UNIVERSIDADE FEDERAL FLUMINENSE

Bruno Olímpio Costa

VERIFICAÇÃO FORMAL DE MODELOS DE BLOCKCHAIN

NITERÓI

2019

UNIVERSIDADE FEDERAL FLUMINENSE

Bruno Olímpio Costa

VERIFICAÇÃO FORMAL DE MODELOS DE BLOCKCHAIN

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Engenharia de Sistemas e Informação

Orientador:
Bruno Lopes

NITERÓI

2019

Ficha catalográfica automática - SDC/BEE
Gerada com informações fornecidas pelo autor

C837v Costa, Bruno Olímpio
Verificação formal de modelos de blockchain / Bruno Olímpio Costa ; Bruno Lopes Vieira, orientador. Niterói, 2019.
67 p. : il.

Dissertação (mestrado)-Universidade Federal Fluminense, Niterói, 2019.

DOI: http://dx.doi.org/10.22409/PGC.2019.m.11783433752

1. Base de dados distribuída. 2. Transferência eletrônica de fundos. 3. Produção intelectual. I. Vieira, Bruno Lopes, orientador. II. Universidade Federal Fluminense. Instituto de Computação. III. Título.

CDD -

Bibliotecária responsável: Fabiana Menezes Santos da Silva - CRB7/5274

BRUNO OLÍMPIO COSTA

VERIFICAÇÃO FORMAL DE MODELOS DE BLOCKCHAIN

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Engenharia de Software

Aprovada em agosto de 2019.

BANCA EXAMINADORA

Prof. Bruno Lopes - Orientador, UFF

Prof. Uéverton dos Santos Souza, UFF

Prof. Edward Hermann Haeusler, PUC-Rio

Prof. Jefferson de Barros Santos, PUC-Rio & FGV

Prof. Mario Roberto Folhadela Benevides, UFRJ

Niterói

2019

Dedicatória: Ao meu amigo-pai-anjo-da-guarda João Batista da Costa (in memorian) por me ensinar, sempre pelo exemplo, que a felicidade está na simplicidade, e me mostrar que eu era mais forte do que todos os meus medos e carências.

Agradecimentos

Sempre em primeiro lugar ao meu Pai Amoroso, Deus, e ao irmão mais velho, Jesus, por me proporcionarem viver e evoluir com propósito. Quanto mais estudo, aprendo e me aprofundo na ciência, mais creio e conheço Deus.

À Universidade Federal Fluminense, por me abrigar primeiramente como profissional, quando pude compreender o que é o serviço público e lapidar minhas definições sobre ética e comprometimento com o Bem e os bens de todos; depois como discente, em particular no Instituto de Computação, onde o contato com mentes geniais me ofereceu vasto campo ao aprendizado seguro e a construção do profissional e do cidadão que sou. Ser Uffiano é um dos meus maiores orgulhos.

Ao meu Orientador (com "O" maiúsculo), Prof. Bruno Lopes Vieira, sem cuja disciplina, seriedade e compreensão, este trabalho não teria sido realizado. Tenho consciência de que orientar alguém com a minha (in)disponibilidade de tempo e a minha inexperiência no meio, custou-lhe bastante esforço e, por isto agradeço-lhe com profundo respeito e admiração, pois grande parte deste projeto só tornou-se concreto porque você acreditou em mim e me guiou, incansável, até aqui.

À minha Mãe, Maezinha, Roseléa Olímpio, por se entregar no mais alto grau de doação e Amor, com o melhor que pode dar, desde antes que eu nascesse. Por ter me proporcionado oportunidades, confiado em mim e permitido que, por meus esforços e escolhas, eu trilhasse o meu caminho. Pelos valores, pelas lições, pelos sacrifícios...ser seu filho é a maior honra da minha vida e a maior prova de que Deus me ama muito.

À Vivi Maia, por me acolher na dor, me respeitar na dúvida, me compreender na ausência e me amar a cada passo; por aprender e por se abrir, por compartilhar, por me e se permitir, por me dar lugar e me deixar assistir, por me convidar a ser seu par e por querer construir, mais do que acertar. É Amor, é convicção, é escolha, e é construção.

À toda a comunidade do DIJ-IEBM por me acolherem e me propiciarem o imprescindível campo de trabalho ao cumprimento de minha mais importante missão na vida, e também por ofertarem tanto Amor e tantas demonstrações de que eu não estou só.

Ao meus amigos e familiares, por continuarem me amando e suportando as minhas negativas, sucedidas de “tenho que trabalhar na dissertação”.

Ao André Ximenes, cuja sensibilidade e destreza foram capazes de me conduzir a um profundo, doloroso, necessário e maravilhoso processo de autodescobrimento e cura. Profunda gratidão e carinho.

À Mariana Quintal por me incentivar, por suportar minhas ausências e por confiar vigorosamente no meu potencial, ao mesmo tempo em que contribuía para o meu desenvolvimento profissional e tornava-se cada vez mais amiga.

Ao Bruno Belizário, por sua disponibilidade e prestatividade no auxílio direto à realização deste trabalho, tanto quanto nos incontáveis cafés para tratar do tema ou simplesmente me ouvir reclamar.

Aos colegas de trabalho da STI-UFF e da FEC, por me proporcionarem desafios e me ajudarem a crescer profissionalmente enquanto me permitiam contribuir para o crescimento das instituições.

Resumo

A Blockchain é considerada uma das tecnologias mais relevantes e disruptivas desde a concepção da Internet. Diversas aplicações tem sido propostas desde sua criação, em 2008, como o núcleo da criptomoeda Bitcoin. Diversos episódios de furtos de dados e de valores financeiros envolvendo aplicações que utilizam Blockchain foram registrados, o que evidencia a necessidade de maior verificação por confiabilidade de que tanto Blockchains quanto suas aplicações sejam efetivamente seguras. Neste contexto, a verificação formal de modelos pode trazer significativas contribuições, dado que ela se utiliza de técnicas de validação de forte embasamento matemático, possibilitando assegurar a ausência de erros avaliados. Dentre os trabalhos existentes que, de alguma forma, fazem uso da verificação formal de Blockchains, o foco em geral está nas aplicações que a utilizam, apresentando apenas "modelos minimalistas" de Blockchains. Este trabalho apresenta uma formalização de Blockchain mais completa do que as existentes e utiliza o verificador de modelos nuXmv para realizar inferências. Além disto, apresenta um compilador para código SMV para criar modelos de Blockchains parametrizados por suas características.

Palavras-chave: Blockchain, Verificação formal, Bitcoin, Verificação de modelos

Abstract

The Blockchain is considered one of the most relevant and disruptive technologies created since the Internet. Several applications have been proposed since its creation in 2008, as the core of the cryptocurrency Bitcoin. Despite it, many episodes of stolen data and money already happened involving Blockchain applications which evidence the need for more reliability that both Blockchains and its applications are effectively safe. In this context, model checking can bring significant contributions, since it provides verifications over a formal model, pointing out errors before they cause problems. Some works addressed, in some way, formal Blockchain models, but they focus on the applications that use it, presenting just a "minimalistic" model, lacking on completeness of the protocol. This work presents a formalization of Blockchain that is more complete than the existent others, and uses the model checking tool nuXmv to proceed formal verifications on that. The base model is the Bitcoin blokchain, and are explored the main decentralization features and conditions needed to build the Blockchain. A compiler to SMV code from the description of a Blockchain is also provided.

Keywords: Blockchain, Model checking, Formal verification, Bitcoin, Model checking

Lista de Figuras

2.1 Representação da fórmula CTL AXφ 5

2.2 Representação da fórmula CTL EFφ 6

2.3 Representação da fórmula CTL AφUψ 6

2.4 Representação da fórmula CTL EGφ 6

2.5 Exemplo de árvore binária 9

2.6 Compactação com BDD 9

3.1 Overview de uma Blockchain, adaptado de [28] 14

3.2 Cadeia de blocos 16

4.1 Modelagem formal de blockchain 18

4.2 Conexão de um nó à rede P2P 19

4.3 Autômato de um nó da rede P2P 20

4.4 Autômato de transação 20

4.5 Autômato de blocos 21

Lista de Tabelas

5.1 Configurações das blockchains 35

5.2 Resultados no Experimento 1 36

5.3 Resultados obtidos no Experimento 2 37

C.1 Tempos de execução do Experimento 1 53

C.2 Tempos de verificação do Experimento 1 54

C.3 Tempos de execução do Experimento 2 54

C.4 Tempos de verificação do Experimento 2 55

Sumário

1 Introdução **1**

1.1 Objetivos e contribuições 2

1.2 Considerações quanto à implementação do trabalho 3

1.3 Organização do trabalho 3

2 Fundamentação Teórica **4**

2.1 *Computation Tree Logic* (CTL) 4

2.2 Verificação de modelos e o nuXmv 7

2.3 *Binary Decision Diagram* (BDD) 8

2.4 *Bounded Model Checking* e o MiniSAT 10

2.5 Trabalhos relacionados 11

3 O Blockchain **13**

3.1 O protocolo 13

3.1.1 Transações 14

3.1.2 Blocos e a cadeia de blocos 15

3.1.3 Prova de trabalho e mineração de blocos 16

3.1.4 Solução de conflitos na Blockchain 17

4 Modelagem formal da Blockchain **18**

4.1 Autômatos 18

4.1.1 Rede *peer-to-peer* 19

4.1.2 Transações 19

4.1.3 Blocos 20

4.2 Modelagem no nuXmv 21

4.2.1 Módulo createNode 21

4.2.2 Módulo createUser 23

4.2.3 Módulo createTransaction 23

4.2.4 Módulo createBlock 25

4.2.5 Módulo updateNode 26

4.2.6 Módulo main 27

4.3 Compilador de código nuXmv 31

5 Exemplos de uso 35

5.1 Experimento 1 36

5.2 Experimento 2 37

5.3 Observações gerais 37

6 Conclusões e trabalhos futuros 39

Referências 41

Apêndice A – Modelagem dos elementos da Blockchain no nuXmv 44

Apêndice B – Código fonte do gerador de instâncias Blockchain 47

Apêndice C – Tempos de execução e de verificação dos experimentos 53

Capítulo 1

Introdução

Em 2008 Satoshi Nakamoto propôs um sistema de transações eletrônicas sem a necessidade de utilização de outras entidades que não as envolvidas diretamente nas transações [28] e, para provar este conceito, criou a moeda eletrônica chamada Bitcoin. O Bitcoin iniciou sua circulação no início de 2009 e, dez anos depois, mais de dezessete milhões e setecentos mil bitcoins já circulam na rede, com um valor de mercado estimado em quase duzentos e dez bilhões de dólares[1].

Embora o Bitcoin tenha se tornado bastante valioso do ponto de vista financeiro, sua tecnologia de base, a Blockchain, tem o potencial de ser ainda mais importante, sendo considerada pelos analistas de todo o mundo a invenção de tecnologia da informação mais disruptiva desde a Internet. Apesar de muitas aplicações diferentes do Blockchain já fazerem parte dos produtos e serviços disponíveis, já houve uma série de episódios de roubo de algumas Blockchains, como o DAO Attack, que roubou mais de 3,6 milhões de dólares de *smart contracts* da blockchain do Ethereum [39], ou o caso envolvendo a Ethereum Classic, em que um invasor conseguiu obter controle de mais da metade do poder computacional daquela rede, utilizando este poder para reescrever o histórico de transações [31]. De acordo com Orcutt [31], desde 2017 aproximadamente dois bilhões de dólares em valor de criptomoedas já foram roubados em todo o mundo.

Diante disto, evidencia-se a necessidade de buscar formas de garantir a segurança e confiabilidade da Blockchain, e não apenas de suas aplicações, na medida em que, se a tecnologia de base não possuir confiabilidade, todas as demais aplicações que se façam desta poderão estar sob risco.

A verificação de modelos, conhecida pelo termo em inglês *model checking* é uma

[1] De acordo com o CoinMarketCap: `https://coinmarketcap.com`, acessado em 27/06/2019.

maneira automática de realizar a verificação de sistemas através da criação de uma representação formal de alto nível do modelo e das especificações a serem verificadas. Esta é uma forma bastante eficaz de encontrar erros antes que eles causem problemas e danos. Por este motivo, se aplicada à Blockchain, a verificação de modelos pode auxiliar a constatar e prevenir impactos negativos e prejuízos financeiros.

Verificadores simbólicos de modelos, ou *model checkers* em inglês, por sua vez, são ferramentas de verificação de modelos implementadas usando um arcabouço lógico para responder a questões sobre sistemas complexos. Sua base formal os torna muito úteis para a indústria em virtude, especialmente, de sua capacidade de apresentar contraexemplos quando a resposta é negativa [14].

1.1 Objetivos e contribuições

Neste contexto, este trabalho tem como objetivo apresentar a formalização de modelos de Blockchain personalizáveis, contendo suas principais propriedades, desde os nós da rede *peer-to-peer*, à criação e validação de blocos e transações, e seus autômatos detalhados. Para tal define-se uma base em autômatos do funcionamento de cada módulo, haja vista que até o presente momento não foi encontrado na literatura uma formalização dessas operações. Além disso, realizar-se-á a verificação desta utilizando a ferramenta de verificação de modelos nuXmv [8] e, ainda, apresentar-se-á, como ferramenta de auxílio, um compilador de código SMV, que permite gerar um modelo formal de Blockchain para ser verificado pelo nuXmv. Serão discutidas a forma de construção do modelo formal e sua semântica, comparando diferentes configurações de Blockchain e seus resultados de verificação formal.

Já existem diversos trabalhos descrevendo aplicações da Blockchain e propondo métodos e ferramentas para verificar a solidez e confiabilidade destes modelos aplicados [16,20,21]. Entretanto, a revisão da literatura não indicou nenhum outro trabalho cujo foco seja a verificação formal de um modelo genérico de Blockchain.

Esta é, portanto, a principal contribuição do presente trabalho: oferecer uma formalização de Blockchain mais completa e personalizável do que as existentes, e autômatos detalhando seu funcionamento, para que possam ser utilizados tanto para realizar a verificação formal de características e propriedades da tecnologia em questão, como para expandir e adaptar para Blockchains específicas, mantendo a correção e as características fundamentais do modelo original. Secundariamente, um compilador de configurações de

Blockchain em código SMV, de modo manual ou automático e randômico, poderá, fornecer de forma ágil e simples, diferentes modelos prontos para serem verificados e adaptados segundo as necessidades do usuário.

1.2 Considerações quanto à implementação do trabalho

Dados os objetivos acima expressos, este trabalho não se proporá realizar a modelagem formal de características e propriedades que envolvam a criptografia, por entender que isto foge aos objetivos desta pesquisa, ao mesmo tempo em que as ferramentas e formatos de criptografia utilizados em Blockchain já foram exaustivamente avaliados.

1.3 Organização do trabalho

Este trabalho está estruturado da seguinte forma: o Capítulo 2 destina-se à fundamentação teórica, contendo os conceitos e definições utilizados na pesquisa, com destaque para a verificação de modelos e a ferramenta utilizada, além de uma revisão da literatura, apresentando os principais trabalhos relacionados e suas abordagens e diferenças em relação a este.

O Capítulo 3 apresenta os conceitos relativos ao funcionamento e à implementação de uma cadeia de blocos, enquanto o Capítulo 4 destina-se à modelagem formal das características e propriedades referentes aos conceitos explicitados no capítulo anterior em linguagem SMV. Além disto, apresenta um compilador simples de código SMV, escrito em linguagem Python, que permite criar, manual ou automaticamente, modelos de Blockchains para serem formalmente verificados.

No Capítulo 5 são analisados alguns exemplos de Blockchains gerados com o compilador e verificados de acordo com o modelo formal, ambos apresentados no capítulo anterior. Por fim, o Capítulo 6 descreve as conclusões e as indicações de possíveis melhorias e trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Este capítulo destina-se à fundamentação relativa aos conceitos basilares para o entendimento e a construção dos resultados deste trabalho. A seguir, esclarecimentos acerca da lógica CTL verificação de modelos e a ferramenta de verificação utilizada, além dos principais algoritmos de verificação disponíveis.

Por fim, uma análise dos trabalhos relacionados é apresentada, no intuito de explicitar suas contribuições e diferenças em relação a este.

2.1 *Computation Tree Logic* (CTL)

A percepção do transcurso do tempo e a necessidade de representação formal de sentenças em função do tempo são as motivações fundamentais para a criação das chamadas lógicas temporais [37]. Dentre elas, a *Computation Tree Logic* (CTL), ou Lógica de Árvore de Computações, é uma lógica temporal modelada em uma estrutura de árvore na qual se verificam sentenças ligadas a estados, ou seja, representações da sequenciação e ramificação do tempo. Esta noção de estados em que uma dada proposição é verificada, propicia que uma mesma afirmação possa ser verdadeira em um dado instante de tempo e falsa em outro.

Segundo Baier & Katoen [5] a CTL é uma lógica temporal baseada na lógica proposicional, com uma noção discreta de tempo e apenas modalidades futuras, A sintaxe da CTL classifica as fórmulas em dois tipos: fórmulas de estado e fórmulas de caminho. Intuitivamente, as fórmulas de estados expressam uma propriedade acerca de um dado estado, enquanto as fórmulas de caminho expressam as propriedades temporais dos mesmos, por exemplo, uma sequência infinita de estados.

Pode-se interpretar as fórmulas CTL como um sistema de transições M, tal que, $M = \langle S, R \rangle$, onde S é um conjunto de estados e R, uma relação de transição, com $R \subseteq S \times S$. Assim, cada estado poderia ter ao menos um sucessor e, com isto, o sistema de transições pode ser representado como uma árvore.

A CTL utiliza operadores modais, aqui geralmente representados por **A** (necessário) e **E** (possível), e outros que tratam especificamente da relação temporal. Assim, as fórmulas são compostas por pares do tipo **PS**, sendo P um dos operadores de caminho (*path*) já conhecidos (E ou A) e S um dos seguintes operadores de estados (*state*) específicos:

X - Ne**X**t: no próximo estado,

U - **U**ntil: até que,

F - **F**inally: em algum estado futuro,

G - **G**lobally: em todos os estados futuros.

Desta forma, considerando proposições genéricas $\boldsymbol{\varphi}$ e $\boldsymbol{\psi}$, um conjunto de fórmulas pode ser, por exemplo:

1. **AX**$\boldsymbol{\varphi}$: φ vale em todos (A) os próximos estados (X);
2. **EF**$\boldsymbol{\varphi}$: existe (E) algum estado futuro (F) em que φ vale;
3. **A**$\boldsymbol{\varphi}$**U**$\boldsymbol{\psi}$: para todos os estados (A) vale φ ao menos até que (U) valha ψ;
4. **EG**$\boldsymbol{\varphi}$: existe (E) um estado em que a partir dele φ vale em todo um caminho subsequente (G).

A Figura 2.1, abaixo, explicita visualmente um modelo que satisfaz a primeira fórmula supracitada. A partir de um dado estado, destacado em amarelo, em todos os estados do nível seguinte da árvore, vale φ.

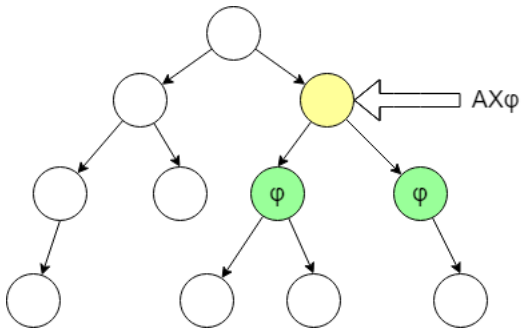

Figura 2.1: Representação da fórmula CTL AXφ

A Figura 2.2 representa um modelo para a fórmula CTL EFφ na árvore, indicando que em algum estado futuro vale EFφ.

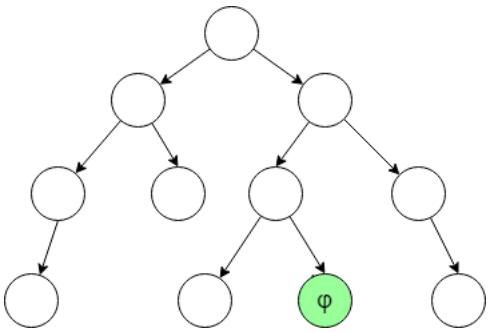

Figura 2.2: Representação da fórmula CTL EFφ

A fórmula 3, supracitada, afirma que AφUψ, ou seja, para todos os estados vale φ, ao menos até que valha ψ. Isto está representado no modelo da Figura 2.3, a seguir:

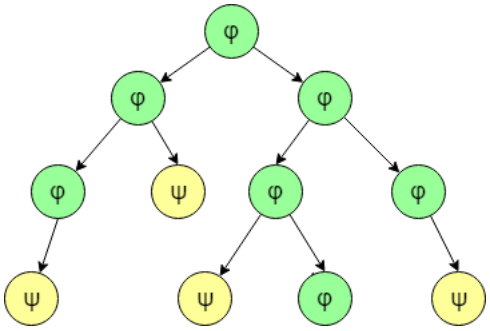

Figura 2.3: Representação da fórmula CTL AφUψ

Por fim, a Figura 2.4 apresenta um modelo para a fórmula 4, que afirma que existe um estado a partir do qual φ vale para todo um caminho subsequente.

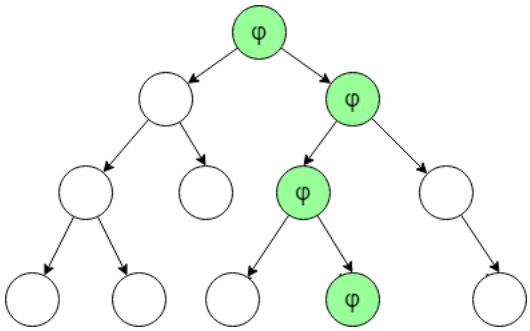

Figura 2.4: Representação da fórmula CTL EGφ

Com isto, uma interpretação possível a partir das noções de estados e de transições

entre estados é a de execução, ou seja, uma sequência finita ou infinita de transições. Por permitir a formalização desta abstração de execuções de programas, a CTL é muito utilizada na verificação de modelos computacionais e, por conseguinte, nos chamados *model checkers*, ou verificadores de modelos.

2.2 Verificação de modelos e o nuXmv

A verificação de modelos (*model checking*, em inglês) é uma alternativa automática para realizar a checagem de sistemas de estados (também chamados de máquinas de estados ou, simplesmente, autômatos) através da criação de uma representação formal de alto nível do modelo e das especificações a serem verificadas. Os algoritmos de *model checking* utilizam uma linguagem lógica temporal para realizar as verificações e retornam um valor *booleano* em resposta a uma dada "pergunta" em relação ao modelo verificado.

Ferramentas de verificação de modelos são muito importantes para localizar e identificar diversos tipos de potenciais problemas, em geral difíceis de encontrar em testes usuais de sistemas já construídos, uma vez que, usando *model checking*, se verifica o modelo formal, as regras e especificações como devem ser, e não como foram implementadas. Problemas de concorrência e *deadlocks* são bons exemplos de casos em que a verificação de modelos pode ser bastante útil.

Existem diversos *model checkers* implementados utilizando técnicas diferentes, baseadas em linguagens específicas e que atendem, portanto, a demandas particulares. Dentre eles, podemos citar o PRISM [22], LTSMin [7], nuXmv [8], TAPAs [9], UPPAAL [6] e ROMEO [24], sendo que apenas os quatro últimos fornecem contraexemplos em caso de o resultado de uma validação lógica ser FALSO.

Neste trabalho optou-se por utilizar a ferramenta nuXmv em virtude de ser gratuita, leve, de ter portabilidade em relação aos sistemas operacionais e por conter nativamente os recursos necessários a execução da pesquisa; também pela simplicidade da linguagem e consequente facilidade de aprendizagem, por ser capaz de analisar sistemas de estados infinitos, por sua expressividade e pela documentação disponível.

De acordo com Cavada et al [10], o nuXmv é um verificador de modelos simbólico que permite analisar sistemas síncronos de estados finitos e também de estados infinitos. Ele herda e estende as funcionalidades do NuSmv [11], que combina componentes de verificação de modelos baseado em diagramas de decisão binária (BDD) e baseado em SAT, incluindo dois SAT-Solvers: Minisat e o ZChaff, além de outros componentes também

usados nos algoritmos de verificação de modelo delimitado (*Bounded Model Checking*, ou BMC).

Embora o nuXmv suporte a lógica CTL* [15], neste trabalho utiliza-se apenas a lógica CTL, descrita acima, em virtude da característica transição de estados em estrutura de árvore, necessária à modelagem formal da Blockchain.

O nuXmv tem sido usado para diversas aplicações[1], como em projetos de desenvolvimento tecnológico aeroespacial, de certificações de segurança de software, entre outros, além de ser também utilizado por outras ferramentas de *model checking* que atendem a demandas mais específicas[2], justamente por sua eficiência na verificação dos modelos, em especial em relação à satisfatibilidade, mesmo em autômatos de estados infinitos.

Além disso importante ressaltar que a estrutura de construção do código em módulos no nuXmv se adequa naturalmente ao protocolo da Blockchain, possibilitando-se modelar as principais entidades da Blockchain (transações, blocos, nós, e outros) e suas propriedades, cada qual descrita em um módulo SMV, conforme se poderá verificar na seção 4.2 deste trabalho. Isso aumenta a legibilidade do modelo e, principalmente, simplifica a rastreabilidade a partir de inferências.

2.3 *Binary Decision Diagram* (BDD)

O Diagrama de Decisão Binária, ou BDD (na sigla em inglês) é uma estrutura de dados criada com a motivação de representar funções booleanas [25]. BDDs são uma generalização de árvores de decisão binárias, que, por sua vez, nada mais são do que uma compactação de árvores binárias [36].

Uma árvore binária (não-vazia) T é uma estrutura de dados caracterizada por um conjunto de elementos, chamados nós, de modo que há um nó especial r, chamado raiz, e os demais podem ser divididos em dois subgrupos, chamados *subárvore esquerda de* r e *subárvore direita de* r, os quais também são árvores binárias. A figura 2.5 abaixo apresenta um exemplo simples de árvore binária.

No exemplo da figura 2.5, os nós c, f, g e h são chamados *folhas* ou *terminais*, porque não há outros nós abaixo destes. Uma árvore de decisão binária é uma árvore binária em que os nós armazenam regras, enquanto os resultados ou decisões a serem tomadas estão

[1]Sobre projetos relacionados, ver: `https://es-static.fbk.eu/tools/nuXmv/index.php?n=Projects.Home`

[2]Ver `https://es-static.fbk.eu/tools/nuXmv/index.php?n=Extenstions.Home`

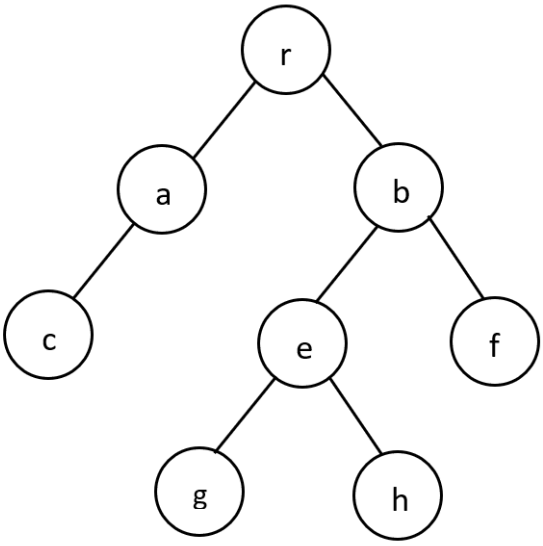

Figura 2.5: Exemplo de árvore binária

nos nós-folha. Desta forma, computacionalmente pode-se percorrer a árvore e, de acordo com as regras, seguir por um dado caminho até que se chegue a um nó-folha contendo uma decisão específica.

Uma vez que se percorra este caminho desde a raiz até as folhas ou terminais, pode-se eliminar da árvore de decisão binária as funções e resultados duplicados, para chegar-se a uma estrutura mais otimizada. Esta estrutura recebe o nome de Diagrama de Decisão Binária, ou BDD, e está ilustrada na Figura 2.6 a seguir.

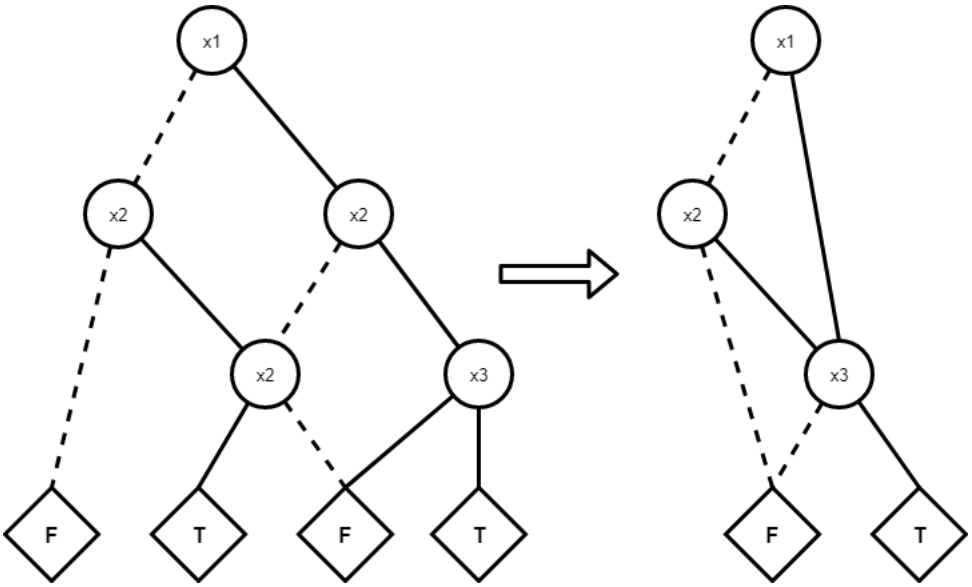

Figura 2.6: Compactação com BDD

O nuXmv utiliza BDDs como compactadores de modelos booleanos em uma abor-

dagem chamada ALLSMT [23], que particiona a abstração do sistema que está sendo verificado em uma combinação de abstrações menores, descartando a cada passo os contraexemplos que não servem para o modelo [8].

2.4 *Bounded Model Checking* e o MiniSAT

Bounded Model Checking (BMC) é uma técnica de verificação de modelos criada por Biere et al. [4], cujo objetivo é buscar contraexemplos em subpartes finitas dos caminhos possíveis de um modelo, limitadas por um inteiro k. Caso não sejam encontrados erros nesta execução, o inteiro k é incrementado e a verificação é reexecutada. Isto se repete até que um erro seja encontrado, a execução esgote os recursos computacionais disponíveis ou que um limite já conhecido seja atingido. Desta forma, um problema de BMC pode ser convertido em um problema de satisfatibilidade proposicional, e resolvido por um SAT-Solver de maneira eficiente, ou seja, utilizando uma quantidade razoável de recursos computacionais e tempo.

Com o nuXmv é possível utilizar dois algoritmos: Zchaff [34] e MiniSAT [17], ambos incrementais. Segundo nuXmv [8] estes algoritmos exploram o fato de que problemas de satisfatibilidade gerados por um dado problema de *Bounded Model Checking*(BMC) [4] em geral compartilham subpartes em comum. Desta forma, informações obtidas durante a solução de um dado problema podem ser utilizados para a solução dos outros. É a técnica conhecida como *Conflict Learning*. Com isto pode-se usar algoritmos incrementais, que normalmente são mais rápidos, mas exigem um SAT *solver* que possua interfaces para o uso da técnica BMC e, até o momento, Zchaff e MiniSAT são os dois que oferecem estas interfaces no nuXmv [8].

O Zchaff [34] é a implementação em *software* mais conhecida e utilizada do algoritmo Chaff [27], tendo sucesso na solução de problemas com mais de um milhão de variáveis e dez milhões de cláusulas. O MiniSAT [17], por sua vez, é um SAT-solver de código aberto criado para ser pequeno, completo e eficiente, e é o resultado da simplificação de outros dois algoritmos (SATZOO e SATNIK), dos mesmos autores, sem sacrificar a eficiência. A primeira implementação do MiniSAT tinha menos de 600 linhas de código. Os procedimentos de propagação do MiniSAT, assim como os do ZChaff, são amplamente inspirados no algoritmo Chaff.

A possibilidade de uso da técnica BMC é particularmente importante neste trabalho, pois o modelo apresentado utiliza variáveis que podem conter transições infinitas de es-

tados e, assim, algumas verificações de modelos podem tornar-se grandes e lentas demais para serem eficientemente resolvidas por algoritmos que utilizam apenas *Binary Decision Diagram* (BDD) que requerem muita capacidade computacional disponível à medida em que o número de estados cresce muito. Neste cenário é muito útil o uso de técnicas de BMC.

O algoritmo de BMC utilizado neste trabalho foi o MiniSAT, que mostrou-se mais capaz de resolver os problemas modelados do que o ZChaff.

2.5 Trabalhos relacionados

Esta seção destina-se à debater os principais trabalhos relacionados e suas diferenças em relação a este.

Foram escolhidas as bases Scopus e Springer Link como motores de busca e, de acordo com as especificidades de cada uma, formatou-se uma *string* de busca que compreendesse a palavra-chave principal, "*Blockchain*", além de "*model checking*" ou "*formal verification*". Inicialmente também foi utilizado o motor de busca Google Scholar, mas este retornou uma grande quantidade de falsos positivos, enquanto os resultados relevantes foram também indexados pelos motores supracitados e, por este motivo, este motor foi abandonado.

Destaca-se como o principal trabalho relacionado, o artigo de Nehai et al. [39], por aproximar-se deste em abordagem e ferramenta utilizada. Neste trabalho os autores utilizam o nuSmv [11], que é o precursor do nuXmv, para a modelagem formal de uma Blockchain e abordam mais profundamente a modelagem de *smart contracts*, que é seu foco principal. Embora haja diferença de objetivos, a modelagem de uma Blockchain utilizando o nuSmv contribuiu muito como referência para o presente trabalho. Entretanto, dado que o objetivo principal é a verificação formal de *smart contracts*, a Blockchain modelada é superficial, contendo apenas módulos simples de usuários, denominados "clientes" pelo autor, e transações, sem mencionar, por exemplo, os blocos ou nós.

Piriou & Dumas [32], também modelam o protocolo de uma Blockchain genérica, avaliando a consistência do modelo e a habilidade de refutar um ataque de *double-spending*. Entretanto, o trabalho realiza uma análise estocástica do modelo, e não uma verificação formal.

Nash et al. [29], Molina-Jimenez et al. [26], Abdellatif & Brousmiche [1], Amani et al. [3], Van Der Meyden [38] e Qu et al. [33], entre outros, abordam majoritariamente a

modelagem de *smart contracts* — programas computacionais desenvolvidos para reger a transferência de ativos entre duas ou mais partes — envidando esforços e utilizando-se de ferramentas de verificação formal, como o UPPAAL [6], e assistentes de prova, como o Isabelle [30], para realizar a verificação de vulnerabilidades, de liquidez, segurança e outros aspectos relacionados aos *smart contracts*, mas não apresentam modelagens formais ou verificações sobre as Blockchains em que os contratos são hospedados.

Fehnker & Chaudhary [18] modelam e verificam um ataque à rede Bitcoin, forçando um *fork* na rede, mas não apresentam uma modelagem formal da respectiva Blockchain. Em outro trabalho, Chaudhary et al. [12] criam uma modelagem formal de Blockchain contendo transações, blocos e nós, trazendo ainda uma modelagem de entidades maliciosas. Este modelo é construído com o uso do UPPAAL [6], que é uma ferramenta de modelagem e verificação não-determinística. Setzer [35], por sua vez, utiliza o assistente de provas Agda [2] para construir duas modelagens do Bitcoin, contendo os conceitos principais. Entretanto, a ferramenta não proporciona uma verificação formal de propriedades e ainda está em aberto, de acordo com o autor, como provar características como unicidade de identificador de transações.

Desta forma, pode-se constatar que, de acordo com a revisão da literatura, há trabalhos que propõem a modelagem formal de alguns aspectos da Blockchain, utilizando ferramentas de verificação não-determinísticas, e ainda trabalhos que abordam e verificam diversos aspectos relativos aos *smart contracts*. Não foram encontrados, no entanto, trabalhos cuja proposta seja a construção de um modelo formal de Blockchain genérica utilizando ferramentas determinísticas de verificação de modelos. Este é, precisamente, o principal objetivo do presente trabalho, que abordará os principais elementos da Blockchain, construído em uma linguagem de fácil assimilação e alta legibilidade, produzindo o modelo formal de Blockchain mais completo e expansível até o momento.

Capítulo 3

O Blockchain

Este capítulo destina-se à apresentação do conceito de Blockchain e suas principais características e propriedades. Será explicitado o funcionamento da rede *peer-to-peer*, o que são e como são realizadas transações, formação e encadeamento de blocos, prova de trabalho e, por fim, a solução de conflitos na Blockchain.

3.1 O protocolo

O conceito de Blockchain (cadeia de blocos, em Inglês) foi criado pelo pseudônimo Satoshi Nakamoto em 2008, com a publicação do artigo "*Bitcoin: A peer-to-peer eletronic cash system*" [28]. Embora o artigo, não diferencie o conceito de Blockchain do de Bitcoin, é importante estabelecer estas diferenças. De modo simples, o conceito de encadeamento de blocos foi desenvolvido e aplicado por Nakamoto na criação da moeda eletrônica que ele chamou Bitcoin.

Uma Blockchain é uma base de dados de transações, armazenadas em blocos encadeados e distribuídos por todos os nós em uma rede *peer-to-peer*. Este conceito é, na verdade, o resultado da utilização de outros conceitos já bastante conhecidos de criptografia para conceber um meio não rastreável de realizar transferências, originalmente de dinheiro, eletronicamente.

Já em 1982 Chaum [13] propôs um modelo de assinatura "cega" baseada em criptografia, comparando-o a um envelope de papel com uma folha de papel carbono dentro. Quando alguém usasse uma caneta para fazer uma assinatura do lado de fora do envelope, produziria também uma assinatura idêntica do lado de dentro do mesmo. Sua proposta buscava atender a algumas propriedades, dentre elas: 1) impedir o controle de

transações entre indivíduos por terceiros; 2) que os indivíduos sejam capazes de provar suas identidades e verificar pagamentos.

A Figura 3.1 abaixo apresenta uma visão geral da Blockchain, contendo usuários, que interagem em por meio de transações (Tx) enviando ou recebendo valores. Estas transações são verificadas e agrupadas em blocos que, por sua vez, contem o *hash* do bloco anterior, criando, assim, uma cadeia de blocos. De forma geral, estes são os principais elementos da Blockchain, e serão melhor explicados nas sessões subsequentes.

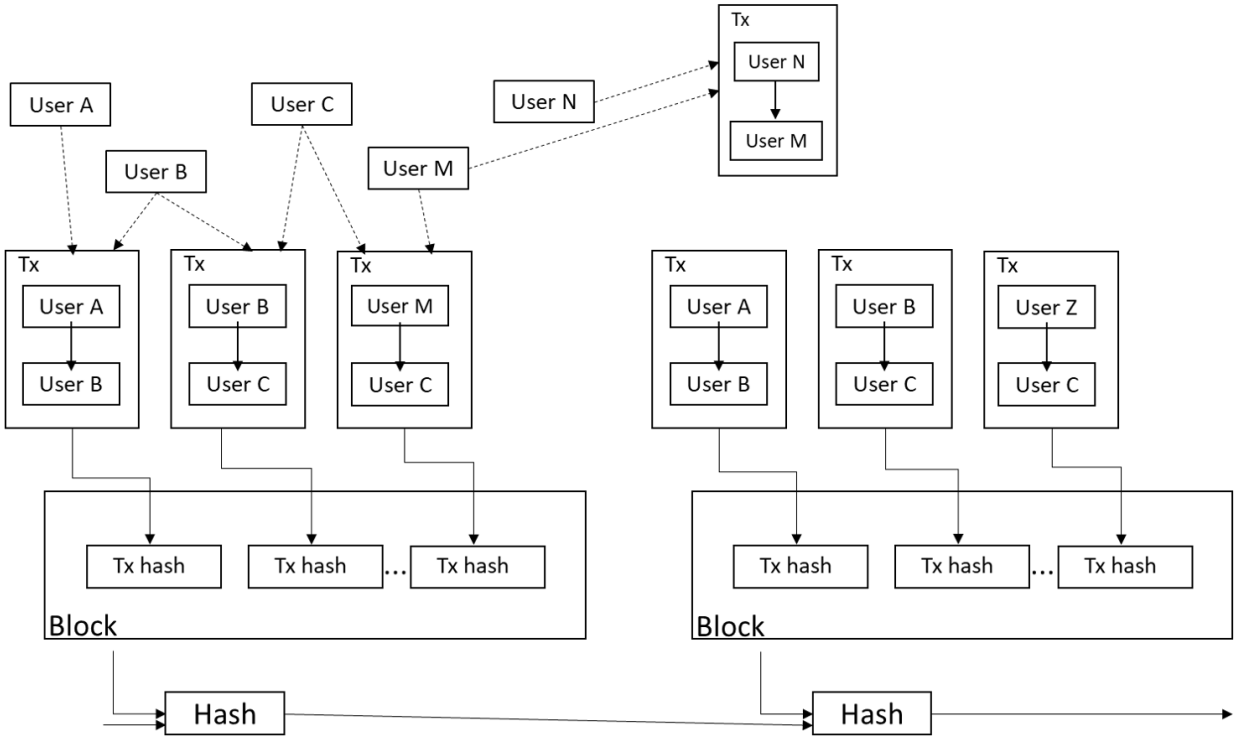

Figura 3.1: Overview de uma Blockchain, adaptado de [28]

3.1.1 Transações

Valendo-se desse conceito, Nakamoto [28] definiu uma transação de moeda eletrônica como uma cadeia de assinaturas digitais, em que o remetente transfere moedas que tenham sido a ele transferidas anteriormente, simplesmente assinando um *hash* desta transação anterior adicionado da chave pública do destinatário. Desta forma, apenas o destinatário pode, com sua chave privada, assinar esta nova transação quando quiser transferir o valor recebido para outrem, o que atinge o segundo objetivo definido por Chaum [13] e mencionado anteriormente. Cada transação é composta de entradas e saídas, comumente chamadas de *txin* e *txout*, respectivamente. A *txin* necessariamente é uma (ou mais) transação anterior, mais precisamente, uma *txout* anterior. Desta maneira é que as transações são

encadeadas, para que um dado remetente somente transfira valores dos quais tenha sido o destinatário anteriormente.

3.1.2 Blocos e a cadeia de blocos

Uma simples cadeia de transações não garante o primeiro objetivo descrito por Chaum, acima, dado que ainda é necessário um mecanismo de controle de transações para evitar que uma mesma transação seja referenciada várias vezes como fonte de valores que poderiam, portanto, ser transferidos mais de uma vez para outras pessoas. A isto Nakamoto chamou "*double-spending*".

Para evitar o "*double-spending*" Nakamoto novamente valeu-se de conceitos já existentes no mundo da computação, como visto em Massias et al. [19], para definir um servidor de *timestamp* que construa *hashes* a partir de conjuntos de transações. Estes servidores de *timestamp* como o nome indica, encapsulam um dado conjunto de transações com uma informação de tempo única (timestamp!) em um *hash*. Estes *hashes* são os chamados blocos.

Três regras principais são estabelecidas na criação de um bloco: 1) em um dado bloco, todas as transações são consideradas como tendo sido feitas no mesmo instante, independentemente de quando elas efetivamente foram realizadas; 2) uma transação só pode ser incluída em um único bloco, e; 3) uma transação só é considerada verificada quando está inclusa em um bloco.

Além disso, todos os blocos, exceto o primeiro, possuem, entre outras informações, a identificação e o *timestamp* do bloco anterior, garantindo que um dado bloco sempre "saiba" que bloco é seu antecessor e que tenha sido criado após este, enfim, criando uma cadeia de blocos, como uma linha do tempo. Assim, dado que uma transação só é considerada verificada se estiver em um bloco e uma transação não pode estar em dois blocos, torna-se teoricamente impossível que haja "*double-spending*" com uma mesma transação duplicada. Entretanto o maior problema seria que duas transações diferentes referenciassem a mesma transação de origem como entrada de um novo pagamento. Para resolver este problema, Nakamoto propôs um modelo em que cada *txout* só pode ser referenciada como *txin* em uma nova transação uma única vez. Com isto, se uma transação possui como *txin* uma outra *txout* que já foi referenciada antes, a transação não é validada e, não é incluída em um bloco e, portanto, o "*double-spending*" é evitado.

Em resumo: um bloco é um *hash* contendo um conjunto de transações, um *timestamp*

único e informações sobre o bloco anterior.

3.1.3 Prova de trabalho e mineração de blocos

A grande dificuldade de implementação deste modelo de criação de blocos através de servidores de *timestamp* está na topologia da rede, que, como dissemos, é do tipo *peer-to-peer*, o que implica na necessidade de diversos servidores de *timestamp* atuando simultaneamente, portanto, criando blocos que poderiam, em teoria, conter o mesmo *timestamp*.

Para implementar servidores de *timestamp* distribuídos em uma rede P2P, Nakamoto [28] utilizou o conceito de "prova de trabalho", que consiste na busca (computacional) de um *hash* que represente um determinado conjunto de informações, por exemplo em SHA-256, e cujo início seja um determinado número de zeros, chamado *nonce*. Este processo de agrupamento de transações, *timestamping*, vinculação ao bloco anterior e solução da prova de trabalho é realizado por um nó e é o que se conhece por *mineração*.

Desta forma, os nós, que atuam como servidores de *timestamp*, precisam despender esforço computacional para encontrar não apenas um *hash* genérico que corresponda às transações, mas um que contenha todas as informações e atenda ao *nonce* requerido, o que torna consideravelmente mais difícil a criação de blocos na rede. A Figura 3.2 abaixo representa o conteúdo de cada bloco em cadeia.

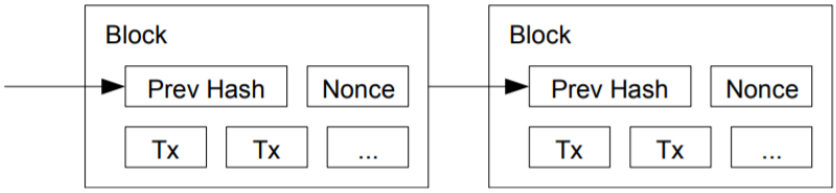

Figura 3.2: Cadeia de blocos

Uma vez que um dado nó da rede consiga resolver a prova de trabalho e, portanto, minerar um bloco válido, este novo bloco é espalhado pela rede, que está estruturada de forma que cada nó é capaz de verificar se o novo bloco é válido e se está corretamente encadeado com o último bloco conhecido por aquele nó. Caso o nó consiga validar o novo bloco ele o inclui como parte da cadeia e, caso contrário, ele o descarta, assim com todos os demais nós, de modo que a própria rede, em regime majoritário, decide que cadeia é válida ou não, baseada na premissa de que a maior cadeia de blocos completamente validada é a que deve ser mantida.

Constantemente é verificado o tempo médio de criação de blocos, ou seja, quanto tempo, em média, um nó da rede leva para conseguir encontrar um novo bloco. O tempo médio padrão que se busca manter na rede do Bitcoin é de 10 minutos. Entretanto, com a entrada de nós na rede e o aumento do poder de processamento dos nós, pode tornar-se cada vez mais rápido resolver a prova de trabalho e, por este motivo, o *nonce* é regularmente ajustado em função da capacidade total de processamento da rede, de forma que se mantenha a média de 10 minutos para a solução das provas de trabalho.

3.1.4 Solução de conflitos na Blockchain

Pode ocorrer, especialmente em uma rede de proporções gigantescas, que dois nós consigam resolver a prova de trabalho e criar blocos diferentes simultaneamente, a partir de um mesmo bloco anterior.

De forma prática, imaginemos que o último bloco conhecido pela rede seja o de id 3. Se dois nós da rede conseguirem criar blocos diferentes, digamos 4-A e 4-B, tendo o bloco 3 como anterior, teremos a chamada ramificação da rede, ou seja, haverá dois ramos da rede a partir do bloco 3 e, neste ponto no tempo, não será possível resolver este conflito.

Entretanto, em média a cada 10 minutos um novo bloco é criado e espalhado para a rede. Este novo bloco, que teria id 5, em sua criação precisaria utilizar um, e apenas um, bloco anterior como base. Desta maneira, ao ser espalhado na rede, ele seria conectado a apenas um dos ramos da rede, por exemplo, o de id 4-A. No momento em que um nó receba o novo bloco, de id 5, ligado ao bloco 4-A, este ramo da cadeia torna-se maior do que o outro, que contém o bloco 4-B e, por este motivo, o bloco 4-B seria invalidado e descartado da cadeia, dirimindo-se, assim, o conflito de ramificação.

Em qualquer tempo, em qualquer nó, o maior ramo da cadeia vence e é mantido, em detrimento dos demais. Esta simples regra da rede previne e soluciona as questões de conflito entre blocos.

Capítulo 4

Modelagem formal da Blockchain

Este capítulo destina-se a apresentar a modelagem da Blockchain, através de autômatos que buscam explicitar seu funcionamento, e também de uma modelagem utilizando a linguagem da ferramenta nuXmv [8].

4.1 Autômatos

Como forma de ilustrar graficamente a modelagem adotada, utilizam-se autômatos finitos (Máquinas de Mealy), que podem ser definidos como um conjunto finito de estados e um conjunto de transações que ocorrem a partir desses estados onde cada transição denota a execução de uma operação na Blockchain. A figura 4.1 abaixo apresenta os relacionamentos entre as principais entidades da blockchain, apontando os módulos desenvolvidos a partir da modelagem formal dos autômatos em linguagem SMV, apresentada na seção 4.2 a seguir.

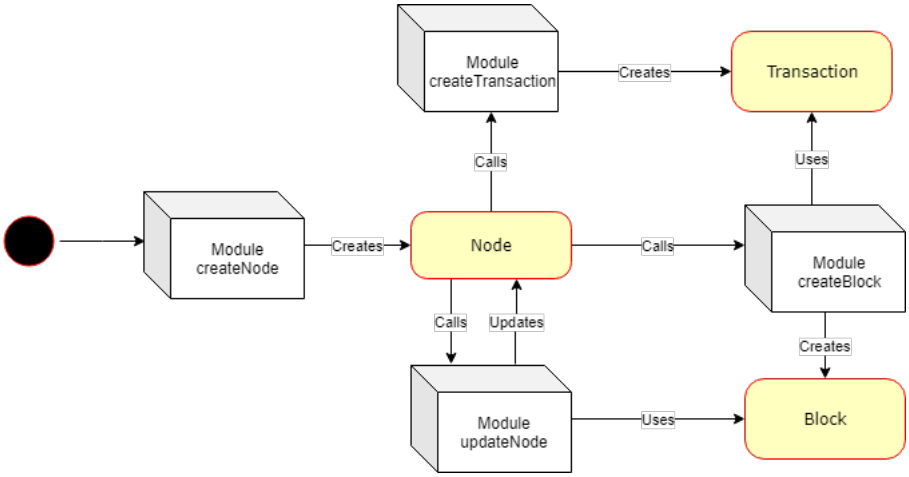

Figura 4.1: Modelagem formal de blockchain

As subseções a seguir apresentam quatro autômatos finitos que descrevem o funcionamento da Blockchain e que serão utilizados como base para a modelagem no nuXmv.

4.1.1 Rede *peer-to-peer*

A Blockchain do Bitcoin se estabelece em uma rede *peer-to-peer*, em que cada computador conectado a ela é um nó. No presente trabalho, optou-se por modelar formalmente apenas um nó genérico, em virtude de a variedade de nós não representar característica suficientemente distintiva para o conceito principal (Blockchain) do ponto de vista da modelagem formal. A Figura 4.2 apresenta a conexão de um nó à rede *peer-to-peer*. Este autômato está modelado formalmente no módulo createNode, na seção 4.2.1 a seguir.

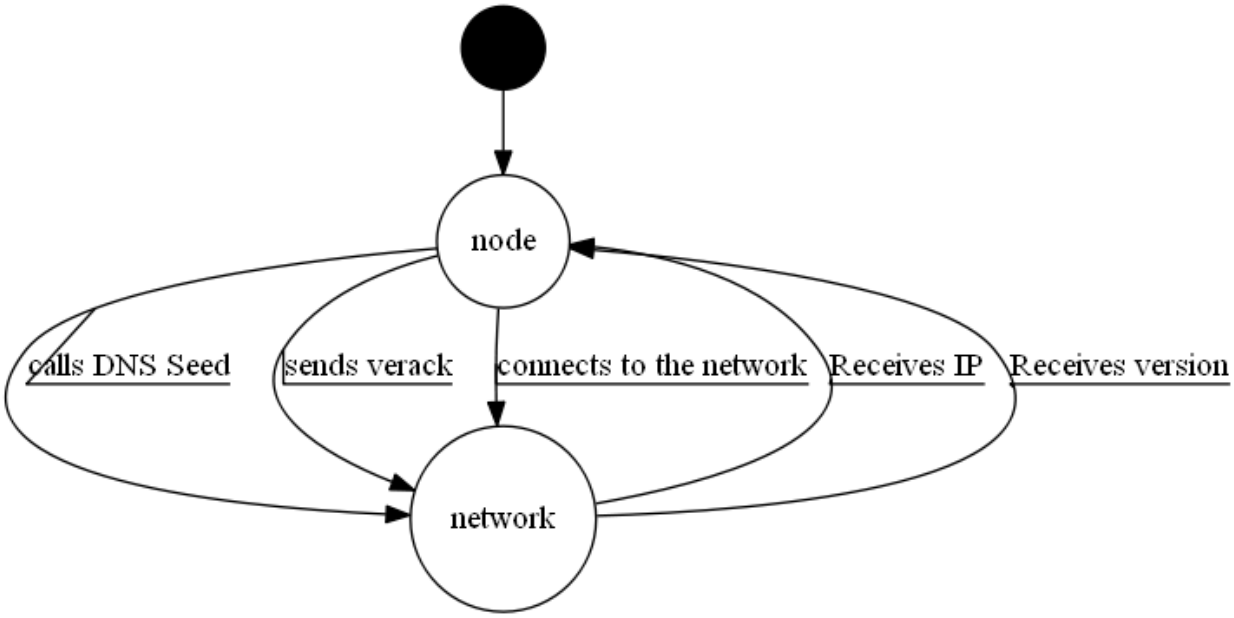

Figura 4.2: Conexão de um nó à rede P2P

Uma vez conectado à rede, um nó pode receber atualizações de outros nós com novos blocos, criar transações entre usuários, criar blocos de transações e transmitir os blocos criados. A Figura 4.3 apresenta este diagrama de estados, e suas implementações das relações estão separadas em diversos módulos na modelagem no nuXmv.

4.1.2 Transações

Como se pode ver na seção anterior, um nó pode criar transações entre usuários. Para que uma transação se efetive é preciso que o saldo do remetente seja igual ou maior do que o montante que se pretende transferir. Em caso verdadeiro, os saldos de remetente e

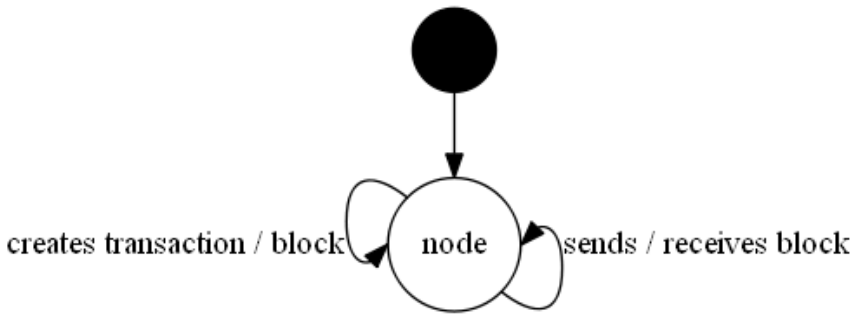

Figura 4.3: Autômato de um nó da rede P2P

destinatário são devidamente atualizados. A Figura 4.4 apresenta este fluxo e a seção 4.2.3 apresenta a implementação deste autômato em linguagem SMV.

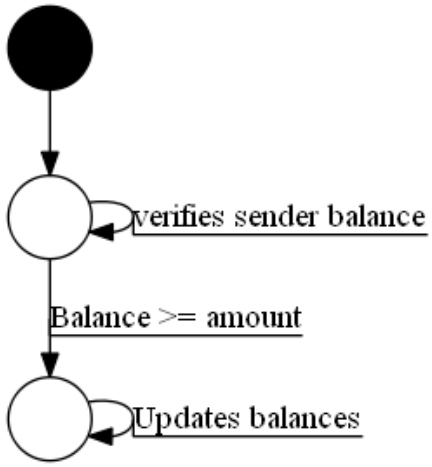

Figura 4.4: Autômato de transação

4.1.3 Blocos

Uma vez que haja transações não vinculadas a blocos, estes podem ser criados. Neste trabalho optou-se por criar modelos formais de blocos com um número fixo de transações, precisamente duas, apenas para simplificar a modelagem. Um número maior, ou mesmo um número variável de transações por bloco, como ocorre na Blockchain do Bitcoin, não implica em uma relevância maior de validação formal, enquanto torna a modelagem bastante mais extensa e complexa.

Outra característica da pesquisa que cabe ratificar refere-se às questões relativas à criptografia, que não são o foco deste trabalho, por considerar-se que a criptografia utilizada já foi amplamente verificada e suas funcionalidade e segurança exaustivamente demonstradas. Sendo assim, a modelagem formal da criação de um bloco resulta em verificar se as transações são válidas, agregá-las e vinculá-las ao ID do bloco, dado que

a mineração do mesmo - busca por um *hash* que atenda ao parâmetros de dificuldade - não é objeto desta pesquisa. A Figura 4.5 apresenta este autômato, que está modelado no nuXmv na seção 4.2.4.

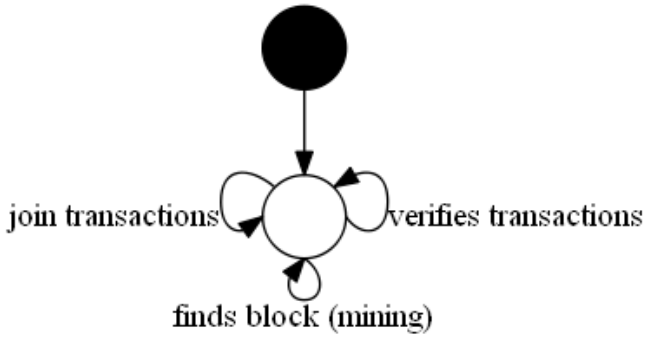

Figura 4.5: Autômato de blocos

4.2 Modelagem no nuXmv

A partir dos autômatos descritos, criou-se um modelo formal utilizando-se a linguagem de entrada do nuXmv, conforme o Apêndice A. Modelo foi dividido nos seguintes módulos:

- **createNode** - para criar os nós da rede *peer-to-peer*;
- **createUser** - destinado à criação dos usuários;
- **createTransaction** - que cria as transações entre usuários;
- **createBlock** - usado para criar blocos a partir de transações;
- **updateNode** - cuja função é atualizar um dado nó com um novo bloco, realizando a difusão dos blocos para a rede *peer-to-peer*;
- **main** - Módulo padrão do nuXmv, onde são instanciados os usuários, transações e blocos e, por fim, são realizadas as verificações semânticas e lógicas sobre o modelo.

4.2.1 Módulo createNode

Neste módulo apenas são definidos os estados possíveis e as transições entre estados de um dado nó que conecta-se à rede.

Baseado no autômato apresentado anteriormente na Figura 4.2, o módulo foi implementado da seguinte maneira:

```
MODULE createNode(nodeId)

  VAR
    IPSent: boolean;
    IPReceived: boolean;
    connectSeed: boolean;
    versionSent: boolean;
    verack: boolean;
    networkOK: boolean;
    lastBlockId: 0..10;

  ASSIGN
    init(IPSent) := TRUE;
    init(IPReceived) := FALSE;
    init(connectSeed) := FALSE;
    init(versionSent) := FALSE;
    init(verack) := FALSE;
    init(networkOK) := FALSE;
    init(lastBlockId) := 0;


    next(IPSent) := TRUE;
    next(IPReceived) := IPSent = TRUE ? TRUE : FALSE;
    next(connectSeed) := (IPSent = TRUE & IPReceived = TRUE) ? TRUE : FALSE;
    next(versionSent) := connectSeed = TRUE ? TRUE : FALSE;
    next(verack) := versionSent = TRUE ? TRUE : FALSE;
    next(networkOK) := verack = TRUE ? TRUE : FALSE;
```

Listing 1: Código do módulo createNode

As variáveis **IPSent, IPReceived, connectSeed, versionSent** e **verack** definem, respectivamente: se o IP de um servidor da rede foi enviado, se foi recebido pelo nó, se o nó se conectou a este servidor, se o servidor enviou a versão de software que deve ser baixada e, por último, se o nó recebeu esta informação.

Na seção ASSIGN pode-se observar que todos os estados iniciais das variáveis são FALSE e as transições seguintes podem variar entre TRUE e FALSE condicionadas em relação à variável anterior.

Por fim, ainda na seção ASSIGN, a variável lastBlockId é inicializada com zero, para indicar que este nó não conhece nenhum bloco.

4.2.2 Módulo createUser

Este é um módulo bastante simples, que recebe como entrada um ID de usuário e um saldo inicial, e armazena estes valores em um objeto com o nome especificado.

Definiu-se a variável *balance* para registrar o saldo inicial do usuário. Esta variável foi criada apenas para receber o parâmetro de entrada do módulo e torná-lo acessível posteriormente. Além desta, a variável *busy* é utilizada como variável de controle durante a execução, para evitar a concorrência na alteração de estados de uma mesma entidade.

A declaração TRANS é uma outra forma de designar uma transição de estados. Aqui ela é utilizada apenas para ser executada quando evocada após as transações, para atualizar os saldos.

A seção DEFINE, por sua vez, expressa o significado da transição supramencionada.

```
MODULE createUser(idUser, initBalance)

  VAR
    balance : 0..400;
    busy : boolean;

  ASSIGN
    init(balance) := initBalance;

  TRANS !update -> conserve ;

  DEFINE
    conserve := (next(balance) = balance);
```

Listing 2: Código do módulo createUser

4.2.3 Módulo createTransaction

Este módulo destina-se à criação de transações, ou seja, transferências de valores entre usuários. Neste sentido, cada transação registra um identificador, os IDs do remetente e do destinatário e o valor transacionado.

O ID do bloco (blockId), é uma FROZENVAR. Segundo Cavada et al. [8], FROZENVAR é uma "variável" que mantém seu estado inicial durante a execução da máquina de estados. Dado que não deve haver nenhuma mudança de estado, só é possível atribuir-lhes o estado inicial, e não os próximos, na seção ASSIGN. As FRONZENVARs comportam-se, portanto, como constantes durante a execução. O *blockId* expressará o bloco em que esta transação for incluída, no momento oportuno. Isto é particularmente importante porque uma transação não pode constar em mais de um bloco, o que poderia gerar erros como o "*double-spending*".

A seção DEFINE estabelece uma regra importante: uma transação só é considerada válida se o remetente possui, no mínimo, o valor estipulado na transação. Esta regra está expressa no atributo *validTransaction*, explicado abaixo, além de outros dois:

- **validtransaction**: com base no id do remetente, que deve ser passado na chamada do módulo, acessa a instância de objeto createUser correspondente (from.balance) e compara com o parâmetro *balance*, também passado na chamada do módulo;
- **newFromBalance**: apenas um totalizador para expressar o novo saldo esperado do remetente após esta transação;
- **newToBalance**: similar à anterior, expressa o novo saldo esperado do remetente após esta transação;.

Por fim, a seção TRANS define uma relação de transição de estado baseada na avaliação do atributo *validTransaction*, que atualiza os atributos *balance* dos dois usuários envolvidos em uma transação, caso ela seja válida, além de alterar a variável de controle *busy*.

- **next(from.balance)** - atualiza o saldo do remetente, com base no atributo *newFromBalance*;
- **next(to.balance)** - similar ao anterior, atualiza o saldo do destinatário com base no atributo *newToBalance*.

Segue abaixo a implementação, com base no autômato da Figura 4.4, anteriormente apresentada:

```
1 MODULE createTransaction(idTrans, from, to, amount)
2
```

```
3   FROZENVAR
4       blockId : 0..100;
5
6   DEFINE
7       validTransaction := !from.busy & !to.busy & from.balance >= amount;
8       newFromBalance := from.balance − amount;
9       newToBalance := to.balance + amount;
10
11  TRANS
12      validTransaction −> (next(from.balance)= newFromBalance) &
13                          (next(to.balance)=newToBalance) &
14                          (next(from.busy)=!from.busy) &
15                          (next(to.busy)=!to.busy);
```

Listing 3: Código do módulo createTransaction

4.2.4 Módulo createBlock

Este é o módulo destinado à agregação de transações em blocos. A implementação foi feita de modo que cada bloco agrega apenas duas transações, simplesmente porque a lógica para agregar mais transações seria a mesma, apenas com maior quantidade de código, o que, por uma questão de praticidade, foi evitado.

Como já visto anteriormente, o *id* do bloco é uma FROZENVAR, por manter-se sempre o mesmo.

A seção ASSIGN, além de associar o parâmetro recebido ao id, como já visto anteriormente em outros módulos, também atribui a cada transação contida no bloco o id dele na FROZENVAR "blockId" criada no módulo createTransaction, mas não iniciada. Esta atribuição está condicionada à validação do bloco em questão, ou seja, se o bloco for inválido, as transações não são atreladas a este. Cabe lembrar que, dado que o atributo *blockId* das transações é uma FROZENVAR, conforme mencionado anteriormente neste capítulo, esta atribuição é feita uma única vez, o que evita o erro de dupla vinculação de uma transação a dois ou mais blocos distintos.

O bloco DEFINE traz uma importante validação: o atributo *validBlock* recebe valor TRUE apenas se:

1. Todas as transações contidas nele possuírem um *id* diferente de zero, ou seja, válido;

2. todas as transações forem, individualmente, válidas.

Adjacentemente, definiu-se um *nonce*, apenas para simbolizar o nível de dificuldade da expressão da prova de trabalho, conforme mencionado no capítulo 3 anterior.

Além disso definiu-se também um *hash* como sendo a concatenação do *nonce*, dos *ids* das transações dentro do bloco e do *id* do bloco anterior. Embora não seja objeto deste trabalho o cálculo preciso dos *hashes*, decidiu-se por dar alguma semântica a estes apenas para auxiliar a legibilidade das informações utilizadas na formação do bloco e para aumentar a fidelidade à estrutura da blockchain.

Segue, abaixo, a implementação do módulo, de acordo com a Figura 4.5 mencionada anteriormente:

```
1  MODULE createBlock(idBloco, txa, txb)
2    FROZENVAR
3      id : 0..10;
4
5    ASSIGN
6      init(id) := idBloco;
7      init(txa.blockId) := validBlock ? id : 0;
8      init(txb.blockId) := validBlock ? id : 0;
9
10
11   DEFINE
12     validBlock := txa.validTransaction & txb.validTransaction;
13     nonce := signed word[32](190);
14     hash := validBlock ? (nonce :: signed word [32](txa.to) :: signed word [32](txb.to) ::
           signed word[32](previousBlock)): unsigned word[128](0);
```

Listing 4: Código do módulo createBlock

4.2.5 Módulo updateNode

Este módulo destina-se a implementar a transmissão de blocos pela rede *peer-to-peer*. Ele recebe como parâmetros de entrada um ID de nó e um ID de bloco, e verifica se aquele nó já conhece o bloco em questão. Dado que os blocos são encadeados sequencialmente, se um nó conhece um bloco n qualquer, então ele necessariamente conhece todos os *n-1* blocos anteriores.

Além disso, é precondição para atualização do nó, que o bloco em questão seja considerado válido, ou seja "block.validBlock" deve ser avaliado como TRUE. Se assim for, a variável "blockToUdate" receberá o valor do ID do bloco dado e, por fim, a variável "shouldUpdate" irá comparar o este ID com a variável "lastNodeBlock", que contém o ID do último bloco conhecido pelo nó em questão, e determinar se o nó deve ou não ser atualizado com o bloco dado.

O código abaixo implementa especificamente as relações denominadas "trasmite bloco" e "recebe bloco" do autômato apresentado na Figura 4.3.

```
1 MODULE updateNode(node, block)
2
3   DEFINE
4     lastNodeBlock := node.lastBlockId;
5     blockToUpdate := block.validBlock ? block.id : −1;
6     shouldUpdate := lastNodeBlock <= blockToUpdate;
7
8   ASSIGN
9     next(node.lastBlockId) := shouldUpdate ? lastNodeBlock : block.id;
```

Listing 5: Código do módulo updateNode

4.2.6 Módulo main

Neste módulo é onde, de fato, a implementação é verificada, pois é este o módulo responsável pela instanciação dos modelos criados nos módulos anteriores.

Na seção VAR apenas são criados os nós, usuários, transações e blocos, e atualizados os nós, conforme as regras explicitadas anteriormente.

A seção ASSIGN faz-se necessária em virtude da necessidade de evitar que em um mesmo estado houvesse duas ou mais atribuições simultâneas para um mesmo usuário. Por este motivo criou-se para cada usuário o atributo lógico *busy*, que funciona como uma variável de controle, impedindo a dupla atribuição.

A seção DEFINE apresenta dois grandes blocos de código:

- *block.previousBlock*: estes blocos de código, individuais para cada bloco, tem a função de determinar qual o bloco anterior ao bloco em questão, a partir de duas condições:

1. se o bloco em questão é válido, conforme as regras definidas no módulo *createBlock*;
2. dado que o bloco em questão seja válido, qual o último bloco válido antes dele.

- *user.verifyBalance*: calcula o saldo final do usuário em questão, com base na soma de valores de transações em que ele foi o destinatário e subtração dos valores de transações em que ele foi remetente. Isto é feito desta forma para ser fiel ao modelo teórico da *blockchain*, que não possui um registro de saldos, mas um cálculo, em tempo de execução dos saldos dos usuários com base nas transações em que eles são referenciados. Neste modelo formal, serve como um verificador do saldo anteriormente atualizado pela validação da transação, no módulo *createTransaction*.

```

MODULE main

  VAR

    node1 : createNode(self);
    node2 : createNode(self);
    node3 : createNode(self);
    node4 : createNode(self);

    user1 : createUser(self, 1);
    user2 : createUser(self, 10);
    user3 : createUser(self, 12);
    user4 : createUser(self, 35);
    user5 : createUser(self, 22);
    user6 : createUser(self, 2);

    tx1 : createTransaction(self, user1, user3, 10);
    tx2 : createTransaction(self, user4, user2, 19);
    tx3 : createTransaction(self, user5, user6, 11);
    tx4 : createTransaction(self, user2, user1, 6);
    tx5 : createTransaction(self, user4, user3, 12);
    tx6 : createTransaction(self, user2, user1, 4);
    tx7 : createTransaction(self, user6, user1, 1);
    tx8 : createTransaction(self, user3, user4, 7);

```

```
27
28    block1 : createBlock(1, tx3, tx5);
29    block2 : createBlock(2, tx1, tx4);
30    block3 : createBlock(3, tx2, tx6);
31    block4 : createBlock(4, tx7, tx8);
32
33    updateNode1 : updateNode(node1, block3);
34    updateNode2 : updateNode(node2, block4);
35    updateNode3 : updateNode(node3, block2);
36
37    ASSIGN
38
39      init(user1.busy) := TRUE;
40      init(user3.busy) := TRUE;
41
42  DEFINE
43
44    user1.update := tx1.validTransaction | tx4.validTransaction | tx6.validTransaction | tx7.
        validTransaction;
45    user2.update := tx2.validTransaction | tx4.validTransaction | tx6.validTransaction;
46    user3.update := tx1.validTransaction | tx5.validTransaction | tx8.validTransaction;
47    user4.update := tx2.validTransaction | tx5.validTransaction | tx8.validTransaction;
48    user5.update := tx3.validTransaction;
49    user6.update := tx3.validTransaction | tx7.validTransaction;
50
51    block1.previousBlock := block1.validBlock ? 0 : −1;
52    block2.previousBlock := case
53                    block2.validBlock : case
54                                  block1.validBlock : block1.id;
55                                esac;
56                    TRUE : −1;
57                  esac;
58    block3.previousBlock := case
59                    block3.validBlock : case
60                                  block2.validBlock : block2.id;
61                                  block1.validBlock : block1.id;
62                                esac;
63                    TRUE : −1;
```

```
64                esac;
65
66    block4.previousBlock := case
67                  block4.validBlock : case
68                              block3.validBlock : block3.id;
69                              block2.validBlock : block2.id;
70                              block1.validBlock : block1.id;
71                            esac;
72                  TRUE : −1;
73                esac;
74
75    user1.verifyBalance := user1.balance − (tx1.blockId != 0 ? tx1.amount : 0) +
76                      (tx4.blockId != 0 ? tx4.amount : 0) +
77                      (tx6.blockId != 0 ? tx6.amount : 0) +
78                      (tx7.blockId != 0 ? tx7.amount : 0);
79
80    user2.verifyBalance := user2.balance + (tx2.blockId != 0 ? tx2.amount : 0) −
81                      (tx4.blockId != 0 ? tx4.amount : 0) −
82                      (tx6.blockId != 0 ? tx6.amount : 0);
83
84    user3.verifyBalance := user3.balance + (tx1.blockId != 0 ? tx1.amount : 0) +
85                      (tx5.blockId != 0 ? tx5.amount : 0) −
86                      (tx8.blockId != 0 ? tx8.amount : 0);
87
88    user4.verifyBalance := user4.balance − (tx2.blockId != 0 ? tx2.amount : 0) −
89                      (tx5.blockId != 0 ? tx5.amount : 0) +
90                      (tx8.blockId != 0 ? tx8.amount : 0);
91
92    user5.verifyBalance := user5.balance + (tx3.blockId != 0 ? tx3.amount : 0);
93
94    user6.verifyBalance := user6.balance + (tx3.blockId != 0 ? tx3.amount : 0) −
95                      (tx7.blockId != 0 ? tx7.amount : 0);
```

Listing 6: Código do módulo main

4.3 Compilador de código nuXmv

Dada a necessidade de realizar verificações diversas que variam de acordo com o número e configuração das entidades do modelo (nós, usuários, transações e blocos), à medida em que são introduzidas entidades em maior número, a quantidade de verificações cresce vertiginosamente, tornando o código SMV bastante verboso e complexo. Em especial as verificações em *loop* tornam-se extensas rapidamente, em virtude das características da linguagem SMV, que não possuem funções otimizadas para tratar repetições e recursividade.

Por este motivo optou-se por criar um compilador de código SMV (Apêndice B), que receba como parâmetros de entrada a quantidade e natureza das entidades do modelo e retorne o código SMV já pronto para ser incluído no programa a ser executado pelo NuXmv.

Desta forma, o compilador é o responsável pela geração automática de código SMV de acordo com os parâmetros inseridos pelo usuário, ou automática e aleatoriamente pelo próprio programa.

O primeiro passo da execução do compilador de código solicita ao usuário que responda se deseja parametrizar manualmente a blockchain ou gerar uma blockchain aleatória de modo automático. No modo automático será solicitado um número inteiro como semente do algoritmo de aleatoriedade. Feito isto, serão parametrizados a quantidade de nós; a quantidade de usuários e o saldo de cada um deles; a quantidade, o remetente, o destinatário e o valor das transações; a quantidade de blocos e as transações envolvidas em cada um deles.

Cabe ressaltar que as quantidades de nós, usuários, transações e blocos produzem crescimento exponencial no tamanho do código SMV, implicando em que sua execução e verificações poderão demandar um volume considerável de recursos computacionais. Entretanto, nada impede que estes valores sejam tão altos quanto se deseje sem nenhum prejuízo, à exceção do tempo de execução.

O compilador de código está disponível em `https://github.com/brunoolimpio/blockchain-smv-model`. Uma vez executado, o compilador gera um arquivo que, no modo automático, possuirá um conteúdo semelhante ao que segue:

```
1 --- Automatic blockchain SMV code generator. ---
2
3 Do you want to automatically generate a blockchain based on randomic parameters? (yes/no)
```

```
    yes

Type an integer number to be used as the seed of ramdomness: 5681
The 7 requested nodes was sucessfully created.
You must create at least 2 users. The users are pairs like (id, balance).
Now you can create transactions. A transaction is a tuple like (from, to, balance)
16 transactions created.
Now it's time to create blocks
7 blocks created.

The blockchain generated has the below parameters:
-->7 nodes;
-->6 users;
-->16 transactions;
-->5 blocks;



MODULE main
VAR

   node1 : createNode(1);
   node2 : createNode(2);
   node3 : createNode(3);
   node4 : createNode(4);
   node5 : createNode(5);
   node6 : createNode(6);
   node7 : createNode(7);

   user1 : createUser(self,24);
   user2 : createUser(self,16);
   user3 : createUser(self,23);
   user4 : createUser(self,1);
   user5 : createUser(self,16);
   user6 : createUser(self,7);

   tx1 : createTransaction(self,user5,user6,4);
   tx2 : createTransaction(self,user5,user5,18);
```

```
41    tx3 : createTransaction(self,user1,user2,8);
42    tx4 : createTransaction(self,user2,user6,4);
43    tx5 : createTransaction(self,user4,user6,17);
44    tx6 : createTransaction(self,user6,user6,5);
45    tx7 : createTransaction(self,user4,user2,19);
46    tx8 : createTransaction(self,user2,user4,4);
47    tx9 : createTransaction(self,user3,user4,18);
48    tx10 : createTransaction(self,user2,user6,6);
49    tx11 : createTransaction(self,user6,user4,3);
50    tx12 : createTransaction(self,user2,user5,19);
51    tx13 : createTransaction(self,user4,user3,10);
52    tx14 : createTransaction(self,user1,user4,2);
53    tx15 : createTransaction(self,user2,user5,19);
54    tx16 : createTransaction(self,user2,user1,4);
55
56    block1 : createBlock(1,tx5,tx10);
57    block2 : createBlock(2,tx1,tx14);
58    block3 : createBlock(3,tx4,tx11);
59    block4 : createBlock(4,tx6,tx2);
60    block5 : createBlock(5,tx7,tx16);
61
62
63 ASSIGN
64    init(user1.busy) := TRUE;
65    init(user2.busy) := TRUE;
66    init(user3.busy) := TRUE;
67
68 DEFINE
69    user1.update := tx3.validTransaction | tx14.validTransaction | tx16.validTransaction;
70    user2.update := tx3.validTransaction | tx4.validTransaction | tx7.validTransaction | tx8.
          validTransaction | tx10.validTransaction | tx12.validTransaction | tx15.validTransaction
           | tx16.validTransaction;
71    user3.update := tx9.validTransaction | tx13.validTransaction;
72    user4.update := tx5.validTransaction | tx7.validTransaction | tx8.validTransaction | tx9.
          validTransaction | tx11.validTransaction | tx13.validTransaction | tx14.validTransaction
          ;
73    user5.update := tx1.validTransaction | tx2.validTransaction | tx12.validTransaction | tx15.
          validTransaction;
```

```
74    user6.update := tx1.validTransaction | tx4.validTransaction | tx5.validTransaction | tx6.
          validTransaction | tx10.validTransaction | tx11.validTransaction;
75
76 block1.previousBlock := block1.validBlock ? 0 : −1;
77 block2.previousBlock := case
78                                 block2.validBlock : case
79                                                     block1.validBlock : block1.id;
80                                                   esac;
81                                 TRUE : −1;
82                               esac;
83 block3.previousBlock := case
84                                 block3.validBlock : case
85                                                     block2.validBlock : block2.id;
86                                                     block1.validBlock : block1.id;
87                                                   esac;
88                                 TRUE : −1;
89                               esac;
90 block4.previousBlock := case
91                                 block4.validBlock : case
92                                                     block3.validBlock : block3.id;
93                                                     block2.validBlock : block2.id;
94                                                     block1.validBlock : block1.id;
95                                                   esac;
96                                 TRUE : −1;
97                               esac;
98 block5.previousBlock := case
99                                 block5.validBlock : case
100                                                    block4.validBlock : block4.id;
101                                                    block3.validBlock : block3.id;
102                                                    block2.validBlock : block2.id;
103                                                    block1.validBlock : block1.id;
104                                                  esac;
105                                TRUE : −1;
106                              esac;
```

Listing 7: Saída do compilador de código em modo automático

Capítulo 5

Exemplos de uso

Neste capítulo são apresentadas quatro configurações de blockchains geradas a partir do compilador de código e verificadas utilizando a modelagem formal apresentada nos capítulos anteriores. Estas blockchains serão utilizadas em dois experimentos, que verificarão diferentes propriedades do modelo. Busca-se aqui estabelecer as diferenças na execução das verificações e as respostas lógicas a cada exemplo de uso.

Em cada experimento serão descritos a verificação lógica realizada e o seu resultado esperado. As execuções e as verificações foram repetidas onze vezes, e os tempos médios de execução e de verificação estão descritos nas tabelas 5.2 e 5.3 abaixo, junto aos desvios padrão, as variâncias e os resultados obtidos. Os tempos de execução de verificação que deram origem aos dados de cada experimento estão descritos no Apêndice C.

Cabe ressaltar que entre a Blockchain 1 e a Blockchain 4 varia apenas o número de usuários, e entre a Blockchain 2 e a Blockchain 3 varia apenas o número de transações. As duas primeiras blockchains foram geradas no modo automático e as duas últimas foram parametrizadas com base nas configurações das anteriores, de modo que pudessem ser comparados os resultados dos experimentos a partir destas duas variáveis, usuários e transações.

A Tabela 5.1 apresenta as configurações das blockchains utilizadas nos experimentos:

	Blockchain 1	**Blockchain 2**	**Blockchain 3**	**Blockchain 4**
Nº de nós	5	5	5	5
Nº de usuários	6	3	3	12
Nº de transações	17	70	35	17
Nº de blocos	6	3	3	6

Tabela 5.1: Configurações das blockchains

5.1 Experimento 1

O primeiro experimento consiste em verificar se em todos os estados da execução do modelo de Blockchain, um dado usuário possui saldo maior ou igual a zero. Ambos os resultados (TRUE e FALSE) são esperados e podem ser considerados coerentes. Isto se deve ao fato de que um usuário deve efetivamente possuir sempre saldo maior ou igual a zero, o que resultaria em uma resposta TRUE. Entretanto, o modelo formal está estruturado de forma que se em um dado estado uma operação resulta em saldo negativo para um usuário (necessariamente o remetente), então esta transação é considerada inválida e não será considerada na composição do saldo do usuário. Apesar disto, neste caso a resposta lógica do experimento seria FALSE.

A Tabela 5.2 a seguir apresenta os resultados do Experimento 1, cabendo ressaltar que nenhum dos tempos de execução e de verificação excedeu o limite de dois desvios padrão inferiores ou superiores à média, sendo que a maior parte das execuções e verificações manteve-se dentro do limite de um desvio padrão.

	Blockchain 1	**Blockchain 2**	**Blockchain 3**	**Blockchain 4**
Tempo médio de execução:	0,63 segundo	1,05 segundo	0,60 segundo	0,78 segundo
Desvio padrão:	0,029	0,037	0,028	0,025
Variância:	0,001	0,001	0,001	0,001
Satisfatibilidade:	SAT	SAT	SAT	SAT
Verificação CTL:	AG user1.balance ≥ 0			
Resultado obtido:	TRUE	TRUE	TRUE	TRUE
Tempo de execução da verificação:	0,01 segundos	0,01 segundos	0,01 segundo	588,67 segundos
Desvio padrão:	0,004	0,004	0	10,483
Variância:	0	0	0	109,902

Tabela 5.2: Resultados no Experimento 1

A partir da comparação dos resultados do Experimento 1 com a Blockchain 2 e com a Blockchain 3 pode-se observar que a influência do aumento do número de transações é baixa. Em ambas as verificações o modelo é satisfatível e o resultado lógico obtido foi TRUE, entretanto, verificou-se que no caso da Blockchain 3, cuja única diferença para a Blockchain 2 é possuir metade das transações, o tempo de execução do código reduziu cerca de 42%, mas o tempo de execução da verificação foi aproximadamente o mesmo.

Relativamente ao número de usuários, o Experimento 1 apontou grande influência no tempo de execução das verificações. Como se pode observar na Tabela 5.1 a Blockchain 1 e a Blockchain 4 diferem apenas pelo número de usuários, que na última é o dobro da primeira. Enquanto os tempos de execução do código variam cerca de 23%, o tempo de execução da verificação aumenta quase 60 mil vezes, saindo de um milésimo de segundo

para quase 10 minutos. Isto se dá em virtude de as verificações quase sempre recaírem sobre os usuários, direta ou indiretamente, o que torna a árvore de estados bastante extensa, demandando muito tempo para ser percorrida.

5.2 Experimento 2

Este experimento objetiva investigar se em algum estado uma dada transação que foi considerada inválida foi incluída em um bloco. O resultado esperado é FALSE, dado que uma transação só pode ser incluída em um bloco se for considerada válida.

A tabela 5.3 a seguir apresenta os resultados do Experimento 2. Assim como no Experimento 1, todos os tempos, tanto de execução como de de verificação, mantiveram-se dentro de dois desvios padrão acima ou abaixo da média.

	Blockchain 1	**Blockchain 2**	**Blockchain 3**	**Blockchain 4**
Tempo de execução do código:	0,67 segundo	0,68 segundo	0,48 segundo	0,78 segundo
Desvio padrão:	0,032	0,019	0,022	0,025
Variância:	0,001	0	0	0,001
Satisfatibilidade:	SAT	SAT	SAT	SAT
Verificação CTL:	EX (tx11.validTransaction = FALSE & tx11.blockId ! = −1)			
Resultado obtido:	FALSE	FALSE	FALSE	FALSE
Tempo de execução da verificação:	0,01 segundo	0,01 segundo	0,01 segundo	1183,44 segundos
Desvio padrão:	0,004	0,004	0	17,558
Variância:	0	0	0	308,287

Tabela 5.3: Resultados obtidos no Experimento 2

Assim como no Experimento 1, percebe-se que não há influência significativa do número de transações nos tempos de execução das verificações. A comparação entre os tempos da Blockchain 2 e da Blockchain 3 aponta tempos iguais.

O mesmo não se verifica em relação ao número de usuários, que, repetindo o comportamento observado no Experimento 1, influíram sobremaneira no tempo da verificação lógica. Comparando as execuções da Blockchain 1 e da Blockchain 4, a variação foi de mais de 118 mil vezes, saltando de um milésimo de segundo para quase 20 minutos, simplesmente dobrando-se o número de usuários.

5.3 Observações gerais

Os experimentos foram desenhados para executarem verificações sobre as principais propriedades de uma Blockchain, ao mesmo tempo em que demandassem verificações com

complexidades diferentes. Com estas duas verificações, foram considerados a composição de saldos de um usuário, a validação de transações e a geração de blocos. Todos os experimentos foram executados em um computador com processador Intel Core i7©de 7ª geração e 16GB de memória RAM. Outras diferentes configurações de Blockchains foram geradas com quantidades maiores de usuários e transações, mas a execução das verificações lógicas tornou-se inviável em virtude da falta de memória RAM suficiente, o que se explica pelos resultados obtidos nos dois experimentos com as quatro configurações de Blockchain utilizadas.

Capítulo 6

Conclusões e trabalhos futuros

O presente trabalho apresentou, até onde se pode verificar, a mais completa modelagem formal de Blockchain disponível, contendo suas principais entidades e propriedades; e a respectiva implementação em um *model checker*, o nuXmv, com base na Blockchain do Bitcoin.

A revisão da literatura indicou que, embora haja diversas outras modelagens formais e trabalhos acerca de aplicações utilizando Blockchain, em especial os *smart contracts*, nenhuma outra modelagem formal disponível nos trabalhos encontrados abordou como foco principal a Blockchain, de modo detalhado como neste trabalho.

A partir da modelagem formal, foi desenvolvido em linguagem Python um compilador de código SMV, capaz de gerar, de modo parametrizável ou aleatório, modelos formais de Blockchain de diferentes configurações, para serem utilizados em verificações diversas com o modelo criado no nuXmv.

Em seguida, como forma de aplicação da modelagem formal e do compilador de código, foram gerados quatro configurações de Blockchain, que foram utilizados em dois diferentes experimentos, que demonstraram a influência de certas características dos modelos, que influem na execução das verificações lógicas.

A comparação dos resultados dos experimentos evidencia que o principal fator que impacta no tempo de execução das verificações é o número de usuários, o que se justifica pela quantidade de verificações que são realizadas sobre esta entidade. A diferença no número de transações, por sua vez, não influi significativamente no tempo de execução.

A partir do modelo formal de Blockchain, do compilador e dos experimentos realizados, pode-se concluir que este trabalho atinge os seus objetivos, oferecendo o mais detalhado modelo formal de Blockchain em um *model checker* e estabelecendo importan-

tes bases sobre as quais um grande número de outros estudos, verificações e expansões podem ser realizados. Além disso, a alta legibilidade oferecida pela linguagem de entrada SMV, torna o refinamento, a adaptação, e as futuras expansões do modelo formal bastante mais simples e viáveis.

Vencido o desafio primordial de construir um modelo formal mais detalhado do que todas as opções existentes, e ainda construída a primeira versão de um compilador de código para alimentar este modelo formal, uma série de melhorias poderão ser realizadas em trabalhos futuros.

A adaptação mais aplicável, do ponto de vista do uso prático do modelo formal, é a que tornaria o modelo capaz de suportar *smart contracts*. O objetivo aqui não seria verificar os *smart contracts*, pois, como dito anteriormente, há uma série de abordagens para verificações diversas sobre estes, mas estudar e proceder verificações sobre a Blockchain que os suporta (i.e. como a execução de códigos de *smart contracts* interagindo através da Blockchain pode afetar seu desempenho e demais características). A partir do estágio em que se encontra a modelagem da Blockchain, o que difere uma rede em que trafeguem bitcoins para outra em que circulem *smart contracts* é majoritariamente o tipo de dado que está incluso em uma transação.

No que concerne às evoluções no detalhamento do modelo formal propriamente dito, há também diversas oportunidades, como a modelagem dos diferentes tipos de nós presentes na rede *peer-to-peer* em lugar de nós genéricos, a transmissão de transações, e não apenas blocos, entre nós e a modelagem mais aprofundada de transações descrevendo suas as entradas e saídas. Isso possibilitaria modelar também a cadeia de transações, em adição à de blocos.

Quanto ao compilador de código SMV, diversas melhorias poderão ser feitas, como o desenvolvimento de uma interface gráfica, o melhor detalhamento dos limites mínimos e máximos entre os quais as entidades do modelo devem ser geradas. Além disso, a adaptação do código para comportar todas as melhorias a serem construídas no modelo formal, que precisam ser refletidas no compilador.

Referências

[1] Abdellatif, T.; Brousmiche, K. Formal verification of smart contracts based on users and blockchain behaviors models. In *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)* (Feb 2018), pp. 1–5.

[2] Agda. What is agda?

[3] Amani, S.; Bortin, M.; Bégel, M.; Staples, M. Towards verifying ethereum smart contract bytecode in isabelle/hol. In *CPP 2018 - Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, Co-located with POPL 2018* (2018), vol. 2018-January, pp. 66–77. Cited By :11.

[4] Armin Biere, Alessandro Cimatti, E. M. C. O. S.; Zhu, Y. Bounded model checking. In *Advances in Computers* (2003), A. Press, Ed., vol. 58.

[5] Baier, C.; Katoen, J.-P. *Principles of Model Checking*. The MIT Press, 2008.

[6] Behrmann, G.; David, A.; Larsen, K. G.; Hakansson, J.; Petterson, P.; Yi, W.; Hendriks, M. Uppaal 4.0. In *Proceedings of the 3rd International Conference on the Quantitative Evaluation of Systems* (Washington, DC, USA, 2006), QEST '06, IEEE Computer Society, pp. 125–126.

[7] Blom, S.; Pol, J.; Weber, M. *Computer Aided Verification (CAV)*. Springer, 2010, ch. LTSmin: Distributed and Symbolic Reachability, pp. 354–359.

[8] Bozzano, M.; Cavada, R.; Cimatti, A.; Dorigatti, M.; Griggio, A.; Mariotti, A.; Micheli, A.; Mover, S.; Roveri, M.; Tonetta, S. nuxmv 1.1.1 user manual, 2016. Disponível em `https://es.fbk.eu/tools/nuxmv/downloads/nuxmv-user-manual.pdf`.

[9] Calzolai, F.; De Nicola, R.; Loreti, M.; Tiezzi, F. *TAPAs: A Tool for the Analysis of Process Algebras*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 54–70.

[10] Cavada, R.; Cimatti, A.; Dorigatti, M.; Griggio, A.; Mariotti, A.; Micheli, A.; Mover, S.; Roveri, M.; Tonetta, S. The nuxmv symbolic model checker. In *CAV* (2014), pp. 334 – 342.

[11] Cavada, R.; Cimatti, A.; Jochim, C. A.; Keighren, G.; Olivetti, E.; Pistore, M.; Roveri, M.; Tchaltsev, A. Nusmv 2.6 user manual. Acessado em 08/10/2018.

[12] Chaudhary, K.; Fehnker, A.; van de Pol, J.; Stoelinga, M. Modeling and verification of the bitcoin protocol. In *Proceedings Workshop on Models for Formal*

Analysis of Real Systems, MARS 2015, Suva, Fiji, November 23, 2015. (2015), R. J. van Glabbeek, J. F. Groote, and P. Höfner, Eds., vol. 196 of *EPTCS*, pp. 46–60.

[13] Chaum, D. Blind signatures for untraceable payments.

[14] Cimatti, A. Industrial applications of model checking. In *MOVEP 2000: Modeling and Verification of Parallel Processes* (2001), Summer School on Modeling and Verification of Parallel Processes, Springer, pp. 153–168.

[15] Clarke, E. M.; Emerson, E. A. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logics of Programs* (Berlin, Heidelberg, 1982), D. Kozen, Ed., Springer Berlin Heidelberg, pp. 52–71.

[16] Dorri, A.; Steger, M.; Kanhere, S. S.; Jurdak, R. Blockchain: A distributed solution to automotive security and privacy. *IEEE Communications Magazine 55*, 12 (Dec 2017), 119–125.

[17] Eén, N.; Sörensson, N. An extensible sat-solver. *SAT 2919 of LNCS* (2003), 502—-518.

[18] Fehnker, A.; Chaudhary, K. Twenty percent and a few days – optimising a bitcoin majority attack. In *NASA Formal Methods* (Cham, 2018), A. Dutle, C. Muñoz, and A. Narkawicz, Eds., Springer International Publishing, pp. 157–163.

[19] H. Massias, X. A.; Quisquater, J.-J. Design of a secure timestamping service with minimal trust requirement. In *20th Symposium on Information Theory in the Benelux* (Maio 1999).

[20] Karame, G. On the security and scalability of bitcoin's blockchain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 1861–1862.

[21] Kiayias, A.; Russell, A.; David, B.; Oliynykov, R. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *Advances in Cryptology – CRYPTO 2017* (Cham, 2017), J. Katz and H. Shacham, Eds., Springer International Publishing, pp. 357–388.

[22] Kwiatkowska, M.; Norman, G.; Parker, D. Probabilistic symbolic model checking with PRISM: A hybrid approach. In *Proc. 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)* (2002), J.-P. Katoen and P. Stevens, Eds., vol. 2280 of *LNCS*, Springer, pp. 52–66.

[23] Lahiri, S. K.; Nieuwenhuis, R.; Oliveras, A. Smt techniques for fast predicate abstraction. In *Computer Aided Verification* (Berlin, Heidelberg, 2006), T. Ball and R. B. Jones, Eds., Springer Berlin Heidelberg, pp. 424–437.

[24] "Lime, D.; Roux, O. H.; Seidner, C.; Traonouez, L.-M. Romeo: A parametric model-checker for petri nets with stopwatches. In *Tools and Algorithms for the Construction and Analysis of Systems* (Berlin, Heidelberg, 2009), S. Kowalewski and A. Philippou, Eds., Springer Berlin Heidelberg, pp. 54–57.

[25] Meinel, C.; Theobald, T. *Algorithms and data structures in VLSI design: OBDD-foundations and applications.* Springer, 1998.

[26] Molina-Jimenez, C.; Sfyrakis, I.; Solaiman, E.; Ng, I.; Weng Wong, M.; Chun, A.; Crowcroft, J. Implementation of smart contracts using hybrid architectures with on and off–blockchain components. In *2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)* (Nov 2018), pp. 83–90.

[27] Moskewicz, M. W.; Madigan, C. F.; Zhao, Y.; Zhang, L.; Malik, S. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th Annual Design Automation Conference* (New York, NY, USA, 2001), DAC '01, ACM, pp. 530–535.

[28] Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system. 2009.

[29] Nash, D. M. . P. M. . W. Validation and verification of smart contracts: A research agenda. *Computer 50*, 9 (Sept. 2017), 50 – 57.

[30] Nipkow, T.; Paulson, L.; Wenzel, M.; Klein, G.; Haftmann, F.; Weber, T.; Hölzl, J. Isabelle.

[31] Orcutt, M. Once hailed as unhackable, blockchains are now getting hacked, Feb 2019.

[32] Piriou, P.; Dumas, J. Simulation of stochastic blockchain models. In *2018 14th European Dependable Computing Conference (EDCC)* (Sep. 2018), pp. 150–157.

[33] Qu, M.; Huang, X.; Chen, X.; Wang, Y.; Ma, X.; Liu, D. Formal verification of smart contracts from the perspective of concurrency. In *SmartBlock* (2018), M. Qiu, Ed., vol. 11373 of *Lecture Notes in Computer Science*, Springer, pp. 32–43.

[34] SAT Research Group, P. U. Zchaff. Disponível em 12/02/2019.

[35] Setzer, A. Modelling bitcoin in agda. *CoRR abs/1804.06398* (2018).

[36] Szwarcfiter, J. L.; Markenzon, L. *Estruturas de dados e seus algoritmos.* LTC, 2010.

[37] Valentin Goranko, A. G. Temporal logic, 2015. Disponível em `https://plato.stanford.edu/entries/logic-temporal/`.

[38] van der Meyden, R. On the specification and verification of atomic swap smart contracts. *CoRR abs/1811.06099* (2018).

[39] Zeinab Nehai, Pierre-Yves Piriou, D. F. Model-checking of smart contracts. In *The 2018 IEEE International Conference on Blockchain* (2018).

APÊNDICE A – Modelagem dos elementos da Blockchain no nuXmv

```
MODULE createNode(nodeId)

  VAR
    IPSent: boolean;
    IPReceived: boolean;
    connectSeed: boolean;
    versionSent: boolean;
    verack: boolean;
    networkOK: boolean;
    lastBlockId: 0..10;

  ASSIGN
    init(IPSent) := TRUE;
    init(IPReceived) := FALSE;
    init(connectSeed) := FALSE;
    init(versionSent) := FALSE;
    init(verack) := FALSE;
    init(networkOK) := FALSE;
    init(lastBlockId) := 0;


    next(IPSent) := TRUE;
    next(IPReceived) := IPSent = TRUE ? TRUE : FALSE;
    next(connectSeed) := (IPSent = TRUE & IPReceived = TRUE) ? TRUE : FALSE;
    next(versionSent) := connectSeed = TRUE ? TRUE : FALSE;
    next(verack) := versionSent = TRUE ? TRUE : FALSE;
    next(networkOK) := verack = TRUE ? TRUE : FALSE;

```

```
29 MODULE createUser(idUser, initBalance)
30
31   VAR
32     balance : 0..400;
33     busy : boolean;
34
35   ASSIGN
36     init(balance) := initBalance;
37
38   TRANS !update -> conserve ;
39
40   DEFINE
41     conserve := (next(balance) = balance);
42
43 MODULE createTransaction(idTrans, from, to, amount)
44
45   FROZENVAR
46     blockId : 0..10;
47
48
49   DEFINE
50     validTransaction := !from.busy & !to.busy & from.balance >= amount; --- Does the
         sender user have enough balance?
51     newFromBalance := from.balance - amount;
52     newToBalance := to.balance + amount;
53
54   TRANS
55     validTransaction -> (next(from.balance)= newFromBalance) &
56                 (next(to.balance)=newToBalance) &
57                 (next(from.busy)=!from.busy) &
58                 (next(to.busy)=!to.busy);
59
60
61 MODULE createBlock(idBloco, txa, txb)
62   FROZENVAR
63     id : 0..10;
64
65   ASSIGN
```

```
66      init(id) := idBloco;
67      init(txa.blockId) := validBlock ? id : 0;
68      init(txb.blockId) := validBlock ? id : 0;
69
70
71    DEFINE
72      validBlock := txa.validTransaction & txb.validTransaction;
73      nonce := signed word[32](190);
74      --hash := validBlock ? (nonce :: signed word [32](txa.to) :: signed word [32](txb.to) ::
            signed word[32](previousBlock))
75      --  : unsigned word[128](0);
76
77
78
79  MODULE updateNode(node, block)
80
81    DEFINE
82      lastNodeBlock := node.lastBlockId;
83      blockToUpdate := block.validBlock ? block.id : -1;
84      shouldUpdate := lastNodeBlock <= blockToUpdate;
85    ASSIGN
86      next(node.lastBlockId) := shouldUpdate ? lastNodeBlock : block.id;
```

Listing 8: Modelagem dos elementos da Blockchain

APÊNDICE B – Código fonte do gerador de instâncias Blockchain

```
#!/usr/bin/env python
# coding: utf-8

import random
from random import seed
from random import choice
import datetime

print("--- Automatic blockchain SMV code generator. ---")

autoGen = input("Do you want to automatically generate a blockchain based on randomic
    parameters? (yes/no)")



if autoGen == "no":
    numNodes = int(input("How many nodes do you want in your P2P network?"))
else:
    seed(input("Type an integer number to be used as the seed of ramdomness: "))
    numNodes = 10 #random.randint(2,20)


nodes = []
for i in range(1,numNodes+1):
        nodes.append("node{} : createNode({});".format(i, i))


#for i in range(0,len(nodes)):
```

```
    # print(nodes[i])


print("The {} requested nodes was sucessfully created.".format(len(nodes)))
print("You must create at least 2 users. The users are pairs like (id, balance).")

users = []
if autoGen == "no":
    numUsers = int(input("How many users do you want to create?"))
    if numUsers >= 2:
        for i in range(1,numUsers+1):
            balance = int(input("Please type the initial balance of the user{}:".format(i)))
            users.append("user{} : createUser(self,{});".format(i, balance))
    else:
        print("You must create at least 2 users")
else:
    numUsers = 30 #random.randint(2,25)
    for i in range(1,numUsers+1):
        balance = random.randint(0,100)
        users.append("user{} : createUser(self,{});".format(i, balance))

userList = []
for i in range(1, numUsers+1):
    userList.append("user{}".format(i))
#print(userList)
#for i in range(0,len(users)):
# print(users[i])


print("Now you can create transactions. A transaction is a tuple like (from, to, balance)")


transactions = []
userTxns = dict()

for i in range(1,len(users)+1):
    userTxns["user{}".format(i)] = []

```

```
if autoGen == "no":
    anotherTxn = "yes"
    countTxn = 1
    while anotherTxn == "yes":
        ufrom = input("Please type the sender user:")
        uto = input("Please type the receiver user:")
        amount = int(input("Please type the amount transfered:"))
        transactions.append("tx{} : createTransaction(self,{},{},{});".format(countTxn,
            ufrom, uto, amount))
        anotherTxn = input("Do you want to create a new transaction? (yes/no)")
        if anotherTxn == "yes":
            countTxn += 1
else:
    countTxn = 17 #random.randint(2,60)
    for i in range(1,countTxn+1):
        userTxn = []
        ufrom = random.choice(userList)
        uto = random.choice(userList)
        amount = random.randint(1,20)
        transactions.append("tx{} : createTransaction(self,{},{},{});".format(i,ufrom, uto,
            amount))
        if ufrom == uto :
            userTxns[ufrom].append("tx{}".format(i))
        else:
            userTxns[ufrom].append("tx{}".format(i))
            userTxns[uto].append("tx{}".format(i))

print("{} transactions created.".format(countTxn))
#print(userTxns)


txnList = []
for i in range(1,countTxn+1):
    txnList.append("tx{}".format(i))
#for i in range(0, len(transactions)):
# print(transactions[i])


```

```
print("Now it's time to create blocks")


blocks = []
if autoGen == "no":
    anotherBlock = "yes"
    countBlock = 1
    while anotherBlock == "yes":
        txa = input("Which transaction do you include in this block?")
        txb = input("Which other transaction do you include in this block?")
        blocks.append("block{} : createBlock({},{},{});".format(countBlock, countBlock,txa,
            txb))
        anotherBlock = input("Do you want to create a block? (yes/no)")
        if anotherBlock == "yes":
            countBlock += 1
else:
    countBlock = 6 #random.randint(1,int(countTxn/2))
    for i in range(1,countBlock+1):
        txa = random.choice(txnList)
        txnList.remove(txa)
        txb = random.choice(txnList)
        blocks.append("block{} : createBlock({},{},{});".format(i, i,txa,txb))

print("{} blocks created.".format(countBlock))

#for s in range(0,len(blocks)):
# print(blocks[s])

print("")
print("The blockchain generated has the below parameters:")
print("--> {} nodes;".format(len(nodes)))
print("--> {} users;".format(len(users)))
print("--> {} transactions;".format(len(transactions)))
print("--> {} blocks;".format(len(blocks)))
print("")
print("--- Copy all the below code and paste it at the end of the file blockchain.smv ---"
    )
print("")
```

```
138 print("")
139 base = open("blockchain−model.smv", "r")
140 model = base.readlines()
141 f = open("blockchain.smv", "w+")
142 f.writelines(model)
143 f.write("MODULE main" + "\nVAR" + "\n")
144 ("\n")
145 for i in range(0,len(nodes)):
146     f.write("\n " + nodes[i])
147 f.write("\n")
148 for j in range(0,len(users)):
149     f.write("\n " + users[j])
150 f.write("\n")
151 for k in range(0,len(transactions)):
152     f.write("\n " + transactions[k])
153 f.write("\n")
154 for s in range(0,len(blocks)):
155     f.write("\n " + blocks[s])
156 f.write("\n")
157 f.write("\nASSIGN")
158 for i in range(1,int((len(users)))):
159     f.write("\n init(user{}.busy) := TRUE;".format(i))
160 f.write("\n")
161 f.write("\nDEFINE")
162
163 for i in range(1,len(users)+1):
164     userTxn = userTxns["user{}".format(i)]
165     string_update = "\n user{}.update := ".format(i)
166     if len(userTxn) != 0:
167         for i in range(0, len(userTxn)):
168             if i != len(userTxn) − 1:
169                 string_update = string_update + (userTxn[i]+".validTransaction | ")
170             else:
171                 string_update = string_update + (userTxn[i]+".validTransaction;")
172     else:
173         string_update = string_update + "FALSE;"
174     f.write(string_update)
175
```

```
f.write("\n")
f.write("\nblock1.previousBlock := block1.validBlock ? 0 : −1;")
for i in range(2,len(blocks)+1):
    f.write("\nblock{}.previousBlock := case".format(i))
    f.write("\n block{}.validBlock : case".format(i))
    j = i − 1
    while(j > 0):
        f.write("\n block{}.validBlock : block{}.id;".format(j,j))
        j = j − 1
    f.write("\n esac;")
    f.write("\n TRUE : −1;")
    f.write("\n esac;")

f.close()
```

Listing 9: Código fonte do gerador de instâncias Blockchain

APÊNDICE C – Tempos de execução e de verificação dos experimentos

	Blockchain 1	Blockchain 2	Blockchain 3	Blockchain 4
Tempos de execução	0,63	1,05	0,61	0,78
	0,69	1,03	0,62	0,77
	0,59	1,06	0,61	0,82
	0,61	1,08	0,63	0,79
	0,63	1,12	0,58	0,78
	0,65	1,01	0,65	0,83
	0,64	1,08	0,60	0,82
	0,65	1,05	0,59	0,81
	0,63	0,98	0,63	0,77
	0,62	1,05	0,61	0,75
	0,59	1,04	0,68	0,78
Média	0,63	1,05	0,62	0,79
Desvio Padrão	0,029	0,037	0,028	0,025
Variância	0,001	0,001	0,001	0,001

Tabela C.1: Tempos de execução do Experimento 1

	Blockchain 1	Blockchain 2	Blockchain 3	Blockchain 4
Tempos de verificação	0,01	0,01	0,01	588,67
	0,01	0,01	0,01	567,56
	0,01	0,01	0,01	592,13
	0,02	0,01	0,01	569,76
	0,01	0,01	0,01	601,48
	0,01	0,02	0,01	598,64
	0,02	0,01	0,01	586,97
	0,01	0,01	0,01	588,79
	0,01	0,02	0,01	587,41
	0,01	0,01	0,01	588,82
	0,01	0,01	0,01	580,15
Média	0,01	0,01	0,01	586,40
Desvio Padrão	0,004	0,004	0	10,483
Variância	0	0	0	109,902

Tabela C.2: Tempos de verificação do Experimento 1

	Blockchain 1	Blockchain 2	Blockchain 3	Blockchain 4
Tempos de execução	0,67	0,68	0,48	0,78
	0,65	0,68	0,52	0,77
	0,66	0,68	0,46	0,75
	0,63	0,65	0,52	0,79
	0,65	0,67	0,48	0,78
	0,63	0,71	0,47	0,85
	0,71	0,70	0,47	0,79
	0,68	0,66	0,48	0,80
	0,65	0,67	0,46	0,78
	0,72	0,68	0,49	0,77
	0,71	0,65	0,51	0,79
Média	0,67	0,68	0,49	0,79
Desvio Padrão	0,032	0,019	0,022	0,025
Variância	0,001	0,000	0,000	0,001

Tabela C.3: Tempos de execução do Experimento 2

	Blockchain 1	Blockchain 2	Blockchain 3	Blockchain 4
Tempos de verificação	0,01	0,01	0,01	1183,44
	0,01	0,01	0,01	1162,37
	0,01	0,01	0,01	1182,82
	0,02	0,01	0,01	1175,64
	0,01	0,01	0,01	1201,39
	0,01	0,02	0,01	1198,43
	0,02	0,01	0,01	1201,49
	0,01	0,01	0,01	1165,38
	0,01	0,02	0,01	1188,62
	0,01	0,01	0,01	1204,98
	0,01	0,01	0,01	1153,64
Média	0,012	0,012	0,010	1183,473
Desvio Padrão	0,004	0,004	0	17,558
Variância	0	0	0	308,287

Tabela C.4: Tempos de verificação do Experimento 2