DANILO YUDI FUTATA KASSUYA

TRANSFORMAÇÃO ENTRE CONTRATOS INTELIGENTES E REDES DE PETRI

LONDRINA
2023

DANILO YUDI FUTATA KASSUYA

TRANSFORMAÇÃO ENTRE CONTRATOS INTELIGENTES E REDES DE PETRI

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Ciência da Computação da Universidade Estadual de Londrina para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Adilson Luiz Bonifacio

LONDRINA
2023

Ficha de identificação da obra elaborada pelo autor, através do Programa de Geração Automática do Sistema de Bibliotecas da UEL

Sobrenome, Nome.
Título do Trabalho : Subtitulo do Trabalho / Nome Sobrenome. - Londrina, 2017.
100 f. : il.

Orientador: Nome do Orientador Sobrenome do Orientador.
Coorientador: Nome Coorientador Sobrenome Coorientador.
Dissertação (Mestrado em Ciência da Computação) - Universidade Estadual de Londrina, Centro de Ciências Exatas, Programa de Pós-Graduação em Ciência da Computação, 2017.
Inclui bibliografia.

1. Assunto 1 - Tese. 2. Assunto 2 - Tese. 3. Assunto 3 - Tese. 4. Assunto 4 - Tese. I. Sobrenome do Orientador, Nome do Orientador. II. Sobrenome Coorientador, Nome Coorientador. III. Universidade Estadual de Londrina. Centro de Ciências Exatas. Programa de Pós-Graduação em Ciência da Computação. IV. Título.

DANILO YUDI FUTATA KASSUYA

TRANSFORMAÇÃO ENTRE CONTRATOS INTELIGENTES E REDES DE PETRI

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Ciência da Computação da Universidade Estadual de Londrina para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA

Orientador: Prof. Dr. Adilson Luiz Bonifacio
Universidade Estadual de Londrina

Prof. Dr. Segundo Membro da Banca
Universidade/Instituição do Segundo Membro da Banca – Sigla instituição

---

Prof. Dr. Terceiro Membro da Banca
Universidade/Instituição do Terceiro Membro da Banca – Sigla instituição

---

Prof. Ms. Quarto Membro da Banca
Universidade/Instituição do Quarto Membro da Banca – Sigla instituição

Londrina, 24 de novembro de 2023.

AGRADECIMENTOS

Eu agradeço à minha mãe que me deu o suporte necessário para atingir meus objetivos, me incentivando a seguir meu sonhos. Aos meus colegas que me acompanharam durantes esses anos de estudo, pois sem eles essa fase teria sido bem mais árdua. Agradeço também à Universidade Estadual de Londrina e ao Departamento de Computação que me deram a oportunidade de aprender e aos professores que me concederam conhecimento, em especial ao professor Adilson, que me orientou durante todo o processo, sempre me ajudando no desenvolvimento deste trabalho e nunca desistindo de mim.

KASSUYA, D. Y. F.. **Transformação entre Contratos Inteligentes e Redes de Petri**. 2023. 47f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Universidade Estadual de Londrina, Londrina, 2023.

RESUMO

Com a popularização da tecnologia blockchain vem se tornando mais comum o uso de aplicações que dependam da tecnologia, entre elas se destaca a linguagem Solidity que permite armazenar e executar códigos em linguagem de programação dentro do ambiente Ethereum e que será o foco desse trabalho. Permitir armazenar programas dentro da rede blockchain apresenta um problema, um código armazenado na rede não poderá mais ser alterado uma vez que a forma como as redes blockchain garantem segurança não permitem alterações o que torna essencial que tais programas não possuam erros e funcionem da maneira esperada por qualquer duração que seja necessário. O objetivo desse trabalho é facilitar esse processo criando uma ferramenta para converter o código Solidity, que como muitas linguagens de programação, oferece certa dificuldade de entendimento do seu fluxo de execução, para uma Rede de Petri um modelo criado para observar sistemas distribuídos. Além disso a ferramenta também converterá Redes de Petri corrigidas para código Solidity novamente e poderá permitir até que sejam elaborados códigos Solidity por meio de Redes de Petri.

Palavras-chave: Contratos inteligentes, Solidity, Ethereum, Redes de Petri, Redes de Petri Colorida.

KASSUYA, D. Y. F.. **Converting Smart Contracts into Petri Nets**. 2023. 47p. Final Project (Bachelor of Science in Computer Science) – State University of Londrina, Londrina, 2023.

ABSTRACT

The popularization of blockchain technology has made more common to find tools that rely on it, and among all stands out the Solidity language which allow to store and execute codes in programming language inside the Ethereum environment and that will be the focus of this article. Allowing the storage of programs inside the blockchain presents a problem, once a code is stored inside in the blockchain it cannot be altered because of the way the blockchain ensures security a data stored in it cannot be altered in any way, this compels the programs to be save in the blockchain to be flawless, working the expected way regardless of how long it takes to finish. The objective of this work is to smooth the process creating a tool to convert solidity code, which is like many others computer languages thought to follow the execution, to Petri Net a model created to describe distributed systems. Moreover the tool will convert Petri Nets to Solidity allowing the user to create a Solidity code based on a new Petri Net or correct his code in Petri Net and redo the code.

Keywords: Smart contratcts, Solidity, Ethereum, Petri Net, Colored Petri Net.

LISTA DE ILUSTRAÇÕES

Figura 1 – Exemplo Rede de Petri 21
Figura 2 – Exemplo de Sequenciamento 22
Figura 3 – Exemplo de Distribuição 22
Figura 4 – Exemplo de Junção 22
Figura 5 – Exemplo de Escolha Não-Determinística 23
Figura 6 – Exemplo Rede de Petri Colorida 25
Figura 7 – Exemplo de Rede de Petri Colorida com Extensões 27
Figura 8 – Arquitetura do processo de transformação 29
Figura 9 – Representação visual da Rede de Petri Gerada 43

LISTA DE TABELAS

Tabela 1 – Variáveis de Solidity 19

LISTA DE ABREVIATURAS E SIGLAS

RP	Rede de Petri
RPC	Rede de Petri Colorida
TCC	Trabalho de Conclusão de Curso

SUMÁRIO

1 INTRODUÇÃO . . . 11

2 TRABALHOS RELACIONADOS . . . 13

3 FUNDAMENTAÇÃO . . . 14
3.1 Contratos Inteligentes . . . 14
3.2 Blockchain . . . 14
3.3 Ethereum . . . 15
3.4 Solidity . . . 17
3.5 Redes de Petri . . . 20
3.6 Redes de Petri Coloridas . . . 22
3.6.1 Extensão para RPCs . . . 27

4 DE CONTRATOS PARA REDES DE PETRI . . . 29
4.1 Arquitetura . . . 29
4.2 Processo de Transformação . . . 29
4.3 Aplicação prática da transformação . . . 35

5 CONCLUSÃO . . . 44

REFERÊNCIAS . . . 46

1 INTRODUÇÃO

Os contratos inteligentes[1] [1] surgiram com a tecnologia *Blockchain* [2], inicialmente concebida como um sistema criptográfico para armazenar dados. Contratos inteligentes foram criados com o intuito de tornar os contratos mais acessíveis, automatizando o processo de execução por meio das *Blockchains*. Uma característica importante na execução de contratos dessa natureza não requererem supervisão e, além disso, podem ser criados e usados por qualquer agente da Blockchain. A liberdade de uso e o baixo custo são os maiores atrativos da tecnologia.

Um contrato inteligente é implementado por linguagens de alto nível, tais como Vyper [3], Yul [4], Cairo [5] e Rust [6]. Porém, uma das linguagens mais difundidas é a Solidity [7], uma linguagem Turing-completa que permite a escrita de contratos inteligentes mais complexos, e não apenas para transferências de valores. A Solidity é uma linguagem orientada a objetos, com uma sintaxe similar ao *Javascript*, e concebida para a plataforma Ethereum [8], uma blockchain com moeda própria, o *ether*, usada nas transações da rede.

A tecnologia de contratos inteligentes ainda é nova, o que potencializa erros cometidos pelos programadores na implementação desses contratos. Para se evitar ou diminuir os equívocos cometidos na concepção de contratos inteligentes é importante que alguma verificação mais rigorosa seja realizada, e que seja preferencialmente automatizada. Neste sentido, o objetivo deste trabalho é colaborar para a área de verificação de contratos inteligentes usando modelos mais rigorosos para representação desses contratos.

As redes de Petri (RPs) [9] são um modelo para especificar sistemas concorrentes e linhas de produção. A visualização de uma RP torna a análise do sistema especificado mais simples, usando a ideia de consumo e geração de recursos. Essa característica permite que contratos inteligentes sejam representados por RPs. No entanto, para tornar mais fácil a modelagem de contratos inteligentes usando RPs, uma extensão mais apropriada tem sido empregada, as redes de Petri coloridas (RPCs) [10, 11]. Uma RPC possui a noção de cores para determinar diferentes tipos de recursos.

Neste cenário, o objetivo deste trabalho é transformar contratos inteligentes em modelos de RPC de forma sistemática. Com uma tradução sistemática entre os dois modelos é possível que análises e verificações mais rigorosas sejam realizadas sobre o modelo formal. Assim, possíveis falhas de contratos inteligentes descritos em Solidity podem ser encontradas através de uma verificação rigorosa sobre o modelo mais formal, resultante dessa transformação.

O restante do trabalho está organizado da seguinte forma. O Capítulo 2 aborda

[1] do inglês, *smart contracts*

os trabalhos relacionados ao desenvolvimento dessa proposta. O Capítulo 3 apresenta a fundamentação teórica e os conceitos relevantes sobre contratos inteligentes e blockchain, bem como os modelos de Redes de Petri. No Capítulo 4 é abordada a transformação dos contratos em Solidity para Redes de Petri Coloridas. Por fim, o Capítulo 5 apresentada as considerações finais do trabalho.

2 TRABALHOS RELACIONADOS

Este capítulo apresenta os trabalhos mais intimamente relacionados com o desenvolvimento desse Trabalho de Conclusão de Curso.

Alguns trabalhos na literatura já foram propostos com o objetivo de propiciar uma análise mais rigorosa sobre contratos inteligentes usando uma abordagem sobre autômatos finitos [12, 13]. Essa abordagem permite a verificação sobre os autômatos finitos para se encontrar erros, e então transformá-los em contratos. No entanto, RPCs representam contratos inteligentes de forma mais clara e direta, pelo uso das transições, que modelam ações e transações realizadas num contrato, e os marcadores, que representam de maneira mais precisa a ideia de valores e recursos, bem como seus diferentes tipos. Já os autômatos finitos, compostos de estados e transições, não capturam de maneira tão direta a ideia de recursos sendo gerados e consumidos.

Já García-Bañuelos at el. [14] apresenta um algoritmo que transforma um BPMN [1] num contrato inteligente. Nesse processo há uma etapa intermediária onde o BPMN é convertido, primeiramente, numa RP para só então se obter o respectivo contrato inteligente. Assim como o algoritmo desenvolvido nesse TCC, o algoritmo deles busca criar um contrato inteligente em Solidity.

[1] do inglês, *Business Process Model and Notation*

3 FUNDAMENTAÇÃO

Nesse capítulo são abordados os tópicos e referenciais teóricos utilizados como base para o desenvolvimento desse trabalho. As próximas seções descrevem os conceitos de *Blockchain* e contratos inteligentes, bem como a linguagem Solidity e plataforma Ethereum. Em seguida, também são descritos os modelos de Redes de Petri, ordinárias e sua extensão colorida.

3.1 Contratos Inteligentes

Um contrato é um conjunto de acordos realizado por dois indivíduos ou grupos, com a descoberta de novas tecnologias se tornou possível transformar o conceito de um contrato para um ambiente digital. Os Contratos Inteligente, inicialmente proposto por Nick Szabo [1], foram a primeira ideia nesse caminho de contratos virtuais, visando garantir que acordos pudessem ser executados de forma independente, sem o envolvimento de um agente certificador.

A ausência de um agente certificador é a principal diferença entre um contrato inteligente e contrato convencional. Uma máquina de refrigerantes, por exemplo, pode vista como um contrato inteligente onde o comprador deposita um dinheiro e a máquina entrega a bebida desejada. A máquina, neste caso, tem a a função de um agente certificador, garantindo que o comprador receba seu produto e o dinheiro seja entregue ao vendedor.

Um contrato inteligente é depositado numa blockchain, cuja função é garantir que os dados depositados em sua rede sejam imutáveis. Para que seja depositado numa blockchain, um contrato inteligente deve ser escrito em linguagem de programação, para então ser executado de acordo com as condições estabelecidas.

3.2 Blockchain

A *Blockchain* foi inicialmente proposta por Stuart Haber e W. Scott Stornetta [2], como um sistema de criptografia que armazenava os dados de forma que não pudessem ser alterados. A tecnologia se tornou mais conhecida em 2008 quando Satoshi Nakamoto desenvolveu o conceito que hoje conhecemos como *blockchain*. Em 2009 a tecnologia foi então usada para a criação de sistemas peer-to-peer [15], resultando na criação do bitcoin.

Uma *blockchain* funciona como um histórico onde ficam armazenadas as transações de um banco, através de diversos blocos e em cada bloco ficam registradas as transações aceitas pela rede. No bitcoin um bloco possui o *timestamp* da transação, a chave pública

do dono atual da criptomoeda e uma assinatura feita com a chave privada do usuário anterior. Com essa assinatura é possível verificar se a transação é válida usando a chave pública do bloco anterior e o valor *hash* do bloco. A hash de um bloco é composta a partir da hash do bloco anterior, os dados das transações armazenadas e um valor. Para a geração de um bloco considerado válido cada rede decide uma condição para o formato da hash. Como a geração da hash é dependente de um bloco anterior se torna muito difícil qualquer alteração no conteúdo do bloco sem tornar os blocos seguintes inválidos. Para alterar os dados de um bloco seria necessário calcular um valor hash válido para todos os blocos seguintes ao bloco alterado, garantindo assim a segurança na imutabilidade dos dados na blockchain.

Os blocos de uma blockchain devem então ser verificados pelos usuários da rede para que a garantia de segurança serja mantida. O bitcoin usa o método *proof-of-work*, onde os membros da rede precisam criar uma hash válida para um determinado bloco. Assim que uma primeira hash válida é criada, o usuário recebe uma recompensa a criptomoeda.

O Ethereum diferente do bitcoin utiliza o *proof-of-stake*, onde a hash não é calculada pelo usuário mais rápido, mas sim de forma aleatória. Caso o usuário selecionado aprove um bloco inválido as criptomoedas depositadas pelo usuário são tomadas. Usuários que depositam mais moedas na rede possuem mais chance de serem escolhidos para criar o bloco. A ideia é que o custo seja alto para àqueles que tentam burlar o sistema com blocos inválidos, pois se alguma trapaça é detectada com os usuários que possuem mais moedas, todas serão perdidas. Um dos problemas questionados no sistema *proof-of-stake* é o fato da rede recompensar com mais frequência usuário com mais criptomoedas gerando uma espécie de monopólio nas redes. Em geral, o sistema *proof-of-stake* apresenta mais riscos que o *proof-of-work*, porém continua sendo mais usado já que o sistema *proof-of-work* quando aplicado no bitcoin apresentam problemas de escalabilidade onde o consumo enérgico dos usuários competindo se tornam um problema.

3.3 Ethereum

A plataforma Ethereum [8] foi desenvolvida em 2012 por Vitalik Buterin com objetivo de dar mais liberdade para a escrita de contratos inteligentes, permitindo contratos mais complexos, não apenas para transações monetárias. Essa rede, inspirada no bitcoin, foi proposta para ser Turing completa, com uma linguagem de programação própria, possibilitando a escrita de contratos inteligentes com modelo de negócio mais complexos.

O funcionamento da rede Ethereum é baseada em entidades chamadas “accounts” ou contas que possuem 20 bytes de endereço e armazenam valores referentes: a quantidade de *ether*, a criptomoeda utilizada pelo *Ethereum*; ao *nonce*, um contador de transações

para garantir que uma transação seja processada apenas uma vez; ao *storageRoot*, que armazena o código hash da conta; e ao *code hash*, que armazena o valor do hash com um código de contrato. Essas contas podem ser divididas em contas externas e de contrato. As contas externas não possuem código associado, mas podem enviar mensagens criando contratos e assinando transações, normalmente são contas associadas a uma pessoa. Já as contas de contrato possuem código associado, ou seja, as linhas de código que representam o contrato inteligente, e quando recebem uma mensagem, executam seu código para então enviar uma mensagem ou criar novos contratos [8].

O Ethereum funciona em uma máquina virtual chamada Ethereum Virtual Machine (EVM). Nesse ambiente condições ou operações podem ser especificadas para que uma transação seja executada, tal como uma data para que uma transferência de recursos seja executada. O Ethereum cobra um valor (Wei ou ether) para executar uma transação ou toda vez que um contrato precisa realizar uma operação. Esses valores podem ser recarregados durante o tempo de vida do contrato. Porém, se não houver um valor suficiente para a execução de um comando antes do fim do contrato, todas as alterações realizadas são revertidas. De outra forma, caso reste algum valor após a execução do contrato, o valor é retornado para o criador do contrato. Após a execução do contrato, uma função é chamada para finalizá-lo, então as transações realizadas são salvas de forma permanente.

A rede Ethereum possui mais detalhes com relação a troca de dados comparado ao bitcoin. Existem dois métodos para realizar essas trocas, transações e mensagens. Uma transação é um pacote de dados enviado por uma conta externa que possui diversos dados tais como:

- Destinatário: O endereço de destino da mensagem, se for um conta externa será realizado um transferência e se for um conta de contrato será ativado o código da conta;
- Assinatura: A assinatura da conta que enviou a transação;
- Nonce: Valor que identifica a transição enviada pela conta;
- Valor: O valor de Ether enviado pela transição;
- Dados: Um campo opcional de dados;
- Limite de gás: O máximo de gás permitido a ser consumido pelo contrato;
- Taxa máxima: O valor máximo de gás a ser pago para validar a transação;
- Taxa máxima total: O valor máximo de gás a ser pago pela transação;

A mensagem por sua vez funciona de forma similar a uma transação, mas é apenas enviada por contas de contrato e existem apenas dentro da rede Ethereum, propiciando a interação entre contratos. Os dados de uma mensagem são:

- Remetente: A contra que enviou a mensagem, sendo implícito;
- Destinatário: A conta para onde a mensagem foi enviada;
- Valor: O valor de Ether enviado pela mensagem;
- Dados: Um campo opcional de dados;
- Limite de gás: O máximo de gás permitido ser consumido pelo contrato;

3.4 Solidity

A linguagem de programação Solidity [7] é uma das principais ferramentas usadas para se escrever contratos inteligentes na plataforma Ethereum. A Solidity é uma linguagem orientada a objetos, que foi fortemente influenciada por outras linguagens, como C++, Python e JavaScript. Isso permite que os programadores possam aproveitar as vantagens dessas linguagens e adaptá-las para o contexto dos contratos inteligentes. Além disso, a Solidity também oferece recursos avançados, como herança, sobrecarga de funções e modificadores, permitindo que os programadores escrevam contratos inteligentes mais complexos e flexíveis.

Na escrita de um código em Solidity é preciso, primeiramente, declarar a versão utilizada, como pode ser visto na Descrição 3.1. Em seguida, o contrato pode ser iniciado por um usuário ou então ser chamado por um outro contrato existente, quando seu construtor é executado. O contrato possui ainda as variáveis “owner”, uma variável com um tipo especial responsável por guardar o endereço do proprietário do contrato; e “ cupcakeBalances”, uma variável que armazena a quantidade de cupcakes que mapeia endereços distintos para cada chave.

```
1   pragma solidity 0.8.17;
2
3   contract VendigMachine{
4     address public owner;
5     mapping (address => uint) public cupcakeBalances;
6
7     constructor() public {
8       uint a = 1;
9       owner = msg.sender;
10      cupcakeBalances[address(this)] = 100;
11    }
12
```

```
13      function refill(uint amount) public {
14        require(msg.sender == owner, "Only the owner can refill.");
15        cupcakeBalances[address(this)] += ammount;
16      }
17
18      function purchase(uint amount) public {
19        require(msg.value >= amount  1 ether, "You must pay at least 1 ETH
              per cupcake");
20        require(cupcakeBalances[address(this)] >= amount, "Not enought
             cupcakes in stock to complete this purchase");
21        cupcakeBalances[address(this)] -= amount;
22        cupcakeBalances[msg.sender] += amount;
23      }
24    }
```

Listing 3.1 – Exemplo de código solidity

Já a variável “msg” é uma variável especial que não precisa ser declarada e permite que o usuário entre em contato com o contrato. Outros exemplos de variáveis globais são as unidades de ether wei, gwei e ether, com as seguintes equivalências:

$$1wei == 1$$

$$1gwei == 1e9$$

$$1ether == 1e18$$

O restante das variáveis, funções de bloco e transações do contrato são definidas como segue.

- block.basefee (uint): A taxa do bloco atual (EIP-3198 and EIP-1559).
- block.chainid (uint): O identificador da corrente atual.
- block.coinbase (address payable): O endereço do minerador do bloco atual.
- block.difficulty (uint): A dificuldade do bloco atual (EVM < Paris). Para outras versões do EVM atua como a função block.prevrandao (EIP-4399).
- block.gaslimit (uint): current block gaslimit.
- block.number (uint): current block number.
- block.prevrandao (uint): retorna um número aleatório (EVM >= Paris).
- block.timestamp (uint): O timestamp do bloco atual em segundos.
- gasleft() returns (uint256): gás restante.

Tabela 1 – Variáveis de Solidity

Nome	Palavra Chave	Descrição
Integers	int\uint	Inteiros podem ser do tipo “Unsigned” , sem valores decimais
Booleans	bool	Tipo de dado que pode ser definido apenas como verdadeiro ou falso
Addresses	address\address payable	Armazenam os endereços dos usuários na rede Ethereum
string	string	Guarda uma sequencia de códigos no formato UTF-8 formando palavras
Bytes	bytes	Armazena de forma dinamica bytes
Bytes32	bytesX	Armazena de forma fixa uma quantidade X de bytes, sendo X o valor na declação do tipo da variável
Enums	enum	Variáveis com valores definidos pelo usuário
struct	struct	Tipo de variável criada pelo usuário

- msg.data (bytes calldata): calldata completa.
- msg.sender (address): Endereço do enviador atual.
- msg.sig (bytes4): Os 4 primeiros bytes do calldata.
- msg.value (uint): A quantidade de wei enviado junto da mensagem.
- tx.gasprice (uint): O preço em gás da mensagem.
- tx.origin (address): O enviador da transação.

Vale ressaltar que a linguagem Solidity oferece ao programador diversos tipos de variáveis. A Tabela 1 apresenta essas variáveis, suas palavras-chave e respectivas descrições. As variáveis podem ainda ser classificadas em três tipos:

variáveis locais: definidas dentro do escopo de uma função onde os valores não são armazenados de forma permanente sendo apagados ao fim da execução da função;

variáveis globais: definidas para suportar valores envolvendo propriedades da blockchain e das transações;

variáveis de estado: definidas pelos usuários, mas diferente das variáveis locais são definidas fora do escopo das funções e armazenadas de forma permanente na blockchain junto ao contrato.

Na Descrição 3.1, linhas 4 e 5, as variáveis “owner” e “cupcakeBalances” são do tipo estado, enquanto que na linha 8 a variável “a” é local.

Outra classificação que variáveis e funções podem assumir, em relação a um contrato, são os níveis de visibilidade: externa, consideradas interfaces do contrato e não podem ser chamadas pelo contrato em questão; pública, quando não há restrições para que outras funções possam chamá-la; interna, quando apenas o contrato em questão pode acessar ou contratos derivados; e privada, quando nenhum outro contrato, exceto o próprio, pode acessá-la.

3.5 Redes de Petri

As Redes de Petri são modelos de estados, proposto inicialmente por Carl Adam Petri [9], para especificar sistemas paralelos, concorrentes, assíncronos e não-determinísticos. Uma Rede de Petri é, basicamente, formada por lugares, que representam os estados do sistema e transições, que representam eventos (ações). Os lugares são conectados às transições que por sua vez podem ser disparadas quando um evento ocorre. Essas conexões, chamadas de arcos, são direcionadas e podem ter pesos associados. Os recursos de uma Rede de Petri sã representados por círculos pretos, os marcadores, que possibilitam o disparo de uma transição, realizando a ação associada. Logo, uma transição habilitada é uma transição que possui seus lugares de entrada com marcadores suficientes para dispará-la.

O funcionamento das Redes de Petri depende de sua marcação inicial. Quando um lugar possui marcadores que habilitam as transições, esses recursos são consumidos do lugar de entrada e gerados nos respectivos lugares de saída, após o disparo dessas transições. Após o disparo, os recursos são consumidos, e as marcas são geradas nos lugares de saída conforme o peso associado ao arco. A ausência de pesos explícitos nos arcos indica, por convenção, o peso 1. A Definição 1 descreve formalmente uma Rede de Petri.

Definição 1 *Uma Rede de Petri é dada por $R = (P, T, A, O, K, C)$, onde*

- $P = \{\ p_1, p_2, \ldots, p_n\}$ *é um conjunto finito de lugares;*
- $T = \{t_1, t_2, \ldots t_q\}$ *é um conjunto finito de transições;*
- $A \subseteq (P \times T) \cup (T \times P)$ *é um conjunto finito de arcos;*
- $O : A \to \{1, 2, \ldots\}$ *é a função peso associada aos arcos;*
- $K : P \to \{0, 1, 2, \ldots\}$ *é a marcação inicial.*

A Figura 1 apresenta uma Rede de Petri que modela um sistema de máquina de doces. Os lugares representam os estados do sistema, como “Esperando moeda” e “Pronto para entrega”, enquanto as transições representam os eventos que ocorrem no sistema, como “Entrega doces” e “Rejeita moeda”. Os marcadores representam recursos e

condições necessárias para que as transições sejam disparadas, ou seja, doces e moedas. Observe que, inicialmente, apenas a transição “Insere moeda” está habilitada, pois existe um marcador no lugar “Esperando moeda” ligado a transição por um arco com peso 1. Quando a transição é disparada o marcador é então consumido, e um marcador é gerado no lugar “Segura moeda“. Este passo habilitará novas transições na rede.

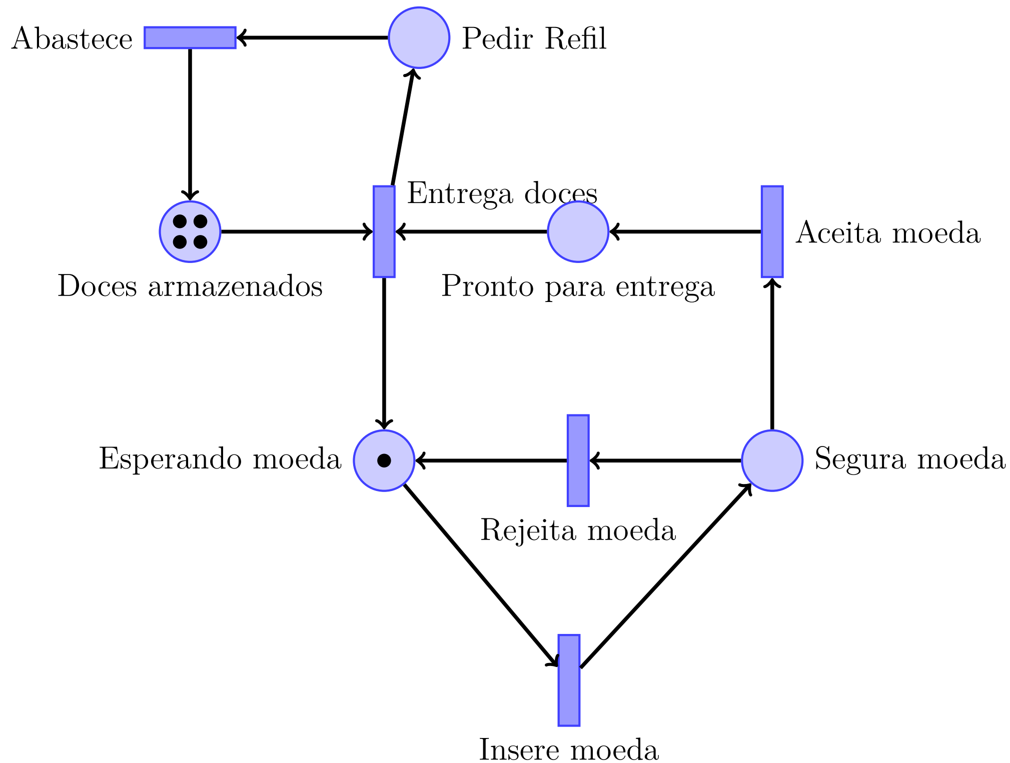

Figura 1 – Exemplo Rede de Petri

Uma Rede de Petri permite uma análise mais detalhada e um melhor entendimento de um sistema modelado, permitindo assim identificar pontos de falha ou otimizar o seu funcionamento. Note também que as Redes de Petri podem ser utilizadas para representar sistemas mais complexos. Nesse caso, uma rede mais complexa pode ser composta por redes mais básicas. As redes básicas que combinadas podem dar origem a redes mais complexas são definidas a seguir:

Sequenciamento: a rede começa em um lugar, que representa uma condição, seguido pela transição que representa uma ação, que quando disparada avança para um novo lugar. Veja a Figura 2.

Distribuição: similar a um sequenciamento, mas o disparo da transição, ou seja a realização da ação, dá origem a processos paralelos, ou seja avança para dois lugares de saída simultaneamente. Veja a Figura 3.

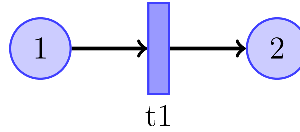

Figura 2 – Exemplo de Sequenciamento

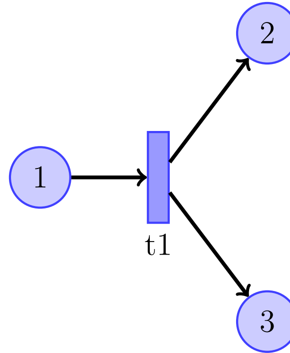

Figura 3 – Exemplo de Distribuição

Junção: processo inverso à distribuição, onde os lugares de entrada devem habilitar simultaneamente a transição, gerando recursos apenas no lugar de saída. Veja a Figura 4.

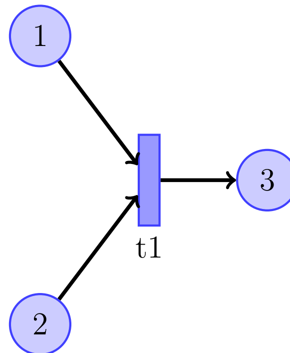

Figura 4 – Exemplo de Junção

Não-Determinismo : Um único lugar de entrada possui uma marcação para habilitar duas transições, porém suficiente para disparar apenas uma delas. Veja a Figura 5.

3.6 Redes de Petri Coloridas

As Redes de Petri Coloridas (RPC) [10, 11] são uma extensão das Redes de Petri ordinárias, que facilitam a representação de múltiplos recursos de um sistema mais complexo. A principal diferença entre redes ordinárias e uma RPC é a capacidade, da segunda, de atribuir valores através dos marcadores coloridos. As cores categorizam o tipo de valor atribuído, permitindo que os usuários identifiquem rapidamente o tipo de valor associado.

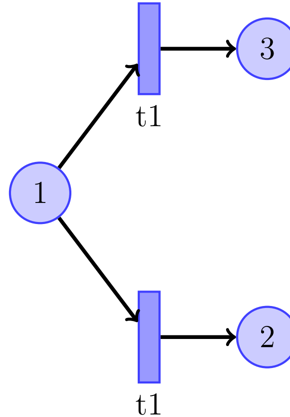

Figura 5 – Exemplo de Escolha Não-Determinística

A definição formal de Redes de Petri Coloridas é dada a seguir [10], mas antes precisamos de alguns conceitos importantes.

Primeiramente, precisamos da noção de *multiconjunto*. Um multiconjunto m sobre um conjunto não vazio M é definido por uma função $m : M \to \mathbb{N}$, e representada pelo somatório $\sum_{x \in M} m(x)'s$, onde $m(x)$ é o número de elementos x no multiconjunto. Por exemplo, se $M = \{x, y, z\}$ então $1`x + 4`y$ e $2`x + 2`y + 1`z$ são somatórios do multiconjunto sobre M. As operações de adição , subtração, multiplicação escalar e comparação também são definidas sobre multiconjuntos de um conjunto M. Seja m_1 e m_2 sobre M temos que:

- $m_1 + m_2 = \sum_{x \in M}(m_1(x) + m_2(x))'s$
- $m_1 - m_2 = \sum_{x \in M}(m_1(x) - m_2(x))'s$
- $nm_1 = \sum_{x \in M}(nm_1(x))'s$, com $n \in mathbbN$
- $m_1 \neq m_2 \iff \exists x \in M, m_1(x) \neq m_2(x)$
- $m_1 \leq m_2 \iff \forall x \in M, m_1(x) \leq m_2(x)$

Também precisamos definir o conjunto de variáveis de uma expressão. Seja $expr$, uma expressão lógica, o conjunto de variáveis de $expr$ é denotado por $Var(expr)$. Para uma expressão $expr : x = y + 1$ temos que $Var(expr) = x, y$. Além disso, a avaliação de uma expressão também é requerida quando valores associados as variáveis de uma expressão são considerados. O resultado de uma avaliação sobre uma expressão $expr$ associado a g é denotado por $expr(g)$. Assim, para o subconjunto $Var(expr)$ de g, a avaliação é obtida substituindo toda variável $v \in V(expr)$ pelo valor de $g(v) \in Type(v)$, onde $Type(v)$ é um conjunto de tipos, ou cores, da variável v.

A Definição 2 descreve formalmente o modelo de Redes de Petri Coloridas.

Definição 2 *Uma Rede de Petri Colorida é dada pela tupla* $N = (P, T, \sum, C, G, A, E, I)$*, onde*

- $P = \{\ p_1, p_2, \ldots, p_n\}$ *é um conjunto finito de lugares;*
- $T = \{t_1, t_2, \ldots t_q\}$ *é um conjunto finito de transições;*
- $\sum$ *é um conjunto finito de tipos não nulos chamados de conjuntos de cores;*
- $C : P \rightarrow \sum$ *é uma função de cor de* P *para* $\sum$*, onde* $C(p)$ *é a cor de* p *tal que* $C(p) \subseteq \sum$ *e* $\cup_{p \in P} C(p) = \sum$*;*
- $G : T \rightarrow expr$ *é uma função de guarda de* T *para expressões tal que* $\forall t \in T$*,* $G(t)$ *é uma expressão booleana e* $Type(Var(G(t))) \subseteq \sum$*, onde* $Var(expr)$ *é o conjunto de variáveis de* $expr$ *e* $Type(v)$ *é o conjunto de cores de* v*;*
- $A \subseteq (P \times T) \cup (T \times P)$ *é um conjunto finito de arcos;*
- $E : A \rightarrow expr$ *é função de expressão de arco de* A *para* $expr$ *tal que* $\forall a \in A$ *temos* $Type(Var(E(a))) = C(p)_{MS}$*, onde* p *é um lugar adjacente ao arco* a *e* $C(p)_{MS}$ *indica todos os multiconjuntos de* $C(p)$*, e* $Type(Var(E(a))) \subseteq \sum$*, indicando que todas as variáveis na expressão assumem valores do conjunto de cores.*
- $I : P \rightarrow expr_{closed}$ *é a função que gera a marcação inicial definida de* P *sobre as expressões fechadas, ou seja, sem variáveis, tal que* $\forall p \in P : Type(I(p)) = C(p)_{MS}$*.*

A marcação de uma CPN é dada pela função M sobre P tal que $p \in P$ temos que $C(p) \rightarrow \mathbb{N}$. Logo, a marcação $M(p)$ representa a soma das cores de p definido formalmente como

$$M(p) = \sum_{i=i}^{u} n_i' c_i,$$

onde n_i é o número de marcas da cor c_i no lugar p, ou seja, $M(p)(c_i) = n_i$, e u é o tamanho do conjunto de cores de p, onde $u = |C(p)|$.

A Figura 9 ilustra uma RPC que modela um protocolo de rede. O conjunto dos lugares e transições, P e T, são representados por círculos e retângulos, respectivamente. Já o conjunto $\sum$ representa as cores que os marcadores podem assumir, ou seja, os tipos de dados NO definido pelos naturais ($\mathbb{N}$), e $DATA$ definido como *string*. O produto cartesiano desses dois conjuntos, $NOxDATA$, define um terceiro tipo, que pertence ao conjunto $\sum$.

A função C que relaciona o conjunto de cores $\sum$ com o conjunto P define as cores que podem ocorrer nos respectivos lugares. Por exemplo, o lugar “D” tem associado a cor NO. Já a função G define os guardas através do mapeamento de cada transição em expressões lógicas. No exemplo, assuma que t é a transição “Envio de Pacotes”, então

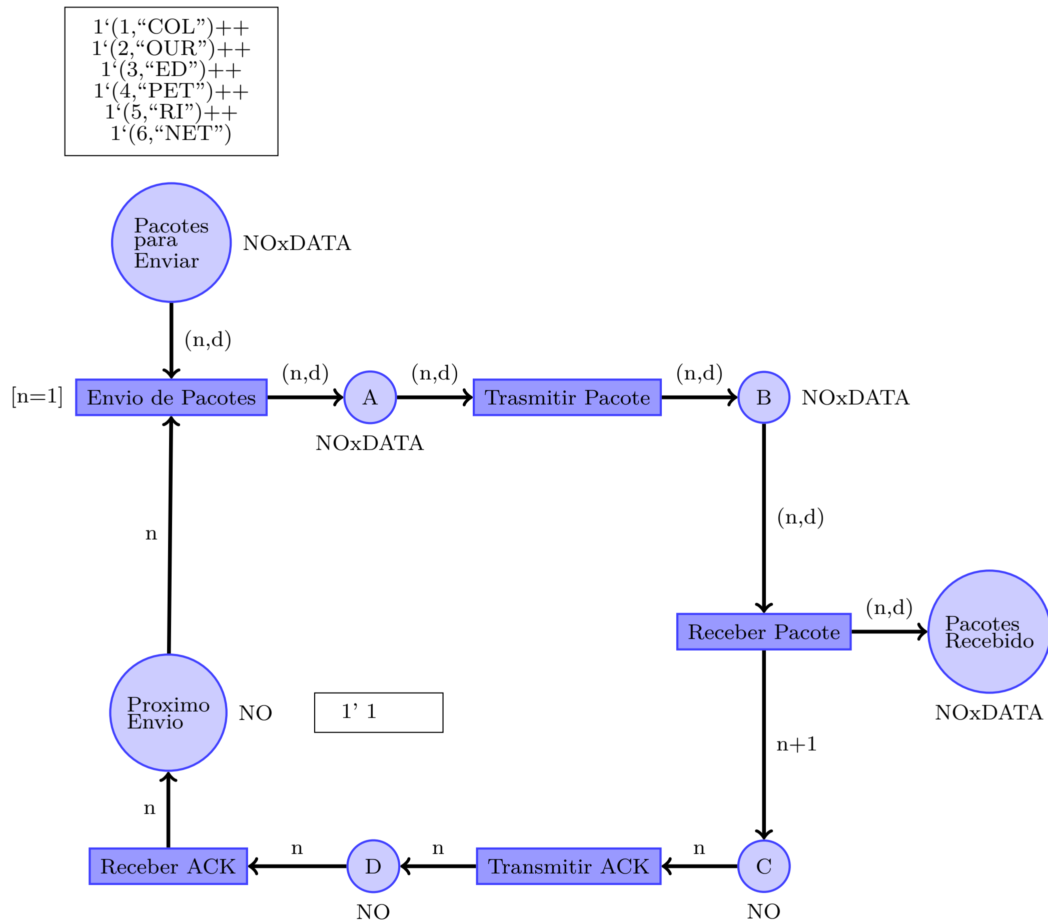

Figura 6 – Exemplo Rede de Petri Colorida

$G(t)$ é o guarda com a expressão $n = 1$ que deve ser satisfeita para que a transição seja disparada. O conjunto de variáveis $Var(G(t))$, nesse caso, é composto apenas por n e, por isso, $Type(Var(G(t)))$ retornaria NO, o tipo da variável n. O conjunto de arcos, conectando lugares e transições, é dado por A, enquanto E denota a função que mapeia as expressões lógicas dos arcos. Na figura temos a expressão (n, d) associada ao arco que liga a transição $EnviodePacotes$ ao lugar A. Nesse caso, $Type(Var(E(a)))$ retorna a cor $NOxDATA$, o produto entre os conjuntos NO e $DATA$, onde $Type(Var(E(a))) \subseteq \sum$ tal que $Type(Var(E(a))) = C(p)_{MS}$.

A marcação inicial é dada pela função I, onde os marcadores devem satisfazer a condição $\forall p \in P : Type(I(p)) = C(p)_{MS}$, ou seja, os tipos dos marcadores devem ser os mesmos de $C(p)$. Observe que na Figura 9 que no lugar $PacotesparaEnviar$ possui seis marcadores com a cor (tipo) $NOxDATA$.

O disparo numa RP tradicional é realizado verificando quais são os lugares de entrada através dos arcos que os conectam com a transição, o número de marcadores em cada lugar e o peso de cada arco. Se o número de marcadores em cada lugar de entrada da transição é maior do que o peso nos arcos que os conectam a transição, dizemos que a

transição está habilitada. Quando o disparo efetivamente ocorre, o número de marcadores é diminuído de cada lugar de entrada conforme o peso dos respectivos arcos, e um número de marcadores é gerado nos lugares de saída, também conforme os pesos associados em cada arco.

Os disparos de uma RPC ocorrem de forma diferente, levando-se em consideração as expressões dos arcos. Essas expressões, que podem ser compostas por variáveis de tipos distintos e valores diferentes como em linguagens de programação, definem como os marcadores devem satisfazer o fluxo da rede. Para disparar uma transição os marcadores nos lugares de entrada da transição devem ser selecionados, com valores específicos para que sejam consumidos de acordo com as restrições impostas na expressão de arco. Para o disparo da transição “Envio de Pacotes”, os lugares de entrada “Pacotes para Enviar” possui 6 marcadores com a cor $NOxDATA$ com valores distintos,

$$(1, COL)$$

$$(2, OUR)$$

$$(3, ED)$$

$$(4, PET)$$

$$(5, RI)$$

$$(6, NET)$$

e o lugar “Próximo Envio” possui um marcador 1. Como há marcadores suficientes nos lugares de entrada dizemos que a transição está habilitada. O arco entre o lugar “Próximo Envio” e a transição “Envio de Pacotes” possui a expressão de arco com a variável n. Logo, para que o disparo ocorra, o valor de um marcador precisa ser associado a variável do arco. Como o lugar “Próximo Envio” possui apenas um marcador então seu valor é associado a n com $< n = 1 >$. No caso do lugar “Pacotes para Enviar” existem 6 marcadores para a associação com as variáveis n e d da expressão de arco.

$$< n = 1, d = COL >$$

$$< n = 2, d = OUR >$$

$$< n = 3, d = ED >$$

$$< n = 4, d = PET >$$

$$< n = 5, d = RI >$$

$$< n = 6, d = NET >$$

Porém, como o lugar “Próximo Envio” teve a variável n associada ao valor 1, então para que o disparo ocorra o marcador de “Pacotes para Enviar” também deve ter o valor 1 associado a variável n. Nesse caso, a única opção é a associação $(1, COL)$ com valor 1 para n e o valor COL para a variável d.

3.6.1 Extensão para RPCs

Da mesma forma que as RPCs surgiram para suprir limitações das Redes de Petri tradicionais, extensões para RPCs também foram propostas. Algumas dessas extensões incorporam alguns aspectos importantes que permitem uma RPC simular a execução de um contrato inteligente [16]: Contratos de transição (Transition contracts) [17, 18]; Places abertos (Open places) [19]; Places oráculos (Oracle places) e Transições temporizadas (Timeout transitions) [20]. Essas extensões são descritas a seguir e ilustradas pela Figura 7:

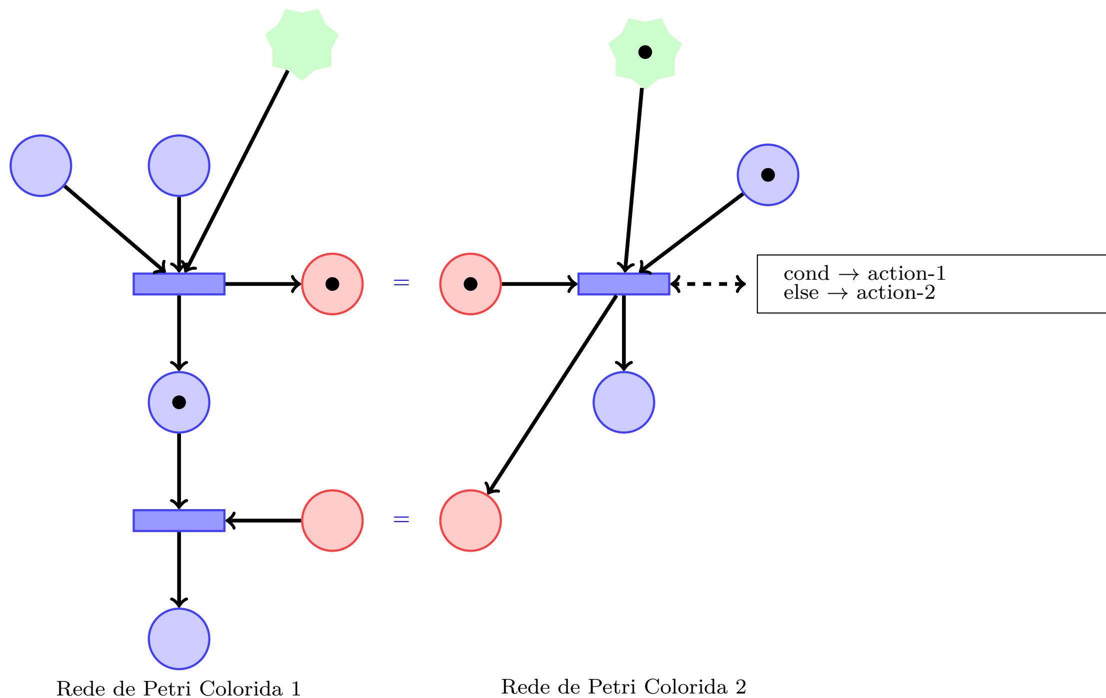

Figura 7 – Exemplo de Rede de Petri Colorida com Extensões

Lugar aberto: um lugar aberto é representado por um círculo pontilhado vermelho que permite conectar duas Redes de Petri. Entre dois lugares abertos os marcadores podem ser compartilhados para facilitar a visualização da rede. Com o uso desse lugares especial se torna possível adicionar para uma Rede de Petri novos marcadores de fora da rede, funcionando similar com um contrato que pode ter dados inseridos por usuário de formas externa em um contrato inteligente.

Lugar oracle: é outro lugar especial, representado por uma estrela verde, que pode receber marcadores de fontes externas à rede. Dessa forma, novas informações podem ser inseridas através de marcadores na RPC, permitindo que a rede crie novos marcadores sem o disparo de transições.

Contratos de Transição: é um tipo especial de transição, onde podem ser adicionadas condições ou regras [21], que restringem o disparo da transição e guiam os resultados

da mesma. Essas transições permitem simular decisões baseadas em condições que assim como os comandos existentes em linguagens como Solidity.

Transições de timeout: é uma transição que gera um marcador de *timeout*, ou seja, quando a transição é habilitada um contador é iniciado. Caso o disparo não seja realizado de acordo com a restrição de tempo, a transição gera um marcador com a informação de *timeout* na saída. Logo, um sinal de erro é enviado em cenários onde a rede permanece estática por não existirem transições habilitadas. Esse lugar permite que assim como um contrato inteligente ou linguagens de programação no geral, uma mensagem de timeout possa ser emitida como sinal de erro quando a Rede de Petri ficar travada.

4 DE CONTRATOS PARA REDES DE PETRI

Este capítulo descreve o processo para se obter modelos de redes de Petri coloridas a partir de contratos inteligentes. A sistematização dessa transformação tem como objetivo garantir as características desses contratos quando especificados por modelos mais precisos tais como as redes de Petri. Nas próximas seções são apresentadas, primeiramente, uma arquitetura da transformação descrevendo aspectos relacionados aos artefatos de entrada e saída, em seguida o processo de transformação em detalhes, e por fim um exemplo para ilustrar a aplicação do processo.

4.1 Arquitetura

A arquitetura para o processo de transformação é apresentada na Figura 8. A entrada do processo de transformação deve ser um contrato inteligente escrito em Solidity e a saída uma rede de Petri colorida conforme Definição 2.

Como a entrada do processo é um arquivo texto em linguagem Solidity é necessário um parser para realizar a tradução dos comandos do contrato para os respectivos componentes em RPC, bem como a leitura de cada expressão e variáveis envolvidas no contrato. Vale ressaltar que o processo proposto está restrito a versão Solidity 0.8.4, devido aos comandos específicos dessa versão. Porém, a proposta é extensível para englobar qualquer versão, desde que novos comandos adicionados e alguma variação nas estruturas das expressões sejam consideradas no parser de entrada.

Após o processo de transformação, uma Rede de Petri Colorida deve ser obtida no formato de um arquivo de texto, conforme sua definição formal.

4.2 Processo de Transformação

O processo de transformação dos contratos inteligentes em modelos de redes de Petri coloridas é composto de três etapas. As duas primeiras etapas da transformação consistem de dois analisadores que realizam a leitura do contrato para obtenção dos identificadores de interesse. A primeira análise leva em consideração as funções e variáveis

Figura 8 – Arquitetura do processo de transformação

globais do contrato. Já na segunda análise o objetivo é obter a lógica implementada em cada função do contrato, tais como comandos, operações aritméticas, expressões condicionais e chamadas de subfunções. A última etapa do processo constrói efetivamente a RPC, gerando os elementos da rede, com base nos dados obtidos nas etapas anteriores. O resultado do processo deve ser um arquivo texto descrevendo a RPC conforme a Definição 2.

Primeira etapa

A primeira análise sobre o contrato tem como objetivo identificar as *variáveis globais* e funções do contrato. Para cada elemento obtido dessa busca um registro é gerado contendo as informações relevantes e gravados em tabelas. As variáveis globais são armazenadas na tabela "variaveisGlobais", onde cada registro possui os seguintes campos N, T, V, e TI. O campo N se refere ao nome da variável, T é o seu tipo, V o seu valor e TI se refere ao tipo do seu índice, caso exista.

Existem três modos de declarar variáveis que devem ser identificados durante essa leitura. A primeira define seu tipo, seguido pelo nível de visibilidade, e termina com o nome da variável. A visibilidade, como visto na Seção 3.4, é dividida em quatro, definindo os acessos às variáveis ou funções, com a visibilidade pública definida por padrão. Uma variável pode ter um valor definido na declaração, quando usado o símbolo de igualdade. Caso não exista um valor, V deve assumir valor nulo. O valor para TI também deve ser nulo nesse tipo de declaração.

O segundo modo de se declarar uma variável global é através de um array. Uma variável array A é declarada por $A[n]$, onde n é o tamanho do array. O campo TI nesse caso é definido como uint, o tipo para inteiros positivos em Solidity.

O último modo de declarar a variável é um mapeamento, definido por $mapping(A => B)\ C\ D$. A variável A é o tipo do índice que define o campo de TI, B é o tipo da variável que define T, C a visibilidade e D é o nome que define N.

Ainda na primeira leitura devem ser identificadas as funções. Uma função pode ser encontrada pela palavra-chave "function". Cada função encontrada será descrita em uma tabela chamada "funcoes". A tabela deve gerar um registro para cada função com os campos N, T, V, P, I e F. N armazena o nome da função, T o seu tipo de retorno, V o seu nível de visibilidade, P os tipo e nomes dos parâmetros da função,I e F a linha de início e fim da função. Considere o exemplo de uma função chamada $getResult$ que é declarada do seguinte modo:

```
1  function getResult() public view returns(uint product, uint sum){
2    uint a = 1;
3    uint b = 2;
4    require(a < 10 && b < 5){
```

```
5           product = a * b;
6           sum = a + b;
7         }
8     }
```

Listing 4.1 – Exemplo de função

Essa declaração deve formar um registro com os seguintes campos: N sendo $getResult$, T com o valor, *uint product*, *uint sum*, V deve ser *public*, P deve ser nulo pois a função não possuir parâmetros, I deve ser 1 e F deve ser 8. Existe um tipo de função chamada construtora, que usa a palavra-chave "construtor" e possui apenas o campo de parâmetros. Para essa função os campos I e F devem ser preenchidos normalmente, o campo do nome N deve ser construtor, V deve ser os parâmetros da função e os outros campos devem ser nulos.

Segunda etapa

Nessa fase, será lida apenas as linhas que pertencem ao escopo de uma função, sabemos quais linhas devem ser lidas pelos registros feitos na tabela de "funcoes". Ao ler essas linhas devem ser gerados registros para cada operação, condicional e chamada de função encontrada. Para cada elementos encontrado será formado um registro, que deve conter a linha onde o elemento foi encontrado, com isso será possível identificar a função que o elemento pertence relacionando o escopo das funções. Assim como no passo anterior cada elemento deve ser identificado durante a leitura e será armazenado em uma tabela.

A tabela usada para guardar as operações será chamado "operacoes". Deverá ser gerado um registro para cada operação com os seguintes campos, V, O e P O valor do campo V é o nome da variável que recebe o resultado da operação, O recebe toda a operação e P recebe o nome de todas as variáveis na operação. Ainda usando o exemplo 4.1 na linha 2 a variável a é declarada com um valor de 1. O registro gerado terá V como a, O como 1 e P como nulo, pois a operação não possui variáveis. Agora usando a linha 5 o registro gerado teria V como *product*, O como $a * b$ e P como a b.

A tabela usada para guardar os registros das chamadas terá o nome de "chamadas". O formato do registro deve ser N, P e L. N é o nome da chamada e P são as varáveis usadas na chamada da função e L é a linha da chamada. Uma chamada $increaseBalance(receiver, amount)$;, teria o registro gerado N sendo $increaseBalance$ e P sendo *receiver amount*.

A tabela na qual serão guardados os registros das condicionais deve ter o nome "condicionais" e cada registro deve ter os campos T, C, I e F. T representa o tipo de condicional, C guarda a condição, uma condição pode ser composta de várias expressões, todas devem ser armazenadas nesse campo, I representa a primeira linha do comando e F representa a última linha do comando. Usando o exemplo 4.1, considere a linha 4,

nela temos o comando condicional *require*, convertendo-o para um registro, obteremos os campos T sendo *require*, C sendo $a < 10$ && $b < 5$, I sendo 4 e F como 7.

Terceira etapa

Na última fase será gerado a RPC em um documento de texto com os elementos como descritos na definição da definição. Os elementos serão criados baseados nos registros feitos nas outras duas etapas.

O primeiro processo deste passo será gerar os lugares com base nos registros de variáveis globais e funções. A primeira tabela a ser lida será a "variaveisGlobais". Cada lugar necessita de um nome, uma cor e os seus marcadores iniciais. Dos registros gerados da tabela "variaveisGlobais", o campo N será o nome do lugar. As cores são formadas por multiconjuntos e devem ser definidas conforme os lugares são criados. O campo T possui os multiconjuntos(tipos) das variáveis globais. Cada multiconjunto deve possuir uma cor que funcione como uma espécie de nome, caso um multiconjunto ainda não possua uma cor, o multiconjunto deve receber uma nova, essa relação de conjunto com cor deve ser armazenada em memória e registrada no arquivo da RPC ao final da execução. Se o registro possuir um ou mais valores em V eles serão seus marcadores iniciais. Se ele possuir um valor TI, esse será um conjunto que deve ser adicionado ao multiconjunto desse lugar, esse conjunto servirá para armazenar um valor que funcionará como um índice de um array, permitindo localizar um marcador específico.

Após ler a tabela "variaveisGlobais" deve ser lida a tabela "funcoes". Os registros dessa tabela também devem gerar lugares. No caso, o nome dos lugares gerados será o prefixo par- seguido pelo campo N do registro. Esse lugar deve ser um oráculo caso V seja pública ou externa, em outros casos o lugar é normal. Um lugar oráculo é um dos lugares modificados introduzidos no capítulo sobre RPCs servindo como um lugar onde novos marcadores podem ser gerados com dados externos. A visibilidade define se um lugar deve ser oráculo, pois funções públicas e externas podem ser chamadas fora do contrato, então seus parâmetros possuem informações externas. Os lugares gerados nesse passo terão uma cor que representa o multiconjunto dos tipos presente no campo P.

A função construtora, se encontrada na tabela "funcoes" e deve possuir um lugar como entrada mesmo se seus parâmetros estiverem vazios. Caso o parâmetro seja vazio, o marcador será do tipo booleano com o valor "true". Isso se deve pelo construtor ser uma função disparada apenas uma vez durante a inicialização do contrato, seu lugar deve receber apenas os marcadores no início da RPC, não servindo como um oráculo.

Após ler a tabela "funções" deve ser lida a tabela "operacoes" também ler a tabela "condicionais". Para a primeira tabela será buscado no registro as informações do campo P, que contém o nome das variáveis usadas nas operações. Já a segunda tabela terá o campo C buscado, pois ele armazena as expressões das condicionais, e nelas podem ser

encontrados os nomes das variáveis usadas. Os nomes buscados serão os das variáveis globalmente disponíveis, sendo as variáveis listadas no capítulo sobre Solidity que podem ser chamadas em qualquer contrato. Essas variáveis devem ter um lugar criado, mesmo não existindo um registro para elas, pois não são declaradas no contrato. Portanto, a transformação deve identificar cada uma individualmente e tratá-la de acordo. Assim como as variáveis globais, a cor do lugar dependerá do seu tipo, deve ser uma variável oráculo, pois seus valores variam entre execuções, dependendo de aspectos externos ao contrato, e o nome será o nome da própria variável.

O segundo processo deste passo é gerar as transições, para isso será lido a tabela “funcoes”. Cada registro da tabela será usado para criar uma transição com o nome definido por N.

O terceiro processo deste passo é formar os arcos, como está escrito na definição de Redes de Petri Coloridas cada arco deve conectar um lugar e uma transição, nunca dois lugares ou duas transições. Como para a RPC apenas as funções geram transições, todos os lugares serão relacionados com uma função.

Os primeiros arcos serão criados entre os parâmetros e suas funções na RPC. Cada parâmetro deve ser conectar por apenas um arco uma única função e vice e versa. Esses arcos têm a função de conectar os marcadores que servem como valor dos parâmetros com as funções.

Após gerar os arcos que conectam parâmetros, devem ser representados os arcos que conectam variáveis globais com funções. As primeiras variáveis globais tratadas devem ser as variáveis globalmente disponíveis. A transição deve ser conectada com o lugar por um arco entrando na transição, caso a função que representa essa transição utilize a variável. Será analisada as tabela de operações e condicionais para verificar se uma variável global foi chamada.

Após tratar as variáveis globalmente disponíveis serão tratadas as variáveis globais comuns. Assim como para as variáveis globalmente disponíveis, os arcos serão criados entre as transições que representam funções que usam as variáveis globais e os lugares com o mesmo nome das variáveis usadas. Quando o uso de uma variável global for detectado deve ser gerado dois arcos, um entrando e outro saindo da transição. O arco entrando representa o uso da variável global no contrato. O arco saindo da transição tem a função de retornar o marcador consumido durante o disparo da transição, o valor retornado deve refletir o valor da variável global ao final da função.

Os últimos arcos a serem criados representarão o fluxo de execução das funções. A RPC deve ser capaz de simular situações em que uma função chama outra internamente. Uma função na RPC gerada por esse algoritmo é o equivalente de uma transição. Sabemos que uma transição não pode se conectar com outra transição, então, para isso ser possível

devemos conectar as duas transições por um lugar.

Para uma chamada ser representada deverá ser gerado um arco entre a função principal com o parâmetro da função chamada. O arco deve ir da transição para o lugar, simbolizando o envio dos parâmetros durante uma chamada. Desse modo é possível por meio desses arcos indicar como os parâmetros são passados.

Com os arcos finalizados devem ser geradas as expressões de arco. Cada arco deve possuir um conjunto de variáveis. Todo conjunto que forma o multiconjunto(cor) do lugar deve ter uma variável nos arcos conectados a ela. As variáveis em arcos conectados com a mesma transição possuem o mesmo valor. Quando uma variável é associada a um arco que vai para uma transição, alguma variável da função de mesmo nome da transição conectada é associada com essa variável de arco. Primeiro deve ser gerada as expressões para os arcos que saem dos lugares que representam os parâmetros. As variáveis de arco não devem repetir com nenhuma outra em um arco de entrada com a mesma transição. A junção dos tipos de todas as variáveis deve formar o multiconjunto do lugar que elas se conectam. A variável gerada pelo construtor quando seu parâmetro é nulo, é uma de duas variáveis, que não são associadas a nenhuma variável do contrato.

Logo em seguida os arcos de saída de lugares que representam variáveis globais devem ter suas expressões geradas. As suas variáveis não devem repetir com nenhuma outra, em um arco de entrada com a mesma transição. Os lugares formados por arrays ou mapeamentos devem ter uma variável representando o valor do índice e outra o seu valor. Caso o índice seja uma constante, o valor será usado no arco. Se uma variável associada com alguma variável de arco seja usada, então essa variável de arco deve ser repetida no novo arco. Caso essa variável de arco passe por operações e tenha seu valor diferente do primeiro arco, as operações deverão ser repetidas na expressão de arco.

Agora precisamos processar a representação dos arcos entrando no lugares. O método para representar parâmetros e variáveis globais é o mesmo, mas para fins de clareza, trataremos primeiro dos parâmetros. A expressão arco deve ter a mesma variável de arco atribuída à variável global nesta transição, mas representa as manipulações e alterações feitas no valor da variável global dentro da função. Se o arco for para um parâmetro e a chamada de função para esse parâmetro estiver dentro de um loop, vários marcadores devem ser gerados. Se uma variável global recebe uma constante, você só pode usar o valor atribuído.

Por exemplo imagine uma função qualquer, e dentro dessa função é realizada uma operação com uma variável global de valor um. O arco de entrada deve possuir uma variável com o valor um. Vamos dar o nome para essa variável de arco o nome “b”. A operação com a variável global da função soma um ao valor da variável global. O arco de entrada do lugar dessa variável global deve ser a operação $b + 1$. A variável de arco “b” é usada, pois ela afeta o resultado da variável global. Caso a operação fosse uma atribuição,

o arco de saída teria as operações que resultaram no valor atribuído.

4.3 Aplicação prática da transformação

O objetivo aqui é mostrar a aplicação prática da transformação usando um exemplo de contrato inteligente em Solidity para se obter o respectivo modelo de RPC. O contrato é descrito no Código 4.2 e simulam saldos de uma moeda fictícia.

O contrato *Coin* mapeia um saldo usando os endereços das contas como saldo. Ele permite que o seu criador, o usuário que iniciou o contrato, possa aumentar o saldo dos usuários. Qualquer conta pode acessá-lo e enviar um valor do saldo da própria conta para outro usuário.

O contrato tem como objetivo simular uma moeda fictícia, primeiro salvando o endereço do usuário que inicia o contrato. É criada a variável *balances* que mapeia um array de *uint* para os endereços de contas dos usuários do Ethereum. *Uint* sendo o tipo para inteiros positivos em Solidity. O construtor chamado durante o início do contrato atribui para *minter* o endereço do usuário que iniciou o contrato.

O contrato possui 3 funções, *mint* é a função responsável por colocar mais moedas nos saldos dos usuários do contrato. Possui 2 parâmetros *receiver* e *amount* que definem o endereço do usuário que deve ser saldo aumentado e a quantidade do aumento. A função verifica o endereço do usuário que chamou a função com a variável globalmente disponível "msg.sender" e compara o resultado com o endereço do usuário salvo em *minter*. Caso os endereços sejam iguais o saldo da conta com endereço *receiver* será aumentado pelo valor de balances.

A segunda função *send* possui dois parâmetros assim como a função *mint*. A função realiza o envio do valor *amount* da conta do usuário que chama a função para a conta do endereço *receiver*. Para isso primeiro a função faz uma verificação se o usuário que chamou a função possui saldo o suficiente para a transferência, caso o resultado seja negativo a função falha e as alterações são anuladas. Caso a verificação seja positiva o valor é somado ao saldo da conta em *receiver* e subtraído da conta do chamador da função.

A última função *increaseBalance* é privada, ou seja, não é possível chamá-la. Ela realiza o aumento nas contas, sendo chamadas por *mint* e *send* para realizar essas alterações. Os parâmetros da função *receiver* e *amount* definem assim como em *mint* o endereço da conta que deve ter seu saldo aumentado em *amount*.

```
1  pragma solidity ^0.8.4;
2
3  contract Coin {
4    address public minter;
```

```
5     mapping(address => uint) public balances;
6 
7     event Sent(address from, address to, uint amount);
8 
9     constructor() {
10      minter = msg.sender;
11    }
12 
13    function mint(address receiver, uint amount) public {
14      require(msg.sender == minter);
15      increaseBalance(receiver,amount);
16    }
17 
18    error InsufficientBalance(uint requested, uint available);
19 
20    function send(address receiver, uint amount) public {
21      if (amount > balances[msg.sender])
22      revert InsufficientBalance({
23        requested: amount,
24        available: balances[msg.sender]
25      });
26 
27      balances[msg.sender] -= amount;
28      increaseBalance(receiver,amount);
29      emit Sent(msg.sender, receiver, amount);
30    }
31 
32    function increaseBalance(address receiver,uint amount) private {
33      balances[receiver] += amount;
34    }
35  }
```

Listing 4.2 – Exemplo de código solidity

Começando a transformação devemos gerar os registros de cada tabela para o contrato. O primeiro passo é gerar os registros das funções e variáveis globais. Todos os elementos em cada passo são analisados aos mesmo tempo, mas por clareza na explicação iremos passar um elemento por vez começando com as variáveis globais. O registro das variáveis globais tem os campos $N\ T\ V\ TI$ e o contrato possui duas variáveis globais, *minter* e *balances*, então deve ser gerado dois registros.

$$N = minter;\ T = address;\ V = NULO;\ TI = NULO$$

$$N = balances;\ T = uint;\ V = NULO;\ TI = address$$

O registro das funções possui os campos $N\ T\ V\ P\ I\ F$. O contrato possui três

funções, *sent*, *mint* e *increaseBalance* e um construtor, então devem ser gerados 4 registros.

$$N = construtor;\ T = NULO;\ V = NULO, P = NULO;\ I = 9;\ F = 11$$

$$N = mint;\ T = NULO, V = public;\ P = address,\ uint;\ I = 13;\ F = 16$$

$$N = send;\ T = NULO, V = public;\ P = address,\ uint;\ I = 20;\ F = 30$$

$$N = increaseBalance;\ T = NULO;\ V = private;\ P = addressuint;\ I = 32;\ F = 35$$

Com o primeiro passo terminado devemos gerar os registros do segundo passo começando com as operações. Como sabemos onde cada função inicia e finaliza com os registros do último passo, deve ser analisado apenas dentro do escopo das funções por operações. A função construtor possui uma operação na linha 10 atribuindo o valor da variável globalmente disponível *msg.sender* para a variável global *minter*. Deve ser gerado um registro dessa operação com os seguintes campos:

$$V = minter;\ O = msg.sender;\ P = minter;\ msg.sender$$

A função *mint* não possui operações, já a função *send* possui uma. Na linha 27 a função *send* subtrai o valor do mapeamento balances de índice *msg.sender* no valor de *amount*. O operador $- =$ simboliza subtração e atribuição então o campo O que representa toda a operação deve representar essa característica. O registro gerado dessa operação terá os campos:

$$V = balances[msg.sender];\ O = balances[msg.sender] - amount;$$

$$P = balances[msg.sender];\ amount$$

A última função *increaseBalance* possui uma operação. Na linha 33 é somado o valor de *amount* para a variável global *balances* com índice *receiver*. Aqui a operação possui o sinal $+ =$ que realiza uma soma e atribuição então deve ser representado no campo O essa subtração. O registro gerado dessa operação terá os campos:

$$V = balances[receiver];\ O = balances[receiver]{+}amount;\ P = balances[receiver],\ amount$$

Agora será explicado como deve ser gerado os registros das chamadas. O registro possui os campos $N\ P\ L$. Vamos observar dentro das funções, o construtor não realiza nenhuma chamada. $Mint$ realiza uma chamada na linha 15 da função *increaseBalance*. Para o registro dessa chamada os campos devem ser:

$$N = increaseBalance;\ P = receiver,\ amount\ \ L = 15$$

Observando a função *send* na linha 28, ela também realiza a chamada da função *increaseBalance*. Então deve ser gerado um registro com os campos preenchidos da seguinte forma:

$$N = increaseBalance;\ P = receiver,\ amount\ \ L = 29$$

A última função a verificar é a própria *increaseBalance*. Como nenhuma chamada de função é exibida, a verificação da chamada está concluída. Veja como os registros da tabela das condicionais são geradas. As funções construtoras não possuem nenhuma condicional. A função *mint*, por outro lado, usa o comando *require*. Este comando requer que a função saia se a condição expressa no parâmetro for falsa. Como este comando verifica uma condição para continuar a execução da função, precisamos gerar um registro para esta instância do comando. Os campos do cadastro devem ser preenchidos da seguinte forma:

$$T = require;\ C = msg.sender == minter;\ I = 14;\ F = 16$$

Nesse registro, o campo F é 16, devido ao comando para a execução de tudo após ele.

Na linha 21 da função *send* existe um comando if que verifica a expressão do parâmetro e, se verdadeiro, executa a linha dentro daquele escopo. Essa verificação no contrato garante que o valor enviado pela função *send* seja menor que o valor que o usuário que chama a função possui na variável balances. Deve ser criado um registro para este comando. O registro deve preencher os campos da seguinte forma:

$$T = if;\ C = amount > balances[msg.sender];\ I = 21;\ F = 25$$

A função *increaseBalance* não possui um comando condicional, então não será gerado um registro.

Depois de verificar a última função, a segunda etapa do processo de transformação é concluída. Agora precisamos passar para o terceiro passo, no qual descreveremos os elementos da RPC em um documento de texto.

Conforme explicado na sistematização, cada variável global deve ser transformada em um lugar. Portanto, há um lugar para *minter* e um lugar para *balances*. Isso ocorre porque ambos têm entradas na tabela “variaveisGlobais”. A cor desses lugares deve corresponder ao tipo do campo T no registro, portanto *minter* deve ser da cor que contém o endereço. Ainda não definimos nenhuma cor, então vamos criar uma. Por exemplo, azul representa endereços.

$$Azul = address$$

balances possui o campo T como *uint*, mas como o campo TI possui o valor endereço, o tipo de *balances* deve ser o produto do conjunto dos *uint* com o dos endereços. Como não definimos a cor para *uint* essa deve ser definida:

$$Verde = uint$$

A cor para *uint* foi definida, então é possível definir a cor para *balances*:

$$Amarelo = produto\, Azul * Verde$$

Este é o fim dos registros da tabela "variaveisGlobais", então precisamos construir os próximos lugares. Os lugares são criados com base na tabela "funções". Os campos importantes para a criação dos lugares são N, V e P. Cada função precisa de um lugar, então vamos criar um lugar para *mint*. Seu nome deve ser definido com o prefixo par seguido do campo N, daí o nome *parmint*. O campo V define se o local deve ser um oráculo. Esse campo para *mint* é público, então deve ser um oráculo. P especifica a cor a ser dada ao lugar. Para *mint*, este campo possui os tipos *address* e *uint*, portanto sua cor deve ser o produto dos dois tipos. Como já temos uma cor amarela que define este multiconjunto, não precisamos criar uma nova cor, basta atribuir *parmint* a cor amarela.

O lugar criado para a função *send* segue o mesmo padrão que usamos na função *mint*, seu nome será *parsend*, o campo V é *public*, então ele será um oráculo, seus parâmetros são formados pelos conjuntos *address* e *uint*, então deve ser da cor amarelo.

Da mesma forma que as outras funções, o registro de *increaseBalance* deve gerar um lugar, e esse lugar é chamado de *parincreaseBalance*. O campo V é *private*, então não será do tipo oráculo. O campo P consiste nos tipos address e amount, portanto, o lugar é amarelo.

Precisamos criar um lugar para o construtor, e como qualquer outra função, seu nome é baseado em N, *parconstrutor*. Diferente de outras funções, o lugar do construtor não deve ser um oráculo, independente do valor do campo V. Como o construtor de example 4.2 contém o campo P como nulo, o lugar deve ter um conjunto de cores booleano conforme descrito na seção anterior. Este multiconjunto ainda não tem uma cor, então deve ser definida uma:

$$Roxo = \{true,\ false\}$$

Os últimos lugares criados são para as variáveis globalmente disponíveis. É necessário ler as entradas da tabela "operacoes" e "condicionais", pois nelas podemos encontrar nos registro o uso da variável *msg.sender*. A documentação da linguagem Solidity representa o valor *msg.sender* como um endereço, portanto o local criado deve ser do mesmo tipo.

Agora precisamos criar uma transição conforme explicado no capítulo anterior. A tabela 'funcoes' precisa ser lida e uma transição é criada para cada registro. Há um total de 4 transições: *mint*, *send*, *constructor*, *increaseBalance*. As transições terão o nome igual o campo N do registro. Feitas as transições o passo seguinte é criar os arcos. Primeiro são conectando os locais que representam os parâmetros com as suas respectivas funções. O arco deve ir do parâmetro para a função. Cada parâmetro deve estar associado com suas funções *parmint* e *mint*, *parend* e *send*, e *parincreaseBalance* e *increaseBalance*.

Os próximos arcos para serem criados serão entre as variáveis globais e as transições. Deverão ser criados dois arcos por variável, um entrando na transição, outro saindo. Começando pelas funções *mint*, *increaseBalance*, ao olhar os registros da tabela "operacoes" e "condicionais", a variável globalmente disponível *balances* foi usada, então deve ser conectado o lugar *balances* com as transições *mint* e *increaseBalance*.

No escopo das funções *construtor* e *mint* é realizado uma operação com a variável global *minter*. O *construtor* realiza uma atribuição com a variável então podemos encontrá-la no registro da tabela "operacoes". A função *mint* realiza uma comparação, então a variável será encontrada no registro da tabela "condicionais" Devido aos usos da variável o lugar *minter* deve ser conectado como entrada, com as transições *construtor* e *mint*.

A última variável global usada no contrato é *msg.sender*, uma variável globalmente disponível. A variável pode ser encontrada em operações dentro do escopo das funções *mint*, *send* e *construtor*. Por ser uma variável globalmente disponível deve ser gerado apenas um arco entre o lugar *msg.sender* e as funções *mint* e *send*, com o arco indo do lugar para a transição.

Os últimos arcos que devem ser gerados representam as chamadas de função. Na tabela "chamadas", ao observar os registros de quais funções realizam chamadas. Existem registros que indicam chamadas nas linhas 15 e 29 que estão no escopo das funções *mint* e *send* respectivamente. A função chamada em ambos os casos é a *increaseBalance*, por esse motivo as transições de mesmo nome devem ser conectadas por arcos saindo da transição e indo para o lugar *ParincreaseBalance*.

Com os arcos criado,s o próximo elemento a ser criado são as expressões de arco. Primeiro iremos gerar as expressões para os arcos de saída dos lugares que representam os parâmetros. Os arcos devem possuir variáveis segundo os conjuntos que formam a cor do lugar. Temos quatro parâmetros *ParMint*, *ParSend*, *ParincreaseBalance* e *ParConstructor*. Os três primeiros parâmetros possuem a mesma cor, então as mesmas variáveis podem ser usadas.

$$Z = endereço$$

$$E = uint$$

As variáveis do contrato que são associadas com essas variáveis de arco são: Z tendo o valor de *receiver* e E o valor de *amount*.

O lugar *parconstrutor* não possui parâmetros, portanto possui a cor roxa que representa um conjunto com os elementos $true$ e $false$. Ainda não existe uma variável para esse conjunto então uma nova será declarada.

$$A == \{true,\ false\}$$

Com as expressões dos arcos de parâmetros concluída, as expressões nos arcos saindo dos lugares representando variáveis globais devem ser preparados.

As variáveis globalmente disponíveis devem ser geradas primeiro. $Msg.sender$ é a única variável desse tipo, sendo usada em *mint*, *send* e *construtor*. $Msg.sender$ é azul, um endereço, já existem variáveis de arco desse tipo definidas, como D e Z, D será usado com o arco indo para a transição *send*, mas *mint* já está usando essa variável em outro arco de entrada da transição, então deve ser declarada uma nova variável de arco.

$$Y = endereço$$

O lugar *minter* possui um arco indo para a transição *mint*. A cor do lugar *minter* é azul, existe a variável Z com esse tipo, mas ela já está em uso para a transição *mint*. Então iremos criar uma nova variável de endereço, essa variável de arco será associada com a variável global *minter* para a transição *mint*.

$$D = endereço$$

A variável *balances* possui um arco indo para a transição *increaseBalance*. A cor desse lugar é amarela, sabemos que *balances* é um mapeamento. Devemos definir como a chave que seleciona o valor desse mapeamento é formada. Para descobrir basta analisar nos registros de operações qual variável foi usada como chave em *balances*. Descobrindo que é a variável *receiver*, vamos ver se ela é um parâmetro ou variável global. O registro de funções diz que essa variável é parâmetro da função *increaseBalance*. A transição *increaseBalance* possui a variável de arco Z associada com a variável *receiver*. A variável do índice foi selecionada, para o segundo conjunto da cor amarela, existe a variável E. Entretanto a variável E não pode ser usado, pois já está em uso em outro arco dessa transição, então teremos que criar uma nova que será associada, com a variável *balances*.

$$F = uint$$

O lugar *balances* também possui um arco indo para *send*. Assim como no processo anterior deve ser encontrada as operações que formam o índice de busca de *balances*. Nesse

caso o índice usado é *msg.sender*, que foi atribuída a variável de arco Y nessa transição. Precisamo de uma variável para o outro valor de *balances*. A variável de arco E já está em uso nessa transição, mas F está disponível. Portanto, o arco saindo de *balances* e indo para *send* será (Y, F).

Agora iremos começar a preparar os arcos que entram nos lugares. Começando com os arcos que entram nos parâmetros. A transição *parincreaseBalance* possui dois arcos de entrada. A função *send* chama função *increaseBalance* com as variáveis *receiver* e *amount*. Sabemos que ambas são variáveis de parâmetro da função *send*. Portanto, Z e E que são as variáveis de arco atribuídas para os parâmetros de *send* devem ser usadas como variáveis de arco de entrada para *ParincreaseBalance*.

O problema na função *send* é que a chamada da função deve ocorrer apenas se a condição da linha 21 não for cumprida. Isso ocorre, pois dentro da função existe um chamado do comando *revert*. Ele reverte todas as alterações feitas no banco de dados até o momento e finaliza a chamada. Conhecemos a variável de arco de *ammount* nessa transição e conhecemos a de *balances* também. Nenhuma das duas sofre alterações até o momento da comparação. A condição envolve as variáveis do contrato *amount* e *balances*, e as suas variáveis de arco são E e F respectivamente. Portanto, a expressão no arco deve ficar assim:

$$E > F- > NULLE <= F- > (Z, E)$$

A função *mint* também chama a função *increaseBalance*. Como no caso anterior, a chamada também é condicional. Neste caso, a chamada está na condição da linha 14. *Require* funciona de forma semelhante à condição if. Uma condição deve ser atendida para que esse disparo seja executado. Sabendo a variável que representa *msg.sender* é D nessa transição e também sabendo que a de *minter* é Y. Assim como no arco anterior os parâmetros da chamada da função *increaseBalance* são as variáveis *receiver* e *amount* com as variáveis de arco atribuídas Z e E. A expressão do arco deve ficar assim:

$$D = Y- > (Z, E)D! = Y- > NULL$$

Terminado os arcos de parâmetros vamos tratar os arcos de entrada das variáveis globais. Temos três arcos de entrada para eles um em *minter* e dois em *balances*. Começando por *minter*, nenhuma operação é realizada *minter* em *mint* então o arco de entrada para o lugar deve ser simples. Apenas a variável do arco de saída é repetida para a entrada, no caso D.

O arco de *increaseBalance* indo para *balances* retorna a variável Z que representa o índice sem alteração, mas a variável F passa por uma soma. A soma é realizada com *amount*, uma variável de parâmetro. A variável de arco de *amount* é E, logo a expressão de arco deve ser:

Figura 9 – Representação visual da Rede de Petri Gerada

$$(Z, F + E)$$

O arco da transição *send* indo para *balances* é mais complicado. Tal como acontece com o arco de *send* para $ParincreaseBalance$, as mesmas condições existem ao alterar o valor desta variável global. Porém, neste caso o arco deve gerar um marcador modificado se a condição for falsa e o marcador original se for verdadeiro. A expressão gerada fica assim:

$$E > F- > (Y, F)E <= F- > (Y, F - E)$$

5 CONCLUSÃO

Este trabalho aborda contratos inteligentes, uma tecnologia desenvolvida recentemente cujo objetivo é permitir que mais pessoas possam usar contratos no seu cotidiano, sem depender do governo e sua burocracia, para garantir o seu cumprimento. Também são usadas Redes de Petri Colorida, uma técnica de modelagem que permite representar sistemas concorrentes e linhas de produção. Com elas é possível visualizar o sistemas de modo a beneficiar o entendimento e clareza de como eles se comportam.

Como contratos inteligentes ainda são uma tecnologia nova, os seus escritores ainda são relativamente inexperientes. Essa inexperiência potencializa erros que podem ser cometidos durante a escrita do contrato. Para buscar diminuir os erros cometidos com essas novas tecnologias, este trabalho tentou colaborar com o desenvolvimento de técnicas de verificação mais rigorosas para contratos inteligentes. Para auxiliar essa área foi desenvolvido uma sistematização para representar contrato inteligentes em RPC. Com essa representação é esperado que seja possível realizar verificações mais rigorosas por falhas no funcionamento de um contrato inteligente.

Para desenvolver essa transformação, primeiro foi verificado se algo nesse caminho havia sido desenvolvido, achando ferramentas como [16] mostraram a possibilidade de representar um contrato inteligente em um RPC, pois o inverso já havia sido criado. Na ferramenta [16], a transformação foi feita passando as transições do contrato para funções e os lugares para variáveis globais, os arcos representavam o fluxo das execuções.

O algoritmo criado realiza a transformação do contrato inteligente em uma RPC, representada em um arquivo os elementos da definição da RPC. Não foi usado uma representação gráfica pela falta de um método que automatize o processo devido aos seguintes motivos. A PNML [22] que desenha Redes de Petri, poderia ser usada, mas para representar uma RPC seria necessário muitas alterações na representação. A ferramenta CPN IDEs [23] permite representar graficamente uma RPC, mas não foi possível desenvolver um modo de usar essa ferramenta de modo automático em um tempo hábil.

O algoritmo falha em representar comandos que não afetam o valor de variáveis, como exemplo os comandos event e error, que enviam um log para o usuário, são perdidos quando transformado na RPC. Isso ocorre pelo modo como a RPC é construída, as únicas operações possíveis são alterações de valores e chamadas de funções. Outra dificuldade do algoritmo é lidar com comando como revert que revertem alterações feitas no banco de dados e parando a execução da transação. A RPC não reconhece o que são alterações feitas por uma transação na blockchain ou múltiplas.

Como proposta de trabalhos futuros, seria possível aperfeiçoamento da sistemati-

zação, com o uso de dos contratos de transição e as transições temporizadas que foram introduzidas no capítulo 3.6.1. Continuar o processo da transformação, criando uma sistematização para converter as RPCs geradas nesse trabalho em um contrato inteligente.

REFERÊNCIAS

[1] SZABO, N. Formalizing and securing relationships on public networks. *First Monday*, v. 2, n. 9, Sep. 1997. Disponível em: <https://firstmonday.org/ojs/index.php/fm/article/view/548>.

[2] IREDALE, G. *History Of Blockchain Technology: A Detailed Guide*. 2020. Disponível em: <https://101blockchains.com/history-of-blockchain-timeline/>.

[3] VYPER. 2017. <https://docs.vyperlang.org/en/stable/toctree.html#>, Last accessed on 2023-04-30;.

[4] YUL. 2016. <https://docs.soliditylang.org/en/v0.8.17/yul.html>, Last accessed on 2023-04-30;.

[5] CAIRO. 2016. <https://www.cairo-lang.org/docs/>, Last accessed on 2023-04-30;.

[6] RUST. 2016. <https://www.rust-lang.org/learn>, Last accessed on 2023-04-30;.

[7] SOLIDITY. 2016. <https://docs.soliditylang.org/en/v0.8.13/index.html#>, Last accessed on 2022-07-25;.

[8] BUTERIN, V. A next generation smart contract & decentralized application platform. In: . [S.l.: s.n.], 2015.

[9] PETRI, C. *Kommunikation mit Automaten*. Rheinisch-Westfälisches Institut f. instrumentelle Mathematik an d. Univ., 1962. (Schriften des Rheinisch-Westfälischen Institutes für Instrumentelle Mathematik an der Universität Bonn). Disponível em: <https://books.google.com.br/books?id=NCZMvAEACAAJ>.

[10] JENSEN, K. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Vol 1, Basic Concepts*. 2. ed., 2. corr. printing. ed. Netherlands: Springer, 1997. (Monographs in theoretical computer science: an EATCS series). ISBN 3540609431.

[11] KRISTENSEN, L.; CHRISTENSEN, S.; JENSEN, K. The practitioner's guide to coloured petri nets. *International Journal on Software Tools for Technology Transfer*, Springer, v. 2, n. 2, p. 98–132, 1998. ISSN 1433-2779.

[12] MAVRIDOU, A.; LASZKA, A. Tool demonstration: Fsolidm for designing secure ethereum smart contracts. *CoRR*, abs/1802.09949, 2018. Disponível em: <http://arxiv.org/abs/1802.09949>.

[13] MAVRIDOU, A.; LASZKA, A. Designing secure ethereum smart contracts: A finite state machine based approach. *CoRR*, abs/1711.09327, 2017. Disponível em: <http://arxiv.org/abs/1711.09327>.

[14] GARCÍA-BAÑUELOS, L. et al. Optimized execution of business processes on blockchain. *CoRR*, abs/1612.03152, 2016. Disponível em: <http://arxiv.org/abs/1612.03152>.

[15] NAKAMOTO, S. Bitcoin: A peer-to-peer electronic cash system. maio 2009. Disponível em: <http://www.bitcoin.org/bitcoin.pdf>.

[16] ZUPAN, N. et al. Secure smart contract generation based on petri nets. In: _____. [S.l.: s.n.], 2020. p. 73–98. ISBN 978-981-15-1136-3.

[17] KASINATHAN, P.; CUELLAR, J. Workflow-aware security of integrated mobility services. In: LOPEZ, J.; ZHOU, J.; SORIANO, M. (Ed.). *Computer Security*. Cham: Springer International Publishing, 2018. p. 3–19. ISBN 978-3-319-98989-1.

[18] KASINATHAN, P.; CUELLAR, J. Securing the integrity of workflows in iot. In: *Proceedings of the 2018 International Conference on Embedded Wireless Systems and Networks*. USA: Junction Publishing, 2018. (EWSN '18), p. 252–257. ISBN 9780994988621.

[19] HECKEL, R. Open petri nets as semantic model for workflow integration. In: _____. *Petri Net Technology for Communication-Based Systems: Advances in Petri Nets*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. p. 281–294. ISBN 978-3-540-40022-6. Disponível em: <https://doi.org/10.1007/978-3-540-40022-6_14>.

[20] ZHANG, F. et al. Town crier: An authenticated data feed for smart contracts. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2016. (CCS '16), p. 270–282. ISBN 9781450341394. Disponível em: <https://doi.org/10.1145/2976749.2978326>.

[21] DIJKSTRA, E. W. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, Association for Computing Machinery, New York, NY, USA, v. 18, n. 8, p. 453–457, aug 1975. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/360933.360975>.

[22] BILLINGTON, J. et al. The petri net markup language: Concepts, technology, and tools. In: *Proceedings of the 24th International Conference on Applications and Theory of Petri Nets*. Berlin, Heidelberg: Springer-Verlag, 2003. (ICATPN'03), p. 483–505. ISBN 3540403345.

[23] CPNIDE. 2016. <https://cpnide.org/>, Last accessed on 2023-04-30;.