Modeling Smart Contracts in UPPAAL

Project-2 (CS4D002) end-term report submitted in partial fulfillment for the award of the degree of

BACHELOR OF TECHNOLOGY,

In

COMPUTER SCIENCE

Submitted by

Ashwin Kottapally

Roll No: - 19CS01041

Under the supervision of

Dr. Srinivas Pinisetty,

SCHOOL OF ELECTRICAL SCIENCES,

INDIAN INSTITUTE OF TECHNOLOGY, BHUBANESWAR

(2019-2023)

INDIAN INSTITUTE OF TECHNOLOGY, BHUBANESWAR

Bhubaneswar, India

CERTIFICATE

Certified that the project work entitled **Modeling Smart Contracts in UPPAAL** was carried out by **Mr. Kottapally Lucky Ashwin Kumar**, Roll No **19CS01041**, a bonafide student of **Indian Institute of Technology Bhubaneswar** in partial fulfilment for the award of Bachelor of Technology in Computer Science Engineering during the year **2022-23**. It is certified that all corrections/suggestions indicated for internal assessment have been incorporated in the report deposited in the departmental library. The project report has been approved as it satisfies the academic requirements in respect of project work prescribed for the said degree.

Name and Signature of Guide

Name and Signature of Examiner

Name and Signature of Head of the School

Declaration

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea / data / fact / source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

_____

(Signature)

_____

(Name of the student)

_____

(Roll No.)

Acknowledgments

With deep regards and profound respect, I avail this opportunity to express my deep sense of gratitude and indebtedness to my supervisor Dr. Srinivas Pinisetty, School of Electrical Sciences, Indian Institute of Technology, Bhubaneswar for his inspiring guidance, constructive criticism and valuable suggestion throughout this work. He has helped me in times where I got struck in between to understand the concepts and he shared the references to go through to get out from that phase. Always motivated and encouraged to work towards the goal with very positive attitude. It would have not been possible for me to bring out this thesis without his help and constant encouragement.

Contents

1. Abstract

2. Introduction

- **2.1.** Problem Domain
- **2.2.** Work Summary

3. Background

- **3.1.** Introduction to UPPAAL
  - **3.1.1.** Interface
  - **3.1.2.** Edges
  - **3.1.3.** Channels
  - **3.1.4.** Locations
  - **3.1.5.** Properties
  - **3.1.6.** Examples
- **3.2.** Introduction to Bitcoin

4. Literature Study:

- **4.1.** Introduction to Bitcoin transactions
  - **4.1.1.** The keys, the secret strings, and the signatures
  - **4.1.2.** The transactions
  - **4.1.3.** The parties
  - **4.1.4.** The Adversary
  - **4.1.5.** BlockChainAgent and notion of time
  - **4.1.6.** Modeling the Bitcoin-based commitment scheme
  - **4.1.7.** Results of verification

5. Case Study

- **5.1.** Land Registry
- **5.2.** Flow Chart
- **5.3.** Solution

6. Modeling and Verification of Land Registry Scenario

- **6.1.** Case-1: Land Registration using three parties
- **6.2.** Case-2: Land Registration using seven parties

7. Conclusion

8. References

1. Abstract

UPPAAL model checking tool is used for modeling, validation, and verification of real-time systems which involve timing constraints. It is appropriate for systems that can be modeled as a collection of nondeterministic processes with finite control structures and communicating through channels or shared variables. UPPAAL model checking is generally preferred over many other model checkers because we can simulate the outcomes according to our need in the simulator window and it is very easy to visualize the automata in UPPAAL as it contains an inbuilt GUI to create the automata. Using a verifier, we can check the safety and liveliness properties of any model based on the given requirements. For this reason, UPPAAL has been used in the following project to model the land registration process using blockchain technology.

In the real-estate business, land is a high valued asset that must have accurate records which identify the current owner and also all known previous owners' history. [4] Land registration is a critical process that involves the transfer of ownership of a piece of land from one entity to another. This process is traditionally paper-based, which can lead to inefficiencies, delays, and fraud. However, with the use of blockchain technology, land registration can be made more secure, transparent, and efficient. Land registration using blockchain technology is more efficient because the world is going digital and people are accessing everything in just one click through the internet. Blockchain transactions are done instantly and transparently. Smart contracts are digital contracts stored on a blockchain that is automatically executed when predetermined terms and conditions are met. In the project, smart contracts are broadcasted as advertisements that the party is interested to sell its land for some value.

Currently, research is going on these blockchain-based techniques and I think we need an automated way to verify the correctness of the system against various desired properties related to security, safety etc. For this reason, I have used UPPAAL as a model-checking tool to model land registration to check and verify its security. In this work, land registry system is modeled using UPPAAL, and we have analysed various properties such as ADVERSARY cannot confuse the blockchain by introducing malicious transactions and parties involved will have safe and secure transaction and exchange of ownership.

2. Introduction

Modeling is the process of creating an abstract representation of a system using mathematical, graphical or other technique. [7] It involves simplifying of the real-world system into a set of variables, rules, and relationships that can be manipulated and analyzed. Modeling is also used in many different fields including engineering, physics, economics, and computer science to understand complex systems in real life and predict their behavior easily using modeling tools using automated analysis.

Automated analysis generally refers to the use of algorithms and tools that analyze and verify the models automatically. [7] It involves running the simulations to check for errors and also generating report of the behavior of the system under different conditions. Automated analysis helps to identify and improve the problems and weakness in the system and also suggests improvements in the simulation or model. It is very useful in complex systems which involves analysis and time-consuming or impractical.

In real life modeling and automated analysis is used to design and to test the systems under consideration. [7] Software models are created to check the behaviour of the system and then we use automated analysis to verify whether the model behaves as expected or not under different conditions. This will help to identify errors in the model and suggest improvements in the model before we can deploy the software in the main domain, which reduce the risk of failure or security issues in the software model.

There are several approaches to modeling and automated analysis, each with its own strengths and weaknesses. Some of the most commonly used approaches are:

1) **Formal methods:** Formal methods involve the use of mathematical logic and formal languages to create precise models of the systems. [7] These models can be analyzed using automated tools to verify their correctness and behavior. Formal methods are particularly useful for safety-critical systems, such as medical devices.

2) **Simulation**: Simulation on the other hand involves creating a simplified model of a system and running it under different conditions to observe its behavior. Simulations can be used to test hypotheses and optimize

system performance, or predict future behavior. Simulation is widely used in engineering, physics, and economics, among other fields.

3) **Model checking:** Model checking involves the automatic verification of a model against a set of properties or specifications. [7] Model checking can help to identify potential problems or errors in the model and suggest improvements. Model checking is commonly used in software engineering and hardware design. There are several model checking tools available and the most commonly used are UPPAAL, SPIN, NuSMV, CBMC, PRISM, Alloy and VeriFast. UPPAAL is a tool for modeling and verifying real-time systems. It supports several modeling formalisms, including timed automata and networks of timed automata.

2.1. Problem Domain

I have explored the features of UPPAAL to analyze and verify the security of smart contracts. Smart contracts in blockchain technology offers various range of benefits that make it an attractive solution for many different use cases, from financial transactions to supply chain management to identity verification. The benefits of blockchain technology are decentralization, transparency, security, efficiency, traceability and innovation. Land registration is one such process which has many challenges in real life system. And blockchain technology can help the process to make it more safe, secure and fast. For this reason, there is potential research can be done in this context using and automated and modeling analysis. UPPAAL is one such tool which offers its services and I have explored this idea of land registration using smart contracts in UPPAAL.

2.2. Work Summary

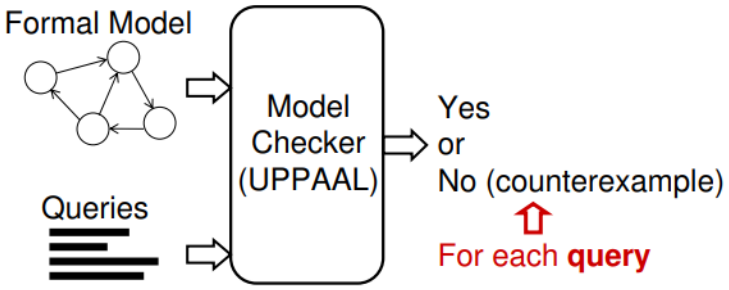
I have considered the case study of Land registration process and have explored the different cases with some assumptions and analyzed and verified the security of the model in UPPAAL. As land a high valued asset required an automated and trusted way to transfer the ownership between the two involved parties is currently cumbersome in real life as it involves storing, organizing and tracking of property ownership records. Some of the challenges are documents loss, inefficient verification, delay in document signing, time taking for money transfer and involvement of 3rd parties as brokers. These all challenges can be eliminated for some extent if we use blockchain technology in land registration process. I have explored this idea and worked on the model of creating a safer and secured system which can be used as an alternative to current process. Further research is also going on in this domain to make land registration process to make easy, safe and secure. The work that I have done might help to certain extent to understand how it can be done and can be generalized further with more complex parameters and involving many factors into account.

3. Background

3.1) Introduction to UPPAAL

[9] In applications that involve hardware and software is used we need to ensure that failure doesn't occur. Therefore, failure is unacceptable. For this reason, we need to validate the design of the application (also called "**Design Validation**") ensuring the correctness of the design as early as possible in the development stage. In design validation the most used traditional techniques which are Simulation (on a model of the system) and Testing (on the actual product) which won't get exhaustive validation. Formal methods in design validation are Deductive Verification (costly, slow, and only partially automatic) and **Model Checking** (for finite-state concurrent systems automatic) which aim ate exhaustive validation of the design. Therefore, we use model checking as a validation method to work toward our problem statement.

The most general steps of model checking are (1) Building a model for the system which typically looks like a set of automata that has different states and transitions in between states. (2) Formalizing the properties that need to be verified using the expressions in logic. (3) Use the model checker as a tool to generate the space of all possible states and then exhaustively check whether the properties hold in every one of the possible **dynamic behaviors** of the model. Below Fig1 shows the steps of model checking clearly. And **UPPAAL** is one such model-checking tool to check whether the model satisfies the given requirements.



(source)

Fig1: Steps in Model Checking

3.1.1) Interface

UPPAAL is a model-checking tool based on the java interface used for design validation via graphical simulation and verification of the model via automatic model checking of real-time systems. [9] The main parts of the UPPAAL tool are (1) a Graphical User Interface (GUI) which is executed on the user work station application. (2) a Model-Checker Engine which by default is executed on the same computer as the user interface but can also run on a more powerful server as well. It has been jointly developed by **Upp**sala University in Sweden and **Aal**borg University in Denmark. Below Fig2.1 shows the interface of the UPPAAL tool where we can see we have three tabs in the left window which are Editor (model designing in fig2.1), Simulator (timed model checking in fig2.2), and Verifier (model verifier in fig2.3).

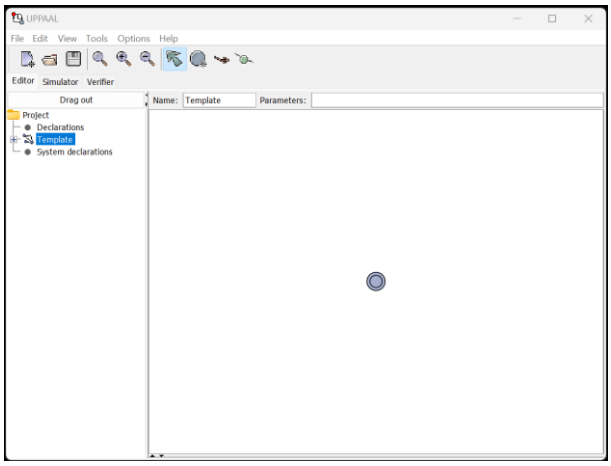


Fig2.1: Editor Window

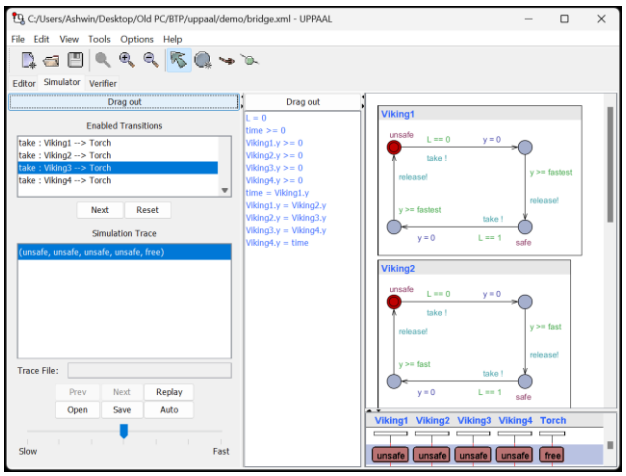


Fig2.2: Simulator Window

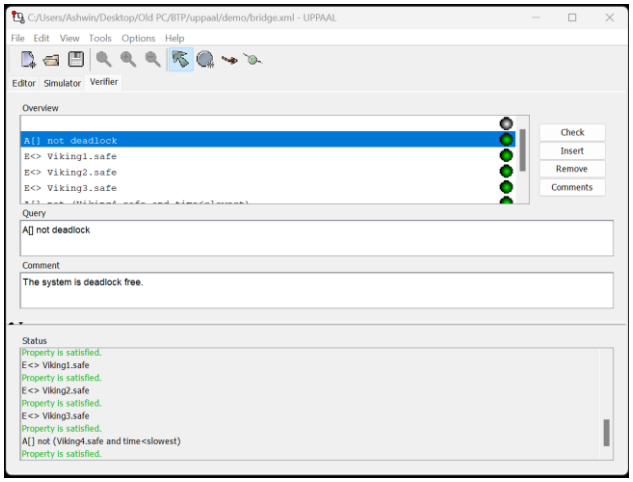


Fig2.3: Verifier Window

While modeling the systems in UPPAAL we use timed-automata which are finite state machines with clocks which are variables that are defined in declarations in the simulator (fig2.1) which can evaluate to a real number and can be defined in each automaton to measure the time progress of the model. And all clocks start and evolve at the same pace to represent the global progress of time. The actual value of the clock can be either tested or reset but cannot be assigned to some value. As UPPAAL is specially designed for the verification of real-time systems, clocks are the fundamental modeling and verification feature.

3.1.2) Edges

The structure of a uppaal model is a timed-automaton represented as a graph that has locations as nodes and edges as arcs between locations. Edges are annotated with guards, updates, synchronizations, and selections. A **guard** is an expression that uses the variables and clocks of the model which are defined in declarations to indicate when the transition is enabled. [7] An **update** is an expression that is evaluated as soon as the corresponding edge is enabled and this evaluation changes the state of the system. Synchronization is the basic mechanism used to coordinate the action of two or more processes. It causes two or more processes to take transition at the same time. In the below fig3.1 we can see that we have 2 nodes start and end which is joined by an edge that has guard condition as a==1 and update statement as b=0.

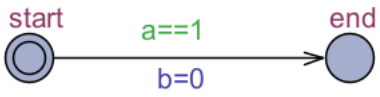


Fig3.1: Simple model

3.1.3) Channels

A channel (c) is declared in synchronization, then one process will have an edged annotated with c! and the other(s) process(es) another edge annotated with c? (Refer to fig3.2). And three different kinds of synchronizations are (1) Regular channel (3) Urgent channel (3) Broadcast channel.

A regular channel is declared as e.g., **chan c**. When a process is in allocation from which there is a transition labeled with c! the only way it is enabled is if there is another process in the location from which there is a transition labeled with c? and vice versa. And there are several possible pairs of c! and c? one of them is non-deterministically chosen during model checking. And update expression on edge with c! is executed before that of edge with c? An urgent channel is similar to a regular channel except that no time delay occurs to trigger synchronization and it is declared as an **urgent chan c**. Graphically looks similar to regular channels and clock guards are not allowed on edges synchronizing over the urgent channels. A broadcast channel is declared as **broadcast chan c**. And it is also similar to a regular channel except that if a transition c! labeled and one or more processes with transition labeled with c? all these transitions are enabled. And similar to urgent channel clock guards are not allowed on edges.

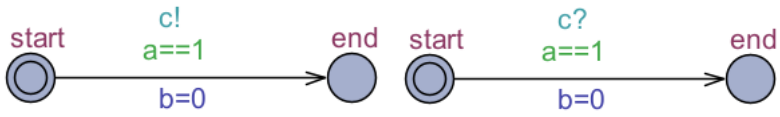


Fig3.2: Model with chan c

3.1.4) Locations

Locations are of three different types which are (1) Initial (2) Urgent (3) Committed (4) Normal. **The initial** location (node) is the start node which is identified with a **double circle**. **Urgent** locations **freeze time** which is time that is not allowed to pass when a process is in an urgent location. And it is identified by "U" where the clock will not change as long as it is in at the urgent location. The committed locations also freeze time but when a model has one or more active committed locations, no transitions other than those leaving said locations can be enabled. It is identified as "C". Normal locations are the rest of the locations other than those discussed above. Another parameter that is used is

Invariants which are conditions that must be fulfilled while the automaton is in that location. Below fig3.3 shows all the locations and invariants involved.

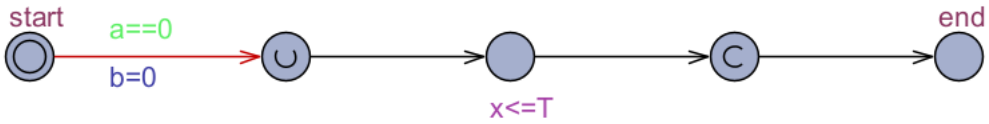
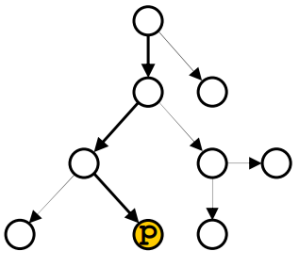


Fig3.3: Simple model with locations and invariant

3.1.5) Properties

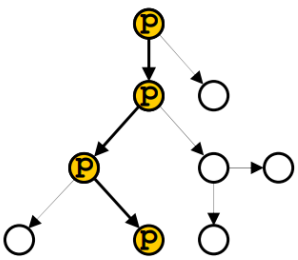
After using the simulator to ensure that the model behaves as the system we wanted to model and sometimes also to detect some errors in the original design, the next phase is to check that the model verifies the properties. For that, we need to decide what the properties are and then formalize them. After that, we need to translate those properties into the UPPAAL query language which the uppaal tool understands. The specific types of properties that can be expressed in the UPPAAL query language are (1) Reachability properties (2) Safety properties (3) Liveness properties (4) Deadlock properties

[9] In **Reachability Properties** there is an execution path in which there exists a path to reach a node that will eventually hold. And it is represented by E<>p which means "Exists eventually p". Below fig4.1 shows the path to reach p. In **Safety Properties** there is an execution path which there holds for all states of the path. And it is represented by E[]p which means "Exists globally p". Below fig4.2 shows such a path that involves all the states as p.



(source)

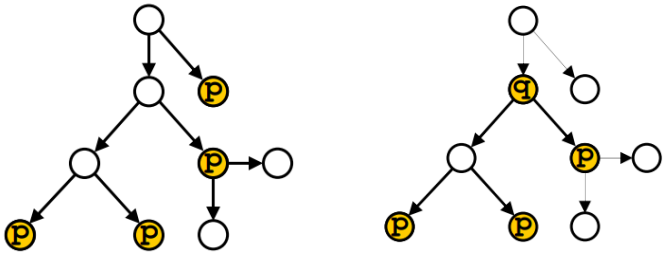
Fig4.1: Reachability property



(source)

Fig4.2: Safety property

In **Liveness Properties** have 2 types which are (1) A<>p means "Always eventually p" for all execution path p holds for at least one state of the path (2) q->p means "q always leads to p" as any path that starts with a state in which q holds reaches later a state in which p holds (Refer to fig4.3).



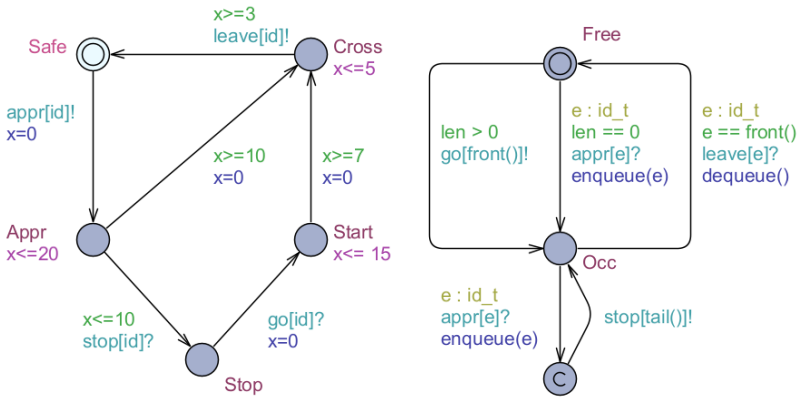
(source)

Fig4.3: Liveness property

Deadlock Properties when a state is in a deadlock state if it is not possible that the model evolves to the next state neither by waiting some time nor by a transition between locations. And most typical examples are (1) E<>deadlock means "Exists deadlock" (2) A[]not deadlock means "There is no deadlock".

3.1.6) Examples:

Eg1: Train-Gate Model (Train and Gate models respectively)



Declarations:

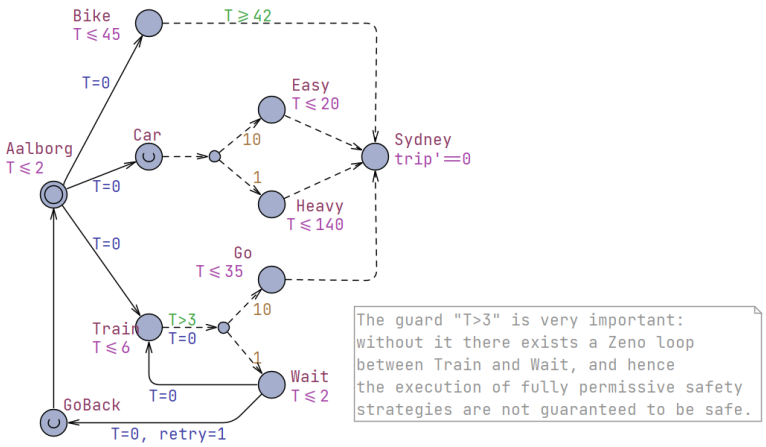
```
const int N = 6;          // number of trains
typedef int[0,N-1] id_t;
chan    appr[N], stop[N], leave[N];
urgent chan go[N];
```

Explanation:

We can observe that each train has a channel to communicate between the train model and the gate model. Initially, any train starts at the Safe state and moves towards the Appr state with invariant at Appr as x<=20. And if x<=10 means it has come before and needs to wait sometime until the gate is free. And if x>=10 (initialized to x=0) it will cross and the time to cross is x<=5 then it will again reach a safe state. But if it stopped before then the gate will allow trains to cross based on FCFS. Therefore, the gate maintains a queue where the first train that arrived in the queue is sent first and accordingly others based on the availability.

Eg2: A person named who has to reach Sydney from Aalborg. There are a few modes of transportation that he can take but each will have different possibilities.

Model: Transportation model



Declarations:

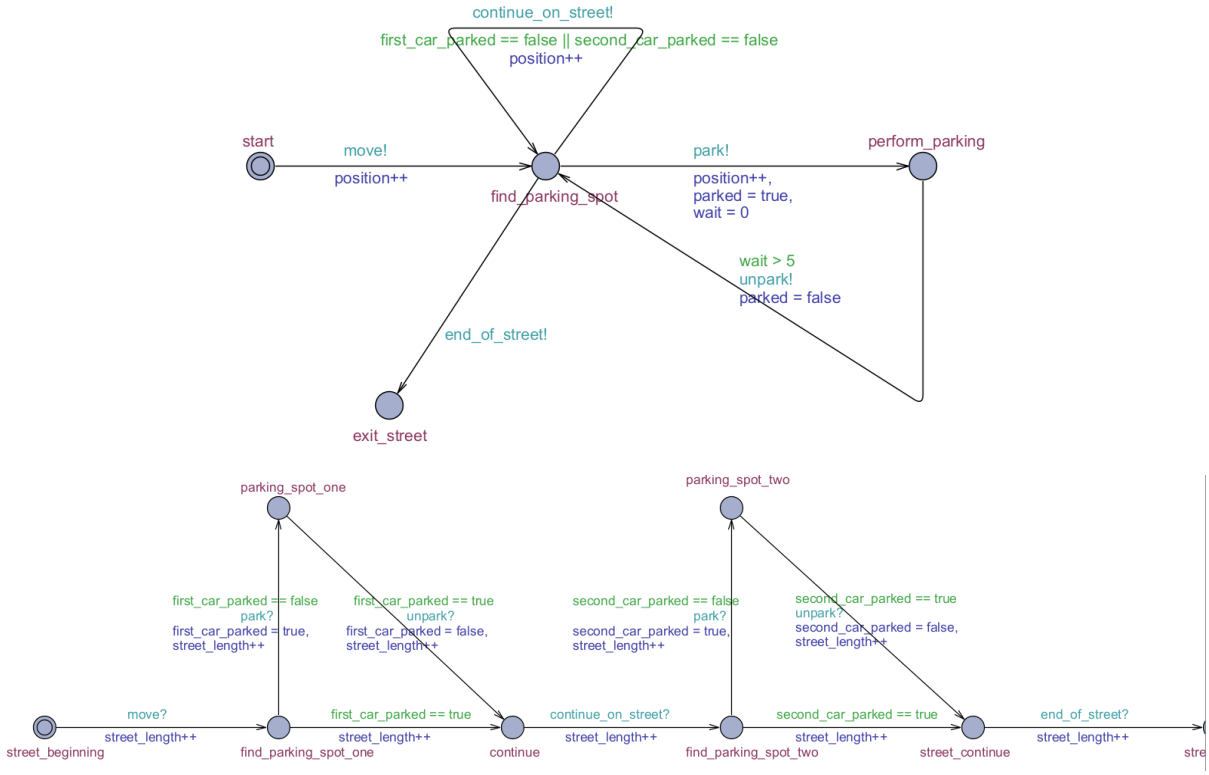
```
clock time; // global time reference in minutes
clock T;    // clock to track the transportation.
hybrid clock trip; // stop-watch to measure trip time
```

Explanation:

In the above case, there is a person who has to reach Sydney from Aalborg. There are a few modes of transportation that he can take but each will have different possibilities. And here we used UPPAAL Stratego which gives us the strategy to go from the initial state to the final state in some specific time. It can be used in writing some verification conditions in a verifier and it provides us with a simulator which is a plot that gives us an idea of how it can be achieved.

Eg3: Two lanes and car checks for parking spots.

Model:



Explanation:

Based on Car and Lane models the system checks the available slots for parking and it can be automatically tested using the simulator and writing verifier properties. We can also check the deadlock condition property to know whether a car is struck or not.

3.2) Introduction to Bitcoin

Bitcoin is a digital currency system that was introduced in 2008 by an unknown person or group of people using the name "Satoshi Nakamoto". [2] Bitcoin is a peer-to-peer cryptographic currency system. Since its introduction it has gained huge popularity because of a few reasons that are (1) transaction fees are very low (2) it's not controlled by any central authority which means no one can print the money to generate inflation. And the transaction involves so-called contracts where several mutually-distrusting parties engage in a protocol to perform financial tasks and fairness is guaranteed by the properties of bitcoin. Even though bitcoin contracts have several potential applications in the digital economy but still it's not widely used because contracts are hard to create and analyze and hence quite risky.

As nobody can print money to generate inflation the financial transactions are published on a public ledger maintained jointly by the users of the bitcoin network. Now people are showing interest in the bitcoin network because transaction fees are less which in the case of doing online transactions via current banking systems charges a huge amount. And similarly, our money is deposited into banks that are controlled by some authority and there is no guarantee that our money is safe. Bitcoin not only allows simple transactions but also complicated transactions such that one can claim some amount of money under certain conditions. And these conditions are written in the form of bitcoin scripts which might involve timing constraints. The main obstacle that prevents bitcoin to become widely used is that contracts are hard to write and analyze. Because while writing the scripts we can make a few subtle mistakes and protocol which involves several parties and timing constraints are hard to analyze by hand. And these problems can be used in the wrong way by malicious parties for their financial gain.

4. Literature Study

4.1) Introduction to Bitcoin Transactions

[8] Bitcoin transactions are generally time-dependent which are (1) takes time for transactions to get confirmed (2) bitcoin transactions can come with a "time lock" which means when a particular transaction becomes valid. Generally, the transactions happen between participants based on their private key and public key. And it is called key pairs frequently denoted by capital letters as A.sk (private or secret key) and A.pk (public key). We can use A = (A.sk, A.pk) as another convention to represent key pair. $sig_A(m)$ denotes the signature on a message m. The main difficulty of peer-to-peer networks is that users jointly maintain the ledger in a way that should not be manipulated by an adversary and is publicly accessible.

The ledger is implemented as a chain of blocks also called a "block chain". When a transaction is posted in the block chain it takes some time before the user can be sure that this transaction will not be cancelled. Let's assume an upper bound on this waiting time as "MAX_LATENCY". And an address is simply a public key pk and every such key have a corresponding private key sk known only to the owner of this address. Here the private key is used for signing the transactions and the public key is for verifying the signatures. Therefore, in our model, each party executing the protocol is modeled as a timed automaton with a structure assigned to its based-on party's knowledge. For communication between the parties, the model can use synchronization on channels offered by the UPPAAL tool. Protocols were verified by us and the messages exchanged between parties were signatures. Therefore, we model the communication indirectly using shared variables. Each party keeps the set of known signatures by simply adding them to the set.

4.1.1) The keys, the secret strings, and the signatures

Assuming the number of key pairs is known in advance and constant. And here the key pairs are referred to by consecutive natural numbers. A type key is defined as an integer from a given range. Secret strings are also modeled similarly. As assumed public keys and hashes of all secrets are known to all parties. From the below fig6.1, we can see how the signature is structured.

```
typedef struct {
    Key key;
    TxId tx_num;
    Nonce input_nonce;
} Signature;
```

Fig6.1: Structure for Signature

4.1.2) The transactions

We assume that all transactions that can be created by the honest parties come from a set, which is known in advance and of size T. [8] Additionally, the adversary can create its own transactions. As we assumed the upper-bound the number of adversarial transactions by T. Therefore, a total upper bound on the number of transactions is 2T. From the below implementation of fig6.1, the num field is the identifier of the transaction, and the input field is the identifier of its input transaction. The value field denotes the value of the transaction (in B). The timelock field indicates the time lock of the transaction, and the timelock_passed is a boolean field indicating whether the timelock has passed. The status contains the following values: UNSENT (indicating that the transaction has not yet been sent to the block chain), SENT (the transaction has been sent to the block chain and is waiting to be confirmed), CONFIRMED (the transaction is confirmed on the block chain, but not spent), SPENT (the transaction is confirmed and spent), and CANCELLED (the transaction was sent to the block chain, but while it was waiting for being included in the block chain its input was redeemed by another transaction). The out_script denotes the output script. In case the transaction is standard it simply contains a public key of the recipient of the transaction.

```
typedef struct {
    TxId num;
    TxId input;
    Status status;
    OutputScript out_script;
    int value;
    int timelock;
    bool timelock_passed;
    Nonce nonce;
    bool reveals_secret;
    Secret secret_revealed;
} Tx;
```

Fig6.1: Structure for Transactions

4.1.3) The Parties

From below fig6.2, the boolean tables know_key[KEYS_NUM] and know_secret[SECRETS_NUM] describe the sets of keys and secrets respectively known to the party: know_key[i] = true if and only if the party knows the i-th secret key, and know_secret[i] = true if and only if the party knows the i-th secret string. The integer known_signatures_size describes the number of the additional signatures known to the party (i.e., received from other parties during the protocol), and the array known_signatures contain these signatures.

```
typedef struct {
    bool know_key[KEYS_NUM];
    bool know_secret[SECRETS_NUM];
    int[0,KNOWN_SIGNATURES_SIZE] known_signatures_size;
    Signature known_signatures[KNOWN_SIGNATURES_SIZE];
} Party;
```

Fig6.2: Structure for Party

4.1.4) The Adversary

The real-life bitcoin adversary can create an arbitrary number of transactions with arbitrary output scripts, so we need to limit its possibilities so that possible states are finite and are of reasonable size. [8] We use a generic automaton, which does not depend on the protocol being verified and allows us to send to the block chain any transaction at any time assuming some conditions are met, e.g., that the transaction is valid and that the adversary can create its input script.

There are two ways the adversary can influence the execution of protocols that are either (1) creating a transaction identical to the transaction that is verified (2) a transaction redeems one of the transactions from the protocol.

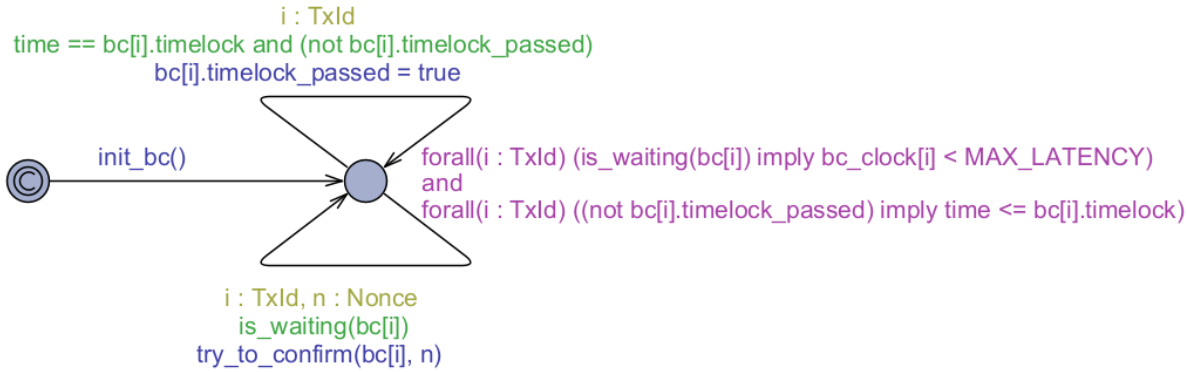
The output scripts in the transactions of 2nd type do not matter, so we may assume that an adversary always sends them to one particular key known only to him. Now adversary is a party, that can send an arbitrary transaction from this set if only he can do so (e.g., he can evaluate the input script, and the transaction's input is confirmed, but not spent). If the only actions of the honest parties are to post transactions on the block chain, then one can assume that this is also the only thing that the adversary does. In this case, his automaton, denoted Adversary is very simple: it contains one state and one loop, that simply tries to send an arbitrary transaction from the mentioned set. That can be seen in the right loop of fig6.3.

Fig6.3: Model for Adversary

In some protocols, the parties besides posting the transactions on the block chain, also exchange messages with each other. This can be exploited by the adversary, and hence we need to take care of this in our model. This is done by adding more actions in the Adversary automaton. This is reflected by the left loop in the Adversary automaton.

4.1.5) BlockChainAgent and notion of time

Block chain is a shared structure containing information about the status of all the transactions denoted BlockChainAgent. One of the duties of BlockChainAgent is to ensure that the transactions which were broadcast are confirmed within appropriate time frames. To post a transaction t on the block chain, a party p first runs the try_to_send (Party p, Tx t) function, which broadcasts the transaction if it is legal. And the can_send function checks if (a) the transaction has not been already sent, (b) all its inputs are confirmed and unredeemed and (c) a given party p can create the corresponding input script. The only non-trivial part is (c) in case of non-standard transactions, as this check is protocol-dependent.



Once a transaction t has been broadcast, the BlockChainAgent automaton attempts to include it in the block chain (lower loop in the above figure). The BlockChainAgent automaton also takes care that every transaction gets included in the block chain in less than MAX_LATENCY time, which is a constant that is defined. This is done by the invariant on the right state in the figure that guarantees that every transaction is waiting for confirmation less than MAX_LATENCY.

4.1.7) Results of Verification

Before running the verification procedure in UPPAAL it is necessary to choose, which parties are honest and which are malicious.

In the case of honest Bob, the property that we checked is the following:

```
A[] (time >= PROT_TIMELOCK+MAX_LATENCY) imply (hold_bitcoins(parties[BOB]) ==
1 or parties[BOB].know_secret[0] or BobTA.failure)
```

which, [8] informally means: "after time PROT_TIMELOCK + MAX_LATENCY one of the following cases takes place: either (a) Bob earned 1B, or (b) Bob knows the committed secret or (c) Bob rejected the commitment in the commitment phase".

In the case of honest Alice, the property we verified means that Alice does not lose any bitcoins in the execution of the protocol (even if Bob is malicious).

```
A[] (time >= PROT_TIMELOCK) imply (parties[BOB].know_secret[0])
A[] (time >= PROT_TIMELOCK) imply (hold_bitcoins(parties[ALICE]) == 1)
```

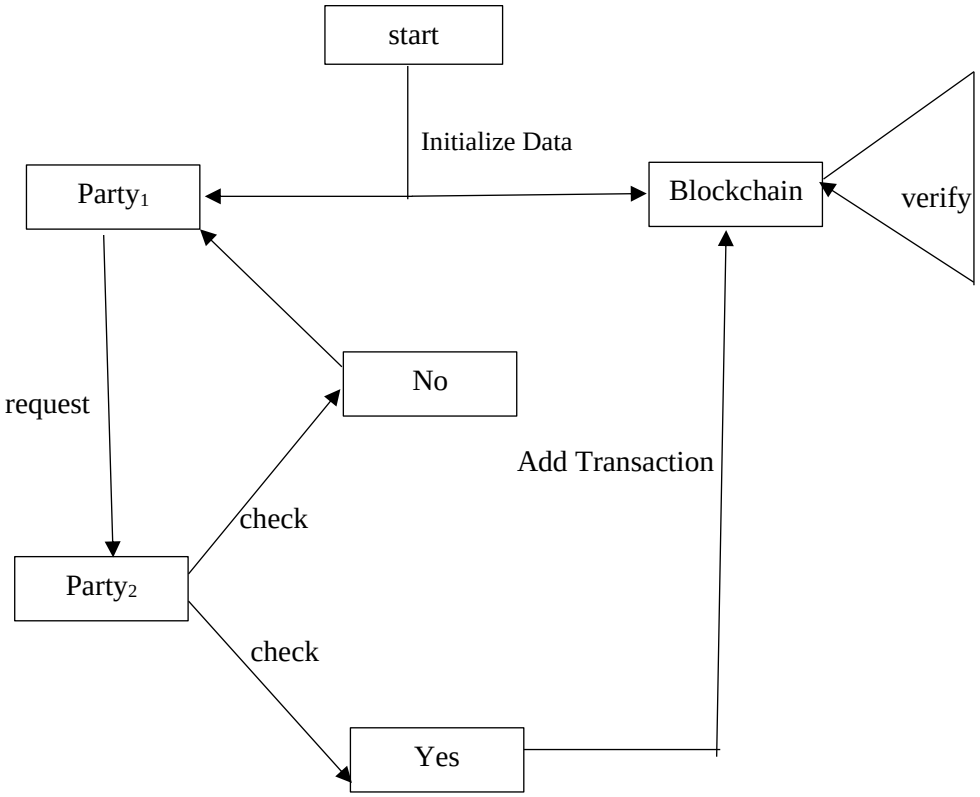
The first one states that after time PROT_TIMELOCK Bob knows the secret (which can be not true, if Alice refused to send it). The second one states that after time PROT_TIMELOCK Alice holds 1 B (which occurs only if Alice is honest, but not in general). The UPPAAL model checker confirmed that these properties are violated if one of the parties is malicious, but holds if both parties follow the protocol (i.e. when honest automata are used for both parties).

5) Case Study

5.1) Land Registry

In the real-estate business, the land-a-high valued asset must have accurate records which identify the current owner and also all known previous owners' history. Currently, the existing solutions to trust the system are out of date. Storing, organizing, and tracking all the information of property ownership records was challenging for the land registry agency when we have many land records to maintain and there might be many discrepancies within the paperwork such as forged documents and misinformation. Also, if the property owners want to buy any land they look into many aspects of all previous owners and the history behind the land and area they are going to buy for their future. In this way there are many challenges like (a) Incidents of documents loss (b) Inefficient Verification (c) Delay due to cumbersome document-signing (d) Time taking process for registration and money transfer (e) Involvement of third parties or brokers

5.2) Flow Chart



5.3) Solution

If we can develop a land registry model in UPPAAL using similar lines of smart contracts in bitcoin protocols we can reduce many before-mentioned challenges. The solution developed must be decentralized that enables buyers and sellers to deal directly without intermediaries. And using UPPAAL we can trace the malicious parties involved and inform victim parties about malicious activity before happening. As it is a multi-agent system, we involve two parties that are buyer and seller and Land Inspector to verify the documents and initiates transfer of property. And it creates a channel between them for a transaction that is done in the blockchain. And electronic signature is used in between sellers and buyers to sign the property ownership transfer of documents and those signatures also stored in blockchain. Transactions are stored in blockchain to ensure authenticity and traceability. [4] And the benefits of using smart contracts in land registration are, (a) Incorporation of record integrity protection (b) Eliminating cost of third parties and brokers (c) Delivered transparency with smart contracts (d) Automated land registry and transaction process (e) Accelerated land registry and safer.

6) Modeling & Verification

In our model, we have LandInspector, Buyer and Seller models involved and as this is basic model, I have considered two cases in which the 1st case covers the basic model involving only single seller (ALICE), single buyer (BOB) and single ADVERSARY. In the 1st case the seller who is ALICE has ownership to certain lands and it wants to sell to BOB and ADVERSARY tries to get access to transactions from blockchain and imitate those transactions as such it has sent. Finally, transaction between ALICE and BOB happens correctly and ADVERSARY transactions are rejected by the blockchain. Similarly in the 2nd case as well we used same seller and buyer concept but with more complex model using five SELLERs, one BUYER and one ADVERSARY. In this case as well each SELLER has ownership over certain lands and it sells its lands to BUYER and ADVERSARY disturbs these transactions. Finally, the model is able to identify the correct transactions and make a safe transaction possible.

5.1) Case-1

6.1.1) Global Declarations

Initially, each party contains its own ID and balance that each party owns. And it also maintains the secret key and public key for signing the smart contract and further adding that into blockchain. Each party has ownership over certain lands and it is maintained as an array with all lands and the lands which the party owns is true else false.

```
typedef struct
{
    PartyId Id;
    int balance;
    bool own_land[LAND_NUM]; //bool of land owned
    int secretkey;
    int publickey;
} Party;
```

Before initiating a transaction, the party that wants to sell (ALICE) will create a smart contract and broadcast that contract to all the other parties involved in the model. These contracts are maintained in an array where other parties have access, so that they can initiate a transaction. And the “amnt” is the price that seller is interested to sell the “landId”.

```
typedef struct
{
    PartyId Id;
    int amnt;
    LANDID landId;
    int signature;
} smart_contract;
```

Transactions are maintained in structure format were the blockchain is maintained as an array of transactions. Transactions have fromId and toId which specifies that seller id and buyer id and amnt for which they are ready to do the transaction on landId. Transaction maintains a status to know whether its is just added into blockchain (CONFIRMED) or already the transaction is spent (SPENT) or transaction is cancelled (CANCELED). Transaction also maintains the respective signatures of seller and buyer. Finally blockchain is maintained as array of transactions.

```
//Different states of transaction
const Status CONFIRMED = 0;
const Status SPENT = 1;
const Status CANCELED = 2;

typedef struct
{
    PartyId fromId;
    PartyId toId;
    int amnt;
    LANDID landId;
    Status status;
    int signature1;
    int signature2;
} Tx;

Tx blockchain[TX_NUM];
```

Before initiating a transaction, the party creates a smart contract and that process is taken up by the sell_land function which creates a smart contract and broadcasts them.

```
void sell_land(PartyId id, int amount, LANDID landid)
{
    smart_contract contract;
    contract.Id = id;
    contract.amnt = amount;
    contract.landId = landid;
```

```
    contract.signature = parties[id].secretkey * parties[id].publickey;
    sc[sc_curr] = contract;
    sc_curr++;
}
```

When buyer and seller both come to an agreement to initiate a transaction then the buyer will add the transaction copy with its signatures into blockchain.

```
void add_trans(PartyId fromid, PartyId toid, int amount, LANDID landid, int
Signature1, int Signature2)
{
    Tx transaction;
    transaction.fromId = fromid;
    transaction.toId = toid;
    transaction.amnt = amount;
    transaction.landId = landid;
    transaction.status = 0;
    transaction.signature1 = Signature1;
    transaction.signature2 = Signature2;
    blockchain[bc_curr] = transaction;
    bc_curr++;
}
```

6.1.2) LandInspector

Local Declarations

Before starting the model, the blockchain makes sure that all parties have their respective values and ownership over lands. And this process is taken up in initialize function which initialize the values to parties. Below all the even landIds are assigned to ALICE and all the odd landIds are assigned to BOB and nothing is assigned to ADVERSARY.

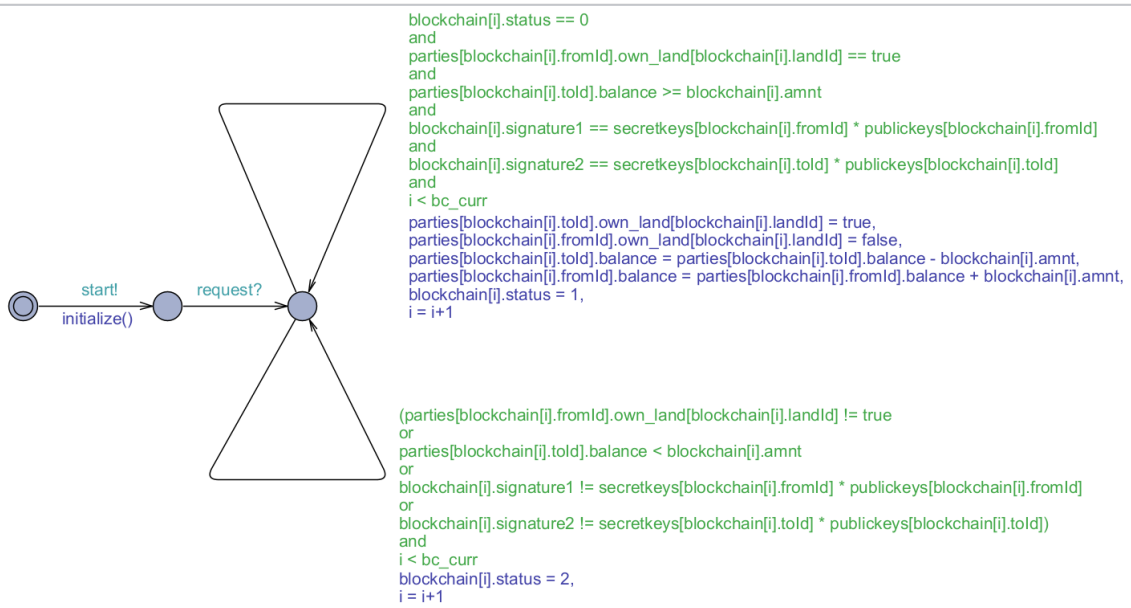
```
// Place local declarations here.
int secretkeys[3] = {90,88,22};
int publickeys[3] = {16,8,18};
void initialize()
{
    for(i : PartyId)
    {
        parties[i].Id = i;
        parties[i].balance = INIT_AMNT;
        parties[i].secretkey = secretkeys[i];
        parties[i].publickey = publickeys[i];
        for(j : LANDID)
        {
```

```
            if(i == 0)
            {
                if(j%2 == 0)
                {
                    parties[i].own_land[j] = true;
                }
                else
                {
                    parties[i].own_land[j] = false;
                }
            }
            else if(i == 1)
            {
```

Model

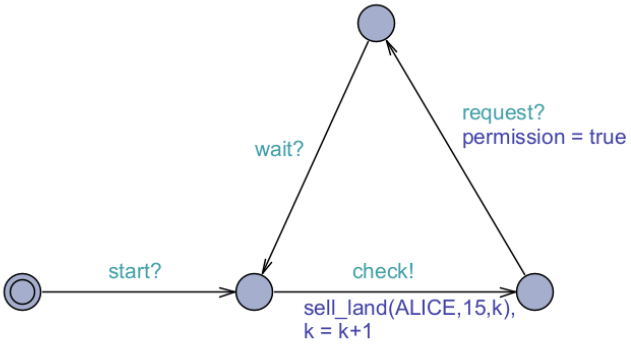
Initially the LandInspector initialize the parties with its data using initialize function which is defined in its local declarations. And the two loops that we see signifies that each transaction in blockchain is thoroughly verified before any updates happen to the values of parties. In the above loop we observe that status of transaction is CONFIRMED and ownership of land is owned by ALICE and BOB has enough amount in its balance to purchase the land and the signature are checked using basic algorithm of multiplication where the other parties involved won't decode the signature as it involves secret key and public key which forms a signature. When all the things satisfy then the ownership is transferred to BOB and ALICE will get money credited in its balance.

In the second loop if any of above-mentioned values won't satisfy then the status of transaction is CANCELED and further moved to next transaction. In this way all the transactions are cover until a pointer in blockchain array which signifies the number of transactions added.



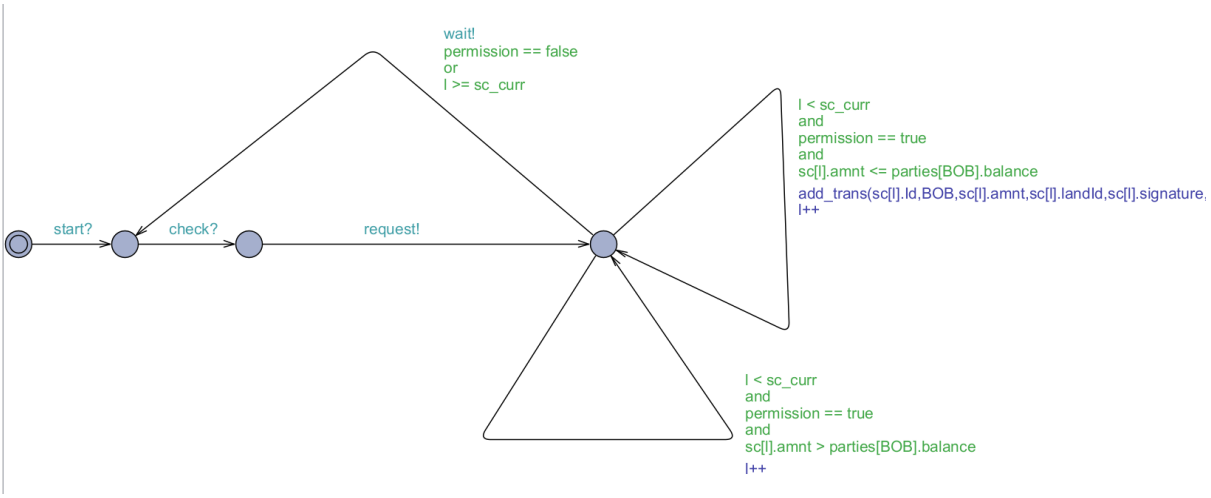
6.1.3) ALICE

ALICE as a seller will initiate a transaction using a smart contract which is broadcasted to others. And here it makes use of sell_land function to do so. Again, after completing the triangle loop to gets back to selling other lands.



6.1.4) BOB

BOB the buyer checks whether ALICE has added it transaction in smart contract and then bob access the contract and asks for permission to buy the land from ALICE if it has enough amount to buy the land. On accepting the permission, BOB initiate a transaction into blockchain using add_trans function by adding the transaction into blockchain. Again, loops back to starting to check whether ALICE has again added some transactions.



6.1.5) ADVERSARY

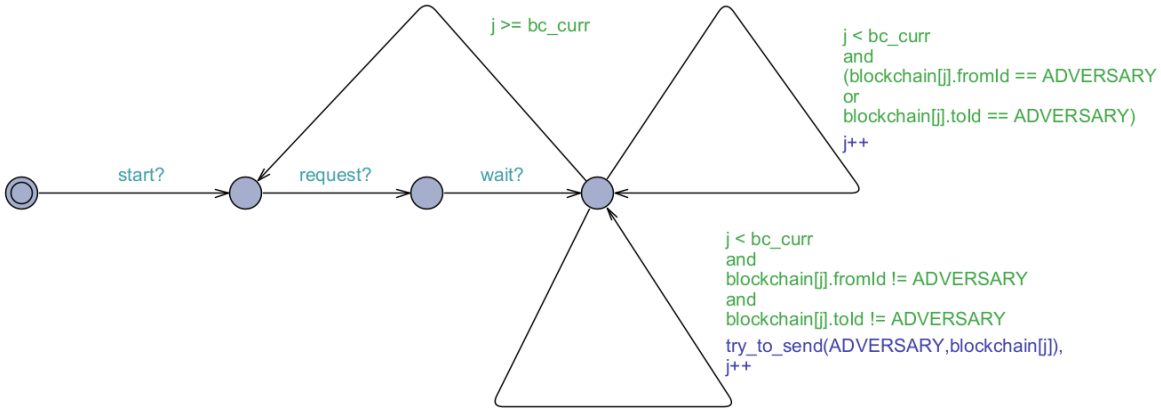
Local Declarations

Adversary task is to disturb the transaction of ALICE and BOB using its own transactions adding into blockchain to confuse the LandInspector. Adversary tries to add two types of transactions which are manipulating the fromId and toId to its party id and adding these transactions into blockchain hoping that it gets benefited from it.

```
void try_to_send(PartyId id, Tx transaction)
{
    add_trans(transaction.fromId,id,transaction.amnt,transaction.landId,transa
ction.signature1,transaction.signature2);
    add_trans(id,transaction.toId,transaction.amnt,transaction.landId,transact
ion.signature1,transaction.signature2);
}
```

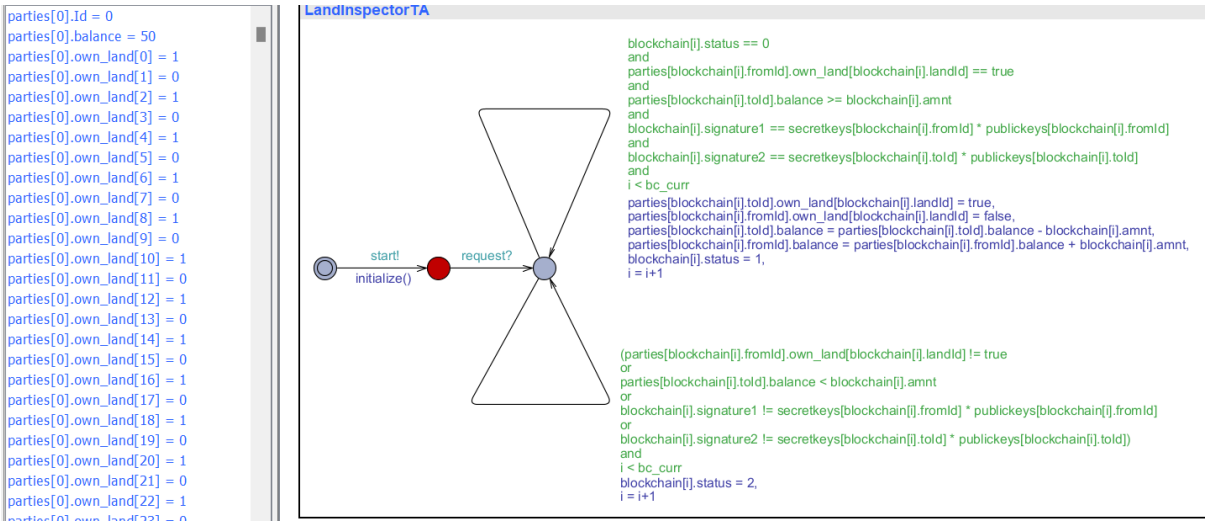
Model

Adversary makes sure that it won't duplicate its own transaction as it knows the transactions in blockchain that are fair are between ALICE and BOB. So its tries to check the fromId and toId that are not its id and then its try_to_send function to add the two transactions into blockchain to confuse the LandInspector.

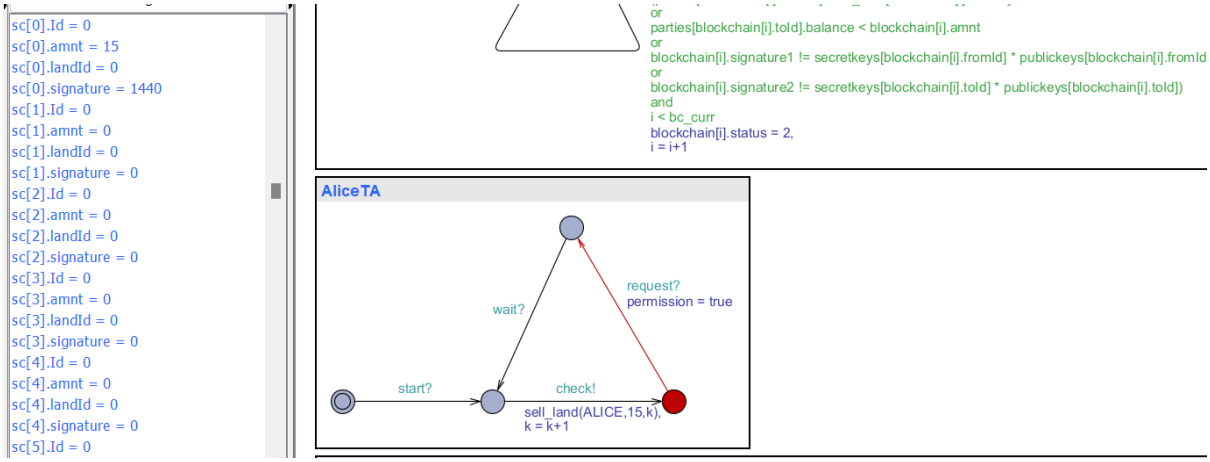


6.1.6) Simulation Results

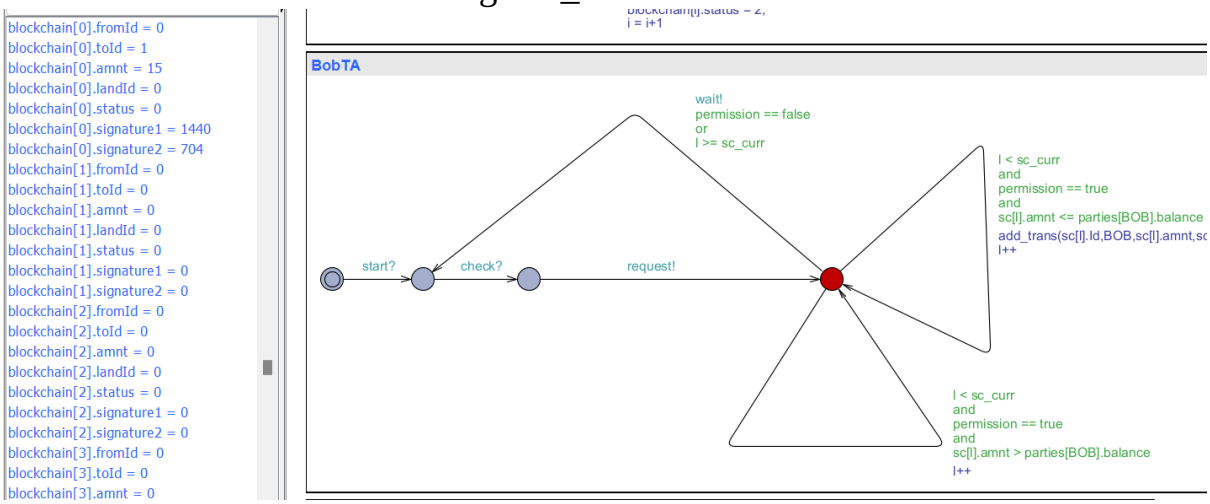
1) Initially the party are initialized using initialize function to assign some values and data to them. All parties are assigned with 50 balance and ALICE is given even lands and BOB with odd lands and ADVERSARY with no lands.



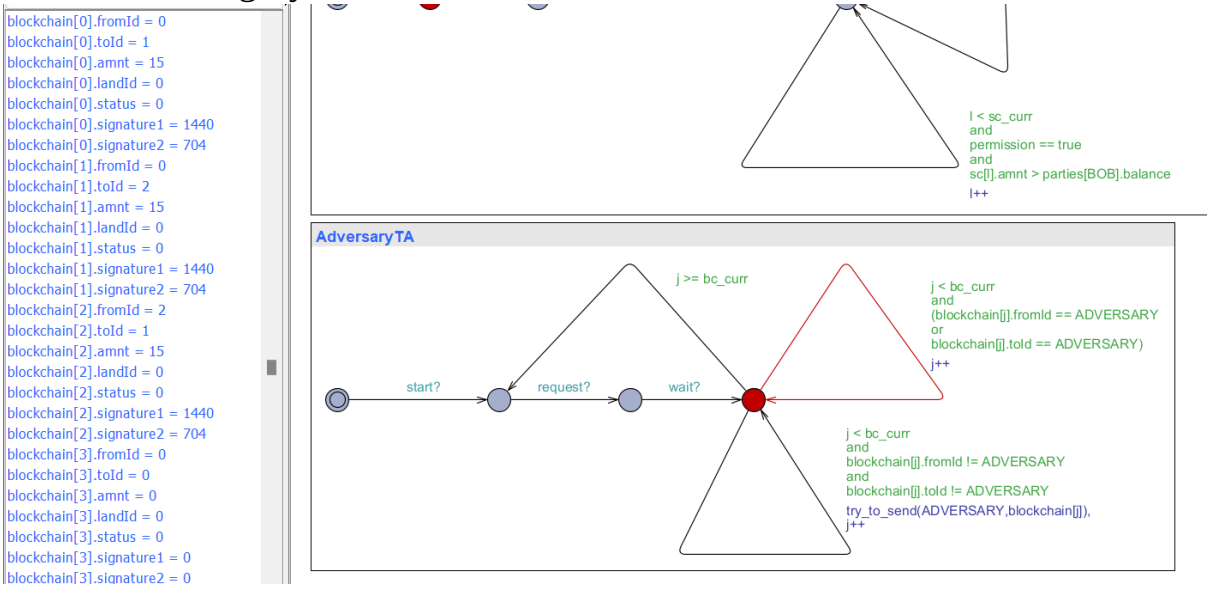
2) Next ALICE adds the smart contract into array of smart contracts using sell_land function.



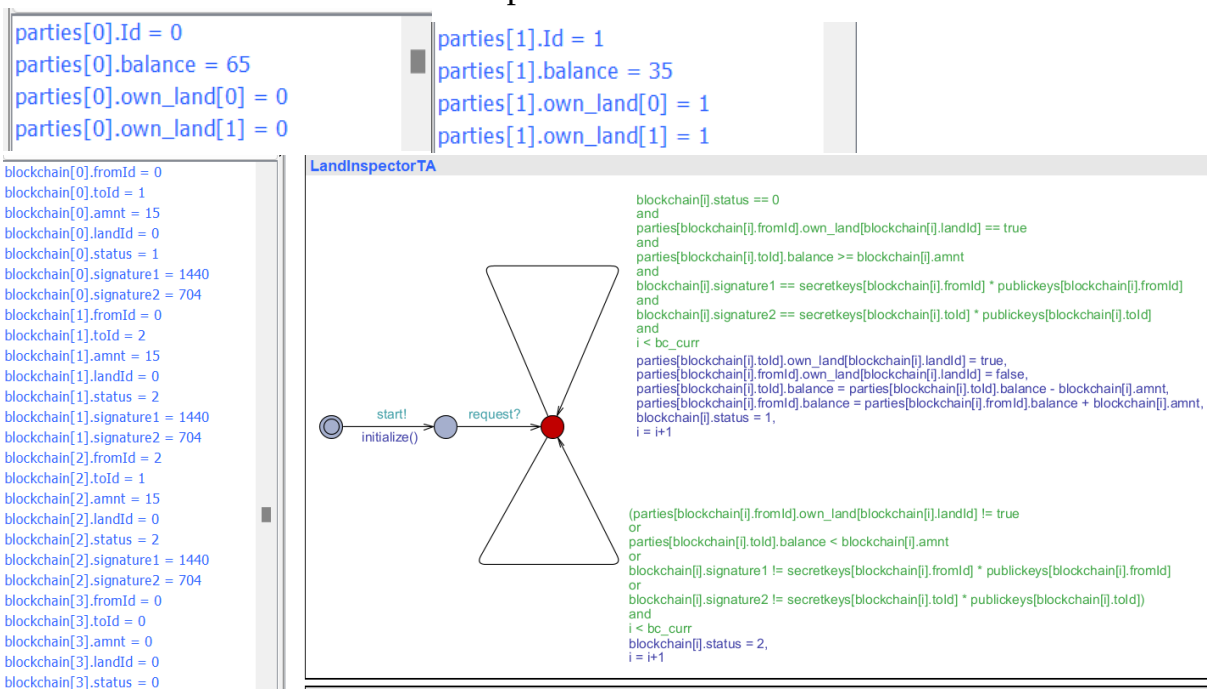
3) Later BOB checks from smart contracts that whether it has enough balance to purchase and its asks permission. After getting permission it adds the transaction into blockchain using add_trans function.



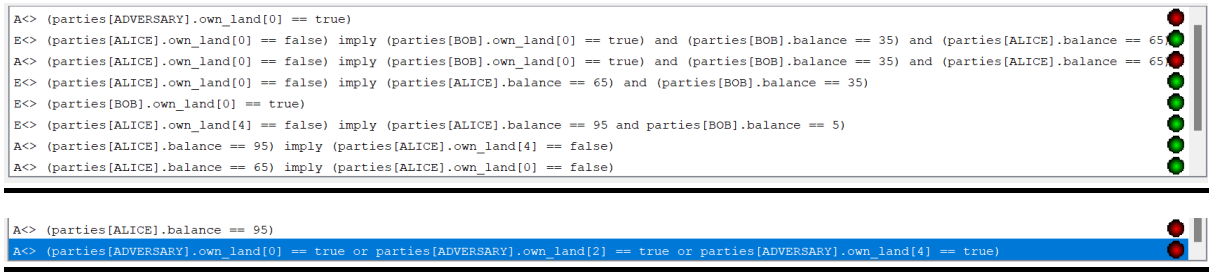
4) Then ADVERSARY tries to add the manipulated transactions into the blockchain using try_to_send function.



5) Then LandInspector does the task of verifying the transactions and correct transactions are SPENT and manipulated are CANCELED.



6.1.7) Verifier Results



1) ***A[] (parties[ADVERSARY].own_land[0] == true)*** - Adversary will not be able to own any lands even introducing malicious transactions. The query should result in false because for "All the paths eventually adversary owns landId 0" is completely false because we made sure that blockchain won't allow adversary transactions and it cancels its transactions.

2) ***E<> (parties[ALICE].own_land[0] == false) imply (parties[BOB].own_land[0] == true) and (parties[BOB].balance == 35) and (parties[ALICE].balance == 65)*** – "There exists a path eventually if ALICE won't own landId 0 imply that BOB owns it and BOB has balance of 35 and ALICE has 65" which is completely true because if alice buys

its land bob must buy it for 15 amount and adversary is eliminated by blockchain.

3) ***A<> (parties[ALICE].own_land[0] == false) imply (parties[BOB].own_land[0] == true) and (parties[BOB].balance == 35) and (parties[ALICE].balance == 65)*** – "All paths eventually if ALICE won't own landId 0 imply that BOB owns it and BOB has balance of 35 and ALICE has 65" which is completely false because not for all paths it is true as the maximum number of transactions are limited and alice can keep on adding its transactions into blockchain which finally occupies maximum limit and it will go into deadlock.

4) ***E<> (parties[ALICE].own_land[0] == false) imply (parties[ALICE].balance == 65) and (parties[BOB].balance == 35)*** – Similar to 2nd query which is also true.

5) ***E<> (parties[BOB].own_land[0] == true)*** – "There exists a path eventually BOB owns landId 0" which is completely true because there exists a path and eventually BOB owns landId 0 because ALICE is ready to buy its land.

6) ***E<> (parties[ALICE].own_land[4] == false) imply (parties[ALICE].balance == 95 and parties[BOB].balance == 5)*** – "There exists a path eventually if ALICE doesn't own landId 4 imply ALICE has a balance of 95 and BOB has 5" which is completely true because as ALICE owns all even lands at first then it sells it land one by one that it owns which are 0,2,4 landIds. Eventually after selling its land 4 its balance must be 50 + 3(15) = 95. And that means BOB balance must be 50 – 3(15) = 5 which means further BOB won't be able to buy any land even though ALICE broadcasts to sell its all even lands into smart contract.

7) ***A<> (parties[ALICE].balance == 95) imply (parties[ALICE].own_land[4] == false)*** – "All paths eventually if ALICE balance is 95 imply that ALICE doesn't own landId 4" which is completely true because for all paths if ALICE balance is 95 means ALICE doesn't own landId 4 and ADVERSARY also doesn't own any land because BOB buys the lands from ALICE.

8) ***A<> (parties[ALICE].balance == 65) imply (parties[ALICE].own_land[0] == false)*** – Similar to 7th query which is also true.

9) ***A<> (parties[ALICE].balance == 95)*** – "All paths eventually ALICE balance is 95" which is completely false because as there is a limit to number of transactions in blockchain the ALICE can keep on adding its transactions into blockchain without giving LandInspector chance to perform its operations through loop as a result the system will reach deadlock after it reaches maximum limit of transactions.

10) ***A<> (parties[ADVERSARY].own_land[0] == true or parties[ADVERSARY].own_land[2] == true or parties[ADVERSARY].own_land[4] == true)*** – "All paths eventually ADVERSARY owns landId 0 or 2 or 4" which is completely false because adversary is not allowed to own land at first and later as well it may confuse the blockchain on introducing malicious transactions but blockchain will cancel all its transactions and in all paths at least it won't even own a single land.

6.2) Case-2

6.2.1) Global Declarations

Similar to that of case-1 but in addition here we take five SELLERs, BUYER and ADVERSARY each.

```
const int LAND_NUM = 100; //Maximum lands allowed
const int TX_NUM = 100; //Maximum transactions allowed
const int SC_NUM = 100; //Maximum contracts allowed
const int PARTIES_NUM = 7; //6 Parties + 1 Adversary
const int INIT_AMNT = 100; //Initial amount provided to all parties

typedef int[0,PARTIES_NUM-1] PartyId;
typedef int[0,LAND_NUM-1] LANDID;
typedef int[0,TX_NUM-1] TXID;
typedef int[0,2] Status;

//Different parties
const PartyId SELLER1 = 0;
const PartyId SELLER2 = 1;
const PartyId SELLER3 = 2;
const PartyId SELLER4 = 3;
const PartyId SELLER5 = 4;
const PartyId BUYER = 5;
const PartyId ADVERSARY = 6;
```

6.2.1) LandInspector

Local Declarations

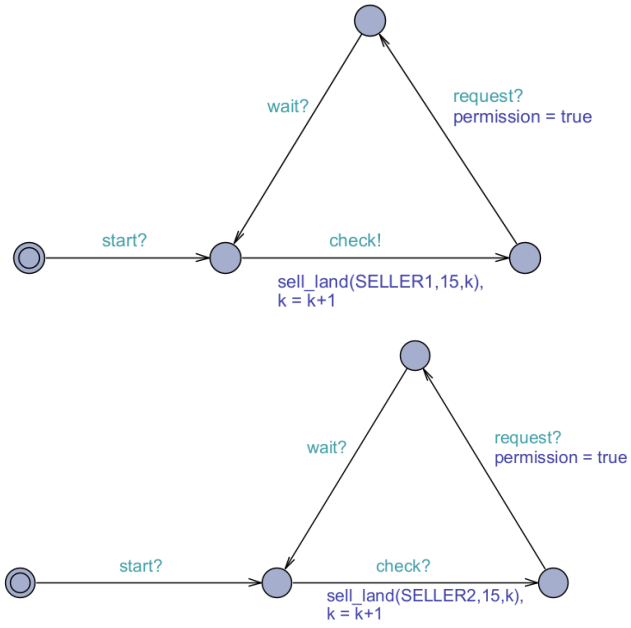
Same declarations and model as that of case-1 except that we involve more seller parties and here each party is assigned to %5 of landIds. Buyer and Adversary are not assigned with any land.

```
// Place local declarations here.
int secretkeys[7] = {90,88,22,34,18,6,26};
int publickeys[7] = {16,8,18,9,4,11,12};
void initialize()
{
    for(i : PartyId)
    {
        parties[i].Id = i;
        parties[i].balance = INIT_AMNT;
        parties[i].secretkey = secretkeys[i];
        parties[i].publickey = publickeys[i];
        for(j : LANDID)
        {
```

```
            if(i == BUYER or i == ADVERSARY)
            {
                parties[i].own_land[j] = false;
            }
            if(j%5 == i)
            {
                parties[i].own_land[j] = true;
            }
        }
    }
}
```

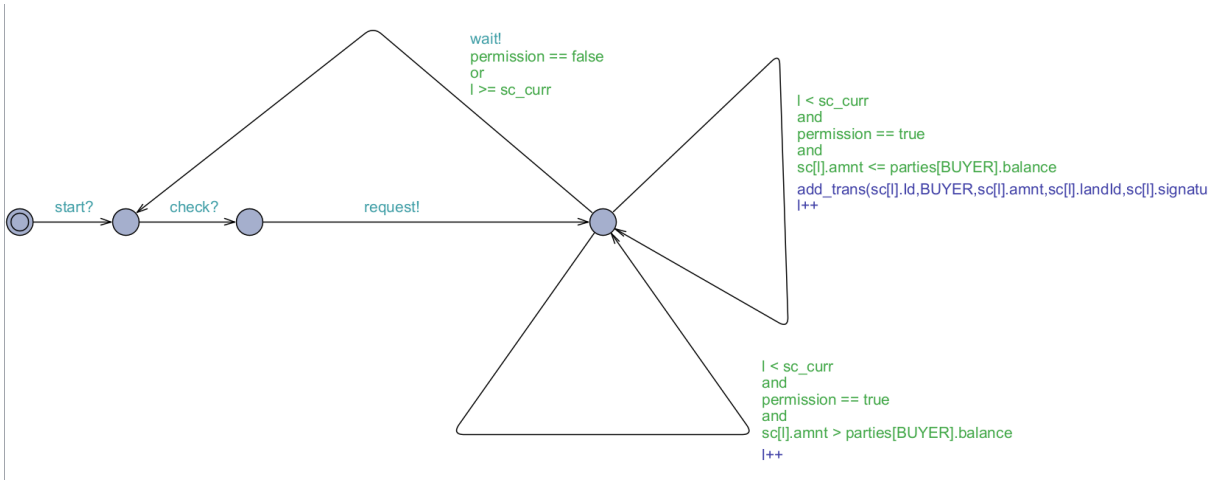
6.2.2) SELLERS

Similar to the kind of ALICE in previous case except that sell_land contains their own party ids for all five sellers.



6.2.3) BUYER

Similar to BOB of previous case except we change BOB to BUYER in add_trans function.

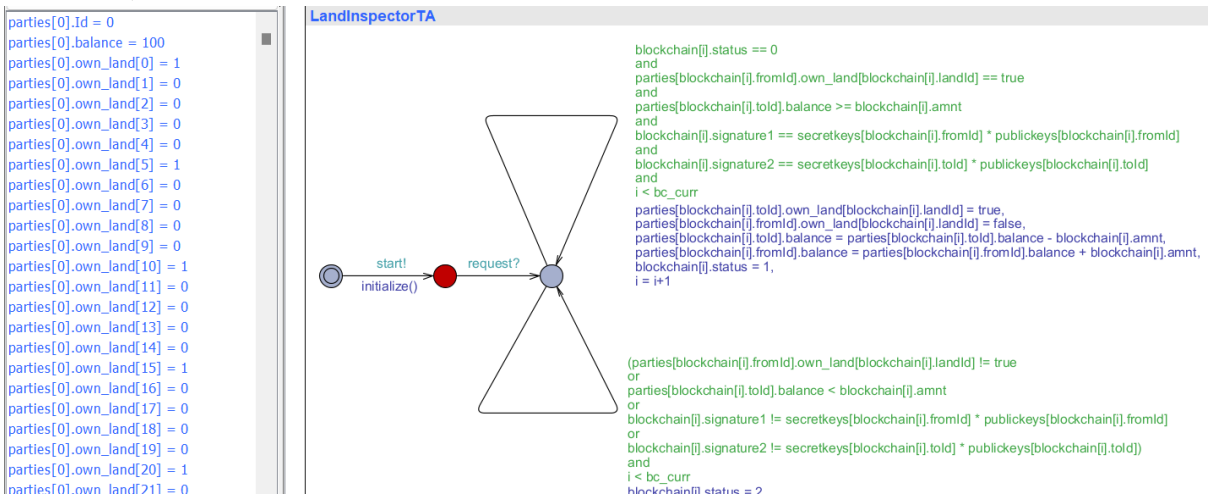


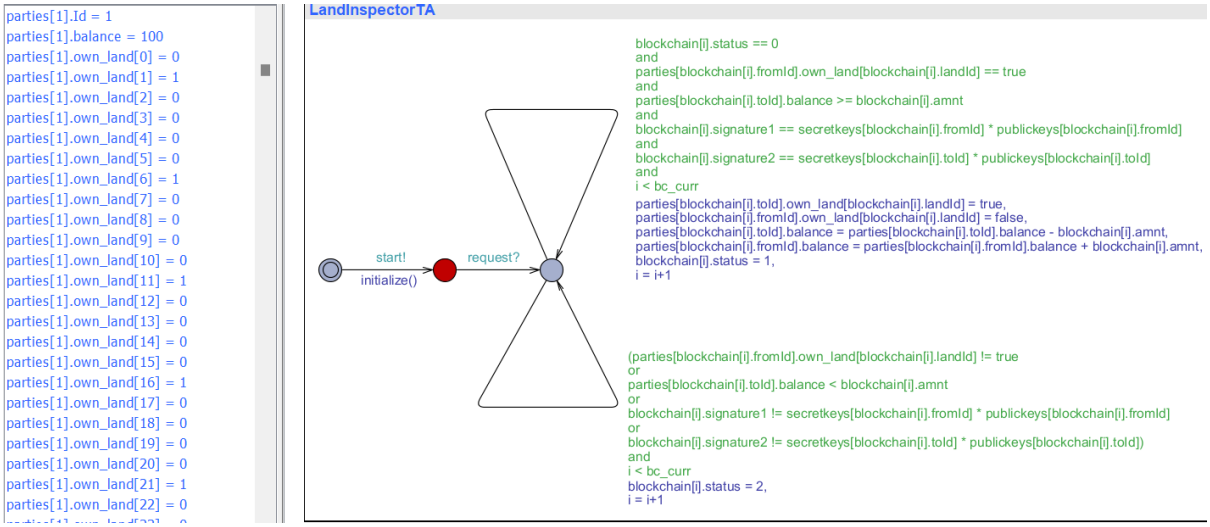
6.2.4) ADVERSARY

Same as that of ADVERSARY in previous case.

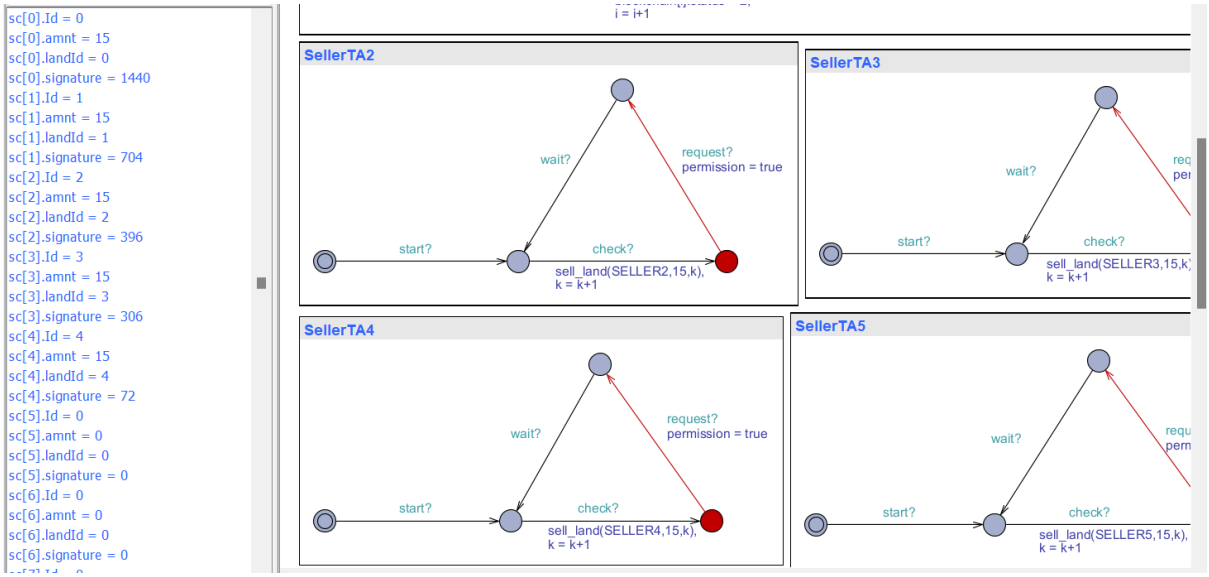
6.2.5) Simulation Results

1) Initially the party are initialized using initialize function to assign some values and data to them. All parties are assigned with 100 balance and each SELLER will get ownership over %5 of landIds (for every five lands they get one land based on remainder of its landId same as partyId) and BUYER, ADVERSARY with no lands.

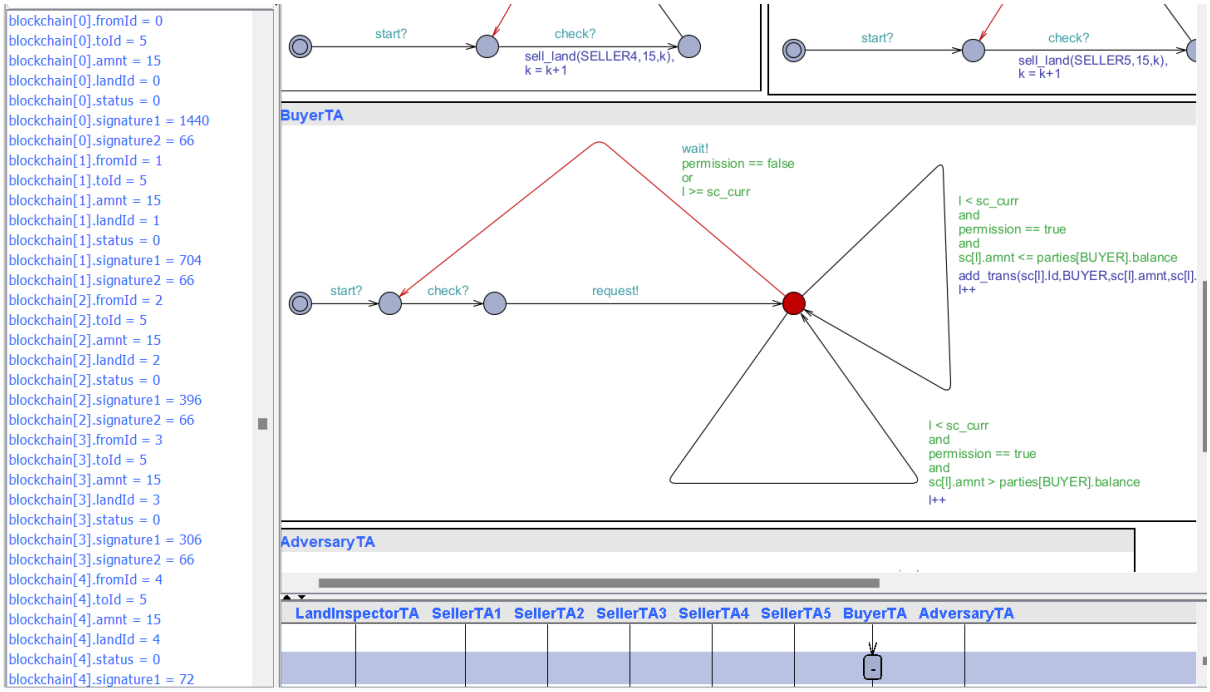




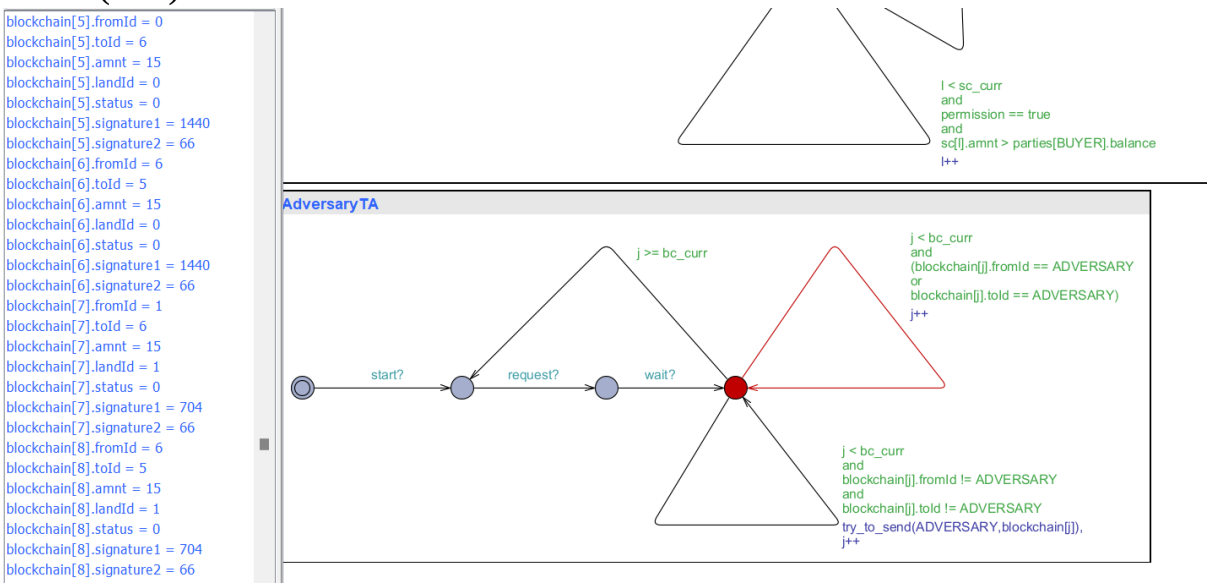
2) Next SELLERs add the smart contract into array of smart contracts using sell_land function.

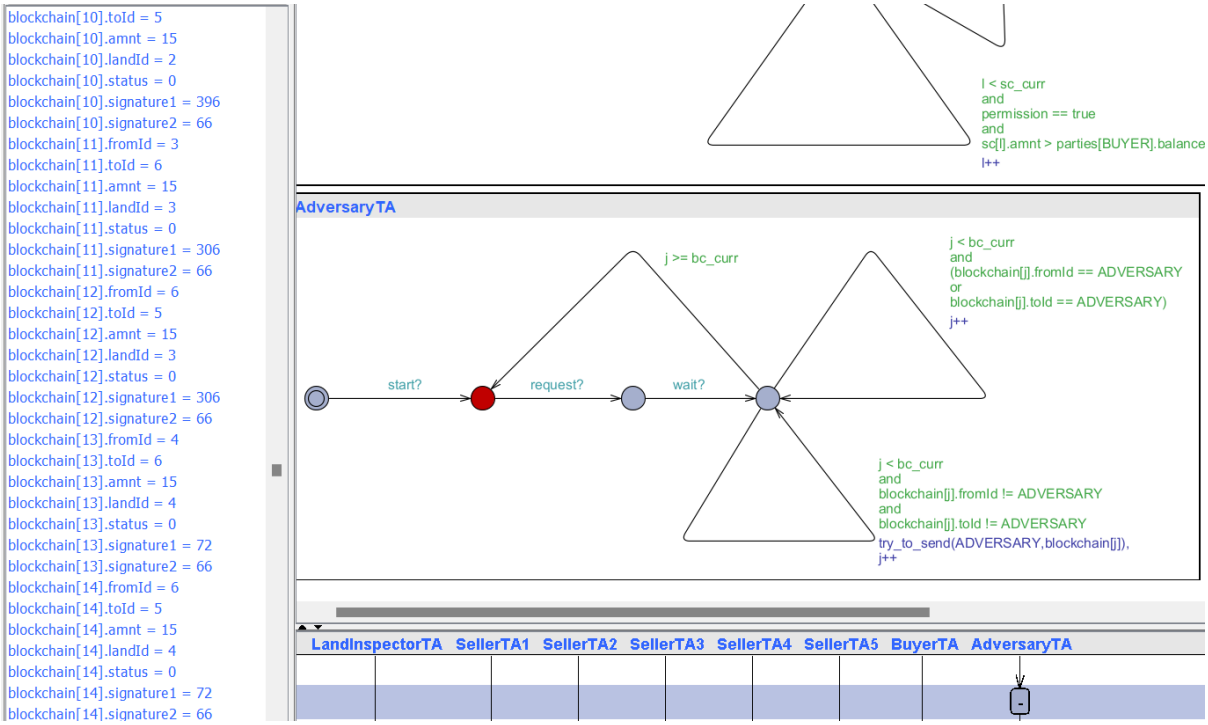


3) Later BUYER checks from smart contracts that whether it has enough balance to purchase and its asks permission. After getting permission it adds the transaction into blockchain using add_trans function.

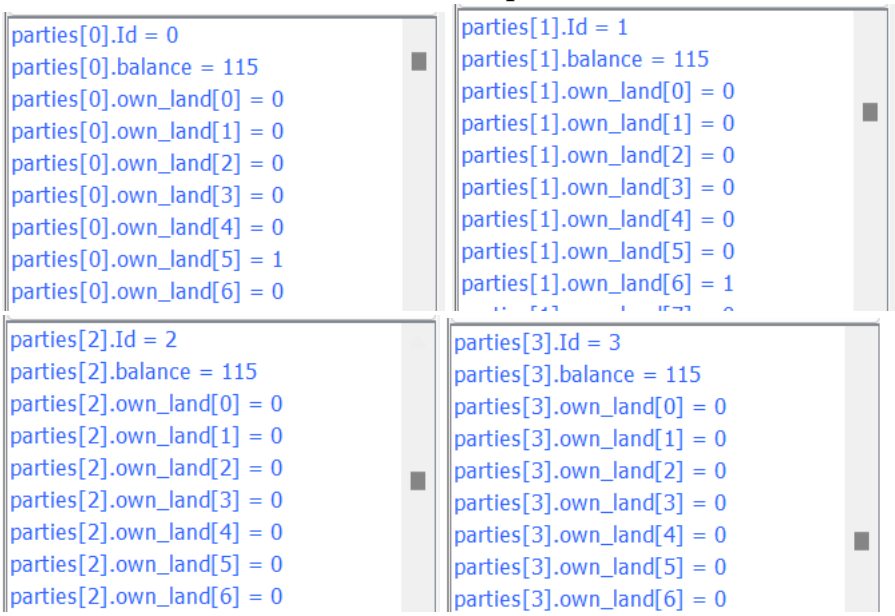


4) Then ADVERSARY tries to add the manipulated transactions into the blockchain using try_to_send function. As we have 5 SELLERs in the model each will add a transaction of their own and each transaction gets manipulated by ADVERSARY that adds two more transactions of each transaction of SELLER. Total after execution of ADVERSARY we will see 5*(1+2) = 15 transactions in the blockchain from 0 to 14 indices.





5) Then LandInspector does the task of verifying the transactions and correct transactions are SPENT and manipulated are CANCELED.



```
parties[4].Id = 4
parties[4].balance = 115
parties[4].own_land[0] = 0
parties[4].own_land[1] = 0
parties[4].own_land[2] = 0
parties[4].own_land[3] = 0
parties[4].own_land[4] = 0
parties[4].own_land[5] = 0
parties[4].own_land[6] = 0
```

```
parties[5].Id = 5
parties[5].balance = 25
parties[5].own_land[0] = 1
parties[5].own_land[1] = 1
parties[5].own_land[2] = 1
parties[5].own_land[3] = 1
parties[5].own_land[4] = 1
parties[5].own_land[5] = 0
parties[5].own_land[6] = 0
```

```
parties[6].Id = 6
parties[6].balance = 100
parties[6].own_land[0] = 0
parties[6].own_land[1] = 0
parties[6].own_land[2] = 0
parties[6].own_land[3] = 0
parties[6].own_land[4] = 0
parties[6].own_land[5] = 0
parties[6].own_land[6] = 0
```

7. Conclusion

Land is a high valued asset that need to have accurate records which identify the current owner and also the all known previous owners' history. We have used blockchain technology in the above system to maintain a secure and decentralized that enables buyers and sellers to deal directly without intermediaries. We have developed a model which is used in between sellers and buyers to sign the property ownership transfer of documents and those also stored in blockchain. Similarly smart contracts and blockchain are maintained as array to structure. Smart contracts are broadcasted to all the parties and every party can check the list of smart contracts proposed by the sellers and they can request for buying the interested lands. In this way the transactions are added into blockchain and even though adversary tries to confuse the blockchain by introducing the malicious transactions but blockchain restricts the transactions as it has access to all secret and public keys and adversary don't know the function used for signature generation but only seller, buyer and blockchain knows them. In this way adversary transactions are cancelled by blockchain and there will be secure and fast transactions between two fair parties. Blockchain technology can be used for record integrity protection, eliminating cost of third parties and brokers, transparency with smart contracts automated land registry and transaction process and accelerated land registry and safer. As the research is continuing in this domain, I think the above work will help in some basic way to automate the real-life model using UPPAAL.

8. References

1) https://uppaal.org/
2) https://en.wikipedia.org/wiki/Bitcoin
3) https://en.wikipedia.org/wiki/Uppaal_Model_Checker
4) Blockchain-based Land Registry Platform (thegatewaydigital.com)
5) What are smart contracts on blockchain? | IBM
6) Why is Blockchain Important and Why Does it Matters? [2022] | Simplilearn
7) Automated modelling: a discussion and review | The Knowledge Engineering Review | Cambridge Core
8) Modeling Bitcoin Contracts by Timed Automata - Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski and Łukasz Mazurek - Cryptology and Data Security Group - University of Warsaw - 2014
9) The UPPAAL Model Checker - Julián Proenza - Systems, Robotics and Vision Group - UIB. SPAIN – 2008