Modeling Bitcoin Contracts by Timed Automata[⋆]

Marcin Andrychowicz, Stefan Dziembowski[⋆⋆], Daniel Malinowski and Łukasz Mazurek

Cryptology and Data Security Group
`www.crypto.edu.pl`
University of Warsaw

Abstract. Bitcoin is a peer-to-peer cryptographic currency system. Since its introduction in 2008, Bitcoin has gained noticeable popularity, mostly due to its following properties: (1) the transaction fees are very low, and (2) it is not controlled by any central authority, which in particular means that nobody can "print" the money to generate inflation. Moreover, the transaction syntax allows to create the so-called *contracts*, where a number of mutually-distrusting parties engage in a protocol to jointly perform some financial task, and the fairness of this process is guaranteed by the properties of Bitcoin. Although the Bitcoin contracts have several potential applications in the digital economy, so far they have not been widely used in real life. This is partly due to the fact that they are cumbersome to create and analyze, and hence risky to use.
In this paper we propose to remedy this problem by using the methods originally developed for the computer-aided analysis for hardware and software systems, in particular those based on the timed automata. More concretely, we propose a framework for modeling the Bitcoin contracts using the timed automata in the UPPAAL model checker. Our method is general and can be used to model several contracts. As a proof-of-concept we use this framework to model some of the Bitcoin contracts from our recent previous work. We then automatically verify their security in UPPAAL, finding (and correcting) some subtle errors that were difficult to spot by the manual analysis. We hope that our work can draw the attention of the researchers working on formal modeling to the problem of the Bitcoin contract verification, and spark off more research on this topic.

1 Introduction

Bitcoin is a digital currency system introduced in 2008 by an anonymous developer using a pseudonym "Satoshi Nakamoto" [23]. Despite of its mysterious origins, Bitcoin became the first cryptographic currency that got widely adopted — as of January 2014 the Bitcoin capitalization is over €7 bln. The enormous success of Bitcoin was also widely covered by the media (see e.g. [16,5,25,21,22]) and even attracted the attention of several governing bodies and legislatures, including the US Senate [21]. Bitcoin owes its popularity mostly to the fact that it has no central authority, the transaction fees are

[⋆] This work was supported by the WELCOME/2010-4/2 grant founded within the framework of the EU Innovative Economy (National Cohesion Strategy) Operational Programme.
[⋆⋆] On leave from the *Sapienza* University of Rome.

very low, and the amount of coins in the circulation is restricted, which in particular means that nobody can "print" money to generate inflation. The financial transactions between the participants are published on a public ledger maintained jointly by the users of the system.

One of the very interesting, but slightly less known, features of the Bitcoin is the fact that it allows for more complicated "transactions" than the simple money transfers between the participants: very informally, in Bitcoin it is possible to "deposit" some amount of money in such a way that it can be claimed only under certain conditions. These conditions are written in the form of the *Bitcoin scripts* and in particular may involve some timing constrains. This property allows to create the so-called *contracts* [27], where a number of mutually-distrusting parties engage in a Bitcoin-based protocol to jointly perform some task. The security of the protocol is guaranteed purely by the properties of the Bitcoin, and no additional trust assumptions are needed. This Bitcoin feature can have several applications in the digital economy, like creating the assurance contracts, the escrow and dispute mediation, the rapid micropayments [27], the multi-party lotteries [8]. It can also be used to add some extra properties to Bitcoin, like the certification of the users [17], or creating the secure "mixers" whose goal is to enhance the anonymity of the transactions [19]. Their potential has even been noticed by the media (see, e.g., a recent enthusiastic article on the *CNN Money* [22]).

In our opinion, one of the obstacles that may prevent this feature from being widely used by the Bitcoin community is the fact that the contracts are tricky to write and understand. This may actually be the reason why, despite of so many potential applications, they have not been widely used in real life. As experienced by ourselves [6,7,8], developing such contracts is hard for the following reasons. Firstly, it's easy to make subtle mistakes in the scripts. Secondly, the protocols that involve several parties and the timing constraints are naturally hard to analyze by hand. Since mistakes in the contracts can be exploited by the malicious parties for their own financial gain, it is natural that users are currently reluctant to use this feature of Bitcoin.

In this paper we propose an approach that can help designing secure Bitcoin contracts. Our idea is to use the methods originally developed for the computer-aided analysis for hardware and software systems, in particular the timed automata [1,2]. They seem to be the right tool for this purpose due to the fact that the protocols used in the Bitcoin contracts typically have a finite number of states and depend on the notion of time. This time-dependence is actually two-fold, as (1) it takes some time for the Bitcoin transactions to be confirmed (1 hour, say), and (2) the Bitcoin transactions can come with a "time lock" which specifies the time when a transaction becomes valid.

Our contribution We propose a framework for modeling the Bitcoin contracts using timed automata in the UPPAAL model checker [9,18] (this is described in Sec. 2). Our method is general and can be used to model a wide class of contracts. As a proof-of-concept, in Sec. 3 we use this framework to model two Bitcoin contracts from our previous work [8,6]. This is done manually, but our method is quite generic and can potentially be automatized. In particular, most of the code in our implementation does not depend on the protocol being verified, but describes the properties of Bitcoin system. To model a new contract it is enough to specify the transactions used in the contract, the knowledge of the parties at the beginning of the protocol and the protocol followed by the parties. We then automatically verify the security of our contracts in UPPAAL (in Sec. 3.1). The UPPAAL code for the contracts modeled and verified by us is available at the web page `http://crypto.edu.pl/uppaal-btc.zip`.

Future work We hope that our work can draw the attention of the researchers working on formal modeling to the problem of the Bitcoin contracts verification, and spark off more research on this topic. What seems especially interesting is to try to fully automatize this process. One attractive option is to think of the following workflow: (1) a designer of a Bitcoin contract describes it in UPPAAL (or, possibly, in some extension of it), (2) he verifies the security of this idealized description using UPPAAL, and (3) if the idealized description verifies correctly, then he uses the system to "compile" it into a real Bitcoin implementation that can be deployed in the wild. Another option would be to construct a special tool for designing the Bitcoin contracts, that would produce two outputs: (a) a code in the UPPAAL language (for verification) and (b) a real-life Bitcoin implementation.

Of course, in both cases one would need to formally show the soundness of this process (in particular: that the "compiled" code maintains the properties of the idealized description). Hence, this project would probably require both non-trivial theoretical and engineering work.

Preliminaries Timed automata were introduced by Alur and Dill [1,2]. There exist other model checkers based on this theory, like Kronos [31] and Times [4]. It would be interesting to try to implement our ideas also in them. Other formal models that involve the notion of the real time include the timed Petri nets [10], the timed CSP [26], the timed process algebras [30,24], and the timed propositional temporal logic [3]. One can try to model the Bitcoin contracts also using these formalisms. For the lack of space, a short introduction to UPPAAL was moved to Appendix D. The reader may also consult [9,18] for more information on this system.

We assume reader's familiarity with the public-key cryptography, in particular with the signature schemes (an introduction to this concept can be found e.g. in [20,12]). We will frequently denote the key pairs using the capital letters (e.g. A), and refer to the private key and the public key of A by: $A.sk$ and $A.pk$, respectively. We will also use the following convention: if $A = (A.sk, A.pk)$ then $\mathsf{sig}_A(m)$ denotes a signature on a message m computed with $A.sk$ and $\mathsf{ver}_A(m, \sigma)$ denotes the result $(\mathrm{true}$ or $\mathrm{false})$ of the verification of a signature σ on message m with respect to the public key $A.pk$. We will use the "Ƀ" symbol to denote the Bitcoin currency unit.

1.1 A short description of Bitcoin

Since we want the exposition to be self-contained, we start with a short description of Bitcoin, focusing only on the most relevant parts. For the lack of space we do not describe how the coins are created, how the transaction fees are charged, and how the Bitcoin "ledger" is maintained. A more detailed description of Bitcoin is available on the *Bitcoin wiki* site [11]. The reader may also consult the original Nakamoto's paper [23].

Introduction In general one of the main challenges when designing a digital currency is the potential double spending: if coins are just strings of bits then the owner of a coin can spend it multiple times. Clearly this risk could be avoided if the users have access to a trusted ledger with the list of all the transactions. In this case a transaction would be considered valid only if it is posted on the board. For example suppose the transactions are of a form: "user X transfers to user Y the money that he got in some previous transaction T_p", signed by the user X. In this case each user can verify if money from transaction T_p has not been already spent by X. The main difficulty in designing the fully-distributed peer-to-peer currency systems is to devise a system where the users jointly maintain the ledger in such a way that it cannot be manipulated by an adversary and it is publicly-accessible.

In Bitcoin this problem is solved by a cryptographic tool called *proofs-of-work* [15]. We will not go into the details of how this is done, since it is not relevant to this work. Let us only say that the system works securely as long as no adversary controls more computing power than the combined computing power of all the other participants of the protocol[1]. The Bitcoin participants that contribute their computing power to the system are called the *miners*. Bitcoin contains a system of incentives to become a miner. For the lack of space we do not describe it here.

Technically, the ledger is implemented as a chain of blocks, hence it is also called a "block chain". When a transaction is posted on the block chain, it can take some time before it appears on it, and even some more time before the user can be sure that this transaction will not be cancelled. However, it is safe to assume that there exists an upper bound on this waiting time (1-2 hours, say). We will denote this time by `MAX_LATENCY`.

As already highlighted in the introduction, the format of the Bitcoin transactions is in fact quite complex. Since it is of a special interest for us, we describe it now in more detail. The Bitcoin currency system consists of *addresses* and *transactions* between them. An address is simply a public key pk[2]. Normally every such key has a corresponding private key sk known only to one user, which is an *owner* of this address. The private key is used for signing the transactions, and the public key is used for verifying the signatures. Each user of the system needs to know at least one private key of some address, but this is simple to achieve, since the pairs (sk, pk) can be easily generated offline.

Simplified version We first describe a simplified version of the system and then show how to extend it to obtain the description of the real Bitcoin. Let $A = (A.sk, A.pk)$ be a key pair. In our simplified view a transaction describing the fact that an amount v (called the *value* of a transaction) is transferred from an address $A.pk$ to an address $B.pk$ has the following form $T_x = (y, B.pk, v, \mathsf{sig}_A(y, B.pk, v))$, where y is an index of a previous transaction T_y. We say that $B.pk$ is the recipient of T_x, and that the transaction T_y

[1] It is currently estimated [25] that the combined computing power of the Bitcoin participants is around 64 exaFLOPS, which exceeds by factor over 200 the total computing power of world's top 500 supercomputers, hence the cost of purchasing the equipment that would be needed to break this system is huge.

[2] Technically an address is a *cryptographic hash* of pk. In our informal description we decided to assume that it is simply pk. This is done only to keep the exposition as simple as possible, as it improves the readability of the transaction scripts later in the paper.

is an *input* of the transaction T_x, or that it is *redeemed* by this transaction (or redeemed by the address $B.pk$). More precisely, the meaning of T_x is that the amount v of money transferred to $A.pk$ in transaction T_y is transferred further to $B.pk$. The transaction is valid only if (1) $A.pk$ was a recipient of the transaction T_y, (2) the value of T_y was at least v (the difference between the value of T_y and v is called the *transaction fee*), (3) the transaction T_y has not been redeemed earlier, and (4) the signature of A is correct. Clearly all of these conditions can be verified publicly.

The first important generalization of this simplified system is that a transaction can have several "inputs" meaning that it can accumulate money from several past transactions $T_{y_1}, \ldots, T_{y_\ell}$. Let $A_1, \ldots, A_\ell$ be the respective key pairs of the recipients of those transactions. Then a multiple-input transaction has the following form: $T_x = (y_1, \ldots, y_\ell, B.pk, v, \mathsf{sig}_{A_1}(y_1, B.pk, v), \ldots, \mathsf{sig}_{A_\ell}(y_\ell, B.pk, v))$, and the result of it is that $B.pk$ gets the amount v, provided it is at most equal to the sum of the values of transactions $T_{y_1}, \ldots, T_{y_\ell}$. This happens only if *none* of these transactions has been redeemed before, and *all* the signatures are valid.

Moreover, each transaction can have a *time lock* t that tells at what time in the future the transaction becomes valid. The lock-time t can refer either to a measure called the "block index" or to the real physical time. In this paper we only consider the latter type of time-locks. In this case we have $T_x = (y_1, \ldots, y_\ell, B.pk, v, t, \mathsf{sig}_{A_1}(y_1, B.pk, v, t), \ldots, \mathsf{sig}_{A_\ell}(y_\ell, B.pk, v, t))$. Such a transaction becomes valid only if time t is reached and if none of the transactions $T_{y_1}, \ldots, T_{y_\ell}$ has been redeemed by that time (otherwise it is discarded). Each transaction can also have several outputs, which is a way to divide money between several users and to divide transactions with large value into smaller portions. We ignore this fact in our description since we will not use it in our protocols.

More detailed version The real Bitcoin system is significantly more sophisticated than what is described above. First of all, there are some syntactic differences, the most important being that each transaction T_x is identified not by its index, but by its hash $H(T_x)$. Hence, from now on we will assume that $x = H(T_x)$.

The main difference is, however, that in the real Bitcoin the users have much more flexibility in defining the condition on how the transaction T_x can be redeemed. Consider for a moment the simplest transactions where there is just one input and no time-locks. Recall that in the simplified system described above, in order to redeem a transaction, its recipient $A.pk$ had to produce another transaction T_x signed with his private key $A.sk$. In the real Bitcoin this is generalized as follows: each transaction T_y comes with a description of a function (*output-script*) π_y whose output is Boolean. The transaction T_x redeeming the transaction T_y is valid if π_y evaluates to true on input T_x. Of course, one example of π_y is a function that treats T_x as a pair (a message m_x, a signature σ_x), and checks if σ_x is a valid signature on m_x with respect to the public key $A.pk$. However, much more general functions π_y are possible. Going further into details, a transaction looks as follows: $T_x = (y, \pi_x, v, \sigma_x)$, where $[T_x] = (y, \pi_x, v)$ is called the *body*[3] of T_x and σ_x is a "witness" that is used to make the script π_y evaluate to

[3] In the original Bitcoin documentation this is called "simplified T_x". Following our earlier work [8,6,7] we chosen to rename it to "body" since we find the original terminology slightly misleading.

true on T_x (in the simplest case σ_x is a signature on $[T_x]$). The scripts are written in the Bitcoin scripting language [28], which is stack-based and similar to the Forth programming language. It is on purpose not Turing-complete (there are no loops in it), since the scripts need to evaluate in (short) finite time. It provides basic arithmetical operations on numbers, operations on stack, if-then-else statements and some cryptographic functions like calculating hash function or verifying a signature.

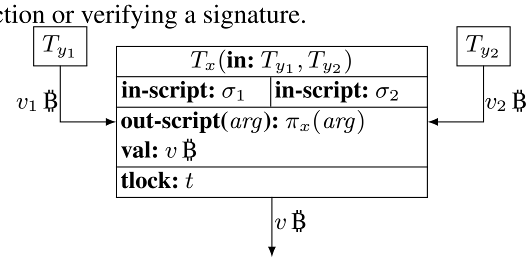

T_{y_1}
T_{y_2}
T_x(**in:** T_{y_1}, T_{y_2})
in-script: σ_1
in-script: σ_2
out-script(arg): $\pi_x(arg)$
val: v ₿
tlock: t
v_1 ₿
v_2 ₿
v ₿

Fig. 1. A graphical representation of a transaction $T_x = (y_1, y_2, \pi_x, v, t, \sigma_1, \sigma_2)$.

The generalization to the multiple-input transactions with time-locks is straightforward: a transaction has a form: $T_x = (y_1, \ldots, y_\ell, \pi_x, v, t, \sigma_1, \ldots, \sigma_\ell)$, where the body $[T_x]$ is equal to $(y_1, \ldots, y_\ell, \pi_x, v, t)$, and it is valid if (1) time t is reached, (2) *every* $\pi_i([T_x], \sigma_i)$ evaluates to true, where each π_i is the output script of the transaction T_{y_i}, and (3) none of these transactions has been redeemed before. We will present the transactions as boxes. The redeeming of transactions will be indicated with arrows (the arrows will be labelled with the transaction values). An example of a graphical representation of a transaction is depicted in Fig. 1.

The transactions where the input script is a signature, and the output script is a verification algorithm are the most common type of transactions. We will call them *standard transactions*. Currently some miners accept only such transactions, due to the fact that writing more advanced scripts is hard and error-prone, and anyway the vast majority of users does not use such advanced features of Bitcoin. Fortunately, there exist other miners that do accept the non-standard (also called *strange*) transactions, one example being a big mining pool[4] called *Eligius* (that mines a new block on average once per hour). We also believe that in the future accepting the general transaction will become standard, maybe at a cost of a slightly increased fee. Actually, popularizing the Bitcoin contracts, by making them safer to use, is one of the goals of this work.

2 Modeling the Bitcoin

To reason formally about the security of the contracts we need to describe the attack model that corresponds to the Bitcoin system. The model used in [8,6] was inspired by the approach used in the complexity-based cryptography. This way of modeling protocols, although very powerful, is not well-suited for the automatic verification of cryptographic protocols. In this section we present the approach used in this paper, based on timed-automata, using the syntax of the UPPAAL model checker.

In our model each party executing the protocol is modeled as a timed automaton with a structure assigned to it that describes the party's knowledge. States in the automaton describe which part of the protocol the party is performing. The transitions in the automaton contain conditions, which have to be satisfied for a transition to be taken and actions, which are performed whenever a transition is taken.

[4] Mining pools are coalitions of miners that perform their work jointly and share the profits.

The communication between the parties may be modeled in a number of various ways. It is possible to use synchronization on channels offered by UPPAAL and shared variables representing data being sent between the parties. In all protocols verified by us, the only messages exchanged by the parties were signatures. We decided to model the communication indirectly using shared variables — each party keeps the set of known signatures, and whenever a sender wants to send a signature, he simply adds it to the recipient's set.

The central decision that needs to be made is how to model the knowledge of the honest parties and the adversary. Our representation of knowledge is symbolic and based on Dolev-Yao model [13], and hence we assume that the cryptographic primitives are perfectly secure. In our case it means, for example, that it is not possible to forge a signature without the knowledge of the corresponding private key, and this knowledge can be modeled in a "binary" way: either an adversary knows the key, or not. The hash functions will be modeled as random oracles, which in particular implies that they are collision-resilient and hard to invert. We also assume that there exists a secure and authenticated channel between the parties, which can be easily achieved using the public key cryptography. Moreover, we assume that there is a fixed set of variables denoting the private/public pairs of Bitcoin keys. A Bitcoin protocol can also involve secret strings known only to some parties (like e.g. a string s in the commitment protocol in Appendix B), we assume that there is a fixed set of variables denoting such strings. For each private key and each secret string there is a subset of parties, which know them, but all public keys and hashes of all secret strings are known to all the parties (if this is not the case then they can be broadcast by parties knowing them).

A block chain is modelled using a shared variable (denoted `bc`) keeping the status of all transactions and a special automaton, which is responsible for maintaining the state of `bc` (e.g. confirming transactions).

In the following sections we describe our model in more details.

2.1 The keys, the secret strings, and the signatures

We assume that the number of the key pairs in the protocol is known in advance and constant. Therefore, key pairs will be simply referred by consecutive natural numbers (type **Key** is defined as an integer from a given range). Secret strings are modelled in the same way. As already mentioned we assume that all public keys and hashes of all secrets are known to all parties.

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

Fig. 2. The signatures type.

Moreover, we need to model the signatures over transactions (one reason is that they are exchanged by the parties in some protocols). They are modelled by structures containing a transaction being signed, the key used to compute a signature and an `input_nonce`, which is related to the issue of transaction malleability and described in Appendix A.

2.2 The transactions

We assume that all transactions that can be created by the honest parties during the execution of the protocol comes from a set, which is known in advance and of size T. Additionally, the adversary can create his own transactions. As explained later (see Sec. 2.4 below) we can upper-bound the number of adversarial transactions by T. Hence the total upper bound on the number of transactions is $2T$.

For simplicity we refer the transactions using fixed identifiers instead of their hashes. We can do this because we know all the transactions, which can be broadcast in advance (compare Sec.2.4 and Appendix A for further discussion). A single-input and single-output transaction is a variable of a record type **Tx** defined in Fig. 3. 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 ₿). The timelock field indicates the time lock of the transaction, and the timelock_passed is a boolean field indicating whether the timelock has passed.

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

Fig. 3. The transactions type

The status field is of a type **Status** that contains following values: UNSENT (indicating that 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. Otherwise, it refers to a hard-coded function which implements the output script of this transaction (see Sec. 2.5 for more details).

The inputs scripts are modelled only indirectly (the fields reveals_secret and secret_revealed). More precisely, we only keep information about which secrets are included in the input script (see e.g. the CS protocol in Fig. 9, transaction *Open*).

The above structure can be easily extended to handle multiple inputs and outputs.

2.3 The parties

The parties are modelled by timed automata describing protocols they follow. States in the automata describe which part of the protocol the party is performing. The transitions in the automata

```
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;
```

Fig. 4. The parties type.

contain conditions, which have to
be satisfied for a transition to be
taken and actions, which are performed whenever a transition is taken. An example of such automaton appears in Fig. 7. and is described in more details in Appendix D.2. The adversary is modelled by a special automaton described in Sec. 2.4.

Moreover, we need to model the knowledge of the parties (both the honest users and the adversary) in order to be able to decide whether they can perform a specific action in a particular situation (e.g. compute the input script for a given transaction). Therefore for each party, we define a structure describing its knowledge. More technically: this knowledge is modelled by a record type **`Party`** defined in Fig. 4.

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` contains these signatures.

2.4 The adversary

The real-life Bitcoin adversary can create an arbitrary number of transactions with arbitrary output scripts, so it is clear that we need to somehow limit his possibilities, so that the space of possible states is finite and of a reasonable size. We show that without loss of generality we can consider only scenarios in which an adversary sends to the block chain transactions only from a finite set depending only on the protocol.

The knowledge of an adversary is modeled in the similar way to honest parties, but we do not specify the protocol that he follows. Instead, we use a generic automaton, which (almost) does not depend on the protocol being verified and allows 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 is able to create its input script.

We observe that the transactions made by the adversary can influence the execution of the protocol only in two ways: either (1) the transaction is identical to the transaction from the protocol being verified or (2) the transaction redeems one of the transactions from the protocol. The reason for above is that honest parties only look for transactions of the specific form (as in the protocol being executed), so the only thing an adversary can do to influence this process is to create a transaction, which looks like one of these transactions or redeem one of these. Notice that we do not have to consider transactions with multiple inputs redeeming more than one of the protocol's transactions, because there is always an interleaving in which the adversary achieves the same result using a number of transactions with single inputs. The output scripts in the transactions of type (2) do not matter, so we may assume that an adversary always sends them to one particular key known only to him.

Therefore, without loss of generality we consider only the transactions, which appear in the protocol being verified or transactions redeeming one of these transactions. Hence, the total number of transactions, which are modeled in our system is twice as big as the number of transactions in the original protocol.

The adversary is then a party, that can send an arbitrary transaction from this set if only he is able to do so (e.g. he is able to evaluate the input script and the transaction's input is confirmed, but not spent). If the only actions of the honest parties is 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. This is depicted in Fig. 5 on page 10 (for a moment ignore the left loop).

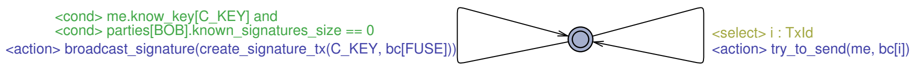

<cond> me.know_key[C_KEY] and
<cond> parties[BOB].known_signatures_size == 0
<action> broadcast_signature(create_signature_tx(C_KEY, bc[FUSE]))
<select> i : TxId
<action> try_to_send(me, bc[i])

Fig. 5. The automaton for the Adversary

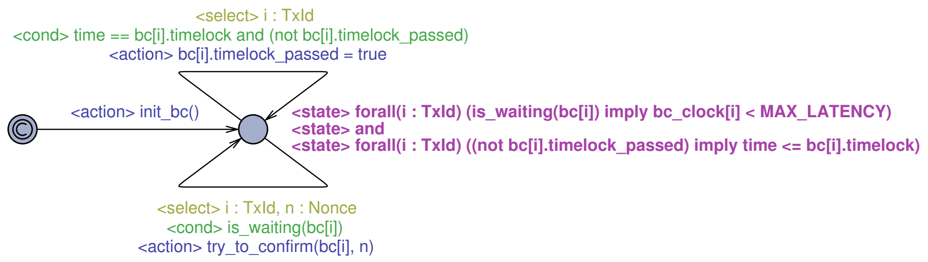
In some protocols the parties besides of posting the transactions on the block chain, also exchange messages with each other. This can be exploited by the adversary, and hence we need to take care of this in our model. This is done by adding more actions in the `Adversary` automaton. In our framework, this is done manually. For example in the protocol that we analyze in Sec. 3 Alice sends a signature on the $Fuse$ transaction (cf. Step 3, Fig. 9, Appendix B). This is reflected by the left loop in the `Adversary` automaton in Fig. 5, which should be read as follows: if the adversary is able to create a signature on the $Fuse$ transaction and Bob did not receive it yet, then he can send it to Bob.

Of course, our protocols need to be analyzed always "from the point of view of an honest Alice" (assuming Bob is controlled by the adversary) and "from the point of view of an honest Bob" (assuming Alice is controlled by the adversary). Therefore, for each party we choose whether to use the automaton describing the protocol executed by the parties or the already mentioned special automaton for an adversary.

2.5 The block chain and the notion of time

In Bitcoin whenever a party wants to post a transaction on the block chain she broadcasts it over a peer-to-peer network. In our model this is captured as follows. We model the block chain as a shared structure denoted `bc` containing the information about the status of all the transactions and a timed automaton denoted `BlockChainAgent` (see Fig. 6), which is responsible for maintaining the state of `bc`. One of the duties of `BlockChainAgent` is ensuring that the transactions which were broadcast are confirmed within appropriate time frames.

In order 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. In particular, 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. Therefore, the exact condition on when the party p can create the appropriate input script, has to be extracted manually from the description of the protocol. If all these tests succeed, then the function communicates the fact of broadcasting the transaction using the shared structure bc.



<select> i : TxId
<cond> time == bc[i].timelock and (not bc[i].timelock_passed)
<action> bc[i].timelock_passed = true
<action> init_bc()
<state> forall(i : TxId) (is_waiting(bc[i]) imply bc_clock[i] < MAX_LATENCY)
<state> and
<state> forall(i : TxId) ((not bc[i].timelock_passed) imply time <= bc[i].timelock)
<select> i : TxId, n : Nonce
<cond> is_waiting(bc[i])
<action> try_to_confirm(bc[i], n)

Fig. 6. The `BlockChainAgent` automaton

Once a transaction `t` has been broadcast, the `BlockChainAgent` automaton attempts to include it in the block chain (lower loop in Fig. 6). The `BlockChainAgent` automaton also enforces that every transaction gets included into the block chain in less than `MAX_LATENCY` time, which is a constant that is defined in the system. This is done by the invariant on the right state in Fig. 6 that guarantees that every transaction is waiting for confirmation less than `MAX_LATENCY`.

Eavesdropping on the network The other issue with the block chain is that the peers in the network can see transactions before they are confirmed. Therefore if a transaction t contains (e.g. in its input script) a secret string x then an adversary can learn the value of x before t is confirmed and for example use it to create a different transaction redeeming the input of t (a similar scenario is possible for a two-party lottery protocol from [8], which is only secure in a "private channel model"). To take such possibilities into account, broadcasting a transaction results in disclosure of the secret string x, what in our model corresponds to setting appropriate knowledge flags for all parties.

Malleability of transactions `BlockChainAgent` automaton is also responsible for choosing the nonces, which imitate the attacks involving the malleability of transactions. This is described in details in Appendix A.

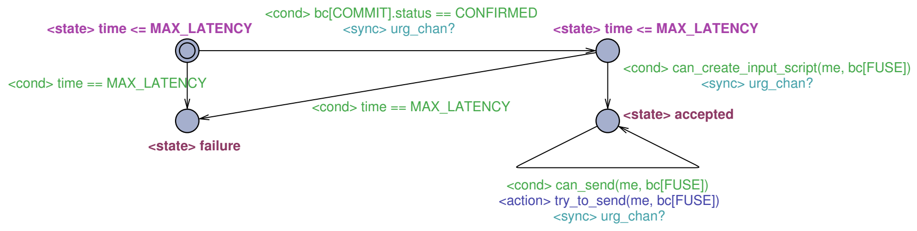
3 Modeling the Bitcoin-based timed commitment scheme from [8]

In this section we describe the "contract-dependent" part of our model. Our method of modeling and verifying Bitcoin contracts as timed automata is generic and can be applied to a large class of Bitcoin contracts (and even possibly automatized as described in the paragraph "Future work" on page 3). However, it is easier to describe it using a concrete example. As a proof-of-concept we constructed the automata corresponding to a very simple contract called the "Bitcoin-based timed commitment scheme" from [8]. For the lack of space we only sketch informally what the protocol is supposed to do. In the protocol one of the parties (called Alice) commits herself to a secret string s. A key difference between this protocol and classic commitment schemes is that Alice is forced to open the commitment (i.e. reveal the string s) until some agreed moment of time (denoted `PROT_TIMELOCK`) or pay 1 Ƀ to Bob. The full description can be found in Appendix B. Although the verification of correctness is quite straightforward in this case, we would like to stress that our method is applicable to more complicated contracts, like the NewSCS protocol from [8] (see Section 3.2), for which the correctness is much less obvious.

3.1 The results of the verification

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

In UPPAAL it is done by selecting an automaton following the protocol or the malicious automaton for an adversary described in Sec. 2.4 for each of the parties. We started with verification of the security from the point of view of honest Bob. To this end we used an honest automaton for Bob (see Fig. 7) and an adversary automaton described before for Alice (see Fig. 5).



<cond> bc[COMMIT].status == CONFIRMED
<sync> urg_chan?
<state> time <= MAX_LATENCY
<state> time <= MAX_LATENCY
<cond> time == MAX_LATENCY
<cond> can_create_input_script(me, bc[FUSE])
<sync> urg_chan?
<cond> time == MAX_LATENCY
<state> accepted
<state> failure
<cond> can_send(me, bc[FUSE])
<action> try_to_send(me, bc[FUSE])
<sync> urg_chan?

Fig. 7. The automaton for an honest Bob in timed-commitment scheme

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, informally means: "after time `PROT_TIMELOCK` + `MAX_LATENCY` one of the following cases takes place: either (a) Bob earned 1 Ƀ, or (b) Bob knows the committed secret, or (c) Bob rejected the commitment in the commitment phase". This is exactly the security property claimed in [8], and hence the verification confirmed our belief that the protocol is secure. We verified the security from the point of view of Alice in the similar way.

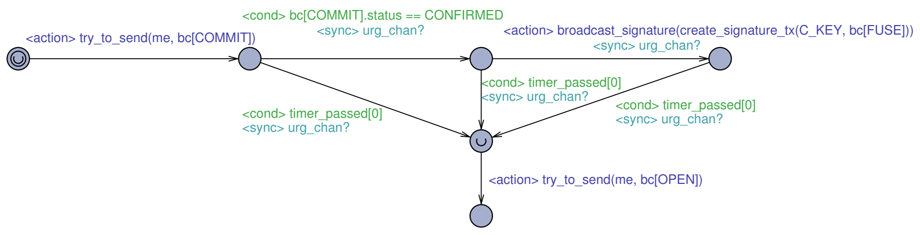

<action> try_to_send(me, bc[COMMIT])
<cond> bc[COMMIT].status == CONFIRMED
<sync> urg_chan?
<action> broadcast_signature(create_signature_tx(C_KEY, bc[FUSE]))
<sync> urg_chan?
<cond> timer_passed[0]
<sync> urg_chan?
<cond> timer_passed[0]
<sync> urg_chan?
<cond> timer_passed[0]
<sync> urg_chan?
<action> try_to_send(me, bc[OPEN])

Fig. 8. The automaton for an honest Alice in timed-commitment scheme

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

As a test we also run the verification procedure on the following two statements:

```
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_TIME-LOCK Alice holds 1 Ƀ (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 hold if both parties follow the protocol (i.e. when honest automata are used for both parties). Moreover, UPPAAL provides diagnostic traces, which are interleavings of events leading to the violation of the property being tested. They allow to immediately figure out, why the given property is violated and turned out to be extremely helpful in debugging the automata.

3.2 The NewSCS protocol from [8]

We also modeled and verified the Simultaneous Commitment Scheme (NewSCS) protocol from [8], which is relatively complicated as it contains 18 transactions. To understand it fully the reader should probably look in the [8], but as reference we included the description of these contracts in Appendix C. Informally speaking, the NewSCS scheme is a protocol that allows two parties, Alice and Bob, to simultaneously commit to their secrets (s_A and s_B, respectively) in such a way that each commitment is valid only if the other commitment was done correctly. Using UPPAAL we automatically verified the following three conditions, which are exactly the security statements claimed in [6]:

- After the execution of the protocol by two honest parties, they both know both secrets and hold the same amount of coins as at the beginning of the protocol, which in UPPAAL syntax was formalized as:

```
A[] (time >=  PROT_TIMELOCK+MAX_LATENCY) imply
   (parties[ALICE].know_secret[SB_SEC] and parties[BOB].know_secret[SA_SEC]
     and hold_bitcoins(parties[ALICE]) == 2
     and hold_bitcoins(parties[BOB]) == 2)
```

  (here SA_SEC and SB_SEC denote the secrets of Alice and Bob, respectively, and 2 is the value of the deposit).
- An honest Bob cannot lose any coins as a result of the protocol, no matter how the dishonest Alice behaves:

```
2) A[] (time >= PROT_TIMELOCK) imply hold_bitcoins(parties[BOB]) >= 2
```

– If an honest Bob did not learn Alice's secret then he gained Alice's deposit as a result of the execution.

```
3) A[] ((time >= PROT_TIMELOCK+2*MAX_LATENCY) imply
         ((parties[ALICE].know_secret[SB_SEC]
             and !parties[BOB].know_secret[SA_SEC])
         imply hold_bitcoins(parties[BOB]) >= 3))
```

The analogous guarantees hold for Alice, when Bob is malicious. The verification of each of the mentioned properties took less than a minute on a dual-core 2.4 GHz notebook. We confirmed that the protocol NewSCS is correct, but there are some implementation details, which are easy to miss and our first implementation (as an automaton) turned out to contain a bug, which was immediately found due to the verification process and diagnostic traces provided by UPPAAL. We describe this in more detail in Appendix C.1. Moreover UPPAAL turned out to be very helpful in determining the exact time threshold for the time locks, for example we confirmed that the time at which the parties should abort the protocol claimed in [7] ($t - 3\texttt{MAX\_LATENCY}$) is strict.

These experiments confirmed that the computer aided verification and in particular UPPAAL provides a very good tool for verifying Bitcoin contracts, especially since it is rather difficult to assess the correctness of Bitcoin contracts by hand, due to the distributed nature of the block chain and a huge number of possible interleavings. Therefore, we hope that our paper would encourage designers of complex Bitcoin contracts to make use of computer aided verification for checking the correctness of their constructions.

References

1. R. Alur and D. L. Dill. Automata for modeling real-time systems. In *ICALP'90*.
2. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 1994.
3. R. Alur and T. A. Henzinger. A really temporal logic. *Journal of the ACM*, 1994.
4. T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. TIMES - a tool for modelling and implementation of embedded systems. TACAS '02.
5. Marc Andreessen. Why Bitcoin Matters, Jan 2013. The New York Times, dealbook.nytimes.com/2014/01/21/why-bitcoin-matters, accessed on 26.01.2014.
6. M. Andrychowicz, S. Dziembowski, D. Malinowski, and Ł. Mazurek. Fair two-party computations via the bitcoin deposits. Cryptology ePrint Archive, Report 2013/837, 2013. `http://eprint.iacr.org/2013/837`, accepted to the 1st Workshop on Bitcoin Research.
7. M. Andrychowicz, S. Dziembowski, D. Malinowski, and Ł. Mazurek. How to deal with malleability of Bitcoin transactions. *ArXiv e-prints*, December 2013.
8. M. Andrychowicz, S. Dziembowski, D. Malinowski, and Ł. Mazurek. Secure Multiparty Computations on Bitcoin. Cryptology ePrint Archive, 2013. `http://eprint.iacr.org/2013/784`, accepted to the 35th IEEE Symposium on Security and Privacy (Oakland) 2014.
9. Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on uppaal 4.0, 2006.
10. Bernard Berthomieu and Michel Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE Trans. Softw. Eng.*, 17(3):259–273, March 1991.
11. Bitcoin. Wiki. en.bitcoin.it/wiki/.
12. H. Delfs and H. Knebl. *Introduction to Cryptography: Principles and Applications*. Information Security and Cryptography. Springer, 2007.

13. D. Dolev and A. C. Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 1983.
14. Danny Dolev, Cynthia Dwork, and Moni Naor. Nonmalleable cryptography. *SIAM Journal on Computing*, 30(2):391–437, 2000.
15. C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *CRYPTO*, 1992.
16. The Economist. The Economist explains: How does Bitcoin work?, Apr 2013. www.economist.com/blogs/economist-explains/2013/04/economist-explains-how-does-bitcoin-work, accessed on 26.01.2014.
17. G. Ateniese et al. Certified bitcoins. Cryptology ePrint Archive, Report 2014/076.
18. J. Bengtsson et al. UPPAAL - a tool suite for automatic verification of real-time systems. In *Hybrid Systems*, volume 1066 of *Lecture Notes in Computer Science*, 1995.
19. S. Barber et al. Bitter to better - how to make bitcoin a better currency. FC'12.
20. J. Katz and Y. Lindell. *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series)*. Chapman & Hall/CRC, 2007.
21. T. B. Lee. Here's how Bitcoin charmed Washington. www.washingtonpost.com/blogs/the-switch/wp/2013/11/21/heres-how-bitcoin-charmed-washington, accessed on 26.01.2014.
22. David Z. Morris. Bitcoin is not just digital currency. It's Napster for finance, Jan 2014. CNN Money, finance.fortune.cnn.com/2014/01/21/bitcoin-platform, accessed on 26.01.2014.
23. S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.
24. Xavier Nicollin and Joseph Sifakis. The algebra of timed processes, atp: Theory and application. *Inf. Comput.*, 114(1):131–178, 1994.
25. R. Cohen. Global Bitcoin Computing Power Now 256 Times Faster Than Top 500 Supercomputers, Combined! Forbes, www.forbes.com/sites/reuvencohen/2013/11/28/global-bitcoin-computing-power-now-256-times-faster-than-top-500-supercomputers-combined/.
26. G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. *Theor. Comput. Sci.*, 58(1-3):249–261, June 1988.
27. Bitcoin wiki. Contracts. en.bitcoin.it/wiki/Contracts, Accessed on 26.01.2014.
28. Bitcoin wiki. Script. en.bitcoin.it/wiki/Script, Accessed on 26.01.2014.
29. Bitcoin wiki. Transaction malleability. en.bitcoin.it/wiki/Transaction_Malleability, Accessed on 26.01.2014.
30. Wang Yi. CCS + time = an interleaving model for real time systems. In *Automata, Languages and Programming*, volume 510 of *Lecture Notes in Computer Science*. 1991.
31. Sergio Yovine. Kronos: a verification tool for real-time systems. *Journal on Software Tools for Technology Transfer*, 1, October 1997.

A Malleability of transactions

Let us now describe the reason why we introduced the variables of a type **Nonce** in the transaction and the signatures. The reason is that they are used to model the fact that the transactions in Bitcoin can be slightly modified without changing its functionality. Malleability is a very general concept introduced in cryptography in a seminal paper of Dolev et al. [14]. It is a (usually) undesired property of the cryptographic schemes that very informally means that an adversary, after seeing an output of a scheme, can produce another output that is in some non-trivial way "related" to the original output.[5] As it turns out, malleability of transaction is a feature of Bitcoin, which poses a serious

[5] For example, the one time pad encryption scheme (see, e.g., [20]) defined as: $\mathrm{Enc}(K, M) := K \oplus M$ and $\mathrm{Dec}(K, C) := K \oplus C$ (where "$\oplus$" denotes the coordinate-wise xor) *is* malleable since by negating every bit in C one obtains a ciphertext C' of a message M' that is equal

risk for almost all contracts using time locks. We now briefly describe this problem. More detailed descriptions can be found in [7,29].

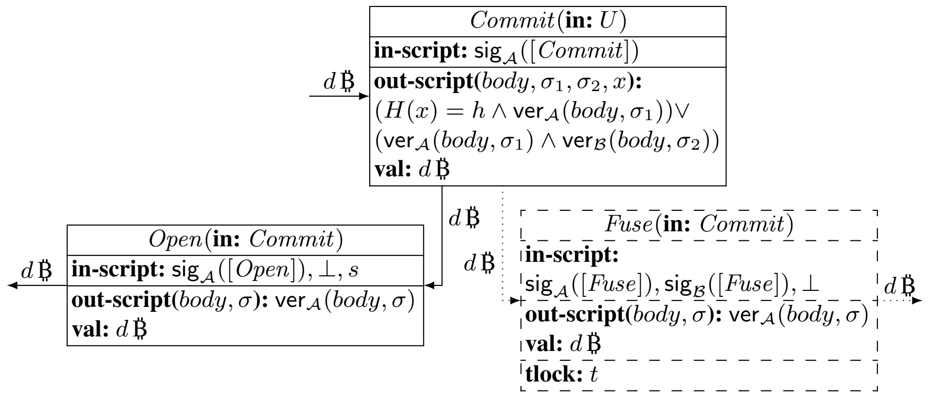
Bitcoin transactions are malleable in the following way: given a valid transaction t, an adversary is able to create a functionally equivalent and valid transaction t' which has a different hash, even if he does not know the secret key used to produce t. Both transactions have the same input transactions, the same output script and differ only in the hash. Therefore, the following scenario is possible. A transaction t is sent to the block chain, which technically means that it is broadcast over the network. An adversary can therefore see t and create and broadcast t'. If he is lucky then eventually t' becomes included in the block chain, instead of t. It means that we can not assume the knowledge of the hash of the transaction before it is confirmed. It poses a serious problem for most of the protocols using time locks, because such protocols often need to sign a transaction s which redeems t, *before* t is broadcast. Such transaction s contains a hash of t and in case when t' is included in the block chain instead of t, the transaction s becomes invalid.

In our model the transactions are addressed by fixed identification numbers instead of hashes, so a special technique should be applied to take the malleability of transactions into account. To this end we extend the transaction structure with a "malleability nonce", which contains an integer from a fixed and small set **`Nonce`** (in most cases a set **`int`**$[0, 1]$ of size 2 is enough). Each transaction, which has not been sent to the block chain has the `nonce` field equal to 0, which indicates that the transaction was not modified. Whenever a transaction is being confirmed, its malleability nonce is set to a random value. Moreover, the structure describing the signature on the transaction t which redeems s (see Sec. 2.1) contains the assumed malleability nonce of s (set when the transaction t was being signed) and is considered valid only if this nonce matches the real nonce of the transaction s. Therefore, if the transaction s was confirmed after the signature structure was created, the signature may or may not become invalidated, what catches the issue of malleability in real Bitcoin network.

The above solution with "malleability nonces" makes the assumption that whenever s is modified, then only the transaction t becomes invalid. In reality it would also influence another transaction u which redeems t and so on. However, in all Bitcoin protocols we are aware of, this is not an issue as the signatures are computed at most "one step forward". It is also not difficult to extend the model to handle "malleability nonce of the second (or arbitrary) level" if necessary.

to a "negated" M. Note that this works even if one does not know K. Since the one-time pad is perfectly secure, thus the non-malleability is not implied by the standard security of an encryption scheme.

B Bitcoin-based timed commitment scheme


$Commit(\textbf{in: } U)$
in-script: $\mathsf{sig}_{\mathcal{A}}([Commit])$
out-script$(body, \sigma_1, \sigma_2, x)$**:**
$(H(x) = h \wedge \mathsf{ver}_{\mathcal{A}}(body, \sigma_1)) \vee$
$(\mathsf{ver}_{\mathcal{A}}(body, \sigma_1) \wedge \mathsf{ver}_{\mathcal{B}}(body, \sigma_2))$
val: d ฿
d ฿
$Open(\textbf{in: } Commit)$
in-script: $\mathsf{sig}_{\mathcal{A}}([Open]), \perp, s$
out-script$(body, \sigma)$**:** $\mathsf{ver}_{\mathcal{A}}(body, \sigma)$
val: d ฿
d ฿
d ฿
d ฿
$Fuse(\textbf{in: } Commit)$
in-script:
$\mathsf{sig}_{\mathcal{A}}([Fuse]), \mathsf{sig}_{\mathcal{B}}([Fuse]), \perp$
out-script$(body, \sigma)$**:** $\mathsf{ver}_{\mathcal{A}}(body, \sigma)$
val: d ฿
tlock: t
d ฿

Pre-condition:

1. The key pair of Alice is $\mathcal{A}$ and the key pair of Bob is $\mathcal{B}$.
2. Alice already sent to Bob the value $h = H(s)$, where s is the value she wants to commit to.
3. The block chain contains a confirmed and unredeemed transaction U, which can be redeemed with key $\mathcal{A}$ of value d ฿.

The $\mathsf{CS.Commit}(Alice, Bob, d, t, s)$ **phase**

1. Alice sends to the block chain the transactions $Commit$.
2. If within time `MAX_LATENCY` the $Commit$ transaction does not appear on the block chain, or if it looks incorrect (e.g. it differs in the h value) then Bob aborts.
3. Alice creates the body of the transaction $Fuse$, signs it and sends the signature to Bob.
4. Bob accepts the commitment if he receives the appropriate signature from Alice.

The $\mathsf{CS.Open}(Alice, Bob, d, t, s)$ **phase**

5. Alice sends to the block chain the transactions $Open$, what reveals the secret string s.
6. If within time t the transaction $Open$ does not appear on the block chain then Bob sends to the block chain the transaction $Fuse$ and earns d ฿.

Fig. 9. The $\mathsf{CS}(Alice, Bob, d, t, s)$ protocol. The scripts' arguments, which are omitted are denoted by $\perp$.

A *commitment scheme* is one of the basic cryptographic primitives (see, e.g., [12]). It is executed between two parties: Alice and Bob. In the first phase (called, the *commitment phase*) Alice *commits* herself to some secret value s. That means that she sends to Bob a value $c = f(s)$ for some agreed randomized function f. In the second phase (called the *opening phase*) Alice sends to Bob s plus some extra information.

From the commitment scheme we require two properties — *hiding* and *binding*. These properties will be satisfied if we use f as $f(s) = H(s||r)$ where H is a cryptographic hash function and r is a random, fix length padding (if we model H as a random oracle). In the Bitcoin-based timed commitment scheme this construction is used as a building block (with often used in Bitcoin function SHA-256 as a hash function). In its description we assume, that the padding was already added and simply use H instead of f.

One of the problems with the classical commitment scheme is that there is no way to force Alice to open the commitment and hence reveal the secret. To remedy this, in [8] we proposed to use the Bitcoin system to punish financially Alice in case she does not open the commitment. This is done as follows. Assume that Alice wants to commit herself to some secret s. To do it, she creates and sends to the block chain a Bitcoin transaction ($Commit$) that contains a commitment $f(s)$ and has some value d ₿ (this money is called the "deposit"). This transaction can be spent in one of two ways: either by revealing the secret (this is an expected action of Alice, she will have to reveal the secret to get her money back), or the signatures of both Alice and Bob (by this Bob can punish Alice). After the transaction is included in the block chain, Alice creates the $Fuse$ transaction — it spends the $Commit$ transaction and sends the money to Bob. Alice sets a time lock t (which forces her to open the commitment by time t) adds her signature and sends it to Bob. Bob accepts the commitment if the transactions are correct (e.g. they have proper values, time lock, and $Fuse$ has a good signature) and adds his signature to the $Fuse$ transaction. Otherwise he quits. Alice should open the commitment (i.e. send to the block chain the transaction $Open$ spending the $Commit$ transaction; it will contain the secret) before the time lock ends to get back her money. If she does not do that, then Bob sends the $Fuse$ transaction to get agreed amount of bitcoins. The graph of transactions and a detailed description of the protocol is presented in Fig. 9. We require that this scheme, besides of being binding and hiding, has the following extra properties:

1. honest Alice will never lose her money and she will always open the commitment,
2. if both Alice and Bob are honest, then Bob will accept the commitment,
3. if honest Bob accepts the commitment, then he will learn the secret or gain agreed value of bitcoins.

We verified these conditions using UPPAAL and the model described in this paper (see Sec. 3.1).

C Simultaneous Commitment scheme from [7]

Pre-conditions:

1. A holds the key pair A and B holds the key pair B.
2. A knows the secret s_A, B knows the secret s_B, both players know the hashes $h_{s_A} = H(s_A)$ and $h_{s_B} = H(s_B)$.
3. There are four unredeemed transactions T_1^A, T_2^A and T_1^B, T_2^B, which can be redeemed with the keys A and B respectively, each having the value of d Ƀ.

The $\mathsf{NewSCS.Commit}_R(\mathsf{A}, \mathsf{B}, d, t, s_A, s_B)$ **phase:**

1. A draws a random string r_A and B draws a random string r_B.
2. The parties execute $\mathsf{CS.Commit}(\mathsf{A}, \mathsf{B}, d, t, r_A)$ and $\mathsf{CS.Commit}(\mathsf{B}, \mathsf{A}, d, t, r_B)$ using T_1^A and T_1^B respectively. The former execution will be denoted CS^A and the latter CS^B. Recall that the parties quit the whole NewSCS protocol if they detect the misbehavior of the other party during one of the $\mathsf{CS.Commit}$ executions.

The $\mathsf{NewSCS.Commit}_S(\mathsf{A}, \mathsf{B}, d, t, s_A, s_B)$ **phase:**

3. Both players construct the body of the transaction *Commit* using T_2^A and T_2^B as inputs.
4. A signs the transaction *Commit* and sends the signature to B.
5. B signs the transaction *Commit* and broadcasts it.
6. Both parties wait until the transaction *Commit* is confirmed.
7. If the transaction *Commit* does not appear on the blockchain until the time $t - 3\mathsf{max}_{\mathsf{BB}}$, where $\mathsf{max}_{\mathsf{BB}}$ is the maximal possible delay between broadcasting the transaction and including it in the blockchain, then A immediately redeems the transaction T_2^A and after T_2^A is redeemed she opens her CS^A commitment and quits the protocol. Analogously, if A did not send her signature to B until the time $t - 3\mathsf{max}_{\mathsf{BB}}$, then B opens his CS^B commitment and quits the protocol.

The $\mathsf{NewSCS.Open}(\mathsf{A}, \mathsf{B}, d, t, s_A, s_B)$ **phase:**

8. A and B broadcast the transactions *Open*^{A} and *Open*^{B} respectively, what reveals the secrets s_A and s_B.
9. After the transactions *Open*^{A} and *Open*^{B} are confirmed, A and B open their CS commitments.
10. If A did not broadcast *Open*^{A} until time t, then depending on whether she opened her commitment CS^A or not, B broadcasts *Fuse*^{A} or $\mathsf{CS}^A.\mathit{Fuse}$ to get extra d Ƀ (in addition to $2d$ Ƀ already claimed from *Open*^{B} and $\mathsf{CS}^B.\mathit{Open}$). Similarly, if B misbehaved then A broadcasts *Fuse*^{B} or $\mathsf{CS}^B.\mathit{Fuse}$ to get her extra d Ƀ.

Fig. 10. The description of the NewSCS protocol.

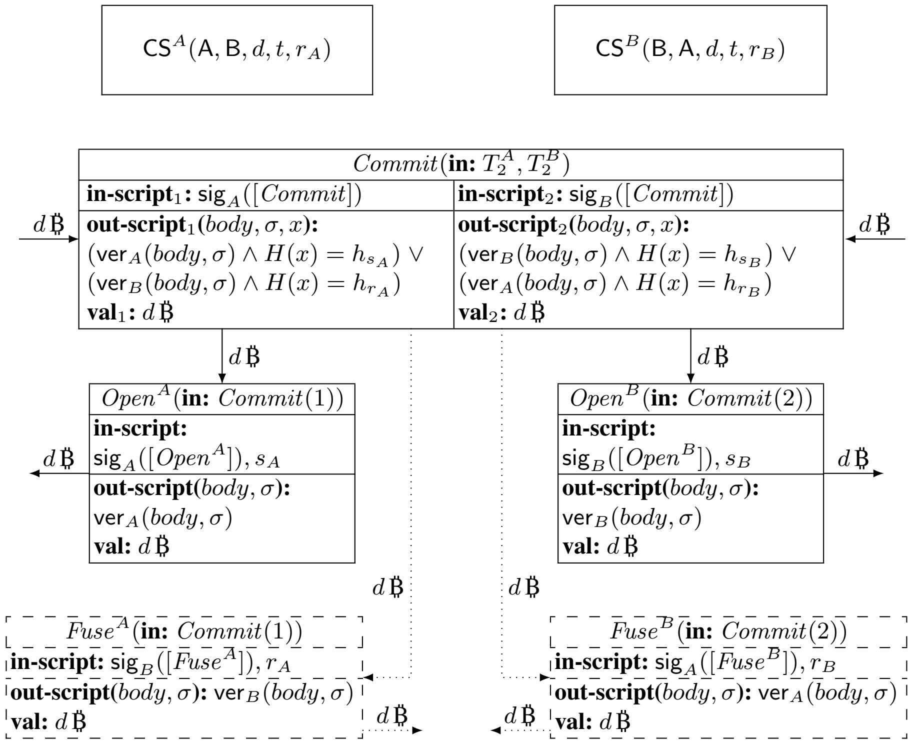

$\mathsf{CS}^A(\mathsf{A},\mathsf{B},d,t,r_A)$
$\mathsf{CS}^B(\mathsf{B},\mathsf{A},d,t,r_B)$
$Commit$(**in:** T_2^A, T_2^B)
in-script$_1$: $\mathsf{sig}_A([Commit])$
out-script$_1$($body,\sigma,x$): $(\mathsf{ver}_A(body,\sigma) \wedge H(x)=h_{s_A}) \vee (\mathsf{ver}_B(body,\sigma) \wedge H(x)=h_{r_A})$
val$_1$: d ฿
in-script$_2$: $\mathsf{sig}_B([Commit])$
out-script$_2$($body,\sigma,x$): $(\mathsf{ver}_B(body,\sigma) \wedge H(x)=h_{s_B}) \vee (\mathsf{ver}_A(body,\sigma) \wedge H(x)=h_{r_B})$
val$_2$: d ฿
d ฿
$Open^A$(**in:** $Commit(1)$)
in-script: $\mathsf{sig}_A([Open^A]), s_A$
out-script($body,\sigma$): $\mathsf{ver}_A(body,\sigma)$
val: d ฿
$Open^B$(**in:** $Commit(2)$)
in-script: $\mathsf{sig}_B([Open^B]), s_B$
out-script($body,\sigma$): $\mathsf{ver}_B(body,\sigma)$
val: d ฿
$Fuse^A$(**in:** $Commit(1)$)
in-script: $\mathsf{sig}_B([Fuse^A]), r_A$
out-script($body,\sigma$): $\mathsf{ver}_B(body,\sigma)$
val: d ฿
$Fuse^B$(**in:** $Commit(2)$)
in-script: $\mathsf{sig}_A([Fuse^B]), r_B$
out-script($body,\sigma$): $\mathsf{ver}_A(body,\sigma)$
val: d ฿

Fig. 11. The graph of transactions for the NewSCS protocol. Two boxes labeled with $\mathsf{CS}(\ldots)$ denote the transactions broadcast in the appropriate execution of the Bitcoin-based timed commitment scheme. h_x denotes the value $H(x)$, but it is used in the output scripts to stress that the value of the hash is directly included in the transaction (instead of value of x and an application of the hash function).

C.1 A bug in the first version of NewSCS

As mentioned in Section 3.2 the protocol NewSCS is correct, but there are some implementation details, which are easy to miss and our first implementation turned out to contain a bug, which was immediately found due to verification process. More precisely, in Step 10 of the NewSCS protocol it is said that Bob sends to the block chain one of the two transactions: $Fuse^A$ or $\mathtt{CS}^A.Fuse$ and as a result gains 1 bitcoin, what means that at least one of these two transactions becomes confirmed. Therefore, our first implementation just tried to send these two transactions. However, it turns out that there is a strategy of malicious Alice and an interleaving of events in which none of these transactions becomes confirmed and it is necessary to try to sends the transaction $Fuse^A$ again after waiting for time `MAX_LATENCY`. UPPAAL provides diagnostic traces, which allows to easily find bugs like the one mentioned.

D UPPAAL

D.1 A very short introduction to timed automata and UPPAAL

For the lack of space we only sketch the description of UPPAAL and the semantics of the timed automata (focusing on the most relevant features). The reader may consult [9,18] for more information on this topic.

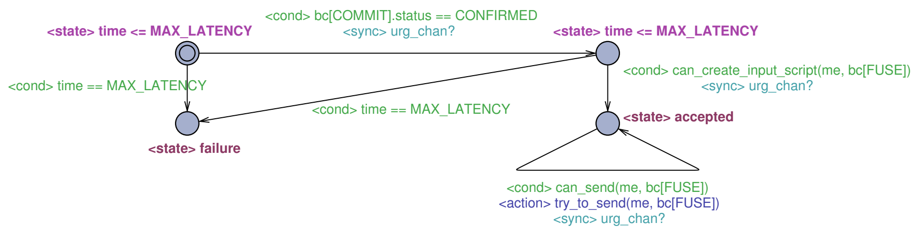
In UPPAAL the automata are generally finite-state automata equipped with clocks. UPPAAL system consists of some number of timed automata, clocks and variables of discrete, bounded sets (also user defined, like records) and functions working on variables and clocks. The state of the system consists of locations of the automata, values of the variables and the clocks evaluation. The clocks evaluate to positive real numbers, but because of bounded set of possible constraints number of different state it which the system can be is finite.

The only possible transitions in UPPAAL are moving all the clocks forward with the same value or to use some of edges to change the state of some of the automata. Both kind of transitions are correct only if all *invariants* and *guards* are satisfied. The invariants are properties of the locations and have to be satisfied each time the system is in this location. The locations may have also names (used in a verification), be urgent (time cannot pass when any automaton is in such location) or committed (the system immediately has to use an edge outgoing from such location). Moreover, edges can use selectors to set a local variable to any value of some type. When an edge is used, it may also run a connected update — it changes values of variables and resets some clocks. The pairs of edges may also be synchronized — in such case they can be used, but only together. The synchronization may also be urgent — then it has to be used if only it can be used. More details and a syntax used in UPPAAL can be found in Appendix D.2.

UPPAAL comes also with a simulator (to e.g. run transitions in a random order) and a verifier. The verifier checks whether the given properties (written in the simplified version of TCTL — timed computation tree logic) are satisfied by the system.

D.2 UPPAAL syntax

Let us analyze the UPPAAL syntax on the example of automaton for Bob from a timed commitment scheme (Fig. 12). It has 4 states: the upper-left is a starting state, the upper-right corresponds to a situation when the $Commit$ transaction is confirmed, but the signature on $Fuse$ from Step. 3 has not yet been received by Bob. The lower-left state (denoted *failure*) means that the commitment was not accepted by Bob and the lower-right (denoted *accepted*) state means that Bob accepted the commitment (Step 4).



<state> time <= MAX_LATENCY
<cond> bc[COMMIT].status == CONFIRMED
<sync> urg_chan?
<state> time <= MAX_LATENCY
<cond> can_create_input_script(me, bc[FUSE])
<sync> urg_chan?
<cond> time == MAX_LATENCY
<cond> time == MAX_LATENCY
<state> accepted
<state> failure
<cond> can_send(me, bc[FUSE])
<action> try_to_send(me, bc[FUSE])
<sync> urg_chan?

Fig. 12. The automaton for Bob in timed-commitment scheme

There are 5 types of labels on the picture[6]:

<select> These labels are placed on the edges to set a local variable used in following condition and/or action to any value from a specified set, e.g. `i` : `TxId`, `cond(i)`, `action(i)` placed on an edge means that this edge can be used for any `i` from `TxId`, provided `cond(i)` is true and then `action(i)` is performed. The mentioned automation for Bob does not have any edges of this kind.

<cond> These labels are placed on the edges and they represent conditions (called *guards* in UPPAAL), which have to be satisfied, e.g. `bc[COMMIT].status == CONFIRMED` means that this transition can be taken only if the transaction $Commit$ has been already confirmed and `can_create_input_script(me, bc[FUSE])` checks if Bob is able to create input script for $Fuse$ transaction, what is equivalent to the fact that he has already received the signature from Step. 3 (Fig. 9).

<action> These labels are placed on the edges and they represent actions (called *updates* in UPPAAL), i.e. functions which are called whenever a transition is taken. In our example there is just one action `try_to_send(me, bc[FUSE])`, which checks whether it is possible to send the given transaction in the current state (e.g. it was not sent earlier, its inputs are confirmed, but not spent, the current party is able to evaluate the corresponding input scripts) and changes the state of the transaction to `SENT` if all conditions are met.

<state> These labels, placed on the states, represent names of the states (*failure* and *accepted*) and *invariants* specifying conditions, which has to be satisfied in the given state. Execution in which the invariant is not satisfied are ignored by UPPAAL. Therefore a node with an invariant `time <= MAX_LATENCY` and an outgoing edge with a guard `time == MAX_LATENCY` guarantee that the edge will be taken exactly at time `MAX_LATENCY` (Moreover it makes impossible to enter the state after time `MAX_LATENCY`, but it is not important in our example.)

<sync> These labels, placed on the edges, represent synchronization channels. Here they are used for a special purpose of obtaining urgent transitions. This is the technical trick described in Sec. D.3, but for understanding this picture it is enough to know that the automaton is not allowed to wait whenever there is an edge available with the label `urg_chan?`.

[6] We color them using the default UPPAAL coloring. We also decided to add special text labels (in the “<>” brackets) in the Figures of automata to make the text readable when printed black and white

D.3 Helper automaton

A transition is called *urgent*, when it has to be taken immediately whenever it is possible. More precisely, the time cannot pass, whenever there is an automaton with an urgent transaction available (i.e. with a satisfied guard). UPPAAL does not provide urgent transitions, but there exists a simple workaround, described e.g. in [9]. The solution is based on the so-called urgent *channels*. Here, the “urgency” means that the time cannot pass whenever there is an automaton with an available transition synchronizing on an urgent channel (availability means that in particular there is another automaton, which can synchronize on the same channel). Therefore, it is enough to mark edges we would like to be urgent with a synchronization label `urg_chan`? and create a special automaton (called `Helper` in our implementation) with one state and a loop with label `urg_chan`! for some urgent channel `urg_chan`.

UPPAAL does not allow to put guards involving clocks on edges with synchronization on urgent channels, so another workaround is needed to achieve this functionality. The simple solution is to check on the edge a value of some boolean shared variable and make sure that the value of this variable is always the same as the value of the clock condition we would like to have on the edge. The correct value of the shared variable can be maintained by another loop in the `Helper` automaton. The `Helper` automaton is presented in Fig. 13.

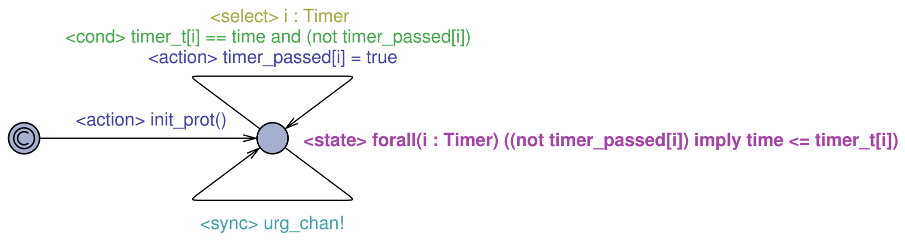

<select> i : Timer
<cond> timer_t[i] == time and (not timer_passed[i])
<action> timer_passed[i] = true
<action> init_prot()
<state> forall(i : Timer) ((not timer_passed[i]) imply time <= timer_t[i])
<sync> urg_chan!

Fig. 13. The `Helper` automaton