Model Checkers Are Cool:
How to Model Check Voting Protocols in Uppaal

Abstract The design and implementation of an e-voting system is a challenging task. Formal analysis can be of great help here. In particular, it can lead to a better understanding of how the voting system works, and what requirements on the system are relevant. In this paper, we propose that the state-of-art model checker Uppaal provides a good environment for modeling and preliminary verification of voting protocols. To illustrate this, we present an Uppaal model of Prêt à Voter, together with some natural extensions. We also show how to verify a variant of receipt-freeness, despite the severe limitations of the property specification language in the model checker.

1 Introduction

The design and implementation of a good e-voting system is highly challenging. Real-life systems are notoriously complex and difficult to analyze. Moreover, elections are *social* processes: they are run by humans, with humans, and for humans, which makes them unpredictable and hard to model. Last but not least, it is not always clear what *good* means for a voting system. A multitude of properties have been proposed by the communities of social choice theory (such as Pareto optimality and nonmanipulability), as well as secure voting (ballot secrecy, coercion-resistance, voter-verifiability, etc.). The former are typically set for a very abstract view of the voting procedure, and consequently miss many real-life concerns. For the latter ones, it is often difficult to translate the informal intuition to a formal definition that is commonly accepted.

In a word, we deal with processes that are hard to understand and predict, and evaluate them against criteria that are not exactly clear. Formal analysis can be of great help here: perhaps not in the sense of providing the ultimate answers, but rather to strengthen our understanding of both how the voting system works and how it should work. The main goal of this paper is to propose that model checkers from distributed and multi-agent systems can be invaluable tools for such an analysis.

Model checkers and Uppaal. Much research on model checking focuses on the design of logical systems for a particular class of properties, establishing their theoretical properties, and development of verification algorithms. This obscures the fact that a model checking framework is valuable as long as it is actually *used* to analyze something. The analysis does not necessarily have to result in a "correctness certificate" of the system under scrutiny. A readable model of the system, and an understandable formula capturing the requirement are already of substantial value.

In this context, two features of a model checker are essential. On the one hand, it should provide a *flexible model specification language* that allows for modular and succinct specification of processes. On the other hand, it must offer a *good graphical user interface*. Paradoxically, tools satisfying both criteria are rather scarce. Here, we suggest that the state of the art model checker UPPAAL can provide a nice environment for modeling and preliminary verification of voting protocols and their social context. To this end, we show how to use UPPAAL to model a voting protocol of choice (in our case, Prêt à Voter), and to verify some requirements written in the temporal logic **CTL**.

Contribution. The main contribution of this paper is methodological: we demonstrate that specification frameworks and tools from distributed and multi-agent systems can be useful in analysis and validation of voting procedures. An additional, technical contribution consists in a reduction from model checking of temporal-epistemic specifications to purely temporal ones, in order to verify a variant of receipt-freeness despite the limitations of UPPAAL.

Structure of the paper. We begin by introducing the main idea behind modeling and model checking of multi-agent systems, including a brief introduction to UPPAAL (Section 2). In Section 3, we provide an overview of Prêt à Voter, the voting protocol that we will use for our study. Section 4 presents a multi-agent model of the protocol; some interesting extensions of the model are proposed in Section 6. We show how to specify simple requirements on the voting system, and discuss the output of model checking in Section 5. The section also presents our main technical contribution, namely the model checking reduction that recasts knowledge-related statements as temporal properties. We discuss related work in Section 7, and conclude in Section 8.

2 Towards Model Checking of Voting Protocols

Model checking is the decision problem that takes a model of the system and a formula specifying correctness, and determines whether the model satisfies the formula. This allows for a natural separation of concerns: the model specifies how the system is, while the formula specifies how it should be. Moreover, most model checking approaches encourage systematic specification of requirements, especially for the requirements written in modal and temporal logic. In that case, the behavior of the system is specified by a transition network, possibly with additional modal relations to represent e.g. the uncertainty of agents. The structure of the network is typically given by a higher-level representation, e.g., a set of agent templates together with a synchronization mechanism.

We begin with a brief overview of UPPAAL, the model checker that we will use in later sections. A more detailed introduction can be found in [5].

2.1 Modeling in UPPAAL

The system model in UPPAAL consists of a set of concurrent processes. The processes are defined by templates, each possibly having a set of parameters.

The parameterized templates are used for defining a large number of almost identical processes. Every template consists of *locations*, *edges*, and optional local declarations. An example template is shown in Figure 2; we will use it to model the behavior of the voter.

Locations are graphically represented as circles. *Initial* locations are marked by a double circle. *Committed* locations are marked by the circled letter C. If any process is in a committed location, then the next transition must involve an edge from one of the committed locations. Those are used for creating atomic sequences or encoding synchronization between more than two components.

The edges are annotated by selections (in yellow), guards (green), synchronizations (teal), and updates (blue). The syntax of expressions mostly coincides with that of C/C++. *Selections* bind the identifier to a value from the given range in a nondeterministic way. *Guards* enable the transition if and only if the guard condition evaluates to true. *Synchronizations* allow processes to synchronize over a common channel `ch` (labeled `ch?` in the receiver process and `ch!` for the sender). Note that a transition on the side of the sender can be fired only if there exists an enabled transition on the receiving side labeled with the same channel identifier, and vice versa. *Update* expressions are evaluated when the transition is taken. For a synchronizing transition, the update expressions on the sender side are executed before the receiver ones. Straightforward value passing over a channel is not allowed; instead, one has to use shared global variables for the transmission. In between writing and reading update expressions, a committed location will be used.

For convenience, we will place the selections and guards at the top or left of an edge, and the synchronizations and updates at the bottom/right.

2.2 Specification of Requirements

To specify requirements, UPPAAL uses a fragment of the temporal logic **CTL** [19]. **CTL** allows for reasoning about the possible execution paths of the system by means of the *path quantifiers* E ("there is a path") and A ("for every path"). A path is a maximal[1] sequence of states and transitions. To address the temporal pattern on a path, one can use the *temporal operators* $\bigcirc$ ("in the next moment"), $\Box$ ("always from now on"), $\Diamond$ ("now or sometime in the future"), and U ("until"). For example, the formula $\mathsf{A}\Box\big(\mathsf{has\_ballot_i} \to \mathsf{A}\Diamond(\mathsf{voted_{i,1}} \vee \cdots \vee \mathsf{voted_{i,k}})\big)$ expresses that, on all paths, whenever voter i gets her ballot form, she will eventually cast her vote for one of the candidates $1, \ldots, k$. Another formula, $\mathsf{A}\Box\neg\mathsf{punished_i}$ says that voter i will never be punished by the coercer.

More advanced properties usually require a combination of temporal modalities with *knowledge operators* K_a, where $K_a\phi$ expresses that "agent a knows that ϕ holds." For example, formula $\mathsf{E}\Diamond(\mathsf{results} \wedge \neg\mathsf{voted_{i,j}} \wedge \neg K_c\neg\mathsf{voted_{i,j}})$ says that the coercer c might not know that voter i hasn't voted for candidate j, even if the results are already published. Moreover, $\mathsf{A}\Box(\mathsf{results} \to \neg K_c\neg\mathsf{voted_{i,j}})$ expresses that, when the results are out, the coercer won't know that the voter

[1] I.e., infinite or ending in a state with no outgoing transitions.

(a)

Obelix	
Idefix	
Asterix	
Panoramix	
	7304944

(b)

X
7304944

Figure 1: (a) Prêt à Voter ballot form; (b) Receipt encoding a vote for "Idefix"

refused to vote for j. Intuitively, both formulas capture different strength of receipt-freeness for a voter who has been instructed to vote for candidate j.

3 Outline of Prêt à Voter

In this paper, we use UPPAAL for modeling and analysis of a voting protocol. The protocol of choice is Prêt à Voter. A short overview of Prêt à Voter is presented here; the full details can be found, for example, in [34,23].

The key innovation of the Prêt à Voter approach is to encode the vote using a randomised candidate list. This contrasts with earlier verifiable schemes that involved the voter inputting her selection to a device that then produces an encryption of the selection. Here what is encrypted is the candidate order which can be generated and committed in advance, and the voter simply marks her choice on the ballot in the traditional manner.

Suppose that our voter is called Anne. At the polling station, Anne registers and chooses at random a ballot form sealed in an envelope and saunters over to the booth. An example of such a form is shown in Figure 1a. In the booth, Anne extracts her ballot form from the envelope and makes her selection in the usual way by placing a cross in the right hand column against the candidate of her choice (for approval or ranked voting, she marks her ranking against the candidates). Once her selection has been made, she separates the left and right hand strips along a thoughtfully provided perforation and discards the left hand strip. She is left with the right hand strip which now constitutes her *privacy protected receipt*, as shown in Figure 1b.

Anne now exits the booth clutching her receipt, returns to the registration desk, and casts her receipt. Her receipt is placed over an optical reader or similar device that records the random value at the bottom of the strip and records in which cell her X is marked. Her original, paper receipt is digitally signed and franked and returned to her to keep and later check that her vote is correctly recorded on the bulletin board. The randomisation of the candidate list on each ballot form ensures that the receipt does not reveal the way she voted, so ensuring the secrecy of her vote. Incidentally, it also removes any bias towards the candidate at the top of the list that can occur with a fixed ordering.

The value printed on the bottom of the receipt is what enables extraction of the vote during the tabulation phase: buried cryptographically in this value is the information needed to reconstruct the candidate order and so extract the vote encoded on the receipt. This information is encrypted with secret keys shared across a number of tellers. Thus, only a threshold set of tellers acting together are able to interpret the vote encoded in the receipt. In practice, the value on the receipt will be the output of an agreed cryptographic hash of the ciphertext committed to the bulletin board.

After the election, voters (or perhaps proxies acting on their behalf) can visit the secure Web Bulletin Board (WBB) and confirm that their receipts appear correctly. Once any discrepancies are resolved, the tellers take over and perform anonymising mixes and decryption of the receipts. At the end, the plaintext votes will be posted in secret shuffled order, or in the case of homomorphic tabulation, the final result is posted. All the processing of the votes can be made universally verifiable, i.e., any observer can check that no votes were manipulated. These are carefully designed so as not to impinge on ballot privacy.

Prêt à Voter brings several advantages in terms of privacy and dispute resolution. Firstly, avoidance of side channel leakage of the vote from the encryption device. Secondly, improved dispute resolution: ballot assurance is based on random audits of the ballot forms, which can be performed by the voter or independent observers. A ballot form is either well-formed, i.e. the plaintext order matches the encrypted order, or not. This is independent of the voter or her choice, hence there can be no dispute as to what choice the voter provided. Such disputes can arise in Benaloh challenges and similar cut-and-choose style audits. Furthermore, auditing ballots does not impinge on ballot privacy, as nothing about the voter or the vote can be revealed at this point.

4 Modelling Prêt à Voter in Uppaal

In this section, we present how the components and participants of Prêt à Voter can be modeled in Uppaal. To this end, we give a detailed description of each module template, its elements, and their interactions. The templates represent the behavior of the following types of agents: *voters*, *coercers*, *mix tellers*, *decryption tellers*, *auditors*, and the *voting infrastructure*. For more than one module of a given type, an identifier $i = 0, 1, \dots$ will be associated with each instance.

The code of the model is available at `https://github.com/pretvsuppaal/model`.

In the model, we use the *ElGamal* encryption algorithm, which allows for re-encryption. The public key is a tuple (p, α, β), where α is a generator of group $\mathbb{Z}_p^*$, $\beta = \alpha^k$ and k is a secret (private key). A plain-text m encrypted with public key PK with randomness y will be noted as $E_{PK}(m, y)$, if the value of y is not known, then it will be noted as $E_{PK}(m, *)$ (or simply $E_{PK}(m)$) instead. A ciphertext c decrypted with private key K is noted by $D_K(c)$.

4.1 Environment

We begin by an overview of the shared environment of action, i.e., the data structures and variables shared by the modules. To capture a possibly repeated block of atomic update expressions, some procedures are introduced. This will allow for more complex expressions and a shorter, reader-friendly form of labels.

The environment includes some global read-only configuration variables:

- `c_total[=3]`: the number of candidates,
- `v_total[=3]`: the number of voters,
- `mt_total[=3]`: the number of mix tellers,
- `dt_total[=3]`: the number of decryption tellers,
- `dt_min[=2]`: the number of participants needed to reconstruct a secret key,
- `z_order[=7]`: an order of cyclic group $\mathbb{Z}_p^*$,
- `pk[=(3,6)]`: the pair of generator α of group $\mathbb{Z}_p^*$ and $\beta = \alpha^k \ (mod\ p)$, where k is a secret key.

From the model configuration values, we derive some auxiliary variables, such as lists of permutations of the batch terms `P_b`, list of permutations of the candidates `P_c`, list of cyclic shifts of the candidates `S_c`, lookup table `dlog`, that maps onion to its seed ($\pm$ possible candidate choice), list of combinations to select a batch's subset for audit `audit_ch` and list of ways of splitting that in two (odd and even) `audit_lr`.

To facilitate readability and manageability of the model code, we define some data structures and type name aliases based on the configuration variables:

- **Ciphertext**: a pair (y_1, y_2), representing some cipher-text.
- **Ballot**: a pair (θ, cl) of onion $\theta = E_{PK}(s, *)$ and candidate list $cl = \pi(s)$, where s is a seed associated with the ballot, $\pi : \mathbb{R} \to Perm_C$ is a function that associates a seed with a permutation of the candidates. To implement basic absorption of marked cell index into the onion, we will only use cyclic shifts of base candidate order and use a Voter's id as a seed of associated with her ballot.
- **Receipt**: a pair (θ, r) of onion θ and an index r of marked cell. It can be used to verify if a term was recorded and if it was done correctly.
- **c_t**: an integer with range [0, c_total), a candidate;
- **v_t**: an integer with range [0, v_total), a voter;
- **z_t**: an integer with range [0, z_total), an element of $\mathbb{Z}_p^*$.

The writable global variables include:

- `board`: a 2-dimensional list of ciphertexts, representing the web bulletin board. The first column is reserved for the batch of onions with absorbed indices from the receipt. The next $(mt\_total \cdot 2)$-columns store re-encryption mixes. The remaining $(dt\_min - 1)$-columns store the intermediate results of threshold decryption; the last one holds the decrypted message.
- `mixes`: encodes which mix teller has a turn at the moment;
- `dt_curr`: encodes the number of currently participating decryption tellers;
- `decryptions`: the number of decryptions made.

There is also a set of globally shared variables used to simulate the passing of a value through a channel; their use will be specified in transition descriptions.

Finally, the global procedures are:

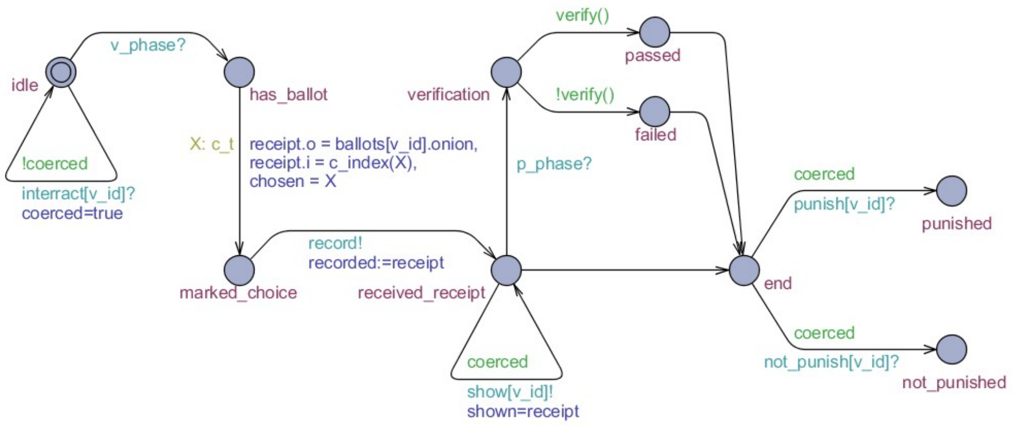


Figure 2: Voter template for the model of Prêt à Voter

– `zpow(a,b)`: returns an element a^b in $\mathbb{Z}_p^*$;
– `encr(m,r)`: returns a ciphertext $E_{PK}(m,r)$;
– `decr(c,k)`: returns a message $D_k(c)$.

4.2 Voter Template

The structure of the Voter template was already shown in Figure 2. The idea is that while the voter waits for the start of election she can possible get coerced. When the ballots are ready, the voter selects a candidate, and transmits the receipt to the system. Then she decides if she wants to confirm that her vote has been accurately recorded and if she wants to show the receipt to the coercer. If coerced, the voter additionally waits for the Coercer's decision to punish her or refrain from punishment.

The module includes the following private variables:

– `receipt`: instance of `Receipt`, obtained after casting a vote;
– `coerced[=false]`: a Boolean, indicating if coercer has established a contact;
– `chosen` - integer value of chosen candidate.

Moreover, the following procedures are included:

– `c_index(target)`: returns an index, at which `target` can be found on the candidate list of a ballot;
– `verify()`: returns *true* if voter's `receipt` can be found on the board list, else returns *false*.

States:

– *idle*: waiting for the election, might get contacted by coercer;

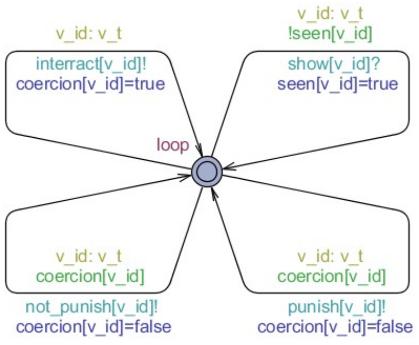


Figure 3: Coercer template

- *has_ballot*: obtained the ballot form;
- *marked_choice*: marked an index of chosen candidate (and destroyed left hand side with candidate list);
- *received_receipt*: receipt is obtained and might be shown to the coercer;
- *verification*: decided to verify the receipt;
- *passed*: has a ZK confirmation that the receipt appears correctly;
- *failed*: has an evidence that the receipt either does not appear or appears incorrectly (in case of index absorption both cases are the same);
- *end*: the end of the voting ceremony;
- *punished*: has been punished by the Coercer;
- *not_punished*: has not been punished by the Coercer.

Transitions:

- *idle→idle*: if was not already coerced, enable transition; if taken, then set `coercion` to *true*;
- *idle→has_ballot*: enabled transition; if taken, then voter acquired a ballot form;
- *has_ballot→marked_choice*: mark a cell with chosen candidate;
- *marked_choice→received_receipt*: send `receipt` to the System over channel `record` using shared variable `recorded`;
- *received_receipt→received_receipt*: if was coerced, enable transition; if taken, then pass the `receipt` to Coercer using shared variable `shown`;
- *received_receipt→verification*: enabled transition; if taken, then Voter decided to verify whether receipt appears on board;
- *(received_receipt || passed || failed)→end*: end of voting ceremony for voter;
- *end→punished*: if was coerced, enable transition; if taken, then Voter was punished by Coercer;
- *end→not_punished*: if was coerced, enable transition; if taken, then Voter was not punished by Coercer.

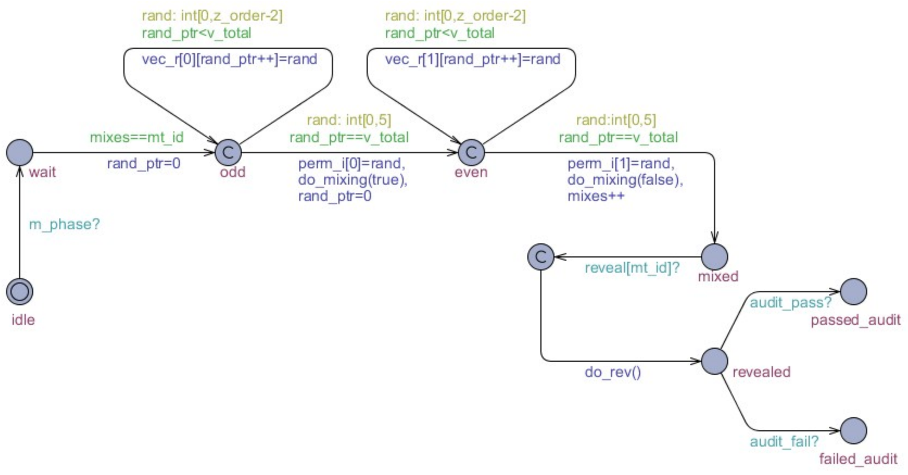


Figure 4: Mteller template

4.3 Coercer

The coercer can be thought of as a party that can influence voters by forcing them to follow certain instructions. To enforce this, the Coercer can punish the voter, or refrain from the punishment. The structure of the Coercer is presented in Figure 3.

Private variables:

– `coercion`: a Boolean list used to keep track of voters that were coerced;
– `seen`: a Boolean list, that indicates if voter has shown the proof of vote.

There is only one state called *loop* for Coercer. It has 4 looped transitions. Their update expressions take form of Boolean value assignment. A more common approach would be cloning that state for possible Boolean evaluations and finding a reachable subset there. However, this would lead to loss of generality and readability of the module (e.g. for 3 voters Coercer should have $2^{3\cdot 2}$ states).

Transitions with respect to Voter `v_id` (clockwise starting with top left):

– establish a contact with Voter, set `coercion[v_id]` to *true*;
– if have not seen proof of vote, enable transition; if taken, set `seen[v_id]` to *true*;
– if Voter was coerced, enable transition; if taken, then punish a Voter, set `coercion[v_id]` to *false*, finalizing interaction;
– if Voter was coerced, enable transition; if taken, then do not punish a Voter, set `coercion[v_id]` to *false*, finalizing interaction.

4.4 Mix Teller (Mteller)

Once the mixing phase starts, each mix teller performs two re-encryption mixes. The order of turns is ascending and determined by their identifiers. The randomization factors and permutation of each mix are selected in a nondeterministic way and stored for a possible audit. When audited, the mix teller reveals the requested links and associated to them factors. The structure of the mix teller is shown in Figure 4.

Private variables:

- `vec_r`: 2 dimensional integer list (size $2 \times |v\_total|$) of randomization factors used for re-encryption;
- `perm_i`: 2 dimensional integer list (size $2 \times |v\_total|$) of permutation indices used for re-encryption;
- `rand_ptr`: (meta variable) index for `vec_r`;
- `mycol`: a pair of `board` column indices reserved for a given mix teller.

Procedures:

- `do_mixing(mi)`: using board column `(mycol[mi]-1)` as an input, re-encrypt each term using randomization factors from `vec_r[mi]`, shuffle them with `perm_i[mi]` permutation and paste result to `mycol[mi]`;
- `do_rev()`: use shared variables rev_r and rev_p to reveal (pass to Auditor) randomization factors and links for the audited terms.

States:

- *idle*: waiting start of mixing phase;
- *wait*: waiting for a turn for mixing;
- *odd*: performing odd mix;
- *even*: performing even mixing;
- *mixed*: finished mixing, passed turn to the next mix teller (if any), waiting for a possible audit;
- *revealed*: revealed values needed for audit, waiting for Auditor's verdict;
- *passed_ audit*: mix teller passed Auditor's correctness check;
- *failed_ audit*: mix teller failed Auditor's correctness check.

Transitions:

- *idle*→*wait*: enabled transition;
- *wait*→*odd*: if it is current mix teller's turn, enable transition; if taken, then initialize `rand_ptr` to zero;
- *odd*→*odd*: if new randomization factors not selected yet, enable transition; if taken, insert random value *rand* into `vec_r[0][rand_ptr]` and then increment `rand_ptr`;
- *odd*→*even*: if randomization factors are ready, enable transition; randomly select permutation index, perform re-encryption mix using those and reset `rand_ptr` counter to generate a new randomness;

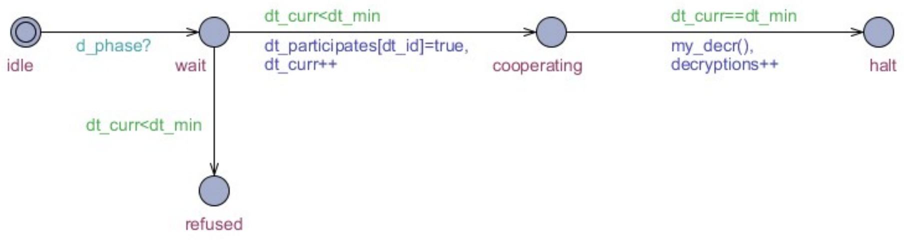


Figure 5: Dteller template

– *even→even*: if new randomization factors not selected yet, enable transition; if taken, insert random value *rand* into `vec_r[1][rand_ptr]` and then `rand_ptr`;
– *even→mixed*: if randomization factors are ready, enable transition; randomly select permutation index, perform re-encryption mix using those and pass turn incrementing `mixes`;
– *mixed→audit*: enabled transition; if taken, then mix teller will be in committed state, from where will have to reveal mix factors for audited terms;
– *revealed→(passed_ audit* || *failed_ audit)*: enabled transition.

4.5 Decryption Teller (Dteller)

We will use *Shamir (2,3)-Threshold Scheme* for decryption. Consider a polynomial $a(x)$ of a degree 1, such that $a(0) = k$, where k is a secret key. Each decryption teller $d_i \in \{0, 1, 2\}$ will have a point (x_{d_i}, y_{d_i}) on that polynomial, where x_{d_i} is publicly known and $y_{d_i} = a(x_{d_i})$ is a key share. In order to reconstruct the secret k, a group of 2 participants will have to cooperate, using Lagrange interpolation formula. We assume that a polynomial $a(x)$ was set and secret shares were assigned to each participant in advance.

In this module, after the re-encryption mixes are done, a decryption teller chooses a subset of participants. If their number is sufficient, they proceed with collaborative decryption. Note that it is possible that none of the decryption tellers agrees to cooperate, which would lead to a deadlock. In order to discard this pathological case, we add an unconditional fairness assumption that only one of them can refuse.

Private variables:

– `k_share`: secret's share of a given teller;
– `x`: a value of first variable in (x, y) pair.

Procedures:

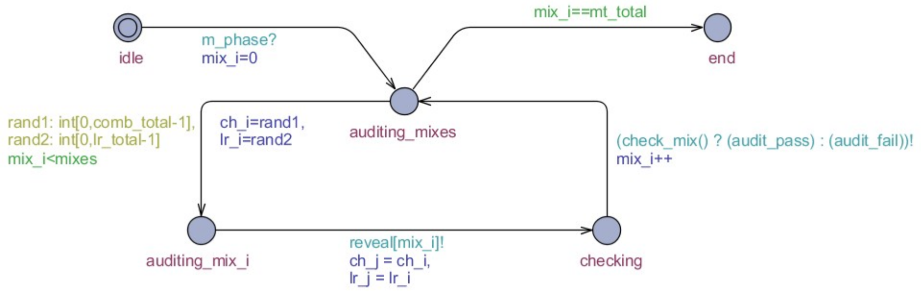


Figure 6: Auditor template

– `my_decr()`: depending on a set of participants, multiply k_share by a proper Lagrange basis and use the result as a key for decryption of an input column. To determine a set of participants, a shared Boolean list *dt_paricipants* is used.

States:

– *idle*: waiting for decryption phase;
– *wait*: wait for turn to make a decision;
– *refused*: refused to cooperate;
– *cooperating*: will participate in decryption;
– *halt*: finished his decrypting.

Transitions:

– *idle*→*wait*: enabled transition;
– *wait*→*refused*: if number of participants less than required for key reconstruction, enable transition;
– *wait*→*cooperating*:if number of participants less than required for key reconstruction, enable transition; if taken, then set `dt_paricipants[dt_id]` to *true* and increment the current number of participants `dt_curr`;
– *cooperating*→*halt*: if number of participants is enough, enable transition; if taken, then proceed to decryption.

4.6 Auditor

In order to confirm that the mix tellers performed their actions correctly, the Auditor conducts an audit based on the randomized partial checking technique (RPC). To this end, he requests each mix teller to reveal the factors for the selected half of an odd-mix batch, and verify whether the input corresponds to the output. The choice s check in-phase or after mixing is nondeterministic. Possible selections for terms and sides of links are encoded as list elements.

The control flow of the Auditor is presented in Figure 6.

Private variables:

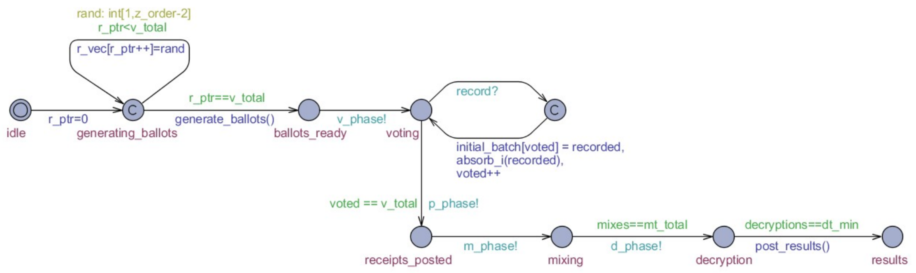


Figure 7: Sys Module

- `mix_i`: index of currently audited mix teller;
- `ch_i`: index of `audit_ch` terms combination (a subset for audit);
- `lr_i`: index of `audit_lr` splitting, used for the left and right linkage reveal.

Procedures:

- `check_mix()`: return true if audited terms correspond to encryption of linked ones from `rev_p` using randomization factors from `rev_r`, otherwise return false.

States:

- *idle*: wait for the start of mixing phase;
- *auditing_mixes*: in a process of auditing mix tellers;
- *auditing_mix_i*: in a process of auditing mix teller *mix_i*;
- *cheking*: received revealed link and randomization factors from mix teller, can now proceed to correctness check;
- *end*: finished mix tellers audit.

Transitions:

- *idle*→*auditing_mixes*: enabled transition; if taken, then set `mix_i` counter to 0;
- *auditing_mixes*→*auditing_mix_i*: if have not audited all mix tellers, enable transition; if taken, then randomly select indices `ch_i` for batch subset and `lr_i` for left-right split;
- *auditing_mix_i*→*checking*: pass the `ch_i` and `lr_i` indices to Mix Teller `mix_i` over channel `reveal[mix_i]` using shared variables `ch_j` and `lr_j`;
- *checking*→*auditing_mixes*: depending on result of `check_mix()` procedure, pass the correctness check verdict to Mix teller using either `audit_pass` or `audit_fail`, then increment `mix_i` counter by 1;
- *auditing_mixes*→*end*: if all Mix tellers were audited, enable transition.

4.7 Voting Infrastructure Module (Sys)

This module represents the behavior of the election authority that prepares the ballot forms, monitors the current phase, signals the progress of the voting procedure to the other components, and posts the results of the election at the end. In addition, the module plays the role of a server that receives receipts and transfers them to the database throughout the election.

Private variables:

- `vote_sum`: list of integers, that maps candidates to the sum of votes they have;
- `r_vec`: list of randomization factors, used for encrypting the seed into onion during ballot generation;
- `r_ptr`: (meta variable) index of `r_vec` element;
- `voted`: a counter for receipts scanned.

Procedures:

- `generate_ballots()`: encrypt the list of seeds using randomization factors from `r_vec`;
- `absorb_i(recorded)`: absorb an index i of the marked cell in from `recorded` receipt into its onion and paste result to the board;
- `post_results()` : map terms of the form g^{r+s} from the last column to the candidate x using formula $x = (r + s)\ (mod\ c\_total)$ and a look-up table `dlog`.

States:

- *idle*: initial state;
- *generating_ballots*: there are either no ballots or they are being generating at the moment;
- *ballots_ready*: all the ballots were prepared;
- *voting*: election phase, when voters can obtain a ballot form and cast their vote;
- *receipts_posted*: batch of initial receipts is now publicly seen and can be checked by voters;
- *mixing*: wait for re-encryption mixes to finish;
- *decryption*: wait for decryption to finish;
- *results*: results tally is posted.

Transitions:

- *idle*→*generating_ballots*: enabled transition; if taken, then reset `r_ptr` counter to generate a list of randomization factors;
- *generating_ballots*→*generating_ballots*: if randomization factors not prepared, enable transition; if taken, then insert a random value `rand` into `r_vec[r_ptr]` and increment `r_ptr` iterator;

– *generating_ballots→ballots_ready*: if randomization factors are prepared, enable transition; if taken, generate ballots using randomization factors from `r_vec`;
– *ballots_ready→voting*: broadcast the start of voting phase to Voters using `v_phase` channel;
– *voting→→voting*: receive a receipt from the voter and from the committed state append it to the board, then increment a counter of receipts scanned;
– *voting→receipts_posted*: if all voters submitted receipts, enable transition; if taken, then broadcast that Voters that initial receipts were posted using the `p_phase` channel;
– *receipts_posted→mixing*: broadcast the start of mixing phase to mix tellers using `m_phase` channel;
– *mixing→decryption*: if all mixes are done, enable transition; if taken, then broadcast the start of decryption phase to decryption tellers using `d_phase` channel;
– *decryption→results*: if all decryptions were done, enable transition; if taken, then calculate and post results tally.

5 Verification

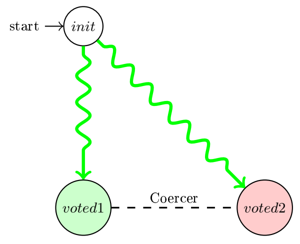
As we already mentioned, we chose Uppaal for this study mainly because of its modeling functionality. The requirement specification capabilities of the tool were of secondary importance. In fact, they are quite limited. First, Uppaal admits only a fragment of **CTL**: it excludes the "next" and "until" modalities, and does not allow for nesting of operators (with one exception that we describe below). Thus, the supported properties fall into the following categories: *reachability* ($\mathsf{E}\Diamond\mathsf{p}$), *liveness* ($\mathsf{A}\Diamond\mathsf{p}$), and *safety* ($\mathsf{A}\Box\mathsf{p}$ and $\mathsf{E}\Box\mathsf{p}$). The only allowed nested formulas come in the form of the *p leads to q* property, written $\mathsf{p} \leadsto \mathsf{q}$, and being a shorthand for $\mathsf{A}\Box(\mathsf{p} \rightarrow \mathsf{A}\Diamond\mathsf{q})$.

Still, Uppaal allows to model-check simple properties of Prêt à Voter, as we show in Section 5.1. Moreover, by tweaking models and formulas in a clever way, one can also verify some more sophisticated requirements, see Section 5.2.

5.1 Model Checking Temporal Requirements

The model in Section 4 allows us to verify properties of Prêt à Voter for different configurations of participants. In particular, we have analyzed variants of the model, generated by different numbers of instances for the Voter template. The values should be chosen carefully, to avoid the state-space explosion problem. We have considered the following properties for verification:

1. $\mathsf{E}\Diamond\mathsf{failed\_audit}_0$: there is a path, where the first mix teller eventually fails an audit.
2. $\mathsf{A}\Box\neg\mathsf{punished}_i$: voter i will never be punished by the coercer.
3. $\mathsf{has\_ballot}_i \leadsto \mathsf{marked\_choice}_i$: on all paths, whenever voter i gets a ballot form, she will eventually mark her choice.

(a) 

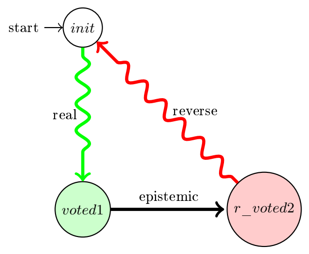
(b) 

Figure 8: (a) Epistemic bisimulation triangle; (b) turning the triangle into a cycle by reversing the transition relation

We verified each formula on several configurations differing by the number of voters ranging from 1 to 5. For the first property, the UPPAAL verifier returns 'Property is satisfied' for configurations with $1, 2, 3$ and 4 voters. In case of 5 voters, we get 'Out of memory.' Formula (2) produces the answer 'Property is not satisfied' and pastes a counter-example into the simulator (if the diagnostic trace was set in the option panel) for all the five configurations. Finally, formula (3) ends with 'Out of memory' regardless of the number of voters.

Optimizations. To keep the model manageable and in attempt to reduce the state space, every numerical variable is defined as a bounded integer in a form of `int[min,max]`, restricting its range of values.[2] The states violating the bounds are discarded at run-time. For example, transition *has_ballot*→*marked_choice* of the Voter (Figure 2) has a selection of value `X` in the assignment of variable `chosen`. The type of `X` is `c_t`, which is an alias to `int[0,c_total-1]`, i.e., the range of meaningful candidate choices.

We also tried to keep the number of used variables minimal, as it plays an important role in model checking procedure.

5.2 How to Make Model Checker Do More Than It Is Supposed To

Many important properties of voting refer to the knowledge of its participants. For example, receipt-freeness expresses that the coercer should never know how the voter has voted. Or, better still, that the coercer will never know that the voter did not vote the way she had been instructed. Similarly, voter-verifiability says that the voter will eventually know whether her vote has been registered and tallied correctly (assuming that she follows the verification steps).

A clear disadvantage of UPPAAL is that its language for specification of requirements is restricted to purely temporal properties. We show that, with some care, one can embed verification of more sophisticated properties in the more limited framework. In particular, we show how to add the verification of some knowledge-related requirements by a technical reconstruction of models

[2] Without the explicit bounds, the range of values would be [-32768,32768].

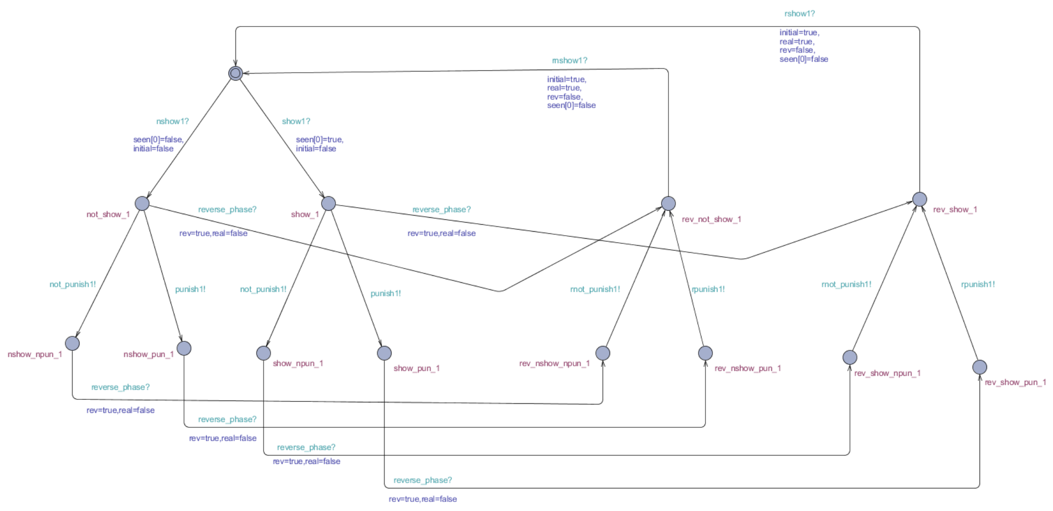


Figure 9: Coercer module augmented with the converse transition relation

and formulas. The construction has been inspired by the reduction of epistemic properties to temporal properties, proposed in [21,24]. Consequently, Uppaal and similar tools can be used to model check formulas of **CTLK** (i.e., **CTL** + Knowledge) that express variants of receipt-freeness and voter-verifiability.

In order to simulate the knowledge operator K_a under the **CTL** semantics, the model needs to be modified. The first step is to understand the interpretation behind the formula $\neg K_c \neg \mathsf{voted}_{\mathsf{i,j}}$ (saying that the coercer doesn't know that the voter hasn't voted for candidate j). That means that, if there is a reachable state in which $\mathsf{voted}_{\mathsf{i,j}}$ is true, there must also exist another reachable state, which is indistinguishable from the current one, and in which $\neg\mathsf{voted}_{\mathsf{i,j}}$ holds. The idea is shown in Figure 8a. We observe that to simulate the epistemic relation we need to create copies of the states in the model (the "real" states). We will refer to those copies as the *reverse states*. They are the same as the real states, but with reversed transition relation. Then, we add transitions from the real states to their corresponding reverse states, that simulate the epistemic relation between the states. This is shown in Figure 8b.

To illustrate how the reconstruction of the model works on a concrete example, we depict the augmented templates in Figures 9–14.

The next step is the reconstruction of formulas. Let us take the formula for the weak variant of receipt-freeness from Section 2.2, i.e., $\mathsf{E}\Diamond(\mathsf{results} \land \neg\mathsf{voted}_{\mathsf{i,j}} \land \neg K_c \neg \mathsf{voted}_{\mathsf{i,j}})$. In order to verify the formula in Uppaal, we need to replace the knowledge operator according to our model reconstruction method (see Figure 8 again). This means that the verifier should find a path that closes the cycle: from the initial state, going through the real states of the voting procedure to the vote publication phase, and then back to the initial state through the

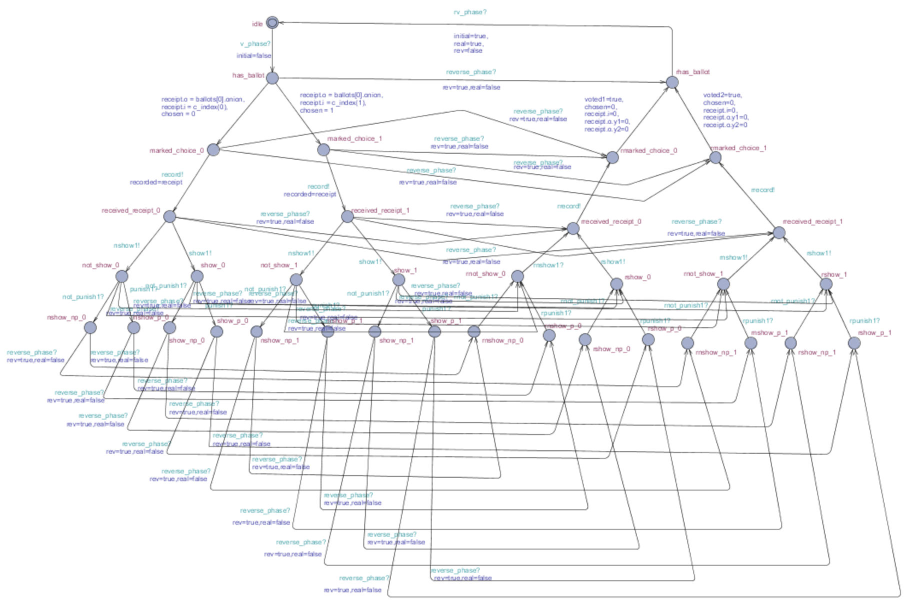


Figure 10: Voter1 with reversed transitions

reversed states. In order to "remember" the relevant facts along the path, we use persistent Boolean variables $\mathsf{voted}_{\mathsf{i,j}}$ and $\mathsf{negvoted}_{\mathsf{i,j}}$: once set to true they always remain true. We also introduce a new persistent variable $\mathsf{epist\_voted}_{\mathsf{i,j}}$ to refer to the value of the vote after an epistemic transition. Once we have all that, we can propose the reconstructed formula: $\mathsf{E}\Diamond(\mathsf{results}\wedge\mathsf{negvoted}_{\mathsf{i,j}}\wedge\mathsf{epist\_voted}_{\mathsf{i,j}}\wedge\mathsf{initial})$. Uppaal reports that the formula holds in the model.

A stronger variant of receipt-freeness is expressed by another formula from Section 2.2, i.e., $\mathsf{A}\Box(\mathsf{results} \rightarrow \neg K_c\neg\mathsf{voted}_{\mathsf{i,j}})$. Again, the formula needs to be rewritten to a pure **CTL** formula. As before, the model checker should find a cycle from the initial state, "scoring" the relevant propositions on the way. More precisely, it needs to check if, for every real state in which election has ended, there exist a path going back to the initial state through a reverse state in which the voter has voted for the selected candidate. This can be captured by the following formula: $\mathsf{A}\Box\big((\mathsf{results} \wedge \mathsf{real}) \rightarrow \mathsf{E}\Diamond(\mathsf{voted}_{\mathsf{i,j}} \wedge \mathsf{init})\big)$. Unfortunately, this formula cannot be verified in Uppaal, as Uppaal does not allow for nested path quantifiers. In the future, we plan to run the verification of this formula using another model checker LTSmin [26] that accepts Uppaal models as input, but allows for more expressive requirement specifications.

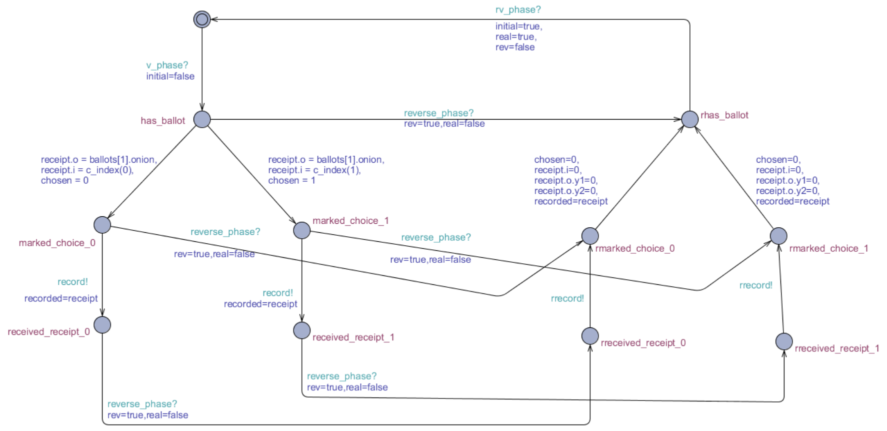


Figure 11: Voter2 with reversed transitions

6 Replicating Pfitzmann's Attack

An adapted version of *Pfitzmann's attack* is a known attack on mix-nets with randomized partial checking (RPC). It can be used to break the privacy of a given vote with probability 1/2 of being undetected during RPC. The leaked information may differ depending both on the implementation of the attack as well as the voting protocol.

The main idea is that the first mix teller, who is corrupted, targets a ciphertext c_i from the odd mix input, and replaces some output term c_j with c_i^δ, where δ is a properly chosen integer (the criteria depend on the implementation). After the decryption results are posted, a pair of decrypted messages m and m' satisfying equation $m' = m^\delta$ can be used to deduce sensitive information.

Clearly, the model presented in Section 4 is too basic to allow for detection of the attack. Still, this can be obtained by a simple extension of the model. For that, we change the Mteller template as shown in Figure 15. The only difference lies in how the first re-encryption mix is done: the corrupted mix teller targets c_0, chooses a random non-zero δ, and uses c_0^δ instead of some other output term. In all other respects, the teller behaves honestly.

Using the Uppaal model checking functionality, it can be verified that there exist executions where the corrupted mix teller has failed the audit, as well as paths where he has passed. That is, both $\mathsf{E}\Diamond\mathsf{failed\_audit}_0$ and $\mathsf{E}\Diamond\mathsf{passed\_audit}_0$ produce "Property satisfied" as the output. We note that, in order to successfully verify those properties in our model of Prêt à Voter, the search order option in Uppaal should be changed from the (default) Breadth First to either Depth First or Random Depth First.

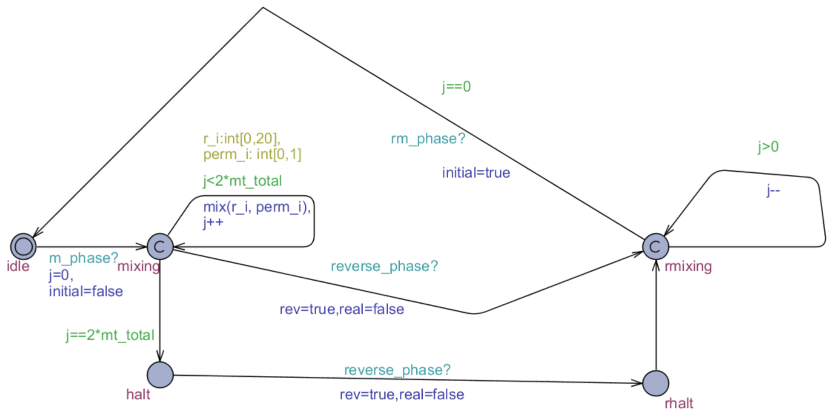


Figure 12: Mix Teller with reversed transitions

7 Related Work

Coercion-resistant and voter-verifiable voting systems. Over the years, the properties of *ballot secrecy*, *receipt-freeness*, *coercion resistance*, and *voter-verifiability* were recognized as important for an election to work properly. In particular, receipt-freeness and coercion-resistance were studied and formalized in multiple ways [6,32,17,27,18], see also [31,38] for an overview. More recently, significant progress has been made in the development of voting systems that would be coercion-resistant and at the same time allow the voter to verify "her" part of the election outcome [37,15]. A number of secure and voter-verifiable schemes have been proposed, most notably the Prêt à Voter protocol for supervised elections [12,34], the Pretty Good Democracy approach to internet voting [36], and Selene, an enhanced form of tracking number-based scheme [35].

Nowadays, such schemes are starting to move out of the laboratory and into use in real elections. For example, Prêt à Voter has been successfully used in one of the state elections in Australia [10] while the Scantegrity II system [11] has been used in municipal elections in the Takoma Park county, Maryland. Moreover, a number of verifiable schemes have been used in non-political elections. E.g., Helios [2] was used to elect officials of the International Association of Cryptologic Research and the Dean of the University of Louvain la Neuve. This strongly emphasizes the need for extensive analysis and validation of such systems.

Formal verification of voting protocols. In voting systems, *verifiable* means that the voters are able to verify the outcome of the election. This is different from *formal verification* whose task is to provide algorithmic tools for checking

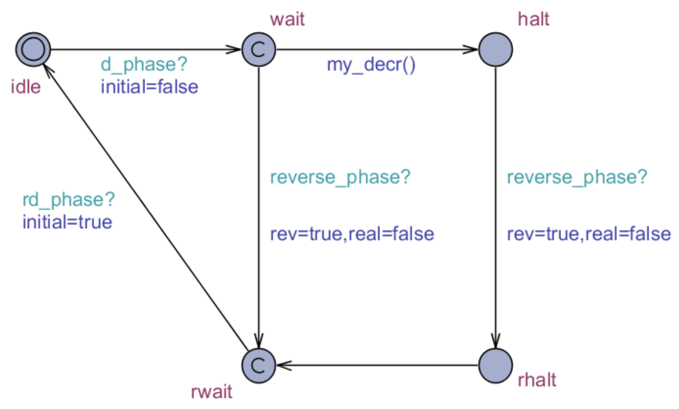


Figure 13: Decryption Teller with reversed transitions

if a system satisfies a given requirement. General verification tools for security protocols include ProVerif [13], Scyther [16], AVISPA [1], and Tamarin [30].

Formal analysis of selected voting protocols, based on theorem proving in first-order logic or linear logic, includes attempts at verification of vote counting [3,33]. The Coq theorem prover for higher-order logic [7] was used to implement the STV counting scheme in a provably correct way [20] and to produce a provably voter-verifiable variant of the Helios protocol [22]. Moreover, the theorem prover Tamarin [30] was used to verify receipt-freeness in Selene [9] and Electryo [39]. Approaches based on model checking are fewer and include the analysis of risk-limiting audits [4] with the CBMC model checker [14].

Multi-agent models and model checkers in verification of security. Multi-agent model checking is virtually unexplored in analysis of voting systems. The only relevant work that we are aware of is [25] where a simple multi-agent model of Selene was proposed and verified using the MCMAS model checker [29]. Related research includes the use of multi-agent methodologies to specify and verify properties of authentication and key-establishment protocols [28,8] with MCMAS. In particular, [8] used the MCMAS model checker to obtain and verify models, automatically synthesized from high-level protocol description languages such as CAPSL, thus creating a bridge between multi-agent and process-based methods.

In all the above cases, the focus is on the verification itself. Indeed, all the tools mentioned above provide only a text-based interface for specification of the system. As a result, their model specifications closely resemble programming code, and "encourage" the usual pitfalls of programming: unreadability of the code, lack of modularity, and opaque control structure. In this paper, we draw attention to tools that promote modular design of the model, emphasize its control structure, and facilitate inspection and validation.

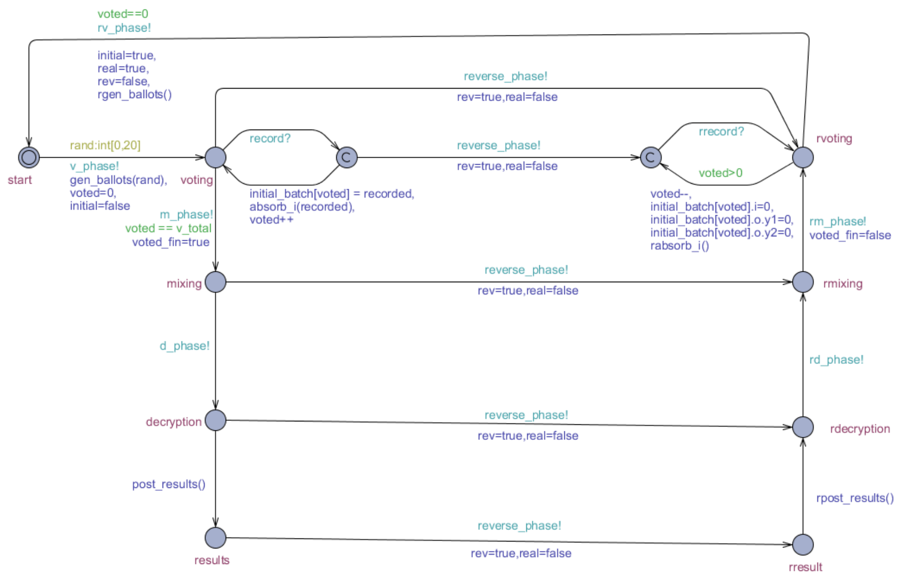


Figure 14: Sys module with reversed transitions

8 Conclusions

Formal methods are well established in proving (and disproving) the correctness of cryptographic protocols. What makes voting protocols special is that they prominently feature human and social aspects. In consequence, an accurate specification of the behaviors admitted by the protocol is far from straightforward. An environment that supports the creation of modular, compact, and – most of all – easy to read specifications can be an invaluable help in the design and validation of voting systems.

In this context, the Uppaal model checker has a number of advantages. Its modeling language encourages modular specification of the system behavior. It provides flexible data structures, and allows for parameterized specification of states and transitions. Last but not least, it has a user-friendly GUI. Clearly, a good graphical model helps to understand how the voting procedure works, and allows for a preliminary validation of the system specification just by looking at the graphs. Anybody who ever inspected a text-based system specification or the programming code itself will know what we mean.

In this paper, we try to demonstrate the advantages of Uppaal through a case study based on the Prêt à Voter protocol. The models that we have obtained are neat, readable, and easy to modify. On the other hand, Uppaal has not performed very well with the verification itself. This was largely due to the fact that its requirement specification language turned out to be very limited – much more than it seemed at the first glance. We managed to partly

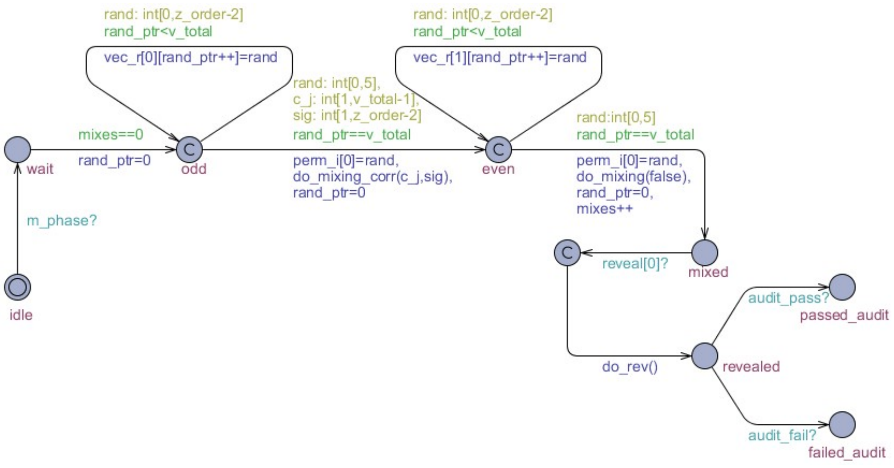


Figure 15: Corrupted Mix Teller module

overcome the limitations by a smart reconstruction of models and formulas. In the long run, however, a more promising path is to extend the implementation of verification algorithms in UPPAAL so that they handle nested path quantifiers and knowledge modalities, given explicitly in the formula.

References

1. A. Armando et al. The AVISPA tool for the automated validation of internet security protocols and applications. In *Proceedings of CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 281–285. Springer, 2005.
2. B. Adida. Helios: web-based open-audit voting. In *Proceedings of the 17th conference on Security symposium*, SS'08, pages 335–348, Berkeley, CA, USA, 2008. USENIX Association.
3. B. Beckert, R. Goré, and C. Schürmann. Analysing vote counting algorithms via logic - and its application to the CADE election scheme. In *Proceedings of CADE*, volume 7898 of *Lecture Notes in Computer Science*, pages 135–144. Springer, 2013.
4. B. Beckert, M. Kirsten, V. Klebanov, and C. Schürmann. Automatic margin computation for risk-limiting audits. In *Proceedings of E-Vote-ID*, volume 10141 of *Lecture Notes in Computer Science*, pages 18–35. Springer, 2016.
5. G. Behrmann, A. David, and K. Larsen. A tutorial on UPPAAL. In *Formal Methods for the Design of Real-Time Systems: SFM-RT*, number 3185 in LNCS, pages 200–236. Springer, 2004.
6. J. Benaloh and D. Tuinstra. Receipt-free secret-ballot elections. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of Computing*, pages 544–553. ACM, 1994.
7. Y. Bertot, P. Casteran, G. Huet, and C. Paulin-Mohring. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Springer, 2004.

8. I. Boureanu, P. Kouvaros, and A. Lomuscio. Verifying security properties in unbounded multiagent systems. In *Proceedings of International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 1209–1217, 2016.
9. A. Bruni, E. Drewsen, and C. Schürmann. Towards a mechanized proof of selene receipt-freeness and vote-privacy. In *Proceedings of E-Vote-ID*, volume 10615 of *Lecture Notes in Computer Science*, pages 110–126. Springer, 2017.
10. C. Burton, C. Culnane, J. Heather, T. Peacock, P. Ryan, S. Schneider, V. Teague, R. Wen, Z. Xia, and S. Srinivasan. Using Prêt à Voter in victoria state elections. In *Proceedings of EVT/WOTE*. USENIX, 2012.
11. D. Chaum, R. Carback, J. Clark, A. Essex, S. Popoveniuc, R. Rivest, P. Ryan, E. Shen, A. Sherman, and P. Vora. Scantegrity II: end-to-end verifiability by voters of optical scan elections through confirmation codes. *Trans. Info. For. Sec.*, 4(4):611–627, 2009.
12. D. Chaum, P. Y. A. Ryan, and S. A. Schneider. A practical voter-verifiable election scheme. In *Proceedings of ESORICS*, pages 118–139, 2005.
13. V. Cheval and B. Blanchet. Proving more observational equivalences with ProVerif. In *Proceedings of POST*, volume 7796 of *Lecture Notes in Computer Science*, pages 226–246. Springer, 2013.
14. E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Proceedings of TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.
15. V. Cortier, D. Galindo, R. Küsters, J. Müller, and T. Truderung. SoK: Verifiability notions for e-voting protocols. In *IEEE Symposium on Security and Privacy*, pages 779–798, 2016.
16. C. Cremers and S. Mauw. *Operational Semantics and Verification of Security Protocols*. Information Security and Cryptography. Springer, 2012.
17. S. Delaune, S. Kremer, and M. Ryan. Coercion-resistance and receipt-freeness in electronic voting. In *Computer Security Foundations Workshop, 2006. 19th IEEE*, pages 12–pp. IEEE, 2006.
18. J. Dreier, P. Lafourcade, and Y. Lakhnech. A formal taxonomy of privacy in voting protocols. In *Communications (ICC), 2012 IEEE International Conference on*, pages 6710–6715. IEEE, 2012.
19. E. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. Elsevier, 1990.
20. M. Ghale, R. Goré, D. Pattinson, and M. Tiwari. Modular formalisation and verification of STV algorithms. In *Proceedings of E-Vote-ID*, volume 11143 of *Lecture Notes in Computer Science*, pages 51–66. Springer, 2018.
21. V. Goranko and W. Jamroga. Comparing semantics of logics for multi-agent systems. *Synthese*, 139(2):241–280, 2004.
22. T. Haines, R. Goré, and M. Tiwari. Verified verifiers for verifying elections. In *Proceedings of CCS*, pages 685–702. ACM, 2019.
23. F. Hao and P. Ryan. *Real-World Electronic Voting: Design, Analysis and Deployment*. Auerbach Publications, 2016.
24. W. Jamroga. Knowledge and strategic ability for model checking: A refined approach. In *Proceedings of MATES'08*, volume 5244 of *Lecture Notes in Computer Science*, pages 99–110, 2008.
25. W. Jamroga, M. Knapik, and D. Kurpiewski. Model checking the SELENE e-voting protocol in multi-agent logics. In *Proceedings of E-VOTE-ID*, volume 11143 of *Lecture Notes in Computer Science*, pages 100–116. Springer, 2018.

26. G. Kant, A. Laarman, J. Meijer, J. van de Pol, S. Blom, and T. van Dijk. LTSmin: High-performance language-independent model checking. In *Tools and Algorithms for the Construction and Analysis of Systems. Proceedings of TACAS*, volume 9035 of *Lecture Notes in Computer Science*, pages 692–707. Springer, 2015.
27. R. Küsters, T. Truderung, and A. Vogt. A game-based definition of coercion-resistance and its applications. In *Proceedings of the 2010 23rd IEEE Computer Security Foundations Symposium*, pages 122–136. IEEE Computer Society, 2010.
28. A. Lomuscio and W. Penczek. LDYIS: a framework for model checking security protocols. *Fundamenta Informaticae*, 85(1-4):359–375, 2008.
29. A. Lomuscio, H. Qu, and F. Raimondi. MCMAS: An open-source model checker for the verification of multi-agent systems. *International Journal on Software Tools for Technology Transfer*, 19(1):9–30, 2017.
30. S. Meier, B. Schmidt, C. Cremers, and D. Basin. The TAMARIN prover for the symbolic analysis of security protocols. In *Computer Aided Verification, Proceedings of CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 696–701. Springer, 2013.
31. B. Meng. A critical review of receipt-freeness and coercion-resistance. *Information Technology Journal*, 8(7):934–964, 2009.
32. T. Okamoto. Receipt-free electronic voting schemes for large scale elections. In *Security Protocols*, pages 25–35. Springer, 1998.
33. D. Pattinson and C. Schürmann. Vote counting as mathematical proof. In *Advances in Artificial Intelligence, Proceedings of AI*, volume 9457 of *Lecture Notes in Computer Science*, pages 464–475. Springer, 2015.
34. P. Ryan. The computer ate my vote. In *Formal Methods: State of the Art and New Directions*, pages 147–184. Springer, 2010.
35. P. Ryan, P. Rønne, and V. Iovino. Selene: Voting with transparent verifiability and coercion-mitigation. In *Financial Cryptography and Data Security: Proceedings of FC 2016. Revised Selected Papers*, volume 9604 of *Lecture Notes in Computer Science*, pages 176–192. Springer, 2016.
36. P. Ryan and V. Teague. Pretty good democracy. In *Security Protocols XVII*, volume 7028 of *Lecture Notes in Computer Science*, pages 111–130. Springer Berlin Heidelberg, 2013.
37. P. Y. A. Ryan, S. A. Schneider, and V. Teague. End-to-end verifiability in voting systems, from theory to practice. *IEEE Security & Privacy*, 13(3):59–62, 2015.
38. M. Tabatabaei, W. Jamroga, and P. Y. A. Ryan. Expressing receipt-freeness and coercion-resistance in logics of strategic ability: Preliminary attempt. In *Proceedings of the 1st International Workshop on AI for Privacy and Security, PrAISe@ECAI 2016*, pages 1:1–1:8. ACM, 2016.
39. M. Zollinger and P. R. P. Ryan. Mechanized proofs of verifiability and privacy in a paper-based e-voting scheme. In *Proceedings of 5th Workshop on Advances in Secure Electronic Voting*, 2020.