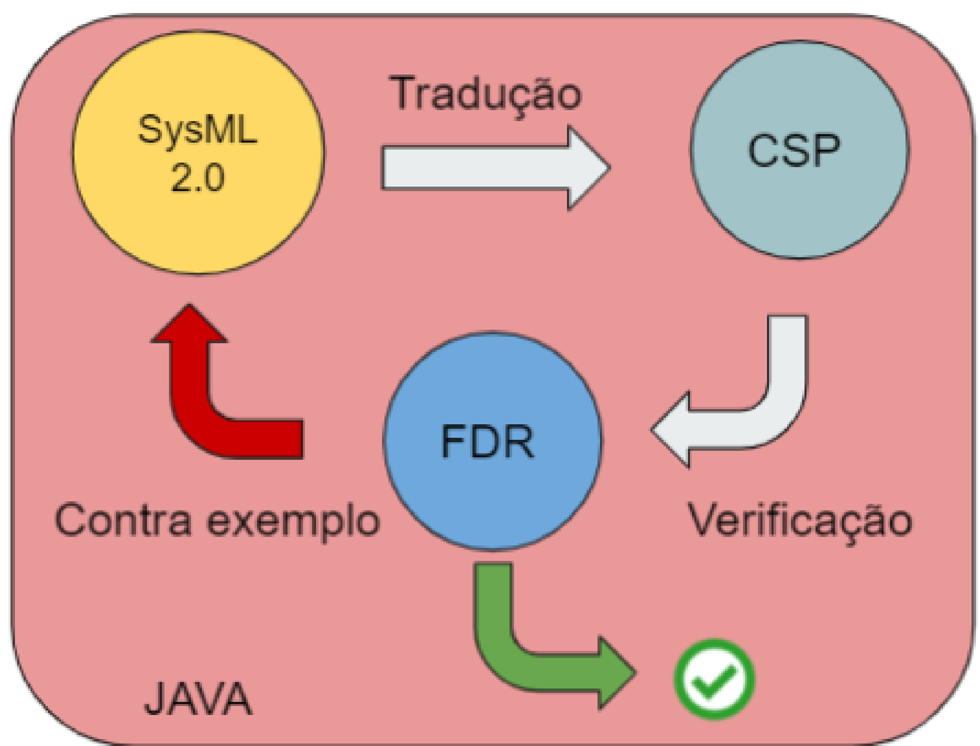


Figura 11 – Visão geral do processo do verificador

O processo de verificação de *deadlock* ou não-determinismo, inicia com a especificação da ação na linguagem SysML 2.0, passando essa especificação para o verificador é realizado o primeiro passo, que é a tradução da ação para a linguagem CSP. O módulo *Controller* realiza todo o controle do processo de tradução, primeiramente a especificação passada como entrada é convertida em objetos utilizando os adaptadores do submódulo *Adapters/SysML 2.0* (falaremos mais dos adaptadores na

Subseção 3.2.1), em seguida esses objetos são passados para o módulo *Parser CSP* onde é realizada a tradução dos objetos para uma especificação na linguagem CSP. Com essa especificação CSP, o módulo *Controller* utiliza o módulo *FDR Bridge* para fazer uma chamada ao FDR e realizar a verificação do *deadlock* ou não-determinismo, após essa verificação o FDR pode retornar uma sequência de eventos (*trace*) caso tenha encontrado algum problema na ação verificada. Caso nenhum erro tenha sido encontrado pelo FDR, uma mensagem indicando que a ação é livre de *deadlock* ou que a ação é determinística é apresentada ao usuário. Em caso de algum erro ser encontrado pelo FDR, o módulo *Controller* manda a sequência de eventos para o módulo *Traceability*, onde é criado um contraexemplo da ação SysML 2.0 e retornado para o módulo *Controller*, e por fim este contraexemplo é retornado para o usuário com comentários indicando quais nós e arestas foram percorridos pelo FDR na verificação. Entraremos em mais detalhes sobre a verificação e rastreabilidade do verificador na Seção 3.3.

Agora que entendemos melhor como o verificador funciona, vamos entender melhor como ele é organizado. O verificador foi construído tendo como dependência o checador de modelos FDR, na Figura 12 podemos observar a arquitetura do verificador e como é dividido seus módulos.

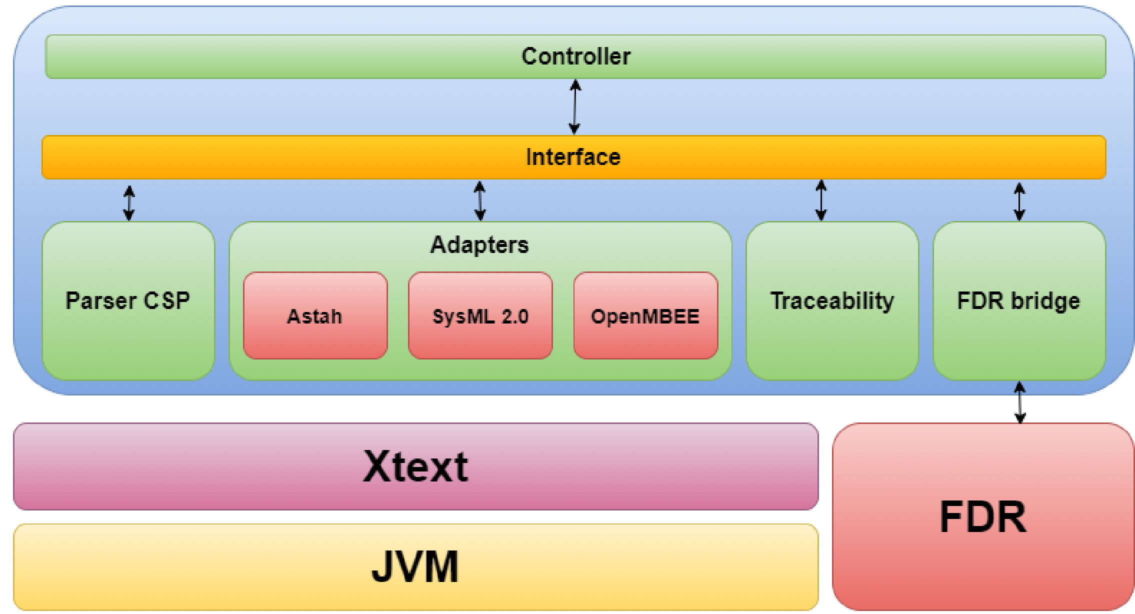


Figura 12 – Arquitetura do verificador

A *Java Virtual Machine* (JVM) é uma máquina virtual que permite que um computador execute programas na linguagem de programação Java (GOSLING et al., 2000), que neste caso é utilizado para executar os códigos do verificador e do Xtext (que será explicado mais a frente) que são escritos nesta linguagem. O FDR é uma ferramenta para analisar e animar especificações CSP, e nos permite verificar *deadlocks*, *livelock*, não-determinismo, refinamentos e dentre outros.

O verificador desenvolvido é dividido em módulos, que são: *Controller*, *Interface*, *Parser CSP*, *FDR Bridge*, *Traceability* e *Adapters*. O módulo *Controller* é responsável por receber informações (comandos e ações SysML 2.0), gerenciar todo o funcionamento do verificador e retornar uma resposta (mensagens e/ou contraexemplo). O módulo *interface* é responsável por fornecer uma API para ações SysML 2.0 que possa ser utilizada por diversas ferramentas e que permita comunicação com o módulo *controller*. O módulo *FDR Bridge* é responsável pela comunicação com o FDR, utilizando a técnica *Java Reflection*, que nos permite descobrir métodos e atributos de uma classe em tempo de execução, podendo carregar a API do FDR de forma dinâmica. O módulo *Traceability* é responsável por receber uma lista de eventos (*trace*) do módulo *Controller*, e criar um contraexemplo de uma ação SysML 2.0 que mostra o caminho percorrido pelo FDR até o *deadlock* ou não-determinismo e retorna a ação criada para o módulo *Controller*. O módulo *Adapters* é dividido em 3 submódulos, sendo eles SysML 2.0, Astah e OpenMBEE. Neste trabalho iremos focar no submódulo *Adapters/SysML 2.0*. Caso queira mais detalhes sobre como é implementado os outros módulos e submódulos, você pode encontrar em (LIMA; TAVARES; NOGUEIRA, 2020). O submódulo *Adapters/SysML 2.0* é responsável por traduzir a ação na linguagem textual SysML 2.0 para adaptadores java (objetos) que estejam de acordo com a interface da linguagem SysML 2.0. E por fim, o módulo *Parser CSP* é responsável por receber um objeto de uma ação SysML 2.0 do módulo *Controller* que respeita a interface para ações SysML 2.0, traduzir para a linguagem CSP de acordo com a semântica que iremos apresentar na Seção 3.2, e devolver um arquivo CSP ao módulo *Controller*.

3.2 Tradução de Ações SysML 2.0

Nesta Seção iremos descrever como é feita a tradução da especificação SysML 2.0 para a especificação CSP. A estrutura da especificação CSP é apresentada na Figura 14, esta figura representa a organização de uma única ação SysML 2.0 na especificação CSP. Os retângulos azuis com bordas arredondadas nomeados de `nodes_x` são ilustrações de processos CSP criados pela abordagem, e os || são ilustrações de paralelismos com um canal de sincronização $\alpha(n_x)$. O processo `MainProcess` é especificado como o paralelismo entre o processo `Nodes` e o processo `TokenManager`, e também é responsável por iniciar a execução do processo ao receber um sinal no canal `startActivity` e ao encerrar o processo envia um sinal no canal `endActivity`, tanto `startActivity` quanto o `endActivity` podem comunicar dados para outras ações SysML 2.0 caso a ação possua *parameters*. O processo `Nodes`, por sua vez, é especificado pelo paralelismo de vários processos internos `node_x` que são sincronizados (||) nos canais $\alpha(n_x)$, além de também sincronizarem eventos e sinais de nós *send signal* e *accept event* com outras ações SysML 2.0. Cada processo interno é dividido

em dois processos, um chamado de `Node(id)` que detalha o comportamento do *node* e que também é recursivo, e outro chamado de `Node_t(id)` que invoca o primeiro e que é interrompido (/\) pelo processo `END_DIAGRAM` que é o processo de conclusão da ação, como pode ser observado na Figura 13. Iremos omitir o segundo processo nas explicações dos nós porque o formato é o mesmo para todos.

```
Node(id) = ...; Node(id)
Node_t(id) = Node(id) /\ END_DIAGRAM(id)
```

Figura 13 – Definição dos processos de um *node*

O processo `TokenManager` que está sincronizado com o processo `Nodes` nos canais `update`, `clear` e `endDiagram`, é responsável por gerênciar os fluxos ativos na ação. Quando o número de fluxo ativos chega a 0 ou o canal `clear` é comunidado, o processo `TokenManager` envia um sinal no canal `endDiagram` que está sincronizado com o processo `END_DIAGRAM` que encerra a execução de todos os *nodes*. A semântica CSP que é utilizada na especificação da ação é definida como diagrama de atividades em (LIMA; TAVARES; NOGUEIRA, 2020).

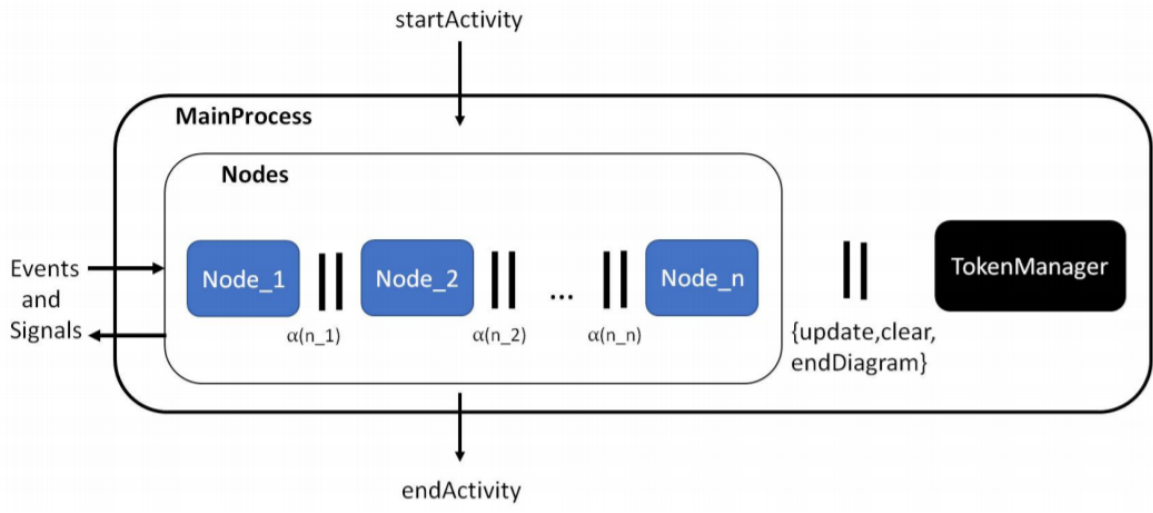


Figura 14 – Estrutura da especificação CSP apresentada em (LIMA; TAVARES; NOGUEIRA, 2020)

3.2.1 Xtext

O Xtext é um *framework* para desenvolvimento de linguagens de programação e linguagens de domínio específico. Com o Xtext é possível definir sua linguagem usando uma linguagem gramatical. Como resultado, você obtém uma infraestrutura completa, incluindo tradutor, vinculadores, checadores de tipos, compiladores, entre

outros (BETTINI, 2016). A linguagem SysML 2.0 é definida utilizando o Xtext, e no repositório (OMG, 2021) é possível encontrar a gramática da SysML 2.0 no formato Xtext, onde também está incluído a gramática da KerML que é a base da SysML 2.0. Neste mesmo repositório também é possível encontrar alguns exemplos da notação SysML 2.0. Com isso a estratégia de tradução consistiu em usar a gramática da SysML 2.0, utilizando o Xtext para gerar o *parser* textual da SysML 2.0 para objetos Xtext.

Após utilizar o Xtext para gerar a linguagem SysML 2.0, são gerados os objetos Xtext que são utilizados para construir os adaptadores de elementos SysML 2.0 para ações que respeitam a *interface* do nosso *framework*. Os objetos Xtext são utilizados para definir primeiramente o adaptador *Activity Diagram* que representa a ação, em seguida é definido o adaptador *Activity* que representa o processo interno da ação. Dentro do adaptador *Activity* são encapsulados todos os adaptadores dos outros *nodes* e *flows*. Os adaptadores *Control Flow* e *Object Flow* herdam do adaptador *Flow* que define a base para os dois tipos de *flows*. O adaptador *Activity Node* é herdado diretamente por todos os *control nodes*, *actions nodes* e *object nodes*. Os adaptadores *Input Pin* e *Output Pin* herdam do adaptador *Pin* que define a base para os *pins*, já o adaptador *Pin* herda do adaptador *Object Node*, que por sua vez herda do adaptador *Activity Node*. E por fim o adaptador *Activity Parameter Node* que também herda do adaptador *Object Node*. Todos os adaptadores e suas relações podem ser encontrados na Figura 15. Nas próximas seções vamos explicar como os elementos de ações SysML 2.0 são traduzidos para CSP.

3.2.2 Nós de Objetos

Os nós de objetos traduzidos pelo verificador são os *pins* e *parameters*, as traduções podem ser encontradas na Tabela 3, onde na primeira linha existe um pino de entrada `x` do tipo inteiro, e na segunda linha temos um parâmetro de entrada `x` do tipo inteiro. Os *pins* e *parameters* são traduzidos em processos recursivos que desempenham o papel de uma memória onde através do operador de escolha externa (□) é possível sincronizar com processos de outros nós nos canais de `get_x_actionNode.id?c!x` para recuperar o valor atual do dado guardado e `set_x_actionNode.id?c!x` para atualizar o valor do dado para um novo valor, onde `get_x_actionNode` e `set_x_actionNode` são os nomes dos canais, `id` é o identificador único da ação, `c` é o identificador único do canal e `x` é o nome do dado que vai ser enviado ou recebido dependendo se o canal utiliza um símbolo de `!` (para enviar dados) ou se utiliza um símbolo de `?` (para receber dados). Os processos de memória criados pelos *pins* são sincronizados com o processo do *action* ou *call behavior* correspondente nos canais `set_x_actionNode` e `get_x_actionNode`.

Já os *parameters* criam um processo correspondente ao parâmetro de entrada

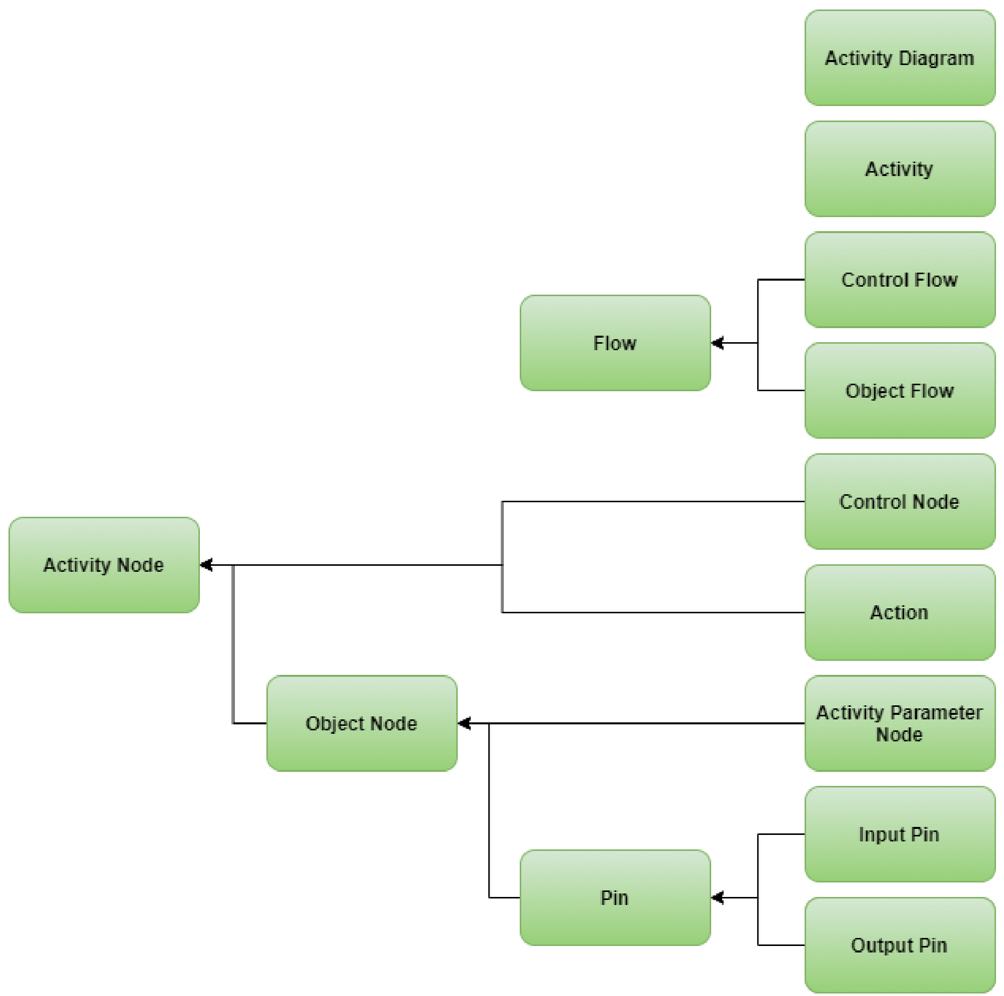


Figura 15 – Adaptadores SysML 2.0

ou saída chamado de `parameter_x`, este processo é responsável por buscar o dado e envia-lo para suas arestas de destino. Primeiramente a ação recebe o dado no canal `startActivity` e em seguida guarda este dado enviando para o canal `set_x_actionNode`, após isso o processo `parameter_x` inicia comunicando o número de fluxos ativos que será gerado pelo processo enviando no canal `update` o número de arestas de saída, em seguida o dado é buscado sincronizando no canal `get_x_actionNode`, e por fim os canais `oe` representam os eventos de arestas de objetos de saída que será detalhado mais a frente. Ambos o canais `set_x_actionNode` e `get_x_actionNode` estão sincronizados com o processo de memória `Mem_actionNode_x`.

3.2.3 Arestas

As arestas traduzidas pelo verificador são os *control flows* e *object flows*, as traduções podem ser encontradas na Tabela 4. Os *control flows* são traduzidos em canais na forma `ce.id.x` que são sincronizados pelos processos do nó de origem *sourceNode* e nó de destino *targetNode*, onde `ce` é o nome do canal, `id` é o identificador único da ação e `x` é o identificador único do canal.

<table>
<tr><td>

Pins

```
action actionNode(in x : Integer, out y : Integer) {
    language "Alf"
    /* y = x; */
}
```

</td><td>

$Mem_actionNode_x(id,x)=$
$get_x_actionNode.id?c!x\rightarrow$
$Mem_actionNode_x(id,x)$
$\square$
$set_x_actionNode.id?c?x\rightarrow$
$Mem_actionNode_x(id,x)$

</td></tr>
<tr><td>

Parameters

```
action nameDiagram(in x : Integer, out y : Integer) {}
```

</td><td>

$parameter_x(id)=$
$update_nameDiagram.id.u!(\#Outputs-\#Inputs)\rightarrow$
$get_x_nameDiagram.id.t?x\rightarrow$
$oe.id!x\rightarrow SKIP));$

$Mem_actionNode_x(id,x)=$
$get_x_actionNode.id?c!x\rightarrow$
$Mem_actionNode_x(id,x)$
$\square$
$set_x_actionNode.id?c?x\rightarrow$
$Mem_actionNode_x(id,x)$

</td></tr>
</table>

Tabela 3 – Traduções dos nós de objetos

Os *object flows* são divididos em dois tipos, o tipo 1 é utilizado para transportar dados de um *pin* para outro, ou para um *control node*, como *fork*, *join*, *merge* e *decision*. Já o tipo 2 é enviado de um *parameter* da ação para seu destino seja ele um *pin* ou *control node*. Ambos os tipos dos *object flows* são traduzidos em canais na forma `oe.id!(x)` ou `oe.id?(x)`, que é enviado da sua origem para seu destino, onde `oe` é o nome do canal, `id` é o identificador único da ação e `x` é o nome do dado que vai ser enviado ou recebido dependendo se o canal utiliza um símbolo de `!` (para enviar dados) ou se utiliza um símbolo de `?` (para receber dados). Antes de enviar o dado, ele é recuperado utilizando o canal `get_y_sourceNode.id.t?y` que está sincronizado com o processo de memória detalhado na Subseção 3.2.2, após isso o dado é enviado para seu nó de destino. Após o dado ser recebido pelo nó de destino, ele é enviado para o processo de memória pelo canal `set_x_targetNode.id.t!x`. Caso o nó de origem realize alguma operação em cima do dado de saída, como por exemplo `x + 1`, então essa expressão precisa ser validada, em `(x + 1) >= minValue and (x + 1) <= maxValue`, onde `minValue` e `maxValue` são referentes aos valores mínimo e máximo do tipo de dado especificado, e após a verificação o dado só é enviado se estiver de acordo com os limites do tipo de dado especificado, esses limites são importantes para permitir que a análise do FDR conclua, caso contrário um erro semântico seria apresentado na especificação CSP pois estariamos utilizando um valor não previsto para um determinado tipo.

Control Flow `succession sourceNode then targetNode;`	$sourceNode(id)=$ $(entradas\ do\ no);\ (comportamento\ do\ no)\rightarrow$ $(ce.id.x\rightarrow SKIP);$ $sourceNode(id)$ $targetNode(id)=$ $(ce.id.x\rightarrow SKIP);$ $(comportamento\ do\ no)\rightarrow(saidas\ do\ no);$ $targetNode(id)$

Object Flow
`flow sourceNode::y to targetNode::x;`
ou
`bind nameDiagram::x = targetNode::x;`
	$parameter_x(id)=$
	$update_nameDiagram.id.t!(1-0)\rightarrow$
	$get_x_nameDiagram.id.t?x\rightarrow$
	$(oe.id!x\rightarrow SKIP)$
	
	$targetNode(id)=$
	$(oe.id?x\rightarrow set_x_targetNode.id.t!x\rightarrow SKIP);$
	$(comportamento\ do\ no)\rightarrow(saidas\ do\ no);$
	$targetNode(id)$

Tabela 4 – Traduções das arestas

3.2.4 Nós de Ações

Os nós de ações traduzidos pelo verificador são os nós *actions*, *call behaviors*, *send signals* e *accept events*, as traduções podem ser encontradas na Tabela 5. Os nós *actions* são traduzidos em processos onde todos os eventos das arestas de entrada estão compostos em paralelo entrelaçado (|||), só após todos os eventos de entrada serem recebidos, podemos dar continuidade ao comportamento do nó. Após isso o processo atualiza o número de fluxos ativos enviando para o canal `update_diagramName.id.x!(#Outputs-#Inputs)` do processo *TokenManager*, onde `update_diagramName` é o nome do canal, `id` é o identificador único da ação, `x` é o identificador único do canal, `#Outputs` é o número de arestas de saída do nó e `#Inputs` é o número de arestas de entrada do nó. Em seguida, o nó envia o evento `event_actionNode_diagramName.id`, que simboliza a ação interna do nó, e por fim, os eventos das arestas de saída do nó que também estão compostos em um paralelismo entrelaçado (|||) são enviados.

Os nós *call behaviors* também são traduzidos em processos onde todos os eventos das arestas de entrada estão compostos em paralelo entrelaçado (|||), após isso o processo passa a se comportar como o processo `CB1(id)`, onde `CB1` é o nome do processo principal da ação invocada no *call behavior* e `id` é o identificador único desta ação. Após esta instância da ação `CB1` finalizar, o processo retorna o fluxo para o *call behavior* onde os eventos das arestas de saída do nó que também estão compostos

<table>
<tr><td>

Action

```
action actionNode;
succession node_x then actionNode;
.
.
.
succession node_z then actionNode;
succession actionNode then node_m;
.
.
.
succession actionNode then node_n;
```

</td><td>

$actionNode(id)=$
$(ce.id.x \to SKIP|||...|||ce.id.z \to SKIP);$
$update_diagramName.id.x!(\#Outputs-\#Inputs) \to$
$event_actionNode_diagramName.id \to$
$(ce.id.m \to SKIP|||...|||ce.id.n \to SKIP);$
$actionNode(id)$

</td></tr>
<tr><td>

Call Behavior

```
action def CB1 {}
action cb1 : CB1;
succession node_x then cb1;
.
.
.
succession node_z then cb1;
succession cb1 then node_m;
.
.
.
succession cb1 then node_n;
```

</td><td>

$callBehaviorNode(id)=$
$(ce.id.x \to SKIP|||...|||ce.id.z \to SKIP);$
$CB1(id);$
$(ce.id.m \to SKIP|||...|||ce.id.n \to SKIP);$
$callBehaviorNode(id)$

</td></tr>
<tr><td>

Send Signal

```
action actionNode (out x : Integer);
action sendSignalNode
  send actionNode::x to acceptEventNode;
```

</td><td>

$sendSignalNode(id)=$
$(oe.id?x \to set_sendSignalNode.id.t!x \to SKIP);$
$update_diagramName.id.x!(\#Outputs-\#Inputs) \to$
$get_sendSignalNode.id.t?x \to$
$signal_sendSignalNode.id.u!x \to$
$sendSignalNode(id)$

</td></tr>
<tr><td>

Accept Event

```
action acceptEventNode accept y : Integer;
flow acceptEvent::y to node_m::x;
.
.
.
flow acceptEvent::y to node_n::x;
```

</td><td>

$acceptEventNode(id)=$
$(accept_sendSignalNode.id.t?event_signal \to$
$(oe.id!event_signal \to SKIP|||...|||$
$oe.id!event_signal \to SKIP);$
$acceptEventNode(id)$

</td></tr>
</table>

Tabela 5 – Traduções dos nós de ações

em um paralelismo entrelaçado (|||) são enviados.

Os nós *send signals* são traduzidos em processos que sincronizam em um canal `oe` que transportam dados, em seguida este dado é guardado em um processo de memória utilizando o canal `set_sendSignalNode`. Após isso o processo atualiza o número de fluxos ativos enviando para o canal `update_diagramName` do processo *TokenManager*, em seguida é buscado o valor guardado no processo de memória utilizando o canal `get_sendSignalNode`, e por fim o dado é enviado para uma *pool* (que funciona como um *buffer* de comunicação) utilizando o canal `signal_sendSignalNode` do processo `pool_sendSignalNode`. O processo `pool_sendSignalNode` (*Listing* 3.1) está sincronizado com os processos do *send signal* e *accep event* (que será detalhado a seguir), este processo pode realizar duas funções, a primeira é receber um dado do nó *send signal* no canal `signal_sendSignalNode` e guardá-lo em uma lista de dados chamada de `l`, o dado é guardado na cabeça da lista, porém a lista tem uma limitação de armazenamento de no máximo cinco dados. A segunda função deste processo é

enviar os dados que estão na lista para os nós *accept events* (quando isso for possível), neste caso se a lista possuir algum dado pendente e algum nó *accept event* está tentando sincronizar no canal `accept_sendSignalNode`, o dado da cabeça da lista é enviado (`head(l)`) e a lista permanece com o restante dos dados armazenados (`tail(l)`) para futuras sincronizações neste canal.

Os nós *accept events* são traduzidos em processos que primeiramente sincronizam no canal `accept_sendSignalNode` do processo `pool_sendSignalNode` e recebem um dado de entrada, após receber o dado de entrada, o dado é enviado para os eventos das arestas de saída do nó que estão compostos em um paralelismo entrelaçado (|||).

$pool_sendSignalNode(l) =$
$\quad (signal_sendSignalNode?id?event_signal \rightarrow$
$\quad if \quad length(l) < 5$
$\quad pool_sendSignalNode(l \wedge < event_signal >)$
$\quad else \quad pool_sendSignalNode(l))$
$\quad \Box$
$\quad (length(l) > 0$
$\quad \& \; accept_sendSignalNode.id.t!head(l) \rightarrow$
$\quad pool_sendSignalNode(tail(l)))$

Listing 3.1 – Definição do processo pool_sendSignalNode

3.2.5 Nós de Controles

Os nós de controles traduzidos pelo verificador são os *initials*, *finals*, *decisions*, *merges*, *forks* e *joins*, as traduções podem ser encontradas na Tabela 6. A SysML 1.0 possui também o nó *final flow*, porém a SysML 2.0 não possui uma representação para este tipo de nó.

Os *initials* são traduzidos em processos que iniciam os fluxos das ações, sendo um dos primeiros processos que são iniciados na ação, o processo começa atualizando o número de fluxos ativos enviando o valor para o canal `update_diagramName.id.x!(# Outputs-#Inputs)`, onde `update_diagramName` é o nome do canal, `id` é o identificador único da ação, `x` é o identificador único do canal, `#Outputs` é o número de arestas de saída e `#Inputs` é o número de arestas de entrada. Após esse passo, todas suas arestas de saída são enviadas em paralelo (|||). Os *finals* são traduzidos em processos que encerram os fluxos das ações, onde a partir de uma escolha externa (□) o primeiro canal sincronizado que for enviado um sinal irá encerrar o fluxo da ação, o fluxo é encer-

rado enviando um sinal no canal `clear_diagramName.id.x`, onde `clear_diagramName` é o nome do canal, `id` é o identificador único da ação e `x` é o identificador único do canal. O canal `clear_diagramName` está sincronizado com o processo *TokenManager*, que ao receber um sinal, envia outro sinal no canal `endDiagram_diagramName.id` que está sincronizado com outro processo chamado de *END_DIAGRAM_digramName*, que é responsável por encerrar todos os nós que estão ativos na ação, após isso a ação encerra seu processo enviando um sinal para o canal `endActivity_digramName.id`.

SysML	Tradução

Initial
`first initialNode;`
`succession initialNode then node_m;`
`.`
`.`
`.`
`succession initialNode then node_n;`
	$initialNode(id) =$
	$update_diagramName.id.x!(\#Outputs - \#Inputs) \rightarrow$
	$(ce.id.m \rightarrow SKIP \mid\mid\mid ... \mid\mid\mid ce.id.n \rightarrow SKIP);$

Final
`succession node_x then done;`
`.`
`.`
`.`
`succession node_z then done;`
`action done: Action :>> endShot;`
	$done(id) =$
	$(ce.id.x \rightarrow SKIP \Box ... \Box ce.id.z \rightarrow SKIP);$
	$clear_diagramName.id.x \rightarrow SKIP$

Decision
`flow sourceNode::x to decisionNode;`
`decide decisionNode;`
`    if guard1 then targetNode1;`
`    if guard2 then targetnode2;`
	$decisionNode(id) =$
	$oe.id?x \rightarrow (guard1\ \&\ (dc \rightarrow oe.id!x \rightarrow SKIP)$
	$\Box$
	$guard2\ \&\ (dc \rightarrow oe.id!x \rightarrow SKIP));$
	$decisionNode(id)$

Merge
`succession node_x then mergeNode;`
`.`
`.`
`.`
`succession node_z then mergeNode;`
`merge mergeNode;`
`succession mergeNode then targetNode;`
	$mergeNode(id) =$
	$(ce.id.u \rightarrow SKIP \Box ... \Box ce.id.t \rightarrow SKIP);$
	$ce.id.x \rightarrow$
	$mergeNode(id)$

Fork
`succession sourceNode then forkNode;`
`fork forkNode;`
`succession forkNode then node_m;`
`.`
`.`
`.`
`succession forkNode then node_n;`
	$forkNode(id) =$
	$ce.id.x \rightarrow$
	$update_diagramName.id.x!(\#Outputs - \#Inputs) \rightarrow$
	$(ce.id.m \rightarrow SKIP \mid\mid\mid ... \mid\mid\mid ce.id.n \rightarrow SKIP);$
	$forkNode(id)$

Join
`succession node_x then joinNode;`
`.`
`.`
`.`
`succession node_z then joinNode;`
`join joinNode;`
`succession joinNode then targetNode;`
	$JoinNode(id) =$
	$(ce.id.x \rightarrow SKIP \mid\mid\mid ... \mid\mid\mid ce.id.z \rightarrow SKIP);$
	$update_diagramName.id.x!(\#Outputs - \#Inputs) \rightarrow$
	$(ce.id.x \rightarrow SKIP);$
	$JoinNode(id)$

Tabela 6 – Traduções dos nós de controles

Já os *decisions* são traduzidos em processos que sincronizam em uma aresta de entrada, e em seguida, o processo avalia as guardas das suas arestas de saída para decidir qual aresta enviará um sinal, lembrando que apenas uma aresta de saída pode receber o sinal. Após avaliar as guardas e decidir a aresta de destino que receberá o sinal, o processo envia um sinal no canal `dc` (que é comum entre todas as arestas

de saída), este canal é escondido no processo principal do *decision* para simular um não-determinismo no caso em que o *decision* não tenha guarda ou quando mais de uma guarda é válida. E por fim, envia o sinal ou dado para a aresta de destino selecionada. Os *merges* são traduzidos em processos que sincronizam com várias arestas de entradas utilizando escolha externa ($\Box$), e caso alguma dessas arestas envie um sinal, o processo envia um sinal para uma única aresta de destino definida. Os *forks* são traduzidos em processos que sincronizam em uma única aresta de entrada, em seguida é atualizado o número de fluxos ativos na ação enviando o valor pelo canal `update_diagramName`, e por fim, envia em paralelo um sinal para todas as arestas de saídas. Os *joins* são traduzidos em processos que esperam que todas as suas arestas de entradas sejam sincronizadas ($|||$), em seguida é atualizado o número de fluxos ativos na ação enviando um o valor pelo canal `update_diagramName`, e por fim, é enviado um sinal para uma única aresta de destino.

Utilizando estas semânticas detalhadas acima, conseguimos gerar um arquivo de texto com a especificação CSP resultante da aplicação dos conceitos descritos anteriormente. A próxima seção mostrará como este arquivo é utilizado como entrada para o processo de verificação formal e em seguida realizar sua rastreabilidade.

3.3 Verificação e Rastreabilidade

O verificador realiza verificações das propriedades *deadlock* e não-determinis mo, estas verificações são realizadas pelo FDR ao inserir na especificação CSP a asserção `assert MAIN :[deadlock free]` para verificação de *deadlock* e `assert MAIN :[deterministic]` para verificação de não-determinismo, onde `MAIN` é o nome do processo principal que especifica a primeira ação da especificação. Para que o FDR consiga utilizar tipos de dados, é necessário defini-los com seus limites na especificação SysML 2.0. Esta definição é inserida dentro do escopo de uma ação na forma de `doc /* expressão */`, onde a expressão, por exemplo, pode ser um tipo de dado `Integer` limitado de 0 à 10, e que seria definida da seguinte forma `doc /* Integer={0..10} */`.

O FDR verifica com base na análise global, onde todo o modelo é expandido e verificado exaustivamente. Se algum *deadlock* ou não-determinismo for encontrado na verificação do FDR, é retornado uma sequência de eventos que foi percorrido até chegar ao problema. O verificador guarda internamente a lista de nós e arestas da ação verificada com seus respectivos eventos CSP, e com isso, esta sequência de eventos é utilizada pelo verificador para identificar quais foram os nós e arestas percorridos pelo FDR. Após identificar esses nós, o verificador percorre a especificação SysML 2.0 da ação linha por linha para identificar a linha que representa o uso do nó ou

aresta identificado, e por fim, é adicionado um comentário `"// [deadlock trace]"` ou `"// [non-determinism trace]"` nesta linha. Após isso, é gerado e retornado para o usuário um novo arquivo com este contraexemplo.

Podemos tomar como exemplo da verificação e rastreabilidade do verificador o diagrama `actionDeterminism1` da Figura 5, que tem sua especificação em SysML 2.0 apresentada na Figura 6, e após usar o verificador para checar a propriedade de não-determinismo, o FDR retorna para o verificador a sequência de eventos [`startActivity_nonDeterminism1.1` → `update_nonDeterminism1.1.1.1` → `ce_nonDeterminism1.1.1` → `event_act1_nonDeterminism1.1` → `ce_nonDeterminism1.1.2`], e com essa sequência o verificador consegue identificar quais nós e arestas estes eventos pertencem, como apresentado na Figura 10. Após identificar os nós e arestas que foram percorridos, o verificador percorre a especificação SysML 2.0 da ação para marcar os nós e arestas com o comentário `"// [non-determinism trace]"` que ajuda o usuário a identificar onde está o problema, e que também poderá ser utilizado por ferramentas de modelagem (que utilizem a linguagem SysML 2.0) para ilustrar o problema de forma visual.

4 Estudos de Casos

Esta seção apresenta a aplicação do verificador para analisar *deadlock* e não-determinismo em 4 estudos de casos. O primeiro estudo de caso é uma representação de um carregador de bateria baseada no exemplo *Fork Join Example* que pode ser encontrado no repositório (OMG, 2021). A especificação SysML 2.0 do primeiro estudo de caso pode ser encontrada na Figura 16, e uma representação visual desta ação pode ser encontrado na Figura 17. A ação `chargeBattery` tem um *parameter* de entrada chamado `battery` que recebe o valor inicial da bateria do tipo de dado real com um limite de 0 à 100 que é definido como `doc /* Real={0..100} */`. O valor inicial da bateria é passado para o nó *merge* `continueCharging`, que passa esse valor para o nó *action* `monitor` que apenas apresenta o valor atual da bateria e em seguida passa o valor para o nó *decision* `decisionNode`. O nó `decisionNode` direciona o fluxo dependendo do valor atual da bateria, se o valor for menor que 100, o nó `decisionNode` envia o valor para o nó *action* `addCharge`, mas caso o valor seja maior que 100, o nó `decisionNode` direciona o fluxo para o nó *action* `endCharging` que em seguida finaliza a ação. O nó `addCharge` adiciona mais 1 ao valor atual da bateria e envia este valor de volta para o nó `continueCharging` que inicia novamente o fluxo.

```
import ScalarValues::*;

action chargeBattery (in battery : Real) {

    doc /* Real={0..100} */

    bind chargeBattery::battery = continueCharging;
    merge continueCharging;
    flow continueCharging to monitor::batteryIn;
    action monitor (in batteryIn : Real, out batteryOut : Real){
        language "Alf"
        /* batteryOut = batteryIn; */
    }
    flow monitor::batteryOut to decisionNode;
    decide decisionNode;
        if monitor::batteryOut < 100 then addCharge::batteryIn;
        if monitor::batteryOut > 100 then endCharging;
    action addCharge (in batteryIn : Real, out batteryOut : Real) {
        language "Alf"
        /* batteryOut = batteryIn + 1; */
    }
    flow addCharge::batteryOut to continueCharging;
    action endCharging;
    succession endCharging then done;
}
```

Figura 16 – Especificação SysML 2.0 da ação do carregador de bateria

Após utilizar o verificador para analisar *deadlock* na ação, é identificado um *deadlock* durante a análise e o contraexemplo retornado por ser encontrado na Figura 18, e sua representação visual pode ser encontrada na Figura 19. As linhas comentadas com `// [deadlock trace]` na especificação SysML 2.0 são os mesmos nós e arestas

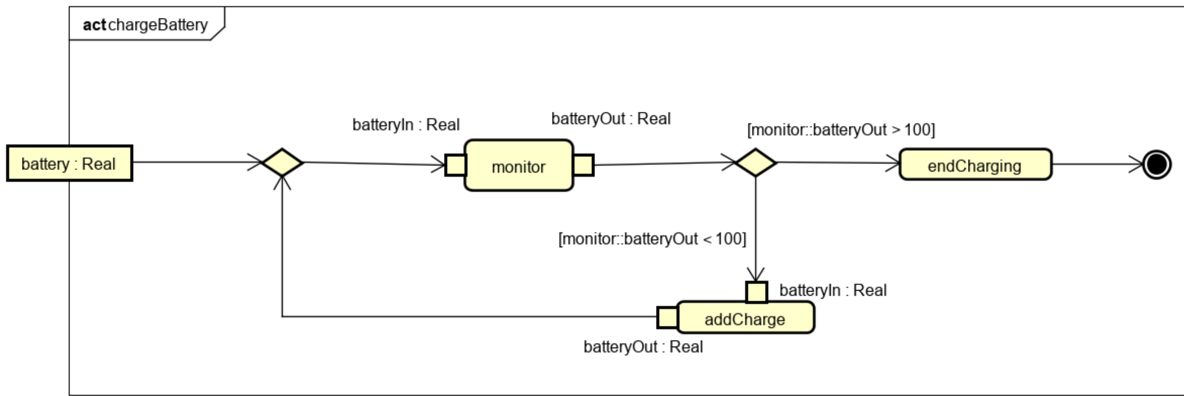


Figura 17 – Representação visual da ação do carregador de bateria

que estão pintados de vermelho na representação visual. Logo o usuário consegue identificar até onde exatamente a ação foi percorrida até encontrar o *deadlock*. Neste caso o *deadlock* ocorreu no nó *decision* pelo fato de que o valor 100 da bateria não é satisfeito por nenhumas das arestas de saída, logo a ação fica presa neste estado sem conseguir finalizar. Uma possível solução para este problema seria adicionar o valor 100 à uma das arestas de saída, como por exemplo, uma aresta trataria dos valores menores que 100 e a outra aresta trataria dos valores maiores ou igual a 100.

```
import ScalarValues::*;

action chargeBattery (in battery : Real) { // [deadlock trace]

    doc /* Real={0..100} */

    bind chargeBattery::battery = continueCharging; // [deadlock trace]
    merge continueCharging; // [deadlock trace]
    flow continueCharging to monitor::batteryIn; // [deadlock trace]
    action monitor (in batteryIn : Real, out batteryOut : Real){ // [deadlock trace]
        language "Alf"
        /* batteryOut = batteryIn; */
    }
    flow monitor::batteryOut to decisionNode; // [deadlock trace]
    decide decisionNode; // [deadlock trace]
        if monitor::batteryOut < 100 then addCharge::batteryIn;
        if monitor::batteryOut > 100 then endCharging;
    action addCharge (in batteryIn : Real, out batteryOut : Real) {
        language "Alf"
        /* batteryOut = batteryIn + 1; */
    }
    flow addCharge::batteryOut to continueCharging;
    action endCharging;
    succession endCharging then done;
}
```

Figura 18 – Contraexemplo SysML 2.0 da ação do carregador de bateria

Para analisar um não-determinismo, suponhamos que o usuário adicione o valor 100 da bateria as duas arestas de saída do `decisionNode`, como apresentado na Figura 20 (representação visual na Figura 21).

Após utilizar o verificador para analisar não-determinismo na ação, é identificado um não-determinismo durante a análise e o contraexemplo retornado por ser encontrado na Figura 22 com sua representação visual na Figura 23. As linhas comen-

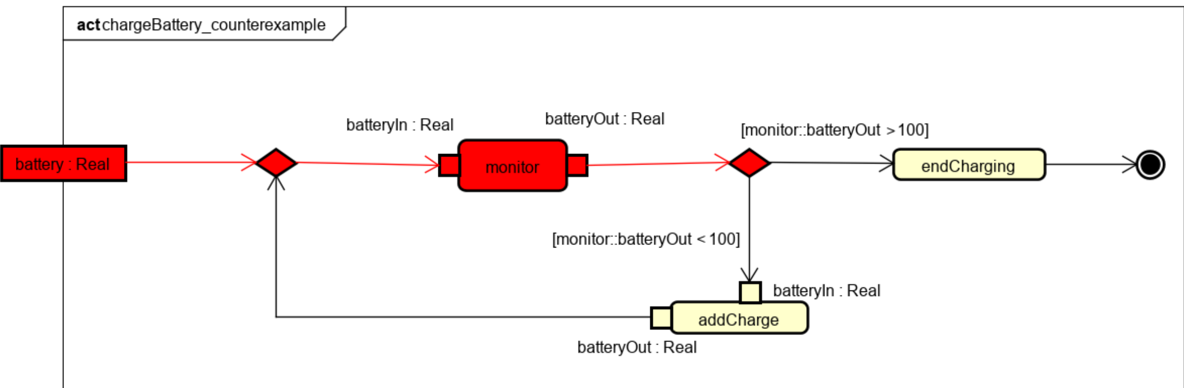


Figura 19 – Representação visual do contraexemplo da ação do carregador de bateria

```
decide decisionNode;
    if monitor::batteryOut <= 100 then addCharge::batteryIn;
    if monitor::batteryOut >= 100 then endCharging;
```

Figura 20 – Modificação da especificação SysML 2.0 da ação do carregador de bateria

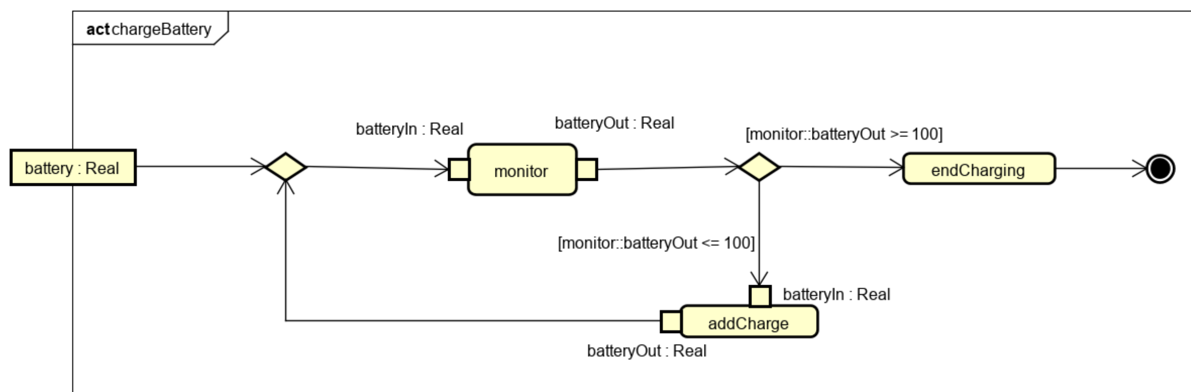


Figura 21 – Representação visual da modificação da ação do carregador de bateria

tadas com `// [non-determinism trace]` na modificação da especificação SysML 2.0 são os mesmos nós e arestas que estão pintados de vermelho na representação visual. Logo o usuário consegue identificar até onde exatamente a ação foi percorrida até encontrar o não-determinismo. Neste caso o não-determinismo ocorreu no nó *decision* pelo fato de que o valor 100 da bateria é satisfeito por ambas as arestas de saída, logo a ação precisa fazer uma escolha não-determinística entre as duas arestas para continuar o seu fluxo, tornando assim a ação não-determinística. Uma possível solução para este problema seria adicionar o valor 100 a apenas uma das arestas de saída.

O segundo estudo de caso é uma representação de um sistema de freios baseado no exemplo *Decision Example* que pode ser encontrado no repositório (OMG, 2021). A especificação SysML 2.0 pode ser encontrada na Figura 24, e uma representação visual desta ação pode ser encontrado na Figura 25. A ação `brake` inicia no nó *initial* e em seguida o sistema é ligado no nó `TurnOn`, após isso, é iniciado simulta-

```
import ScalarValues::*;

action chargeBattery (in battery : Real) { // [non-determinism trace]

    doc /* Real={0..100} */

    bind chargeBattery::battery = continueCharging; // [non-determinism trace]
    merge continueCharging; // [non-determinism trace]
    flow continueCharging to monitor::batteryIn; // [non-determinism trace]
    action monitor (in batteryIn : Real, out batteryOut : Real){ // [non-determinism trace]
        language "Alf"
        /* batteryOut = batteryIn; */
    }
    flow monitor::batteryOut to decisionNode; // [non-determinism trace]
    decide decisionNode; // [non-determinism trace]
        if monitor::batteryOut <= 100 then addCharge::batteryIn;
        if monitor::batteryOut >= 100 then endCharging;
    action addCharge (in batteryIn : Real, out batteryOut : Real) {
        language "Alf"
        /* batteryOut = batteryIn + 1; */
    }
    flow addCharge::batteryOut to continueCharging;
    action endCharging;
    succession endCharging then done;
}
```

Figura 22 – Contraexemplo da modificação SysML 2.0 da ação do carregador de bateria

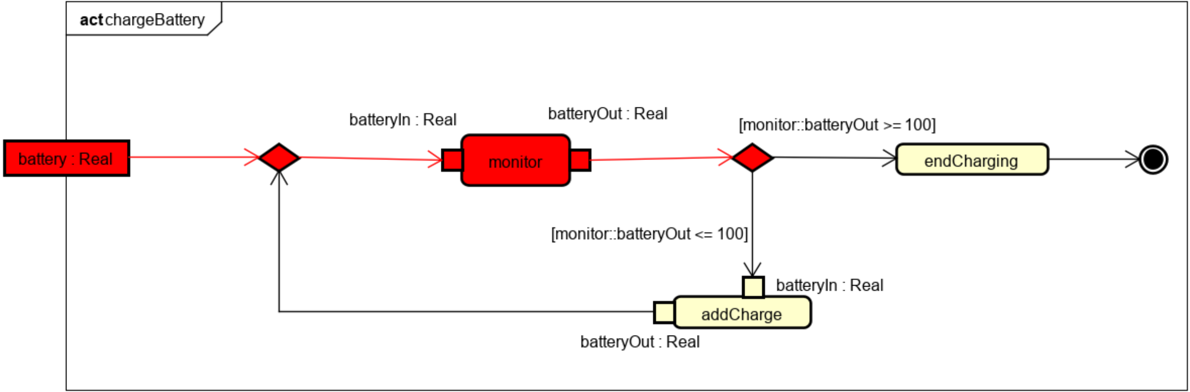


Figura 23 – Representação visual do contraexemplo da modificação da ação do carregador de bateria

neamente três ações chamadas de `monitorBrakePedal`, `monitorTraction` e `braking` onde as duas primeiras ações enviam dados para a terceira ação. A ação é encerranda apenas quando todas os três nós ações sincronizarem no nó *join*.

Após utilizar o verificador para analisar *deadlock* na ação, é identificado um *deadlock* durante a análise e o contraexemplo retornado por ser encontrado na Figura 26, e sua representação visual pode ser encontrada na Figura 27. Logo o usuário consegue identificar até onde exatamente a ação foi percorrida até encontrar o *deadlock*. Neste caso o *deadlock* ocorreu no nó `monitorTraction` porque não conseguiu enviar todas as suas arestas de saída, isso oconteceu pelo fato do valor enviado no *pin* de saída `modulationFrequency` que é do tipo de dado Real ser igual a 2, e este valor não está incluíndo nos limites do tipo Real que foi definido como `Real={0..1}`, ou seja, va-

```
import ScalarValues::*;

attribute def BrakePressure;

action brake {

    doc /* Real={0..1}; BrakePressure={0..1} */

    first start;
    succession start then TurnOn;
    action TurnOn;
    succession TurnOn then forkNode;
    fork forkNode;
    succession forkNode then monitorBrakePedal;
    succession forkNode then monitorTraction;
    succession forkNode then braking;
    action monitorBrakePedal (out pressure : BrakePressure) {
        language "Alf"
        /* pressure = 1; */
    }
    succession monitorBrakePedal then joinNode;
    flow monitorBrakePedal::pressure to braking::brakePressure;
    action monitorTraction (out modulationFrequency : Real){
        language "Alf"
        /* modulationFrequency = 2; */
    }
    succession monitorTraction then joinNode;
    flow monitorTraction::modulationFrequency to braking::modulationFrequency;
    action braking (in brakePressure : BrakePressure, in modulationFrequency : Real);
    succession braking then joinNode;
    join joinNode;
    succession joinNode then done;
}
```

Figura 24 – Especificação SysML 2.0 da ação do sistema de freios

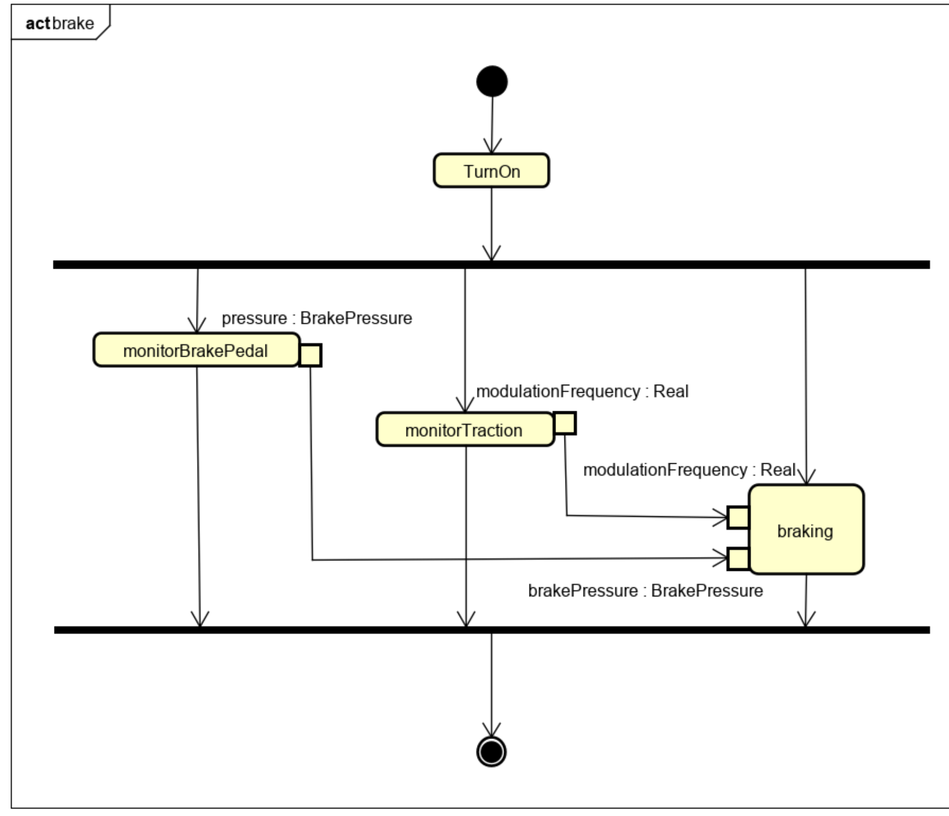


Figura 25 – Representação visual da ação do sistema de freios

lores de 0 à 1 (incluindo ambos os limites). Uma possível solução para este problema seria aumentar o limite superior do tipo de dado Real para um valor que englobe o valor 2.

```
import ScalarValues::*;

attribute def BrakePressure;

action brake {

    doc /* Real={0..1}; BrakePressure={0..1} */

    first start; // [deadlock trace]
    succession start then TurnOn; // [deadlock trace]
    action TurnOn;   // [deadlock trace]
    succession TurnOn then forkNode; // [deadlock trace]
    fork forkNode; // [deadlock trace]
    succession forkNode then monitorBrakePedal; // [deadlock trace]
    succession forkNode then monitorTraction; // [deadlock trace]
    succession forkNode then braking;    // [deadlock trace]
    action monitorBrakePedal (out pressure : BrakePressure) { // [deadlock trace]
        language "Alf"
        /* pressure = 1; */
    }
    succession monitorBrakePedal then joinNode;  // [deadlock trace]
    flow monitorBrakePedal::pressure to braking::brakePressure; // [deadlock trace]
    action monitorTraction (out modulationFrequency : Real){ // [deadlock trace]
        language "Alf"
        /* modulationFrequency = 2; */
    }
    succession monitorTraction then joinNode; // [deadlock trace]
    flow monitorTraction::modulationFrequency to braking::modulationFrequency;
    action braking (in brakePressure : BrakePressure, in modulationFrequency : Real);
    succession braking then joinNode;
    join joinNode; // [deadlock trace]
    succession joinNode then done;
}
```

Figura 26 – Contraexemplo SysML 2.0 da ação do sistema de freios

O terceiro estudo de caso é uma representação de um sistema de vendas online que é apresentado em (LIMA; TAVARES; NOGUEIRA, 2020). Como a especificação SysML 2.0 desta ação é muito grande, ela pode ser encontrada no Apêndice A, já a representação visual desta ação pode ser encontrada na Figura 28. A ação `ecommerc` inicia no nó *initial* e em seguida executa o pedido do cliente no nó `Receive order`, após este momento, o fluxo em dividido em dois simultâneos. No primeiro fluxo o sistema checa o estoque do produto no nó `Check stock`, caso não tenha estoque será realizado o plano de produção do produto no nó `Make production plan`, após isso, o fluxo entra em espera no nó `WAIT-1`. O segundo fluxo é uma checagem que o cliente precisa realizar, sendo novamente divido em dois novos fluxos, o primeiro fluxo apenas realiza a confirmação do cliente para dar continuidade no fluxo do produto, já o segundo fluxo é a realização do pagamento pelo cliente, onde é enviado a conta do pagamento para o cliente no nó `Send bill`, e em seguida é aguardado o pagamento, caso passe mais de duas semanas sem realizar o pagamento, é enviado um lembrete para o cliente no nó `Send reminder`.

Após o usuário realizar o pagamento, este pagamento é tratado no nó `Handle`

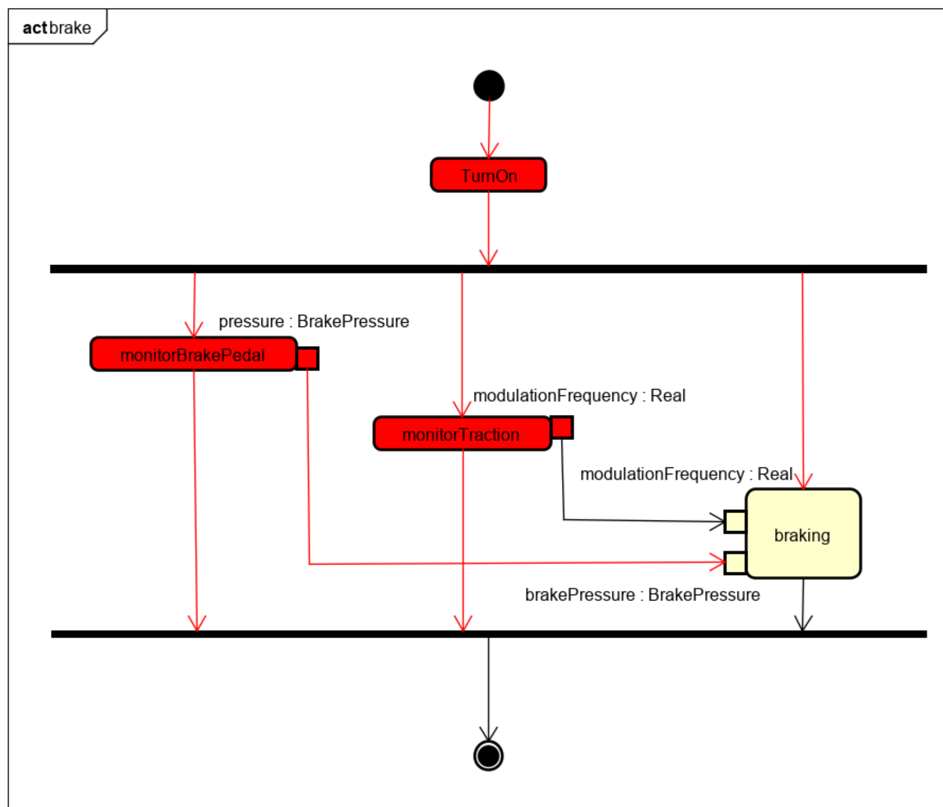


Figura 27 – Representação visual do contraexemplo da ação do sistema de freios

`payment`, e caso o pagamento não seja aprovado, o cliente é notificado no nó `Notify customer`, caso contrário, o fluxo atual aguarda o fluxo do produto para finalizar a compra. Continuando o fluxo do produto, caso não tenha estoque é executado o plano de produção para só após isso finalizar a compra, caso possua estoque, é preenchido os dados do pedido no nó `Fill order`. E por fim, o produto é enviado no nó `Ship order` e é finalizado o fluxo da ação.

Após utilizar o verificador para analisar *deadlock* na ação, é identificado um *deadlock* durante a análise e o contraexemplo retornado por ser encontrado no Apêndice B (pelo fato de ser uma especificação muito grande), já a sua representação visual pode ser encontrada na Figura 29. Logo o usuário consegue identificar até onde exatamente a ação foi percorrida até encontrar o *deadlock*. Neste caso o *deadlock* ocorreu no nó `WAIT-3`, porque como sabemos um nó *action* precisa que todas as sua arestas de entrada sejam comunicadas antes de comunicar suas arestas de saída, e neste caso o nó `WAIT-3` e `Send reminder` tem uma dependência cíclica entre si, sendo assim o nó `WAIT-3` nunca completará todas suas arestas de entrada. Uma possível solução para este problema (que é apresentada na Figura 30) seria adicionar um nó *merge* antes do nó `WAIT-3` para que suas entradas sejam conectadas nesse novo nó *merge* e a saída do nó *merge* ser conectada ao nó `WAIT-3`, com isso o nó `WAIT-3` não é mais dependente das duas entradas e sim de apenas uma delas para executar.

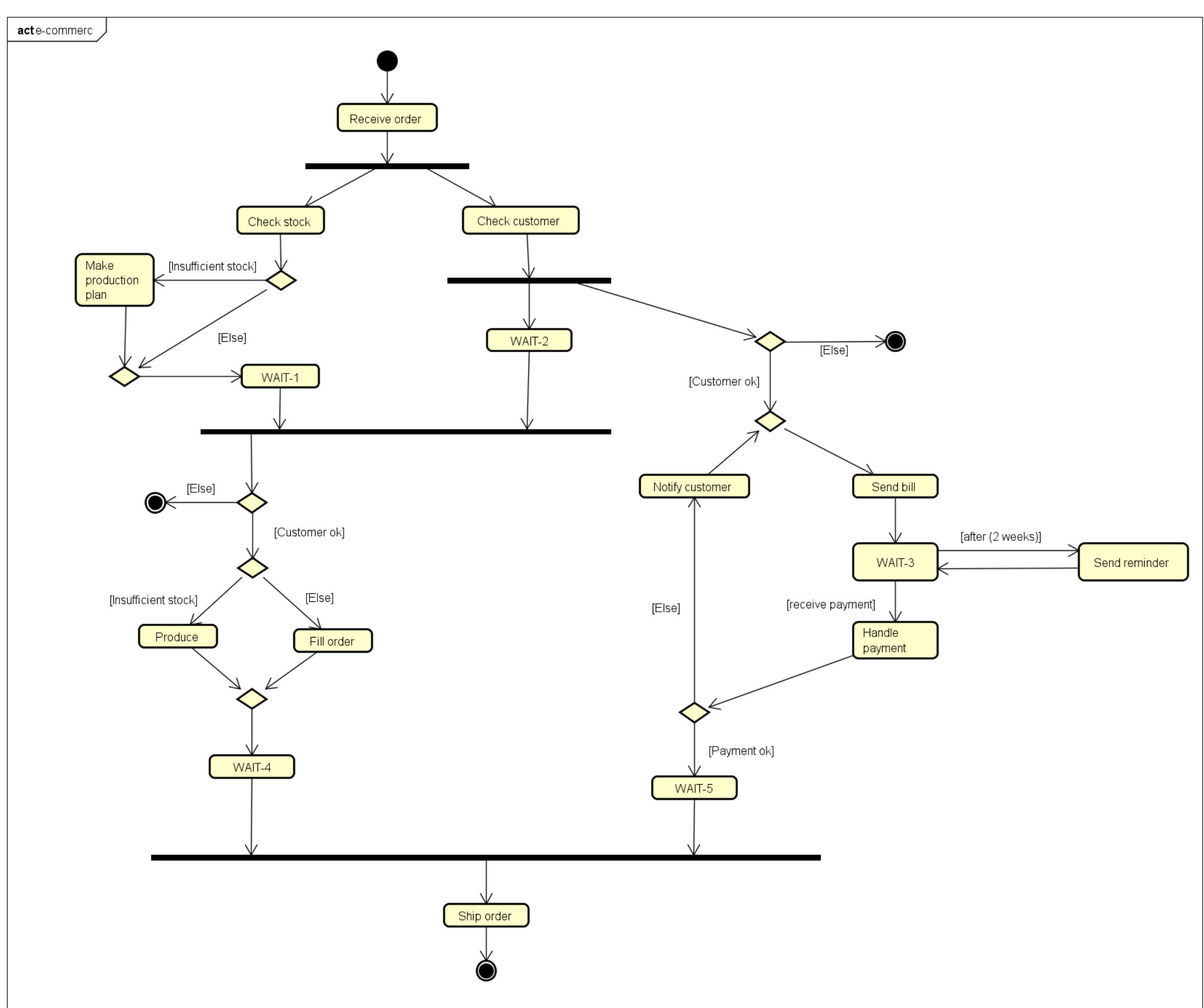


Figura 28 – Representação visual da ação do sistema de vendas online

O quarto e último estudo de caso é uma representação de um plano de produção de produtos. Como a especificação SysML 2.0 desta ação é bastante extensa, por questões de legibilidade, ela não é apresentada neste capítulo, mas pode ser vista no Apêndice C, já a representação visual desta ação pode ser encontrada na Figura 31. A ação `ProductionPlan` é subdividida em três outras ações, sendo a primeira ação chamada de `Comunication` que é responsável pela comunicação com o fornecedor das peças do produto, a segunda ação é chamada de `Process` que é responsável pela produção do produto e a terceira ação é chamada de `Inicialization` que é responsável pelos processos iniciais da fabricação do produto. Os nós *call behaviors* `comunication1` e `comunication2` são instâncias da ação `Comunication`, os nós `step1` e `step2` são instâncias da ação `Process` e o nó `inicialization` é uma instância da ação `Inicialization`.

A ação `ProductionPlan` é responsável por orquestrar todas as interações destas subações. O fluxo inicia a partir do nó *parameter* e em seguida o fluxo é dividido em três fluxos simultâneos, sendo dois fluxos para comunicação com o fornecedores das

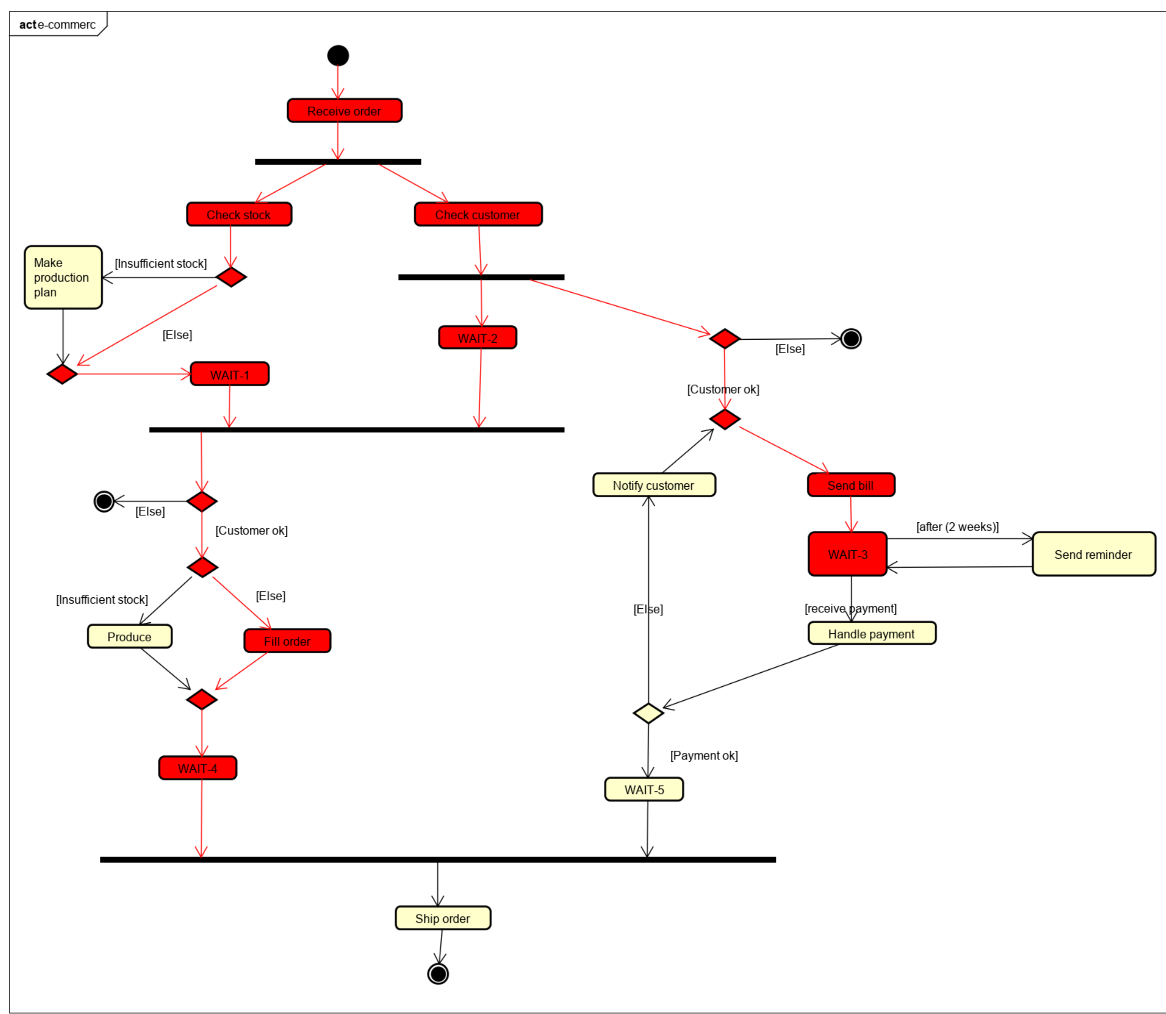


Figura 29 – Representação visual do contraexemplo da ação do sistema de vendas online

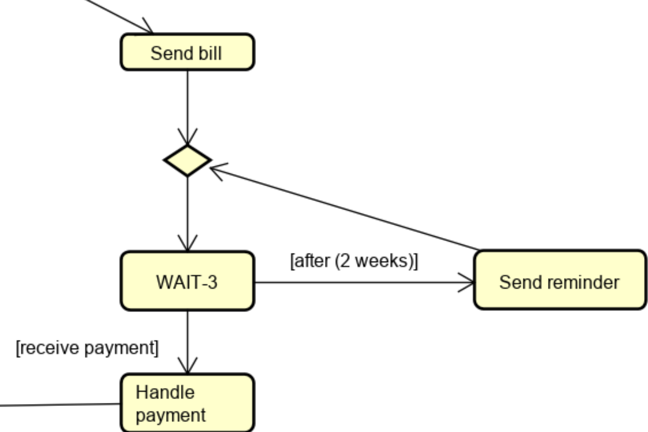


Figura 30 – Representação visual da correção da ação do sistema de vendas online

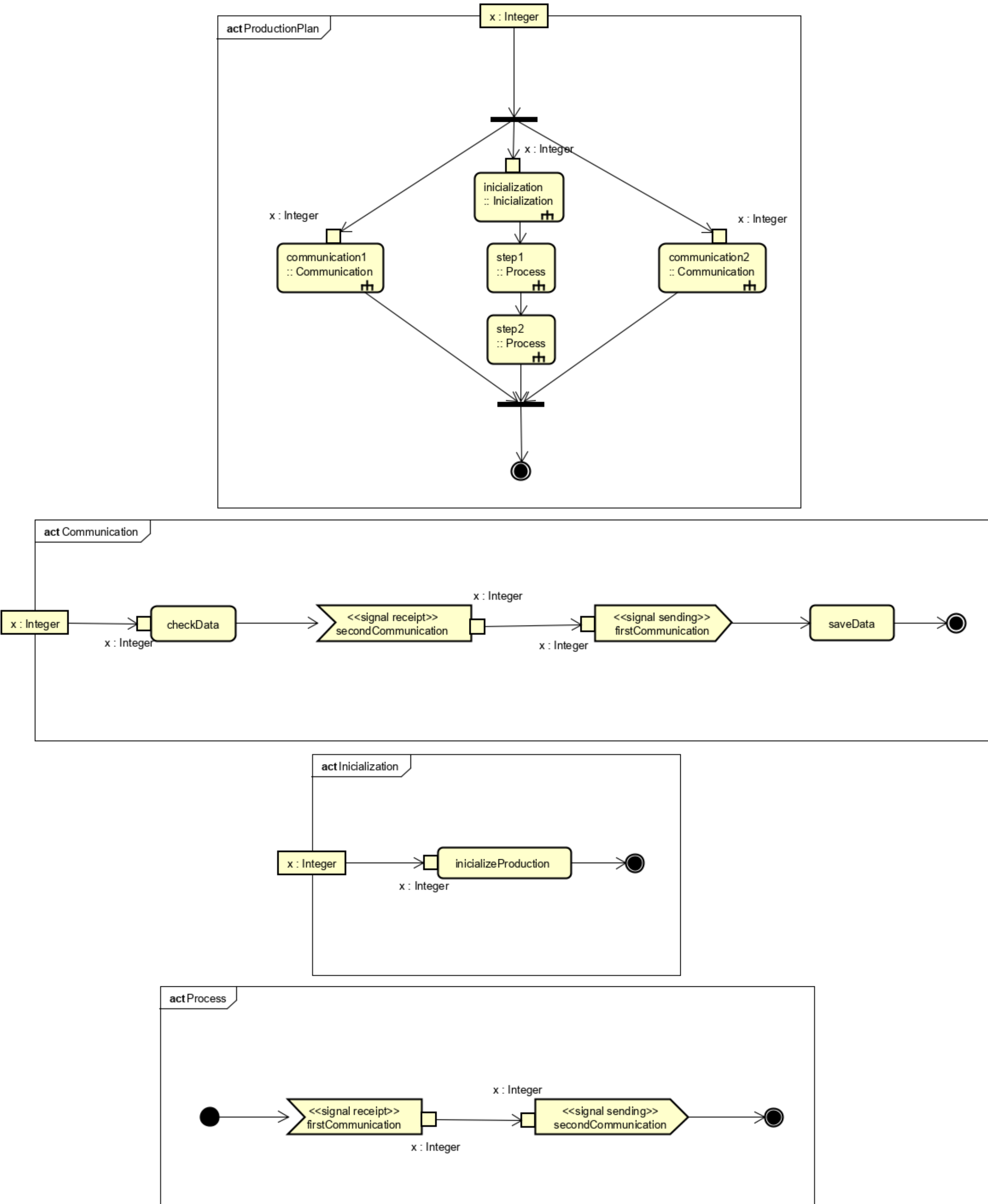


Figura 31 – Representação visual da ação do plano de produção de produtos

peças nos nós `comunication1` e `comunication2`, e outro fluxo de produção que inicial com o nó `inicialization`. Quando o nó `inicialization` finaliza, é iniciado o nó `step1`, mas tanto o nó `step1` quanto o nó `step2` precisam receber uma mensagem de comunicação para continuar seus fluxos, essa mensagem pode vir dos nós `comunication1` ou `comunication2`, que também precisa de mensagem vindas dos nós `step1` e `step2` para continuarem seus fluxos, logo esses nós são dependentes entre si para funcionarem. Após todos esses nós concluirem, são sincronizados com o nó *join* que aguarda todos os fluxos serem sincronizados para finalizar a ação.

Casos como este podem se tornar complexos para se verificar manualmente, por isso utilizando o verificador é possível obter mais agilidade nesta análise. Sendo assim após utilizar o verificador para analisar *deadlock* na ação, é identificado um *deadlock* durante a análise e o contraexemplo retornado por ser encontrado no Apêndice D (pelo fato de ser uma especificação extensa), já a sua representação visual pode ser encontrada na Figura 32. Logo o usuário consegue identificar até onde exatamente a ação foi percorrida até encontrar o *deadlock*. Neste caso o *deadlock* ocorreu pelo fato dos nós `comunication1` e `comunication2` esperarem sinais `secondCommunication` antes de enviar o sinal `firstCommunication`. Só que o `secondCommunication` só é enviado após a recepção do `firstCommunication` nos nós `step1` e `step2`. Uma possível solução para este problema é inverter a ordem do nó `secondCommunication` com o nó `firstCommunication`, como pode ser observado na Figura 33 (com sua representação visual na Figura 34). Com isso os nós não ficam travados em uma espera indefinida pelo sinal e o deadlock deixa de existir.

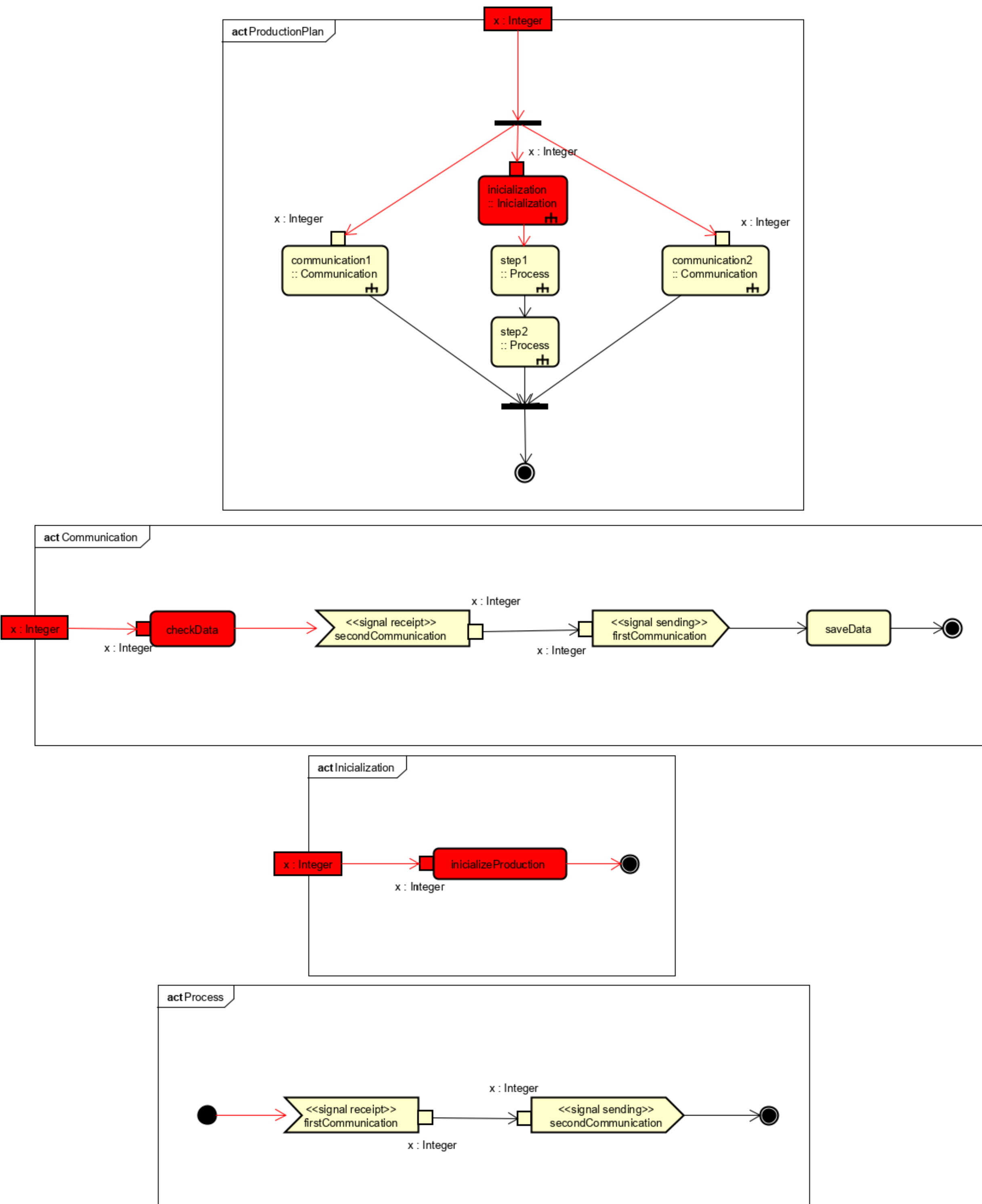


Figura 32 – Representação visual do contraexemplo da ação do plano de produção de produtos

```
action def Communication (in x : Integer) {
    doc /* Integer={0..1} */
    bind Communication::x = checkData::x;
    action checkData (in x : Integer, out y : Integer) {
        language "Alf"
        /* y = x; */
    }
    action sendFirstCommunication send checkData::y to acceptFirstCommunication;
    succession sendFirstCommunication then acceptSecondCommunication;
    action acceptSecondCommunication accept x : Integer;
    flow acceptSecondCommunication::x to saveData::x;
    action saveData (in x : Integer);
    succession saveData then done;
}
```

Figura 33 – Especificação SysML 2.0 da solução da ação do plano de produção de produtos

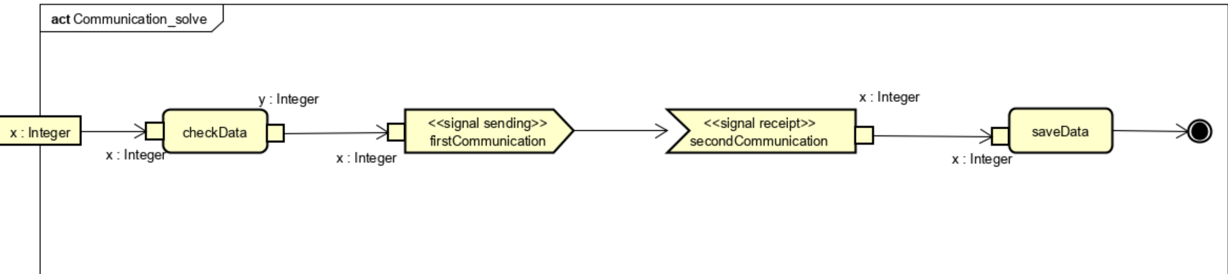


Figura 34 – Representação visual da solução da ação do plano de produção de produtos

5 Conclusão

Neste trabalho apresentamos um verificador que analisa propriedades em ações de SysML 2.0, mais precisamente, verificando a presença de *deadlock* e não-determinis mo. O verificador utiliza uma semântica formal definida em termos da álgebra do processo CSP de acordo com trabalhos anteriores, os quais foram adaptados e incrementados a fim de aumentar a expressividade da ferramenta. O verificador proposto, possui definições para todos os tipos de nós (*action*, *control* e *object*) e arestas (*control flow* e *object flow*) seguindo uma estratégia composicional, ou seja, cada construtor possui uma semântica dissociada das demais. Isso facilita a mecanização da estratégia, a validação do significado de cada elemento e a semântica pode ser facilmente estendida. Também aproveitamos a ampla gama de operadores CSP que suportam essa composicionalidade.

Embora o CSP forneça uma linguagem rigorosa e inequívoca aliada a um suporte de ferramenta estável (FDR), os engenheiros de sistema e de *software*, geralmente, não apreciam a manipulação de notações matemáticas complexas para projetar seus modelos. Portanto, um ponto forte do verificador é evitar que os usuários tenham qualquer contato com notações formais para fornecer entradas ou para ler saídas de nossa ferramenta. Isso é conseguido ocultando todas as traduções de modelos SysML 2.0 para especificações CSP, e também pelo mecanismo de rastreabilidade da saída de modelos FDR para SysML 2.0. Acreditamos que esse seja um recurso atrativo para os profissionais e ferramentas de modelagem que irão utilizar a SysML 2.0 em comparação com outros trabalhos.

Outra característica distinta do nosso trabalho é a verificação do não-determinis mo. Embora o *deadlock* seja amplamente discutido e coberto na literatura, o não-determinismo é uma propriedade que geralmente é deixada de lado. No entanto, se considerarmos a crescente complexidade de sistemas e comportamentos com muitas decisões a serem avaliadas, essa propriedade torna-se extremamente relevante. Portanto, detectar a presença de propriedades pode não ser uma tarefa trivial. Mostramos alguns cenários com a detecção de *deadlock* e não-determinismo em ações de SysML 2.0, entretanto, também foram exercitados diversos exemplos de ações de SysML 2.0 para validar a estratégia e demonstrar sua viabilidade, incluindo alguns que são propostos na próprio repositório da SysML 2.0 (OMG, 2021). Por fim, encorajamos a aplicação do verificador em qualquer cenário onde a ausência ou presença de *deadlock* e não-determinismo deva ser garantida.

5.1 Trabalhos Relacionados

A maior parte das abordagens apresentadas a seguir seguem o padrão de traduzir diagramas de atividades (forma de especificação de ações em SysML 1.0) para uma linguaguem que seja possível de verificar propriedades e que seja equivalente ao diagrama, e em seguida usa um verificador de modelos para analisar essas propriedades. Porém nenhuma delas apresenta rastreabilidade automática de volta para diagrama de atividade e também não trabalham com a sintaxe proposta pela SysML 2.0, visto que é uma linguagem que ainda está sendo proposta e oficialmente não foi lançada ainda.

A abordagem apresentada por (BALDAN; CORRADINI; GADDUCCI, 2004), representa diagramas de classes, instâncias e atividade como um Sistema de Transformação de Grafo. O diagrama de classe determina a estrutura dos grafos que irão ser usados para modelar os diagramas de instância, que são chamados de grafos de instâncias. Os diagramas de atividade são representados como regras de transformação de grafos e cada nó de atividade é transformado em uma regra. Em seguida é introduzida a lógica proposicional μL2 para representar algumas propriedades comportamentais. E sobre certas restrições, propriedades podem ser verificadas automaticamente, mas apesar da verificação ser automática, a transformação precisa ser feita manualmente, além de também não apresentar rastreabilidade de volta para o diagrama de atividade.

A abordagem apresentada por (ELMANSOURI; HAMROUCHE; CHAOUI, 2011), utiliza transformação de grafo, definindo um metamodelo para diagrama de atividade e uma gramática de grafo (uma generalização da gramática de Chomsky para grafos) que executa automaticamente a transformação dos diagramas de atividade criados na ferramenta ATOM. É apenas realizada a transformação do diagrama de atividade para CSP, mas não realizada nenhuma verificação de propriedades em cima da especificação CSP.

A abordagem apresentada por (ESHUIS, 2006), utiliza um processo de duas etapas de traduções para traduzir o diagrama de atividade para a linguagem do verificador de modelo NuSVM, traduzindo em uma máquina de estados finita. Porém o diagrama não suporta nós de *fork* e *join*, sendo necessário antes transformar o diagrama de atividade em um hipergrafo de atividade. Algumas propriedades são expressadas e verificadas em *Past Linear Temporal Logic* (PLTL-X), no entanto, não é verificada a propriedade de deadlock, e também não é apresentada rastreabilidade de volta para diagrama de atividade.

A abordagem apresentada por (ABDELHALIM et al., 2010), traduz um subconjunto da *Foundational Subset for Executable UML* (fUML) para CSP e em seguida usa

o verificador de modelo FDR para verificar *deadlock*. Apesar de verificar *deadlock*, não é apresentada nenhuma rastreabilidade de volta para diagrama de atividade.

A abordagem apresentada por (OUCHANI; MOHAMED; DEBBABI, 2014), traduz o diagrama de atividade para a linguagem do verificador de modelos PRISM. O PRISM verifica as especificações probabilísticas sobre modelos probabilísticos, essas especificações podem ser expressadas em Probabilistic Computation Tree Logic (PCTL). Apesar de ser verificada a propriedade de *deadlock*, o *framework* não apresenta rastreabilidade de volta para diagramas de atividade.

A abordagem apresentada por (BANTI; PUGLIESE; TIEZZI, 2011) traduz automaticamente os modelos UML4SOA em termos COWS para verificar automaticamente as propriedades. Embora o processo não exija conhecimento sobre os métodos formais utilizados, o método não possui uma forma automática de rastreabilidade. Além disso, a semântica de alguns termos, como o nó *merge*, não parece ser composicional.

A abordagem apresentada por (Alawneh et al., 2006) é uma estrutura desenvolvida para verificar diagramas UML comportamentais (máquinas de estado, diagramas de atividades e diagramas de sequência). Para cada diagrama, um modelo semântico formal é derivado refletindo suas características e expressa suas propriedades como fórmulas de lógica temporal. O modelo semântico é chamado de *Configuration Transition System* (CTS), que são propriedades descritas na lógica temporal que são usadas como entrada para a ferramenta de verificação NuSMV. A proposta é traduzir o diagrama para CTS e verificar as propriedades com a ferramenta NuSMV. Embora façam verificações de propriedades, falta rastreabilidade de volta ao diagrama de atividades, deixando a avaliação e interpretação dos problemas para o usuário de acordo com sua experiência na notação formal.

5.2 Trabalhos Futuros

No estado atual, o verificador oferece suporte a tipos básicos. Como qualquer mecanismo de verificação de modelo limitado, há alguns casos em que a quantidade de estados podem crescer exponencialmente se o modelo lidar com tipos de dados com um grande número de valores. Isso pode inviabilizar a análise. Para evitar isso, é utilizado um mecanismo onde o usuário pode restringir o intervalo de valores a serem considerados na análise. Para tornar esse processo menos dependente do usuário, planejamos integrar o verificador á solucionadores((MOURA; BJØRNER, 2008), (DUTERTRE; MOURA, 2006), (CIMATTI et al., 2013), entre outros) para definir automaticamente os intervalos apropriados. O verificador também suporta um número considerável de construtores de ações, incluindo os mais utilizados. No entanto, devido ao aspecto de composicionalidade de nossa semântica, esperamos estendê-la com mais

construtores, como por exemplo, o nó *final flow* que ainda não possui uma definição em SysML 2.0, mas também elementos de outros tipos de diagramas, para fornecer o máximo de expressividade possível aos usuários do verificador. Um outro ponto a ser analisado no futuro é a escalabilidade do verificador não apenas em termos de quantidades de nós e arestas, mas também tem termos de fluxos paralelos, assim como também realizar uma rastreabilidade gráfica utilizando o PlantUML (ROQUES, 2016).

Outros estudos como utilizar notação de metadados SysML 2.0 para marcar o *trace* do *deadlock* e não-determinismo, assim como também verificar outras propriedades como *livelock* e até propriedades mais específicas do domínio estão nos nossos planos futuros.

Referências

ABDELHALIM, I. et al. Formal verification of tokeneer behaviours modelled in fuml using csp. In: SPRINGER. *International Conference on Formal Engineering Methods*. [S.l.], 2010. p. 371–387. Citado 2 vezes nas páginas 14 e 52.

Alawneh, L. et al. A unified approach for verification and validation of systems and software engineering models. In: *13th Annual IEEE International Symposium and Workshop on Engineering of Computer-Based Systems (ECBS'06)*. [S.l.: s.n.], 2006. p. 10 pp.–418. Citado na página 53.

BAIER, C.; KATOEN, J.-P. *Principles of model checking*. [S.l.]: MIT press, 2008. Citado na página 13.

BALDAN, P.; CORRADINI, A.; GADDUCCI, F. Specifying and verifying uml activity diagrams via graph transformation. In: SPRINGER. *International Workshop on Global Computing*. [S.l.], 2004. p. 18–33. Citado 2 vezes nas páginas 14 e 52.

BANTI, F.; PUGLIESE, R.; TIEZZI, F. An accessible verification environment for uml models of services. *J. Symb. Comput.*, Academic Press, Inc., Duluth, MN, USA, v. 46, n. 2, p. 119–149, fev. 2011. ISSN 0747-7171. Disponível em: <http://dx.doi.org/10.1016/j.jsc.2010.08.005>. Citado na página 53.

BETTINI, L. *Implementing domain-specific languages with Xtext and Xtend*. [S.l.]: Packt Publishing Ltd, 2016. Citado na página 29.

CIMATTI, A. et al. The mathsat5 smt solver. In: SPRINGER. *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. [S.l.], 2013. p. 93–107. Citado na página 53.

DUTERTRE, B.; MOURA, L. D. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, v. 2, n. 2, p. 1–2, 2006. Citado na página 53.

ELMANSOURI, R.; HAMROUCHE, H.; CHAOUI, A. From uml activity diagrams to csp expressions: A graph transformation approach using atom^ sup 3^ tool. *International Journal of Computer Science Issues (IJCSI)*, International Journal of Computer Science Issues (IJCSI), v. 8, n. 2, p. 368, 2011. Citado 2 vezes nas páginas 14 e 52.

ESHUIS, R. Symbolic model checking of uml activity diagrams. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, ACM New York, NY, USA, v. 15, n. 1, p. 1–38, 2006. Citado 2 vezes nas páginas 14 e 52.

GIBSON-ROBINSON, T. et al. Fdr3 — a modern refinement checker for csp. In: ÁBRAHáM, E.; HAVELUND, K. (Ed.). *Tools and Algorithms for the Construction and Analysis of Systems*. [S.l.]: Springer Berlin Heidelberg, 2014, (Lecture Notes in Computer Science, v. 8413). p. 187–201. ISBN 978-3-642-54861-1. Citado 2 vezes nas páginas 14 e 23.

GOSLING, J. et al. *The Java language specification*. [S.l.]: Addison-Wesley Professional, 2000. Citado na página 26.

HASKINS, B. et al. 8.4.2 error cost escalation through the project life cycle. *INCOSE International Symposium*, v. 14, n. 1, p. 1723–1737, 2004. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/j.2334-5837.2004.tb00608.x>. Citado 2 vezes nas páginas 9 e 12.

HAUSE, M. et al. The sysml modelling language. In: *Fifteenth European Systems Engineering Conference*. [S.l.: s.n.], 2006. v. 9, p. 1–12. Citado 2 vezes nas páginas 12 e 18.

HOARE, C. A. R. *Communicating and Sequential Processes*. [S.l.]: Prentice Hall, 1985. Citado 2 vezes nas páginas 14 e 22.

KLOETZER, M.; BELTA, C. Dealing with nondeterminism in symbolic control. In: SPRINGER. *International Workshop on Hybrid Systems: Computation and Control*. [S.l.], 2008. p. 287–300. Citado na página 13.

LIMA, L.; TAVARES, A.; NOGUEIRA, S. C. A framework for verifying deadlock and nondeterminism in uml activity diagrams based on csp. *Science of Computer Programming*, Elsevier, p. 102497, 2020. Citado 6 vezes nas páginas 7, 13, 14, 27, 28 e 43.

MOURA, L. D.; BJØRNER, N. Z3: An efficient smt solver. In: SPRINGER. *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. [S.l.], 2008. p. 337–340. Citado na página 53.

OMG. Omg systems modeling language v1. In: OBJECT MANAGEMENT GROUP. 2007. Disponível em: <https://www.omg.org/spec/SysML/1.0/About-SysML/>. Citado na página 18.

OMG. Systems modeling language (sysml) v2 request for proposal (rfp). In: OBJECT MANAGEMENT GROUP. 2017. Disponível em: <http://www.omg.org/cgi-bin/doc.cgi?ad/2017-12-2>. Citado 2 vezes nas páginas 13 e 18.

OMG. Kernel modeling language (kerml). In: OBJECT MANAGEMENT GROUP. 2020. Disponível em: <https://www.omg.org/spec/UML/About-UML/>. Citado na página 19.

OMG. Omg systems modeling language v2. In: OBJECT MANAGEMENT GROUP. 2020. Disponível em: <https://www.omg.org/spec/SysML/2.0/About-SysML/>. Citado 4 vezes nas páginas 7, 13, 19 e 20.

OMG. Systems modeling application programming interface (api) and services. In: OBJECT MANAGEMENT GROUP. 2020. Disponível em: <https://www.omg.org/spec/UML/About-UML/>. Citado na página 13.

OMG. *Systems-Modeling/SysML-v2-Release*. 2021. <https://github.com/Systems-Modeling/SysML-v2-Release>. (Accessed on 06/02/2021). Citado 4 vezes nas páginas 29, 38, 40 e 51.

OUCHANI, S.; MOHAMED, O. A.; DEBBABI, M. A formal verification framework for sysml activity diagrams. *Expert Systems with Applications*, Elsevier, v. 41, n. 6, p. 2713–2728, 2014. Citado 2 vezes nas páginas 14 e 53.

REGGIO, G. et al. What are the used uml diagrams? a preliminary survey. In: *EESSMOD@MoDELS*. [S.l.: s.n.], 2013. Citado na página 12.

ROQUES, A. Drawing uml with plantuml. *PlantUML Language Reference Guide*, 2016. Citado na página 54.

ROSCOE, A.; HOARE, C.; BIRD, R. The theory and practice of concurrency (upper saddle river, nj, usa, 1997). *URL: https://www. cs. ox. ac. uk/people/bill. roscoe/publications/68b. pdf*, 1997. Citado 2 vezes nas páginas 13 e 23.

VISION, C. *Astah*. 2019. Disponível em: <http://astah.net/>. Citado na página 14.

A Especificação SysML 2.0 da ação para o sistema de vendas online

```
action ecommerc {
    first start;
    succession start then receiveOrder;
    action receiveOrder;
    succession receiveOrder then fork1;
    fork fork1;
    succession fork1 then checkCustomer;
    action checkCustomer;
    succession checkCustomer then fork2;
    fork fork2;
    succession fork2 then decision1;
    merge merge1;
    decide decision1;
        if "Customer ok" then merge1;
        else done;
    succession merge1 then sendBill;
    action sendBill;
    succession sendBill then wait3;
    action wait3;
    succession wait3 then sendReminder;
    succession sendReminder then wait3;
    action sendReminder;
    succession wait3 then handlePayment;
    action handlePayment;
    succession handlePayment then decision2;
    action wait5;
    action notifyCustomer;
    succession notifyCustomer then merge1;
    decide decision2;
        if "Payment ok" then wait5;
        else notifyCustomer;
    succession wait5 then join1;
    join join1;
    succession join1 then shipOrder;
    action shipOrder;
    succession shipOrder then done;
    succession fork1 then checkStock;
    action checkStock;
    succession checkStock then decision3;
    decide decision3;
        if "Insufficient stock" then makeProductionPlan;
        else merge2;
    action makeProductionPlan;
    merge merge2;
    succession makeProductionPlan then merge2;
    action wait1;
    succession merge2 then wait1;
    succession wait1 then join2;
    join join2;
    succession fork2 then wait2;
    action wait2;
    succession wait2 then join2;
    succession join2 then decision4;
    decide decision4;
        if "Customer ok" then decision5;
        else done;
    decide decision5;
        if "Insufficient stock" then produce;
        else fillOrder;
    action fillOrder;
    action produce;
    succession fillOrder then merge3;
    succession produce then merge3;
    merge merge3;
    succession merge3 then wait4;
    action wait4;
    succession wait4 then join1;
}
```

Figura 35 – Especificação SysML 2.0 da ação para o sistema de vendas online

B Contraexemplo SysML 2.0 da ação para o sistema de vendas online

```
action ecommerc {
    first start; // [deadlock trace]
    succession start then receiveOrder; // [deadlock trace]
    action receiveOrder; // [deadlock trace]
    succession receiveOrder then fork1; // [deadlock trace]
    fork fork1; // [deadlock trace]
    succession fork1 then checkCustomer; // [deadlock trace]
    action checkCustomer; // [deadlock trace]
    succession checkCustomer then fork2; // [deadlock trace]
    fork fork2; // [deadlock trace]
    succession fork2 then decision1; // [deadlock trace]
    merge merge1; // [deadlock trace]
    decide decision1; // [deadlock trace]
        if "Customer ok" then merge1; // [deadlock trace]
        else done;
    succession merge1 then sendBill; // [deadlock trace]
    action sendBill; // [deadlock trace]
    succession sendBill then wait3; // [deadlock trace]
    action wait3;
    succession wait3 then sendReminder;
    succession sendReminder then wait3;
    action sendReminder;
    succession wait3 then handlePayment;
    action handlePayment;
    succession handlePayment then decision2;
    action wait5;
    action notifyCustomer;
    succession notifyCustomer then merge1;
    decide decision2;
        if "Payment ok" then wait5;
        else notifyCustomer;
    succession wait5 then join1;
    join join1; // [deadlock trace]
    succession join1 then shipOrder;
    action shipOrder;
    succession shipOrder then done;
    succession fork1 then checkStock; // [deadlock trace]
    action checkStock; // [deadlock trace]
    succession checkStock then decision3; // [deadlock trace]
    decide decision3; // [deadlock trace]
        if "Insufficient stock" then makeProductionPlan;
        else merge2; // [deadlock trace]
    action makeProductionPlan;
    merge merge2; // [deadlock trace]
    succession makeProductionPlan then merge2;
    action wait1; // [deadlock trace]
    succession merge2 then wait1; // [deadlock trace]
    succession wait1 then join2; // [deadlock trace]
    join join2; // [deadlock trace]
    succession fork2 then wait2; // [deadlock trace]
    action wait2; // [deadlock trace]
    succession wait2 then join2; // [deadlock trace]
    succession join2 then decision4; // [deadlock trace]
    decide decision4; // [deadlock trace]
        if "Customer ok" then decision5; // [deadlock trace]
        else done;
    decide decision5; // [deadlock trace]
        if "Insufficient stock" then produce;
        else fillOrder; // [deadlock trace]
    action fillOrder; // [deadlock trace]
    action produce;
    succession fillOrder then merge3; // [deadlock trace]
    succession produce then merge3;
    merge merge3; // [deadlock trace]
    succession merge3 then wait4; // [deadlock trace]
    action wait4; // [deadlock trace]
    succession wait4 then join1; // [deadlock trace]
}
```

Figura 36 – Contraexemplo SysML 2.0 da ação para o sistema de vendas online

C Especificação SysML 2.0 da ação para o plano de produção de produtos

```
import ScalarValues::*;

action ProductionPlan (in x : Integer) {
    doc /* Integer={0..1} */
    fork fork1;
    bind ProductionPlan::x = fork1;
    flow fork1 to communication1::x;
    flow fork1 to communication2::x;
    flow fork1 to inicialization::x;
    action inicialization : Inicialization (in x : Integer);
    succession inicialization then step1;
    action communication1 : Communication (in x : Integer);
    succession communication1 then join1;
    action communication2 : Communication (in x : Integer);
    succession communication2 then join1;
    action step1 : Process;
    succession step1 then step2;
    action step2 : Process;
    succession step2 then join1;
    join join1;
    succession join1 then done;
}

action def acceptFirstCommunication;
action def acceptSecondCommunication;

action def Communication (in x : Integer) {
    doc /* Integer={0..1} */
    bind Communication::x = checkData::x;
    action checkData (in x : Integer);
    succession checkData then acceptSecondCommunication;
    action acceptSecondCommunication accept x : Integer;
    action sendFirstCommunication send acceptSecondCommunication::x to acceptFirstCommunication;
    succession sendFirstCommunication then saveData;
    action saveData;
    succession saveData then done;
}

action def Inicialization (in x : Integer) {
    doc /* Integer={0..1} */
    bind Inicialization::x = inicializeProduction::x;
    action inicializeProduction (in x : Integer);
    succession inicializeProduction then done;
}

action def Process {
    doc /* Integer={0..1} */
    first start;
    succession start then acceptFirstCommunication;
    action acceptFirstCommunication accept x : Integer;
    action sendSecondCommunication send acceptFirstCommunication::x to acceptSecondCommunication;
    succession sendSecondCommunication then done;
}
```

Figura 37 – Especificação SysML 2.0 da ação para o plano de produção de produtos

D Contraexemplo SysML 2.0 da ação para o plano de produção de produtos

```
import ScalarValues::*;

action ProductionPlan (in x : Integer) { // [deadlock trace]
    doc /* Integer={0..1} */
    fork fork1; // [deadlock trace]
    bind ProductionPlan::x = fork1; // [deadlock trace]
    flow fork1 to communication1::x; // [deadlock trace]
    flow fork1 to communication2::x; // [deadlock trace]
    flow fork1 to inicialization::x; // [deadlock trace]
    action inicialization : Inicialization (in x : Integer); // [deadlock trace]
    succession inicialization then step1; // [deadlock trace]
    action communication1 : Communication (in x : Integer);
    succession communication1 then join1;
    action communication2 : Communication (in x : Integer);
    succession communication2 then join1;
    action step1 : Process;
    succession step1 then step2;
    action step2 : Process;
    succession step2 then join1;
    join join1;
    succession join1 then done;
}

action def acceptFirstCommunication;
action def acceptSecondCommunication;

action def Communication (in x : Integer) {
    doc /* Integer={0..1} */
    bind Communication::x = checkData::x; // [deadlock trace]
    action checkData (in x : Integer); // [deadlock trace]
    succession checkData then acceptSecondCommunication; // [deadlock trace]
    action acceptSecondCommunication accept x : Integer;
    action sendFirstCommunication send acceptSecondCommunication::x to acceptFirstCommunication;
    succession sendFirstCommunication then saveData;
    action saveData;
    succession saveData then done;
}

action def Inicialization (in x : Integer) {
    doc /* Integer={0..1} */
    bind Inicialization::x = inicializeProduction::x; // [deadlock trace]
    action inicializeProduction (in x : Integer); // [deadlock trace]
    succession inicializeProduction then done; // [deadlock trace]
}

action def Process {
    doc /* Integer={0..1} */
    first start; // [deadlock trace]
    succession start then acceptFirstCommunication;  // [deadlock trace]
    action acceptFirstCommunication accept x : Integer;
    action sendSecondCommunication send acceptFirstCommunication::x to acceptSecondCommunication;
    succession sendSecondCommunication then done;
}
```

Figura 38 – Contraexemplo SysML 2.0 da ação para o plano de produção de produtos