Towards Model Checking of Voting Protocols in UPPAAL

Wojciech Jamroga[1,2], Yan Kim[1], Damian Kurpiewski[2], and Peter Y. A. Ryan[1]

[1] Interdisciplinary Centre for Security, Reliability, and Trust, SnT, University of Luxembourg
[2] Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland
{wojciech.jamroga,yan.kim,peter.ryan}@uni.lu, kurpiewski@ipipan.waw.pl

Abstract The design and implementation of a trustworthy 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 modelling and preliminary verification of voting protocols. To illustrate this, we demonstrate how to model a version of Prêt à Voter in UPPAAL, 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.
The aim of this work is to open a new path, rather then deliver the ultimate outcome of formal analysis. A comprehensive model of Prêt à Voter, more accurate specification of requirements, and exhaustive verification are planned for the future.

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 community of social choice theory (such as Pareto optimality and nonmanipulability), as well as researchers who focus on the security of voting (cf. ballot secrecy, coercion-resistance, voter-verifiability, and so on). The former kind of properties 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 will be commonly accepted.

In a word, we deal with processes that are hard to understand and predict, and seek to evaluate them against criteria for which we have no clear consensus. 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 characteristics, 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 have to result in a "correctness certificate". 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 modelling 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, a version of 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.

We emphasize that this is a preliminary work, aimed at exploring a path rather then delivering the ultimate outcome of formal analysis. A comprehensive model of Prêt à Voter, more accurate specification of requirements, and exhaustive verification are planned for the future. We also plan to cover social engineering-style attacks involving interactions between coercers (or vote-buyers) and voters. This will require, however, a substantial extension of the algorithms in UPPAAL or a similar model checker.

Structure of the paper. We begin by introducing the main ideas behind modelling 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 represented by a transition network, possibly with additional modal relations to capture 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 Modelling in UPPAAL

An UPPAAL model consists of a set of concurrent processes. The processes are defined by templates, each possibly having a set of parameters. The templates are used for defining a large number of almost identical processes. Every template consists of *nodes*, *edges*, and optional local declarations. An example template is shown in Figure 2; we will use it to model the behavior of a voter.

Nodes are depicted by circles and represent the local states of the module. *Initial* nodes are marked by a double circle. *Committed* nodes are marked by circled C. If any process is in a committed node, then the next transition must involve an edge from one of the committed nodes. Those are used to create atomic sequences or encode synchronization between more than two components.

Edges define the local transitions in the module. They 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. Straightforward value passing over a channel is not allowed; instead, one has to use shared global variables for the transmission.

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** [14]. **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 $\mathsf{A}\Box\big(\mathsf{has\_ballot_i} \rightarrow \mathsf{A}\Diamond(\mathsf{voted_{i,1}} \vee \cdots \vee \mathsf{voted_{i,k}})\big)$ 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 "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, $\mathsf{A}\Box(\mathsf{results} \rightarrow \neg K_c\neg\mathsf{voted_{i,j}})$ expresses that, when the results are out, the coercer won't know that the voter 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 modelling and analysis of a voting protocol. The protocol of choice is a version of Prêt à Voter. We stress that this is not an up to date version of Prêt à Voter but it serves to illustrate how some attacks can be captured with UPPAAL. A short overview of Prêt à Voter is presented here; the full details can be found, for example, in [32] or [19].

Most voter-verifiable voting systems work as follows: at the time of casting, an encryption or encoding of the vote is created and posted to a secure public bulletin board (BB). The voter can later check that her encrypted ballot appears correctly. The set of posted ballots are then processed in some verifiable way to reveal the tally or outcome. Much of this is effectively a secure distributed computation, and as such is well-established and understood in cryptography. The really challenging bit is the creation of the encrypted ballots, because it involves interactions between the users and the system. This has to be done in a way that assures the voter that her vote is correctly embedded, while avoiding introducing any coercion or vote buying threats.

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 paper ballot in the traditional manner.

Suppose that our voter is called Anne. At the polling station, Anne is authenticated and registered and she chooses at random a ballot form sealed in an envelope and saunters over to the booth. An example of such a form is shown

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

(a)

Discard	Retain
Obelix	
Idefix	
Asterix	
Panoramix	
	7304944

(b)

Retain
X
7304944

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

in Figure 1a. In the booth, she extracts her ballot form from the envelope and marks her selection in the usual way by placing a cross in the right hand column against the candidate (or candidates) of her choice. Once her selection has been made, she separates the left and right hand strips and discards the left hand strip. She keeps 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 the receipt: it is placed over an optical reader or similar device that records the string at the bottom of the strip and registers which cells are 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, thus 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 a pointer (e.g. a hash) to a ciphertext committed to the bulletin board during the setup phase.

After the voting phase, voters can visit the Bulletin Board 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.

Prêt à Voter brings several advantages in terms of privacy and dispute resolution. Firstly, it avoids side channel leakage of the vote from the encryption device. Secondly, it improves on 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 modelled in UPPAAL. To this end, we give a 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 $i = 0, 1, \ldots$ will be associated with each instance.

The code of the model is available at `https://github.com/pretvsuppaal/model`. Here, we present in detail only the Voter template. The details of the other modules can be found in the extended version of the paper, available at `https://arxiv.org/abs/2007.12412`.

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). For the simplicity of modeling, we assume that ElGamal encryption is used.
- **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, and $\pi : \mathbb{R} \to Perm_C$ is a function that associates a seed with a permutation of the candidates. To allow absorption of the index of a marked cell into the onion, we use cyclic shifts of the base candidate order. This means that we just have simple ElGamal ciphertexts to mix.
- **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 $\mathbb{Z}_p^*$.

4.1 Voter Template

The structure of the Voter template is shown in Figure 2. The idea is that while the voter waits for the start of election she might be subject to coercion. When the ballots are ready, the voter selects a candidate, and transmits the receipt to the system. Then she decides if she wants to check how her vote has been recorded, and if she wants to show the receipt to the coercer. If coerced, she also waits for the coercer's decision to punish her or refrain from punishment. The module includes the following private variables:

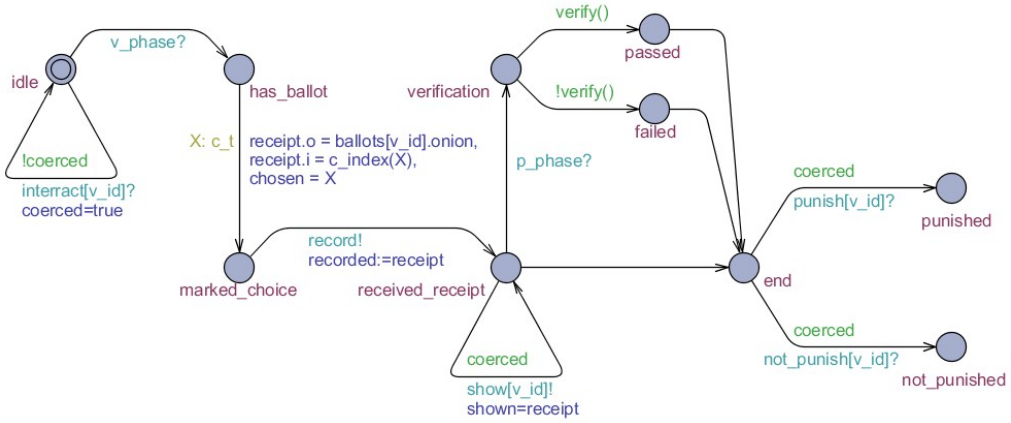

Figure 2: Voter template for the model of Prêt à Voter

- `receipt`: an instance of `Receipt`, obtained after casting a vote;
- `coerced[=false]`: a Boolean value, 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 the voter's `receipt` can be found on the Web Bulletin Board, else it returns *false*.

Local states:

- *idle*: waiting for the election, might get contacted by coercer;
- *has_ballot*: the voter has already obtained the ballot form;
- *marked_choice*: the voter has marked an index of chosen candidate (and destroyed left hand side with candidate list);
- *received_receipt*: the receipt is obtained and might be shown to the coercer;
- *verification*: the voter has decided to verify the receipt;
- *passed*: the voter got a confirmation that the receipt appears correctly;
- *failed*: the voter obtains evidence that the receipt does not appear on BB or appears incorrectly;
- *end*: the end of the voting ceremony;
- *punished*: the voter has been punished by the coercer;
- *not_punished*: the coercer refrained from punishing the voter.

Transitions:

- *idle*→*idle*: if was not already coerced, enable transition; if taken, then set `coercion` to *true*;

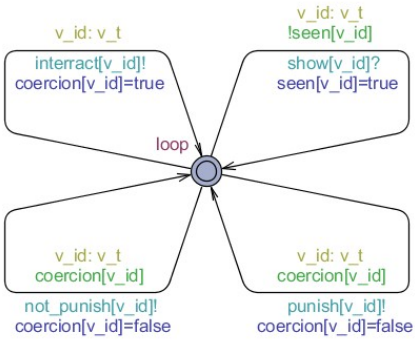

Figure 3: Coercer template

- *idle*→*has_ ballot*: always enabled; if taken, the voter acquires a ballot form;
- *has_ ballot*→*marked_ choice*: mark the cell with the selected candidate;
- *marked_ choice*→*received_ receipt*: send `receipt` to the Sys process over channel `record` using shared variable `recorded`;
- *received_ receipt*→*received_ receipt*: if was coerced, enable transition; if taken, then pass the `receipt` to the coercer using shared variable `shown`;
- *received_ receipt*→*verification*: always enabled; if taken, the voter decides to verify whether the receipt appears on the BB;
- *(received_ receipt* || *passed* || *failed)*→*end*: voting ceremony ends for the voter;
- *end*→*punished*: if was coerced, enable transition; if taken, then the voter has been punished by the coercer;
- *end*→*not_ punished*: if was coerced, enable transition; if taken, the coercer has refrained to punish the voter.

4.2 Coercer

The coercer can be thought of as a party that tries to influence the outcome of the vote by forcing voters to obey certain instructions. To enforce this, the coercer can punish the voter. The structure of the Coercer module is presented in Figure 3; see the extended version of the paper at `https://arxiv.org/abs/2007.12412` for the technical details.

4.3 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 of re-encryption mixes. When audited, the mix teller reveals the requested links and the associated factors, thus allowing Auditor to verify that the input ciphertext maps to the output. The structure of the mix teller is shown in Figure 4.

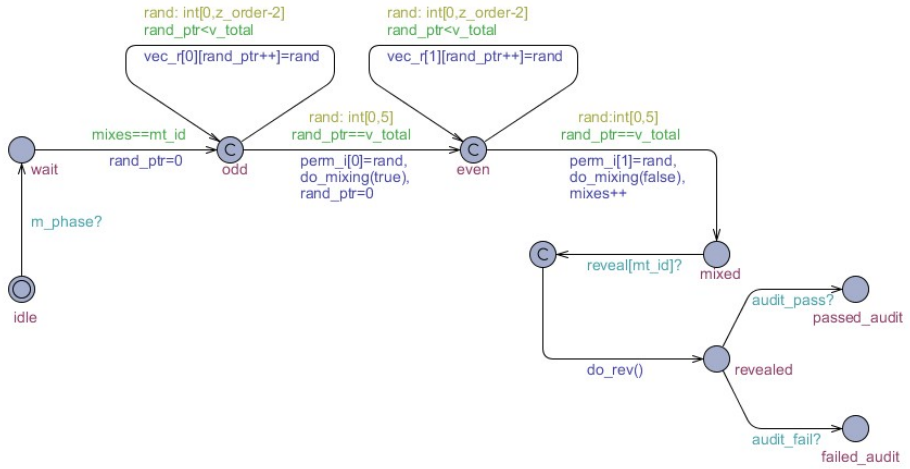

Figure 4: Mteller template

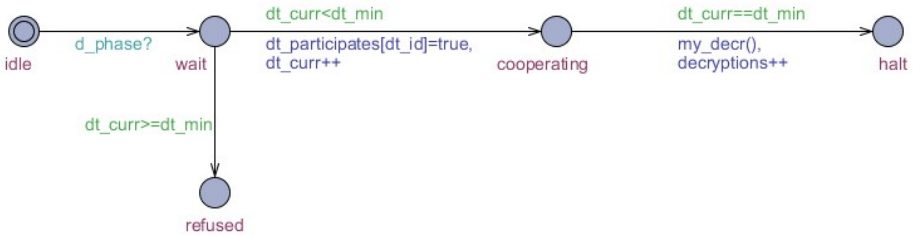

Figure 5: Dteller template

4.4 Decryption Teller (Dteller)

In this module, after the re-encryption mixes are done, a subset of cooperating decryption tellers is chosen nondeterministically. Note that if a subset has less than two elements (e.g. when two or more decryption tellers refused to cooperate), then they should not be able to reconstruct a secret key, which would lead to a deadlock. In order to avoid that, only subsets with cardinality of 2 are considered in our simplified model.

4.5 Auditor

In order to confirm that the mix tellers performed their actions correctly, the auditor conducts an audit. In this paper, we assume that the audit is based on the randomized partial checking technique, RPC in short [20]. To this end, each mix teller is requested to reveal the factors for the selected half of an odd-mix batch, and verify whether the input corresponds to the output. The control flow

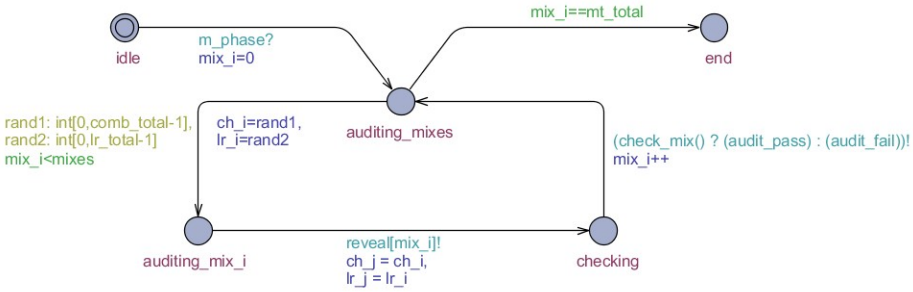

Figure 6: Auditor template

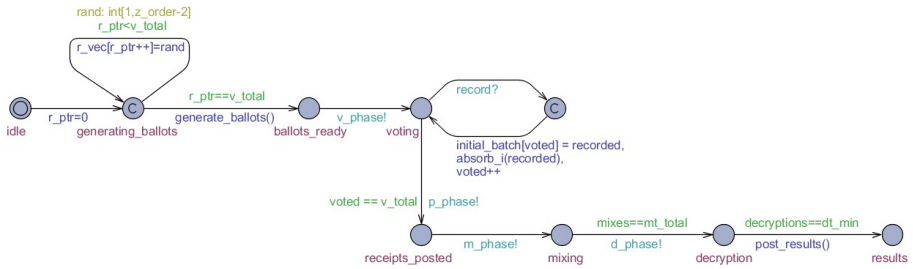

Figure 7: Module Sys

of the Auditor module is presented in Figure 6. In the future, we plan to extend the model with auditing techniques that rely on zero-knowledge proofs.

4.6 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 at the end posts the results of the election. In addition, the module plays the role of a server that receives receipts and transfers them to the database throughout the election. We assume that all the ballots were properly generated and thus omit procedures (e.g. ballot audits) which can ensure that. Capturing related attacks and possible defences remains a subject for future work.

5 Verification

We chose UPPAAL for this study mainly because of its modelling functionality. Interestingly, the model checking capabilities of UPPAAL turned out rather limited for analysis of voting protocols, due to the limitations of its requirement specification language. 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: simple *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 $\mathsf{p} \rightsquigarrow \mathsf{q}$, and being a shorthand for $\mathsf{A}\Box(\mathsf{p} \rightarrow \mathsf{A}\Diamond\mathsf{q})$.

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

5.1 Model Checking Temporal Requirements

It is difficult to encode meaningful requirements on voting procedures in the input language of UPPAAL. We managed to come up with the following properties:

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

We verified each formula on the parameterized model in Section 4. Several configurations were used, with the number of voters ranging from 1 to 5. For the first property, the UPPAAL verifier returns 'Property is satisfied' for the configurations with $1, 2, 3$ and 4 voters. In case of 5 voters, we get 'Out of memory' due to the state-space explosion. This is a well-known problem in verification of distributed systems; typically, the blow-up concerns the system states to be explored in model checking and proof states in case of theorem proving. Formula (2) produces the answer 'Property is not satisfied' and pastes a counter-example into the simulator 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 the 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 if the

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

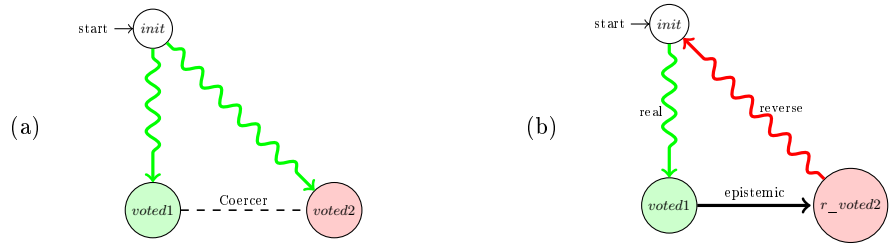

Figure 8: (a) Epistemic bisimulation triangle; (b) turning the triangle into a cycle by reversing the transition relation

voter disobeyed his instructions. 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. Here we show that, with some care, one can use it to embed the verification of more sophisticated properties. In particular, we show how to enable model checking of some knowledge-related requirements by a technical reconstruction of models and formulas. The construction has been inspired by the reduction of epistemic properties to temporal properties, proposed in [17,21]. Consequently, UPPAAL and similar tools can be used to model check some 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 how the formula $\neg K_c \neg \mathsf{voted}_{\mathsf{i,j}}$ (saying that the coercer doesn't know that the particular voter i hasn't voted for candidate j) is interpreted. Namely, 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 Coercer template in Figure 9.

In order to effectively modify the model and verify the selected properties according to the previously defined procedure, the model was first simplified. In the simplified version there are two voters and the coercer can interact only with one of them. Furthermore we removed the verification phase and the tallying phase from the model.

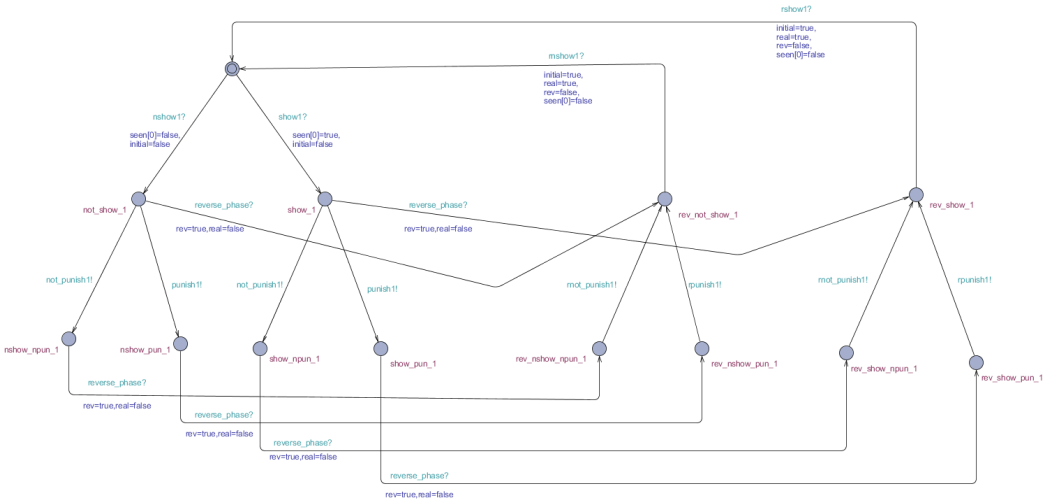

Figure 9: Coercer module augmented with the converse transition relation

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., $\mathsf{E}\Diamond(\mathsf{results} \wedge \neg\mathsf{voted}_{\mathsf{i,j}} \wedge \neg K_c \neg\mathsf{voted}_{\mathsf{i,j}})$. 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 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 of 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 [23] that accepts UPPAAL models as input, but allows for more expressive requirement specifications.

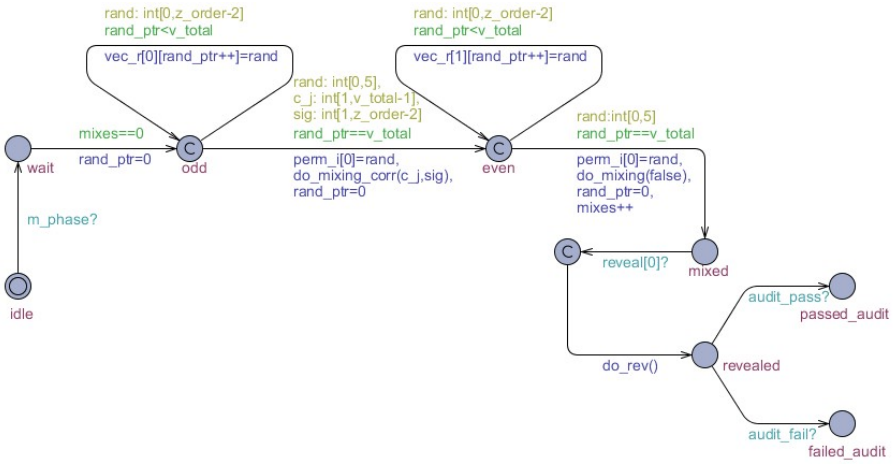

Figure 10: Corrupted Mix Teller module

6 Replicating Pfitzmann's Attack

A version of *Pfitzmann's attack* is known to compromise mix-nets with randomized partial checking [25]. It can be used to break the privacy of a given vote with probability 1/2 of being undetected. The leaked information may differ depending on both the implementation of the attack and the voting protocol.

The 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, After the decryption results are posted, a pair of decrypted messages m and m' satisfying equation $m' = m^\delta$ can be used to identify the corresponding input terms.

Clearly, the model presented in Section 4 is too basic to allow for detection of the attack. Instead, we can examine attacker's behavior by a simple extension of the model. For that, we change the Mteller template as shown in Figure 10. 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. We assume that the corrupt mix teller will always try to cheat. In all other respects, the teller behaves honestly.

Using UPPAAL, it can be verified that there exist executions where the corrupt mix teller's cheating behaviour is not detected during the audit. 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 had to be changed from the (default) `Breadth First` to either `Depth First` or `Random Depth First`.

7 Related Work

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, see also [29] 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 [31,12]. A number of secure and voter-verifiable schemes have been proposed, notably Prêt à Voter for supervised elections [32], Pretty Good Democracy for internet voting [34], and Selene, a coercion mitigating form of tracking number-based, internet scheme [33].

Such schemes are starting to move out of the laboratory and into use in real elections. For example, (a variant of) Prêt à Voter has been successfully used in one of the state elections in Australia [9] while the Scantegrity II system [10] was used in municipal elections in the Takoma Park county, Maryland. Moreover, a number of verifiable schemes were used in non-political elections. E.g., Helios [1] was used to elect officials of the International Association of Cryptologic Research and the Dean of the University of Louvain la Neuve. This underlines the need for extensive analysis and validation of such systems.

Formal analysis of selected voting protocols, based on theorem proving in first-order logic or linear logic, includes attempts at verification of vote counting in [3,30]. The Coq theorem prover [6] was used to implement the STV counting scheme in a provably correct way [16], and to produce a provably voter-verifiable variant of the Helios protocol [18]. Moreover, Tamarin [28] was used to verify receipt-freeness in Selene [8] and Electryo [35]. Approaches based on model checking are fewer and include the analysis of risk-limiting audits [4] with the CBMC model checker [11]. Moreover, [22] proposed and verified a simple multi-agent model of Selene using MCMAS [27]. Related research includes the use of multi-agent methodologies to specify and verify properties of authentication and key-establishment protocols [26,7] with MCMAS. In particular, [7] used MCMAS 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 insufficiently protect from 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.

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 – readable specifications can be an invaluable help.

In this context, the UPPAAL model checker has a number of advantages. Its modelling 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 a version of Prêt à Voter. The models that we have obtained are neat, easy to read, and easy to modify. On the other hand, UPPAAL has not performed 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 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.

The model proposed here is far from complete. We intend to refine and expand it to capture a broader range of attacks, in particular coercion (or vote-buying attacks) that involve subtle interactions between coercer and voters. Prime examples include chain voting and randomisation attacks, where the coercer requires the voter to place an "X" in, say, the first position. Such an attack does not violate any privacy property – the coercer does not learn the vote – but it does deny the voter the freedom to cast her vote as intended. Still more subtle styles of attack have been identified against many verifiable schemes by Kelsey, [24]. Essentially any freedom the voter may have in executing the voting ceremony can potentially be exploited by a coercer.

A comprehensive discussion of coercion-resistance and its possible formalizations is also planned for future work. Another important line of research concerns data independence and saturation results. It is known that, to verify some properties, it suffices to look for small counterexamples [2]. It is also known that such results are in general impossible [15] or incur prohibitive blowup [13]. We will investigate what saturation can be achieved for the verification of Prêt à Voter.

Acknowledgements. The authors acknowledge the support of the Luxembourg National Research Fund (FNR) and the National Centre for Research and Development Poland (NCBiR) under the INTER/PolLux projects VoteVerif (POLLUX-IV/1/2016) and STV (POLLUX-VII/1/2019).

References

1. Ben 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.

2. M. Arapinis, V. Cortier, and S. Kremer. When are three voters enough for privacy properties? In *Proceedings of ESORICS*, volume 9879 of *Lecture Notes in Computer Science*, pages 241–260. Springer, 2016.
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.G. 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. 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.
7. 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.
8. 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.
9. C. Burton, C. Culnane, J. Heather, T. Peacock, P.Y.A. 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.
10. D. Chaum, R.T. Carback, J. Clark, A. Essex, S. Popoveniuc, R.L. Rivest, P.Y.A. Ryan, E. Shen, A.T. Sherman, and P.L. 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.
11. E.M. 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.
12. 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.
13. W. Czerwiński, S. Lasota, R. Lazić, J. Leroux, and F. Mazowiecki. The reachability problem for petri nets is not elementary. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing STOC*, pages 24–33. Association for Computing Machinery, 2019.
14. E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. Elsevier, 1990.
15. S.M. German and A.P. Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 39(3):675–735, 1992.
16. M.K. 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.
17. V. Goranko and W. Jamroga. Comparing semantics of logics for multi-agent systems. *Synthese*, 139(2):241–280, 2004.
18. T. Haines, R. Goré, and M. Tiwari. Verified verifiers for verifying elections. In *Proceedings of CCS*, pages 685–702. ACM, 2019.
19. Feng Hao and Peter Y. A. Ryan. *Real-World Electronic Voting: Design, Analysis and Deployment*. Auerbach Publications, USA, 1st edition, 2016.

20. M. Jakobsson, A. Juels, and R.L. Rivest. Making mix nets robust for electronic voting by randomized partial checking. In *USENIX Security Symposium*, pages 339–353, 2002.
21. 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.
22. 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.
23. 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.
24. J. Kelsey, A. Regenscheid, T. Moran, and D. Chaum. *Attacking Paper-Based E2e Voting Systems*, pages 370––387. Springer-Verlag, Berlin, Heidelberg, 2010.
25. S. Khazaei and D. Wikstroem. Randomized partial checking revisited. In *Topics in Cryptology – CT-RSA 2013*, volume 7779 of *Lecture Notes in Computer Science*, pages 115–128. Springer, 2013.
26. A. Lomuscio and W. Penczek. LDYIS: a framework for model checking security protocols. *Fundamenta Informaticae*, 85(1-4):359–375, 2008.
27. 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.
28. S. Meier, B. Schmidt, C. Cremers, and D.A. 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.
29. B. Meng. A critical review of receipt-freeness and coercion-resistance. *Information Technology Journal*, 8(7):934–964, 2009.
30. 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.
31. Peter Y. A. Ryan, Steve A. Schneider, and Vanessa Teague. End-to-end verifiability in voting systems, from theory to practice. *IEEE Security & Privacy*, 13(3):59–62, 2015.
32. P.Y.A. Ryan. The computer ate my vote. In *Formal Methods: State of the Art and New Directions*, pages 147–184. Springer, 2010.
33. P.Y.A. Ryan, P.B. 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.
34. P.Y.A. 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.
35. M-L. Zollinger, P. Roenne, and P.Y.A. 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.