Automatic program verification in Dynamic Logic with applications to smart contracts

Allan Patrick[1] **, Igor Machado Coelho**[1] **, Bruno Lopes**[1]

[1]Instituto de Computação
Universidade Federal Fluminense
Niterói-RJ – Brazil

allanpatrick@id.uff.br, imcoelho@ic.uff.br, bruno@ic.uff.br

Abstract. *In critical systems, failures or errors can cause catastrophes, such as deaths or considerably losses of money. Model checking provides an automated way to prove the correctness of programs' requirements. It is a convenient technique to use in systems that need reliability. Propositional Dynamic Logic (PDL) is a formal system designed to reason about programs. This work presents a compiler implementation from a subset of the C language and also for the Smacco model, both to the PDL language, and after that to the language of the nuXmv model checker. This implementation is linked with a Blockchain model generation system to model and reason about smart contracts.*

1. Introduction

Critical systems do not accept failures or errors, as they can cause major losses, such as deaths or major financial losses [11]. These systems require a high degree of confidence and are present in many areas such as medicine and aviation.

Blockchain [14] is a disruptive technology that may be used to store data and actions (denoted by programs), as an immutable global ledger. But incidents have happened in the past[1], indicating that critical failures can be exploited, thus affecting its integrity.

Formal systems are mathematical models for reasoning about programs, leading to a mathematical proof that these programs are correct. Line 1 of the Paris metro, for example, used formal methods to certify its automated system without human interaction [8]. In the context of the work presented here, the mathematical model used in the formal system of this project was Propositional Dynamic Logic with the nuXmv [4] checker.

In this paper, a formalization of programs was implemented in nuXmv, from the Mini-C [12] and Smacco (JSON) [6, 13] languages. It is compiled to the PDL language and then used to create nuXmv models. Finally, these models can be joined as smart contracts in Blockchain models. This paper is organized as follows. Section 2 describes the notion of the Propositional dynamic logic, Section 3 presents the nuXmv, Section 4 shows how a blockchain works, Section 5 presents the concept of Smart contracts and how it is used in this work, Section 6 show how this work generate the models and Section 7 presents the conclusions of this work.

[1]https://www.gemini.com/cryptopedia/the-dao-hack-makerdao

2. Propositional Dynamic Logic

Propositional Dynamic Logic [10] (PDL) is a multi-modal logic tailored to reason about regular programs. A formula has the form $\langle\pi\rangle\varphi$ that can be read as "after some execution of program π, φ holds, supposing that π halts."

PDL provides a natural abstraction for programs, where many fundamental relationships between programs and propositions can be studied. PDL has been used for the verification of software requirements, including in companies as IBM [15].

Definition 1 (Propositional Dynamic Logic) *The propositional dynamic logic [9] language is described below*

- *Atomic propositions denoted by $p, q, r, \ldots$, where Φ is an enumerable set of all atomic propositions.*
- *Atomic programs denoted by $\alpha, \beta, \ldots$, where Π is an enumerable set of all atomic programs.*
- *A PDL formula can be written according to the following BNF.*

$$\varphi ::= p \mid \top \mid \neg\varphi \mid \varphi \wedge \varphi \mid \langle\pi\rangle\varphi,$$

- *PDL programs can be described according to the following BNF.*

$$\pi ::= \alpha \mid \pi;\pi \mid \pi \cup \pi \mid \pi^\star \mid \varphi?$$

We use the standard abbreviations $\varphi \vee \psi = \neg(\neg\varphi \wedge \neg\psi)$, $\varphi \rightarrow \psi = \neg(\varphi \wedge \neg\psi)$ and $[\pi]\varphi = \neg\langle\pi\rangle\neg\varphi$.

Definition 2 (Frame) *A PDL frame is a tuple $\mathcal{F} = (\mathcal{W}, R_\pi)$ such that*

W *is a non-empty set of states*
R_π *is a binary relation over W inductively defined as below*

$$\begin{aligned} R_{\pi_1;\pi_2} &= R_{\pi_1} \cdot R_{\pi_2} \\ R_{\pi_1 \cup \pi_2} &= R_{\pi_1} \cup R_{\pi_2} \\ R_{\pi^\star} &= R_\pi^\star \\ R_{\varphi?} &= \{(w,w) \mid w \in W \text{ and } \mathcal{M}, w \models \varphi\} \end{aligned}$$

Definition 3 (Model) *A PDL model is a tuple $\mathcal{M} = (\mathcal{F}, \mathbf{V})$, where $\mathcal{F}$ is a PDL frame and $\mathbf{V} \colon W \times \Phi \rightarrow \{True, False\}$ is a valuation function.*

Definition 4 (Semantic interpretation) *The notion of satisfaction of a φ formula in a $\mathcal{M} = (\mathcal{F}, \mathbf{V})$ model in the w state is given by*

$\mathcal{M}, w \models p$ *iff* $\mathbf{V}(w, p) = True$
$\mathcal{M}, w \models \top$ *always*
$\mathcal{M}, w \models \neg\varphi$ *iff* $\mathcal{M}, w \not\models \varphi$
$\mathcal{M}, w \models \varphi_1 \wedge \varphi_2$ *iff* $\mathcal{M}, w \models \varphi_1$ *and* $\mathcal{M}, w \models \varphi_2$
$\mathcal{M}, w \models \langle\pi\rangle\varphi$ *iff exists* $w' \in W$ *such that* $wR_\pi w'$ *and* $\mathcal{M}, w' \models \varphi$

3. The nuXmv checker

nuXmv [4] is a symbolic model checker that extends the functionality of NuSMV, a symbolic model checker originated from SMV [5]. NuXmv uses techniques based on BDD [2] (Binary Decision Diagrams) and SAT-solvers.

A nuXmv program is composed by *MODULES*, the main one being module *main*, where the execution starts. Within a *MODULE* we can have some structures like *VAR* which is a list of variables from the current module and *ASSIGN* which is a list of assignments. These structures can be seen below:

The variables in nuXmv are defined in *VAR*, each variable within this list can be described as an identifier: type, where the identifier is the identification of the variable in the list and type, is the type declaration of the variable, also we can declare and instantiate a *MODULE* as a variable.

The assignments in nuXmv are defined in *ASSIGN*, each assignment within the list can be described as an identifier := expression, init (identifier) := expression or next (identifier) := next_expression, where the identifier is the identification of the variable that will be assigned, the expression is the value that will be passed to the variable and next_expression is the value that the variable will have in the next state. Expressions inside curly brackets denote non-deterministic assignments.

4. Blockchain

Blockchain is a database of transactions, stored in chained blocks and distributed across all nodes in a peer-to-peer network. This protocol was created by Satoshi Nakamoto in 2008 in the article *Bitcoin: A peer-to-peer electronic cash* [14]. Blockchains are commonly used by cryptocurrencies as their technological base; and some of these cryptocurrencies are Ripple[2], Neo[3], Bitcoin Cash[4], Litecoin[5], and Binance Coin[6].

This work incorporates a formal verification of Blockchains model, using a formalization of Blockchains with the use of nuxmv to carry out the formal verification [7]. The Blockchain model can be seen in Figure 1.The formalization of Blockchains has been modified to accept Blockchains with financial transactions and *smart contracts*.

5. Smart contracts

Smart contracts are computer codes that execute after some condition. These conditions can be a program that executes a transaction after a certain number of likes on a social network or after a transaction is carried out. This type of program guarantees that the purpose will be fulfilled only when the requirements are met [1].

This work implements smart contracts modeled from computer programs in the Mini-C language and for the smart contract model Smacco [13]. These models can be defined respectively as a subset of the C without structures such as pointers and composite data types and Smacco as a JSON model.

[2] https://ripple.com/
[3] https://neo.org/
[4] https://bitcoincash.org/
[5] https://litecoin.org/pt/
[6] https://www.binance.com/en

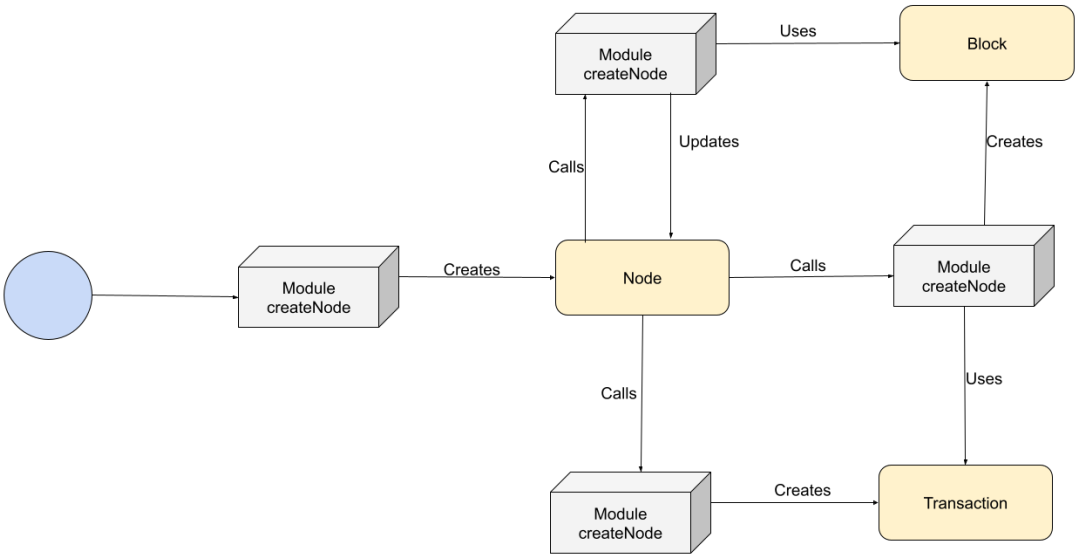

Figure 1. Blockchain Model according to [7]

An Smacco model is defined by a list of parameters that define the properties of the contract such as the version of Smacco, the version of the contract, the rules of the contract that will define whether the contract will fail or be accepted. A simple example of a contract modeled on Smacco can be found at Listing 1.

```
1     "standard": "smacco-1.0",
2     "input_type" : "single",
3     "pubkey_list" : ["036245f426b4522e8a2901be6ccc1f71e37dc376726cc65d"],
4     "rule" : {
5         "rule_type": "ALLOW_IF",
6         "condition" : { "condition_type" : "CHECKSIG" } }
```

Listing 1. Smacco example

6. Automatic model generation

The implementation of this project was done in the Python language and can be found on the following Github:
`https://github.com/frame-lab/Verificador-de-Smart-Contracts`.
The project is structured in 4 stages: Compilation, transformation to PDL, transformation to nuXmv, and linking to a smart contract.

6.1. Build

The user can choose to generate a Blockchain with smart contracts in Smacco or Mini-C, after the decision the program generates the list of tokens and the syntactic tree.

Tokens are generated looking at the list of tokens and reserved words of the language that appear in the program from the execution of the LEX of the PLY library [3].

The generation of the syntactic bottom-up tree is made from the execution of the YACC from the Ply library on the user input based on the defined language rules list.

6.2. Imperative programs to PDL

The translation of imperative programs to PDL is described below:

- if φ then α ::= $(\varphi?;\alpha) \cup \neg\varphi?$
- if φ then α else β ::= $(\varphi?;\alpha) \cup (\neg\varphi?;\beta)$
- while φ do α ::= $(\varphi?;\alpha)^\star;\neg\varphi?$
- do α while φ ::= $\alpha;(\varphi?;\alpha)^\star;\neg\varphi?$
- for (π;φ;ω) do α ::= $\pi;(\varphi?;\alpha;\omega)^\star;\neg\varphi?$

The translation of the tree structure received is divided into 4 parts: scope, expression, conditional, and repetition. Scope translations are denoted by the entire scope content being present within parentheses. Expression translations are performed by treating each expression as an atomic program.

Conditional translations are performed by creating a verification from the condition and in the case it is true the execution of its scope occurs, finally, the verification of the condition contradiction is added non-deterministically so that the program does not end with an error. If the operation is an if-else we have after the contradiction an execution of the else scope and if the operation has an if-else-if structure we add the else-if scope after the contradiction.

The translations of repetitions are carried out by creating a PDL verification for the condition and if it is true the execution of its scope occurs, finally we create an iteration and a verification of the condition contradiction is added so that the program does not abort with an error. If the operation is a do-while, before the condition we have an execution of the scope.

6.3. PDL to nuXmv

The translation from PDL to nuXmv occurs in the following steps. For each function of the PDL program, a module of the same name is created.

VAR is added to MODULE and the program variable is created. The program variable will be filled with all the atomic programs presents in the current PDL program and for each atomic program assigned in this way we will concatenate the atomic program with their position in the PDL program.

For each unique atomic program, a new variable in VAR is created, and it is defined as a set of all of the values that the variable can have during the execution of the program.

ASSIGN is added to MODULE and for each variable in VAR its init is added to ASSIGN. The value of init will be the first value assigned to the variable in the order of execution of the algorithm if the value to be added is a verification we will have non-determinism between verification and its contradiction.

For each variable declared in VAR its next is added to ASSIGN with all possible states of the variable during the execution of the program, if the state leads to a verification, we will have the non-determinism between the verification and its contradiction.

6.4. Smart contracts in nuXmv

The transformation of a model in nuXmv to a smart contract occurs, changing all modules identifiers concatenating the file name with the identifier. This change was necessary to

ensure that all modules were unique, in addition to this change it was necessary to change the Blockchain model in the transaction module so that it could accept both monetary transactions and smart contracts.

6.5. Execution of the algorithm

Now we will approach the step-by-step execution of the algorithm, given the initial entry, contained in the Listing 2.

```
1  int main () {
2      int i; int j; int z;
3      i = 1; j = 2; z = 3;
4      if(i < 5) {
5          z = 4;
6      } else {
7          z = 5;
8      } }
```

Listing 2. Input Mini-C

From the input, respectively we got that the file is read and the list of program tokens is generated, the syntactic tree will be generated and then the PDL program are created, which can be seen below in our example:

$$(int\ i; int\ j; int\ z; i = 1; j = 2; z = 3; (i < 5?; z = 4) \cup\ (\neg i < 5?; z = 5))$$

Then the PDL program is read and the nuXmv program is generated, which can be seen in the Listing 3.

```
1  MODULE main
2      VAR
3          program: {int_i_0, int_j_1, int_z_2, i_=_1_3, j_=_2_4, z_=_3_5, i_<_5_?_6,
               z_=_4_7, ~_i_<_5_?_8, z_=_5_9};
4          i: 1...1;
5          j: 2...2;
6          z: 3...5;
7      ASSIGN
8          init(program) := int_i_0;
9          init(i) := 1;
10         init(j) := 2;
11         init(z) := 3;
12         next(program) := case
13             program = int_i_0: int_j_1;
14             program = int_j_1: int_z_2;
15             program = int_z_2: i_=_1_3;
16             program = i_=_1_3: j_=_2_4;
17             program = j_=_2_4: z_=_3_5;
18             program = z_=_3_5: {i_<_5_?_6, ~_i_<_5_?_8};
19             program = i_<_5_?_6: z_=_4_7;
20             program = ~_i_<_5_?_8: z_=_5_9; esac;
21         next(i) := case
22             program = i_=_1_3: 1;
23             TRUE = i; esac;
24         next(j) := case
25             program = j_=_2_4: 2;
26             TRUE = j; esac;
27         next(z) := case
28             program = z_=_3_5: 3;
29             program = z_=_4_7: 4;
30             program = z_=_5_9: 5;
31             TRUE = z; esac;
```

Listing 3. Example nuXmv Mini-C

Finally, the program created is linked as a smart contrat with the generated Blockchain model.

Now we present a step-by-step execution of the algorithm, given another initial input in the Smacco model, contained in the Listing 4.

```
1    "standard": "smacco",
2    "input_type" : "single",
```

Listing 4. Input in Smacco

From the input, respectively we got that the file is read and the list of program tokens is generated, the syntactic tree will be generated and then the PDL program are created, which can be seen below in our example:

$$(standard: \ smacco; input\_type: \ single; Accept: ALLOW)$$

Then the PDL program is read and the nuXmv program is generated, which can be seen in the Listing 5.

```
1    Module main
2        VAR
3            program: {standard:_smacco_0, input_type:_single_1, Accept:_ALLOW_2};
4
5        ASSIGN
6            init(program) := standard:_smacco_0;
7            next(program) := case
8                program = standard:_smacco_0: input_type:_single_1;
9                program = input_type:_single_1: Accept:_ALLOW_2; esac;
```

Listing 5. Example Smacco nuXmv

Finally, the program created is linked as a smart contrat with the generated Blockchain model.

After the generated programs are ready we can use the nuXmv to make formal verification of the programs properties like test if there is a case that the program will fail, test what is the value of a variable in some context, text if the blockchain is valid and more.

7. Conclusions and future work

With the growing need for increasingly secure systems, the use of a model checker is becoming more and more in demand. PDL unified with nuXmv demonstrated to be an intuitive choice for simple programs.

This work presented an automated process with a compiler to generate models for formal verification of programs. The inputs can be in the Mini-C or Smacco (JSON) languages and are converted to a nuXmv model checker, which can be linked with a Blockchain verificaction model to make formal verifications about smart contracts relying on a compiler provided by the work [7]. The tool can be used to easily provide formal models for programs, from the command line or generate Blockchain verification models with smart contracts.

The focus of future works is the expansion of the Mini-C language to get closer and closer to the standard C language, support for a subset of the Solidity language, and the development of an online platform to run the proposed system.

References

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

[2] H. R. Andersen. An introduction to binary decision diagrams. *Lecture notes, available online, IT University of Copenhagen, p. 5*, 1997.

[3] David M. Beazley. PLY (Python Lex-Yacc). https://www.dabeaz.com/ply/ply.html.

[4] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. The nuxmv symbolic model checker. *CAV, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds., vol. 8559*, Springer, 2014, pp. 334–342.

[5] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. *Computer Aided Verification, E. Brinksma and K. G. Larsen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg*, 2002, pp. 359–364.

[6] Vitor N. Coelho, Thays A. Oliveira, Wellington Tavares, and Igor M. Coelho. Smart accounts for decentralized governance on smart cities (to appear). *Smart Cities*, 2021.

[7] Bruno Olímpio Costa. Verificação formal de modelos de blockchain. Master's thesis, Universidade Federal Fluminense, 2019.

[8] S. Gerhart, D. Craigen, and T. Ralston. Case study: Paris metro signaling system. *IEEE Software, vol. 11, no. 1, pp. 32–28*, 1994.

[9] David Harel, Dexter Kozen, and Jerzy Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[10] Michael J.Fischer and Richard E.Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 1979.

[11] J. C. Knight. Safety critical systems: challenges and directions. *Proceedings of the 24th International Conference on Software Engineering*, ACM, 2002, pp. 547–550.

[12] Yubi Lee. flex-bison. https://github.com/eubnara/flex-bison.

[13] Igor Machado. Smacco. https://neo-smacco.readthedocs.io/en/latest/intro.html.

[14] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `https://bitcoin.org/bitcoin.pdf`, 2009.

[15] C. Sinz, W. Kuchlin, and T. Lumpp. Towards a verification of the rule-based expert system of the ibm sa for os/390 automation manager. In *Proceedings Second Asia-Pacific Conference on Quality Software*, pages 367–374, 2001.