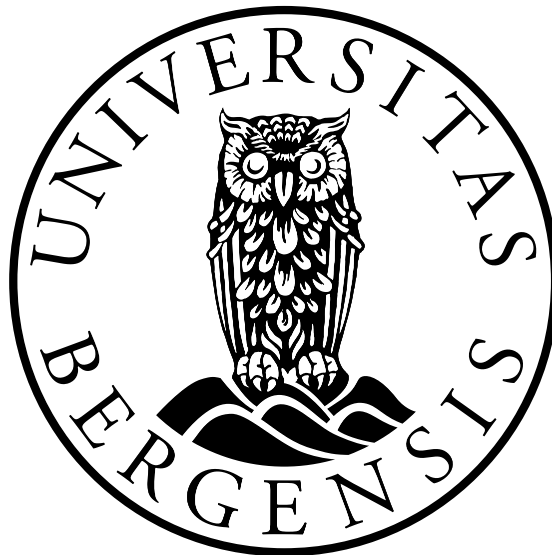
Model Checking Upgrades in Ethereum Smart Contracts

Eirin Maria Larsson Aga

Supervisor: Dr Rustam Galimullin



Master's Thesis
Department of Information Science and Media Studies
University of Bergen

June 2, 2023

Acknowledgements

First and foremost, I would like to thank my supervisor Dr Rustam Galimullin. Thank you for all your guidance and for always being positive. Thank you for all knowledge you shared and for your support throughout the thesis.

Thank you, Damian Kurpiewski, the lead developer of StraTegic Verifier. Thank you for answering questions.

A thousand thanks to Olav, my MVB (Most Valuable Brother), for helping out where my coding skills came short, thank you for teaching me about parsers and thank you for your optimism and kind words throughout the whole process.

Thank you, Solveig, my Golden Retriever, for always greeting me at the door no matter what time or mood. Thank you for cheering me on.

Thank you, Jørgen, for all dinners made, for all walks with Solveig, no matter if it was your turn or not. Thank you for keeping up the spirit and for keeping the plants alive.

Thank you to my study mates André, Børge, Frederik and Sandra. It would not be the same without you. Thank you for your competitive instincts and all competitions for the highest attendance at the study hall; it gave me a much-appreciated incentive to get to uni early every morning. Thank you for all the fun we have had along the way.

Finally, I would like to thank my family and friends for all support given.

Abstract

In this master's thesis, the model-checking tool StraTegic Verifier has been extended to verify upgrades in Smart Contracts. It has been shown how to model Smart Contracts as Concurrent Game Models, enabling them to be verified by the model checker. By implementing Dictatorial Dynamic Coalition Logic, upgrades of Smart Contracts can be expressed, and the models can be changed before verification. The implementation of DDCL is a step in the direction of being able to use formal verification when there are upgrades in Smart Contracts.

Contents

1 Introduction **1**

2 Background **3**

2.1 Blockchain and Cryptocurrency . . . 3
2.2 The Ethereum Network and Smart Contracts . . . 5
2.3 An Example of a Smart Contract . . . 6
2.4 Testing and Verification of Smart Contracts . . . 7
2.4.1 Formal Verification of Smart Contracts . . . 8
2.4.2 Model Checking in General . . . 10
2.4.3 Model Checking Smart Contracts . . . 10

3 Models, Logics and New Algorithms **12**

3.1 Concurrent Game Models . . . 12
3.2 Coalition Logic . . . 15
3.3 Extended Coalition Logic . . . 16
3.4 Algorithms for Extended Coalition Logic . . . 24

4 StraTegic Verifier - A Model Checking Tool **29**

4.1 Imperfect Information Concurrent Game Structure . . . 29
4.2 Model Classes in STV . . . 29
4.3 Logics Implemented . . . 30
4.4 Algorithms . . . 30
4.5 CLI: Input and Output . . . 30

5 Implementation Details **34**

5.1 Formula Parser . . . 34
5.2 Extended Data Structure . . . 39
5.2.1 Example with Abstract Syntax Trees . . . 41
5.3 Implemented Methods in Global Model and Simple Model . . . 42
5.3.1 Methods for Positive Upgrades . . . 42
5.3.2 Methods for Negative Upgrades . . . 45
5.4 Coalition Logic Operator For Verification . . . 50

6 StraTegic Verifier - Extended Edition **52**

6.1 Time Spent on Execution . . . 61

7 Discussion **64**

8 Conclusions and Future Work **67**

References **68**

List of Algorithms

1 Determining New Transitions . . . 24
2 Executability Condition Positive Upgrade . . . 25
3 Determining Forcing Actions . . . 26
4 Executability Condition Negative Upgrade . . . 27

List of Figures

1 Pseudocode of the Atomic Swap Smart Contract . . . 7
2 The Initial CGM M . . . 13
3 CGM M^{+U} with a Positive Upgrade . . . 18
4 Initial CGM M and Upgraded CGM M^{+U} . . . 19
5 CGM M^{+U} with Marked Forcing Actions . . . 21
6 CGM $M^{+U,-U}$ with Positive and Negative Upgrade . . . 22
7 CGM M^{+U} with Positive Upgrade and CGM $M^{+U,-U}$ with both Positive and Negative Upgrade . . . 23
8 Generating Specifications STV . . . 31
9 Verification with Asynchronous Semantics in STV . . . 31
10 Example of Input File . . . 32
11 Verification with Synchronous Semantics in STV . . . 33
12 BooleanLiteral . . . 35
13 IntegerLiteral . . . 35
14 StringLiteral . . . 35
15 Coalition . . . 36
16 SimpleExpressionOperator . . . 36
17 SimpleExpression . . . 36
18 UpgradeFormula . . . 37
19 CoalitionExpression . . . 37
20 UpgradeList . . . 37
21 Upgrade . . . 38
22 Update . . . 38
23 Agent . . . 38
24 UpgradeType . . . 38
25 Class Diagram of Classes for Formula . . . 39
26 Abstract Syntax Tree of $\{[((p = False), a, (p = True))+]\}\langle\langle a\rangle\rangle(p = False)$ . . . 41
27 Abstract Syntax Tree 2: Simple Expression . . . 42
28 Determining New Transitions . . . 43
29 Test of Clashfreeness . . . 44
30 Updating the Model . . . 44
31 Determining Forcing Actions . . . 45
32 Finding Preserved Transitions in an Update . . . 46
33 Determining Preservation of Joint Forcing Actions . . . 47
34 Finding Transitions to Remove . . . 48
35 Updating the Instance of Simple Model with the Negative Upgrade . . . 48

36 Test of Clashfreeness for Negative Upgrade 49
37 Coalition Logic Operator Verification Method 50
38 Initial CGM M 52
39 Input File of the Atomic Swap Example 53
40 Output for Coalition Formula 54
41 CGM M^{+U} with Positive Upgrade 55
42 Output for Coalition Formula with Positive Upgrade 56
43 CGM $M^{+U,-U}$ with Positive Upgrade and Negative Upgrade 57
44 Output for Coalition Formula with Positive and Negative Upgrade Verifying Alice's Abilities 59
45 Output for Coalition Formula with Positive and Negative Upgrade Verifying Bob's Abilities 60
46 Upgrades for Testing Time Spent 61

List of Tables

1 Time of Verification of Formulas with Various Complexity 62

1 Introduction

Cryptocurrency has had increasing popularity since Bitcoin was first introduced in 2008 to users all over the world. Bitcoin's blockchain was developed primarily to buy and sell bitcoin, to have a trustless currency that was decentralised, anonymous and where governments could not interfere [Antonopoulos, 2017, chapter 1]. Bitcoin as a network has limited abilities to evolve, and many networks have been introduced in the later years. A network that contains all of Bitcoin's features and more is Ethereum. Ethereum was developed to be "The Worlds Computer". One of the bigger new features Ethereum introduced was the ability to develop programs that can be executed on the blockchain, so-called Smart Contracts. [Antonopoulos and Wood, 2018, chapter 1]

Cryptocurrency is open for all to buy and sell, and the code of Smart Contracts and networks are open-source. A Smart Contract might not do what is written in the program's requirements because of bugs or other semantic weaknesses. The openness is essential for trustlessness, although it also makes the source code's weaknesses accessible to malicious users.

There are monetary values in all Smart Contracts. To execute a Smart Contract, one has to pay transaction fees. Many Smart Contracts are programs that invest in crowdfunding. If malicious users of the Smart Contract know how to read the source code of a Smart Contract and exploits all the bugs they can find, the Smart Contract can be financially drained.
[Antonopoulos and Wood, 2018, chapter 9]

There are many famous hacks done by malicious users in cryptocurrency. One of the greatest hacks is "The DAO-attack", which was executed on the Ethereum network. One of the members of a crowdfunding organisation found a semantic hole in their Smart Contract and drained the fund for invested money. The member would have drained the entire fund if other members had not started draining the fund immediately once they found out what was going on so they could salvage parts of the fund. The malicious member drained 1/3 of the fund, and an enormous amount of money was lost. The solution became to split the main chain so the transactions done by the malicious user were not included in the new main chain, which is why there are two Ethereum coins, ETH and ETC. [Siegel, 2016]

A Smart Contract is immutable once deployed on the blockchain, and the particular version of the contract cannot be altered, which is necessary to preserve trustlessness. However, there are methods to change Smart Contracts by deploying new versions on the blockchain. While upgrading a Smart Contract might be necessary for changes in business structure or fixing bugs, it hurts the principle of trustlessness by requiring trust between the users and the developers. [Antonopoulos and Wood, 2018, chapter 7]

There is a lot at stake if the source code of a Smart Contract contains bugs due to the enormous amount of money invested and its immutable nature. As a result, it is important to test the Smart Contract extremely detailed. Many different methods are used, one of them being model checking [Almakhour et al., 2020, Tolmach et al., 2021].

While it is important to verify that a Smart Contract entails what it should before deployment, it is also important to verify that upgrades change only what they should. There are numerous

methods for verifying and testing Smart Contracts before deployment, although not as many for verifying changes to a Smart Contract.

The focus of this thesis will be on model checking upgrades of a Smart Contract, checking that only what is intended to be altered is, in fact, the only part of the Smart Contract that will be altered. Granting and restricting agents' dictatorial abilities will be the scope of this thesis.

The contribution is the extension of an explicit-state model checker called StraTegic Verifier (STV) [Kurpiewski et al., 2019a] with logics for granting and restricting agents' abilities. The extension is not trivial as it essentially constructs new models to verify formulas containing upgrades. The approach of the extension enables the addition of other temporal logics because the verification itself includes valid Concurrent Game Models, although this is outside of the scope and will not be implemented. The logic implemented will be Dictatorial Dynamic Coalition Logic, which can reason about upgrades in Smart Contracts modelled as Concurrent Game Models.

The remainder of the thesis will be structured as follows:
In Section 2, all background will be introduced, including the basics of blockchain, the Ethereum Network, Smart Contracts, formal verification and model checking. In addition, an example of a Smart Contract will be presented.

In Section 3, the required theory will be presented, including Concurrent Game Models and Coalition Logic. The example from the background section will be modelled as a Concurrent Game Model and described by the logics presented. The extended version of Coalition Logic, Dictatorial Dynamic Coalition Logic, will be elaborated, and how the example can be upgraded will be explored. Algorithms for model checking Dictatorial Dynamic Coalition Logic will be presented.

In Section 4, there will be a presentation of STV, the original version, before going into the implementation details of the extension in Section 5.

In Section 6, the resulting extension of STV is presented and elaborated before limitations and possible further extensions are discussed in Section 7. At last, in Section 8, the conclusion and future works will be elaborated.

2 Background

Since the first block of Bitcoin was mined on the 3rd of January 2009, until the 22nd of April 2023, the market cap of all cryptocurrencies has increased from zero to 1.2 trillion American Dollars [CoinGecko, 2023], approximately the same size as the market cap of Amazon, the fifth most valuable company in the world [Marketcap, 2023].

While the investments in cryptocurrencies are increasing and new technologies are emerging, it is more attractive for hackers to try to steal the investments, and several hackers have been successful. In 2021 and 2022, there were stolen cryptocurrencies for a total of 7.1 billion American Dollars. A severe part of the stolen monetary values is from hacks on DeFi Protocols, Decentralized Finance Protocols. A DeFi Protocol can be used as a bridge between blockchains and contains considerable amounts of cryptocurrencies, while bridging. An underlying Smart Contract, will be exploited by hackers if there are any flaws in it. In 2022 there were 3.1 billion American Dollars stolen through the exploitation of DeFi Protocols. [Team, 2023]

The list of exploitation of Smart Contracts is inexhaustible. Understanding the basics of blockchain, cryptocurrency and Smart Contracts are required to grasp why this problem exists.

2.1 Blockchain and Cryptocurrency

In Satoshi Nakamoto's whitepaper [Nakamoto, 2008] presenting Bitcoin and blockchain technology, the main motivation was the ability to pay with digital currency without needing to trust a third party to handle the transactions made. Since Bitcoin is decentralised and has no central authority or a single point of control, it cannot easily be attacked or corrupted. This section is based on [Antonopoulos, 2017, chapters 1, 9 and 10].

A blockchain is a secure decentralised database, while cryptocurrency is a digital asset. The assets are bought with traditional currency or other cryptocurrencies, and the transactions of the bought digital asset are stored on the blockchain. The acknowledgement of the blockchain is enough to determine which person owns which asset. The person is not identified, although every person who owns cryptocurrency has a unique address containing the assets.

As a decentralised database, blockchain can store transactions together in a secure manner. Being decentralised entails a peer-to-peer network instead of a client-server network, indicating that there is not a central point where all data is stored but several small points where sections are stored. The blockchain is built of blocks, which are batches of stored data. The blocks are connected from the latter to the former, including their parent's and their own hash. A hash is a unique identification of a block. A block can only have one parent, while a parent can have multiple children, which is the case when forking. A fork is when the main chain has several options for where the next block can be connected. If a fork appears, the longest chain will be the main chain, while everything added to the other branch of the chain will not be recognised by the main chain. The longer the chain is, the more secure is the chain. The latest blocks added to the chain can be altered for some time, while the blocks underneath are permanent.

In each block in the blockchain of Bitcoin, there are stored, on average more than 1900 trans-

actions. To identify a transaction, it contains a unique hash. An efficient way to prove that the transaction is included in a block, that a transaction has found place and is documented, is by using a binary hash tree, also called a Merkle tree which, in essence, contains cryptographic hashes.

Adding blocks to the blockchain is done by mining. The word mining in cryptocurrency refers to using computer power to first gather all transactions needed to build a block. The transactions are found in a transaction pool where all valid transactions are stored from they are initiated until they get stated in a block. Then, they add the parent's hash, followed by trying billions of hashes to find the hash that is the result of the Proof-of-Work-algorithm, which will be the new block's unique hash.

To be able to mine bitcoins, one needs a "mining rig", which is hardware and electricity to be able to run the scripts needed to mine bitcoin. While one "mining rig" is trying to figure out the hash, there are many others trying to figure out the hash for their own new block, which is all competitors to be the next block in the main chain, the first one that gets the correct answer will get their block added to the blockchain, and everyone else gets updated through the miner's network on which block is newly added and the competition starts over again mining for the next block.

If two miners find the correct hash at the same time, a fork will appear. When the other miners are updated with the message that the next block is mined, they will get different information on who won based on their distance in the mining network to the two winners. Due to the mismatch in the information that the miners now have, the miners will start mining for a new block with different parent hashes, some with one and some with the other. The winner of this second competition will decide which block of the two in the fork will actually become part of the main chain. The block that wins is the block with the matching hash to the parent hash of the newest block. The other block will not be recognised as a block on the main chain, and the transactions that are inside will be reallocated to the transaction pool.

The main incentive for mining blocks is to secure the blockchain. In addition, there are monetary incentives due to the cost of power and hardware. If one is able to mine a block that is added to the main chain, one receives all the transaction fees from the transactions within the mined block, and by May 2020, one receives 6.25 bitcoin. The number of bitcoin mined in each block will be cut in half every 4th year, and after 2140 no new bitcoin can be mined, and the monetary incentive will solely be the transaction fees.

The miners are responsible for determining the validity of transactions and blocks added to the blockchain. The rules of consensus are the foundation of the collaboration between all miners of bitcoin. They are also responsible for maintaining one consistent blockchain across the Bitcoin network. The decentralised consensus rules of Bitcoin are based on the intersection of four independent processes, namely, every miner's independent verification of transactions, every miner's aggregation of the validated transactions into new blocks, every miner's verification of the new blocks and the acknowledgement of the longest chain as the main chain.

Due to the decentralised consensus, it is not beneficiary to make up transactions that are not in the transaction pool because every miner independently validates the transactions. If one miner tries to put transactions that are not from the pool in the block they are mining, the whole block

will be rejected by the peers, and all the computer power used and time spent will be wasted.

While most miners' intention is to validate transactions and build blocks, there are some miners that want to attack the blockchain by altering transaction history, trying to get back money they have paid, so-called double-spending their money. To do so, they need to have so much of the total computer power that they can alter the blocks and mine new ones faster than the others can mine new blocks. This type of attack is a consensus attack and is called a 51% attack, but in reality, one does not need 51% of the computer power.

2.2 The Ethereum Network and Smart Contracts

In 2015 the first block of Ethereum was mined. Ethereum is a blockchain with the possibility of storing programs on the blockchain. While Ethereum has some of the same characteristics as Bitcoin, being an open blockchain, Ethereum can be used to build applications with built-in economic functions which Bitcoin can not. The programming language used by Ethereum is Turing Complete which makes the Ethereum platform function as a general-purpose computer. This section is based on [Antonopoulos and Wood, 2018, chapters 1 and 7]

Applications and programs built on Ethereum are called Smart Contracts. The Smart Contracts are stored on the blockchain, and to execute the contract, one has to pay transaction fees. The result of the Smart Contract will be stored on the blockchain as well. Once the Smart Contract is deployed on the blockchain, one cannot change the code of the instance of the Smart Contract. Hence, the Smart Contract is immutable once it is deployed on the blockchain.

In Smart Contracts, it is crucial to write code without bugs because if there is a bug in the contract after it is deployed on the blockchain, there is no way to fix the version of the Smart Contract. Thus, it is essential to check if the code is written with correct semantics.

There have been multiple attacks where vulnerabilities of Smart Contracts have been exploited. The most known attack is "the DAO" attack. DAO is an abbreviation for Decentralized Autonomous Organization and is an organisation/ company without hierarchical management. "The DAO" is a reference to a specific DAO that was initiated by the German startup Slock.it on the Ethereum blockchain. It was a crowd-funding platform where all members invested money, and after 28 days, they could vote on how the fund should invest the money in different projects. The platform was much more attractive than what the startup intended, and before being attacked, the platform had raised close to 150 million dollars. There was more than one weakness in the Smart Contract, but the one that was exploited by the attacker gave him the opportunity to withdraw close to 60 million dollars, which was more than 1/3 of the fund. [Siegel, 2016]

The vulnerability exploited was two lines of code that were swapped around so that the balance of the account was not checked before the user was finished with his withdrawal, and the attacker could withdraw money as many times as he wanted to before finishing, and he could withdraw as much as his initial balance every time he withdrew money in the same session. [Siegel, 2016]

Due to the enormous loss of 60 million dollars, the community around the Ethereum platform decided to act to be able to nullify the transaction the attacker had done. A new main chain

became the solution so that the fund got its money back, and the transaction was not included in the new main chain [Atzei et al., 2017]. The fork resulted in two chains and two coins, ETC and ETH.

Although a Smart Contract is immutable once it is deployed on a blockchain, there are architectural patterns that allow the developers to upgrade the Smart Contract, or parts of it, with newer versions. The architecture structures include proxy patterns, contract migration, data separation, strategy patterns and diamond patterns. In contract migration, one deploys a new version of the Smart Contract on the blockchain and transfers all data and values. In data separation, proxy patterns and diamond patterns, one divide the Smart Contract into two Smart Contracts. One Smart Contract stores the data, and the other contains the business logic and the code that will be executed. In data separation, the business logic Smart Contract is deployed and calls the Smart Contract storing the data, while in proxy patterns and diamond patterns, it is the opposite; the Smart Contract storing data is deployed to the blockchain, and the business logic Smart Contract is called. [Pratap, 2022]

There have been numerous events where Smart Contracts have been exploited, and it has led to substantial financial losses. To avoid future attacks, there has been research on how to check if the Smart Contracts entail the same as their specifications. There are different methods to verify Smart Contracts.

2.3 An Example of a Smart Contract

In [van der Meyden, 2018] the author inspects an example of atomic swaps. The example constructed involves two agents, Alice and Bob. They want to swap assets on a blockchain. Their options when swapping is to trust each other to both swap at the same time, to hand over the assets to a human third party and trust this third party to do the swap correctly or to hand it over to a computer third party, a Smart Contract, and trust that the program works as intended. While both trusting each other or trusting a human third party includes the risk of one of the parts acting maliciously, the computer program code is open for inspection by both Alice and Bob. As long as they believe the program will run as intended, it cannot have malicious intent. Although a Smart Contract cannot have an intent on its own, it is crucial to inspect that it works as intended. To ensure the program executes its code correctly, it can be formally verified using logics.

In the pseudocode of the atomic swap Smart Contract is shown in Figure 1, it is stated that there are two agents, Alice and Bob, two assets, `a` and `b`, and two boolean variables, `depositedA` and `depositedB`, which indicates that Alice has deposited her asset or Bob has deposited his asset. The boolean variables are set to false in the initial state. Both Alice and Bob can deposit their assets after initialisation.

If both Alice and Bob deposit their assets, the contract will go into the finalising state where the assets get swapped and the boolean variables it set to false once again. If only Alice or only Bob deposits their asset, they will be able to cancel their deposit so they get theirs back if the other agent changes their mind and does not want to swap.

```
Contract Escrow
{
depositedA, depositedB : Bool

initialise { depositedA := False ; depositedB := False }

depositA { if sender = Alice and value = a then depositedA := True }

depositB { if sender = Bob and value = b then depositedB:= True }

finalise { if depositedA and depositedB
                then { depositedA := False; depositedB := False;
                        send(a, B); send(b, A) } }

cancelA { if depositedA then { depositedA := False; send(a,A) } }

cancelB { if depositedB then { depositedB := False; send(b,B) } }
}
```

Figure 1: Pseudocode of the Atomic Swap Smart Contract

The presented Smart Contract will be the basis of the examples given trough-out the thesis.

2.4 Testing and Verification of Smart Contracts

There are several methods and tools available, at the Ethereum Network website [Preston, 2023], for testing and verifying Smart Contracts before deployment. There are available tools and libraries for both automated and manual testing. Automated testing includes unit testing, integration testing and property-based testing, which include static analysis and dynamic analysis. Manual testing includes testing the Smart Contracts on a local blockchain or a testnet.

Verifying includes verification of source code and formal verification. Verification of source code compares the source code of a Smart Contract and its compiled bytecode to verify the absence of differences between what the source code states and how the code is actually executed. Formal verification verifies if the source code really does what is intended.

In addition to testing and verification, one can get an independent code review such as an audit from an auditing firm, or one can offer a bug bounty to the community, a financial reward for finding bugs in the Smart Contract, and report it.

Formal verification is a recommended technique to improve the security of Smart Contracts [Corwin, 2023]. While testing cannot exclude the possibility of bugs, formal verification can provide a mathematical proof of correctness.

2.4.1 Formal Verification of Smart Contracts

Formal verification is a method based on mathematical techniques for specification and verification of systems. The techniques are often supported by computer-based tools and can help verify and specify both systems of software and hardware. The formal analysis will take a computer-based mathematical model of a system and specifications of the same system, as input, and will verify if the model meets the specifications or not. The different techniques works best for verification in specific fields. [Holzmann and Peled, 1995]

A survey of different methods to verify Smart Contracts is given in [Almakhour et al., 2020]. The authors investigate two different aspects of verification, namely correctness of Smart Contracts and security assurance of Smart Contracts. Within the aspect of verification of correctness there are two approaches: formal verification and programming correctness. In this particular survey they focus on the formal verification approach because it is more rigorous and reliable then verification by programming correctness. The survey of formal verification includes theorem proving, model checking, and runtime verification. In addition, the survey includes an overview of software used to conduct the verification.

Theorem proving entails mathematically modeled systems and a set of properties needed to be proven. The method derives new theorems and lemmas needed for the proof and thereby prove the correctness of the system. This method can be applied to different types of system as long as they can be modeled mathematically. The theorem prover can be either interactive, automated or a hybrid. The limitations of theorem proving is that there are no fully automated theorem provers, it is needed a lot of human investment to be able to prove theorems and there is needed a deep understanding for this particular method to make it work.

Model checking is a method where one model a system into a finite-state model and check if specifications made as properties in formal logic is satisfied or not. There are six different tools surveyed where three of them is dynamic analysis and the remaining three is static analysis. Dynamic analysis indicates that the program has been executed and each state has been analysed while executing. Static analysis indicates that the source code or byte code has been examined and the program has not been executed.

Runtime verification performs the analyses based on execution of a system while extracting information from a running system and use it to detect how the system behave and decide if it satisfies or violates the correctness properties stated. Runtime verification is not a widespread method when verifying Smart Contracts.

The other aspect of verification covered in [Almakhour et al., 2020] is analysis of vulnerabilities, including the following methods: symbolic execution, abstract interpretation and fuzzing. The survey also includes some platforms used when executing this type of verification within the methods stated above. Analysis of vulnerabilities entails checking for specific known vulnerabilities and can avoid mistakes, while it is ineffective to analyse complex Smart Contracts. When conducting this type of analysis some of the vulnerabilities could easily get ignored because the analysis is non-exhaustive. Considering the above, analysis of vulnerabilities will not be the focus of this thesis.

The authors of [Almakhour et al., 2020] conclude, in general, that the state of the art methods

and software can only verify simple Smart Contracts.

In [Tolmach et al., 2021] the authors provide an overview of formal models, specifications and verification tools of Smart Contracts that is presented in the literature. They divide the modeling formalism into two categories, contract-level and program-level, based on the level of abstraction and analysis performed. The contract-level approach is related to high-level behavior of Smart Contracts and does not usually consider technical details, while program-level approaches are based on doing analysis of the contract implementation, such as source code. They make another distinction between the two categories in formal specifications, where model-oriented specifications are in regards to contract-leveled modeling and property-oriented specifications concern program-leveled modeling.

The users, Smart Contracts and blockchain state is in focus when modeling at the contract-level. The Smart Contracts are inspected on how they interact with the users and what happens in the blockchain due to the interaction. The formal modeling applied is process algebras, state-transition systems and set-based methods.

The model-oriented specifications are expressed in different types of logic. The logics used in the literature are temporal logics, dynamic logic, deontic and defeasible logic and other logics. The temporal logics which express properties of Smart Contracts over time includes Linear Temporal Logic, Computation Tree Logic, Alternating Temporal Logic and Strategic Logics, where the two latter is extensions of Computation Tree Logic.

The majority of contract-level models are verified using model checking applied on mainly finite-state transition systems and given temporal logic specifications, the technique verify systems automatically. While model checking only can be applied to finite-state systems, theorem proving can verify infinite-state systems.

Program-level models focus on understanding the internal details when the Smart Contract executes. The models can be abstract syntax trees, bytecode, control flow graphs or program traces and are analysed by abstract syntax tree-level analysis, control-flow automata and program logics.

The property-oriented specifications for program-level models are Hoare-style specifications, often used in a combination with program logics or are implemented in the actual source code of the Smart Contract.

Symbolic execution, program verification, runtime verification and testing is mostly applied on program-leveled models. Symbolic execution detects vulnerable patterns in bytecode that is defined before execution and it it is difficult to pick up on vulnerabilities that is not predefined. Program verification often uses Hoare-style specifications to verify. Runtime verification verifies one execution trace at the time unlike the other techniques in this survey.

Correctness of Smart Contracts can prove if the system is correct, while security assurance is vulnerable because one have to state the vulnerabilities before checking, and one cannot be certain that all are covered. Formal verification is more rigorous and reliable then verification by programming correctness. The example from [van der Meyden, 2018] has a finite-transition system as a model,

temporal logics as specification and uses model checking as verification technique. The authors conclude that temporal logic is more suitable than Hoare-style specifications for the atomic swap example. Based on [van der Meyden, 2018] conclusion, verification with contract-leveled models and model-oriented specifications will be the right focus for this thesis. Because theorem proving cannot be done automatically and needs human expertise, the focus will be on model checking.

Model checking is an established field that is researched thoroughly. The method is applied in various fields and tools. There are open source tools available for altering and extension to new logics. The formal verification method used in this thesis will be model checking as a result of the research already done in the field, and since it is a rigorous and reliable method of testing Smart Contracts.

2.4.2 Model Checking in General

Model checking is a method that analyses dynamic systems by using a computer. The systems has to be modeled as state-transition systems. Model checking is widely used in industry to verify software and hardware. The method uses logic for bug finding and it can check the whole system, every possible state and which states it goes through after each other. [Clarke et al., 2018]

A model checker is often built on three insights, modeling, specification and algorithms. The modeling has to be a finite state-transition graph which provides the formalism needed to describe a finite-state system. The specification used is temporal logic which describe the correctness of properties in the state-transition system, and provides a natural framework. Algorithms often provide counterexamples if the formula tested is false in the model and also determines if the formula tested is true or false. [Clarke et al., 2018]

There are two main types of model checking, explicit-state model checking and symbolic model checking. Symbolic model checking is the initial version where all states are checked for the specific requirements. Explicit-state model checking only checks specific states where the specification can be either proven right or wrong. Explicit-state model checking is less accurate although much faster. [Merz, 2000]

2.4.3 Model Checking Smart Contracts

There are numerous model checking approaches made especially for verifying Smart Contracts. In [Almakhour et al., 2020] they survey six approaches, part of their result is that VERISOL and Model checker for Smart Contracts are both able to verify Smart Contracts, although due to the running time of VERISOL the authors of the survey does conclude that it is unsuitable for verifying complex Smart Contracts.

Other model checking tools can also check Smart Contracts, such as MCMAS. MCMAS is a model checker for multi-agent systems, [Nam and Kil, 2022] has tested it and concluded that one can in fact verify Smart Contracts with MCMAS. In addition to MCMAS, MCK is developed for model checking knowledge [Gammie and Van Der Meyden, 2004] although, it is also used to verify Smart Contracts in [van der Meyden, 2018].

All tools surveyed are intended to be used before deploying Smart Contracts to the blockchain

so that errors are revealed before execution [Almakhour et al., 2020]. The tools does not open for reasoning about changes after the Smart Contract is executed on the blockchain, which would be needed for verification before a new instance of the Smart Contract would be deployed on the blockchain.

Now, back to the example of Alice and Bob. If Alice or Bob were granted new abilities, there would have to be deployed a new instance of the Smart Contract on the blockchain or parts of the callable Smart Contracts would have to be changed.

In general, if one were to change the functionality of a Smart Contract, in the context that either one of the parts get expanded authority to do something or it decreases their authority, the Smart Contract would have to be upgraded. It is likely that a Smart Contract at some point will go through changes as the business structure evolves with time.

An extension of the atomic swap example would be to research what would happen if the abilities of Alice or Bob would be extended and they would be able to force an action without the others agreement, or if the abilities of either Bob or Alice would be restricted. To be able to reason about these changes in a Smart Contract, [Galimullin and Ågotnes, 2021] has suggested a logic called Dictatorial Dynamic Coalition Logic. The Dictatorial Dynamic Coalition Logic is divided into two different logics, one of them as the positive alternative for extending agents' abilities and one negative alternative for revoking the agents abilities.

3 Models, Logics and New Algorithms

Model checking, as previously mentioned, consists of three main parts, models, specifications and algorithms. The model checking problem is defined as below [Merz, 2000].

Definition 3.1 (The model checking problem)**.** Given one finite model M with an initial state s and one formula (a specification) φ, the model checking problem is the determination of whether or not the formula holds in the model, $(M, s) \models \varphi$.

In this section it will get clear which type of model will be used, how that type of model is defined and how our example is defined in such a model. Following the logics for specifications will be defined and applied on our example of an atomic swap with Alice and Bob and specification-examples will be given. The last part will be to look at the extended logic Dictatorial Dynamic Coalition Logic and how the models change when they are updated and how the specifications will be written and understood.

3.1 Concurrent Game Models

All formal definitions of Concurrent Game Models and Coalition Logic (not extended) are slightly modified although borrowed from [Ågotnes et al., 2015].

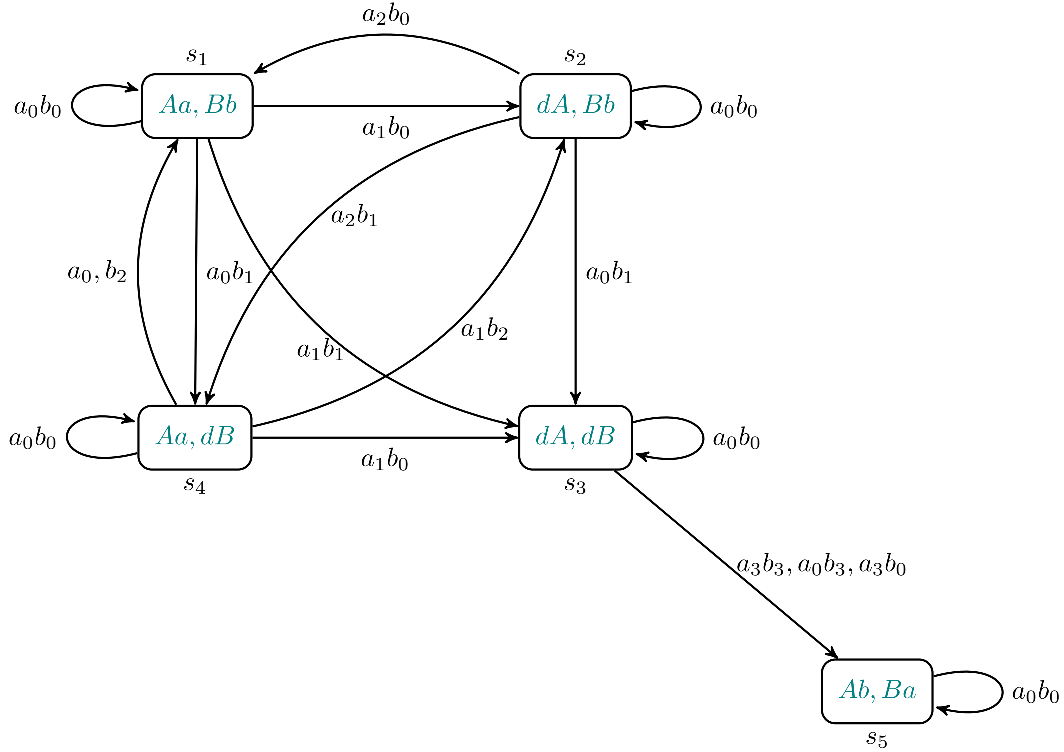
Definition 3.2 (Concurrent Game Structure)**.** A Concurrent Game Structure is a tuple $\mathcal{S} = (\mathrm{Ag}, \mathrm{St}, \mathrm{Act}, \mathrm{act}, \mathrm{out})$ consisting of:

- a finite, non-empty set of *agents* Ag = {a, ..., k}; subsets of Ag are called *coalitions*;
- a non-empty set of *states* St;
- a non-empty set of *actions* Act;
- an *action manager* function $\mathrm{act} : \mathrm{Ag} \times \mathrm{St} \rightarrow \mathcal{P}(\mathrm{Act})$ assigning to every agent a and a state q a non-empty set of actions available for execution by a at the state q.
  An *action profile* is a tuple of actions $\alpha = \langle \alpha_1, ..., \alpha_k \rangle \in \mathrm{Act}^k$. The action profile is *executable at the state* q if $\alpha_i \in \mathrm{act}(i, q)$ for every i $\in$ Ag. We denote by $\mathrm{act}(q)$ the subset of $\Pi_{a \in \mathrm{Ag}} \mathrm{act}(a, q)$ consisting of all action profiles executable at the state q.
- a *transition function* out that assigns a unique *outcome state* $\mathrm{out}(q, \alpha)$ to every state q and every action profile α which is executable at q.

The definition above indicates that there is at least one agent, there is at least one state and there is at least one action. There is an action manager function called act, that outputs a set of states which is available for every agent in every state. All these available actions are included in an action profile. The input of the transition function out is the current state and the action profile for the agents, and its output is an unique outcome state to every action profile and every current state.

Definition 3.3 (Concurrent Game Model)**.** A *Concurrent Game Model* (CGM) is a Concurrent Game Structure with a labeling such as $L : \mathrm{St} \rightarrow \mathcal{P}(\mathrm{PROP})$ of states with sets of atomic propositions from a fixed set PROP. An atomic proposition is a statement which can be either true or false. The labels describe which atomic propositions is true in which state.

Intuitively, a CGM includes nodes which describes the states of the system through labels that express if propositions are true or false. The arrows going from a node to another indicates which abilities each agent has in that particular state and what will be the outcome-state as a result of the actions done by the agents. There are similarities between a Concurrent Game Model and a Smart Contract. The Smart Contract's state is affected by the actions of the users, and a Concurrent Game Model's transitions are determined by what actions the agents choose. A CGM of the atomic swap example, with Alice and Bob, is depicted in Figure 2.



Actions:

i0:	do nothing
i1:	deposit i's asset
i2:	cancel i's deposit
i3:	finalise swap

Figure 2: The Initial CGM M

The above CGM contains five states. The initial state s_1 contains the propositions Aa and Bb. The other propositions are not mentioned in state s_1 meaning that they are not true in that particular state. Proposition Aa is an abbreviation for Alice holding Alice's initial asset and Bb is for Bob holding his initial asset. In s_1 both Alice and Bob have two abilities, they can both choose to deposit their asset or choose to do nothing. Action a_0 indicates that Alice is doing nothing, while action a_1 indicates that Alice deposits her asset. Equally for Bob action b_0 indicates that Bob is doing nothing, while action b_1 indicates that Bob deposits his asset.

Which state will be the next is a result of what Alice and Bob choose to act out in state s_1. If both deposited their asset the next state will be state s_3, the state where the propositions dA and dB is true which entails that Alice has deposited her asset and Bob has deposited his asset, respectively. When in state s_3 Alice and Bob has two abilities each, the first action is to do nothing a_0, b_0 and the second is to finalise the swap with actions a_3 and b_3. As long as one of them choose to finalise, the system will transit to state s_5 where the atomic swap is finished. The propositions Ab and Ba indicates that Alice holds Bob's initial asset and Bob holds Alice's initial asset. The only ability they have in state s_5 is to do nothing, illustrated with actions a_0 and b_0.

Moving back to state s_1, the initial state, there are two more states that can be the next state beside s_3. Which of these three states that will be the next state in the transition system depends on the actions chosen by Alice and Bob in the initial state. If Alice choose to deposit her asset while Bob choose to do nothing, resulting in actions a_1 and b_0, the next state will be state s_2. In state s_2 the true propositions are dA and Bb and the propositions indicates that Alice has deposited her asset, and that Bob holds his initial asset.

In state s_2 Bob has the same abilities as he has in state s_1, namely do nothing, action b_0 and deposit his asset with action b_1. Alice on the other hand has one ability in state s_2 that she does not have in the initial state, s_1. Alice can choose to do nothing with action a_0 or she can cancel her deposit with action a_2. If Bob does nothing, action b_0 and Alice does nothing, action a_0 the transition system will move in a self-loop to the same state, s_2 as it was already in. If Bob does nothing, action b_0, and Alice cancels her deposit, action a_2, the next state will be the initial state s_1. On the other hand, if Bob deposits his asset when the transition system is in state s_2, and then depending on Alices chosen action, there are two different outcomes. If Alice chooses action a_0 do nothing the system will transit to state s_3. If Alice chooses action a_2 cancel deposit, the system will transit to state s_4, the state where propositions Aa and dB is true, entailing that Alice got her initial asset back and Bob has deposited his asset.

The third outcome of the choices made in the initial state, s_1, is that Alice does nothing, action a_0 and that Bob deposits his asset, action b_1, resulting in a transit from state s_1 to state s_4. In state s_4, the true propositions are Aa and dB which entails that Alice holds her initial asset while Bob has deposited his asset. State s_4 and s_2 is similar although the roles are opposite. Bob has two abilities he can act on in sate s_4, he can do nothing, action b_0 or he can cancel his deposit b_2, although the resulting next state also depends on what Alice choose to do. If Alice choose to do nothing, action a_0, and Bob chooses to do nothing, there will be a similar self-loop when the transition happens, and the system will end up in the same state as it was already in, state s_4. If Alice does nothing, action a_0 and Bob cancels his deposit, action b_2, the next state will be the initial state s_1, and it will all start over. If Alice on the other hand deposit her asset when in state

s_4, with action a_1 the next state will either be state s_3 or s_2. If Bob chooses to do nothing the system will transit to state s_3 where the propositions dA and dB are true. If Bob chooses to cancel his deposit in state s_4 while Alice chooses to deposit her asset, the system will transit to state s_2, a state where the propositions dA and Bb is true, entailing that Alice's asset is deposited, and Bob holds his initial asset.

To be able to interpret a CGM we need logic, for this we will use Coalition Logic. To verify the model, formal specifications are needed, and to make the formal specifications it is essential to mention some logic used for this particular purpose.

3.2 Coalition Logic

Coalition Logic is a multi-modal logic introduced by Marc Pauly that captures coalitional abilities in strategic games [Pauly, 2002]. Coalitional abilities is the abilities that one or more agents have when cooperating in a group, we call such a group a coalition. The following section will go through the formal definitions of Coalition Logic before elaborating what the definitions entails intuitively. Following will be to look at formulas that can be used as specifications in our atomic swap example with Alice and Bob.

Definition 3.4 (Language of Coalition Logic)**.** The formulas of Coalition Logic is defined recursively as follows:

$$\varphi \ := \ p \mid \neg\varphi \mid \varphi \vee \psi \mid \langle\!\langle C \rangle\!\rangle \varphi,$$

where p ranges over PROP and is a proposition, and $C \subseteq \mathrm{Ag}$ and is a coalition of agents.

The formal interpretation of $\langle\!\langle C \rangle\!\rangle \varphi$ in a state s_1 of a Concurrent Game Model M is:

$(M, s_1) \models \langle\!\langle C \rangle\!\rangle \varphi$ iff $\mathrm{out}(s_1, \alpha_C) \subseteq [\![\varphi]\!]_M$ for some $\alpha_C \in \mathrm{act}(C, s_1)$,
where $[\![\varphi]\!]_M := \{s_1 \in \mathrm{St} | (M, s_1) \models \varphi\}$.

The formula $\langle\!\langle C \rangle\!\rangle \varphi$ states that in the current state, the coalition C has the ability to guarantee an outcome where φ is true.

When model checking, the program will verify if the formal specifications are true or false in the model. The formal specifications are made by using logics and describes part of the model. The model checker will tell if the formula is true or not. Below are some examples of formulas, in the language of Coalition Logic, that describes parts of the model in Figure 2 from a given state. Alice and Bob is denoted as a and b. All the formulas below are true in the CGM in Figure 2.

$$(M, s_1) \models \langle\!\langle a, b \rangle\!\rangle (dA \wedge dB) \tag{1}$$

In Figure 2 of CGM M in state s_1, if Alice and Bob cooperate in a coalition they can guarantee that they will next be in a state where both of them have deposited their asset.

$$(M, s_1) \models \langle\!\langle a \rangle\!\rangle (dA \wedge (dB \vee Bb)) \tag{2}$$

In the model M, in state s_1, Alice alone can guarantee that she deposit her asset, she cannot guarantee that Bob will.

$$(M, s_1) \models \langle\langle b \rangle\rangle(dB \wedge (dA \vee Aa)) \tag{3}$$

In the model M, in state s_1, Bob can alone guarantee that that he deposit his asset, but cannot guarantee that Alice will.

$$(M, s_1) \models \langle\langle a, b \rangle\rangle\langle\langle a \rangle\rangle(Ab \wedge Ba) \tag{4}$$

In the model M, in state s_1, Alice and Bob can guarantee a state where it is true that Alice can force a state where Alice holds Bob's initial asset and Bob holds Alice's initial asset.

$$(M, s_1) \not\models \langle\langle a \rangle\rangle\langle\langle a, b \rangle\rangle(Ab \wedge Ba) \tag{5}$$

In the model M, in state s_1, it is not the case that Alice can guarantee that in the next state Alice and Bob can force a state where Alice and Bob has swapped their assets.

Coalition Logic can be used for verification of Smart Contracts, although there is no way to capture upgrades. Hence, new logics have been introduced in the literature for granting and revoking abilities, specifically dictatorial abilities. There are a few different approaches in this particular field. Two of them extends Coalition Logic and one uses Concurrent Game Models [Galimullin and Ågotnes, 2021] while the other uses action models [Galimullin and Ågotnes, 2023]. The focus in this thesis will be within the extension using Concurrent Game Models.

3.3 Extended Coalition Logic

In [Galimullin and Ågotnes, 2021] the authors provide extensions of Coalition Logic called Dictatorial Dynamic Coalition Logic (DDCL), both a positive version where dictatorial powers are being granted and a negative version where dictatorial powers are being revoked. Dictatorial powers are abilities of agents that can force a specific state to be true in the next step, no matter what other agents does. All formal definitions in this section are borrowed from [Galimullin and Ågotnes, 2021].

Definition 3.5 (Language of Positive Dictatorial Dynamic Coaliton Logic)**.** The language of positive DDCL is given by the following BNF:

$$\begin{aligned} \varphi &::= p \mid \neg\varphi \mid (\varphi \wedge \varphi) \mid \langle\langle C \rangle\rangle\varphi \mid [+U]\varphi \\ +U &::= (\varphi, a, \varphi)^+ \mid (\varphi, a, \varphi)^+, +U \end{aligned}$$

$[+U]\varphi$ states in natural language that after a positive update $+U$, φ will be true.

$+U$ is the following list/set

$$+U := \{(\psi_1, a_1, \varphi_1)^+, ..., (\psi_n, a_n, \varphi_n)^+\}$$

Each tuple $(\psi, a, \varphi)^+$ in the set $+U$ includes three elements, two formulas φ and ψ, and an agent a. For each state where ψ is true, in the updated model, agent a will be granted a new action such that regardless of what other agents choose to do, the target state will be a state where φ is true.

Because an agent with a dictatorial power can force specific states from the current state, two agents cannot have dictatorial powers over the same state, the authors have addressed this issue by only considering "clash-free" updates and provides the following semantic interpretation of the positive updates:

Let $(M, s) =$ (Ag, St, Act, act, out, L) be a CGM, with s as initial state.

$$(M, s) \models [+U]\varphi \text{ iff } +U \text{ is executable in } M \text{ implies } (M^{+U}, s) \models \varphi$$

where $M^{+U} = (\text{Ag, St, Act}^{+U}, \text{act}^{+U}, \text{out}^{+U}, \text{L})$ is the updated model.

For a update to be executable in the CGM M, the following definition is provided:

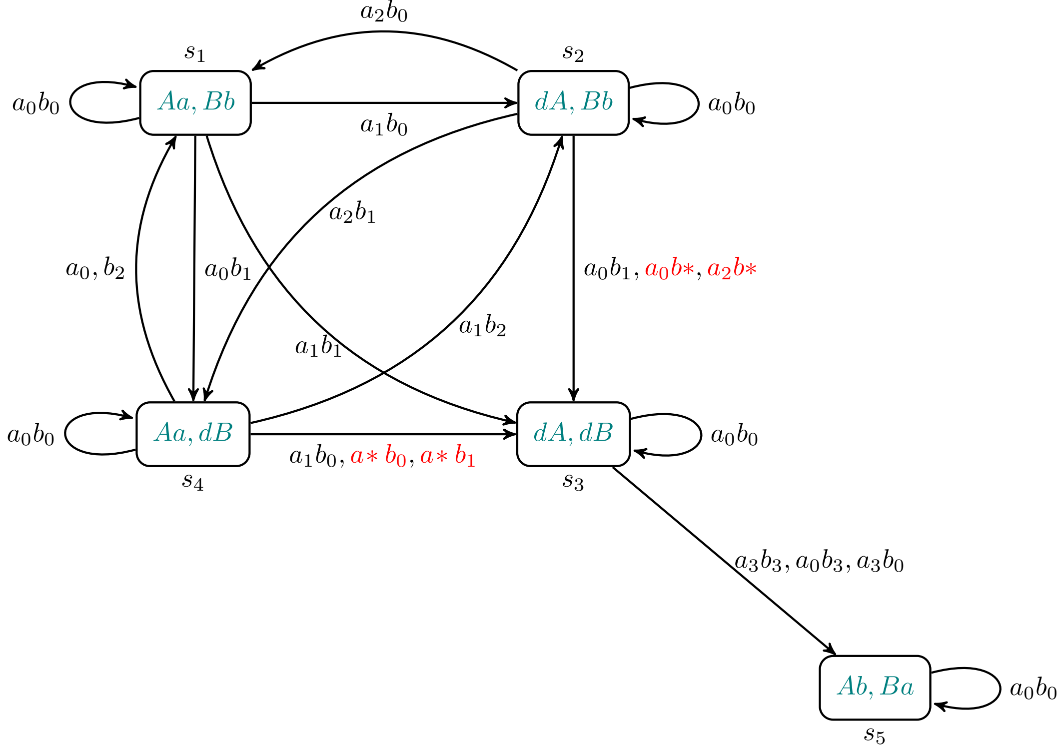
$+U$ is *executable* in M iff for all $(\varphi, i, \psi)^+, (\chi, j, \tau)^+ \in +U : [\![\varphi]\!]_M \cap [\![\chi]\!]_M = \emptyset$ whenever i $\neq$ j.

The definition of executable updates states that there cannot be two different agents with dictatorial powers in the same state. The semantic interpretation of the positive updates states that for a CGM M and a state s, after a positive upgrade $[+U]$, φ is true if the updates in the upgrade is executable and in the updated model in state s, φ is true.

The CGM, in Figure 3, is an updated version of the earlier shown CGM M. Both Alice and Bob have gotten dictatorial powers such that they can both force a state where both have deposited from the state where only the other has deposited and not themselves. Since the dictatorial powers they are granted are granted in different states for different agents the updates in the upgrade are executable. The granting of dictatorial powers is written as follows:

$$+U = \{((dA \wedge Bb),\ b,\ (dA \wedge dB))^+, ((Aa \wedge dB),\ a,\ (dA \wedge dB))^+\}$$

In the CGM the new actions are written as $a*$ and $b*$.



Actions:

i0:	do nothing
i1:	deposit i's asset
i2:	cancel i's deposit
i3:	finalise swap

Figure 3: CGM M^{+U} with a Positive Upgrade

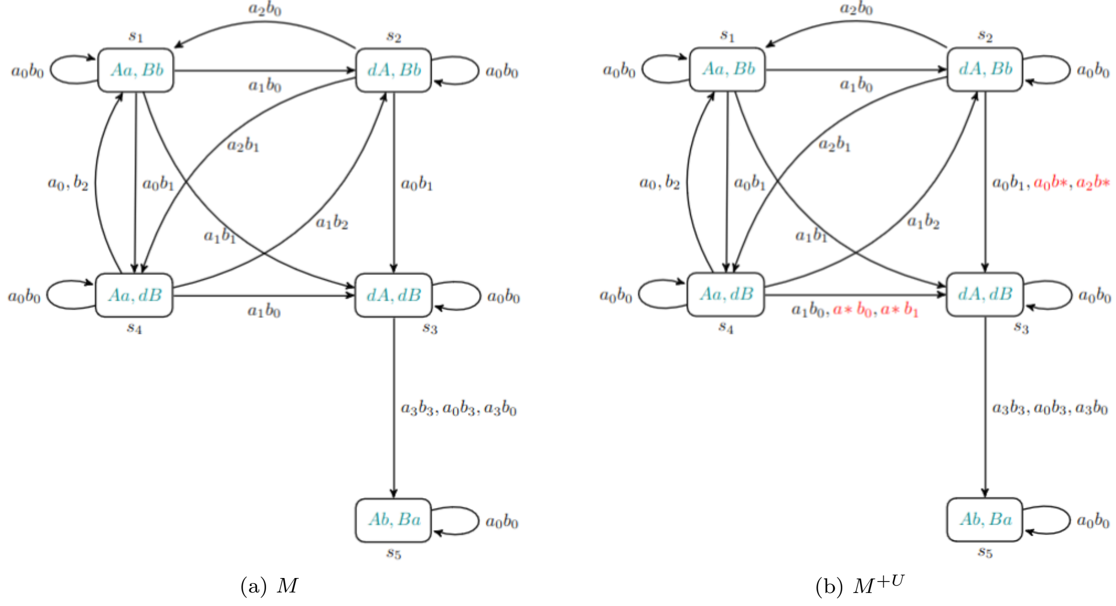
When comparing the two versions of the model one can see that Alice has gotten forcing actions in state s_4 and can force an outcome to be state s_3, while Bob has gotten forcing actions in state s_2 and can force the next state to be state s_3 based on the propositions in the states.

Before the positive upgrade, in state s_2, Bob could not force a transition to a state where both Alice and Bob had deposited their asset, as in formula 6, while after the positive upgrade it is the case that Bob can force a state where both have deposited their asset when in a state where only Alice has deposited her asset while Bob still holds his, as seen in formula 7.

$$(M, s_2) \not\models \langle\langle b \rangle\rangle (dA \wedge dB) \tag{6}$$

$$(M, s_2) \models [+U]\langle\langle b \rangle\rangle (dA \wedge dB) \tag{7}$$

Prior to the upgrade Bob could cancel his deposit in any state where he had deposited his asset



(a) M (b) M^{+U}

Figure 4: Initial CGM M and Upgraded CGM M^{+U}

and Alice had not, as shown in formula 8. After the upgrade the formula is not true as shown in formula 9.

$$(M, s_4) \models \langle\langle b \rangle\rangle (Bb \wedge (Aa \vee dA)) \tag{8}$$

$$(M, s_4) \not\models [+U]\langle\langle b \rangle\rangle (Bb \wedge (Aa \vee dA)) \tag{9}$$

Since both Alice and Bob got the same forcing action the formulae holds for Alice in her respective states, as shown in formulae 10-13.

$$(M, s_4) \not\models \langle\langle a \rangle\rangle (dA \wedge dB) \tag{10}$$

$$(M, s_4) \models [+U]\langle\langle a \rangle\rangle (dA \wedge dB) \tag{11}$$

$$(M, s_2) \models \langle\langle a \rangle\rangle (Aa \wedge (Bb \vee dB)) \tag{12}$$

$$(M, s_2) \not\models [+U]\langle\langle a \rangle\rangle (Aa \wedge (Bb \vee db)) \tag{13}$$

In [Galimullin and Ågotnes, 2021] the authors both define the positive upgrade DDCL+ and a negative upgrade version DDCL-. The negative version revokes dictatorial powers from an agent and will be looked into next. The formal definitions are borrowed from [Galimullin and Ågotnes, 2021].

Definition 3.6 (Language of Negative Dictatorial Dynamic Coalition Logic)**.** The language of the negative DDCL is given by the following BNF:

$$\varphi \ ::= \ p \mid \neg\varphi \mid (\varphi \wedge \varphi) \mid \langle\langle C \rangle\rangle \varphi \mid [-U]\varphi$$
$$-U \ ::= \ (\varphi, a, \varphi)^{-} \mid (\varphi, a, \varphi)^{-}, -U$$

where $p \in \text{PROP}, a \in \text{Ag}, C \subseteq \text{Ag}$.

$[-U]$ should be read as "after negative update $-U$, φ is true". The list of negative updates $-U$ is treated as a set in the same way as in the positive updates. However, in the positive updates the set contains tuples of actions that will be added to the model, while in the negative update set the tuples included in the set are what will be preserved in the model and all dictatorial powers in the model that is not in the negative update set will be removed.

The clauses in the negative update set is compared with the set of forcing actions, the notation used for this type of set is $\mathfrak{f}(agent, state) = \{action^{state,nextstate}\}$. E.g. in Figure 3 the forcing action set of Alice in state s_4 is $\mathfrak{f}(a, s_4) = \{a{*}^{s_4,s_3}\}$, meaning that no matter what Bob chooses to do, Alice can force state s_3 from state s_4 by choosing action $a*$. For the forcing actions to be in the negative updated model the updated forcing action set will include actions that fits with the clauses in the $-U$ set and that was in the original set of forcing actions.

As in the positive update the negative update has to be executable in the model. [Galimullin and Ågotnes, 2021] call $-U$ *executable* in M iff for all $a \in \text{Ag}$ and $s \in \text{St}$ at least one of the following conditions is true:

- $|\mathfrak{f}^{U-}(i, s)| \neq 0$, or
- $\exists \alpha_i \in act(i, s) : \alpha_i \notin \mathfrak{f}(i, s)$

Intuitively, the definition states that there has to be at least one action, either non-forcing, or a forcing action that is allowed by the clauses in $-U$ to be preserved so that the agent still have a action to do in each state.

The update-operator $[-U]$ is defined as follows:
Let $(M, s) = (\text{Ag, St, Act, act, out, L})$ be a CGM, with an initial state s.

$$(M, s) \models [-U]\varphi \text{ iff } -U \text{ is executable in } M \text{ implies } (M^{-U}, s) \models \varphi$$

where $M^{-U} = (\text{Ag, St, Act}^{-U}, \text{act}^{-U}, \text{out}^{-U}, \text{L})$ is the updated model.

The definition of negative updates indicates that if in the model M in state s, the negative update is such that φ is true, then if the negative update is executable, the updated model, will in state s be such that φ is true.

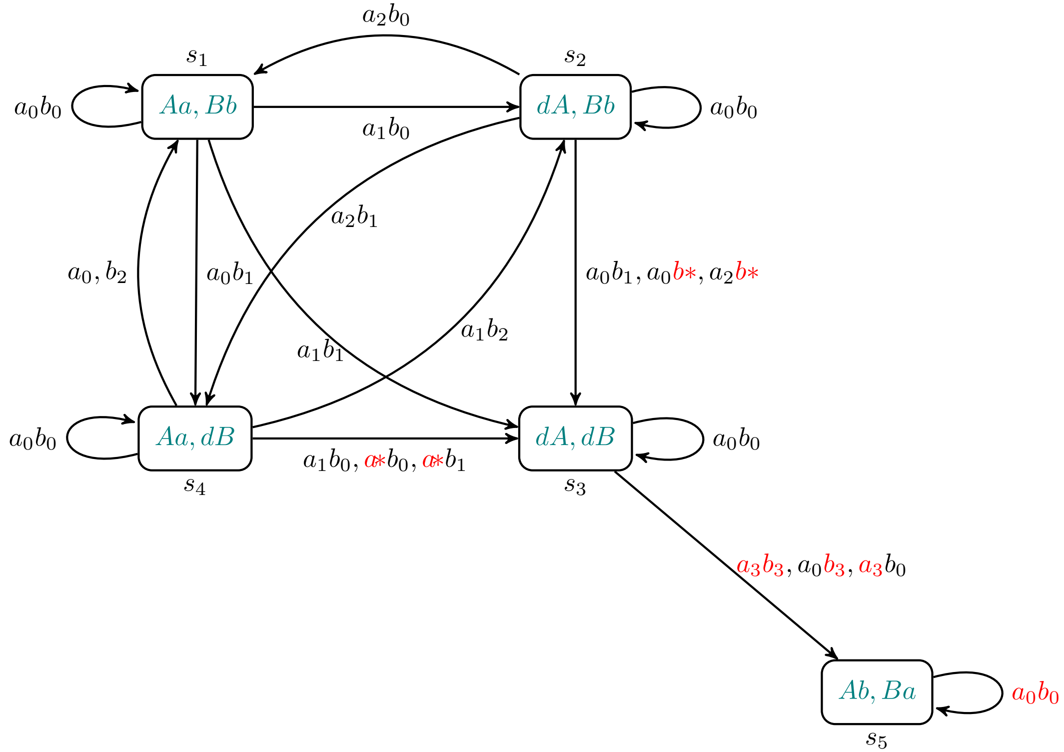
Now, back to our example. Alice and Bob have now both dictatorial powers in the state where the other has deposited while themselves has not, to a state where they both have deposited. There is now decided that Alice should not be able to have this dictatorial power, and the negative upgrade

that will now happen will revoke the powers from Alice. The revoking of dictatorial powers is written as follows:

$$-U = \{((dA \wedge Bb), b, (dAdB))^-, ((dA \wedge dB), a, (Ab \wedge Ba))^-,$$

$$((dA \wedge dB), b, (Ab \wedge Ba))^-, ((Ab \wedge Ba), a, (Ab \wedge Ba))^-, ((Ab \wedge Ba), b, (Ab \wedge Ba))^-\}$$

While the positive upgrade states what is granted, the negative upgrades state what will be preserved. All transitions with non-forcing actions will be preserved and the stated transitions with forcing-actions will be preserved. There are more forcing actions in M^{+U} than the forcing actions granted in that particular upgrade. The CGM in Figure 5 is the model M^{+U} with marked forcing actions for each agent.

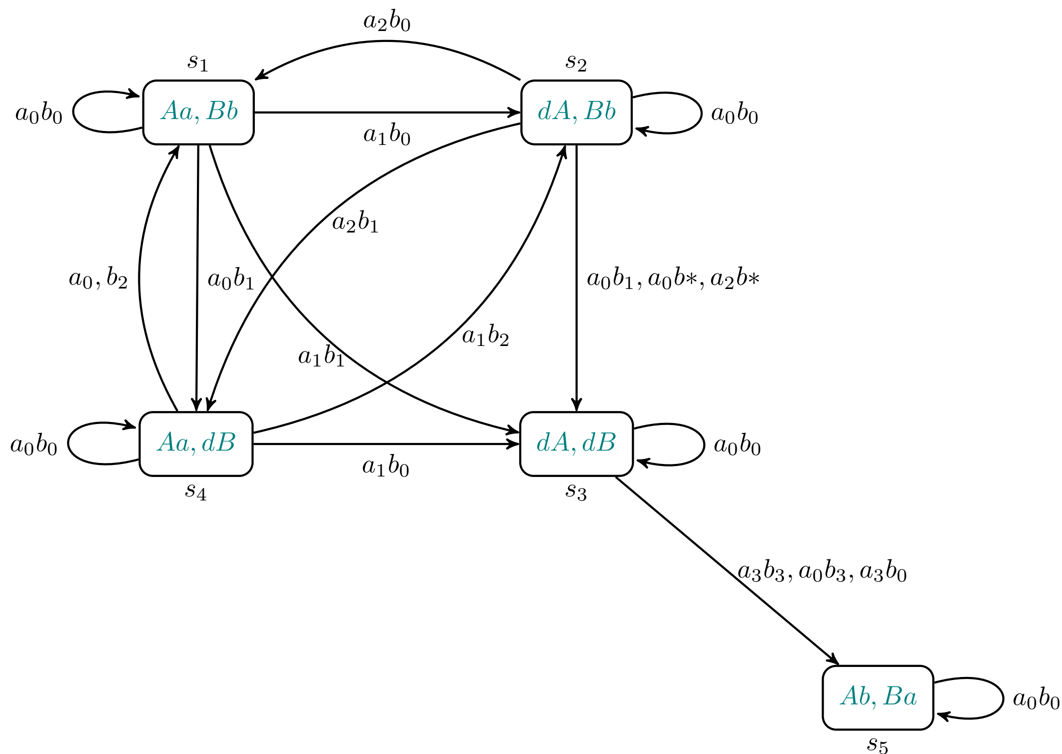


Actions:

i0:	do nothing
i1:	deposit i's asset
i2:	cancel i's deposit
i3:	finalise swap

Figure 5: CGM M^{+U} with Marked Forcing Actions

As seen in Figure 5 both Alice and Bob have forcing actions in state s_3 and s_5, Alice has forcing actions in state s_4 and Bob has forcing actions in state s_2. The forcing actions which are preserved by the negative upgrade are the forcing actions of Bob in state s_2, s_3 and s_5, while for Alice, the forcing actions which are preserved are in states s_3 and s_5. Because the forcing actions in states s_3 and s_5 are in transitions where both agents has forcing actions, if not both agents would have their forcing actions preserved the forcing action of the other would not have been preserved either. The only forcing action which will actually be removed from the negative update is $a*$ in state s_4, because it is not stated in the negative upgrade. Underneath in Figure 6 is the CGM for $M^{+U,-U}$.

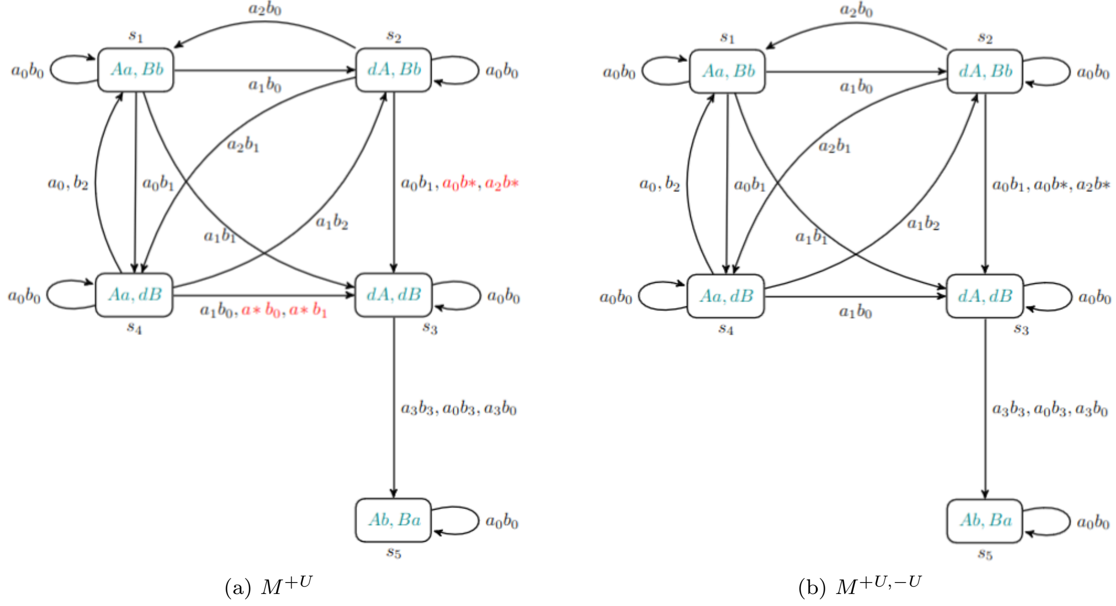


Actions:

i0:	do nothing
i1:	deposit i's asset
i2:	cancel i's deposit
i3:	finalise swap

Figure 6: CGM $M^{+U,-U}$ with Positive and Negative Upgrade

The executability conditions hold in the negative upgrade. There are at least one action per agent in every state and all the combinations of actions the agents chooses has an outcome state, a transition.



(a) M^{+U} (b) $M^{+U,-U}$

Figure 7: CGM M^{+U} with Positive Upgrade and CGM $M^{+U,-U}$ with both Positive and Negative Upgrade

Above in Figure 7 both M^{+U} and $M^{+U,-U}$ is depicted. In formula 14 it is stated in M^{+U} after a negative upgrade $-U$, it is not true that Alice can force a state where both have deposited their asset from state s_4. While in formula 15 it is stated that Bob can still, after a negative upgrade $-U$ to the model M^{+U}, force a state where both have deposited their asset from state s_2.

$$(M^{+U}, s_4) \not\models [-U]\langle\langle a\rangle\rangle(dA \wedge dB) \tag{14}$$

$$(M^{+U}, s_2) \models [-U]\langle\langle b\rangle\rangle(dA \wedge dB) \tag{15}$$

In formula 16 it is stated that Alice is able to force a state where they have swapped assets, from state s_3, and in formula 17 the same is stated for Bob, both is true after the negative upgrade.

$$(M^{+U}, s_3) \models [-U]\langle\langle a\rangle\rangle(Ab \wedge Ba) \tag{16}$$

$$(M^{+U}, s_3) \models [-U]\langle\langle b\rangle\rangle(Ab \wedge Ba) \tag{17}$$

In the event that Bob has deposited and Alice still holds her asset, if Alice, after both the positive and the negative upgrade, does deposit her asset while Bob cancels his deposit the system will end up in state s_2 where Alice has deposited her asset and Bob holds his. Now, Bob has dictatorial powers, if Alice wants to cancel her deposit while Bob wants to deposit, the system will move to state s_4 where both Alice and Bob has deposited their asset. Although if Bob, while the system is in state s_2 has decided not to continue with the swap and does nothing Alice can still cancel her

deposit, as long as Bob chooses to do nothing, and will get her asset back.

The model checking-tools already made do not check for changes in a Smart Contract as they are immutable once deployed on the block chain. While it is immutable one can deploy new instances of the same Smart Contracts at the chain when it needs to be changed. If a company restricts or expands authority to different agents the Smart Contract needs to be changed, it is reasonable to believe that this will happen for companies, considering companies evolve with time.

3.4 Algorithms for Extended Coalition Logic

In this section pseudocode algorithms for upgrading models will be presented and explained. Because the scope of this thesis is to extend an already existing model checker it is assumed that the verification is handled by the original version and what is added is the algorithms for upgrading the models.

Algorithm 1 Determining New Transitions

```
INPUT from_states, to_states, granted_agent
GLOBAL count
CALL agentID_dictionary
SET action_pairs as empty list
IF agentID of granted_agent is 0 THEN
    FOR from_state in from_states
        FOR to_state in to_states
            FOR action in actions_for_other_agent_in_state
                BUMP action_pairs WITH
                [from_state, to_state, ["dictatorial action"+count, action]]
            END FOR
        INCREMENT count
        END FOR
    END FOR
ELSE IF agentID of granted_agent is 1 THEN
    FOR from_state in from_states
        FOR to_state in to_states
            FOR action in actions_for_other_agent_in_state
                BUMP action_pairs WITH
                [from_state, to_state, [action, "dictatorial action"+count]]
            END FOR
        INCREMENT count
        END FOR
    END FOR
END IF
OUTPUT action_pairs
```

Algorithm 1 determines new transitions in positive upgrades. The method determines new transitions for one update at the time. It is given the state IDs of both the from-states and the to-states

in an update inside the upgrade, and the agent who are granted powers. There is a global count used to differentiate the dictatorial actions throughout the verification problem. A method for pairing the name of the agents with their agent IDs is called, and the list that will be returned by the algorithm is set to an empty list.

In this algorithm it is assumed that there are only two agents in the model. The list returned is containing lists with all information required for new transitions. In each element in the main list, there is a list containing three elements, the first is the state ID of the from-state, the second is the state ID of the to-state, while the third element is a list containing the action-pair to the particular transition. Depending on which agent is granted powers, the action-pairs contains different setup, if it is the agent with agent ID 0, the dictatorial powers is the first element and the other agents action is the second, while if the agent ID is 1, the order is opposite. Iterating through all from-states, all to-states and all the actions in the particular from-state for the agent not granted powers will generate all transitions needed for granting dictatorial powers.

Algorithm 2 describes the method for testing executability. The method raises an error if the executability condition are not met, otherwise the method does nothing. The method has a dictionary as input, including all agents with granted powers as keys and the from-state IDs in where the dictatorial actions are granted. First the method checks if there are more than one agent granted dictatorial powers, if not there cannot be any clashes and the executablitiy condition holds. If there are more than one agent granted dictatorial powers in the upgrade the method all from-states, for each from-state there is a counter incremented by one, and the state ID is added to a set. In the end the length of the set is compared to the counter, if the two values are not equal it indicates there is at least one state where more than one agent is granted dictatorial powers from and the upgrade clashes, if this is the case the method raises an error.

Algorithm 2 Executability Condition Positive Upgrade

```
INPUT agent_from_states_dictionary
SET values AS empty set
SET value_count AS 0
IF more than one agent is granted powers THEN
    FOR agents from_states
        BUMP value_count WITH length of elements in from_states
        FOR state in from_states
            BUMP values with state
        END FOR
    END FOR
END IF
IF length of values is not equal to value_count THEN
    RAISE ERROR
```

Determining the removal of transitions in negative upgrades is not as straight-forward determining as new transitions in positive upgrades. In negative upgrades all transitions with forcing actions has to be identified before the system can determine which of them will be preserved. In algorithm

3 the method for determining transitions with forcing actions is described.

Algorithm 3 Determining Forcing Actions

```
INPUT dictionary_action
SET forcing_actions_agent1 AS empty list
SET forcing_actions_agent2 AS empty list
SET agent_count AS 0
WHILE agent_count < number of agents
    FOR state in all states
        SET non_forcing_actions AS empty set
        SET action_list AS empty list
        FOR curr_state, action_pairs IN dictionary_action
            IF state.ID == curr_state THEN
                SET action_set AS empty set
                FOR action in action_pairs
                    ADD agents_action TO action_set
                END FOR
                ADD action_set TO action_list
            END IF
        END FOR
        IF more than one set of actions in action_list THEN
            ADD intersection of sets in action_list TO non_forcing_actions
        END IF
        FOR transition_states, action_pairs IN dictionary_action
            IF state.ID == from_state THEN
                FOR action in action_pairs
                    IF agent_action not in non_forcing_actions THEN
                        IF agent_count == 0 THEN
                            ADD transition_states, action_pair as list
                            TO forcing_actions_agent1
                        ELSE IF agent_count == 1 THEN
                            ADD transition_states, action_pair as list
                            TO forcing_actions_agent2
                        END IF
                    END IF
                END FOR
            END IF
        END FOR
    END FOR
END WHILE
OUTPUT forcing actions for agent1 and forcing actions for agent2
```

The method assumes two agents in the model. The method returns two lists, one with the transitions with forcing actions for agent 1 and one for agent 2. As input to the method there is a

dictionary with tuples of predecessor's and successor's state ID and the action pairs for the transition is the values. For each agent all states in the model are iterated. The method checks if one action for an agent, in one specific predecessor, has more then one successor or not, if it does not it is a forcing actions, while if it does it is not a forcing action.

Algorithm 4 Executability Condition Negative Upgrade

```
INPUT remaining_transitions
INPUT state_action_dictionary, action_pair_counter
SET from_states AS empty list
FOR transition IN remaining transitions
    BUMP from_states WITH from_state in transition
END FOR
FOR state IN all_states
    IF state.ID NOT IN from_states
        RAISE ERROR
    END IF
END FOR

SET remaining_transitions_dictionary AS empty dictionary
SET state_count AS 0
SET test_count AS 0
WHILE state_count < number_of_states
    FOR transition in remaining_transitions
        IF transitions from_state == state_count
            ADD key: state_count, value: action_pair
            TO remaining_transitions_dictionary
        END IF
    END FOR
    INCREMENT count
END WHILE
FOR  key, value IN state_action_dictionary
    FOR k, v IN remaining_transitions_dictionary
        IF key == k:
            BUMP test_count BY number of elements in
            intersection between set of value and set of v
        END IF
    END FOR
END FOR
IF action_pair_counter > test_count
    RAISE ERROR
END IF
```

The method for testing the negative upgrade's executability conditions are described in the method in algorithm 4. The method requires extensive input, including what the remaining transitions are,

action profiles for each state and a number telling how many action pairs is in the new version of the model. First the condition of one transition out from each state is tested, by iterating all states, testing if all state IDs are in from-states retrieved from the input of remaining transitions. If there is a state ID that is not included in the from-state an error is raised, otherwise the next condition is tested. The method then checkes if all combinations of actions in the new version of the model has a transition, if it does not the second condition does not hold and an error is raised, if there is one transition for all combination of actions in all states the condition is met and the method does not do anything.

The next section will present the model checking tool StraTegic Verifier, which will later be extended with the algorithms presented.

4 StraTegic Verifier - A Model Checking Tool

StraTegic Verifier (STV) is a model checking tool developed to verify strategic abilities in multi-agent systems with imperfect information [Kurpiewski et al., 2019a]. The tool is experimental and includes recently developed algorithms intended to deal with the complexity of imperfect information. STV contain both a Command Line Interface (CLI) and a User Interface (UI) for visualisation of some chosen models and scenarios. The tool is intended for logics such as Alternating-Time Temporal Logic (ATL), containing temporal modalities as future (F) and global (G). While CL only consider one step forward, ATL consider sequences going on forever, G all sequences and F one sequence. The standard CL, without upgrades, is a fragment of ATL [Ågotnes et al., 2015].

The current section will include an introduction of the models, logics and algorithms implemented in STV and a brief walk-through of input and output in the CLI of the tool.

4.1 Imperfect Information Concurrent Game Structure

The models used in STV is Concurrent Game Models with imperfect information [Kurpiewski et al., 2019a]. A Concurrent Game Model with imperfect information includes all of the structure of a Concurrent Game Model presented in the theory section. In addition it contains a guard indicating which of the abilities an agent in a state can choose from given what she knows, and a family of equivalence relations, for each agent, indicating which states are indistinguishable (the agent cannot see any difference) from where the agent currently is [Schobbens, 2004].

There are four variations of Concurrent Game Models, containing all combinations of imperfect or perfect information and imperfect or perfect recall [Schobbens, 2004]. Perfect information and perfect recall is the variation that is normally used for ATL, the memory is not taken into consideration, while in the three other combinations the memory is taken into consideration in some extent. STV includes two classes of Concurrent Game Models, one with perfect information and imperfect recall and one with imperfect information and imperfect recall. The classes of models are implemented with asynchronous semantics, meaning that the agents does not necessarily act at the same time.

4.2 Model Classes in STV

There are implemented different classes for models, Local Model, Global Model, Simple Model, ATLIr Model and ATLIR Model. The Local Models are the action profiles per agent and Global Model is a collection of all instances of the Local Model class. When the instance of Global Model is generated, it is generated an instance of the class Simple Model aswell with all needed instance variables to be able to verify the formula in the model. In the class Simple Model there are methods for generating instances of ATLIr Model and ATLIR Model. In the end it is in the classes ATLIr Model and ATLIR Model there are methods for verification.

ATLIr Model is short for ATL model with perfect information and imperfect recall, while ATLIR Model is short for ATL model with imperfect information and imperfect recall. Perfect information indicates that the agents know all their actions and all the actions of the other agents, and what

will be the result after the combination of their actions, while imperfect information indicates that the agents does not know it all. Perfect recall indicates that an agent can remember the history of where he has been up until the current state, while imperfect recall indicates that he does not have any history prior to the current state. [Schobbens, 2004]

4.3 Logics Implemented

STV can parse ATL formulas with the Global-operator and the Future-operator and the combination of the two. The tool cannot parse or verify formulas with the Next-operator or the Until-operator. STV can also parse CTL formulas and Strategy Logic (SL) formulas. The tool includes verification of ATL and of SL. It does not seem to have a verification method for CTL. [Kurpiewski, 2022]

The parsing of formulas is somewhat recursive. When parsing, the parser expects a coalition or a path quantifier, it varies in the different implemented logics, then a temporal operator followed by an expression at the form ($p = True$). The expression can be nested, the left part can be it's own expression or the expression can be a negation of an expression and so on, and is recursive. The expression cannot contain a coalition or temporal operator and the formula cannot be a single expression without the coalition or temporal operator.

4.4 Algorithms

STV includes algorithms for verification of ATL-models with perfect information and imperfect recall, and with imperfect information and imperfect recall. The tool executes explicit-state model checking. There are two verification algorithms implemented for Alternating-time Temporal logic, Approximate fixpoint verification and DominoDFS. In addition there is implemented one algorithm for Strategy Logic with simple goals. [Kurpiewski et al., 2019a]

Approximate fixpoint verification is an algorithm with two fixpoints, one lower approximation and one upper approximation. In the upper bound only perfect information is verified while in the lower approximation also imperfect information is taken into account. If the upper approximation is false the whole algorithm will return false, while if the lower approximation is true then the whole algorithm will return true. [Jamroga et al., 2016]

DominoDFS is a dominance-based strategy depth-first search that is inspired by game theory. The method finds the best winning strategy through checking all one-step strategies in the current states and removing the strategies dominated by others. The main goal is to identify which partial strategy controls the critical parts of the system's execution. [Kurpiewski et al., 2019b]

4.5 CLI: Input and Output

In Figure 8, there are two options in the CLI when one first calls the main-file, either generate specification or verify formula in model. If wanting to generate specification there are three options which are all sai-models. After one of the alternatives is chosen one has to fill out how many AI agents and maximum model quality, the output will be the path to where the specifications are

saved.

```
● eirin@eirin:~/stv-master$ python3 main.py
Usage: main.py [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  generate-spec
  verify
● eirin@eirin:~/stv-master$ python3 main.py generate-spec
Usage: main.py generate-spec [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  sai
  sai-imp
  sai-mim
● eirin@eirin:~/stv-master$ python3 main.py generate-spec sai
Number of AI agents [2]: 2
Maximum model quality [2]: 1
Model spec saved to stv/models/asynchronous/specs/generated/sai_2ai_1mmq.txt
```

Figure 8: Generating Specifications STV

If the user chooses to verify instead, as shown in Figure 9, there are two new options, synchronous and asynchronous semantics in the model. If the user chooses asynchronous semantics she will be asked to input a name of a generated specifications file and the output will be the upper approximation and the lower approximation result, as-well as time used to generate and verify, and the number of states and transitions. The Approximate fixpoint verification method is used. One can also input other specifications if one wants to input a model that is not included in the generated files.

```
● eirin@eirin:~/stv-master$ python3 main.py verify
Usage: main.py verify [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  asynchronous
  synchronous
● eirin@eirin:~/stv-master$ python3 main.py verify asynchronous
Model file name [simple_voting_1v_2c]: train_controller_3t_1c
Generation time: 0.032995362, #states: 162, #transitions: 648
Upper approximation
Time: 0.0027085379999999826, result: True
Lower approximation
Time: 0.004111326000000026, result: True
```

Figure 9: Verification with Asynchronous Semantics in STV

As earlier mentioned the tool has a simple input language, in Figure 10 one can see an example of a input file from which STV generates local and global models and determines whether or

not the formula stated at the end holds or not in the given model. Do to the fact that the implementation is preliminary, the simple input language is not flexible [Kurpiewski et al., 2019a].

```
Agent Train[3]:
init: wait
shared in_aID: wait -> tunnel [aID_in=True]
shared out_aID: tunnel -> away [aID_out=True]
return: away -> wait [aID_return=True]

Agent Controller[1]:
init: green
shared in_Train1: green -> red
shared back_Train1: red -> green
shared in_Train2: green -> red
shared back_Train2: red -> green
shared in_Train3: green -> red
shared back_Train3: red -> green

% REDUCTION: [in_Train1, in_Train2, in_Train3]
% COALITION: [Controller1]
% LOGIC: CTL
% FORMULA: AF(Train1_in=True | Train2_in=True | Train3_in=True)
REDUCTION: [Train1_return]
COALITION: [Train1]
LOGIC: ATL
FORMULA: <<Train1>>F(Train1_return=True)
```

Figure 10: Example of Input File

When modeling a model in the simple input language of STV, first one have to state if the semantics are synchronous or asynchronous, then one have to state the agents name and their initial state. Following is the particular agent's actions, one states the action-name first and then the predecessor and the successor. If a proposition has a different value in the successor, than in the predecessor, it is written in brackets behind the successor. If there are conditions for the action it goes in brackets inside the arrow between the predecessor and the successor. When finished with the first agents actions one can start with the next, if all agents have the same actions one can also write them in the same section and put a bracket with the number of agents after the agent-name.

Below the agents and their actions, in the input-file, there are a few more lines specifying if there should be any reduction, what logic is used in the formula and what the coalition is, additional keywords are `PRESISTENT` indicating which propositions are global and `INITIAL` stating which values the propositions initially contain. Finally, the formula is stated and the input-file is finished.

When choosing the synchronous semantics one gets three more choices, all are implemented as classes and are not generated specification-files, as shown in Figure 11. If one chooses castles as below one has to fill in number of workers for the three castles and how many lives. The output

will be whether or not it is perfect information, time spent generating model, time spent verifying the formula and the approximation result, a boolean value. The algorithm used when synchronous semantics is Approximate fixpoint verification.

```
● eirin@eirin:~/stv-master$ python3 main.py verify synchronous
  Usage: main.py verify synchronous [OPTIONS] COMMAND [ARGS]...

  Options:
    --help  Show this message and exit.

  Commands:
    bridge
    castles
    tianji
● eirin@eirin:~/stv-master$ python3 main.py verify synchronous castles
  Number of workers for the first castle: 2
  Number of workers for the second castle: 1
  Number of workers for the third castle: 2
  Castle life [3]: 2
  Castles (2,1,2), life=2
  Generating model
  Model generated in 5.766383418998885 seconds
  Starting verification
  Perfect information
  Formula verified in 1.3017017660004058 seconds
  Approximation result: False
```

Figure 11: Verification with Synchronous Semantics in STV

The developers of STV has tested their tool and compared it to a state of the art tool MCMAS and one experimental tool SMK, with promising results [Kurpiewski et al., 2019a].

This thesis will build upon STV utilising the ATL-model class for perfect information and imperfect recall, asynchronous semantics so it is possible to input other specifications for new models and the upper bound of the Approximate fixpoint verification algorithm.

5 Implementation Details

STV is built for a different purpose than the scope of this thesis. The scope does not include imperfect information. Thus the ATL-model class with perfect information and imperfect recall, called ATLIr, is the best provided class to use. Because ATLIr includes coalitions and the ATL's next-operator in combination with a coalition expresses the same as the operator in Coalition Logic, it is adequate to use the class ATLIr. Perfect information indicates that the agents have all information about abilities, their own and others, and imperfect recall indicates that the agents can only recall the current state and not the path they have had until the current state.

The intention of this thesis is to be able to verify newly made models that do not have already generated specifications or classes made. The entry-point of verification through the asynchronous semantics is the entry-point utilised. Although the scope of this thesis is synchronous semantics, it will be explained how to deal with this mismatch. The upper bound of the Approximate fixpoint verification algorithm will be the only bound used due to the fact that it takes into consideration perfect information. In contrast, the lower bound considers imperfect information, which is outside the scope of this thesis.

The extension of STV[1] is located in certain parts of the program. It is mainly extended in the Formula Parser and the classes Global Model, Simple Model and ATLIr Model. The initial generating of models is almost the same as in the original version, excluding the transitions containing the asynchronous semantics, which is how the mismatch is dealt with. The main-file is altered to be able to use the CLI for the extended logic. In Simple Model, the changes made are the actual removal or adding of the transitions. In the ATLIr class, one new method is implemented to verify the next-operator with a coalition with synchronous semantics, which is used for verification when verifying the operator in Coalition Logic.

The remainder of this section is structured as follows: First, the implementation of the Coalition Logic operator will be gone through in detail, and second, changes in the formula parser will be elaborated. Third, the implemented formula data structure and an example with abstract syntax trees will be provided and explained. At last, the new methods in the class Global Model regarding the determination of the alteration of the model will be explained through the algorithms in source code for both positive and negative upgrades, and the executability conditions will be explained.

5.1 Formula Parser

The model checker needs a model and a formula as input. The formula should be able to be any formula included in the language of DDCL. In the original version of STV, the formulas are rigid. There has to be first a coalition, second an ATL-operator, and the operators can be "F", "G", "FG", or "GF". The end of the formula can be any simple expression containing propositions, boolean values and conjunction, disjunction and negation. The simple expression can be nested, although it cannot include coalitions or ATL-operators. To be able to parse any formula included in the language of DDCL, the coalition has to be able to be inside the simple expressions as well. Because the scope of the thesis is CL and not ATL, there is no need for parsing of ATL-operators,

[1] The source code of the extension is available at: https://github.com/eirinmla/STV-master-extended

although it can easily be implemented for further research.

The parser has gone through a considerable change to include recursive descent and being able to parse upgrades. To illustrate what the formula parser can and cannot parse, syntax diagrams for all parts of the formula will be provided and explained.

In the original version of the STV there were already implemented a few methods to parse parts of the new formula. There are also some built-in functions that does not need to be parsed in explicit methods since the methods are implemented in the programming language. The built-in functions parses BooleanLiterals, IntegerLiterals and StringLiterals such as shown below:

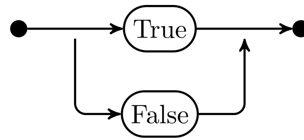


Figure 12: BooleanLiteral

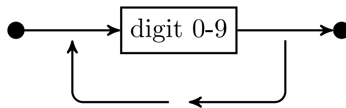


Figure 13: IntegerLiteral

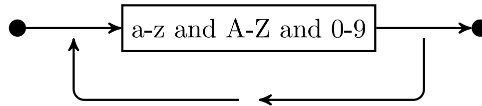


Figure 14: StringLiteral

As illustrated the BooleanLiterals can either be true or false, the IntegerLiterals can be minimum one integer 0-9 although it can be more, and the StringLiteral can be a collection of capital and non-capital letters in the English alphabet joined together with integers.

Further on, the already implemented features of the formula parser were Coalition, SimpleExpressionOperator and SimpleExpression. The first two mentioned have not been altered. A Coalition is comma-separated if more than one agent and one agent is a StringLiteral. The SimpleExpressionOperator can respectively be; and, or, not, equal to, not equal to, greater than.

SimpleExpression is mainly divided in three parts, two of them are new SimpleExpressions and the middle element is a SimpleExpressionOperator. There is one specific case where it can be only a SimpleExpressionOperator and one SimpleExpression and the case occurs if the SimpleExpressionOperator is a ! (not). Each SimpleExpression in the bigger SimpleExpression will then be parsed and can once more be two SimpleExpressions and one SimpleExpressionOperator or one

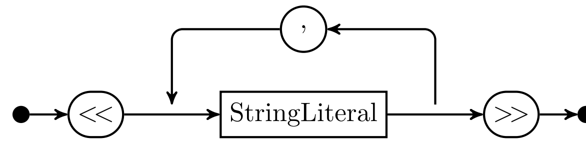


Figure 15: Coalition

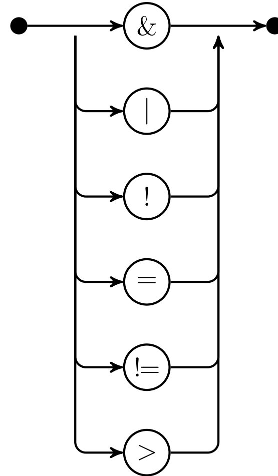


Figure 16: SimpleExpressionOperator

SimpleExpression and a SimpleExpressionOperator or it can be a BooleanLiteral, IntegerLiteral or a Proposition which is a StringLiteral. In the extension the Simple Expression can also be an UpgradeFormula, either as a full UpgradeFormula or as an UpgradeFormula without the Upgrades, called a CoalitionExpression.

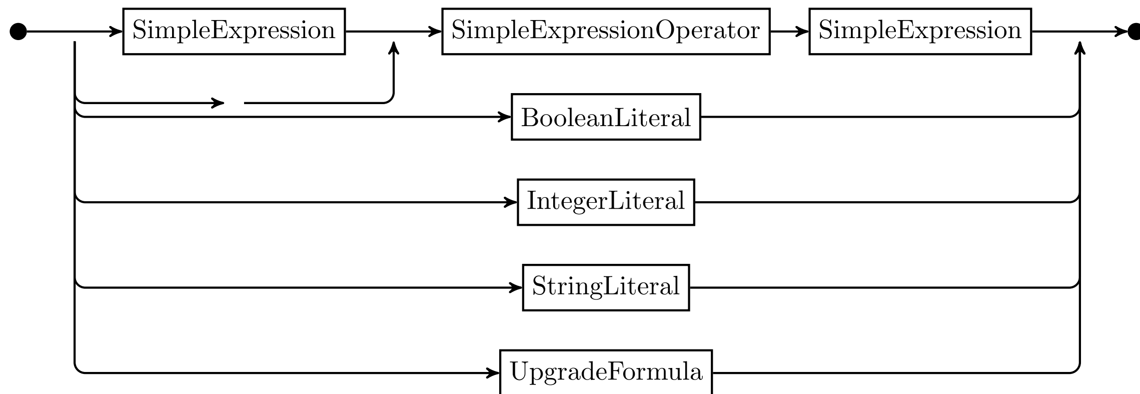


Figure 17: SimpleExpression

In the extension of STV, implementation of the ability to parse upgrades was required. The full formula is called an UpgradeFormula, and in the UpgradeFormula is where the parsing starts. An UpgradeFormula can contain two elements, there has to be a CoalitionExpression and prior to the CoalitionExpression there can be an UpgradeList. The parser will look at the first sign of the whole UpgradeFormula and will determine if there is in fact an UpgradeList or not, if there is, the next section parsed will be the UpgradeList, else the method will start parsing the CoalitionExpression.

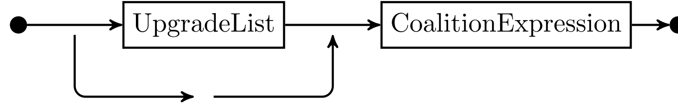


Figure 18: UpgradeFormula

The CoalitionExpression can be divided into two parts, if both are present. There has to be a SimpleExpression, prior to the SimpleExpression there can be a Coalition. The parser will look at the first sign of the CoalitionExpression and determine whether or not there is a Coalition. If there is a Coalition present it will be parsed as earlier explained, if there is not a Coalition present the CoalitionExpression method will go straight to parsing the SimpleExpression as earlier explained.

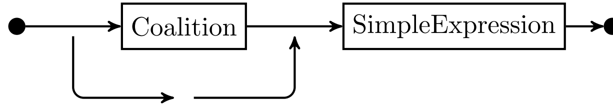


Figure 19: CoalitionExpression

If there was an UpgradeList prior to the CoalitionExpression in the UpgradeFormula, the UpgradeList would be parsed first. When parsing an UpgradeList the parser finds "{" and "," (or "}") and the part in-between will be an upgrade, there can be one or multiple Upgrades in an UpgradeList. If there are multiple Upgrades they are divided by a comma. In each Upgrade in the UpgradeList another method will be called to parse the Upgrades one by one.

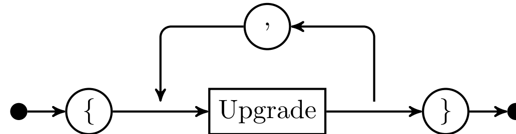


Figure 20: UpgradeList

Parsing an Upgrade has the same structure as when parsing an UpgradeList, although the elements that is parsed is not longer Upgrades but Updates. In one Upgrade there can be one or more Updates. The method identifies the Updates and send them one by one to the next method for parsing.

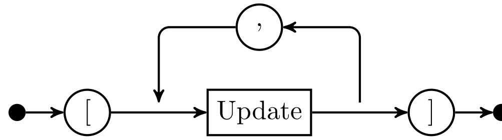


Figure 21: Upgrade

Every Update is, when parsed, divided into four different elements. The elements are a from-state, the agent who is granted or revoked its powers, a to-state and the UpgradeType, negative or positive. Both the from-state and to-state are elements that once again will be parsed from the top level, in the same way as the whole formula was, and will be treated as an UpgradeFormula.

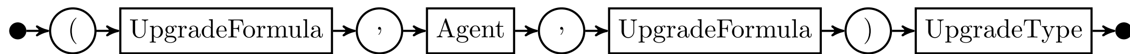


Figure 22: Update

The agent is a StringLiteral while the UpgradeType is either positive or negative, denoted with a + or a −.

Figure 23: Agent

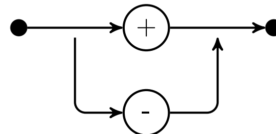


Figure 24: UpgradeType

The recursion is implemented in that the parser starts over again in the lowest level of the formula parser, the method of parsing the Update. When the formula is being parsed the parser calls the provided classes and creates instances of the classes. In the next section the data structure of the classes will be explained.

The implementation of the parser does not include anything specific for propositions. The propositions are merely treated as strings of text, and the truth value has to be stated as a boolean value at the other side of an "equal"-sign to be handled as a proposition.

5.2 Extended Data Structure

In the extension of STV there has been implemented a comprehensive data structure to store all the elements parsed. To visualise the data structure there will be provided class diagrams and an example formula with complementary Abstract Syntax Trees.

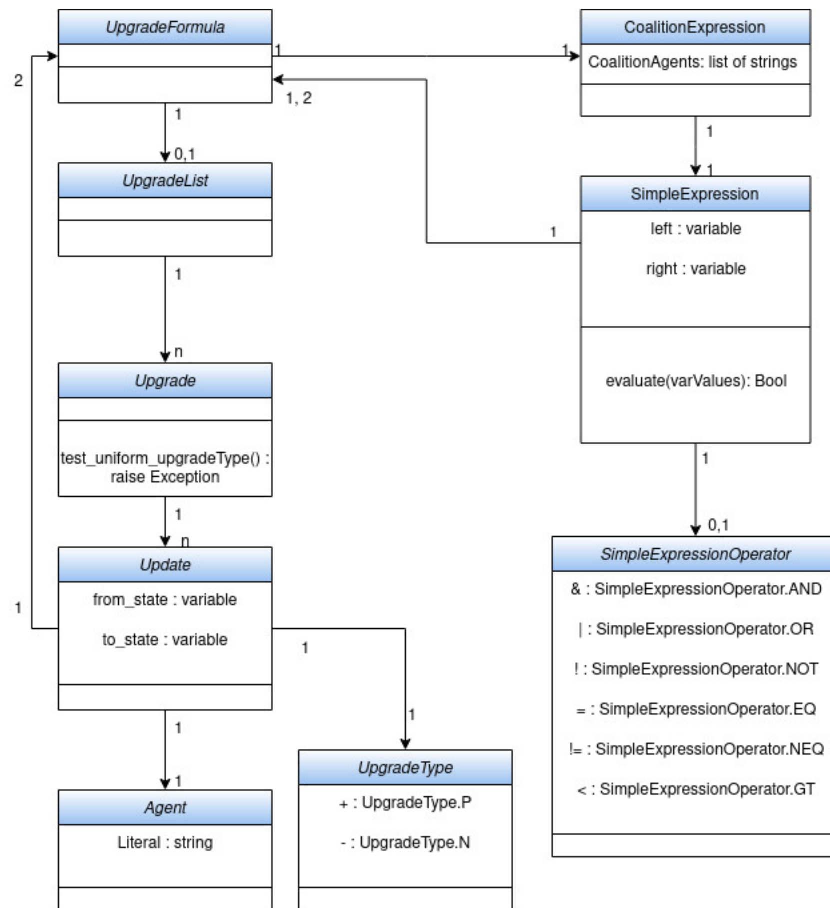


Figure 25: Class Diagram of Classes for Formula

To process the data of a DCL formula, there are nine classes implemented. The main class UpgradeFormula, consists of two instance variables, upgradeList and coalitionExpression. Both instance variables are objects of other classes, respectively UpgradeList and CoalitionExpression. The class UpgradeList contains one instance variable, upgrades. Upgrades is a list of objects of the class Upgrade.

Upgrade's instance variable updates is similar to the list of upgrades in UpgradeList, although it is a list of objects of the class Update. Upgrade has a method to check if there is consistency of the upgradeType, either all updates in an upgrade has to be positive or they have to be negative, there cannot be an upgrade containing both upgrade types.

The class Update contains four instance variables, namely, from_state, agent, to_state and upgradeType. Both from_state and to_state is instances of UpgradeFormula. Agent is an instance of the class Agent, upgradeType is an instance of the class UpgradeType.

The second instance variable of UpgradeFormula, coalitionExpression, is an instance of the class CoalitionExpression. CoalitionExpression contains two instance variables, coalitionAgents and simpleExpression. CoalitionsAgents is a list of agents and each agent is a string-value. SimpleExpression is an instance of the class SimpleExpression.

The class SimpleExpression contains three instance variables and one method. The instance variables are left, operator and right. The instance variables left and right are instances of the class UpgradeFormula, while the instance variable operator is an instance of the class SimpleExpressionOperator. The method evaluate(varValues) returns a boolean value after evaluating instance variable left and instance variable right with instance variable operator.

The classes UpgradeType and SimpleExpressionOperator both inherit from another class called EnumMeta. EnumMeta is a comprehensive class, and includes methods that fixes the instance variable of UpgradeType and SimpleExpressionOperator as a constant value.

5.2.1 Example with Abstract Syntax Trees

The example is the formula $\{[((p = False), a, (p = True))+]\}\langle\langle a\rangle\rangle(p = False)$. It contains one upgrade with one single positive update, from states where p is false, a will have the ability to force a state where p is true. After the upgrade, for the model checker to output true, agent a has to be able to force a state where p is false from the initial state in a model that is not provided, the example focuses solitary on the data structure of the formula.

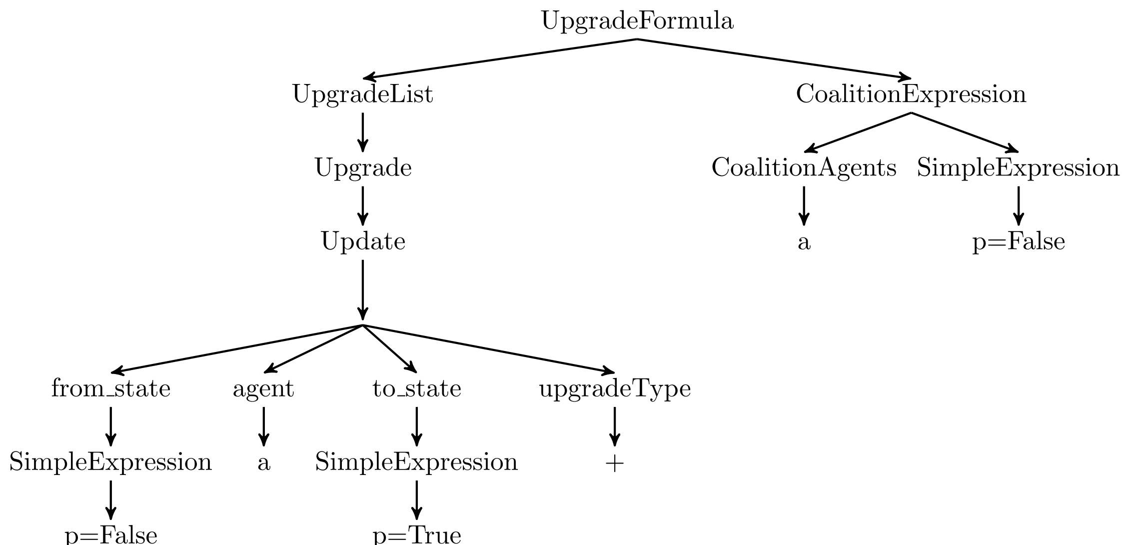


Figure 26: Abstract Syntax Tree of $\{[((p = False), a, (p = True))+]\}\langle\langle a\rangle\rangle(p = False)$

The formula in its totality will be an instance of the class UpgradeFormula, divided into two instance variables, upgradeList and coalitionExpression. The instance variable of upgradeList will be $\{[((p = False), a, (p = True))+]\}$ and the instance variable of coalitionExpression will be $\langle\langle a\rangle\rangle(p = False)$.

Further, the data in the instance variable upgradeList will be divided into upgrades and will be added to the list of upgrades of the class UpgradeList. In this particular case there is one single upgrade, thus the instance variable upgrades will only have one element, $[((p = False), a, (p = True))+]$. Each element in the instance variable upgrades will be a individual object in the Upgrade-class.

The class Upgrade contains a list of updates as instance variable, each element in the list is its own instance of the class Update. In this example there is one single update and only one instance of the class Update. In Update there are, as mentioned, four instance variables. The agent is a. Both instance variables from_state and to_state can be their own instance of UpgradeFormula, CoalitionExpression or SimpleExpression, in this example they are both instances of SimpleExpres-

sion. The upgradeType is UpgradeType.P.

The instance variable coalitionExpression of UpgradeFormula is an instance of the class CoalitionExpression. The two instance variables coalitionAgent and simpleExpression both is indicated to have a concrete value in this example. coalitionAgent is a and simpleExpression where it is divided into two properties, coalitionAgents and a SimpleExpression. In this example, coalitionAgent is a list with one element, "a", and the simpleExpression is $p = False$. The instance variable simpleExpression is moved to its own abstract syntax tree underneath.

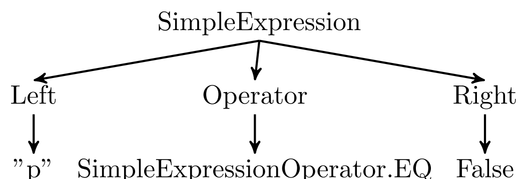


Figure 27: Abstract Syntax Tree 2: Simple Expression

As seen in the class diagrams the SimpleExpression class has three instance variables. The first, left, is in this example a proposition and the value is p, the operator is SimpleExpressionOperator.EQ and the instance variable right is a Boolean Literal False. All SimpleExpression-instances in Figure 26 is at the form of Figure 27, although the Boolean Literal varies from the different objects.

5.3 Implemented Methods in Global Model and Simple Model

In the initial version of STV the formula would be verified and that was the whole intention of the formula stated in the input-file. In the extended version of STV, to be able to verify the formula in the given model, as long as there exists an UpgradeList in the UpgradeFormula there has to be generated at least one new version of the given model to be able to verify the formula.

The majority of implemented methods for generating the new versions of the model is located in the Global Model. There are different methods for positive and negative upgrades, in addition the executability-conditions are different. Next, the key methods for upgrading models, both positive and negative, will be provided and explained.

5.3.1 Methods for Positive Upgrades

To be able to grant actions through a positive upgrade, determining which new transitions should be added and where is required. In addition it is necessary to check the executability-condition for clashfreeness.

In the Global Model class there are already some initial instance variables set from the generation of the inputted model. The states are given, propositions are given and the initial transitions are generated. When granting dictatorial powers, what is needed for verifying a new version of

the model is adding transitions that reflects the new dictatorial powers. A dictatorial power is, as mentioned, an action that an agent can use to get to a specific state no matter what others does.

To be able to add a transition in STV, the predecessor and successor is needed and which actions the agents choose, to transit with that specific transition is required. From the data structure of the formula the program is able to retrieve information of which agent is granted powers and what conditions has to be true in the to_states and from_states. If to_state or from_state is a SimpleExpression a already implemented method will be called to find all states there the SimpleExpression is true, and will return a list of the IDs of the states, a integer value. If the condition in the to_state or the from_state is more complex, the condition will be verified as its own formula and the resulting states ID of where the condition holds will be returned as a set of integer values.

The method implemented for determining the new transitions is depicted below in Figure 28. As input the method has the list if from_states and the list of to_states, both are lists of the state identification value, the last input is the name of the agent granted powers. Further on there is a global count and a dictionary of agents ID and agents names provided. The method returns a list of transitions.

```
1133    def get_positive_transitions(self, from_states, to_states, agent):
1134        """returns new transitions for all actions counterpart has in the
1135        state where the agent is granted dictatorial powers"""
1136        global count
1137        agents_id_dict = self.agents_to_dict()
1138        transitions = []
1139        if agents_id_dict.get(str(agent)) == 0:
1140            for element in from_states:
1141                for elem in to_states:
1142                    for el in self._model.get_possible_strategies_for_coalition(element, [1]):
1143                        transitions.append([element, elem, [f"dict_powers{count}", el[0]]])
1144                    count += 1
1145        elif agents_id_dict.get(str(agent)) == 1:
1146            for element in from_states:
1147                for elem in to_states:
1148                    for el in self._model.get_possible_strategies_for_coalition(element, [0]):
1149                        transitions.append([element, elem, [el[0],f"dict_powers{count}"]])
1150                    count += 1
1151        else:
1152            raise Exception("Only works when two agents")
1153        return transitions
```

Figure 28: Determining New Transitions

The method iterates all from_states, to_states and actions the other agent has and generate a list of one from_state, one to_state and the list of one action-pair which is added to the list of transitions. This approach generates all possible combinations of the given from_states, to_states and all actions the other agent has in the particular from_state such that the result of adding all the new transitions gives the granted agent new dictatorial powers. The global counter is included so the new actions have different names for the CGM to be a valid model.

The executability-condition in the positive upgrades can be checked after what new transitions has to be added to the model. To recap, for a positive upgrade to be clashfree there can not be two agents granted dictatorial powers from the same state in one upgrade. The method implemented has a dictionary as input with agents as keys and a list of state IDs as value, being the states where the agent is getting granted dictatorial powers in.

```
1156    def test_clashfreeness(self, granted_powers_dict):
1157        """ Tests positive executability condition, no two agents
1158            can be granted dictatorial powers from the same state.
1159            Raises exception if it does not hold."""
1160        values = set()
1161        value_count = 0
1162        if len(granted_powers_dict.keys()) > 1:
1163            for _, value in granted_powers_dict.items():
1164                value_count += len(value)
1165                for val in value:
1166                    values.add(val)
1167        if len(values) != value_count:
1168            raise Exception("Two or more updates are clashing.")
```

Figure 29: Test of Clashfreeness

The test is depicted in Figure 29 above. There is one variable called values initiated to an empty set, and a value_count initiated to 0. If there are more than one key in the dictionary, meaning there are more than one agent granted dictatorial powers, the number of state IDs is added to the value_count and all state IDs in the values in the dictionary is added to the set, values. Because values is a set, there are no duplicates. If the length of values is not equal to the value_count there has to be an overlap between the agents in which states they are granted dictatorial powers from and the test raises an exception and the verification-session ends in an error-message.

If the executability-condition holds, the new transitions will be added to the model and the Simple-Expression of the formula will be verified. The new transitions are added in the class Simple Model before creating a class instance of the ATLIr Model, the method is depicted in Figure 30 below.

```
320     def updated_model(self, new_transitions) -> ATLIrModel:
321         """
322         Positive updates Simple Model and generate the Alternating-Time Temporal Logic model
323         with perfect information with dictatorial transitions as list will be in the form
324         [[from state, to state, actions(list)],[...]]
325         :return: ATLIr model
326         """
327         for transition in new_transitions:
328             self.add_transition(transition[0], transition[1], transition[2])
329         updated_model = ATLIrModel(self._no_agents)
330         updated_model = self._copy_model(updated_model, self._actions, epistemic=False)
331         ATLIrModel.print_model(updated_model) # only a print
332         return updated_model
```

Figure 30: Updating the Model

5.3.2 Methods for Negative Upgrades

When an agent is revoked it's dictatorial powers, the given from_states and to_states tells in which states the dictatorial powers will be preserved, and the methods have to remove all dictatorial powers which are not explicitly stated in the updates in the upgrade. For the tool to be able to remove the correct transitions, a method to identify all dictatorial powers is required.

A forcing action or dictatorial power is an action where the outcome will be the same no matter what others do. In the method for determining which transitions contains forcing actions in the different states, the return value is two lists, one for each of the two agents. The lists that are returned has lists inside where the index of the list is the same number as the ID of the state in which the forcing actions are in fact forcing actions. The information needed for this method is gathered from a method in the Simple Model Class which retrieves a dictionary of transitions. All transitions in the model is in this dictionary, the key is a tuple of the predecessor and successor while the value is the actions the agents act on to make the transit happen.

```
1287    def get_forcing_actions(self):
1288        """Determines forcing actions.
1289          Return values : list of transitions with forcing actions for each agent. """
1290        dict_actions = self._model.get_full_transitions()
1291        same_props = self.get_equal_states()
1292        forcing_actions_agent1 = []
1293        forcing_actions_agent2 = []
1294        count = 0
1295        while count < self._agents_count:
1296            from_act_to_all = {}
1297            for state in self._states:
1298                actions = self._model.get_possible_strategies_for_coalition(state.id, [count])
1299                for action in actions:
1300                    all_to_states = []
1301                    for transition in self._model._graph[state.id]:
1302                        if transition.actions[count] == action[0]:
1303                            all_to_states.append(transition.next_state)
1304                    if state.id not in from_act_to_all.keys():
1305                        from_act_to_all[state.id] = [{action[0]: all_to_states}]
1306                    else: from_act_to_all[state.id].append({action[0]: all_to_states})
1307            for from_state, value in from_act_to_all.items():
1308                for d in value:
1309                    for to_states in d.values():
1310                        if len(set(to_states)) == 1:
1311                            if count == 0:
1312                                for i in dict_actions[(from_state,to_states[0])]:
1313                                    if i[count] in d.keys():
1314                                        forcing_actions_agent1.append([(from_state, to_states[0]), i])
1315                            elif count == 1:
1316                                for i in dict_actions[(from_state,to_states[0])]:
1317                                    if i[count] in d.keys():
1318                                        forcing_actions_agent2.append([(from_state, to_states[0]), i])
1319                        else:
1320                            for props in same_props:
1321                                if len(props.intersection(set(to_states))) == len(to_states):
1322                                    for to_state in to_states:
1323                                        if count == 0:
1324                                            for i in dict_actions[(from_state,to_state)]:
1325                                                if [(from_state, to_state), i] not in forcing_actions_agent1:
1326                                                    forcing_actions_agent1.append([(from_state, to_state), i])
1327                                        elif count == 1:
1328                                            for i in dict_actions[(from_state,to_state)]:
1329                                                if [(from_state, to_state), i] not in forcing_actions_agent2:
1330                                                    forcing_actions_agent2.append([(from_state, to_state), i])
1331            count += 1
1332        return forcing_actions_agent1, forcing_actions_agent2
```

Figure 31: Determining Forcing Actions

The approach used for determining forcing actions is to look at one agent and one predecessor at

the time and check whether or not the particular action of the agent has more than one successor. If the agents action only have one successor it is a forcing action, because no matter what the other agent chooses to do the successor will be the same. If the agents action has two or more successors the successors have to be checked in the list same_props to check if they contain the same propositions , if they do, the transition will be added, if it does not, the action is not a forcing action and will not be added to the list.

All transitions with forcing actions in the model are now identified, the next step is to check which of the same transitions are in the update in the negative upgrade and will be preserved. In the method depicted below in Figure 32, it is determined which transitions with forcing actions are the same as the transitions stated in the update and will be preserved. The method first initialises an empty list that will be returned with preserved transitions and an empty set which will be returned with the maybe preserved transitions. It retrieves information from the method above, gathering the forcing actions for each agent. Further it collects the state IDs of the states that hold the conditions in the from_state and to_state of the update, at last it retrieves the identification number of the agent who is revoked its powers.

```
994     def updating_model_update_negative(self, update):
995         """ Determine which transitions with forcing actions should be preserved
996             by the update, and which transitions that might be preserved depending
997             on the other updates in the same upgrade
998             return value : list of transitions that will be preserved by the udate
999             and a list of transtions that should be preserved by the update but
1000            might not be because the other agent also has a forcing action in the
1001            same transition """
1002        preserved_transitions = []
1003        maybe_preserved_transitions = {}
1004        forcing_actions_agent1, forcing_actions_agent2 = self.get_forcing_actions()
1005        from_state_ids = self.updating_model_upgrade_formula(update.fromState)
1006        to_state_ids = self.updating_model_upgrade_formula(update.toState)
1007        agent = self.agent_name_to_id(str(update.agent))
1008        if agent == 0:
1009            for element in forcing_actions_agent1:
1010                for state in from_state_ids:
1011                    for s in to_state_ids:
1012                        if (state, s) == element[0] and element not in forcing_actions_agent2:
1013                            preserved_transitions.append(element)
1014                        elif (state, s) == element[0] and element in forcing_actions_agent2:
1015                            if 0 not in maybe_preserved_transitions.keys():
1016                                maybe_preserved_transitions[0] = [element]
1017                            else:
1018                                maybe_preserved_transitions[0].append(element)
1019        elif agent == 1:
1020            for element in forcing_actions_agent2:
1021                for state in from_state_ids:
1022                    for s in to_state_ids:
1023                        if (state, s) == element[0] and element not in forcing_actions_agent1:
1024                            preserved_transitions.append(element)
1025                        elif (state, s) == element[0] and element in forcing_actions_agent1:
1026                            if 1 not in  maybe_preserved_transitions.keys():
1027                                maybe_preserved_transitions[1] = [element]
1028                            else:
1029                                maybe_preserved_transitions[1].append(element)
1030        return preserved_transitions, maybe_preserved_transitions
```

Figure 32: Finding Preserved Transitions in an Update

It is possible that both agents have an forcing action from the same predecessor to the same successor, typically if there is only one outgoing transition from a state. If this is the case and only one of the agents has been stated to preserve their forcing action the transition will be removed.

Depending on which agent is revoked its powers, the method determines whether or not the agents forcing actions should be preserved by comparing the transitions in their forcing action list to the combination of predecessors and successors of the from_states and to_states of the update. If the transition can be found in the update it is checked whether or not the same transition is in the counterparts forcing actions, if it is, the transition is added to the set of "maybe preserved transitions", if it is not in the counterparts forcing actions it is added to the list of preserved transitions.

At the upgrade-level all of the preserved transitions from the updates are gathered, and the transitions in the maybe-list are determined. As earlier mentioned there can be joint forcing actions, if both agents are stated in the upgrade to preserve their forcing action in the same transition the transition will be preserved, while if only one agent is stated in the upgrade to keep its forcing action the transition with the joint forcing action will not be preserved. The method depicted underneath, in Figure 33 compares the maybe-transitions for the agents and if it is included in both agents maybe-list it is added to the preserved transitions-list, if it is only included in one agents maybe-list it is not added to the preserved transitions-list.

```
974     def updating_model_upgrade_negative(self, upgrade):
975         """ Merges the preserved transitions from updates in the same upgrade,
976             determines whether or not the transitions with
977             joint forcing actions should be preserved.
978             return value : list of all preserved transitions in upgrade. """
979         preserved_transitions_list = []
980         maybe_list = {0:[], 1:[]}
981         for update in upgrade.updates:
982             preserved_transitions, maybe_preserved_transitions = self.updating_model_update_negative(update)
983             for key, value in maybe_preserved_transitions.items():
984                 for v in value:
985                     maybe_list[key].append(v)
986             preserved_transitions_list += preserved_transitions
987         for element in maybe_list[0]:
988             for el in maybe_list[1]:
989                 if element == el:
990                     preserved_transitions_list.append(element)
991         return preserved_transitions_list
```

Figure 33: Determining Preservation of Joint Forcing Actions

The tool has now gathered all information about the transitions with forcing action, it has identified them all and determined which ones will be preserved. In addition to the preserved transitions with forcing actions, all transitions with non-forcing actions will also be preserved, and the change that will actually be done to the model is removing transitions. It is necessary to gather information of all the transitions that will be preserved, forcing and non-forcing, and all transitions in total so the tool can find the transitions that will be removed.

```
1213    def transitions_to_remove(self, preserved_transitions):
1214        """Determine which transitions to remove based on which are forcing
1215            and which are preserved.
1216            Return value : list of remaining transitions with both forcing and
1217            non-forcing actions, list of transitions that will not be preserved
1218            in the upgrade. """
1219        all_transitions = self._model.get_full_transitions()
1220        forcing_act1, forcing_act2 = self.get_forcing_actions()
1221        forcing_actions = []
1222        for element in forcing_act1:
1223            if element not in forcing_actions:
1224                forcing_actions.append(element)
1225        for element in forcing_act2:
1226            if element not in forcing_actions:
1227                forcing_actions.append(element)
1228
1229        forcing_actions_dict = {}
1230        for element in forcing_actions:
1231            if element[0] not in forcing_actions_dict:
1232                forcing_actions_dict[element[0]] = [element[1]]
1233            else:
1234                forcing_actions_dict[element[0]].append(element[1])
1235
1236        temp_removed_transitions = []
1237        for element in forcing_actions:
1238            if element not in preserved_transitions:
1239                temp_removed_transitions.append(element)
1240
1241        removed_transitions = []
1242        for element in forcing_actions:
1243            if element not in preserved_transitions:
1244                removed_transitions.append([element[0], [element[1][0], element[1][1]]])
1245
1246        remaining_transitions = []
1247        for key, value in all_transitions.items():
1248            for val in value:
1249                if [key, val] not in temp_removed_transitions:
1250                    remaining_transitions.append([key, val])
1251
1252        return remaining_transitions, removed_transitions
```

Figure 34: Finding Transitions to Remove

When determining which transitions to remove the information needed is all transitions in the model and the transitions the upgrade preserves, with both forcing and non-forcing actions. The method depicted above, in Figure 34 finds the transitions that will be removed by finding transitions that are not included in preserved transitions. The method also finds the remaining transitions and returns a list of both transitions to remove and remaining transitions.

```
335    def updated_model_negative(self, removed_transitions) -> ATLIrModel:
336        """
337        Negative Updates the Alternating-Time Temporal Logic model with perfect information
338        removing transitions that is not stated to be preserved. Removed transitions as list
339        will be in the form [[(from state, to state), actions(list)],[...]]
340        :return: ATLIr model
341        """
342        for transition in removed_transitions:
343            self.remove_transition(transition[0][0], transition[0][1], transition[1])
344        updated_model = ATLIrModel(self._no_agents)
345        updated_model = self._copy_model(updated_model, self._actions, epistemic=False)
346        ATLIrModel.print_model(updated_model) # only a print
347        return updated_model
```

Figure 35: Updating the Instance of Simple Model with the Negative Upgrade

The list of removed transitions is given the a method in the class Simple Model and all transitions inside the list will be removed from the instance of the Simple Model. There is generated an instance of the class ATLIr Model and the instance is returned to the methods in the Global Model class. The method for updating the instance of the Simple Model class and generating an ATLIr Model is depicted above, in Figure 35.

The instance of the class Simple Model is updated before the executability-check. The main executability condition for negative upgrades is for the CGM to be valid after an upgrade. There are two aspects tested in the method, there has to be at least one out-going transitions from each state in the model and all combinations of the agents actions in a state has to lead somewhere. Underneath, in Figure 36, the executability conditions are checked.

```
1170    def test_negative_clash(self, remaining_transitions):
1171        """ Tests negative executability condition, all states needs at least one out-going transition.
1172            There has to be enough transitions to cover the cartisian product of the actions
1173            in the specific state. Raises exception if it does not hold. """
1174        state_action_dict, action_pairs_counter = self.get_state_actions()
1175        from_states_list = []
1176        for element in remaining_transitions:
1177            from_states_list.append(element[0][0])
1178        for state in self._states:
1179            if state.id not in from_states_list:
1180                raise Exception("There has to be at least one out-going arrow per state.")
1181
1182        state_action_dict = {}
1183        action_pairs_counter = 0
1184        for state in self._states:
1185            actions_agent1 = [i[0] for i in self._model.get_possible_strategies_for_coalition(state.id, [0])]
1186            actions_agent2 = [i[0] for i in self._model.get_possible_strategies_for_coalition(state.id, [1])]
1187            state_action_dict[state.id] = []
1188            for elm in actions_agent1:
1189                for el in actions_agent2:
1190                    state_action_dict[state.id].append((elm, el))
1191                    action_pairs_counter += 1
1192
1193        remaining_transitions_dict = {}
1194        count = 0
1195        test_count = 0
1196        while count < len(self._states):
1197            for el in remaining_transitions:
1198                if el[0][0] == count:
1199                    if count not in remaining_transitions_dict.keys():
1200                        remaining_transitions_dict[count] = [(el[1])]
1201                    else: remaining_transitions_dict[count].append((el[1]))
1202            count += 1
1203
1204        for key, value in state_action_dict.items():
1205            for k, v in remaining_transitions_dict.items():
1206                if key == k:
1207                    test_count += len(set(value).intersection(set(v)))
1208
1209        if action_pairs_counter > test_count:
1210            raise Exception("There is not enough arrows preserved to be a valid concurrent game model.")
```

Figure 36: Test of Clashfreeness for Negative Upgrade

The executability-check happens after the models are updated due to the fact that there can be a smaller set of actions for one or more agents after the updgrade then before, which indicates that if the check was prior to the updating the check might require combinations of more actions in transitions then the actions that are actually preserved.

If the updated version of the model passes the check of executability the formula after the upgrade will be verified and results will appear in the tools CLI. If the updated version does not pass the program will terminate and print out an error-message telling which of the two conditions failed.

5.4 Coalition Logic Operator For Verification

As earlier mentioned the operator in Coalition Logic is a fragment of ATL and the method for verification is the same. In the original version of STV there is no implementation of the fragment, thus it had to be implemented. The developers of STV have implemented verification of the "F"-operator in ATL and has already implemented all needed methods to gather the information required, the "F"-operator is the basis for the new Coalition Logic operator implemented.

```
353    def ucl_next(self, agent_ids: List[int], current_states: Set[int]) -> Set[int]:
354        """Method for verifying coalition operator <<C>>
355            return value : set of states where the formula holds """
356        is_winning_state = self.marked_winning_states(current_states)
357        result_states = set()
358        pre_image = self.prepare_pre_image(current_states)
359        actions = self.get_agents_actions(agent_ids)
360        self.strategy = [None for _ in range(self.number_of_states)]
361        for state_id in pre_image:
362            for action in itertools.product(*actions):
363                if self.is_reachable_by_agents(agent_ids, state_id, list(action), is_winning_state):
364                    self.strategy[state_id] = list(action)
365                    result_states.add(state_id)
366                    is_winning_state[state_id] = True
367                    break
368        print(self.strategy)
369        return result_states
```

Figure 37: Coalition Logic Operator Verification Method

In the theory-section, the semantic interpretation of the Coalition Logic operator was explained as, there is a next state where a coalition can choose to go no matter what anyone else does. To verify that this operator hold there are information needed. It is crucial to know who is included in the coalition and which states does entail what comes after the operator. The algorithm for verifying the CL operator will return a set of states where it holds that the coalition can move to such a state where the right of the formula holds no matter what others do. When verifying the method sets the strategy instance of the ATLIr-model, entailing that when the method has return the states of where the formula holds the strategy for each state, the action that has to be chosen for the formula to hold, is stored.

The variable is_winning_states is a list of Boolean values for each state, where the index of the list is the same as the state ID, if true, the right side of the formula holds in that state, if false, the formula does not hold in that particular state. The variable pre_image is a set of states which are predecessors to the states in winning_states. The current ATLIr-model's instance of strategy is set to a list of as many None as there are states in the model.

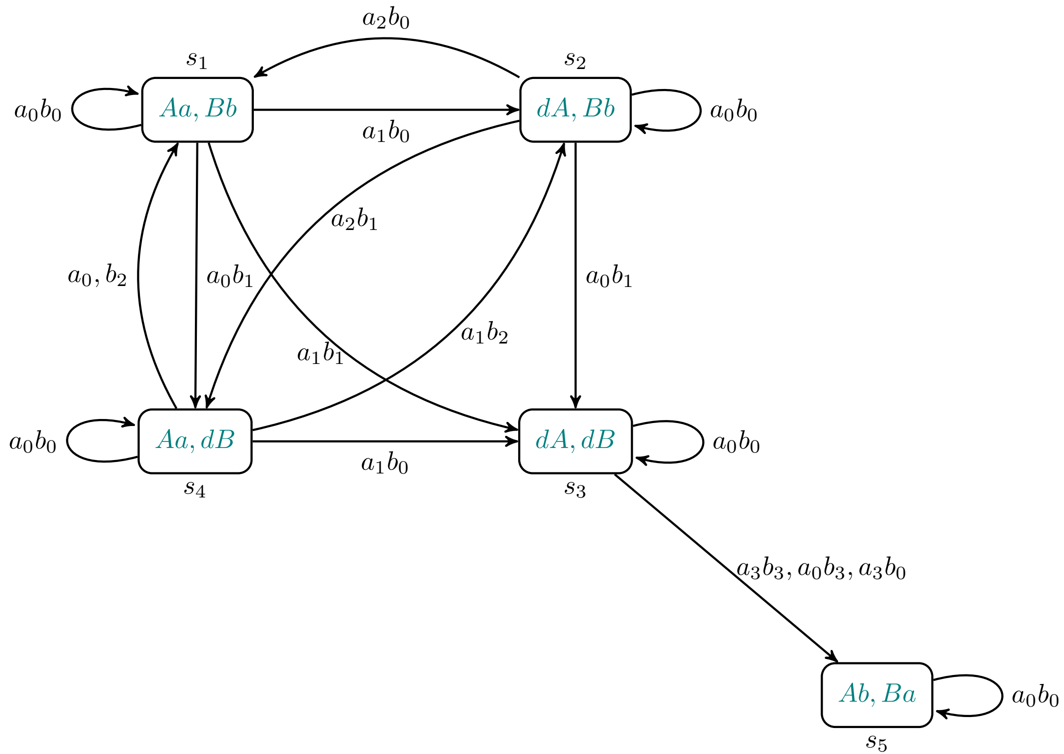
Now, the verifying starts, for each predecessor and each of the coalitions actions, combined in a Cartesian product, goes through a method for determining if the agents, can force a winning

state from a predecessor, if yes, the method returns true and the actions are added to the index of strategy which is the same as the state ID, while if it is false it continues to the next actions.

As mentioned the verification method returns a set of states where the coalition can force a state where the right side of the formula holds. The Boolean value printed as the real result of the model checking-problem is a check if the initial state, which always has the state ID 0, is in the resulting states.

6 StraTegic Verifier - Extended Edition

The extension of STV[2] is intended to verify upgrades in Smart Contracts. With a few limitations it can indeed verify some formulas from the language of DDCL at Smart Contracts when they are modelled as Concurrent Game Models. The limitations are that there can only be to agents in the model and that the formula cannot contain an empty coalition, the third limitation is that the formula verified has to start with either an upgrade or a coalition. In this section we will present the new version of STV and go through the models and formulas from our example and show how new versions are generated and verified using the CLI.



Actions:

i0:	do nothing
i1:	deposit i's asset
i2:	cancel i's deposit
i3:	finalise swap

Figure 38: Initial CGM M

[2]The extension is available at: `https://github.com/eirinmla/STV-master-extended`

To recap, the initial CGM M has five states, the first s_1, Alice and Bob holds their initial assets, they can both either do nothing or deposit their asset. Depending on what Alice and Bob choose to do, the system will transit to one of the three next states which are s_2, s_4, or s_3. It is only possible to finalise the atomic swap when both assets are deposited, in state s_3. If only one of them deposits and the other does not, the one who has deposited can cancel their deposit. Figure 39 illustrates the simple input language of STV and states a version, able to be parsed, of the CGM M in Figure 38.

```
SEMANTICS: synchronous

Agent a:
init: state1
nact: state1 -> state1
nact: state2 -> state2
act: state1 -> state2 [dA=True, Aa=False]
fin: state2 -[dA == True and dB == True]> state4 [Ba=True, Ab=True, dA=False, dB=False]
cancel: state2 -[dA == True and Bb == True]> state1 [Aa=True, dA=False]
nact: state4 -> state4

Agent b:
init: state1
nact: state1 -> state1
nact: state2 -> state2
act: state1 -> state2 [Bb=False, dB=True]
fin: state2 -[dA == True and dA == True]> state4 [Ba=True, Ab=True, dA=False, dB=False]
cancelb: state2 -[dB == True and Aa == True]> state1 [Bb=True, dB=False]
nact: state4 -> state4

PERSISTENT: [Aa, Bb, Ba, Ab, dA, dB]
INITIAL: [Aa=True, Bb=True, Ba=False, Ab=False, dA=False, dB=False]
LOGIC: UCL
FORMULA: <<a>>(dA = True)
```

Figure 39: Input File of the Atomic Swap Example

In the input-file it is stated that the initial state is state1 and the initial values of the propositions are such that Aa and Bb are true, while the rest of the propositions are false, which is the same as in the CGM M's state s_1. The actions are renamed in the input-file, i_0 is renamed as "nact", i_1 as "act", i_2 as "cancel" and i_3 as "fin". When the model is generated the agent's name is added to the action name, e.g. "nact" will become "nact_a" for agent a and "nact_b" for agent b. The tool will give the states identification numbers, in all models the initial state has the number 0, in this particular model the rest of the states in the CGM M are in this order: s_2 is 1, s_4 is 2, s_3

is 3 and s_5 is local states 4-6. In addition the agents has their own identification numbers, Alice's number is 0 and Bob's number is 1, in this particular model.

The CLI is altered to be able to access the extension. The command to start the extension of the program is stated underneath, where the `xxx` should be changed to the filename of the file with the model and the formula that one wants to verify.

```
python3 main.py verify ucl --filename xxx
```

After the command is given the program generates all needed versions of the model and verifies the formula, if the formula holds in the model's initial state the program outputs true, while if the formula does not hold it outputs false. The program does not output a counter-example if false. Since all the output is merely a boolean value, in the extended edition it also returns which states are predecessors end successors in the different updates and which transitions are added or removed, depending on type of upgrade, so it is possible to double-check whether the new implementations work or not. In addition it outputs what the result for all evaluations it carries out, as well as the time spent on the verification of the complete formula.

Now, a review of the result of verification of the formulas. The first formula is the same as in the input-file illustrated in Figure 39, namely;

```
FORMULA: <<a>>(dA = True)
```

The formula indicates that Alice can force a state where it is true that her asset is deposited. Because the initial state is declared in the input-file it is not necessary to write it in the formula.

As seen in Figure 38 the output should be true, as long as Alice choose the action a_1, in the initial state s_1, no matter what Bob does the system will transit to a state where dA is true, either state s_2 or state s_3. The output of the extension looks as below, in Figure 40.

```
● eirin@eirin:~/Master/stv-master$ python3 main.py verify ucl --filename atomic_swap/many_props
  Generation time: 0.0020223000000000046, #states: 7, #transitions: 41
  Formula to be verified:  ['a'](dA = True)
  Coalition [0]
  Winning_states {1, 3}
  Strategy for coalition:  [['act_a'], ['nact_a'], ['act_a'], None, None, None, None]
  Temporary result from coalition expression {0, 1, 2}
  Upper approximation
  Time: 0.0003586279999999997, result: True
○ eirin@eirin:~/Master/stv-master$ 
```

Figure 40: Output for Coalition Formula

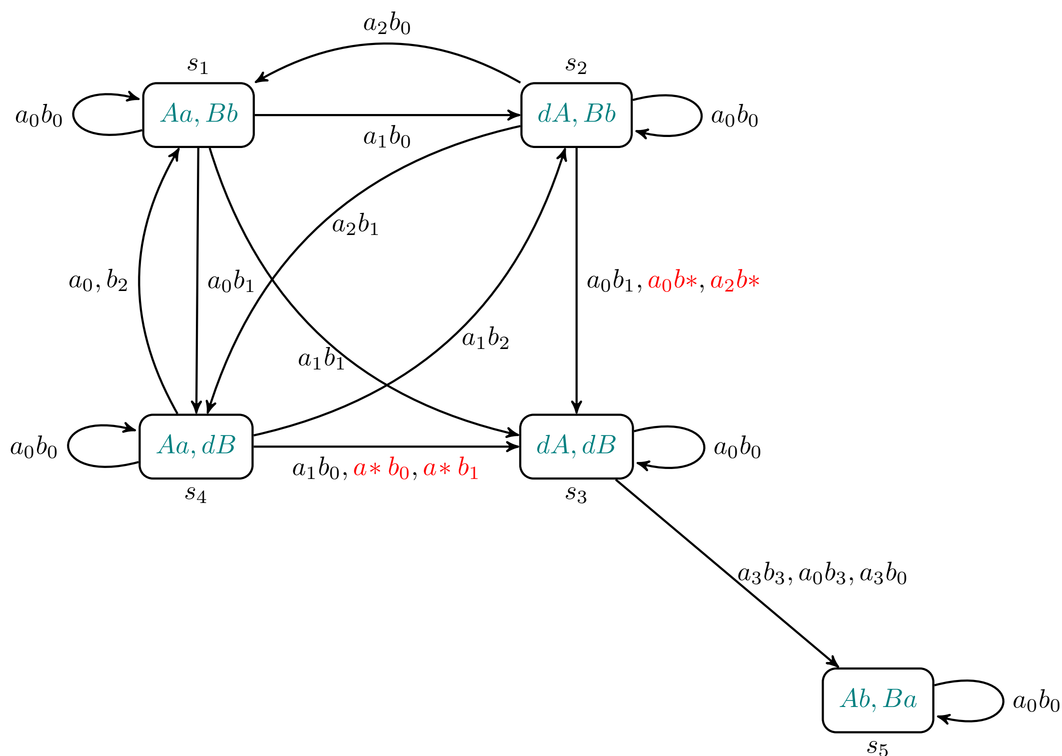
As seen in the output the system needed 0.002 seconds to generate the initial model, generating 7 local states and 41 local transitions. The system can process the entirety of this simple model at once, the coalition is a and the agent ID in the generated model is 0. There are two states where dA is true, state 1 and 3. The verification finds possible strategies for all states in the model, as seen in the result, in this particular verification problem there are only strategies from three states, meaning that the formula in its entirety is true in those particular states, this is reflected in the `Temporary result from coalition expression` which outputs the set of three states as

the result. Because state 0 is in the results the main result is true, the formula holds in the model. The time the system spent on verification was approximately 0.00036 seconds.

To be able to verify a formula in the model M^{+U}, shown in Figure 41, the formula will get more complex. The model in the input-file will still be the same, while the positive upgrade will be added in front of the formula. As remembered the positive upgrade contained the following updates:

$$+U = \{((dA \wedge Bb),\ b,\ (dA \wedge dB))^+, ((Aa \wedge dB),\ a,\ (dA \wedge dB))^+\}$$

Bob is granted a dictatorial power from any state where Alice has deposited her asset and Bob holds his, to any state where both Alice and Bob have deposited their asset. Alice is granted dictatorial powers from any state where Alice holds her initial asset while Bob has deposited his asset, to all states where both Alice and Bob have deposited their asset.



Actions:

i0:	do nothing
i1:	deposit *i*'s asset
i2:	cancel *i*'s deposit
i3:	finalise swap

Figure 41: CGM M^{+U} with Positive Upgrade

The system will generate the model above from a formula including the positive upgrade mentioned. The formula will look such as below in the input language. Instead of an implication, the formula is more generalised then in the theory-section. The formula states that after a positive upgrade Alice and Bob as a coalition can force a state where, from that state, Bob as a coalition can force a state where both Alice and Bob have deposited their assets.

```
FORMULA: {[(((dA = True) & (Bb = True)), b, ((dA = True) & (dB = True)))+,
           (((Aa = True) & (dB = True)), a, ((dA = True) & (dB = True))+]}
           <<a, b>><<b>>((dA = True) & (dB = True))
```

The result of the formula above and the earlier stated model, given in the simple input language is as shown in Figure 42.

```
● eirin@eirin:~/Master/stv-master$ python3 main.py verify ucl --filename atomic_swap/many_props
  Generation time: 0.002321875000000001, #states: 7, #transitions: 41
  Viritual Model is generated
  Update: (((dA = True) & (Bb = True)),b,((dA = True) & (dB = True)))+ grants powers from state
  (s): {1} to state(s): {3}
  New transition(s):  [[1, 3, ['nact_a', 'dict_powers0']], [1, 3, ['cancel_a', 'dict_powers0']]
  ]
  Update: (((Aa = True) & (dB = True)),a,((dA = True) & (dB = True)))+ grants powers from state
  (s): {2} to state(s): {3}
  New transition(s):  [[2, 3, ['dict_powers1', 'nact_b']], [2, 3, ['dict_powers1', 'cancelb_b']
  ]]
  Formula to be verified:  ['a', 'b']['b']((dA = True) & (dB = True))
  Coalition [0, 1]
  Formula to be verified:  ['b']((dA = True) & (dB = True))
  Coalition [1]
  Winning_states {3}
  Strategy for coalition:  [None, ['dict_powers0'], None, None, None, None, None]
  Temporary result from coalition expression {1}
  Winning_states {1}
  Strategy for coalition:  [['act_a', 'nact_b'], ['nact_a', 'nact_b'], ['nact_a', 'cancelb_b'],
   None, None, None, None]
  Temporary result from coalition expression {0, 1, 2}
  Upper approximation
  Time: 0.004906219999999989, result: True
○ eirin@eirin:~/Master/stv-master$ █
```

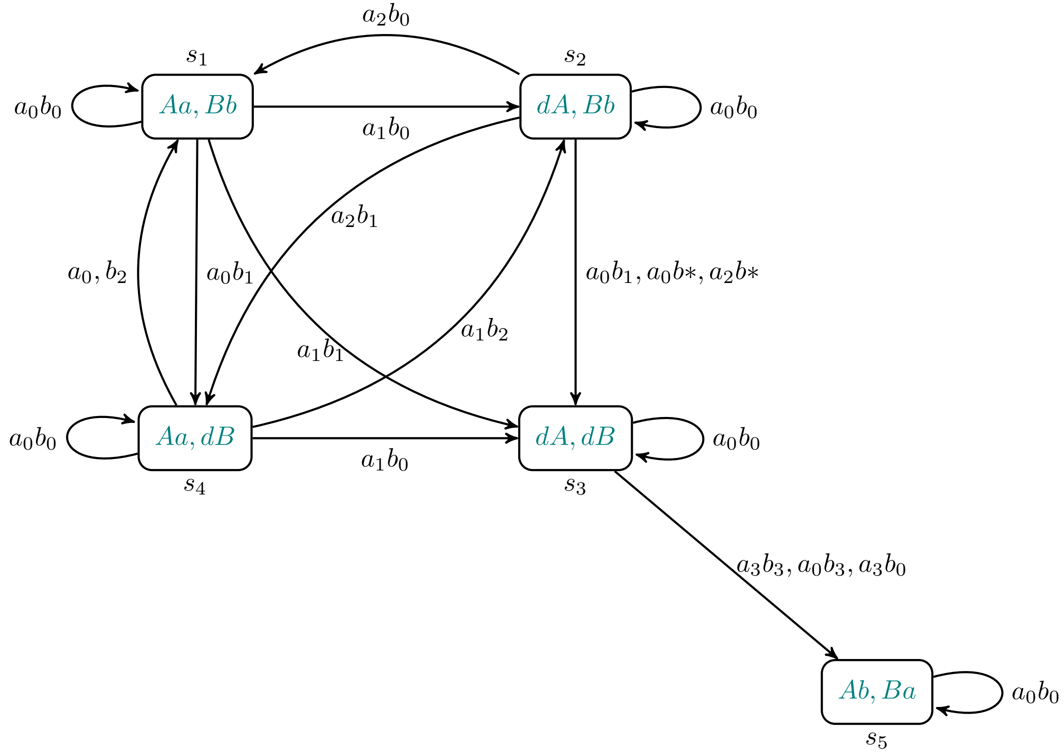
Figure 42: Output for Coalition Formula with Positive Upgrade

First the tool generates the initial model, with 7 local states and 41 local transitions, in 0.002 seconds. Further the system finds predecessors and successors for the updates in the upgrade and generates new transitions. The formula verified contain two coalitions, first a coalition of both agents, Alice and Bob, then a coalition of only Bob. First the coalition of only Bob and the rest of the formula is verified, the temporary results are state 1 (s_2), which is the only state where Bob can force a state where both Alice and Bob has deposited their assets. Then it is verified if the coalition of both Alice and Bob can force state 1 (s_2), they have possible strategies from the first three states and the result is states 0, 1 and 2, (s_1, s_2 and s_4). Because state 0 is included in the result the system outputs a main result being true. The tool spends 0.0049 seconds on the changes in the model and the verification.

As well as positive upgrades, the extension of STV can verify negative upgrades. The next result will include both the previous positive upgrade and the negative upgrade from the theory-section, shown below.

$$-U = \{((dA \wedge Bb), b, (dA \wedge dB))^{-}, ((dA \wedge dB), a, (Ab \wedge Ba))^{-},$$
$$((dA \wedge dB), b, (Ab \wedge Ba))^{-}, ((Ab \wedge Ba), a, (Ab \wedge Ba))^{-}, ((Ab \wedge Ba), b, (Ab \wedge Ba))^{-}\}$$

The negative upgrade includes all transitions with forcing actions that will be preserved. The only forcing actions that will be removed as a result of this negative upgrade is Alice's forcing action to force a state where both Alice and Bob has deposited their asset from any state only Bob has deposited his asset while Alice still holds her initial asset. The final version of the model including both the positive and negative upgrade is shown in Figure 43.



Actions:

i0:	do nothing
i1:	deposit i's asset
i2:	cancel i's deposit
i3:	finalise swap

Figure 43: CGM $M^{+U,-U}$ with Positive Upgrade and Negative Upgrade

$$(M^{+U}, s_4) \not\models [-U]\langle\langle a\rangle\rangle(dA \wedge dB) \tag{18}$$

$$(M^{+U}, s_2) \models [-U]\langle\langle b\rangle\rangle(dA \wedge dB) \tag{19}$$

The formulas that will be verified are formulas 20 and 21. The formulas will be verified in a more generalised way then the formulas from the theory-section, formulas 18 and 19, the formulas will be written with two coalitions, they will not entail the same as formulas 18 and 19, although in this specific model, the result will be the same.

$$M \not\models [+U][-U]\langle\langle a, b\rangle\rangle\langle\langle a\rangle\rangle(dA \wedge dB) \tag{20}$$

$$M \models [+U][-U]\langle\langle a, b\rangle\rangle\langle\langle b\rangle\rangle(dA \wedge dB) \tag{21}$$

In the simple input language formula 20 is written as below.

```
FORMULA: {[(((dA = True) & (Bb = True)), b, ((dA = True) & (dB = True)))+,
            (((Aa = True) & (dB = True)), a, ((dA = True) & (dB = True)))+],
          [(((dA = True) & (Bb = True)), b, ((dA = True) & (dB = True)))-,
            (((dA = True) & (dB = True)), a, ((Ab = True) & (Ba = True)))-,
            (((dA = True) & (dB = True)), b, ((Ab = True) & (Ba = True)))-,
            (((Ab = True) & (Ba = True)), a, ((Ab = True) & (Ba = True)))-,
            (((Ab = True) & (Ba = True)), b, ((Ab = True) & (Ba = True)))-]}
            <<a,b>><<a>>((dA = True) & (dB = True))
```

After the formula and the rest of the input-file is given the tool will output the results. The results for this particular formula and the earlier provided model is shown in Figure 44.

Once again STV spends 0.002 seconds at generating the initial model containing 7 local states and 41 local transitions. One by one update inside the respective upgrades, the predecessors and successors are determined and new transitions are granted in the positive upgrade. In the negative upgrade the removed transitions are determined and removed. As seen the new transitions are the same as in the previous results, and the only removed transitions is Alice's dictatorial powers from state 2 (s_4) to state 3 (s_3).

```
● eirin@eirin:~/Master/stv-master$ python3 main.py verify ucl --filename atomic_swap/many_props
  Generation time: 0.0023241099999999904, #states: 7, #transitions: 41
  Viritual Model is generated
  Update: (((dA = True) & (Bb = True)),b,((dA = True) & (dB = True)))+ grants powers from state
  (s): {1} to state(s): {3}
  Update: (((Aa = True) & (dB = True)),a,((dA = True) & (dB = True)))+ grants powers from state
  (s): {2} to state(s): {3}
  New transition(s):  [[1, 3, ['nact_a', 'dict_powers0']], [1, 3, ['cancel_a', 'dict_powers0']]
  , [2, 3, ['dict_powers1', 'cancelb_b']], [2, 3, ['dict_powers1', 'nact_b']]]
  Update: (((dA = True) & (Bb = True)),b,((dA = True) & (dB = True)))- preserves forcing powers
   from state(s): {1} to state(s): {3}
  Update: (((dA = True) & (dB = True)),a,((Ab = True) & (Ba = True)))- preserves forcing powers
   from state(s): {3} to state(s): {4, 5, 6}
  Update: (((dA = True) & (dB = True)),b,((Ab = True) & (Ba = True)))- preserves forcing powers
   from state(s): {3} to state(s): {4, 5, 6}
  Update: (((Ab = True) & (Ba = True)),a,((Ab = True) & (Ba = True)))- preserves forcing powers
   from state(s): {4, 5, 6} to state(s): {4, 5, 6}
  Update: (((Ab = True) & (Ba = True)),b,((Ab = True) & (Ba = True)))- preserves forcing powers
   from state(s): {4, 5, 6} to state(s): {4, 5, 6}
  Removed transition(s): [[(2, 3), ['dict_powers1', 'cancelb_b']], [(2, 3), ['dict_powers1', 'n
  act_b']]]
  Formula to be verified:  ['a', 'b']['a']((dA = True) & (dB = True))
  Coalition [0, 1]
  Formula to be verified:  ['a']((dA = True) & (dB = True))
  Coalition [0]
  Winning_states {3}
  Strategy for coalition:  [None, None, None, None, None, None, None]
  Temporary result from coalition expression set()
  Winning_states set()
  Strategy for coalition:  [None, None, None, None, None, None, None]
  Temporary result from coalition expression set()
  Upper approximation
  Time: 0.00919603400000002, result: False
○ eirin@eirin:~/Master/stv-master$
```

Figure 44: Output for Coalition Formula with Positive and Negative Upgrade Verifying Alice's Abilities

Now, the final version of the model is generated and the verification begins, there are two coalitions one with both Alice and Bob and one with only Bob. First the one with only Bob is verified, the state that Bob should be able to force is state 3 (s_3). The strategies for Bob states that Bob cannot force state 3 from any states. Due to the fact that Bob cannot force state 3 the coalition with Bob and Alice cannot force a state where Bob can force state 3 and the whole formula turns out to be false as stated in the results. The time spent, to alter the model twice and verify the formula, was approximately 0.009 seconds.

Formula 21 contains the same positive and negative upgrades as formula 20. The interesting part is that it illustrates that it is only Alice who has been revoked her new dictatorial powers, and not Bob. Underneath is the formula in the simple input language of STV.

```
FORMULA: {[(((dA = True) & (Bb = True)), b, ((dA = True) & (dB = True)))+,
            (((Aa = True) & (dB = True)), a, ((dA = True) & (dB = True)))+],
          [(((dA = True) & (Bb = True)), b, ((dA = True) & (dB = True)))-,
            (((dA = True) & (dB = True)), a, ((Ab = True) & (Ba = True)))-,
            (((dA = True) & (dB = True)), b, ((Ab = True) & (Ba = True)))-,
```

```
(((Ab = True) & (Ba = True)), a, ((Ab = True) & (Ba = True)))-,
(((Ab = True) & (Ba = True)), b, ((Ab = True) & (Ba = True)))-]}
<<a,b>><<b>>((dA = True) & (dB = True))
```

As seen in the result underneath, in Figure 45 the results are similar to the results in Figure 44 until both upgrades have been processed, while the verification differ. There are two coalitions, one with both Bob and Alice and one with only Bob. The coalition with only Bob is verified first, the state where both Alice and Bob has deposited their asset is state 3 (s_3). Bob can, as seen from the results, force state 3 from one state, state 1 (s_2). Then verifying that the coalition of both Alice and Bob can force a state where Bob can force state 3, meaning that Alice and Bob can force state 1, results in three states from where Alice and Bob can force state 1. The states are 0, 1 and 2, because 0 is included the main result is true. The time spent on alteration of the model and verification of the formula is 0.0085 seconds.

```
eirin@eirin:~/Master/stv-master$ python3 main.py verify ucl --filename atomic_swap/many_props
Generation time: 0.0023025140000000055, #states: 7, #transitions: 41
Viritual Model is generated
Update: (((dA = True) & (Bb = True)),b,((dA = True) & (dB = True)))+ grants powers from state
(s): {1} to state(s): {3}
Update: (((Aa = True) & (dB = True)),a,((dA = True) & (dB = True)))+ grants powers from state
(s): {2} to state(s): {3}
New transition(s):  [[1, 3, ['cancel_a', 'dict_powers0']], [1, 3, ['nact_a', 'dict_powers0']]
, [2, 3, ['dict_powers1', 'cancelb_b']], [2, 3, ['dict_powers1', 'nact_b']]]
Update: (((dA = True) & (Bb = True)),b,((dA = True) & (dB = True)))- preserves forcing powers
 from state(s): {1} to state(s): {3}
Update: (((dA = True) & (dB = True)),a,((Ab = True) & (Ba = True)))- preserves forcing powers
 from state(s): {3} to state(s): {4, 5, 6}
Update: (((dA = True) & (dB = True)),b,((Ab = True) & (Ba = True)))- preserves forcing powers
 from state(s): {3} to state(s): {4, 5, 6}
Update: (((Ab = True) & (Ba = True)),a,((Ab = True) & (Ba = True)))- preserves forcing powers
 from state(s): {4, 5, 6} to state(s): {4, 5, 6}
Update: (((Ab = True) & (Ba = True)),b,((Ab = True) & (Ba = True)))- preserves forcing powers
 from state(s): {4, 5, 6} to state(s): {4, 5, 6}
Removed transition(s): [[(2, 3), ['dict_powers1', 'cancelb_b']], [(2, 3), ['dict_powers1', 'n
act_b']]]
Formula to be verified:  ['a', 'b']['b']((dA = True) & (dB = True))
Coalition [0, 1]
Formula to be verified:  ['b']((dA = True) & (dB = True))
Coalition [1]
Winning_states {3}
Strategy for coalition:  [None, ['dict_powers0'], None, None, None, None, None]
Temporary result from coalition expression {1}
Winning_states {1}
Strategy for coalition:  [['act_a', 'nact_b'], ['nact_a', 'nact_b'], ['act_a', 'cancelb_b'],
None, None, None, None]
Temporary result from coalition expression {0, 1, 2}
Upper approximation
Time: 0.008445564999999988, result: True
eirin@eirin:~/Master/stv-master$
```

Figure 45: Output for Coalition Formula with Positive and Negative Upgrade Verifying Bob's Abilities

The extension of the tool can verify simple expressions with coalitions, expressions with coalitions and positive upgrades and expressions with coalitions and negative upgrades. In the formula verified there can be multiple upgrades and there can be multiple updates within an upgrade. All combinations has been illustrated above. In addition the tool can verify formulas with coalitions or formulas with upgrades and coalitions within an upgrade's update's from-state or to-state.

6.1 Time Spent on Execution

As seen, the extended version of STV includes features to verify changes in Concurrent Game Models. Formulas of different complexity spends time accordingly. Testing average time of formulas with various complexity has been carried out to get a grasp of execution time. The formulas tested includes various upgrades, shown and elaborated below.

```
[+U¹] = {[(((Aa = True) & (dB = True)), a, ((dA = True) & (dB = True)))+]}
[+U²] = {[(((dA = True) & (Bb = True)), b, ((dA = True) & (dB = True)))+]}
[[+U³]] = {[({[(((Aa = True) & (dB = True)), a,
            ((dA = True) & (dB = True)))+]}<<a>>(dA = True),
            a, ((dA = True) & (dB = True)))+]}
[-U] = {[(((dA = True) & (dB = True)), b, ((Ab = True) & (Ba = True)))-,
        (((dA = True) & (dB = True)), b, ((dA = True) & (dB = True)))-,
        (((dA = True) & (dB = True)), a, ((dA = True) & (dB = True)))-,
        (((Ab = True) & (Ba = True)), a, ((Ab = True) & (Ba = True)))-,
        (((Ab = True) & (Ba = True)), b, ((Ab = True) & (Ba = True)))-]}
```

Figure 46: Upgrades for Testing Time Spent

The first upgrade is granting Alice powers from a state where she holds her initial asset while Bob has deposited his, to a state where both have deposited their asset. The second upgrade is granting Bob powers from a state where he holds his initial asset while Alice has deposited hers, to a state where both have deposited their asset.

The third upgrade is a nested upgrade. The upgrade contains one positive update, inside the update, the from-state is a whole new formula with an upgrade and a formula to verify. The main upgrade states that Alice should be granted powers in all state where the formula with upgrade in the from-state is true, to all states where both Alice and Bob have deposited their assets. Before being able to upgrading the main model a new temporary model has to be generated, updated and verified before moving back to the initial model's updating. The inner upgrade grants Alice dictatorial powers from a state where Alice holds her initial asset and Bob has deposited his, to a state where both have deposited their asset, then the formula inside the from-state has to be verified. From which states in the temporary model, after the upgrading has altered the model, can Alice force a state where both Alice and Bob have deposited. The only state this is true is the state where Alice was just granted her dictatorial powers from, which is the state where Alice holds her initial asset and Bob has deposited his, this is the from-state in the main upgrade.

The fourth upgrade contains negative updates, because it is the preserved abilities that has to be stated in negative upgrades, all other transitions with forcing actions are added to the upgrade

then the forcing actions that will be removed. In this upgrade the transitions with forcing actions that will be preserved are any transition from a state where both Alice and Bob have deposited their asset to any state where the propositions are the same. Any state where the swap is finalised, Alice holds Bob's initial asset and vice versa, to any state where the propositions are the same. And Bob has forcing transitions in any state where both have deposited to a state where Alice holds Bob's initial asset and Bob holds Alice's. All other transitions with forcing actions will be removed from the model.

The table below represents the time it takes from STV has received the command to getting a full result for various formulas. The formulas are run 20 times each and the average is represented in milliseconds. The timing is done in process time indication that only efficient time spent by the CPU for this particular system is taken into consideration.

Time of Verification		
#	Complexity of Formula	Process Time MS
1.	<<a>>(dA = True)	3.527
2.	<<a,b>><<a>>(dA = True)	3.954
3.	[+U^1]<<a>>(dA = True)	6.330
4.	[+U^1, +U^2]<<a>><<b>>((dA = True) & (dB = True))	7.967
5.	[+U^1][+U^2]<<a>><<b>>((dA = True) & (dB = True))	8.243
6.	[-U]<<a>>(dA = True)	10.584
7.	[+U^1,+U^2][-U]<<a,b>><<b>>((dA = True) & (dB = True))	12.886
8.	[[+U^3]]<<a>>(dA = True)	11.339

Table 1: Time of Verification of Formulas with Various Complexity

In Table 1 there are eight formulas with various complexity, starting from the least complex formulas to the most complex. Formula 1 is the smallest formula that the extended version of STV can verify, inside the time of 3.527ms, the tool has generated a model, found states where dA is true and verified if Alice can force at least one of those states. Compared to formula 2, the only difference between them is that formula 1 contains one coalition, while formula 2 contains two coalitions, indicating that the only different time-wise is that in addition to find the states from where Alice can reach a state where dA is true, verification of if Alice and Bob can reach those states is added. As one can see there is minimal difference in time between formula 1 and 2 meaning that the verification itself, does not take much time.

Formula 3 contains one upgrade with one positive update, while the rest of the formula is the same as in formula 1. The additional execution time spent on determining new transitions and adding them is 2.803ms.

In formula 5 there is written [+U^1, +U^2] indicating that the update in upgrade [+U^1] is merged with the update in upgrade [+U^2], including the two updates in the same upgrade adds complexity since it is then necessary to test the executability condition, while if they are tested as two separate upgrades this is not needed since the second upgrade overwrites the first.

Formula 4 and 5 are somewhat similar, although formula four contains two positive updates in-

side one upgrade, while formula 5 contains two upgrades with one positive update in each. The rest of the formulas are the same. The table indicates that two updates in the same upgrade is executed in less time than two upgrades with one update in each, despite the fact that in the first the executability condition has to be checked.

Formula 6, with it's negative upgrade, cost more time than the other formulas up until now. Because the negative upgrade has to contain all transitions with forcing actions that one wants to preserve, it is often more updates inside one negative upgrade than in one positive. Comparing formula 3 and 6 there is one positive upgrade in the first and one negative upgrade in the second, while the rest of the formulas are the same. Inside the positive upgrade there is one update, while in the negative upgrade there are five updates. In addition to determining more from-and to-states, it is also necessary to find transitions with forcing actions when negative upgrade and because there is more then one update in the upgrade the executability conditions have to be checked, thus it seems reasonable that formula 6 spends more execution time than formula 3.

Formula 7 contains both a positive and a negative upgrade, both containing several updates. The formula also includes two coalitions and a simple expression with conjunction. When comparing the formula to the other formulas with similarities like formula 4 and 6, it seems like the negative upgrade spends the lion share of the execution time.

Formula 8 includes a nested positive upgrade. There is needed two generations of models to be able to verify the formula. The formula is interesting in comparison to formula 3. The upgrade `[[+U³]]` really adds the same transitions as `[+U¹]`, in the model tested, although, the calculation of `[[+U³]]` includes several additional steps, making it much more complex. The rest of the formula of formula 8 and 3 is the same, and one can see that the execution time of formula 8 is almost the double of the execution time of formula 3.

The table does not include any surprises. The run-time increases gradually with the complexity of the formula. Next will be a discussions of the assumptions and other limitations.

7 Discussion

The main focus of this thesis has been implementing the ability to generate new versions of models with both revoked and granted dictatorial powers. The approach has been to actually add or remove transitions from the model before verification, in that way the verification algorithms are still the same, although there is much pre-work.

As presented in the results there is a few assumptions made to be able to verify formulas. There has to be two agents, there cannot be an empty coalition and the formula has to start with a coalition or an upgrade. The assumptions gives extensive limitations to the tool.

The limitation of agents restricts the extensiveness of Smart Contracts that can be verified by the tool. Smart Contracts can vary in size and only being able to verify models of Smart Contracts with two agents is a severe limitation. The assumption of being two agents is required throughout most of the implementation of the extension, although improving the code to be more dynamic would fix this limitation. While it is a severe limitation when verifying most Smart Contracts, it is not limiting the conclusion of this thesis, if the logic can be implemented for two agents there is no reason for it to not be possible to be implemented for more agents, although the execution time will be even higher than it is with two agents, because the methods for determining new transitions, forcing actions and removed transitions would be much more extensive. A positive aspect of the limitation is that it prevents state explosion, which will be the case when increasing the number of agents, due to the local states generated for each agent.

The restriction of the formula, by the need of a coalition in the start, is restraining the expressivity of the language of DDCL. The tool cannot verify the whole language of DDCL. All of the language can be parsed, although it is not possible to generate a model without a coalition or an upgrade. Because the main focus of this thesis has been on parsing formulas and changing and verifying models, the generation of the initial model has not been prioritised.

The last assumption made is that there cannot be an empty coalition in the formula verified. Verification with empty coalitions might have been implemented in the original version of STV, although it is not stated anywhere and the sufficient testing to be certain that it is correctly implemented has not been prioritised.

In addition to the three assumptions, there are other limitations. There is no dual to the Coalition Logic operator, the tool does only verify from the initial state and not globally, the tool does not provide a counter-example, the algorithms could be more efficient and the formulas in the input-file could be easier to read and write for the user.

The fact that there is no dual to the operator in $\langle\langle C \rangle\rangle \varphi$ indicates that there is no way to express that all next steps for the coalition will be states where φ is true. Although the dual to $\langle\langle C \rangle\rangle \varphi$ can be rewritten as $\neg\langle\langle C \rangle\rangle \neg\varphi$ this is not possible in the tool, because the formula given has to start with a coalition or an upgrade and cannot start with a negation. There are two alternative ways to fix the limitation, the actual dual can be implemented $[\![C]\!]$ with its own verification algorithm and new additions to the parser stating which of the two versions of the operator the coalition is given in. The second alternative is to fix the generation of the model as earlier mentioned, it has

not been prioritised and the extension of the problem is unknown.

The boolean value that is the end result states whether or not the formula verified holds from the initial state and not from all states with the same propositions as the initial state. Although the strategies are checked for all states, it is only the initial state that determines whether or not the verification problem holds. Extending the model checker to be able to have an implication as the input formula, with an antecedent as all the states that should be the predecessor to the successor where the consequent should be true, would make the model checker global, if then the algorithm for verification would also test that it was true for all the states where the antecedent is true. To be able to extend the model checker to be global, the problem with the initial generation of the model would have to be fixed.

The tool did originally not include counter-examples when the formula does not hold in the model. Including counter-examples would be of great help for the user to be able to locate the semantic weaknesses or bugs in the Smart Contracts. Although the model checker can state if there are weaknesses when verifying formulas in models, it does not show where it does not hold, if it does not. Implementing counter-examples would be of interest to help the user, although it has not been prioritised in this thesis.

The focus has not been at optimisation of the code implemented. Because the tool originally is experimental it is not necessarily optimised to begin with. The optimisation has not been prioritised because the focus in the thesis has been to find out if the extended logic could be implemented, and not how well it would preform in execution time. A model checker without optimisation is still much faster then verifying the model and formula by hand.

The formulas contains many parenthesis in different variations and all propositions has to be stated with their truth-value, which makes the formula difficult to read and write. If the proposition was its own object with values as either true or false, based on if it had a negation in front or not when parsed, the input formula would be much easier to read for the user. One might be able to reduce the amount of parenthesis if the simple expression would be simplified, which would make it easier to write the formulas for the user.

Possible exciting extensions going further would be to expand the tool to include ATL, non-dictatorial powers and asynchronous semantics.

Expanding the tool to include upgrades with ATL would enable all the expressivity that ATL entails, verifying not only one step forward as CL does, but also full sequences. Because the approach chosen in this thesis has been to change the model before verifying the rest of the formula, the verification happens on a regular Concurrent Game Model, making it fully possible to verify with ATL as well as CL.

It is reasonable to believe that not all changes in a Smart Contract is granting and revoking dictatorial powers, it would be interesting to find out how the logic would act out granting and revoking non-dictatorial powers as well. It would be expected that it is easier to grant non-dictatorial powers than revoking them, because of the representation of the upgrades. If one would have to include all actions that will be preserved both dictatorial and non-dictatorial for all agents to be

able to remove some actions, the upgrade would contain an extreme amount of updates. While when granting non-dictatorial powers it would be quit easy to add a transition from all to-states to all from-states without needing to add them for all the different combinations of the other agents to make them dictatorial.

While the scope of this thesis has been synchronous semantics, it would be reasonable to believe that asynchronous semantics is more realistic when it comes to execution of Smart Contracts at the blockchain. This is because the network might not receive the transactions from the agents at the same time.

8 Conclusions and Future Work

It is possible to deploy new versions of Smart Contracts or parts of Smart Contracts on the blockchain even though the initial instance of the Smart Contract is immutable. In formal verification the main focus has been on verifying Smart Contracts before the first deployment and not while comparing two versions of a Smart Contract. In the thesis, we implement a model-checker that allows verification of upgrades of Smart Contracts. In particular, we extend the existing tool, STV, so that it can handle specifications expressed in a recently introduced logic [Galimullin and Ågotnes, 2021]. The logic, Dictatorial Dynamic Coalition Logic, is an extension of Coalition Logic and enables reasoning about upgrades.

The implementation is able to verify upgrades in Smart Contract in some extent. The approach generates new models for each upgrade and verification is located in the new version of the model. Because the approach focuses on changing the models it can easily be extended to other temporal logics that can be verified in Concurrent Game Models. The limitations of the implementation is that there can only be two agents in the model, there are restrictions on how the formula can be formulated and the model checking tool verifies only from the initial state and not globally.

In the future it would be interesting to implement the upgrades to ATL as well as CL, giving the model checking problem a lot more expressivity. Another interesting aspect would be to look into granting and revoking actions that are not dictatorial, it seems likely that not all new actions one can do in a Smart Contract is of the dictatorial sort. In addition looking into asynchronous semantics would be interesting since it is more realistic when working with Smart Contracts and blockchain.

References

[Ågotnes et al., 2015] Ågotnes, T., Goranko, V., Jamroga, W., and Wooldridge, M. (2015). Knowledge and ability.

[Almakhour et al., 2020] Almakhour, M., Sliman, L., Samhat, A. E., and Mellouk, A. (2020). Verification of smart contracts: A survey. *Pervasive and Mobile Computing*, 67:101227.

[Antonopoulos, 2017] Antonopoulos, A. M. (2017). *Mastering Bitcoin: Programming the open blockchain*. " O'Reilly Media, Inc.".

[Antonopoulos and Wood, 2018] Antonopoulos, A. M. and Wood, G. (2018). *Mastering ethereum: building smart contracts and dapps*. O'reilly Media.

[Atzei et al., 2017] Atzei, N., Bartoletti, M., and Cimoli, T. (2017). A survey of attacks on ethereum smart contracts (sok). In *International conference on principles of security and trust*, pages 164–186. Springer.

[Clarke et al., 2018] Clarke, E. M., Henzinger, T. A., and Veith, H. (2018). Introduction to model checking. In *Handbook of Model Checking*, pages 1–26. Springer.

[CoinGecko, 2023] CoinGecko (2023). Crypto market cap charts. https://www.coingecko.com/en/global-charts.

[Corwin, 2023] Corwin, S. (2023). Formal verification of smart contracts. https://ethereum.org/en/developers/docs/smart-contracts/formal-verification/.

[Galimullin and Ågotnes, 2021] Galimullin, R. and Ågotnes, T. (2021). Dynamic coalition logic: Granting and revoking dictatorial powers. In *International Workshop on Logic, Rationality and Interaction*, pages 88–101. Springer.

[Galimullin and Ågotnes, 2023] Galimullin, R. and Ågotnes, T. (2023). Action models for coalition logic. In *Dynamic Logic. New Trends and Applications: 4th International Workshop, DaLí 2022, Haifa, Israel, July 31–August 1, 2022, Revised Selected Papers*, pages 73–89. Springer.

[Gammie and Van Der Meyden, 2004] Gammie, P. and Van Der Meyden, R. (2004). Mck: Model checking the logic of knowledge. In *Computer Aided Verification: 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004. Proceedings 16*, pages 479–483. Springer.

[Holzmann and Peled, 1995] Holzmann, G. J. and Peled, D. (1995). An improvement in formal verification. In *Formal Description Techniques VII*, pages 197–211. Springer.

[Jamroga et al., 2016] Jamroga, W., Knapik, M., and Kurpiewski, D. (2016). Fixpoint approximation of strategic abilities under imperfect information. *arXiv preprint arXiv:1612.02684*.

[Kurpiewski, 2022] Kurpiewski, D. (2022). Blackbat13/stv: Stv - strategic verifier. https://github.com/blackbat13/stv, journal=GitHub.

[Kurpiewski et al., 2019a] Kurpiewski, D., Jamroga, W., and Knapik, M. L. (2019a). Stv: Model checking for strategies under imperfect information. In *Proceedings of the 18th International Conference on Autonomous Agents and Multiagent Systems AAMAS 2019*, pages 2372–2374. IFAAMAS.

[Kurpiewski et al., 2019b] Kurpiewski, D., Knapik, M. L., and Jamroga, W. (2019b). On domination and control in strategic ability. In *Proceedings of the 18th International Conference on Autonomous Agents and Multiagent Systems AAMAS 2019*, pages 197–205. IFAAMAS.

[Marketcap, 2023] Marketcap, C. (2023). Amazon (amzn) - market capitalization. https://companiesmarketcap.com/amazon/marketcap/.

[Merz, 2000] Merz, S. (2000). Model checking: A tutorial overview. *Summer School on Modeling and Verification of Parallel Processes*, pages 3–38.

[Nakamoto, 2008] Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system. *Decentralized business review*, page 21260.

[Nam and Kil, 2022] Nam, W. and Kil, H. (2022). Formal verification of blockchain smart contracts via atl model checking. *IEEE Access*, 10:8151–8162.

[Pauly, 2002] Pauly, M. (2002). A modal logic for coalitional power in games. *Journal of logic and computation*, 12(1):149–166.

[Pratap, 2022] Pratap, Z. (2022). Upgradable smart contracts: What they are and how to deploy your own. https://blog.chain.link/upgradable-smart-contracts/.

[Preston, 2023] Preston, E. (2023). Testing smart contracts. https://ethereum.org/en/developers/docs/smart-contracts/testing/.

[Schobbens, 2004] Schobbens, P.-Y. (2004). Alternating-time logic with imperfect recall. *Electronic Notes in Theoretical Computer Science*, 85(2):82–93.

[Siegel, 2016] Siegel, D. (2016). Understanding the dao attack. https://www.coindesk.com/learn/2016/06/25/understanding-the-dao-attack/.

[Team, 2023] Team, C. (2023). 2022 biggest year ever for crypto hacking. https://blog.chainalysis.com/reports/2022-biggest-year-ever-for-crypto-hacking/.

[Tolmach et al., 2021] Tolmach, P., Li, Y., Lin, S.-W., Liu, Y., and Li, Z. (2021). A survey of smart contract formal specification and verification. *ACM Computing Surveys (CSUR)*, 54(7):1–38.

[van der Meyden, 2018] van der Meyden, R. (2018). On the specification and verification of atomic swap smart contracts. *arXiv preprint arXiv:1811.06099*.