Universidade
Estadual de Londrina

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

K193t Kassuya, Danilo.

Transformação entre contratos inteligentes e redes de petri / Danilo Kassuya. - Londrina, 2023.
44 f. : il.

Orientador: Adilson Luiz Bonifacio.
Trabalho de Conclusão de Curso (Graduação em Ciência da Computação) - Universidade Estadual de Londrina, Centro de Ciências Exatas, Graduação em Ciência da Computação, 2023.
Inclui bibliografia.

1. Contratos inteligentes - TCC. 2. Solidity - TCC. 3. Redes de Petri Coloridas - TCC. 4. Blockchain - TCC. I. Luiz Bonifacio, Adilson. II. Universidade Estadual de Londrina. Centro de Ciências Exatas. Graduação em Ciência da Computação. III. Título.

CDU 519

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. Daniel dos Santos Kaster
Universidade Estadual de Londrina - UEL

Prof. Dr. Vitor Valerio de Souza Campos
Universidade Estadual de Londrina - UEL

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. 44f. 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. 44p. 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 20
Figura 2 – Exemplo de Sequenciamento 21
Figura 3 – Exemplo de Distribuição 21
Figura 4 – Exemplo de Junção 21
Figura 5 – Exemplo de Escolha Não-Determinística 22
Figura 6 – Exemplo Rede de Petri Colorida 24
Figura 7 – Exemplo de Rede de Petri Colorida com Extensões 26
Figura 8 – Arquitetura do processo de transformação 28
Figura 9 – Representação visual da Rede de Petri Gerada 40

LISTA DE TABELAS

Tabela 1 – Variáveis de Solidity 18

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 FUNDAMENTAÇÃO 13

2.1 Contratos Inteligentes 13

2.2 Blockchain 13

2.3 Ethereum 14

2.4 Solidity 16

2.5 Redes de Petri 19

2.6 Redes de Petri Coloridas 21

2.6.1 Extensão para RPCs 26

3 DE CONTRATOS PARA REDES DE PETRI 28

3.1 Arquitetura 28

3.2 Processo de Transformação 28

3.3 Aplicação prática da transformação 33

3.3.1 O Contrato *Coin* 34

3.3.2 Analisando o Contrato Solidity 35

3.3.3 Obtendo a RPC 37

4 CONCLUSÃO 41

REFERÊNCIAS 43

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 apresenta a

[1] do inglês, *smart contracts*

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

2 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.

2.1 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.

2.2 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.

2.3 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;

2.4 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 2.1. 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 2.1 – 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 2.1, 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.

2.5 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;*
- $O : A \rightarrow \{1, 2, \ldots\}$ *é a função peso associada aos arcos;*
- $K : P \rightarrow \{0, 1, 2, \ldots\}$ *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.

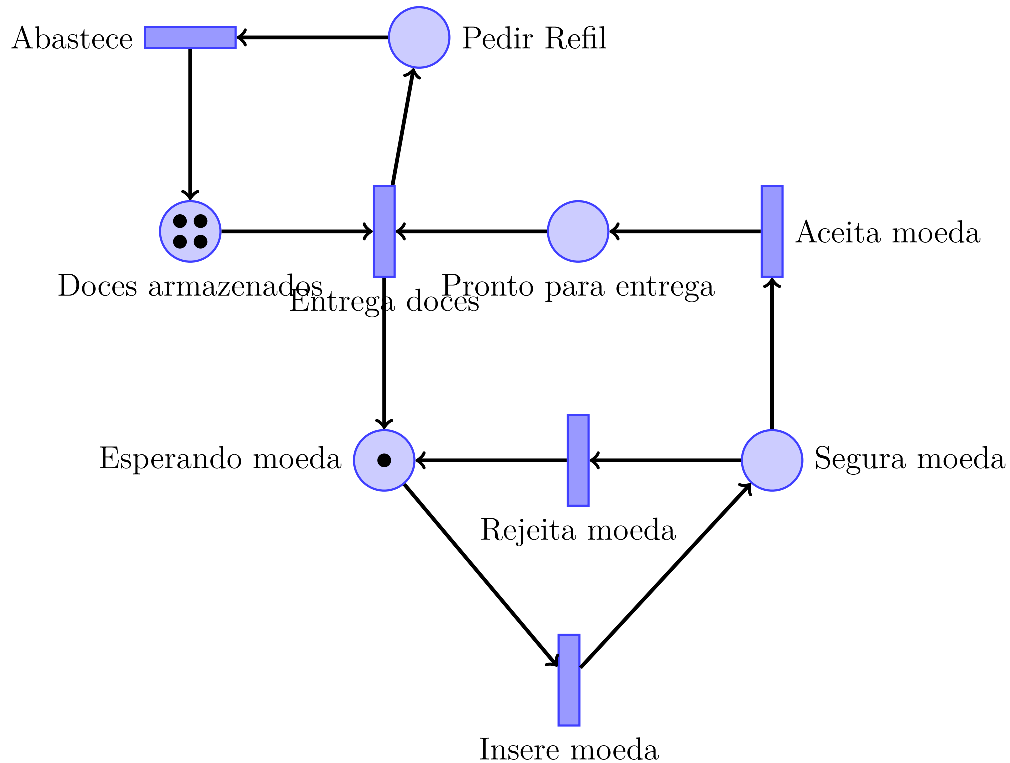

Figura 1 – Exemplo Rede de Petri

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.

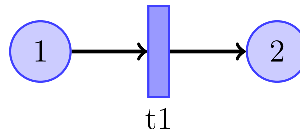

Figura 2 – Exemplo de Sequenciamento

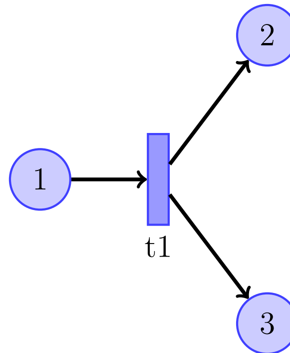

Figura 3 – Exemplo de Distribuição

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.

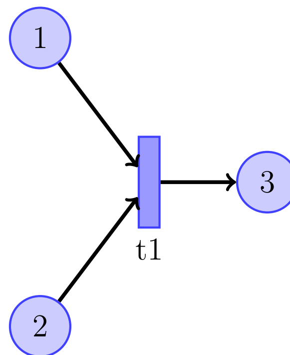

Figura 4 – Exemplo de Junção

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.

2.6 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.

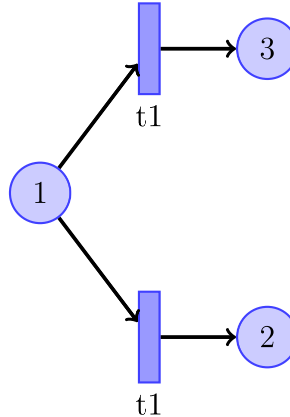

Figura 5 – Exemplo de Escolha Não-Determinística

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 $m : M \rightarrow \mathbb{N}$, 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 6 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

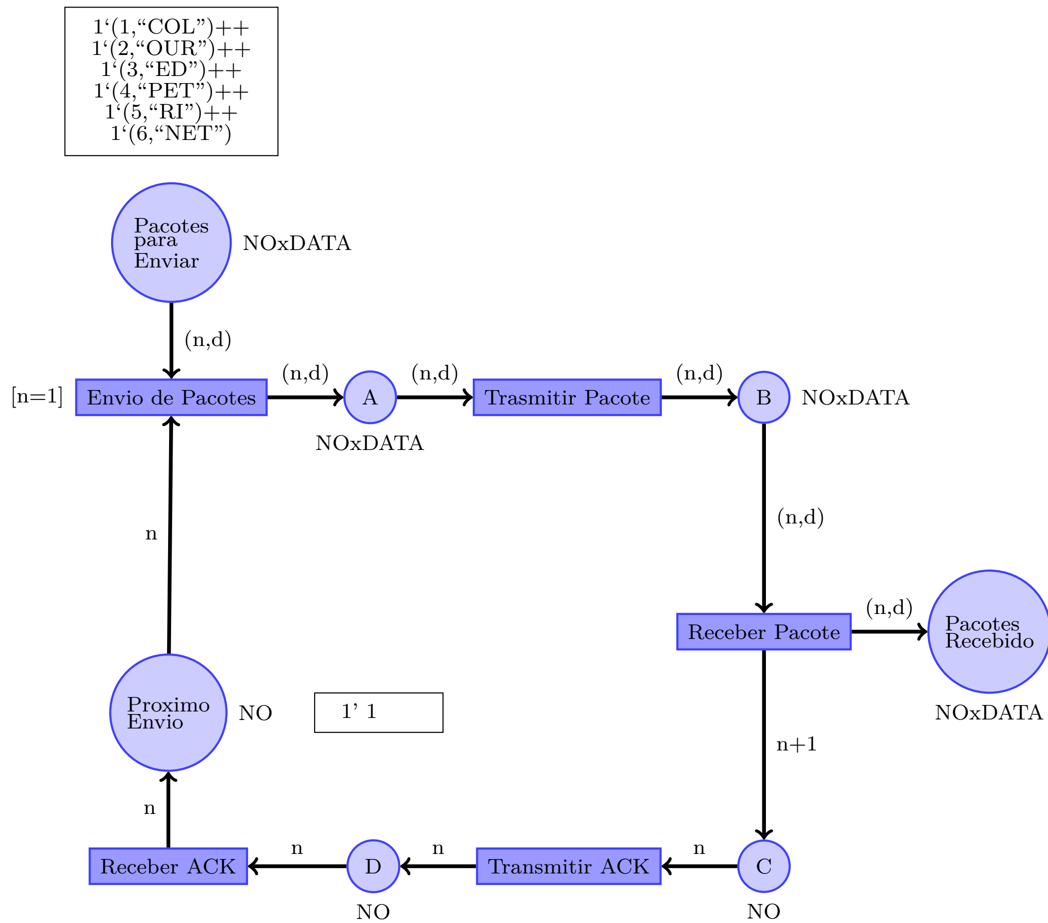

Figura 6 – Exemplo Rede de Petri Colorida

$G(t)$ 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 6 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.

2.6.1 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:

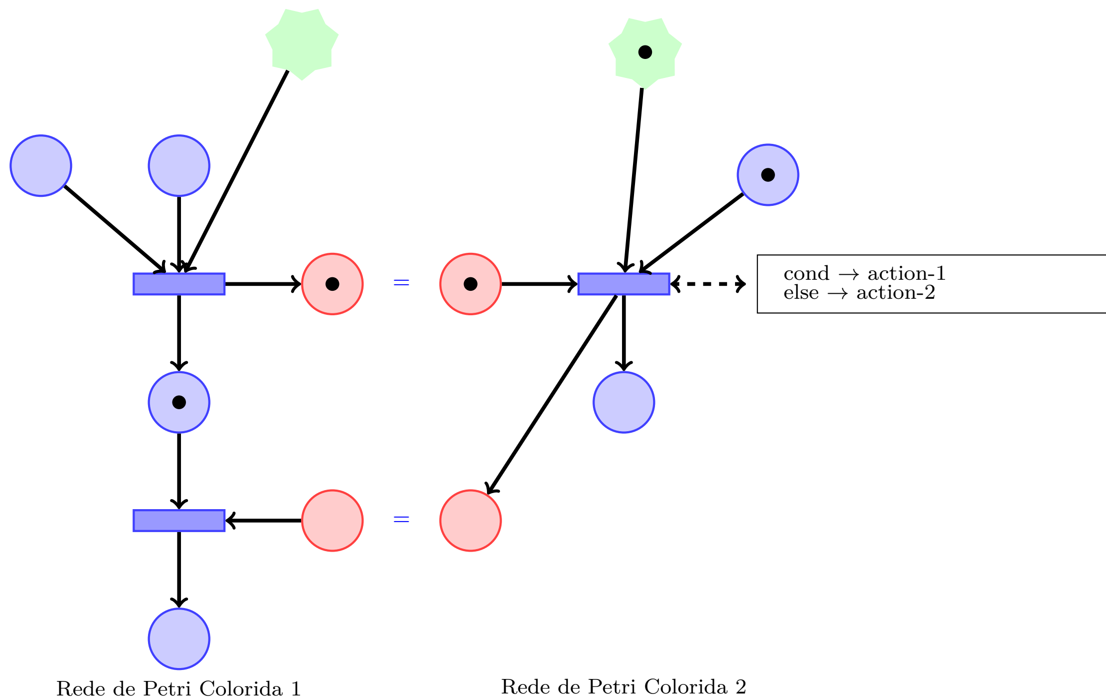

Figura 7 – Exemplo de Rede de Petri Colorida com Extensões

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.

3 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.

3.1 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.

3.2 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 etapa da análise do contrato dado deve identificar as *variáveis globais* e suas funções. Cada elemento obtido dessa análise gera um registro contendo as informações relevantes que por sua vez são gravadas em tabelas. As variáveis globais do contrato, por exemplo, são gravadas na tabela "variáveisGlobais", onde cada registro possui os seguintes campos: N, o nome; T, o tipo; V, o valor; TI o tipo do índice; e VI o nível de visibilidade. Caso a variável não seja indexada o valor de TI deve ser vazio.

Essa primeira análise também lida com o modo como cada variável é declarada, que pode ser de três formas distintas. A primeira forma é a mais comum, uma declaração com tipo simples tal como "int public x = 10", onde os campos para registro dessa variável são x, para o campo N, *int* para o campo T, 10 para o campo V, " " para o campo TI e *public* para VI. No tipo simples a declaração também pode ser apenas "int x" e, nesse caso, o campo V também é vazio.

Note que o campo TI é vazio nos dois casos, pois nas declarações do tipo simples as variáveis não possuem índice. No segundo modo a variável global é um vetor, por exemplo uma declaração do tipo "int a[20]". Nesse modo temos o registro com os seguintes campos, a para o campo N, *int* para T, " " para V, *int* para TI e *public* para VI, pois é o valor padrão da linguagem Solidity para esse campo. No segundo modo para a declaração de vetores o campo TI é sempre "int", ou seja os índices são sempre inteiros. O valor V é diferente de vazio quando neste modo de declaração existir um valor atribuído, por exemplo, "int a[20] = [1,2,3]". Nesse caso o campo V é igual a sequência $1, 2, 3$.

A última forma de declaração é o mapeamento, $mapping(A => B)\ C\ D$, onde A é o tipo do índice (campo de TI), B é o tipo da variável (campo T), C o nível de visibilidade (campo VI) e D é o nome da variável (campo N). Um exemplo desse tipo de declaração é dado por "mapping(address => int) private balances", onde $balances$ é o campo N, *int* é o campo T, V tem valor " ", $address$ é o campo TI, um tipo da linguagem Solidity como visto na Seção 2.4 e $private$ para VI. Vale ressaltar que esse modo de declaração permite que outros tipos de dados sejam usados como índice, além dos inteiros. Nesse último caso a variável se comporta de forma idêntica a de um vetor.

Nessa primeira análise também é realizada a identificação das funções do contrato. Cada função encontrada no código é identificada pela palavra-chave "function" e então

armazenada na tabela "funções". A tabela deve conter um registro para cada função armazenada com os campos N, NT, T, V, NP, P, I e F, respectivamente, o nome da função, uma lista com o nome das variáveis de retorno, uma lista com o tipo das variáveis de retorno, o nível de visibilidade da função, uma lista com o nome das variáveis de parâmetro, uma lista com o tipo das variáveis de parâmetro, a linha de início e de fim da função.

No exemplo da Descrição 3.1, na chamada de função *getResult*, o registro deve ser composto pelos campos: $N =$ "*getResult*", T uma lista com os elementos *uint* e *uint*, NT uma lista com os elementos *product* e *sum*, V contém *public*, P contém "", NP contém "", uma vez que a função não possui parâmetros, além de $I = 1$ e $F = 9$.

```
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    increaseBalance(receiver,amount);
9  }
```

Listing 3.1 – Exemplo de função

No caso de funções construtoras é preciso armazenar os campos I e F, o campo N ser identificado como "construtor" , os campos V, T e NT devem ser nulos, e os campos P e NP, listas com os tipos e nomes dos parâmetros, são obtidos como definido. Um exemplo de função construtora é dado abaixo, onde os campos são, respectivamente, *construtor* para o campo N, $NULO$ para os campos NT, T, V, NP e P, 1 para o campo I e 3 para o campo F.

```
1  constructor() {
2    minter = msg.sender;
3  }
```

Listing 3.2 – Exemplo de função construtora

Após o levantamento das variáveis e funções do contrato é necessário analisar as informações dentro do escopo das funções identificadas. O objetivo é gerar os registros de cada operação, condicional e chamada de função encontrada no contrato. Assim, um registro com a referida linha é criado para cada elemento, permitindo a identificação da função e o seu escopo, além de informações específicas sobre cada elemento. Nessa leitura também devem ser identificadas as variáveis globais que não foram declaradas de acordo com seu tipo pré-definido. A cor do lugar a ser criado dependerá do tipo da variável identificado, logo oráculo, pois seu valor varia a cada execução do contrato, dependendo de fatores externos.

A tabela “operações” deve armazenar os registros das operações identificadas usando os campos, V, O e P. O campo V representa o nome da variável que recebe o resultado da operação, O recebe a operação realizada e P recebe o nome de todas as variáveis da operação. No Exemplo 3.1 a linha 2 indica a declaração da variável a com valor 1. O registro então é gerado com os campos $V = a$, $O = 1$ e P com valor nulo, pois a operação não possui variáveis no escopo da operação. Já na linha 5 do exemplo o registro gerado tem $V = product$, $O = a \ * \ b$ e $P = \{a, b\}$.

Já a tabela “chamadas” armazena os registros através dos campos N, P e L, onde N é o nome da chamada, P as varáveis usadas na chamada da função e L é a linha da chamada. Na chamada “$increaseBalance(receiver, amount);''$ do exemplo 3.1, o registro é gerado com $N = increaseBalance$, $P = \{receiver, amount\}$ e $L = 8$.

Por fim, a tabela “condicionais” armazena os registros das condicionais com os campos T, C, I e F, onde T representa o tipo da condicional, C guarda a condição, composta de várias expressões, I representa a primeira linha da condicional e F representa a última linha do condicional. Considere a linha 4 do Exemplo 3.1. O comando condicional *require* é armazenado num registro com os campos $T = require$, $C = (a < 10 \ \&\& \ b < 5)$, $I = 4$ e $F = 7$.

Segunda etapa

Essa é a etapa responsável pela transformação com base no arquivo que descreve a RPC conforme Definição 2. Os elementos da RPC são gerados de acordo com os registros armazenados nas tabelas da etapa anterior. Essa etapa é ainda dividida em três passos de criação: dos lugares; das transições; e dos arcos.

O primeiro passo para geração dos lugares é realizado com base nos registros das variáveis globais e das funções. Cada lugar da RPC deve ter um nome, uma cor e os seus marcadores iniciais. Através da tabela “variáveisGlobais” o campo N dá nome aos lugares, enquanto as cores (multiconjuntos) são definidas pelo campo T, os tipos das variáveis globais. No arquivo de saída da RPC os lugares são armazenados como $P = \{a, b, c\}$, onde a, b e c são lugares da RPC, além de cada lugar possuir uma descrição com suas informações no formato “a={a;oráculo;azul;1’a}”, listando seu nome, tipo e marcação inicial.

Cada multiconjunto (ou cor) identificado deve ser armazenado no arquivo com as informações sobre a RPC assim como os lugares descrevendo os elementos do multiconjunto. Os valores em V são os marcadores iniciais do lugar gerado, enquanto TI é adicionado como um elemento para o multiconjunto do lugar gerado, caso exista. Quando o campo TI não é nulo a variável original era um array ou mapeamento, por isso o o lugar pode conter múltiplos marcadores que representam os valores da variável identificada. Dessa forma, cada valor pode ser identificado unicamente da mesma forma que

um array mapeia os valores armazenados com um índice. O novo elemento adicionado ao multiconjunto baseado no campo TI tem a função de simular esse índice.

Em seguida é realizada a transformação com a tabela “funções”. Algumas funções também geram lugares para representar os parâmetros de uma função interna, nesse caso usando o prefixo $par-$ seguido pelo campo N do registro. A visibilidade de uma função define se um lugar deve ser oráculo, pois funções públicas e externas podem ser chamadas externamente ao contrato, logo seus parâmetros possuem informações externas. Por isso, os lugares com o campo V igual a pública ou externa são criados como oráculos. Um lugar oráculo é um dos lugares modificados introduzidos na Seção 2.6 com o objetivo de se adicionar novos marcadores com base em dados externos. Cada lugar gerado nesse passo tem a cor definida pelo multiconjunto do tipo presente no campo P.

No caso de uma função construtora, sem parâmetros, o lugar de entrada deve conter um marcador do tipo booleano com o valor “true”. Isso se deve ao fato do 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.

O segundo passo dessa etapa é responsável pela geração das transições e dos arcos que conectam essas transições com os lugares criados no passo anterior através da tabela “funções”. Uma transição então é criada com o nome definido no campo N. Os arcos então devem ser criados conectando as transições e lugares. Cada transição criada é baseada em uma função, enquanto os arcos que conectam as transições representam o fluxo dos processos dessas funções. Primeiramente são criados os arcos entre os lugares baseados em parâmetros e as transições baseadas nas funções desses parâmetros. Cada lugar baseado em parâmetro deve ser conectado por um arco com sua transição correspondente. Em seguida são criados os arcos que conectam os lugares baseados em variáveis globais e as transições. Os lugares baseados em variáveis globais não declaradas são então conectados as respectivas transições, caso a função que representa essa transição utilize a variável. Para isso deve ser analisada as tabelas de operações e condicionais para verificar se uma variável global não declarada foi usada.

Na sequência são tratadas todas as outras variáveis globais com a criação de arcos entre as transições que representam as funções que usam essas variáveis globais e os lugares com o mesmo nome das variáveis usadas. Quando o uso de uma variável global é identificado dois arcos devem ser gerados, um entrando na transição e outro saindo desta. O arco de entrada representa o uso da variável global no contrato enquanto o arco de saída representa o retorno dos marcadores consumidos pelo disparo da transição. O valor retornado deve refletir o valor da variável global após a execução da função.

Os últimos arcos a serem criados representam a ordem com que as funções do contrato são chamadas. Com vimos anteriormente, as transições representam as funções do contrato na RPC e sabemos que uma transição não pode se conectar a outra diretamente.

Essas transições são conectadas usando os lugares criados anteriormente para representar os parâmetros das funções. O arco criado entre a transição e um lugar simula 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 é preciso gerar as expressões com base nas variáveis associadas a esses arcos. 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.

Inicialmente são geradas as expressões para os arcos que vão dos lugares para as transições e que definem os valores dos marcadores que cada variável recebe. As variáveis de arco não devem ser utilizadas em outros arcos de entrada para esta transição, pois nesses arcos estão associados os tipos dos marcadores de cada variável. Associar uma variável em dois arcos distintos é o mesmo que declarar esta variável duas vezes com o mesmo nome numa mesma função. A união de todos os tipos das variáveis devem resultar no multiconjunto do lugar ao qual estão conectadas.

Em seguida são geradas as expressões de arcos saindo das transições. A expressão deve conter as variáveis de arco que representam as variáveis do contrato que foram utilizadas na operação que esse arco representa. No caso de um arco entre uma transição e um lugar que representa um parâmetro de uma função, esse arco debe representar a chamada dessa função. Como a chamada possui variáveis que definem os parâmetros da função, a expressão de arco deve conter o valor das variáveis utilizadas no contrato. Caso a variável utilizada pela operação, chamada de função ou condicional, tenha sido declarada dentro da função, ou seja, não é um parâmetro da mesma, então não deve existir uma variável de arco, já que na RPC também não existirá um lugar representando esta variável. Portanto uma expressão matemática deve representar os possíveis valores dessa variável, assim como outras variáveis podem estar representes nessa expressão. A execução de uma operação sobre a variável de arco também é representada por uma expressão de arco. Da mesma forma as condicionais são representadas por expressões para a geração de marcadores na RPC. Logo, uma condicional numa chamada de função deve resultar na mesma condicional de arco não permitindo a geração de marcadores quando a função não é chamada.

3.3 Aplicação prática da transformação

Esta seção apresenta uma aplicação prática da transformação através de um contrato inteligente em Solidity com o objetivo de se obter a respectiva RPC.

3.3.1 O Contrato *Coin*

O contrato *Coin*, descrito no Código 3.3, simula saldos de uma moeda fictícia, permitindo transferências entre os usuário, além da adição de valores efetuadas pelo proprietário do contrato. Cada usuário do contrato possui uma quantidade da moeda e pode enviá-las para outros usuários. Além disso os usuários tem conhecimento sobre o responsável do contrato, o único capaz de gerar novas moedas.

O contrato *Coin* mapeia os saldos de usuários através dos endereços das contas. O endereço do proprietário é atribuído para *minter* pelo construtor. Já a variável *balances* mapeia um vetor de *uint*, os inteiros positivos em Solidity, para os endereços de contas dos usuários do Ethereum.

O contrato possui, basicamente, três funções: *mint*; *send*; e *increaseBalance*; A função *mint* é responsável por adicionar mais moedas aos saldos dos usuários do contrato. Esta função possui dois parâmetros, *receiver* e *amount*, que definem o endereço do usuário que deve receber o saldo e a quantidade de moedas. A função verifica o endereço do usuário através da variável global“msg.sender” e compara com o endereço do usuário salvo em *minter*. Caso os endereços sejam iguais o saldo da conta com o endereço *receiver* é aumentado pelo valor de *balances*.

A função *send* também possui dois parâmetros e é responsável por enviar o valor *amount* da conta do usuário que chama a função para o endereço da conta em *receiver*. A função verifica, primeiramente, se há saldo suficiente para a transferência na conta do usuário que chamou a função. Caso exista saldo o valor é somado ao saldo da conta em *receiver* e subtraído da conta que chamou a função. Caso contrário as alterações são anuladas.

Já a última função *increaseBalance* é privada e só pode ser chamada por outras funções do contrato, ou seja, não é possível chamá-la externamente. Esta função realiza a adição de moedas às contas através das funções *mint* e *send* que realizam as alterações necessárias. Os parâmetros *receiver* e *amount* definem o endereço da conta a ter o saldo aumentado e o valor, respectivamente, tal como na função *mint*.

Note que na blockchain os usuários podem enviar mensagens para a rede com chamadas de funções para qualquer contrato de interesse. Qualquer usuário pode acessar qualquer contrato da rede desde que o nível de acesso seja satisfeito, ou seja, público ou externo. Para evitar acesso indesejado de usuários ou impedi-los de alterar valores ou termos do contrato, condicionais relacionadas aos endereços desses usuários devem ser adicionadas, assim como no exemplo apresentado, onde o endereço do executor é verificado.

```
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 3.3 – Exemplo de código solidity

3.3.2 Analisando o Contrato Solidity

Conforme o processo apresentado na Seção 3.2, para se obter a RPC resultante precisamos gerar os registros de cada tabela para o contrato. No primeiro passo são gerados os registros das funções e variáveis globais. Todos os elementos em cada passo são obtidos numa única análise sobre o No entanto, para maior clareza na transformação, vamos focar em um elemento por vez, começando pelas variáveis globais. O registro das variáveis globais tem os campos N, T, V, TI e VI e o contrato possui duas variáveis globais, *minter* e *balances*. Assim obtemos dois registros, o primeiro com N = minter; T

= address; V = “”; TI = “” e VI = public, e o segundo com N = balances; T = uint; V = “”; e TI = address e VI = public.

Já os registros das funções possuem os campos N, NT, T, V, NP, P, I e F. Como o contrato possui as funções, *sent*, *mint* e *increaseBalance*, além do construtor, então 4 registros são gerados. O primeiro com N = construtor; NT = “”; T = “”; V = “”, P = “”; NP = “”; I = 9; F = 11, o segundo com N = mint; NT = “”; T = “”, V = public; P = address, uint; NP = “receiver, amount”; I = 13; F = 16, o terceiro com N = send; NT = “”; T = “”, V = public P = address, uint; NP = “receiver amount”; I = 20; F = 30 e o quarto com N = increaseBalance; NP = “”; T = “”; V = private; P = address, uint; NP = “receiver, amount”; I = 32; F = 35.

No segundo passo são obtidos os registros das 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 global *msg.sender* para a variável global *minter*. Deve ser gerado um registro dessa operação com os seguintes campos, V = minter; O = msg.sender; e P = minter; msg.sender.

A função *mint* não possui operações e a função *send* possui apenas uma. Na linha 27 a função *send* subtrai o valor do mapeamento *balances* de índice *msg.sender* no valor de *amount*. Como o operador $-=$ simboliza subtração e atribuição, o campo O armazena os elementos da operação “balances[msg.sender] -= amount” como “balances[msg.sender] = balances[msg.sender] - amount”. Após gerar o registro temos os campos V = balances[msg.sender]; O = balances[msg.sender]-amount; e P = balances[msg.sender], amount.

A função *increaseBalance* possui uma operação de soma na linha 33. O valor de *amount* é somado a variável global *balances* com o índice *receiver*. Aqui a operação possui o sinal $+=$ que realiza uma soma e atribuição. Assim como no caso $-=$ o parser identifica o símbolo e gera um registro da operação de soma e atribuição. O registro dessa operação deve ter os campos, V = balances[receiver]; O = balances[receiver]+amount; P = balances[receiver], amount.

Em seguida são gerados os registros das chamadas. O registro possui os campos N, P e L. O construtor não realiza nenhuma chamada, enquanto a função *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. Observe que a função *send*, na linha 28, também realiza a chamada da função *increaseBalance*. Por isso, um registro é gerado com os campos N = increaseBalance; P = receiver, amount; e L = 29. Já a função *increaseBalance* não possui nenhuma chamada.

Por fim são gerados os registros da tabela das condicionais. As funções construtoras não possuem nenhuma condicional. A função *mint*, por outro lado, usa a função *require*

para finalizar a função chamadora caso a condição expressa sobre seus parâmetros seja falsa. Um registro para essa condição é então gerado com os campos T = require; C = msg.sender == minter; I = 14; e F = 16. Note que o restante da função após a linha 16 ($F = 16$), não será executado quando a condicional é falsa.

A função *send*, na linha 21, possui um comando if com a expressão "amount > balances[msg.sender]". Caso a expressão seja verdadeira, as linhas que delimitam o escopo da condicional devem ser executadas. Essa verificação no contrato garante que o valor do parâmetro da função *send* seja menor do que o valor da variável $balances$. Logo, o registro gerado para este comando deve ter os campos T = if; C = amount > balances[msg.sender]; I = 21; e F = 25. Como a função *increaseBalance* não possui um comando condicional, nenhum registro é gerado.

3.3.3 Obtendo a RPC

Neste passo obtemos a RPC com base na análise anterior. Conforme a sistematização, para cada variável global um lugar deve ser criado. Portanto, há um lugar para *minter* e um lugar para *balances*, seguindo as entradas da tabela "variáveisGlobais". A cor desses lugares deve corresponder ao tipo do campo T no registro. Logo, *minter* deve ser da cor $address$, definida como azul ($Azul = address$). Já a variável *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 os endereços. Nesse caso definimos *uint* com a cor verde ($Verde = uint$). A cor para o tipo *uint* foi definida, assim como para o tipo address, então é possível criar a cor para *balances* utilizando ambas, $Amarelo = produto, Azul * Verde$

Com isso foram criadas as cores e os lugares baseados nas variáveis globais. Os próximos lugares são criados com base na tabela "funções". Os campos da tabela "funções" importantes para a criação dos lugares são N, V e P. Cada função dá origem a um lugar. Criamos então um lugar para *mint* com o prefixo *par* seguido do campo N, daí o nome $parmint$. O campo V define se o lugar deve ser um oráculo, conforme visto na segunda etapa do capítulo anterior. O campo V da função *mint* é público, logo o respectivo lugar deve ser um oráculo. O campo P especifica a cor, que no caso de *mint* 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 é necessário criar uma nova cor, basta atribuirmos *parmint* a cor amarela.

O lugar criado para a função *send* segue o mesmo padrão que usamos na função $mint$. O nome do lugar é definido com $parsend$, o campo V é *public*, logo será um oráculo, e seus parâmetros são formados pelos conjuntos $address$ e *uint*, por isso atribuída a cor amarela. Um lugar também é criado para o registro de *increaseBalance* com o nome *parincreaseBalance*. Como o campo V é *private* não será um oráculo e o campo P com os tipos $address$ e $amount$ faz com o que lugar tenha a cor amarela.

Também precisamos criar um lugar para o construtor, nesse caso com o nome *parconstrutor* baseado no campo N. Diferente de outras funções, o lugar do construtor não deve ser um oráculo, independente do valor do campo V. Como o construtor do Exemplo 3.3 contém o campo P como vazio, o lugar deve ter uma cor do tipo booleana. Assim atribuímos a cor "$Roxa = \{verdadeiro, falso\}$" já que este multiconjunto ainda não possui uma cor definida.

Por fim são criado os lugares para as variáveis globais não declaradas. Na tabela "operações" e "condicionais" são encontrados os registros para a 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.

No próximo passo criamos as transições da RPC usando a tabela "funções". Temos assim um total de 4 transições, *mint*, *send*, *constructor*, e *increaseBalance*, obtidas através do campo N dos registros. Na sequência são criados os arcos, conectando os lugares que representam parâmetros às transições que representam suas respectivas funções. Dessa forma temos os arcos de *parmint* para *mint*, de *parend* para *send*, e de *parincreaseBalance* para *increaseBalance*.

Os próximos arcos são gerados entre os lugares obtidos de variáveis globais e as transições existentes. Neste caso dois arcos são criados por variável, um de entrada para transição e outro de saída. Assim um arco é criado do lugar *balances* para as transições *mint* e *increaseBalance* usando os registros das tabelas "operações" e "condicionais". No escopo do *construtor* há uma atribuição com a variável *minter*, e assim encontramos um registro na tabela "operações". Já na função *mint* existe uma comparação com a mesma variável, e por isso encontramos um registro da tabela "condicionais". Dessa forma, criamos os arcos do lugar *minter* para as transições *construtor* e *mint*. Para a variável global $msg.sender$, encontrada em operações das funções *mint*, *send* e *construtor*, criamos os arcos do lugar $msg.sender$ para as transições *mint* e *send*. Os últimos arcos devem representar as chamadas de função usando a tabela "chamadas". Os registros indicam as chamadas nas linhas 15 e 29 no escopo das funções *mint* e *send*, respectivamente. A função chamada em ambos os casos é a *increaseBalance*, gerando assim os arcos das transições *mint* e *send* para o lugar $ParincreaseBalance$.

No passo seguinte são geradas as expressões de arco. Primeiro criamos as expressões para os arcos de saída dos lugares que representam 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 possuem a mesma cor, e por isso usam as mesmas variáveis, "Z" como endereço e "E" como $uint$. As variáveis do contrato associadas a essas variáveis de arco são: "$Z = receiver''$ e "$E = amount''$. O lugar *parconstrutor*, de cor roxa com valor booleano, não possui parâmetros. Portanto, uma nova variável é declarada com $A = verdadeiro|falso$.

Com as expressões dos arcos de parâmetros concluída, geramos as expressões nos arcos de saída dos lugares que representam variáveis globais. A variável $Msg.sender$ é a única desse tipo, sendo usada em $mint$, $send$ e $construtor$. Como a cor de $Msg.sender$ é azul, ou seja, um endereço, já existem variáveis de arco desse tipo definidas, como "D" e "Z". A variável "D" é usada no arco de entrada para a transição *send*. Porém, como *mint* já usa essa variável em outro arco de entrada para a transição, então uma nova deve ser declarada, a variável de arco "Y" como um endereço.

O lugar $minter$ possui um arco indo para a transição $mint$. A cor do lugar $minter$ é azul, usando a variável "Z", no arco de entrada para a transição $mint$. Então uma nova variável de endereço "D" é criada, sendo associada a variável global $minter$ para a transição $mint$.

O lugar $balances$ possui um arco para a transição $increaseBalance$, com a cor amarela. Para se criar a chave do mapeamento são analisados os registros das operações da variável usada como chave em $balances$. Temos então a variável $receiver$, um parâmetro da função $increaseBalance$. A transição $increaseBalance$ já possui a variável de arco "Z" associada a variável $receiver$ e o segundo conjunto da cor amarela com a variável "E". Entretanto, como a variável "E" já está sendo usada em outro arco dessa transição, uma nova variável, "F" para uint, é criada e associada a $balances$.

O lugar $balances$ também possui um arco para $send$. Assim como no processo anterior deve ser encontrada as operações que formam o índice de busca em $balances$. Nesse caso o índice usado é $msg.sender$, que foi atribuída a variável de arco "Y" nessa transição. Logo, precisamos de uma variável para o outro valor de $balances$. Como a variável de arco "E" já está em uso nessa transição, mas "F" está disponível, então o arco de $balances$ e para $send$ é definido como (Y, F).

Na sequência são gerados os arcos de entrada dos lugares, começando pelos arcos de entrada de lugares que representam parâmetros. A transição $parincreaseBalance$ possui dois arcos de entrada. A função $send$ chama $increaseBalance$ com as variáveis $receiver$ e $amount$. Sabemos que ambas são parâmetros da função $send$. Portanto, "Z" e "E", variáveis de arco para os parâmetros de $send$, devem ser usadas como variáveis de arco para $ParincreaseBalance$.

No caso da função $send$ existe uma condição na linha 21 que precisa ser satisfeita. Isso ocorre pelo uso do comando $revert$, que reverte todas as alterações feitas no banco de dados até o momento da chamada, e a finaliza. Conhecemos as variáveis de arco de $amount$ e $balances$ nessa transição. Nenhuma das duas sofre alterações até o momento da comparação na linha 21, que usam as variáveis de arco "E" e "F". Assim a expressão é definida como $E > F \rightarrow NULL$, $E <= F \rightarrow (Z, E)$.

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, $D = Y \to (Z, E)$, $D! = Y \to NULL$.

Após os arcos de parâmetros são tratados os arcos de entrada das variáveis globais. Temos três arcos de entrada, um para *minter* e dois para *balances*. Para *minter* não há nenhuma operação em *mint*, logo 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* para *balances* retorna a variável “Z” que representa o índice sem alteração. Já a variável “F” passa por uma soma usando *amount*, uma variável de parâmetro. Como a variável de arco para *amount* é “E”, a expressão de arco deve ser $(Z, F + E)$.

O arco da transição de *send* para *balances* segue as condições do arco de *send* para *ParincreaseBalance*. Porém, neste caso o arco deve gerar um marcador modificado caso a condição seja falsa ou um marcador original caso seja verdadeira. A expressão gerada deve ser $E > F \to (Y, F)$, $E <= F \to (Y, F - E)$.

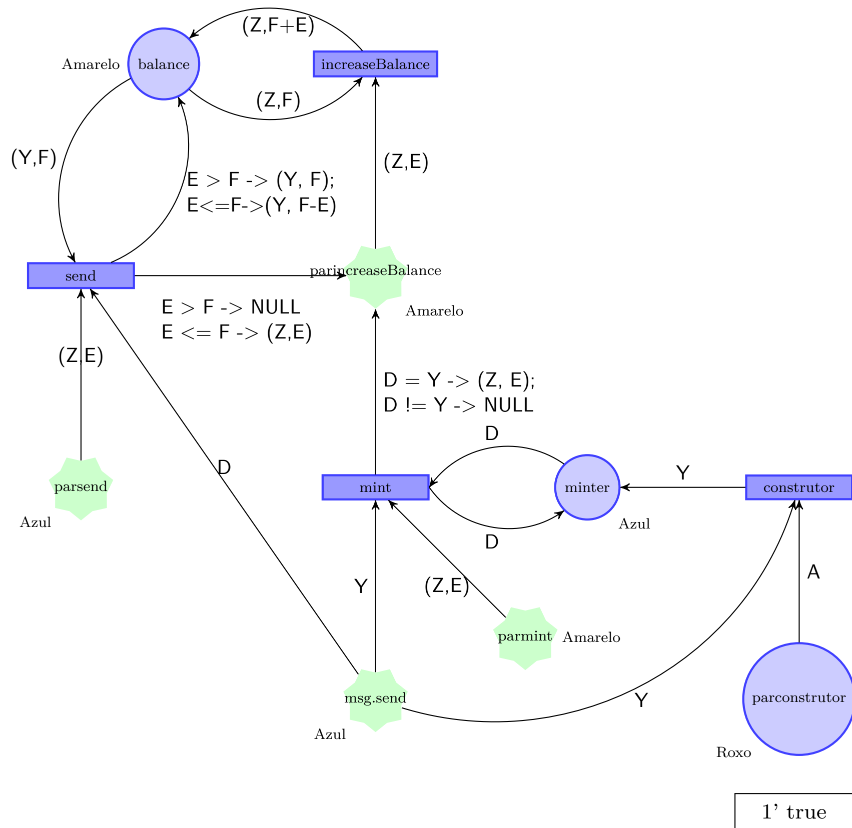

Figura 9 – Representação visual da Rede de Petri Gerada

4 CONCLUSÃO

Este trabalho aborda contratos inteligentes, uma tecnologia recentemente desenvolvida com o objetivo de permitir que mais pessoas possam utilizar contratos no seu cotidiano, sem depender da burocracia governamental para garantir o seu cumprimento. Além disso são usadas Redes de Petri Colorida, uma técnica de modelagem que possibilita representar sistemas concorrentes e linhas de produção. Essa técnica proporciona uma visualização clara e compreensível do comportamento dos sistemas.

Dado que os contratos inteligentes ainda são uma tecnologia nova, os seus programadores ainda são relativamente inexperientes. Essa inexperiência potencializa erros que podem ser cometidos durante a escrita do contrato. Com o intuito de 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.

A fim de 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 uma RPC, pois elas já haviam desenvolvido o processo inverso. Na ferramenta [16], a transformação foi feita passando as transições do contrato para funções, os lugares para variáveis globais e os arcos representavam a ordem em que as operações do contrato inteligente eram executados.

O algoritmo criado realiza a transformação do contrato inteligente em uma RPC, representando os elementos da definição da RPC em um arquivo de texto. 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 utilizar 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. A RPC não possui um modo de reconhecer quais operações devem ser revertidas, e nem armazena os valores para realizar essas reversões.

Como proposta de trabalhos futuros, seria possível aperfeiçoamento da sistematização, 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;.