Ad-hoc e-voting

Koenders, F.A.J.

Award date:
2009

Eindhoven University of Technology

Department of Mathematics and Computer Science

Master's Thesis

Ad-hoc e-voting

by

Frank Koenders

Supervisor: Simona Orzan

Eindhoven, July 24, 2009

Abstract

Mobile devices like smartphones and PDAs are becoming increasingly popular these days, which leads to a demand for mobile applications. Since these mobile devices often support short distance wireless communication technologies like Bluetooth, it is possible to form a small ad-hoc network connecting these devices that can be used in a number of applications.

In this thesis we use an ad-hoc network of mobile devices for the purpose of voting. We define ad-hoc e-voting as a new concept in e-voting, where mobile devices are used to set up a e-voting session on the spot. Since normal e-voting protocols are not suited for an ad-hoc e-voting session we first identify building blocks and properties of an ad-hoc e-voting protocol. After this has been done we give four simple e-voting protocols that are intended to be used in the ad-hoc setting.

For two of the protocols we have constructed we use formal methods to verify some of their properties. We do this for one protocol by performing model-checking on a model of the protocol in the mCRL2 specification language, which is unfortunately only possible for a small number of voters. For the other protocol we prove that the protocol with an arbitrary number of voters preserves privacy. We do this by proving an equivalence relation on processes using recently developed techniques on Parameterized Boolean Equation Systems.

Contents

Chapter 1

Introduction

In this thesis we present the results of our study on ad-hoc electronic voting. Electronic voting (e-voting) in general is an interesting topic that has been actively researched in the last few decades. We introduce a new concept in the world of e-voting: ad-hoc e-voting, e-voting using mobile devices like PDAs or mobile phones. In this chapter we start by giving a brief history of e-voting, after which we introduce the concept of ad-hoc e-voting. We conclude this chapter by providing an overview of the remainder of the thesis.

1.1 A brief history of electronic voting

After the rise of distributed systems in the seventies and early eighties researchers have looked at the numerous possibilities of these systems. In the late eighties security protocols were developed based on distributed computation that could be used in different areas, including that of e-voting [GMW87, BOGW88]. These security protocols were made possible by the invention of two new methods: secure key-agreement and zero-knowledge proofs. Using the Diffie-Hellman key-exchange protocol [DH76] it was made possible to establish a shared secret key over an insecure channel. *Zero-knowledge proofs* introduced by Goldwasser et al. [GMR85] provided a way to prove a certain statement without revealing any of the corresponding secrets. This way it became possible to detect when an entity does not follow the protocol correctly without disrupting the protocol.

Soon after the introduction of those distributed computation schemes that allowed some kind of voting to be possible [GMW87, BOGW88], the first real electronic voting schemes emerged [Cha88, FOO92, BT94]. With these schemes, the concept of electronic voting was further explored. The setting of electronic voting was described and a number of properties of a voting session were defined. Most of these properties, for example that a voter can only vote once, came directly from similar properties in normal non-electronic voting. The challenge has since then been to construct a simple protocol that satisfies as many of these properties as possible.

Since the early nineties, the research on electronic voting has shifted from the small-scale voting schemes in the distributed computation setting to large-scale voting sessions [Cha88, FOO92, BT94, OU98, Sch99, HS00]. Where the early voting schemes based on distributed computation relied on broadcast to achieve their properties, these new large-scale voting schemes needed another method to do that. Using *broadcast communication* each message is sent to every other entity in the network. Thus each entity in the network has full knowledge of what happens in the protocol and can check whether the protocol is performed correctly. Broadcast communication can however not be used in practice in large-scale voting schemes, since it generates too many messages. Therefore large-scale voting schemes use *point-to-point communication* (also called *unicast communication*), where one entity can send a message to only one other entity. Unfortunately, by using point-to-point communication, entities in large-scale voting schemes do not have full knowledge of what happens in the protocol and are unable to check whether the protocol is performed correctly.

Most of the large-scale voting schemes solve the problem of not being able to check whether

the protocol is performed correctly by introducing one or more central authorities in the protocol. Typical roles for these extra parties are administrators (that for example check whether a voter is eligible to vote), collectors or counters (that construct the tally). By giving a small number of central authorities crucial responsibilities in the protocol, it has to be made sure that these parties are trustworthy. In some schemes the presence of a trusted third party is assumed to do this. A *trusted third party* is an entity that is trusted in the protocol and that can be used to establish some sort of secure method to interact between different parties in the protocol. There is unfortunately a major drawback with the introduction of these central authorities, namely that they are often solely responsible for a single task in the voting scheme, ranging from the registration of individual voters to the publishing of a tally. Therefore trust has to be placed in such parties. For a large part this trust can be gained by letting them prove or motivate their actions, but it is difficult to establish a protocol in which a central authority can do absolutely nothing dishonest.

Another disadvantage that comes with the introduction of central authorities is that they each form a single point of failure. Since typically each task is performed by a single party, the protocol is not able to fully function if such party can for some reason not perform its task. Reasons can include hardware failure or denial-of-service attacks from a malicious collective. Thus although the introduction of extra parties can greatly help to satisfy the properties of a voting scheme, one has to be aware of the disadvantages that come with them.

As mentioned earlier, the majority of the voting schemes developed in recent years are targeted towards large-scale elections. A reason for this can be that there have been a number of discussions and plans (some of which that have already been implemented) to use electronic voting schemes over the Internet in state- or nation-wide elections [MM06, JRSW04]. It should be noted that the idea of small-scale voting is still researched, examples include work by Fouque et al. [FPS01], Kiayias and Yung [KY02], Damgård and Jurik [DJ03], and Groth [Gro04]. This research mostly concerns stronger notions of properties of e-voting protocols and how these can be satisfied most efficiently.

1.2 Ad-hoc e-voting

In this thesis we introduce a new concept in electronic voting: ad-hoc e-voting. An *ad-hoc e-voting* session is a small-scale e-voting session that is typically performed using mobile devices, that can be set up without prior arrangements and that does not include a central authority. Think for example of a corporate meeting in which the board of directors wants to vote on a certain proposal. Another example can be a social gathering of a group of people that want to decide what they are going to do next, for example what movie will be shown on the big screen that is present in the hall they are in. In such situations it is likely that one wants to initiate the voting session ad-hoc, that is without the need to first set up some central authorities. Thus one wants to be able to set up a voting session on the spot without preparations. Another property of ad-hoc e-voting networks is that the number of voters is not always known in advance and that the number of participants during the voting session is dynamic. When we think of the social gathering example it is easy to imagine that voters want to join an already started voting session or that voters leave the voting session early.

Ad-hoc networks are actively researched, although most research is not on applications of this type of networks but on creating and using such networks. Many research is on routing protocols for ad-hoc networks or on Media Access Control protocols, which provide control mechanisms for channel access. One of the prototypical applications of ad-hoc networks is mobile conferencing, but, like with many other ad-hoc applications, the idea has been researched [BFHB05] but there are no protocols in literature on how such ideas can be realized in practice.

By using mobile devices like mobile phones or PDAs we have strong building blocks to construct an ad-hoc e-voting protocol. All mobile devices in a certain range can form the network upon which the voting session is carried out. These devices form the nodes of a so called *piconet* (introduced by Bennett et al. [BCE+97]): an ad-hoc computer network that links individual mobile devices. By using mobile devices we can assume both anonymous as well as authenticated broadcast. The

most interesting building block we have is an anonymous broadcast channel, that can not, contrary to authenticated broadcast, easily be established in most types of networks. Since most computer networks do not support anonymous broadcast out-of-the-box, protocol designers have to rely on often complex methods to establish it. A widely used example are mix-nets [Cha81], in which multiple layers of encryption are added on a message, which is then sent through a number of decrypters thereby gaining the desired anonymity.

Another building block is the ability of mobile devices to create a synchronous network. In a *synchronous network* the sending of messages is regulated by a clock and as a result of that the delay of a message is a fixed number of time units. Therefore it is possible, contrary to asynchronous networks, to check in a fixed number of time units whether a message has been received. Furthermore, synchronous networks ensure that messages arrive in the same order as they are sent.

One can think of situations where it is not wanted or even possible to perform an ad-hoc e-voting session on the voter's mobile phones or PDAs. It can be the case that for example not all voters have the possession of a mobile device. In the protocols we have designed each voter is identified by his mobile device in order to prevent that a single voter casts multiple votes, it is thus not possible to share a mobile device. Another reason to not want to use your own mobile device to vote can be for reasons of security. When mobile phones or PDAs are used for ad-hoc e-voting it might be worth the trouble to install some sort of malware on someone else's mobile device. Think for example of a corporate setting where the chief supplies his team PDAs with pre-installed malware to see how they vote.

As an alternative, therefore we come up with some protocols that use trusted devices. In our setting, a *trusted device* is a device that is needed to participate in a voting session and where the users place trust in the fact that the device works as specified. Using these trusted devices voters can set up or participate in a voting session. We have designed two protocols depending on the assumptions made on what is possible with the trusted devices, for example if the trusted device's internal memory can be read or not. Note that trusted devices are not a way to achieve *security-through-obscurity*. The design and implementation of the trusted devices can be open, thereby hopefully gaining the required trust from the voters.

Properties of an ad-hoc e-voting protocol Since the ad-hoc e-voting setting differs from the normal e-voting setting the properties that the protocols ideally satisfy are slightly different. First, let us review the standard list of properties for a voting protocol. Some properties are not always defined exactly the same, the definition given here is thus an interpretation.

eligibility only legitimate voters can vote

fairness no early results can be obtained (that can influence remaining voters)

privacy no one can get to know the vote of another voter

receipt-freeness a voter is unable to prove that he voted in a certain way (to prevent coercion or vote-buying)

individual verifiability a voter can verify that his vote is really counted

universal verifiability a voter can verify that the published tally is equal to the sum of all (valid) votes

unicity no voter can vote more than once

coercion-resistance a voter can not cooperate with a coercer to prove to the coercer that he voted in a certain way

As far as we know, there is no voting protocol that satisfies all of the above properties. This is mainly because the list contains two contradicting properties: namely individual verifiability and receipt-freeness. To explain the problem briefly: in order to have individual verifiability you need to have some sort of identification of your vote, such that you can identify your vote in the list of votes that is published at the end of the voting session. However, when you can identify your vote

uniquely, you have a receipt for your vote. Hence receipt-freeness and individual verifiability are hard (if not impossible) to achieve at the same time. There has been proposed a solution [Cet08], but we feel this does not really solve the problem: it only makes it harder to coerce someone.

In the ad-hoc e-voting setting we add two properties to the list of properties above:

on-line property a voter can join or leave the voting session at any time without losing the possibility to vote once

walk-away property after a voter has cast his vote he can leave the voting session ("walk-away") with the assurance that his vote is counted

Both properties are ideally satisfied in an ad-hoc e-voting protocol. Consider the following example. Say the city council of some town wants to ask the opinion on some topic, for example the closing time of the weekly street market. Instead of setting up some referendum on this topic, they can put up an ad-hoc e-voting on the street market itself. It is suggested that the voter participation increases when the issue on which needs to be voted relates to the voter's time and place [DiM02]. Then it is preferred that voters can join the voting session at any time they like, in order to cast their vote. This is described by the on-line property. The walk-away property [OMA+99] makes sure that voters do not have to come back at a later time after they have cast their vote, for example to disclose some key.

Since mobile devices have limited processing power we want to make our protocols as light as possible. Thereby we do not only want to make them light in terms of computation, but we also want to keep the number of messages as low as possible since we use broadcast communication. In order to do this we may have to weaken, or even sacrifice some properties. We think we are allowed to do this since most ad-hoc e-voting sessions will be in relatively small groups, often consisting of acquaintances, and about relatively light topics. The protocols we design are thus not intended to be used in for example local elections in order to choose members of the city council.

Why existing protocols do not suffice Now the ad-hoc e-voting setting has been defined, one may wonder why we can not use existing broadcast protocols to perform an ad-hoc e-voting session. There are several reasons for this:

On-line property with respect to leaving voters In a number of broadcast protocols the on-line property is not properly satisfied with respect to leaving voters. For example in the voting protocol by Groth [Gro04] the voting part of the protocol needs to be started over when some voter leaves the protocol without casting a vote.

On-line property with respect to joining voters Practically all voting protocols (both small- and large-scale) only consider a fixed set of n voters. They do not consider voters that want to join the protocol at a later stage.

Central authorities A number of broadcast protocols still require the presence of a central authority in order to perform the protocol. For example the self-tallying voting protocol by Kiayias and Yung [KY02] requires a bulletin board authority for administering the election.

Computationally expensive Most broadcast protocols are designed to satisfy as many properties as possible, and in the strongest sense as possible. This is of course a good principle, however this often results in computationally expensive protocols (for example the protocol by Damgård and Yurik [DJ03]). In ad-hoc e-voting protocols the aim is to find a balance between the computational complexity and the satisfaction of properties.

Walk-away property In a number of broadcast protocols the walk-away property is not properly satisfied. For example in the voting protocol by Groth [Gro04]: a voter has to wait for his turn to cast his vote after he has registered his key.

Difficulties in constructing voting protocols In constructing voting protocols, the privacy and the unicity properties are generally the most difficult properties to satisfy. While one can think of situations in which the privacy property is not required, it is hard to find an application for a voting protocol that does not satisfy the unicity property. When either the privacy property or the unicity property is not required, very easy protocols can be constructed.

A voting protocol without privacy When no privacy is required, it is possible to construct a protocol that in general consists of sending the votes encrypted using a key generated by a threshold encryption scheme. In a *threshold encryption scheme* there is a single (public) encryption key that can be used by all voters to encrypt messages. However, in order to decrypt a message encrypted using that key, at least k (the *threshold value*) parties need to collaborate to disclose the decryption key. Assume that this cooperation is done by exchanging messages s_i ($1 \leq i \leq n$, where n is the number of voters) and that using k unique messages s_i it is possible to disclose the decryption key. In order to be an effective threshold scheme, we need to assume that the number of dishonest voters is at most $k-1$ and the number of honest voters at least k. That way, a collective of dishonest voters can not disclose the key when this is not allowed yet.

In the protocol, voters can join and send their votes by encrypting the pair containing their vote v_i and a newly generated *nonce* n_i (basically a random value) with the public threshold encryption key. This continues until a certain deadline has been reached. After that deadline, at least k voters have to send their message s_i. Then the decryption key can be disclosed and everyone can count the tally. For communication, an *authenticated* broadcast channel is used, that is a broadcast channel on which for each message the identity of the sender is known.

In a more formal notation we can describe this protocol as follows, where V_i denotes the voter with identity i and where K denotes the threshold encryption key.

Voter V_i $(1 \leq i \leq n)$

1. $V_i \rightrightarrows : \{|(v_i, n_i)|\}_K$
2. $(\leftleftarrows V_j : \{|(v_j, n_j)|\}_K)^{n-1}$ *for all* $j, 1 \leq j \leq n \wedge i \neq j$

DEADLINE

3. $V_i \rightrightarrows : s_i$
4. $(\leftleftarrows V_j : s_j)^{\geq k-1}$

We explain this notation in detail in Chapter 3. For now it is enough to know that $A \rightrightarrows: m$ means that entity A broadcasts message m, that $\leftleftarrows B : m'$ means that message m' is received from entity B and that $(a)^n$ means that action a is repeated n times.

In this protocol fairness is guaranteed due to the threshold encryption scheme and unicity can be guaranteed by only accepting the first encrypted vote per voter. The privacy property is not satisfied since an authenticated broadcast channel is used.

A voting protocol without unicity When we want to construct a voting protocol without unicity, but with privacy, we can use the same protocol as we have just described with only one difference. We have to replace the authenticated broadcast channel by a anonymous broadcast channel. The anonymous broadcast channel is used to satisfy the privacy property, but by using that it becomes impossible to guarantee unicity.

Since there is thus a substantial difference between regular broadcast e-voting protocols and ad-hoc e-voting protocols and since it is not trivial to come up with a protocol satisfying as many properties as possible, we think the ad-hoc e-voting setting is interesting to research.

1.3 Roadmap

This thesis is structured as follows. In the next chapter we present four ad-hoc e-voting protocols that we have developed. Two of these are using regular mobile devices like mobile phones or PDAs and two are using trusted devices. Since we want to know whether these protocols are correct we verify two of them. In Chapter 3 we make a model of the protocol from Section 2.2 and use this model for the verification of some properties for a specific number of voters. In Chapter 4, we use recently developed techniques to prove the privacy property of the protocol in Section 2.4 for an arbitrary number of voters. Chapter 2 thus contains the descriptions of our protocols and Chapters 3 and 4 contain the verification of (properties of) two protocols. Each of these chapters can be read independently, but for the chapters on verification it is highly advised to read the corresponding protocol description in Chapter 2.

Chapter 2

New protocols

In this chapter we present four ad-hoc e-voting protocols. First we start by describing some general assumptions on the environment in which the protocols can be used and then we discuss each protocol separately. We do this by first giving a general overview of the protocol. After that, we explain the protocol in-depth by dividing it into phases and by clearly stating what happens per phase. Once the protocol is defined we give an analysis of the protocol: we explain why certain constructs are needed, we address potential security risks and we explain ways to improve the protocol when possible. We finish each protocol description by summarizing which of the ad-hoc e-voting properties are satisfied.

2.1 General assumptions

Before we start describing the protocols, we first discuss some general assumptions on the environment and introduce some notation. First, we assume the existence of a reliable authenticated broadcast channel. A channel is *reliable* when no messages get lost on the channel, that is when all messages that are sent arrive at the intended recipient. Then we also assume the existence of a reliable anonymous broadcast channel for the protocols using regular mobile devices. The number of honest voters needs to be at least the threshold value k and the number of dishonest voters can be at most $k-1$. Furthermore we assume that the protocols are used in an *on-line setting*, that is that voters can join the protocol at any time.

On the cryptographic level we use the following notation: $\{m\}_K$ denotes the deterministic symmetric encryption of message m using key K and $\{|m|\}_{\mathrm{K}}$ the deterministic asymmetric encryption of message m using key K. To denote that a message m is signed by signer S we write $\sigma_S(m)$. In one protocol we use blinded signatures, which are a form of digital signing where the content of the message is disguised before it gets signed. After the message has been signed, it is possible to unblind it while preserving the signature. To denote a message m blinded using blinding factor b we write $\chi(m, b)$. Finally we use $\#(m)$ to denote the hash of message m.

For the protocols using the trusted devices we have to make some extra assumptions on what is possible using these devices. We assume that the data in the memory of the trusted device can not be altered illegally. In Section 2.4 we describe a protocol for the situation in which the voter is also unable to read the internal memory, whereas in Section 2.5 we describe a protocol where a voter can read the internal memory. From the inability to illegally alter data it follows that a voter can not alter messages or send extra messages. The only way to prevent a message from being sent is by switching off a trusted device. We assume that it is possible to eavesdrop on the trusted devices using other devices. However it is not possible to send messages using other devices since we assume that the trusted devices have a method to authenticate themselves to each other.

2.2 A poor man's (k,n) threshold ad-hoc voting scheme

The first protocol we describe is a protocol for regular mobile devices. It is called "A poor man's (k, n) threshold ad-hoc voting scheme" since it is inspired by threshold signature schemes but only satisfies some of the properties of those schemes, although in a cheaper way (using less cryptography). In a (k, n) *threshold signature scheme* a group of at least k out of n signers are needed to sign a message. From this message, it is not possible to extract information as to which k signers collaborated to sign the message. We however do not need such strict property in our voting scheme.

The general idea of this protocol is as follows. Someone initiates a voting session by announcing the session and its candidates. A voter who wants to vote picks a vote and a nonce. He combines these into a blinded message and broadcasts this message (authenticated) over the network. By sending this message the voter tells everyone he wants to register his vote. All members of a pre-determined (static) signing group (a subset of the set of voters) check whether this voter did not already register a vote. If he did, the members take no action. If he did not, each member of the signing group signs the blinded vote and broadcasts that. Then the voter unblinds all these messages in order to construct his initial ballot, consisting of all signed unblinded votes.

After a certain deadline, no more votes can be registered. Then the signing group broadcasts which signers have been dishonest. Votes signed by these signers have to be removed by the voters from their ballots. After that, the ballots can be cast over the anonymous channel such that every voter can construct the tally.

A disadvantage of this protocol is that a static group of signers is required. This requirement can however be removed at the cost of computational complexity by using a (k, n) threshold blind signature scheme instead of our signature scheme. From the signatures created by such scheme it is impossible to extract which subset of signers constructed the signature and therefore the dishonesty lists can be removed from the protocol. There exists a few of those schemes in literature [KKL01, LJY02], however they are very expensive in terms of computation and can therefore be impractical when used on mobile devices with limited processing power. Thus we have here a trade-off between the weight and the robustness of the protocol.

2.2.1 The protocol

Announcement

- The initiator announces the voting session, including the candidates.
- All voters that want to vote reply by sending a message over the authenticated channel.

Deadline

Preparation

- The initial number of voters is set. Based on that number and some public formula, the threshold value k is calculated.
- A static group of $2k - 1$ signers is set.

Registration

- Each voter V_i:
  - picks a vote v_i, generates a nonce n_i and a blinding factor b_i
  - constructs the pair (v_i, n_i) and blinds it using the blinded signature scheme with blinding factor b_i; $x_i := \chi((v_i, n_i), b_i)$
  - broadcasts his blinded vote x_i over the authenticated channel, in order to get it registered
- Each signer S_l checks whether voter V_i has not already applied to register a vote:

    - if voter V_i did not, signer S_l signs the blinded vote and sends it back over the authenticated broadcast channel
    - if voter V_i did already apply to register his vote, S_l does not sign the message
  - Each signer S_l keeps a list D_l of signers that he finds dishonest. He adds a signer S_d to the list if and only if:
    - S_d does not (properly) sign a vote when he had to
    - S_d signs a vote from a voter that already registered a vote
  - Voter V_i has at this point received m signed (blinded) votes, where $k \leq m \leq 2k-1$. He unblinds these and constructs his ballot: $(\sigma_{S_1}(v_i, n_i), \sigma_{S_2}(v_i, n_i), \ldots, \sigma_{S_m}(v_i, n_i))$.

DEADLINE

IDENTIFICATION

  - Each signer S_l broadcasts (over the authenticated channel) his list D_l containing the signers that he has found dishonest.

DEADLINE

VOTING

  - Each voter V_i removes the votes signed by any of the signers S_l that are on at least k lists D_j containing dishonest signers.
  - Each voter V_i broadcasts his ballot over the anonymous channel.

TALLYING

  - Everyone can tally the votes, since they are sent unencrypted and unblinded over the anonymous broadcast channel.

2.2.2 Analysis

Announcement phase

Voters that join in the PREPARATION and REGISTRATION phases are likely to have missed the initial announcement of the voting session. To inform potential new voters that a voting session is taking place there are several options. One option can be that the initiator of the voting session repeats the announcement at a regular interval. The main problem that needs to be handled here is the lack of knowledge of the joining voter: he does not know the threshold value nor the identity of the initiator. This makes it for a joining voter impossible to determine whether an announcement is fake or not. In order to try to prevent potential damage we can let a group of at least k voters respond to a fake announcement.

The deadline between the ANNOUNCEMENT and REGISTRATION phase is needed to get an initial number of voters upon which the threshold value k can be computed. In some cases the maximum number of voters is known from the start of the voting session, it is then possible to remove the deadline.

Preparation phase

In the PREPARATION phase, the threshold value k is calculated according to some public formula (or rule of thumb). However it is possible that the threshold value is set too low if a voting session becomes very popular (when many new voters join). This can be prevented by setting a maximum number of voters and to use this number to calculate the threshold value. The number of signers is set to $2k-1$ in order to take the dishonest voters into account. If the maximum number of dishonest voters $(k-1)$ are all in the group of signers, there still remain k honest voters to

construct a proper threshold signature. This way $\mathcal{O}(nk)$ messages are needed to sign n messages, which is optimal in a threshold signature scheme.

Unfortunately the group of signers needs to be static in order to protect privacy. Ideally, one would want that any subset of voters could sign a given blinded vote. The problem when a vote is signed by a dynamic group is that this leaves a footprint on the vote. Consider the case where a dynamic signing group is used. If the signatures are sent authenticated, every voter can observe which blinded vote corresponds to which set of voters. When the ballots are cast over the anonymous broadcast channel, the subset of ballots containing a specific set of signatures can be found. When the signatures are sent over an anonymous channel (encrypted using the receiver's public key) a signer can sign just a single blinded vote without being noticed. When the ballots are cast he can then identify a single ballot. Since the privacy property can not be guaranteed when using a dynamic group of singers we use a static group. Deciding who the members of this group are can be done in various ways: for example by asking voters to volunteer or by using some one-way hash function that decides who is a signer and who is not.

Registration phase

Each blinded vote has to contain a nonce, this is done to make each vote unique and to provide individual verifiability. If the nonce was left out of a blinded vote, two ballots containing a vote for the same candidate can not be distinguished, which is a problem since ballots are sent anonymously. Individual verifiability is contradictory to receipt-freeness [Cet08]. But even if we would be able to remove the nonce here, receipt-freeness still would not be possible to achieve. This is because the blinding factor can also be used as a receipt.

Blinding is used in combination with anonymous broadcasting to remove the link between a voter and his vote. It also provides a commitment here, because the actual vote is included in the blinded message. A blinded message needs to be sent over an authenticated channel, such that a signer can check whether the voter has already applied for registration or not. If a voter did not already apply to register his vote, at least k honest signers sign his blinded vote. If he did already apply to register his vote, it is possible that $k-1$ dishonest signers do not care and still send a signature but this is not enough for a vote to be counted.

Among the set of $2k-1$ signers there can be at most $k-1$ dishonest ones. These need to be identified because they can for example mark votes (as described earlier in this section). In order to do that each signer keeps a list of dishonest signers. A signer S_i puts a signer S_j on the list if and only if: (1) S_j signs a vote from a voter that already registered a vote, or (2) S_j does not (properly) sign a vote corresponding to a voter that did not register yet. Both scenarios can be observed by all voters since all signatures are broadcast authenticated. Scenario 1 can be considered unharmful since the voter can never get enough votes to legitimize a second ballot. Scenario 2 is more serious since then signers can mark a vote.

The deadline between the Registration and the Identification phase is needed to construct complete dishonesty lists. If this deadline would be omitted it is possible that there are votes or signatures sent after some signers have already sent their dishonesty lists, which can make these lists incomplete.

Identification phase

In the Identification phase each signer broadcasts his list of dishonest signers. If a signer S_i is on k or more lists, he has to be dishonest. Since there are at most $k-1$ dishonest voters it is not possible that a voter is falsely marked as dishonest voter. The deadline between the Identification and the Voting phase is needed to know make sure that no voter sends his ballot before all dishonesty lists are made public.

Voting phase

When a certain signer is on at least k dishonesty lists, then all voters have to remove their votes signed by that signer (if it exists) from their ballots. This is because signer S_i could have

(or has, if type 1 offences are discarded) marked votes. Once all signatures of dishonest voters are removed, all ballots look uniform on the outside (except for the nonces and votes). Once the ballots are 'cleaned up' (when all signatures of dishonest signers are removed) they can be broadcast anonymously.

Tallying phase

Since the votes are broadcast unblinded, everyone can tally the votes. Note that double ballots (ballots containing the same (*vote, nonce*) pairs) are only counted once. Ballots containing something else than the right number of unique signed (*vote, nonce*) pairs are not counted at all.

Time analysis

An overview of how the phases are separated by deadlines (denoted by ||), and when voters can still join the protocol:

ANNOUNCEMENT	PREPARATION REGISTRATION	IDENTIFICATION VOTING TALLYING
initial voter group is set up	*new voters can still join*	*no new voters can join anymore*

As can be seen in the overview above, the on-line property is satisfied for a large part. Up until the IDENTIFICATION phase new voters can still join the protocol.

The efficiency of the protocol can slightly be increased by creating c parallel, non-overlapping, signing groups. Each group then signs the votes for a certain group of voters (distributed uniformly over the signing groups by for example a hash function). The privacy property is not satisfied fully in this case since the ballots are not all uniform on the outside now.

2.2.3 Conclusion

The analysis in the previous section provides arguments to support that this protocol satisfies the following properties:

- privacy
- individual verifiability
- unicity
- universal verifiability

Properties that are not satisfied:

- receipt-freeness
- coercion-resistance

Coercion-resistance is not satisfied for the same reason as receipt-freeness.

Partially satisfied:

- fairness
- on-line property
- walk-away property

Fairness is only partially satisfied since the votes do not need to be disclosed at the same time. Based on already cast votes, a voter can decide to not cast his vote, however he can not change his vote at that point. The on-line property is partially satisfied since voters can join until the IDENTIFICATION phase. The walk-away property is partially satisfied since after a voter has cast his vote he can walk-away. He does not have to wait for the TALLYING phase.

2.3 A second voting protocol based on shuffling

This second protocol is also a protocol for regular mobile devices. The general idea of this protocol is to remove the link between a voter and his ballot by repeatedly shuffling the set of ballots. The ballots are initially encrypted using multiple layers of encryption, where one layer is removed in each shuffling round. By using $k+1$ shuffling rounds, it should be impossible to obtain the keys for all rounds. After the last shuffling round, a list remains that contains (*vote*, *nonce*) pairs encrypted with a threshold encryption key. The corresponding private decryption key is then made public, giving all voters the opportunity to tally.

2.3.1 The protocol

Announcement

- The initiator announces the voting session, including the candidates.
- All voters that want to participate reply by sending a message over the authenticated channel.

Deadline

Preparation

- A (k, n) threshold encryption scheme is set up.
- This scheme is used to generate $k+1$ public threshold encryption keys $K_1, K_2, \ldots, K_{k+1}$ for the shuffling rounds.
- Furthermore the scheme is used to generate another public threshold encryption key *K'* for the encryption of the vote itself.

Voting

- Each voter V_i:
  - picks a vote v_i and generates a nonce $n_{i'}$ in order to construct the ballot $x_i := \{|(v_i, n_{i'})|\}_{\mathrm{K}'}$
  - computes $k+1$ nonces $n_{i_1}, n_{i_2}, \ldots, n_{i_{k+1}}$
  - creates his "packed" ballot:

$$p_i := \{|\left(\{|\cdots\{|\left(\{|(x_i, n_{i_{k+1}})|\}_{K_{k+1}}, n_{i_k}\right)|\}_{K_k}\cdots|\}, n_{i_1}\right)|\}_{K_1}$$

  - broadcasts the packed ballot p_i

Deadline

Shuffling

- A list of packed ballots $P_0 := (p_1, p_2, \ldots, p_m)$ is created.
- The following is repeated $k+1$ times, where shuffler F_j is a different voter each round (and where j denotes the round number $(1 \leq j \leq k+1)$).
- A shuffler F_j:
  - announces that he takes care for the shuffling in round j
  - receives, when he has not shuffled before, at least k valid parts of the private decryption key PrK_j
  - constructs private key PrK_j
  - decrypts all ballots $p_i \in P_{j-1}$ to (p'_i, n_i)
  - drops the nonces from the pairs (p'_i, n_i) and constructs the list P_j, containing all p'_i in shuffled order

* broadcasts P_j

TALLYING

- *At this point a list P_{k+1} is known, containing unpacked ballots of the form $\{|(v_i, n_i)|\}_{K'}$.*
- Each voter broadcasts his part of the private decryption key PrK'.
- When k valid parts are known, the private decryption key PrK' can be computed.
- All votes can be opened.

2.3.2 Analysis

Preparation phase

The protocol contains $k + 1$ shuffling rounds, which is the minimum number of rounds needed to support privacy. When only k rounds would be used a honest shuffler can reconstruct the link between a voter and his ballot when $k - 1$ dishonest shufflers leak their round's decryption key. For this reason it is required in this protocol that the number of honest voter is $k + 1$.

Voting phase

In the construction of the "packed" ballot $k + 1$ nonces are used. These are used for obfuscation purposes, that is to hide the link between ballots in the input and output lists in a certain round of the shuffle. Because the shuffler drops the nonces in the shuffling process, it is not possible to re-encrypt ballots from the output list in order to get ballots from the input list. After the shuffling is completed a list containing encrypted (*vote*, *nonce*) pairs remains. When the last shuffling round was performed correctly, the decryption key can be made public such that every voter can tally.

Shuffling phase

Since there are at least $k + 1$ honest voters, there are always enough voters to construct a correct shuffle. Each voter can only perform a single shuffling round, since for a second round he will not receive enough parts of the private decryption key.

There are two types of offences for a dishonest shuffler F_i: (1) shuffler F_i replaces some votes by self-made other votes or by garbage, or (2) shuffler F_i leaks the round's private key, or leaks (p'_i, n_i) pairs. First, let us look at the first offence. A voter can notice when his encrypted ballot is not in the output list of a certain shuffling round, when he notices this, he makes an accusation over the anonymous broadcast channel. He does this by sending the pair (p'_i, n_i) that should have been the result of the decryption. Note that only the voter who casted the vote and the shuffler know this pair. All other voters can re-encrypt the pair (p'_i, n_i) with the round's public key, in order to see if that pair was (in encrypted form) in the round's input list.

Unfortunately, a voter does not always stay anonymous when putting an accusation on the anonymous broadcast channel. When the vote of a certain voter in the first round is not properly decrypted, that voter needs to put an accusation over the anonymous broadcast channel by announcing the pair (p'_i, n_i). From that pair, p'_i contains his encrypted ballot for the second round. Then his identity can also in the second round be coupled to a ballot in the input list of that round. This can follow the voter for a most $k - 1$ rounds, until a honest voter correctly decrypts his ballot. The other type of offence does not pose a problem since there are always at least two shuffles that are performed correctly.

For shufflers who only committed offences of type 2 no extra measures need to be taken since their dishonesty is handled by the number of shuffling rounds. Shuffling rounds that contain an incorrect shuffle (offence 1) need to be done over. Multiple voters get to know the decryption key then, but this poses no problem since the key could have been leaked anyways. Voters themselves can also be dishonest by constructing an invalid packed ballot. When a shuffler comes around

such ballot he publishes the result of the decryption and then removes the ballot. Other voters can re-encrypt the result in order to verify whether the claim was valid.

Time analysis

An overview of how the phases are separated by deadlines (denoted by ||), and when voters can still join the protocol:

ANNOUNCEMENT	PREPARATION VOTING	SHUFFLING TALLYING
initial voter group is set up	*new voters can still join*	*no new voters can join anymore*

All voters that join in the ANNOUNCEMENT phase can participate in setting up the threshold encryption scheme. Voters that join in the PREPARATION or VOTING phase cannot co-operate in setting up that scheme anymore, but they can vote and even participate in the shuffling process. There is a deadline before the SHUFFLING phase because the shuffle needs to be performed over a fixed list of votes; no new votes can be added during the shuffling. Voters can in principle leave after they have cast their votes, but there should remain at least $2k$ voters to complete the shuffling process. When a voter wants to verify that his vote is counted correctly, he has to stay until the tally has been produced.

2.3.3 Conclusion

The analysis in the previous sections provides arguments to support that this protocol satisfies the following properties:

- privacy
- individual verifiability
- unicity
- fairness

The unicity property is not discussed in earlier sections because it is satisfied trivially. Voters can publish multiple votes, but this is noticed since they are sent authenticated.

Properties that are not satisfied:

- receipt-freeness
- coercion-resistance

Partially satisfied:

- universal verifiability
- on-line property
- walk-away property

Universal verifiability is only partially satisfied since the final tally can be verified by every voter, but during the shuffling process a voter can only follow his own vote. The on-line property is partially satisfied since new voters can join until the SHUFFLING phase. The walk-away property is partially satisfied since voters can leave after they have cast their vote. However by doing this, possible fraud with their ballot remains undetected. The walk-away property would be fully satisfied when the shufflers need to provide a zero-knowledge proof to prove that the shuffle has been performed correctly. This also removes the need for an anonymous broadcast channel. The zero-knowledge proof has to prove that for two given sets S and S', set S' contains the projection of the first items of the tuples that are formed by decrypting all elements in set S. Unfortunately, we were not able to set up such a proof procedure.

2.4 Ad-hoc voting using trusted devices (1)

In this section we describe a protocol that uses trusted devices for the purpose of voting. In our setting, a *trusted device* is a device that is needed to participate in a voting, where the users (the voters) place trust in the fact that the device works as specified. The idea of trusted devices comes from a paper by Avoine et al. [AGGV05], where trusted devices are used to solve the fair exchange problem. Depending on the context, it is possible that each voter has its own trusted device or that multiple voters have to share a trusted device. Here we study a protocol where a trusted device is shared among multiple voters. When a device is not shared between multiple voters, the protocol works as if each voter has its own trusted device.

We study how a trusted device can be used in an environment with very strong assumptions on the device itself, in the next section we present a protocol that can be used in an environment with weaker assumptions. In this section we assume that a voter can not read the internal memory of the trusted device and that he can not illegally alter data in the internal memory. Since we have such strict assumptions on the environment, any voter can be dishonest in this protocol.

2.4.1 The protocol

ANNOUNCEMENT

- The initiator announces the voting session, including the candidates.
- The initiator starts a group key exchange protocol (for example one based on the basic Diffie-Hellman key exchange protocol [BCP07]) in order to set up a shared secret key k.
- All active trusted devices participate in the key exchange protocol.

DEADLINE

VOTING

- *At this point, the shared secret key k has been established.*
- Say a voter V_i registers himself at trusted device D_l.
- Trusted device D_l:
  - checks whether voter V_i has not already cast a vote before, by looking in his local list of ballots
  - denies voter V_i access when voter V_i did already cast a vote before
  - logs voter V_i in when voter V_i did not cast a vote earlier
  - waits for voter V_i to input his vote v_i
  - constructs the ballot $b_i := \{(V_i, v_i)\}_k$
  - broadcasts ballot b_i and adds it to its local list of ballots
  - logs out voter V_i
- Upon receiving a ballot $\{(V_i, v_i)\}_k$, all trusted devices immediately decrypt this and add the decrypted ballot to their local lists of ballots. *This step is interleaved throughout the entire* VOTING *phase.*

DEADLINE

TALLYING

- Every trusted device sanitizes its local list of votes (that is, it removes the identity of the voters from the list) and then publishes it.

2.4.2 Analysis

Announcement phase

In this protocol all messages containing votes are encrypted with a shared secret key. The encryption is needed because otherwise an eavesdropper would be able to read the votes, thereby violating the fairness and privacy properties. Since all trusted devices are honest entities in the protocol, we can use a single shared secret key.

Since the internal memory of the trusted devices can not be read, it is possible to use a static shared secret key to encrypt the messages in the protocol. This way, the group key exchange protocol can be removed from the voting protocol, including the deadline that was introduced for it. When a static key is used, a nonce needs to be added to each ballot, to prevent an eavesdropper to extract information from two voting sessions with overlapping sets of candidates.

Ideally this approach works as well as the approach using the group key exchange protocol. However, might the unfortunate situation occur that the key gets disclosed, the trusted devices are rendered useless since other devices can eavesdrop on the messages sent between trusted devices. It is also possible to adapt a middle course here by using a shared key that is semi-dynamic, then the shared key does not change unless the group key exchange protocol is triggered manually.

Voting phase

When the protocol is used in a setting where trusted devices are not shared it is possible to leave out the voter registration step, since the trusted device then identifies the voter. The ballots in this protocol are cast immediately. This has two reasons: to prevent voters from voting twice and to prevent dishonest voters from destroying votes by switching off a device. To prevent a voter from voting twice all local lists of ballots need to be kept synchronized. These lists can be seen as one global list since ballots are cast as soon as a voter has entered its vote and since received ballots are processed immediately.

Time analysis

It is possible for trusted devices to later join the protocol, for example when it was not yet powered on at the start of the voting session. A trusted device can join the protocol in any phase, however it can only participate in setting up the secret key if it joins before the deadline between the Announcement and Voting phases. When a new trusted device joins it announces this, such that some other trusted device can inform it about the state of the voting session. Therefore the two devices first generate a shared secret key to communicate privately. Then the following information needs to be sent to the joining device (encrypted with their secret key): information on the voting session and its candidates, the globally shared secret key and the current list of ballots.

It is no problem for a trusted device to leave the protocol early, this can for example be the case when a voter switches off the trusted device or when the trusted device suffers from some (hardware) failure. If the ballot of a voter that is logged in has not been cast, the voter is still able to vote on another device (or on the same device after it is powered on again). Since the list of votes is kept globally, at least one trusted device has to be on-line at any time, otherwise it is possible that votes get destroyed. When a trusted device that has left the protocol re-enters the protocol, the same procedure as with the new trusted device can be carried out. Nothing needs to be done further since the trusted device does not contain any information that has not been synchronized.

2.4.3 Conclusion

The analysis in the previous sections provides arguments to support that this protocol satisfies the following properties:

- privacy

- unicity
- fairness
- receipt-freeness
- walk-away property
- on-line property

Properties that are not satisfied:

- individual verifiability
- universal verifiability
- coercion-resistance

There is no verifiability for a voter, because he does not know what is sent by the trusted devices. The coercion-resistance property is also not satisfied, because someone can coerce a voter to give (or sell) his login code. Individual verifiability can be satisfied instead of receipt-freeness by including a nonce on each ballot.

2.5 Ad-hoc voting using trusted devices (2)

In this protocol we study how a trusted device can be used in an environment in which the full internal memory of a trusted device can be read be any user. We however need to add some extra assumptions here. The first is that decryption is done in a black box (for example through the use of a smartcard): a voter can not extract the private decryption key of a trusted device. Incoming messages are however immediately decrypted and the black box decryption can thus not be used as a way of caching encrypted values. Furthermore we assume that a set of at least k (the threshold value) trusted devices are active throughout the entire protocol and that a collective of dishonest voters can not get access to k trusted device at the same time.

2.5.1 The protocol

Announcement

- The initiator sets a maximum number of voters n_v.
- The initiator announces the voting session, including the list of n_c candidates. For each candidate there is a corresponding integer encoding that is used in the ballots (candidate e_i is encoded by the integer $(n_v + 1)^i$).
- The set $\mathcal{A}$ of all m active trusted devices is set and based on a threshold value k ($k \leq m$).

Voting

- Say a voter V_i enters a login code l_i' onto a trusted device D_j.
- Trusted device D_j:
  - checks whether $\#(l_i')$ corresponds with the hash of voter V_i's real login l_i stored in its internal memory
  - denies voter V_i access when those two do not correspond and removes personal information of voter V_i from its memory (that is V_i, l_i)
  - continues when those two do correspond
  - checks whether voter V_i is in the list of registered voters
  - denies voter V_i access if he is already on the list of registered voters and removes personal information of voter V_i from its memory (that is V_i, l_i)

* logs voter V_i in when he is not on that list
* broadcasts voter V_i's identity in order to register him and adds the identity to its local list of registered voters
* waits for voter V_i to input his candidate of choice e_l
* generates $k-1$ random coefficients $a_{i_{k-1}}, \ldots, a_{i_1}$ to form voter V_i's polynomial

$$q_i(x) := a_{i_{k-1}} x^{k-1} + a_{i_{k-2}} x^{k-2} + \cdots + a_{i_2} x^2 + a_{i_1} x + (n_v + 1)^l$$

* broadcasts for each trusted device $D_p \in \mathcal{A}$ $(p \neq j)$: $\{|q_i(p)|\}_{\text{PubKey}(D_p)}$
* adds $q_i(j)$ to its own count C_j
* removes all personal information of voter V_i from its memory (that is V_i, l_i, $q_i(x)$, $q_i(j)$, $a_{i_{k-1}}$, ..., a_{i_1}, l, e_l, all messages $\{|q_i(\cdot)|\}_{\text{PubKey}(\cdot)}$, but not V_i's identity in the list of registered voters)
* logs out voter V_i

– Upon receiving the identity of a voter V_i, all trusted devices immediately add this value to their local lists of registered voters.

– Upon reception of a message $\{|q_i(j)|\}_{\text{PubKey}(D_j)}$, trusted device D_j decrypts this message and adds $q_i(j)$ to its count C_j.

DEADLINE

TALLYING

– All trusted devices D_j who have been active throughout the entire protocol publish their count C_j.

– When k or more values C_i are known, the final polynomial $q_F(x)$ can be determined.

– The encoded tally T can be computed: $T := q_F(0)$.

– The encoded tally T can be opened by any trusted device:

* number of votes for candidate e_{n_c-1}: T div $(n_v+1)^{n_c-1}$; $T := T \mod (n_v+1)^{n_c-1}$
* number of votes for candidate e_{n_c-2}: T div $(n_v+1)^{n_c-2}$; $T := T \mod (n_v+1)^{n_c-2}$
* ...
* number of votes for candidate e_1: T div (n_v+1); $T := T \mod (n_v+1)$
* number of votes for candidate e_0: T

2.5.2 Analysis

In this protocol voter registration is done separately from sending the actual ballot corresponding to that voter, this is done to hide the link between a voter and its ballot. We have chosen again for a global list (which can be implemented as synchronized local lists) that contains the identities of the registered voters.

Since voters can read the full internal memory of a trusted device it is not possible to send plain ballots. Encrypting ballots is also not possible since there is a link between each encrypted ballot and a voter. Then, when the decryption key is disclosed, the vote of that voter is revealed. Therefore we use a *(k, n) secret sharing scheme* (based on one by Shamir [Sha79]), where a message is divided into n shares and where k different shares are needed to reconstruct the shared message.

The threshold value k is a trade-off between two factors here: (1) the number of trusted devices that are on-line throughout the entire protocol, and (2) the number of trusted devices a collective of dishonest voters can get a hold of at the same time. Here we use a secret sharing method method described by Benaloh [Ben06], where several small secrets (the individual ballots) together form one big secret (the tally). We combine this with the vote-as-integer representation as described by Groth [Gro04]. When we combine these two techniques, it is not necessary that each individual ballot is opened. Only the tally gets opened in the TALLYING phase, the individual ballots thus

get lost in the final tally. When no trusted device can be powered down, it is possible to replace the secret sharing scheme by an easier scheme (for example one mentioned in the introduction of a paper by Karnin et al. [KGH83]) since we do not have to restrict our choice to (k, n) secret sharing schemes anymore.

Time analysis: basic approach

New trusted devices can join the protocol at any time. However they can only participate in storing parts of the secret when they have been active in the ANNOUNCEMENT phase. Once a trusted device has left the protocol and returns some time later, it can not participate (anymore) in storing parts of the secret. This is because it might have missed values that needed to be added to its count. In order to allow new voters to vote on the recovered trusted device, another trusted device sends the recovered trusted device the current list of registered voters.

Time analysis: extended approach

In this section we present an extended approach to the protocol, which adds two techniques to the protocol: cloning and caching messages. Furthermore the crash recovery procedure is improved a lot: in the basic approach a recovered trusted device was only treated as a new trusted device. Using these improvements we can relax one of the strongest assumptions, that is that a set of at least k (the threshold value) trusted devices need to be active throughout the entire protocol in order for the protocol to finish. We now only need at least k distinct points on the polynomial in the TALLYING phase. Furthermore we assume that the contents of a trusted device's memory are preserved when it suffers from a crash. For the caching part the trusted devices need to have some free storage space. If this is not available, the cloning part can still be used.

It should be noted that this extended approach does not result in the satisfaction of more properties. The main improvement of the approach is that it is more robust against (hardware) failures and collectives of dishonest voters, thereby improving the chance of a successful voting session.

Cloning A new trusted device can join the protocol at any time, however it only gets assigned an individual point on the polynomial in the secret sharing scheme if it is active in the ANNOUNCEMENT phase. All new trusted devices that join the protocol at a later phase get to work as a clone. The idea of cloning is as follows: a new trusted device D_n joins the protocol after the ANNOUNCEMENT phase. Then an existing trusted device D_e informs the new trusted device of the current voting session. Therefore trusted device D_e sends trusted device D_n the following information, encrypted with $PubKey(D_n)$:

- information on the voting session and its candidates
- the current count C_e of the value on the polynomial at point e
- its private key, $PrivKey(D_e)$

This enables trusted device D_n to act as trusted device D_e. The purpose is thus to let multiple devices keep track of the same point on the polynomial, such that the protocol is more robust against hardware failure and dishonest voters. Note that the decryption key has to be kept invisible for the voters. This can for example be done by lending out the smartcard (if one is used), such that the key can be transferred. This cloning technique even has a positive impact on the possibilities of a collective of dishonest voters: it becomes harder to get access to k different points on the polynomial since not every trusted device needs to identify a unique point.

Caching messages To make the protocol more robust against trusted devices that are powered down (being switched off by a dishonest voter or stopped because of a (hardware) failure) we can cache unreceived messages of the form $\{|q_i(j)|\}_{PubKey(D_j)}$. Before we explain how this works we separate two cases in which it can be used:

1. Every trusted device has enough memory to store all messages that are broadcast during a single voting session.
2. A trusted device can not store all message that are broadcast during a single voting session.

Introduction and case 1 Since we only want to cache unreceived messages we need some method to detect whether a message has been received or not. An easy method is to send acknowledgements for every received message. Suppose a trusted device D_j is powered down, then all trusted devices cache for all i ($1 \leq i \leq k$) messages of the form $\{|q_i(j)|\}_{PubKey(D_j)}$. Caching all those messages poses no risk to the protocol, since all messages that are cached are encrypted.

Once trusted device D_j (or one of its clones) returns in the protocol, it announce this. Then one trusted device sends all D_j's cached messages to D_j (or to the clone). From this point D_j's messages do not need to be cached anymore and every trusted device can delete the messages it has cached for D_j. Now there are two cases: either D_j's stored count contains the sum of all messages up to the first cached message, or there is a gap (introduced when there was still an active clone when D_j left). In the first case D_j can just decrypt the received messages, add these to its count and continue business as normal. In the second case D_j sets its counter to the sum of the decrypted values. Then, when its clone that last left returns, the two synchronize their values. Then they are both back in business as normal. Whenever a trusted device (re-)joins, all messages cached for other trusted devices also need to be transferred. Using this caching method, the protocol can still end if at a certain point in time there is only one trusted device active, but only if there are k trusted devices active that all hold a different share of the secret.

Case 2 A serious disadvantage of this caching approach is that a trusted device has to have enough storage to cache for all i, j messages of the form $\{|q_i(j)|\}_{PubKey(D_j)}$. A possible optimization is to let just s trusted devices cache messages for a certain trusted device, instead of letting all trusted devices do this. While this approach is in the worst case still as bad as the old one, it can be useful on average. When a certain trusted device's storage is full, that trusted device broadcasts a message that the protocol has to halt. Then the trusted device offers the other trusted devices the set that includes all cached votes for a single point on the polynomial. Any active trusted device that has enough free space to store that set can then take over the set and the protocol can be resumed. If no active trusted device is able to take over the set a message can be displayed on all active trusted devices stating this, until enough trusted devices (either new or recovering) are added that can take over the set.

Unfortunately this can be exploited by dishonest voters. If they can prevent that trusted devices (re-)join the protocol, the protocol can not continue. To prevent this, we can add a timeout: when after a certain time t there have not (re-)joined enough trusted devices, other measures need to be taken. For example that active trusted devices can 'give up' some count C_i on the polynomial according to some heuristic. By 'giving up' we mean here that all trusted devices remove all messages they cached for trusted device D_i. The heuristic on which a certain trusted device is chosen can be for example: the trusted device for which the largest number of messages are cached or the trusted device that has the lowest chance to return.

Note that this method of giving up points can only be applied a number of times. For the protocol to finish there have to be at least k distinct values C_i known in the TALLYING phase. When a certain value C_i has been given up and trusted device D_i (or one of its clones) returns in the protocol, another trusted device informs it that its value C_i has been given up. Trusted device D_i is then handled as a new trusted device and becomes a clone for another trusted device.

Improvement on caching To optimize the space consumption and the time it takes to re-join the protocol we can use homomorphic encryption for the encryption of the shares. *Homomorphic encryption* is a form of encryption where the contents of an encrypted message can be manipulated by performing an algebraic operation on the encrypted value. We can use this to sum up the contents of encrypted messages, that way only a single value needs to be sent to a re-joining

trusted device instead of a set of values. In order to do this the homomorphic encryption scheme needs to have the following property (where $E(m)$ denotes that message m is encrypted using some encryption scheme and where $\otimes$ is some algebraic operation): $E(m_1) \otimes E(m_2) = E(m_1 + m_2)$. Fortunately a number of schemes have this property [OU98, DJ01, Pai99, BT94].

2.5.3 Variant: a veto protocol

We can easily adapt this protocol to a veto protocol. A *veto protocol* is a voting protocol where voters can only vote against or in favor of a proposal, the result of the protocol being that at least one voter voted against ('vetoed') or that all voted in favor. The main difference with a voting protocol with two candidates ('in favor' and 'against') and a veto protocol is that in a veto protocol it has to be kept secret how many voters vetoed. In literature, there exist only a small number of veto protocols [KY03, Gro04, Bra06, HZ06, BT07], therefore we think it is interesting to show that our protocol can be used as a veto protocol in an ad-hoc setting without any major changes. From the trusted device's point of view, the following in changed. Instead of receiving a candidate e_l as the vote of voter V_i, the trusted device receives a bit representing a 'in favor' or 'against' vote. Once the trusted device has received this bit it computes the voter's polynomial as follows:

$$q_i(x) := a_{i_{k-1}} x^{k-1} + a_{i_{k-2}} x^{k-2} + \cdots + a_{i_2} x^2 + a_{i_1} x + e$$

Here the value e depends on the value of the vote. For an 'in favor' vote, the trusted device uses $e := 0$ in the computation of the polynomial. In case of an 'against' vote the trusted device generates a random r_e and uses $e := r_e$ in the computation of the polynomial.

In the TALLYING phase the opening of the tally can be skipped: the only thing that needs to be checked is whether $q_F(0) = 0$. No voter vetoed if and only if $q_F(0) = 0$, but if only one voter vetoed he can see that he is the only one who did so. This problem is also present in for example the veto protocols of Hao and Zieliński [HZ06] and Groth [Gro04], where it is fixed by introducing a trusted third party.

2.5.4 Conclusion

The analysis in the previous sections provides arguments to support that this protocol satisfies the following properties:

- privacy
- unicity
- fairness
- receipt-freeness
- walk-away property
- on-line property

Properties that are not satisfied:

- individual verifiability
- universal verifiability
- coercion-resistance

There is no verifiability for a voter, because the voter can not observe all trusted devices at the same time (in practice). The coercion-resistance property is also not satisfied, because some voter can coerce a voter to give (or sell) his login code. This scheme does not allow a trivial way to add individual verifiability by for example including a nonce.

Chapter 3

Verifying properties of the poor man's protocol

In this chapter we verify some properties of the poor man's protocol. We do this by constructing a formal model of the protocol on which several techniques can be applied to verify specific properties. The protocol itself has already been discussed in detail in Section 2.2, therefore we start straight away with formalizing the protocol.

For point-to-point protocols there exists a notation to describe communication in a protocol [SNS88]. We illustrate this notation using an example. Say we have a protocol with two entities: a client C and a server S. In the protocol the client sends a message m to the server and the server replies with the hash of the message, computed using a hash function $\#$. Formally we can denote this as follows.

1. $C \rightarrow S \quad : \quad m$
2. $S \rightarrow C \quad : \quad \#(m)$

For protocols using broadcast communication there is no such standard notation to formalize the protocol. Therefore we introduce our own notation, based on the notation for point-to-point protocols, and we explain it by example: the formal notation of the poor man's protocol can be found in Figure 3.1.

Voter V_i $(1 \leq i \leq n)$

1. $\quad \Leftarrow I \quad : (voting, candidates)$
2. $V_i \Rightarrow \quad : \chi((v_i, n_i), b_i)$
3. $\quad \Leftarrow S_l \; : \sigma_{S_l}(\chi((v_i, n_i), b_i)) \qquad m \text{ times}, k \leq m \leq 2k-1$

DEADLINE

4. $\quad \Leftarrow S_l \; : Dishonest_l \qquad m' \text{ times}, k \leq m' \leq 2k-1$
5. $V_i \; ?\Rightarrow \quad : (\sigma_{S'_1}(v_i, n_i), \ldots, \sigma_{S'_h}(v_i, n_i)) \qquad h \geq k$
6. $\quad ?\Leftarrow V_j \; : (\sigma_{S'_1}(v_j, n_j), \ldots, \sigma_{S'_h}(v_j, n_j)) \qquad \text{for all } j,\ 1 \leq j \leq n$

Signer S_l $(1 \leq l \leq 2k-1)$

1. $(\quad \Leftarrow V_i \; : msg$
2. $S_l \Rightarrow \quad : \sigma_{S_l}(msg))^{\leq n}$

DEADLINE

3. $S_l \Rightarrow \quad : Dishonest_l$

Figure 3.1: Poor man's protocol in formal notation.

The symbols used in this notation are defined as follows.

$A \rightrightarrows$	$: m$	A sends message m over the authenticated broadcast channel
$A \; ?\rightrightarrows$	$: m$	A sends message m over the anonymous broadcast channel
$\leftleftarrows B$	$: m$	message m is received from B over the authenticated broadcast channel
$?\leftleftarrows B$	$: m$	message m is received from B over the anonymous broadcast channel

In the sending of messages, as can be seen in the example, we include only the sender, whereas in the point-to-point protocol notation both the sender as well as the receiver are included. There, $A \rightarrow B : m$ means that A sends message m to B *and* that B receives the message. In broadcast communication we have a set of receivers, where it is possible that a subset of the receivers in this set are unknown to the sender. For this reason we model the entities separately, which is also the reason for the separate send and receive actions.

Some actions in the protocol description in Figure 3.1 need to be repeated multiple times. When this is the case (or when other conditions apply) this is added in italics next to the action. In the notation for a signer S_l we see the construct $(a; b)^{\leq n}$ followed by a deadline. This means that actions a and b are executed (in sequence) a number of times until the deadline is reached, where the exact number is unknown but smaller than n.

ProVerif

A well-known tool to analyze security protocols automatically with is ProVerif [Bla01]. It has been used to prove a number of properties for different protocols, for example the FOO'92 voting protocol [KR05], the JFK key establishment protocol [ABF07] and Ferguson's electronic cash protocol [LCPD07]. ProVerif is able to prove (strong) secrecy, authentication (and some more general correspondences) and a number of equivalences between processes. Since ProVerif is especially targeted at cryptographic protocols we would have liked to use it to analyze some of our protocols, but unfortunately ProVerif is not suited for this.

The main reason why ProVerif can not be used is that it is too restrictive. ProVerif does not provide primitives for broadcast communication, all communication is done point-to-point. It may be possible to simulate a broadcast channel in ProVerif, but as far as we know this has not been done before and it is not straightforward. Furthermore ProVerif lacks support for loops, which is a major disadvantage since we need those to properly model the various threshold schemes we use in our protocols. This problem has recently been addressed in a paper by Ryan and Mukhamedov [MR07]. To overcome this problem they have chosen to extend the syntax of ProVerif by a for-loop construct, thereby losing the ability to automatically verify the protocol. Finally ProVerif also lacks support for lists, which we would like to have to for example keep a list of voters who have voted. That way we can easily check if some voter wants to vote twice (note that this property is also not verified by Kremer and Ryan in their analysis of the FOO'92 protocol [KR05]).

For the above reasons we think that it is not possible to come up with a simple, parameterized model for any of our protocols. We could construct a model with a fixed number of voters and a fixed number of dishonest voters, but we think that we then need to spend more time working around ProVerif's limitations rather than modelling the protocol. When we would simplify the protocol such that it is better to model in ProVerif the verification almost becomes trivial since the interesting constructs, which are needed for the satisfaction of a number of properties, are the hardest to model. Therefore we have chosen to model our protocols in mCRL2 [GMR$^+$06], which provides all the constructs we need to model our protocols and includes tools with which we can perform model checking.

In the next section we start with an introduction to the mCRL2 specification language. Then we present in Section 3.2 a first, intuitive model of the protocol with only honest voters. For reasons of efficiency we linearize this model ourselves, which we explain in Section 3.3. In Section 3.4 we explain in detail how we add dishonest voters to the model and to optimize the model even further we introduce the notion of strong synchronicity in Section 3.5. Finally, in the last section we use these models for the formal verification of some properties of the protocol.

3.1 The mCRL2 specification language and toolset

The mCRL2 specification language [GMR+06] can be used to describe the behaviour of distributed systems and therefore we can use it to describe our voting protocols. The language is a direct successor of the μCRL specification language [GP95], which is based on the Algebra of Communicating Processes (ACP) [BW90]. The mCRL2 language includes support for abstract data types on top of a basic timed process algebra.

In this section we give a short introduction to the mCRL2 specification language and to the mCRL2 toolset. In this introduction we do not consider timed processes since we do not use time in the models of our protocols. Furthermore we do not give the formal semantics and axioms of the language, but we rather give an informal introduction using various examples.

Actions

Every process consists of a sequence of (either observable or unobservable) actions, therefore we can see actions as building blocks for processes. The process that can not do any action (nor terminate) is called deadlock and is denoted by δ. In the mCRL2 language actions can contain data. Consider for example a process *cnt* that is used to count how often a certain event occurs. Two actions associated with this process are action `reset`, which resets the counter to zero, and action `inc`, which increases the counter by a given value. These actions can be declared as follows.

act *reset*;
inc : $\mathbb{N}$;

This declares the two actions, action `reset` without parameters and action `inc` with a parameter of type (also called *sort*) natural numbers. We can now use actions like *reset*, *inc(1)* and *inc(8)* in the specification of a process. Actions in the mCRL2 language are atomic, which means that they have no duration. From this it follows that for two actions `a` and `b`, action `a` must happen before action `b` or vice versa: they can not overlap. If two actions `a` and `b` happen at the exact same time, this is denoted by the multi-action `a|b`. The term τ represents the empty multi-action (also called *internal action*), which does not contain any action and is unobservable.

Since actions are the building blocks of processes, the mCRL2 language provides some constructs to combine these actions into processes, namely alternative composition and sequential composition. The sequential composition of two processes p and q (denoted by $p \cdot q$) expresses that the process first acts as process p and after termination of p acts as process q. For example the process $a \cdot b \cdot c$ denotes the process where actions `a`, `b` and `c` are performed in a row and then terminates. Sequential composition is associative, but not commutative or idempotent.

The alternative composition of two process p and q (denoted by $p + q$) expresses choice: either the behaviour of process p can be chosen, or that of process q. For example the process $a \cdot b + c \cdot d$ can do an `a` action followed by a `b` action, or a `c` action followed by a `d` action. The alternative composition is commutative, associative and idempotent. Furthermore the sequential composition right distributes over the alternative composition, but not on the left. It is possible to generalize over the alternative composition operator, which can be very useful for actions with data parameters. Consider for example a process with parameter d of sort $\mathbb{D}$, then we can write $\sum_{d:\mathbb{D}} p(d)$ for some value d of sort $\mathbb{D}$. For finite sorts $\mathbb{D}$ it is possible to replace the sum operator by a sequence of choices. For example:

$$\sum_{d:\mathbb{B}} p(d) = p(true) + p(false)$$

Using the conditional operator it is possible to express deterministic choice. For a Boolean condition c and processes p and q we can denote *if c then p else q* by $c \rightarrow p \diamond q$. It should be noted that condition c can only consist of data, it is not allowed to contain processes.

The mCRL2 language also provides recursion. In order to use recursion, processes need to

have names. For example, our counter can be specified as follows.

$$
\begin{array}{ll}
\textbf{act} & reset; \\
 & inc : \mathbb{N}; \\
\textbf{proc} & cnt(value : \mathbb{N}) = (value \approx 0) \rightarrow \sum_{n:\mathbb{N}} \cdot inc(n) \cdot cnt(n) \\
 & \qquad \diamond\, reset \cdot cnt(0) + \sum_{n:\mathbb{N}} \cdot inc(n) \cdot cnt(n + value); \\
\textbf{init} & cnt(0);
\end{array}
$$

This specification defines the process *cnt* as follows. If the value of the counter is equal to zero, the counter can only be increased by a natural number n, resulting in the counter with value n. If the value of the counter is larger than zero it is possible to reset the counter or increase the counter by n. The initial value of the counter is zero, as denoted by the line with keyword **init**.

Data types

There are a few predefined data types in mCRL2: Booleans, numbers, structured types, lists, sets and functions. To declare arbitrary sorts the keyword **sort** needs to be used. For example the sort *Nat*, representing the natural numbers, can be declared as follows.

$$
\begin{array}{ll}
\textbf{sort} & Nat; \\
\textbf{cons} & zero : Nat; \\
 & succ : Nat \rightarrow Nat;
\end{array}
$$

Functions can be declared using the keywords **map**, **var** and **eqn**. Using the keyword **map** the name and type of the function is specified, using **var** the variables used in the definition of the function and keyword **eqn** is used to give the equations that define the function. An example function on natural numbers is the following function *odd* on the just defined sort *Nat*.

$$
\begin{array}{ll}
\textbf{map} & odd : Nat \rightarrow \mathbb{B}; \\
\textbf{var} & n : Nat; \\
\textbf{eqn} & odd(zero) = false; \\
 & odd(succ(zero)) = true; \\
 & odd(succ(succ(n))) = odd(n);
\end{array}
$$

Structured types can be used to define more complex sorts. Consider the following definition of a tree where each node and each leaf contains a natural number. Using *recognizers isLeaf* and *isNode* we can get to know whether a value t of sort *Tree* is either a node or a leaf. If t is a leaf, we can use *projection function lval(t)* to determine the value of t.

$$\textbf{sort}\ Tree = \textbf{struct}\ leaf(lval{:}\mathbb{N})?isLeaf \mid node(left{:}Tree, nval{:}\mathbb{N}, right{:}Tree)?isNode$$

Parallel processes

The parallel operator can be used to put two processes p and q in parallel (denoted by $p \parallel q$). This means that the actions of process p are performed independently from the actions of process q, for example $a \parallel b = a \cdot b + b \cdot a + a \mid b$. It should be noted that every expression containing the parallel operator can be rewritten using the axioms to an expression without the parallel operator.

Process operators

To manipulate processes the mCRL2 language provides some operators on processes. The *communication operator* $\Gamma_C(p)$ replaces actions by a single action when their data is equal. Communications are of the form $a_1 | \cdots | a_n \rightarrow c$, where a_i and c are action names and $n \geq 2$. An example to illustrate how the operator works: $\Gamma_{\{send|recv \rightarrow comm\}}(send(7) \mid recv(7)) = comm(7)$. The *allow operator* $\nabla_V(p)$ is used to explicitly allow the multi-actions in V and thereby implicitly block all other multi-actions. An example to illustrate how this operator works: $\nabla_{\{comm\}}(comm(7) +$

$send(8)|recv(6)) = comm(7)$. The *blocking operator* $\partial_B(p)$ does the opposite of the allow operator, it explicitly blocks all multi-actions in B and thereby implicitly allows all other multi-actions. The *renaming operator* $\rho_R(p)$ is used to rename action names using the renamings in R. These renamings are of the form $a \rightarrow b$. An example to illustrate how the operator works: $\rho_{\{comm \rightarrow bcast\}}(comm(7) + a(10)) = bcast(7) + a(10)$. Finally, the *hiding operator* $\tau_I(p)$ is used to hide internal behaviour. This is done by removing the action names in I from the multi-actions. Two examples to illustrate how the operator works: $\tau_{\{a\}}(a(10)) = \tau$ and $\tau_{\{send\}}(send(7)|recv(7)) = recv(7)$.

The toolset

The mCRL2 toolset consists of a number of tools that can be divided into three categories: LPS tools, LTS tools and PBES tools. A graphical overview of the toolset is included in Figure 3.2. As can be seen in the graphical overview, the linear process is the heart of the toolset. A *linear process specification* (LPS) (also called *linear process equation* (LPE)) is a process in the mCRL2 language in a restricted form. In an LPS only a single process name can occur at the right hand side of the process definition. Furthermore, every recursive call to the process is preceded by exactly one action. Every mCRL2 specification can be transformed into an LPS. This process is called linearization and can be done using the tool *mcrl22lps*. Since LPSs have such restricted form it is easy to manipulate them. The toolset contains a number of tools to manipulate LPSs, we discuss some of these tools further on when we use them.

In order to study the behaviour of a process we can generate the state space of an LPS (when the model is finite), using the tool *lps2lts*. A *labelled transition system (LTS)* consists of a set of states that are connected using transitions. One of these states is defined as the initial state and all transitions are labelled using actions. We use the tool *ltsview* to visualize LTSs, from these visualizations it is possible to extract the global structure of the process.

The last class of tools are the PBES tools. These tools can be used for verification. Given an LPS and a modal-μ calculus formula [GW05] it is possible to construct a PBES that is equal to true if and only if the formula holds on the LPS. For now it is enough to know that a PBES is an ordered sequence of equations consisting of predicate formulae, which explicitly contain data. PBESs and some of the accompanying techniques are explained in depth in Section 4.2.

Environment

All experiments in this thesis are performed on a system with a dual core AMD Opteron 885 processor running at 2.6GHz, containing 128GB internal memory and running Fedora Core 8 with kernel 2.6.24. We have used the release branch of the mCRL2 toolset, compiled at revision 6517-shared.

3.2 An honest model

When modelling reactive systems often simplifications have to be made. Especially when one wants to be able to construct the state space or traverse all paths of a model, it is not always feasible to include every scenario. The same holds for the voting protocol we model.

The first assumption we make for our model is that the set of voters is static, which means that no voters can join the protocol after it has started. Allowing voters to join the protocol at any time would lead to an exponential blow-up of the state space. Our second assumption is that voters are able to cache messages, but that they do process them in the order they are received in. Therefore we model the broadcast channel as a queue, where each voter has its own queue. If voters are not able to cache messages, we would need a synchronization step after each message that is broadcast to see whether all voters have processed that message. Furthermore we assume that each voter handles in its own interest and therefore first sends his blinded signature before he sends his signatures of other voters blinded votes. This is not so much a complexity issue, but

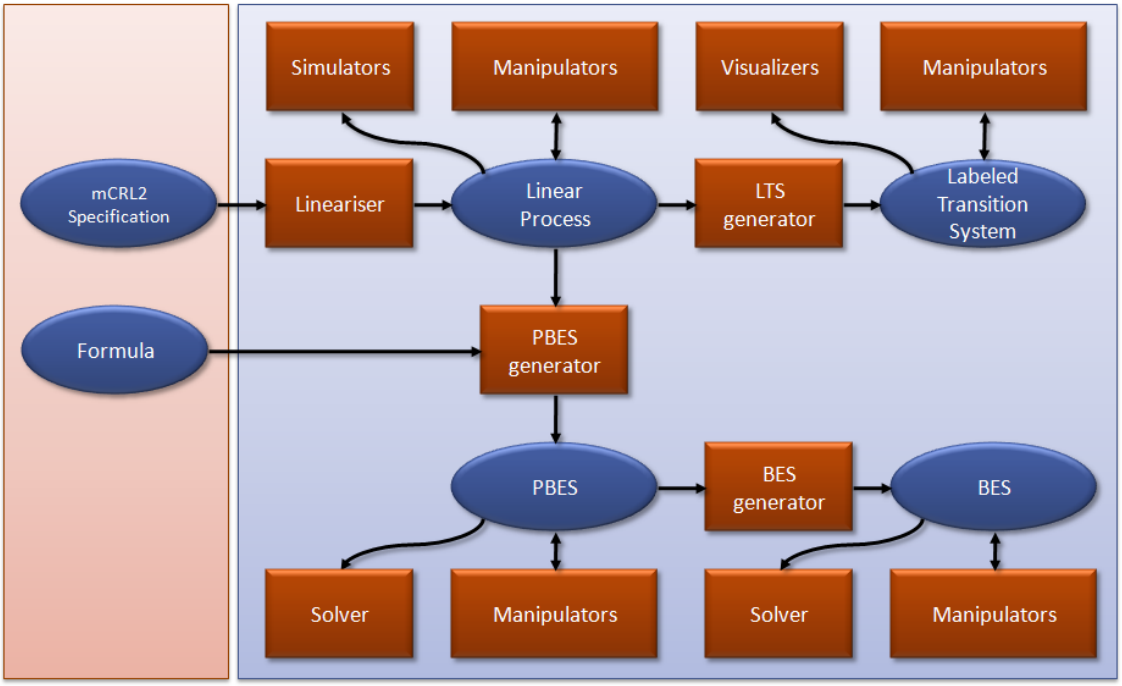


Figure 3.2: Graphical overview of the mCRL2 toolset [wpc].

it is done in order to reduce the state space. Finally, we restrict the type of the votes to Boolean, but any other type can be used without problems.

For now we model that every voter is also a signer and vice versa and that every voter is honest. Later we add dishonest voters and adapt the model such that we can have any number of signers. We model a voter using two processes: one that takes care of voter registration and one that takes care of the rest: receiving messages, sending signatures and sending the ballot. The full model is included in Appendix A.1, we give here a detailed explanation of the model.

The registration process is modelled as follows.

```
0 proc regVoter(i: Nat, v: Bool, N: Nat) =
     send(i, blind(i, vote(v, nonce))) .
       Voter(i, [sign(i, vote(v, nonce))], [], 0, N);
```

The process needs to be called with a natural number i representing the voter's identity, a Boolean v representing his vote and a natural number N representing the number of voters. The process performs action `send` that models putting the blinded vote on the authenticated broadcast channel. After that, the corresponding $Voter$ process is called, where the voter's own signature of the vote is already in the list of signatures. We do this to keep the state space within limits. If we would let a voter explicitly receive and sign its own blinded vote, then for example for two voters the state space consists of 4105 states and 7570 transitions compared to 619 states and 1108 transitions when we put the voter's own signature already in his list of signatures.

The voter process is included in Figure 3.3. The process needs to be called with natural number i that again represents the voter's identity, a list *sigList* of items of type Data containing signatures, another list *tally* containing items of type Data representing the tally, a natural number j representing the index on the broadcast channel and natural number N, representing the total number of voters in the protocol. Action `request(j,d)` models getting the message with index j from the broadcast channel (or local cache). After that a decision is made based on the type of the message and the appropriate action is performed. The action `request(j,d)` is, due to the way the broadcast channel is modelled, only enabled if there is actually a message with index j. When the action is not enabled the process has to wait until it is enabled again, or

```
proc Voter(i: Nat, sigList: List(Data), tally: List(Data), j: Nat, N: Nat) =
  sum d: Data. request(j, d) . (
      isBlind(d) ->
        ((blinder(d) != i) ->
          send(i, sign(i, d)) .
            Voter(i, sigList, tally, j+1, N)
       <> selfSigned(i, d) .
            Voter(i, sigList, tally, j+1, N))
    + isSign(d) ->
        (unblind(i, d) != err) ->
          unblinded(i, d) .
            Voter(i, insListData(unblind(i, d), sigList), tally, j+1, N)
       <> cannotUnblind(i, d) .
            Voter(i, sigList, tally, j+1, N)
    + isBallot(d) ->
        receivedBallot(i, d) .
          Voter(i, sigList, insListData(d, tally), j+1, N)
    )
    + (#sigList == N) -> sdone .
        sum o: Nat . asend(ballot(sigList, o)) .
          Voter(i, sigList, tally, j, N);
```

Figure 3.3: Process *Voter* from the parallel, honest model of the poor man's protocol.

perform action `sdone` if $\#sigList = N$.

The condition $\#sigList = N$ models the deadline in the protocol. In practice this deadline will be a timed deadline, which we can not model realistically using time. After the deadline has passed, action `asend(ballot(sigList, o))`, which models the sending of a ballot over the anonymous broadcast channel, is enabled for some natural number o.

In order to support this model of the deadline, the broadcast channel needs to have some intelligence. The broadcast channel is modelled as follows.

```
proc bChannel(l: List(Data), N: Nat, order: Nat) =
  sum d:Data, i:Nat . recv(i, d) . bChannel(l <| d, N, order) +
  sum i: Nat . (i < #(l)) -> give(i, l.i) . bChannel(l, N, order) +
  (N > 0) -> rdone . bChannel(l, Int2Nat(N-1), order) +
  (N == 0) -> sum d:List(Data) .
                arecv(ballot(d, order)) .
                  bChannel(l <| ballot(d, order), N, order+1);
```

Action `recv(i,d)` models that the broadcast channel receives some data item d from some voter i, after which the item is added to the end of the list l. This list l models a cache of all message received during the current run of the protocol. When a data item with index i is requested, it is given using action `give` if the index is valid. Then the intelligent part for the deadline: after the broadcast channel has performed N `rdone` actions it starts accepting messages over the anonymous broadcast channel.

In the above text we have introduced a number of actions. Some of these actions need to communicate, for example a `send` action from a voter needs to communicate with a `recv` action from the broadcast channel, to make sure that the message the voter wants to send ends up on the broadcast channel. The full set of communicating actions is $\texttt{recv}|\texttt{send} \rightarrow \texttt{bcast}, \texttt{request}|\texttt{give} \rightarrow \texttt{retrieve}, \texttt{rdone}|\texttt{sdone} \rightarrow \texttt{done}, \texttt{asend}|\texttt{arecv} \rightarrow \texttt{abcast}$, where $\texttt{a}|\texttt{b} \rightarrow \texttt{c}$ means that the multi-action `a|b` communicates to the action `c`.

Not all multi-actions communicate to an action, consider for example `give|send` that does not communicate and thus does not make sense in the model. Therefore we only allow a certain set of actions in the process, which is for our model the set {`bcast`, `retrieve`, `unblinded`, `cannotUnblind`, `receivedBallot`, `selfSigned`, `abcast`, `done`}.

After that, we hide all internal behaviour, that is behaviour that can not be externally observed. Actions that are hidden are replaced by the internal action τ. Here all actions except the broadcast action `bcast` and anonymous broadcast action `abcast` are hidden. The set of hidden actions is

thus {retrieve, unblinded, cannotUnblind, receivedBallot, selfSigned, done}.

To make the model complete we need some initial process calls. For example the complete initial call for a model consisting of a broadcast channel and two voters (one with vote true and one with vote false) looks like:

```
init  hide({retrieve, unblinded, cannotUnblind, receivedBallot, selfSigned, done},
      allow({bcast, retrieve, unblinded, cannotUnblind, receivedBallot,
             selfSigned, abcast, done},
      comm({recv|send->bcast, request|give->retrieve, rdone|sdone-> done,
            asend|arecv->abcast},
          bChannel([], numVoters, 0) ||
          regVoter(0, true, numVoters) || regVoter(1, false, numVoters)
      )
    )
  );
```

Using the full model we are able to construct an LPS using the tool *mcrl22lps*. The LPS contains a linearization of the parallel process of the initial call, which can be used by a number of tools. The sending of blinded votes is modelled by the action `bcast(i, blind(i, vote(v, nonce)))`. Since *blind* and *vote* are just constructors, this action `bcast(i, blind(i, vote(v, nonce)))` is present in the LPS, which means that vote v can be coupled to voter i (thus violating the privacy property). Therefore we rename the `bcast` and `abcast` actions using the tool *lpsactionrename* with the following rename file (also included in Appendix A.1.1).

```
var i: Nat;
    d: Data;
rename
isBlind(d) -> bcast(i, d) => bcast(i, blindmsg(blinder(d)));
isSign(d) && isBlind(smsg(d)) -> bcast(i, d) =>
    bcast(i, sign(signer(d), blindmsg(blinder(smsg(d)))));
isBallot(d) ->
  abcast(d) => abcast(ballot(addOrderToNonces(listc(d), order(d)), order(d)));
```

Since in the honest model every voter only sends a single blinded vote we rename the blinded vote of voter i to *blindmsg(i)*. This corresponds with what an observer can observe in practice: he can observe who has sent a certain blinded message but the message itself looks like random data. The observer can identify the message after he has seen it once since he then knows what it looks like.

The action for the casting of a ballot is renamed to casting a ballot where the list of signatures is processed by the function *addOrderToNonces*. This function replaces the nonce in the *vote(v,nonce)* constructor by an ordered nonce *ordNonce(o)*, where o represents the order in which the ballot is cast. The first ballot has order zero, the second order one, and so on. This also corresponds with what an observer can observe: since he has not seen the nonces yet and since all nonces are random, it follows that he can distinguish between them but does not know how to order them.

Issues regarding mCRL2 We have encountered some problems using revision 6517-shared of the mCRL2 toolset and some of these have influenced the model.

First of all, the mCRL2 specification language has support for sets, however in the implementation of the toolset these are not normalized yet. This means that two equal sets can have different denotations. For example, the empty set is syntactically not the same object as the empty set to which an item is added and then removed again. Therefore it is advised [wpa] to use lists when the state space has to be generated. We have done this, but the disadvantage of lists is that the lists $[a, b]$ and $[b, a]$ are two different lists, whereas $\{a, b\}$ and $\{b, a\}$ represent the same set. We have overcome this deficiency by using sorted lists instead of sets, using our own function *insListData*.

The second problem we encountered was using the tool *lpsactionrename*. According to the documentation [wpb] it is possible to define extra constructors, actions and functions in the rename file. We would have liked to move function *addOrderToNonces* to the rename file since it is not used in the model itself. Unfortunately the tool *lpsactionrename* is then not able to parse the LPS anymore.

Generating the state space In order to generate the state space we first need to linearize the mCRL2 specification using the tool *mcrl22lps*. This generates a linear process specification (LPS), which can be used by a number of tools. Generating an LPS from the mCRL model (included in Appendix A.1) and rename file (included in Appendix A.1.1) is done as follows:

```
mcrl22lps --delta <name>.mcrl | lpsconstelm | lpsactionrename
  --renamefile=<renfilename>.ren | lpssuminst | lpsconstelm > <name>.lps
```

The option *delta* is used to add delta summands to the specification. These are not in the original specification to improve readability. If this option is not used the given specification translates into an LPS with time, which is unwanted since the model does not use time. We manipulate the LPS using various tools of the toolset in order to get a smaller or less complex LPS. From tests we have done we have found out that this can drastically reduce the generating of the state space. For example for the linear model in the next section with $n = 3$ it takes using the full toolchain 10s to generate the state space, while when we use only the tools *mcrl22lps* and *lpsactionrename* it takes 1m45s to generate the state space.

The tool *lpsconstelm* tries to find variables that stay constant in the whole process and replaces them by that constant value. Here for example, the voter identity i and the total number of voters N are constant and are replaced by their corresponding value. After that, the tool *lpsactionrename* applies the renaming as defined in file `<renfilename>.ren`. Then the tool *lpssuminst* instantiates summation variables of the LPS. Finally the tool *lpsconstelm* is used to clean up some summands of which the condition is false.

From the generated LPS we can construct a labelled transition system (LTS), using the tool *lps2lts*. This is done as follows:

```
lps2lts [--rewriter=jittyc] <name>.lps <name>.svc
```

Here the parameter *rewriter=jittyc* is optional. By supplying this parameter the tool uses a faster rewriter, although it takes some time to initialize the rewriter. So only for large state spaces time can be gained.

As an example we generate the model for two voters, where the first voter is assigned vote true and the second voter vote false (we use $\mathbf{v} = \langle \mathtt{T},\mathtt{F} \rangle$ to denote this). This model can be generated as described above in under one second. The resulting transition system has 619 states and 1072 transitions. When we visualize this transition system using the tool *ltsview* we get the figure in Figure 3.4. In the figure we can see that the figure branches a few times. Whenever it branches there is a choice that has to be made: either the first voter sends a certain message or the other voter does. We have also constructed a model for three voters, where the first voter votes false, the second true and the third also false. It takes about 10 minutes to generate this model and the resulting state space consists of 7568050 states and 20855299 transitions. This model is unfortunately too large to visualize using *ltsview*.

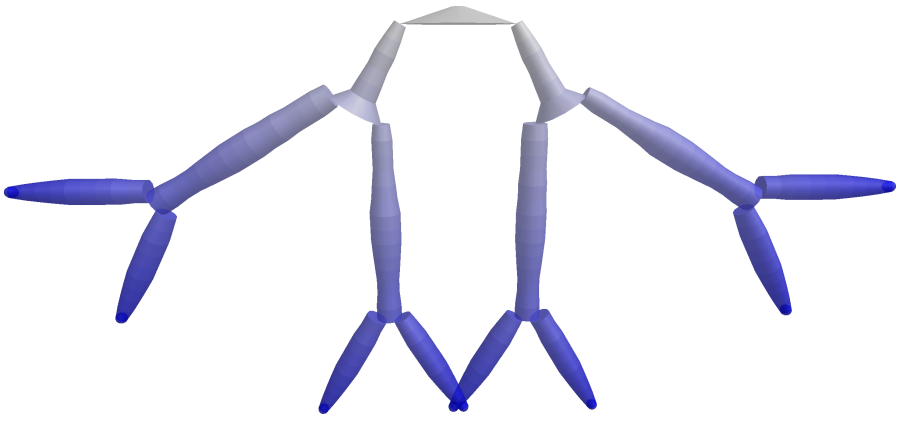


Figure 3.4: Visualization of state space of poor man's protocol for $n = 2$ and $\mathbf{v} = \langle \mathtt{T},\mathtt{F} \rangle$.

3.3 Linearizing the model

As can be seen in the previous section generating the state space can take quite some time, even for small values of n. Therefore we optimize the model before we add dishonest voters. We do this by creating a linear process ourselves instead of letting *mcrl22lps* make a linear process from the parallel model for us. We do not linearize the process by applying all rules by hand, since this would be too much work. Instead we try to construct a linear process that behaves exactly like the parallel model.

Constructing a linear process gives us a number of advantages over the parallel model. First, we can remove the explicit communication between the voters and the broadcast channel. We do this by modelling the broadcast channel as a parameter instead of as a separate process. Another change is that each voter now has its own local cache of the broadcast channel, which we can use to reduce the state space. In the parallel version a voter broadcasts a message to all voters, including himself. In the linear model this is not needed: whenever a voter broadcasts a message, the message is put in all local caches *except* that of the voter that sent the message. Instead the voter processes the message immediately.

Another advantage is that we can make choices in the model based directly on data. For example, in the parallel version we need to explicitly request a message from the broadcast channel and only after that it can be decided, based on the type of the message, what needs to be done with the message. In the linear model we can directly inspect the first message in the local cache. When all voters are modelled in a single process this means that this model needs to keep track of the parameters of all voters. We do this by making a list of all parameters, except the parameters that are shared. Then, the list at index i represents the specific parameter for voter i. The full model, including comments, can be found in Appendix A.2. Here we only show the main process $Voters$ in Figure 3.5. Note the similarities in the structure of the linear model and the structure of the parallel model. The function *updSingleList* is the abstract version of the function *insListData*; it applies the function *insListData* on the list with the given index.

When we compare the size of the state spaces and the time to generate them in the parallel model with the linear model we see that the state spaces in the linear model are a lot smaller and can be generated a lot faster. Some measurements are included in the following table. We have also tried to generate the state space for $n = 4$ and $\mathbf{v} = \langle \texttt{F,T,F,T} \rangle$ but we stopped this when almost 30GB of memory was used. At that time there were more than 37,5 million states processed and the indication was that the state space generation was not merely finished.

n	votes	time		#states		#transitions	
		linear	*parallel*	linear	*parallel*	linear	*parallel*
2	`T,F`	0m01s	*0m01s*	22	*619*	28	*1072*
3	`F,T,F`	0m10s	*10m00s*	37448	*7568050*	101617	*20855299*

Of course we want to know whether our linear model models the same process as the parallel model. While this is not possible to do for an arbitrary n, it is possible using the toolset to compare two processes using some equivalence relation. We use the tool *ltscompare* to compare the LTSs for the parallel model and the linear model. As equivalence relation we choose branching bisimulation since the explicit communication in the parallel model is hidden. For $n = 2$ and $\mathbf{v} = \langle \texttt{T,F} \rangle$ we can verify within one second that the two models are branching bisimilar. For $n = 3$ and $\mathbf{v} = \langle \texttt{F,T,F} \rangle$ we can verify that the models are branching bisimilar in 2m20s. To check whether two state spaces are branching bisimilar we use the tool as follows.

```
ltscompare --equivalence=branching-bisim <file1>.svc <file2>.svc
```

When we visualize the transition system of the linear model we get the figures in Figure 3.6. The figure does not branch as often as the parallel model. Due to the fact that there now only is a single process, the only branching comes from the sending of ballots since these include an order.

```
proc Voters(N: Nat, inBuf: List(List(Data)), regList: List(Nat), sigList:
  List(List(Data)), votes: List(Bool), tally: List(List(Data)), order: Nat,
  castList: List(Nat)) =
  sum i:Nat . (0 <= i && i < N) ->
    !(i in regList) ->
      bcast(i, blind(i, vote(votes.i, nonce))) .
        Voters(N, removeToe(i, updBuf(blind(i, vote(votes.i,
               nonce)), inBuf)), insListNat(i, regList),
               updSingleList(i, sign(i, vote(votes.i, nonce)), sigList),
               votes, tally, order, castList)
    <>
    (
    (#(inBuf.i)>0) -> (
      isBlind(head(inBuf.i)) ->
          bcast(i, sign(i, head(inBuf.i))).
            Voters(N, removeToe(i, removeHead(i, updBuf(sign(i,
                  head(inBuf.i)), inBuf))), regList, sigList, votes,
                  tally, order, castList)
      + isSign(head(inBuf.i)) -> (
        (unblind(i, head(inBuf.i)) != err) ->
          unblinded(i, head(inBuf.i)) .
            Voters(N, removeHead(i, inBuf), regList, updSingleList(i,
                   unblind(i, head(inBuf.i)), sigList), votes, tally,
                   order, castList)
        <> cannotUnblind(i, head(inBuf.i)) .
             Voters(N, removeHead(i, inBuf), regList, sigList, votes, tally,
                   order, castList)
        )
      + isBallot(head(inBuf.i)) ->
        receivedBallot(i, head(inBuf.i)) .
          Voters(N, removeHead(i, inBuf), regList, sigList, votes,
                 updSingleList(i, head(inBuf.i), tally), order, castList)
      )
    + (reachedDeadline(inBuf)) ->
       !(i in castList) ->
        abcast(ballot(sigList.i, order)) .
          Voters(N, removeToe(i, updBuf(ballot(sigList.i, order), inBuf)),
                 regList, sigList, votes, updSingleList(i, ballot(sigList.i,
                 order), tally), order+1, insListNat(i, castList))

);
```

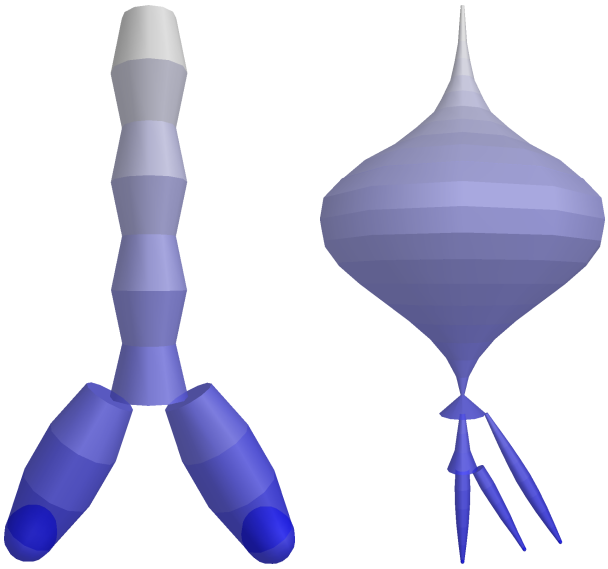
Figure 3.5: Process *Voters* from the linear, honest model of the poor man's protocol.

3.4 Modelling dishonesty

In this section we describe how we model dishonesty in our model of the poor man's protocol. The main focus is on what dishonest actions are possible for voters and for signers and how these are modelled. The full dishonest model is included in Appendix A.3 and the rename file in Appendix A.3.1.

In general the following dishonest actions are possible in the protocol:

- For a voter:
  1. sending multiple blinded votes
  2. casting multiple ballots
- For a signer:
  3. not (properly) signing a blinded vote for a voter that did not register yet
  4. signing a vote from a voter that already registered a vote



(a) $n = 2$, $\mathbf{v} = \langle \texttt{T,F} \rangle$ (b) $n = 3$, $\mathbf{v} = \langle \texttt{F,T,F} \rangle$

Figure 3.6: LTSs for linear model of poor man's protocol (scaled).

We discuss these dishonest actions one-by-one in the following subsections, where the emphasis lies on how these actions are modelled, but first we start with a discussion about some general points on modelling dishonest actions.

3.4.1 General remarks on modelling dishonest actions

There are a number of things we need to consider when modelling dishonest actions. First, since we use mCRL2 for our model we need to model a dishonest entity explicitly. This is not the case for all tools, ProVerif for example includes an implicit attacker. Therefore we have to think of all possible dishonest actions that are possible, since they all should be modelled.

Secondly, we need to keep in mind that we are constructing a model for which we want to construct the state space. A number of dishonest actions consist of repeating a certain action or sending a (slightly different) message over and over. In these cases the honest behaviour is often sending a certain message once, whereas the dishonest behaviour can be sending a number of similar messages instead of or including the honest message. Since it is easy to detect if the exact same message is sent more than once, we only consider unique messages in our model. If we would not do this it is likely that the state space becomes too large to generate. We have introduced a parameter *limit* which limits the number of dishonest actions. This means that each dishonest voter gets to perform a certain action, either honest or dishonest, *limit* times. In our model it is not possible that a certain voter sends the exact same message more than once. This is done in order to keep the state space within limits and since duplicate message are trivial to ignore in an implementation.

In our model we have modelled the number of dishonest voters using the process parameter *numDishonestVoters*. The default value of this parameter is set to the threshold value minus one $(k-1)$, which is the maximum number of dishonest voters that is allowed in our protocol. In the model, the dishonest voters are chosen by their identity; all voters with an identity smaller than *numDishonestVoters* (voter identities are in the range $[0, N)$) can be dishonest. All dishonest voters in the model have the ability to sign, since the signers are also chosen by index: when a voter has index smaller than parameter *numSigners* (at least $2k-1$) he is also a signer.

With the introduction of dishonest actions a large increase of the state space is expected. Every dishonest voter gets for each possible dishonest action the choice to either perform it, or to not

perform it and behave honest for that moment. Since there are a number of points in the protocol where a dishonest voter can decide to either behave honest or dishonest, there are many different combinations of dishonest and honest actions. These combinations are all represented by branches in the LTS.

Since we want to verify properties on the model, we would like to have some action from which we can observe the tally. In the honest model it is only possible to observe the tally at the end of the protocol, after all messages have been processed by inspecting the parameter *tally.i* for some i. Since this can not be done in model checking and since dishonest voters can have an incorrect tally, we introduce an objective external observer which sole task is to produce a final tally at the end of the protocol. For that, he needs all ballots and all dishonesty lists, but not the other messages like blinded votes and signatures. We have therefore modelled an extra cache of the broadcast channel, represented by the parameter *inBufObs*. The external observer publishes the final tally using function *ftally*, which contains a list of ballots as parameter. This function marks the end of the protocol.

3.4.2 Sending multiple blinded votes

In the protocol, a dishonest voter can be dishonest by sending multiple blinded votes. He could do this for example to try to get his vote counted multiple times. The easiest way to do this is by sending the same blinded vote more than once, however this can easily be detected. A smarter way is to generate a number of different blinded votes, using a different nonce each time. However, it can always be detected when a dishonest voter sends multiple blinded votes since the blinded votes need to be sent over an authenticated broadcast channel. So from each voter, only the first blinded message it sends is signed by all (honest) signers. In order to see how this is modelled we first recall how blinded votes are sent in the honest model.

```
0     !(i in regList) ->
        bcast(i, blind(i, vote(votes.i, nonce))) .
          Voters(N, removeToe(i, updBuf(blind(i, vote(votes.i,
                 nonce)), inBuf)), insListNat(i, regList),
                 updSingleList(i, sign(i, vote(votes.i, nonce)), sigList),
5                votes, tally, order, castList)
```

In the honest model, the parameter *regList* is just a list of voters that have cast their votes. In the dishonest model this principle is not changed, however some extra information is kept. For each voter, the parameter *regList* contains a tuple containing:

voterid The identity of the voter.

tries The number of times the voter has sent a blinded vote plus the number of times he has decided to not send a blinded vote. For honest voters, *tries* can not become higher than one, for dishonest voter not higher than *limit*.

bindex The index of the blinded vote. That is, the first blinded vote has index zero, the next one has index one and so on. This index is used in the nonce.

If a dishonest voter decides to send multiple blinded votes, each blinded vote is constructed uniquely using a nonce with index *bindex*. Since the number of signers is variable in this model it is not always the case that every voter is a signer. We thus have to make a case distinction on which honest voters can sign and which can not (recall that dishonest voters are modelled such that they can always sign). In the dishonest model, the part for sending a blinded vote is modelled as follows, where s denotes the number of signers and d the number of dishonest voters.

```
0     % If voter i has not yet cast its blinded vote, he can do this.
        % check whether voter i has already cast a blinded vote
        ((i >= d) && (bindex(regList.i) < 1)) -> (
          (i < s) -> % voter is also a signer, hence he can sign his own vote
            bcast(i, blind(i, vote(votes.i, nonce(0)))) .
5             Voters(N, k, d, s, removeToe(i, updBuf(blind(i, vote(votes.i,
                     nonce(0))), inBuf)), addOneTries(i, addOneBindex(i,
```

```
                    regList)), updSingleList(i, sign(i, vote(votes.i,
                    nonce(0))), sigList), votes, tally, order, castList, limit,
                    signedList, dcList, alldcList, inBufObs)
  <> % voter is not a signer, and can hence not sign his own vote
    bcast(i, blind(i, vote(votes.i, nonce(0)))) .
      Voters(N, k, d, s, removeToe(i, updBuf(blind(i, vote(votes.i,
                    nonce(0))), inBuf)), addOneTries(i, addOneBindex(i,
                    regList)), sigList, votes, tally, order, castList, limit,
                    signedList, dcList, alldcList, inBufObs)
)
<>
% dishonest voter can cast 'limit' blinded votes
((i < d) && (tries(regList.i) < limit)) -> ( %has not yet reached limit
  % internal decision: casting a blinded vote
  bcast(i, blind(i, vote(votes.i, nonce(bindex(regList.i))))) .
    Voters(N, k, d, s, removeToe(i, updBuf(blind(i, vote(votes.i,
           nonce(bindex(regList.i)))), inBuf)),
           addOneTries(i, addOneBindex(i, regList)), updSingleList(i,
           sign(i, vote(votes.i, nonce(bindex(regList.i)))),
           sigList), votes, tally, order, castList, limit, signedList,
           dcList, alldcList, inBufObs)
+ % or not casting a blinded vote
  noBlind . Voters(N, k, d, s, inBuf, addOneTries(i, regList),
                   sigList, votes, tally, order, castList, limit,
                   signedList, dcList, alldcList, inBufObs)
)
```

Where in the honest model a static nonce was included in each blinded vote, we use a parameterized nonce in the dishonest model. This is done to model that a different nonce is used in each blinded vote. That way the first blinded vote can always be recognized since it is modelled such that it contains *nonce(0)*. In the model it is possible to force a dishonest voter to always choose the sending of a blinded vote, that way *limit-1* blinded votes are sent dishonestly. This can be done by blocking the action `noBlind`. This replaces the action `noBlind` by the action δ, which has the result that the action that sends a blinded vote is always chosen.

3.4.3 Casting multiple ballots

Since dishonest voters can sign any vote they like (as we explain in Section 3.4.4), it is possible that a dishonest voter collects a number of signatures for more than just the first blinded vote. First, recall how ballots are cast in the honest model. Note the similarities with sending a blinded vote.

```
(reachedDeadline(inBuf)) ->
% Check whether voter has already cast a ballot or not to avoid that a
% voter sends an infinite number of ballots.
 !(i in castList) ->
  abcast(ballot(sigList.i, order)) .
    Voters(N, removeToe(i, updBuf(ballot(sigList.i, order), inBuf)),
           regList, sigList, votes, updSingleList(i, ballot(sigList.i,
           order), tally), order+1, insListNat(i, castList))
```

In the dishonest model we again need to make a case distinction on honest and dishonest voters. For the honest voters the model for casting a ballot remains largely the same, except that we need to use information from the dishonesty lists. For now it is enough to know that there is a parameter *alldcList* that for each voter contains a list that includes the union of all received dishonesty lists that that voter received. Using function *kTimesDishonest* we can extract all signers from such list that are accused at least k times of dishonesty. From these signers the signatures have to be removed from the ballots, since privacy can otherwise not be guaranteed (as explained in Section 2.2.2), this can be done using function *removeDishonestSignatures*. Since the ballots are needed to construct a tally, these are also added to the cache of the external observer.

The case for the dishonest voter is a generalization of the honest case. Since a dishonest voter can send (at most) *limit* blinded votes, he can construct the same number of ballots. To construct

a ballot, signatures that include the same nonce have to be collected. This is done using the function *filterSigByNonce*. The dishonest voter also filters signatures of dishonest signers from his ballots, since ballots containing signatures of dishonest signers are not accepted (we explain this in Section 3.4.5). Then, dishonest voters also get the choice to not send a ballot, which is represented by the action `noBallot`. This action can not just be blocked like the action `noBlind`, since the alternative (casting a ballot) is preceded by a condition. The part for sending a ballot is modelled as follows.

```
(reachedDeadline2(inBuf, dcList)) -> (
% An honest voter can only cast a single ballot.
((i >= d) && (#(castList.i) < 1)) ->
        abcast(ballot(removeDishonestSignatures(kTimesDishonest(0, k,
               alldcList.i), uniq(sigList.i)),
               order)) .
          Voters(N, k, d, s, removeToe(i, updBuf(ballot(
                  removeDishonestSignatures(kTimesDishonest(0, k,
                  alldcList.i), uniq(sigList.i)), order), inBuf)),
                  regList, sigList, votes, updSingleList(i,
                  ballot(removeDishonestSignatures(kTimesDishonest(0, k,
                  alldcList.i), uniq(sigList.i)), order), tally),
                  order+1, updSingleList(i, ballotIdx(0), castList),
                  limit, signedList, dcList, alldcList, inBufObs <|
                  ballot(removeDishonestSignatures(kTimesDishonest(0, k,
                  alldcList.i), uniq(sigList.i)), order))
<> (i < d) -> (
% A dishonest voter has the choice to broadcast 'limit' ballots.
  sum j: Nat . (0 <= j && j < limit) ->
    !(ballotIdx(j) in castList.i) ->
      ( % Sending a ballot.
        (removeDishonestSignatures(kTimesDishonest(0, k, alldcList.i),
         uniq(filterSigByNonce(j, sigList.i))) != []) -> (
          abcast(ballot(removeDishonestSignatures(kTimesDishonest(0, k,
                 alldcList.i), uniq(filterSigByNonce(j, sigList.i))),
                 order)) .
            Voters(N, k, d, s, removeToe(i, updBuf(ballot(
                   removeDishonestSignatures(kTimesDishonest(0, k,
                   alldcList.i), uniq(filterSigByNonce(j, sigList.i))),
                   order), inBuf)), regList, sigList, votes,
                   updSingleList(i, ballot(removeDishonestSignatures(
                   kTimesDishonest(0, k, alldcList.i), uniq(
                   filterSigByNonce(j, sigList.i))), order), tally),
                   order+1, updSingleList(i, ballotIdx(j), castList),
                   limit, signedList, dcList, alldcList, inBufObs <|
                   ballot(removeDishonestSignatures(kTimesDishonest(0,
                   k, alldcList.i), uniq(filterSigByNonce(j,
                   sigList.i))), order))
        )
        + % Sending no ballot.
          noBallot .
            Voters(N, k, d, s, inBuf, regList, sigList, votes, tally,
                   order, updSingleList(i, ballotIdx(j), castList),
                   limit, signedList, dcList, alldcList, inBufObs)
      )
  )
)
```

3.4.4 Dishonest actions for signers

In the poor man's protocol, all signers have to sign, for every voter, the first blinded message they receive from that voter. There are two dishonest actions a dishonest signer can perform:

1. not (properly) signing a blinded vote for a voter that did not register yet
2. signing a vote from a voter that already registered a vote

As discussed in Section 2.2.2 the second dishonest action does not pose a threat to the protocol, we check this by also including this action in the model. First, recall how signatures are sent in the honest model.

```
0       isBlind(head(inBuf.i)) ->
            bcast(i, sign(i, head(inBuf.i))).
              Voters(N, removeToe(i, removeHead(i, updBuf(sign(i,
                    head(inBuf.i)), inBuf))), regList, sigList, votes,
                    tally, order, castList)
```

In the dishonest model, dishonest signers get the choice between two options: signing or not signing an incoming blinded message. Since the dishonest signer can sign or not sign any message, we do not need to check whether the blinded message that is received is actually the first blinded message that the signer received from a certain voter. For the honest signers we do check this by keeping a list of all voters of which the signer has signed a blinded message. The signing part is thus modelled as follows.

```
0     isBlind(head(inBuf.i)) ->
        (i >= d) -> (%honest voters
          % Honest signers check whether they have already signed a blinded vote
          % by the sender of the current blinded vote. If it is not the case,
          % then sign the blinded vote.
5         (!(voter(blinder(head(inBuf.i))) in signedList.i) && (i < s)) ->
            bcast(i, sign(i, head(inBuf.i))) .
              Voters(N, k, d, s, removeToe(i, removeHead(i, updBuf(sign(i,
                    head(inBuf.i)), inBuf))), regList, sigList, votes,
                    tally, order, castList, limit, updSingleList(i,
10                  voter(blinder(head(inBuf.i))), signedList), dcList,
                    alldcList, inBufObs)
        <> % If it is the case, do not sign the blinded vote. Also do not sign
           % the blinded vote if the voter is not a signer.
            notSign(i, head(inBuf.i)) .
15            Voters(N, k, d, s, removeHead(i, inBuf), regList, sigList, votes,
                    tally, order, castList, limit, signedList, dcList,
                    alldcList, inBufObs)
        )
        <> %i < d; dishonest voters
20      (
          % Dishonest signers sign a blinded vote without checking whether it
          % was the first blinded vote that was sent by the sender.
          bcast(i, sign(i, head(inBuf.i))).
            Voters(N, k, d, s, removeToe(i, removeHead(i, updBuf(sign(i,
25                  head(inBuf.i)), inBuf))), regList, sigList, votes, tally,
                    order, castList, limit, signedList, dcList, alldcList,
                    inBufObs)
          +
          % A dishonest voter can decide to behave dishonest by not signing a
30        % blinded vote.
          notSign(i, head(inBuf.i)) .
            Voters(N, k, d, s, removeHead(i, inBuf), regList, sigList, votes,
                    tally, order, castList, limit, signedList, dcList, alldcList,
                    inBufObs)
35      )
```

3.4.5 Keeping track of dishonest signers

As discussed in the previous sections, it is important to keep track of which signers are dishonest and filter out their signatures. We therefore need to identify signers that do not sign blinded messages when they have to. This is done as follows: every signer S_i stores every incoming signature in a local list represented by parameter *dcList.i*. He does this for every signature, not only for the signatures that are intended for him but also for every other signature. This way the signer can later check which voters did not sign all blinded messages. In the following code snippet, which shows how signatures are processed in the dishonest model, storing signatures is represented by *updDCList(i, head(inBuf.i), dcList)*.

```
isSign(head(inBuf.i)) -> (
  (i < s) -> ( % Signers need to store all incoming signatures to
               % construct the dishonesty lists.
    (unblind(i, head(inBuf.i)) != err) ->
      unblinded(i, head(inBuf.i)) .
        Voters(N, k, d, s, removeHead(i, inBuf), regList, updSingleList(
               i, unblind(i, head(inBuf.i)), sigList), votes, tally,
               order, castList, limit, signedList, updDCList(i,
               head(inBuf.i), dcList), alldcList, inBufObs)
    <> cannotUnblind(i, head(inBuf.i)) .
         Voters(N, k, d, s, removeHead(i, inBuf), regList, sigList,
                votes, tally, order, castList, limit, signedList,
                updDCList(i, head(inBuf.i), dcList), alldcList,
                inBufObs)
  )
  <>
  ( %Voters that are not signers do not need to store all signatures.
  % (skip) same as for the signers, except that dcList is not updated
  )
)
```

Once the deadline has been reached, no new signatures can be sent or received and the signers can thus check which other signers have been dishonest. We have constructed the model such that a signer never accuses himself of being dishonest. This is done using the function *identifyDishonestSigners* which has as input a local list *dcList.i* of all signatures that signer observed on the network. The result of this function is a list of dishonest signers (called the dishonesty list), which is broadcasted over the network if it is not empty. Dishonest signers have the possibility to send an invalid list of dishonest signers, which is modelled by sending a list containing all signers, except the dishonest signer itself. The sending of a dishonesty list is included in the model as follows. Note that the dishonesty lists are also put in the cache of the external observer, since he also needs these to filter out invalid ballots.

```
((dcList.i != [null]) && (i < s)) -> (
  (identifyDishonestSigners(0, remove(voter(i),
  constructSignersList(s)), getListOfRegisteredVoters(regList),
  dcList.i) != []) -> (
    bcast(i, dishonestList(identifyDishonestSigners(0, remove(voter(i),
          constructSignersList(s)), getListOfRegisteredVoters(regList),
          dcList.i))) .
      Voters(N, k, d, s, removeToe(i, updBuf(dishonestList(
             identifyDishonestSigners(0, remove(voter(i),
             constructSignersList(s)), getListOfRegisteredVoters(
             regList), dcList.i)), inBuf)), regList, sigList, votes,
             tally, order, castList, limit, signedList,
             nullSingleList(i, dcList), updADC(i, dishonestList(
             identifyDishonestSigners(0, remove(voter(i),
             constructSignersList(s)), getListOfRegisteredVoters(
             regList), dcList.i)), alldcList), insListListData(
             identifyDishonestSigners(0, remove(voter(i),
             constructSignersList(s)), getListOfRegisteredVoters(
             regList), dcList.i), inBufObs))
  )
  <>
  ( % empty dishonesty list -> no need to broadcast dishonesty list
    emptyDishonestList(i) .
      Voters(N, k, d, s, inBuf, regList, sigList, votes, tally, order,
             castList, limit, signedList, nullSingleList(i, dcList),
             alldcList, inBufObs)
  )
  % dishonest: mark all other signers as dishonest voters
  + (i < d) -> (
  bcast(i, dishonestList(remove(voter(i), constructSignersList(s)))) .
    Voters(N, k, d, s, removeToe(i, updBuf(dishonestList(
           remove(voter(i), constructSignersList(s))), inBuf)), regList,
           sigList, votes, tally, order, castList, limit, signedList,
           nullSingleList(i, dcList), updADC(i,
```

```
                dishonestList(remove(voter(i), constructSignersList(s))),
35              alldcList), insListListData(remove(voter(i),
                constructSignersList(s)), inBufObs))
        )
      )
```

Every voter needs to store all incoming dishonesty lists, such that he can extract the really dishonest signers (signers that are accused of dishonesty at least k times) from them. This is easily done by updating a local list represented by parameter *alldcList.i*.

```
0     isDishonestList(head(inBuf.i)) ->
        receivedDishonestList(i, head(inBuf.i)) .
          Voters(N, k, d, s, removeHead(i, inBuf), regList, sigList, votes,
                 tally, order, castList, limit, signedList, dcList, updADC(i,
                 head(inBuf.i), alldcList), inBufObs)
```

Once all dishonesty lists are processed, the protocol proceeds to the phase where the ballots are cast. There, first the dishonest ballots are filtered out using function *removeDishonestSignatures* which takes local list *alldcList.i* as parameter, as discussed in Section 3.4.3. Finally, ballots that are sent that contain either too few signatures or signatures from dishonest signers, are rejected by honest voters. Dishonest voters accept all ballots, as can be seen in the following code snippet.

```
0     isBallot(head(inBuf.i)) ->
        ((#(uniq(listc(head(inBuf.i)))) < k ||
          containsDishonestSignatures(kTimesDishonest(0, k, alldcList.i),
          listc(head(inBuf.i)))) && !(head(inBuf.i) in tally.i) && i >= d) ->
        % If a ballot contains insufficient signatures or signatures of
5       % dishonest signers or if the ballot is already received (these are
        % called invalid ballots), then do not store the ballot (only done
        % by honest voters).
        receivedBallot(i, head(inBuf.i)) .
          Voters(N, k, d, s, removeHead(i, inBuf), regList, sigList, votes,
10               tally, order, castList, limit, signedList, dcList, alldcList,
                 inBufObs)
        <> % Otherwise the ballot is stored. In case the voter is dishonest,
           % he also stores invalid ballots in order to try to get them
           % accepted.
15      receivedBallot(i, head(inBuf.i)) .
          Voters(N, k, d, s, removeHead(i, inBuf), regList, sigList, votes,
                 updSingleList(i, head(inBuf.i), tally), order, castList,
                 limit, signedList, dcList, alldcList, inBufObs)
```

3.4.6 Rename file

In the dishonest model the nonces in the blinded votes are not static anymore (which they were in the honest model) but are parameterized using an order, as has been discussed in Section 3.4.2. Therefore we need to change how blinded votes are renamed. In the honest model this was done as follows.

```
isBlind(d)-> bcast(i, d)=> bcast(i, blindmsg(blinder(d)));
```

In the dishonest model we also include the order of the nonce. The rename rule then becomes.

```
isBlind(d)-> bcast(i, d)=> bcast(i, blindmsg(blinder(d), ord(nonce(bmsg(d)))));
```

The rename rule for the sending of a signature is adapted in the same way. Since we added an external observer to the dishonest model we need to include the following rename rule, which adds the order to all ballots in a tally in a similar way as is done for a single ballot.

```
isTally(d)-> finalTally(d)=> finalTally(vtally(addOrderToNoncesTally(tallyc(d))));
```

The full rename file is included in Appendix A.3.1.

3.4.7 State spaces

Since we have introduced a large number of options in the dishonest model it was expected that the state space is a lot larger than that for the honest model. That this expectation was correct can be seen in the following results. Recall that n denotes the total number of voters, d the number of dishonest voters, s the number of signers and k the threshold value.

n	k	d	s	limit	$\langle \mathbf{v} \rangle$	time	#states	#transitions
3	2	$k-1$	$2k-1$	2	`F,T,F`	0m57s	234575	668894
3	2	$k-1$	$2k-1$	3	`F,T,F`	2m57s	748309	2197328
3	2	$k-1$	$2k-1$	4	`F,T,F`	10m53s	2302971	7054958
3	2	$k-1$	$2k-1$	5	`F,T,F`	70m44s	9509212	31575572

As an example of what the state space for the dishonest model looks like we have visualized the state space for the model with $n = 3$, $k = 2$, $d = k-1$, $s = 2k-1$, $limit = 2$ and $\mathbf{v} = \langle$`F,T,F`$\rangle$. This figure is included as Figure 3.7.

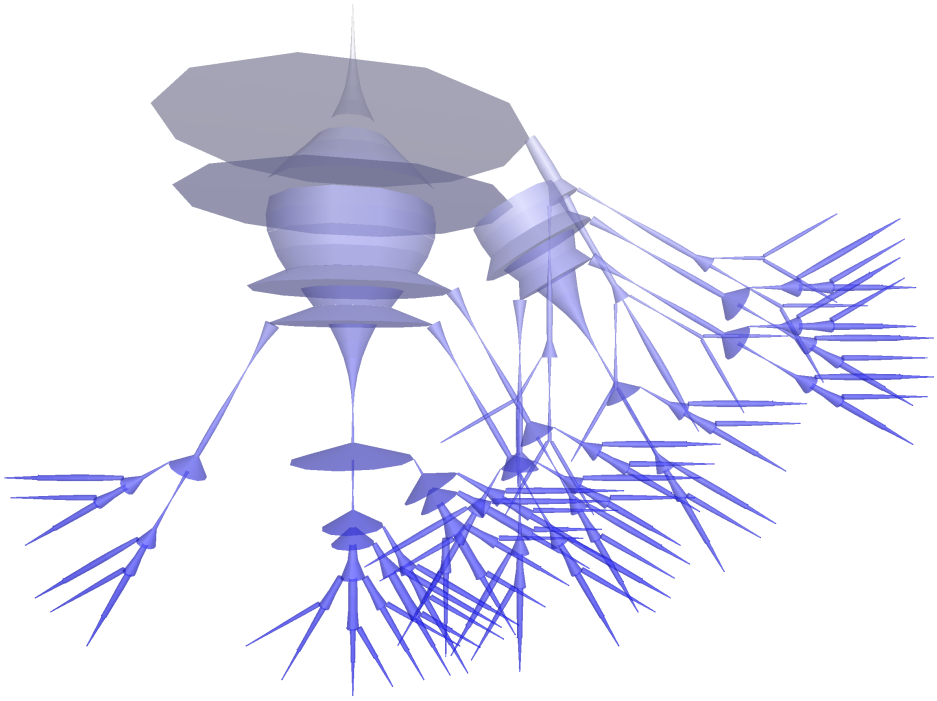


Figure 3.7: LTS for dishonest model of poor man's protocol, with $n = 3$ and $\mathbf{v} = \langle$`F,T,F`$\rangle$.

3.5 Strong synchronicity

In our search for the model that generates the smallest state space while still preserving the properties of the poor man's protocol, we try to adapt our model to the setting of strong synchronicity. The idea of *strong synchronicity* is that every participant in the protocol is processing the same message at the same time. This can be used to add some sort of ordering on the broadcast messages, which reduces the state space. We do not model strong synchronicity in the strictest sense since internal actions and the processing of data can not be observed by other entities. Since the broadcast buffer is modelled as a queue, sometimes a number of internal actions need to be performed before a certain message can be processed.

The strong synchronicity scheme has most effect in the REGISTRATION phase, where the voters send their blinded votes and the signers send their signatures of these votes. What happens when the strong synchronicity scheme is used is the following. After a voter V_i sends a blinded vote, all signers have to send their signature before another voter V_j is allowed to send his blinded vote. Recall that in the other phases of the protocol the (anonymous) broadcast messages are protected by deadlines. The strong synchronicity scheme does therefore not have effect in these phases.

The current dishonest model uses a blind-then-sign approach, where each voter first sends its blinded vote before signing other votes. This does not work in combination with the strong synchronicity scheme since there would arise a deadlock after the first voter sends his blinded vote. Thus in order to model the strong synchronicity we need to drop the blind-then-sign approach. We model the strong synchronicity using semaphores. In this model, we only need a single semaphore s, which is used as follows.

$s = 0$	$\rightarrow$	Some voter can send a blinded vote, after doing that the semaphore s is set to the number of voters minus one (the number of voters that need to process that message). While $s = 0$, sending signatures is prohibited.
$s > 0$	$\rightarrow$	Some signer can send a signature for the blinded vote that was sent when the semaphore was set to the number of voters minus one. After that, s is decreased by one. Also when a voter does not sign the blinded vote (either because he is not a signer or if he is a signer and he has decided not to), s is decreased by one. While $s > 0$, sending blinded votes is prohibited.

After a blinded vote has been cast the semaphore s is set to the number of voters minus one, which is the number of voters that need to process that blinded vote. Since not all voters can sign, some voters process the message by not signing it. This is modelled by the action `notSign`, which is in the set of hidden actions. This action can also be performed in some cases by voters that are signers, for example when a dishonest voter decides to not sign some blinded vote. This gives some trouble here since we do not know exactly how many signatures can be expected for a single blinded vote. We only know that it should be between the number of honest signers and the number of all signers. If we knew the exact number of signatures to expect, we could use that number to set the semaphore to after a blinded vote has been cast. Then the semaphore can be lowered by one each time a signature is broadcasted. Unfortunately, it is not possible to determine that number and therefore we also have to change the semaphore on the action `notSign`.

Using this scheme with semaphores, the sending of messages is now regulated since signatures can only be sent when the semaphore s is larger than zero and blinded votes can only be sent when the semaphore s is equal to zero.

Adapting the model

In order to adapt the model to the strong synchronicity setting, we need to add the semaphore s to the model. We do this by adding an extra parameter *semaphore* of type natural to the end of the parameter list. Then the following changes have to be applied to the model to transform it to a model with strong synchronicity.

- Sending blinded votes is only enabled when the semaphore is equal to zero. This can be done by adding the condition *semaphore = 0* before the part where the blinded votes are sent.
- We replace the blind-then-sign strategy by a strategy where blinded votes and signatures are sent mixed. This is done by replacing the else (<>) between the part where the blinded votes are sent and the rest of the protocol by an alternative composition (+).
- Each time a blinded vote is broadcast, the semaphore needs to be set to *N-1*.
- Each time a blinded vote is processed the semaphore needs to be lowered by one.

- All other recursive calls should be adapted such that they include the semaphore. The value of the semaphore in this calls should remain unchanged.
- The initial value of the semaphore is zero.

The model using strong synchronicity is included in Appendix A.4. Only the process $Voters$ and its initialization is included there since all actions, sorts and functions remain the same as for the dishonest model. We have done some tests by generating the state space of the models with strong synchronicity, which can be found in the table below. An example visualization of the model with parameters $n = 3$, $k = 2$, $d = k - 1$, $s = 2k - 1$, $limit = 2$ and $\mathbf{v} = \langle \mathtt{F,T,F} \rangle$ can be found in Figure 3.8.

model	n	k	d	s	limit	$\langle \mathbf{v} \rangle$	time	#states	#transitions
dishonest	3	2	$k-1$	$2k-1$	2	`F,T,F`	0m57s	234575	668894
strong sync.	3	2	$k-1$	$2k-1$	2	`F,T,F`	0m16s	19232	52586
strong sync.	4	2	$k-1$	$2k-1$	2	`F,T,F,T`	8m48s	1313229	4773822

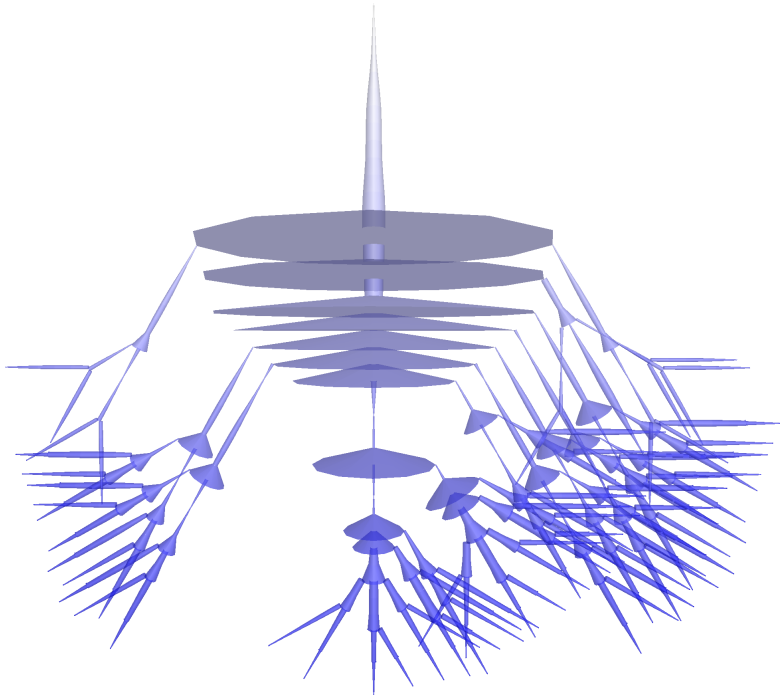


Figure 3.8: LTS for model with strong synchronicity of poor man's protocol, with $n = 3$ and $\mathbf{v} = \langle \mathtt{F,T,F} \rangle$.

3.6 Verification

The next step after the construction of the model of the protocol is the verification of some properties on the model. Since we verify properties on the model of the protocol, it is expected that if the properties hold for the model, they also hold for the protocol. We do two types of verification, where we discuss each type individually. First, we use equivalence relations on processes to verify the privacy property and then we use model checking to verify the unicity property and some other less-general properties.

3.6.1 Privacy property

The privacy property says that no one can get to know the vote of another voter. We can formalize this by checking whether an external observer can tell whether votes are changed in two protocol runs. We do this by generating two processes: one where the votes are assigned using a certain list of votes and one where they are assigned using the permutation of that list. When an external observer can not tell the difference between these two processes, that is when these two models are strongly bisimilar, the privacy property holds for the model.

We however only guarantee the privacy of honest voters, so we only need to check models where the votes of honest voters are permuted. We can not guarantee the privacy of dishonest voters. Consider for example a model with a single dishonest voter. It is then possible that the dishonest voter casts more than one blinded vote and thus also more than one ballot. From the extra blinded vote the observer can not deduce any more information than that the voter is dishonest. However since the ballot contains the plain-text vote, the observer notices when the vote of the dishonest voter is changed. This is not a flaw in the model, the choice of the dishonest voter leads to the fact that privacy can not be guaranteed for dishonest voters.

We can determine whether two processes are related using some equivalence relation on processes using two different techniques:

1. Generating two state spaces using the tool *lps2lts* and comparing these for the given equivalence relation using the tool *ltscompare*.
2. Expressing the equivalence relation on the two processes as a Parameterized Boolean Equation System (PBES) using the tool *bisimulation*.

Technique 1: ltscompare In order to compare two state spaces we first need to generate two models with the the same set of parameters, except for the list of votes. The state spaces can be generated in the normal way, as defined in Section 3.2. After the state spaces have been generated we can use the tool *ltscompare* as follows to check whether the equivalence relation holds.

```
ltscompare --equivalence=bisim <model1>.svc <model2>.svc
```

When the tool is finished it reports whether the equivalence holds or not. We have done some tests using this technique, the results are in the following table. The first parameter specifies which model is used, the dishonest model or the strong synchronicity model. The next two columns specify the values of the parameters of both processes, then the column $\langle \mathbf{v} \rangle$ specifies the distribution of votes for the first process, $\langle \pi(\mathbf{v}) \rangle$ the distribution for the second process, time_{sg} the time to generate the state space for a single process and time_{lc} the time it takes to run *ltscompare*. Finally, the last column contains whether the two processes are bisimilar. The whole technique thus takes for a certain set of parameters, $2{\cdot}\text{time}_{\text{sg}} + \text{time}_{\text{lc}}$ time. All models use the same values for parameters d $(= k - 1)$, s $(= 2k - 1)$ and *limit* $(= 2)$.

model	n	k	$\langle \mathbf{v} \rangle$	$\langle \pi(\mathbf{v}) \rangle$	time_{sg}	time_{lc}	bisimilar
dishonest	2	1	`F,T`	`T,F`	0m01s	0m01s	yes
dishonest	3	2	`F,T,F`	`F,F,T`	0m57s	0m11s	yes
dishonest	3	2	`T,F,F`	`F,F,T`	0m57s	0m06s	no
strong sync.	4	2	`F,T,F,T`	`F,F,T,T`	8m48s	14m39s	yes

Recall that dishonest voters are chosen by their identity, the first d voters are dishonest in a model. We indeed observe that privacy is only guaranteed for honest voters: when the vote of the dishonest voter is changed (third result in the above table), the models are not bisimilar.

Technique 2: bisimulation Using the tool *bisimulation* it is possible to express a equivalence relation on processes as a PBES. We can use the tool *pbes2bool* to solve the resulting PBES to find

out whether the equivalence relation holds or not. This way it is not needed to explicitly generate the state spaces for the two processes.

Since the tool *bisimulation* is not a standard tool in the mCRL2 toolset (revision 6517-shared), we first need to build the tool. This is done as follows. From the source directory `libraries/pbes/example` the following command needs to be run to compile the tool:

```
../../../build/bin/bjam link=static
```

This compiles the tool and places it in the subdirectory `dist`. To construct a PBES expressing the bisimulation relation between the two processes the following command is used. The parameter *bisimulation=1* tells the tool to use strong bisimulation.

```
bisimulation --bisimulation=1 <process1>.lps <process2>.lps <pbesname>.pbes
```

Then we can apply the tool *pbesrewr* (without any additional parameters) on the PBES that rewrites data expressions in a PBES. This can lead to a reduction in the time required to solve the PBES. Note that there exist tools like *pbesconstelm*, *pbesparelm* and *pbeseqelm*. We have tested these tools on some instantiations of the model, but they did not lead to improvements. Most likely this is since the LPSs that are used in the tool *bisimulation* are already transformed using a number of LPS tools before these are given to the *bisimulation* tool.

To solve the PBES we use the tool *pbes2bool* with parameters *precompile* to precompile the PBES and for large PBESs the parameter *rewriter=jittyc*. Both parameters can lead to faster rewriting, although the latter only improves the total running time on large PBESs.

```
pbes2bool --precompile [--rewriter=jittyc] <pbesname>.pbes
```

Unfortunately it takes a lot longer to check the equivalence relation using this technique compared to the previous technique using *ltscompare*, therefore the technique using *ltscompare* is preferred. Some results are presented in the following table. The column time_{lts} contains the time that is required to check the equivalence relation using the technique using *ltscompare* and the column $\text{time}_{\text{pbes}}$ contains the time it takes to run *pbes2bool* (the tools *bisimulation* and *pbesrewr* run within one second for these models). All models use the same values for parameters $d\ (= k-1)$, $s\ (= 2k-1)$ and $limit\ (= 2)$.

model	n	k	$\langle \mathbf{v} \rangle$	$\langle \pi(\mathbf{v}) \rangle$	time_{lts}	$\text{time}_{\text{pbes}}$	#equations	bisimilar
dishonest	2	1	`F,T`	`T,F`	0m01s	0m01s	52	yes
dishonest	3	2	`F,T,F`	`F,F,T`	2m05s	62m08s	8811982	yes
dishonest	3	2	`T,F,F`	`F,F,T`	2m00s	61m15s	8661350	no
strong sync.	4	2	`F,T,F,T`	`F,F,T,T`	32m15s	(*)	(*)	yes

(*): No result after running 23 hours and 15 minutes. Upon termination, more than 60GB of memory was used and more than 100 million Boolean equations were generated.

3.6.2 Model checking

Tools

In order to verify certain properties on our model we express these properties as modal μ-calculus formulae. When we have expressed a property on our model in a modal formula, we can use the tool *lps2pbes* to generate a PBES that equals true if and only if the corresponding modal formula is true on the given LPS. Besides the modal formula the tool *lps2pbes* also needs an LPS of the model as input. This LPS can be generated from the mCRL specification as described in Section 3.2. The PBES expressing the validity of the modal formula on the LPS can be constructed as follows.

```
lps2pbes --formula=<formulaname>.mcf <lpsname>.lps |
  pbesrewr > <pbesname>.pbes
```

We again use the tool *pbesrewr* in the toolchain that can rewrite data expressions in a PBES, simplify expressions and remove quantified variables that are not used. To solve a PBES we use the

tool *pbes2bool*, again with parameters *precompile* and *rewriter=jittyc* to speed up the computation of the result of the PBES. We thus use the following command to solve a PBES.

```
pbes2bool --precompile [--rewriter=jittyc] <pbesname>.pbes
```

Modal μ-formulae

In this section we describe how some properties can be expressed as modal μ-calculus formulae. In the following text we use the model with $n = 2$, $n = 3$ and $n = 4$ to denote one of the models in the following table. In the following paragraphs we discuss each property we verify in isolation. We present all formulae in standard mathematical notation, the formulae in the format of *lps2pbes* are included in Appendix A.6.

parameter	value n=2	value n=3	value n=4
model	dishonest	dishonest	strong sync.
numVoters	2	3	4
threshold	1	2	2
numDishonestVoters	threshold-1	threshold-1	threshold-1
numSigners	2*threshold-1	2*threshold-1	2*threshold-1
votesVector	`[F,T]`	`[F,T,F]`	`[F,T,F,T]`
limit	2	2	2

Always action `finalTally` We start with a property that is easy to express and can be described very short. We want to check whether the external observer is always able to eventually construct a final tally. This means that we need to check whether action `finalTally` is always eventually performed. We can formalize this using the following modal μ calculus formula.

$$[(\overline{\exists d{:}\mathbb{D}.finalTally(d)})^\star]\langle true^\star.\exists d{:}\mathbb{D}.finalTally(d)\rangle true$$

This formula expresses that as long as no action `finalTally(d)` has happened for some $d{:}\mathbb{D}$, it must be the case that there is a path for which it holds that action `finalTally(d)` is possible for some $d{:}\mathbb{D}$.

n	#equations	time	solution
2	43	0m01s	true
3	469149	2m26s	true
4	2626457	19m11s	true

At least one signature We want to check whether every blinded vote that is sent as first blinded vote is signed at least once by one of the signers. We express this using the following modal μ-calculus formula. In this formula we use the *val* operator, this operator evaluates the value of the Boolean data expression that is given as parameter to true or false.

$$\forall n : \mathbb{N}.val(n < numVoters) \rightarrow [true^\star.bcast(n, blindmsg(n, 0))]$$
$$\langle true^\star.\exists m : \mathbb{N}.bcast(m, sign(m, blindmsg(n, 0)))\rangle true$$

n	#equations	time	solution
2	68	0m01s	false
3	705724	2m00s	true
4	5349482	27m07s	true

Note that the solution is false for $n = 2$. This is since the threshold value is in that case one and the number of signers is also one. Therefore, the voter which is the signer signs his own ballot, but does not broadcast this. The answer false is thus the correct answer. The formula is true for $n = 3$ and $n = 4$ since there are two signers then.

At least k signatures As a more general version of the previous formula, we now check whether each first blinded vote is signed at least k (the threshold value, here represented by the parameter *threshold*) times. We express this using the following formula.

$$\begin{array}{l}\forall n{:}\mathbb{N}.val(n < numVoters) \rightarrow ([true^{\star}.bcast(n, blindmsg(n,0))]\\ \quad (\nu X(cnt{:}\mathbb{N} := 0).[(\exists m{:}\mathbb{N}.bcast(m, sign(m, blindmsg(n,0))))]X(cnt) \wedge\\ \quad\quad [\exists m{:}\mathbb{N}.bcast(m, sign(m, blindmsg(n,0)))]X(cnt+1) \wedge\\ \quad\quad\quad (\forall d{:}\mathbb{D}.[finalTally(d)]val((n < numSigners \rightarrow (cnt+1 \geq threshold)) \wedge\\ \quad\quad\quad\quad (n \geq numSigners \rightarrow (cnt \geq threshold))))))\end{array}$$

Let us look at this formula in detail. The first universal quantification says that we want to express something for all voters n. Then the part $[true^{\star}.bcast(n, blindmsg(n,0))]\phi$ says that we want to verify whether property ϕ holds after every action $bcast(n, blindmsg(n,0))$. That property ϕ is here a fixed point formula with data. The first part $\nu X(cnt{:}\mathbb{N} = 0)$ says that parameter *cnt* is initialized with 0. We use this parameter to count signatures. Every time a signature is received for voter n (represented by $[\exists m{:}\mathbb{N}.bcast(m, sign(m, blindmsg(n,0)))]$) the counter is increased by one. If another action occurs (represented by $[\overline{\exists m{:}\mathbb{N}.bcast(m, sign(m, blindmsg(n,0)))}]$), the counter is not increased. Upon encountering action `finalTally(d)` for some $d{:}\mathbb{D}$ it is checked whether enough signatures are received. Since signers sign their own vote, we check whether *cnt+1* $\geq$ *threshold* for signers and *cnt* $\geq$ *threshold* for voters that can not sign.

Note that in this formula duplicate messages are counted multiple times. For example, when a signer m broadcasts his signature for the blinded message of a voter n two times in a row, the parameter *cnt* is increased by one two times. This would make the formula not very useful, since a single signer could broadcast his signature k times, which makes the formula true. Fortunately, it is in our model by construction not possible for a voter to send the same message more than once (the motivation of this decision is included in Section 3.4.1). This also applies to other formulas which are discussed further on.

n	#equations	time	solution
2	86	0m01s	true
3	1404363	4m16s	true
4	10073756	62m30s	true

No signer signs twice The next property we verify is whether no signer signs the same blinded vote twice. We formalize this using the following modal μ calculus formula.

$$\begin{array}{l}\forall n{:}\mathbb{N}.val(n < numSigners) \rightarrow \forall blinder{:}\mathbb{N}.val(blinder < numVoters) \rightarrow\\ \quad \forall order{:}Nat.val(order < limit) \rightarrow [true^{\star}.bcast(n, sign(n, blindmsg(blinder, order))).true^{\star}.\\ \quad\quad bcast(n, sign(n, blindmsg(blinder, order)))]false\end{array}$$

n	#equations	time	solution
2	107	0m01s	true
3	5424986	15m05s	true
4	40772339	363m29s	true

Sound final tally The next property we verify is whether the number of ballots in the final tally is greater than or equal to the number of blinded votes that are sent by honest voters. We can unfortunately only verify this for honest voters and not for all voters. The reason for this is that a dishonest voter can decide to cast a blinded vote and not cast the corresponding ballot. Therefore we do not check whether the number of ballots in the tally is equal to the number of blinded votes that are sent first. The formula is again using the counting construct using the

largest fix point operator with data.

$$\nu X(cnt{:}\mathbb{N} := 0).([\overline{(\exists n{:}\mathbb{N}.val(n \geq numDishonestVoters) \wedge bcast(n, blindmsg(n,0)))}]X(cnt) \wedge$$
$$[\exists n{:}\mathbb{N}.val(n \geq numDishonestVoters) \wedge bcast(n, blindmsg(n,0))]X(cnt+1) \wedge$$
$$\forall d{:}\mathbb{D}.([finalTally(d)]val(cnt \leq countItems(tallyc(d)))))$$

Note the use of function *countItems* in the formula. This function is in the formula since the length operator # does not work in the *val* operator inside a modal formula. The condition we would have like to check instead of *val(cnt ≤ countItems(tallyc(d)))* is *val(cnt ≤ #(tallyc(d)))*. Unfortunately, the tool *pbes2bool* can not rewrite this expression and can therefore not solve the PBES. We have solved this by adding the function *countItems* to the model which counts the number of elements of a list of Data elements. We further discuss this issue at the end of this section.

n	#equations	time	solution
2	23	0m01s	true
3	234576	0m55s	true
4	1313230	8m05s	true

Identification of dishonest signers The last property we verify before the unicity property concerns the identification of dishonest signers. We verify whether it is always the case that a dishonest signer is in the list of dishonest signers of an honest signer if and only if it has not signed all first blinded messages of all voters. We formalize this using the following modal μ calculus formula.

$$\forall n{:}\mathbb{N}.val(n < numDishonestVoters) \rightarrow \nu X(scnt{:}\mathbb{N} := 0, vcnt{:}\mathbb{N} := 0).$$
$$([\overline{(\exists m{:}\mathbb{N}.bcast(n, sign(n, blindmsg(m,0))) \vee \exists m{:}\mathbb{N}.bcast(m, blindmsg(m,0)))}]X(scnt, vcnt) \wedge$$
$$[\exists m{:}\mathbb{N}.bcast(n, sign(n, blindmsg(m,0)))]X(scnt+1, vcnt) \wedge$$
$$[\exists m{:}\mathbb{N}.bcast(m, blindmsg(m,0))]X(scnt, vcnt+1) \wedge$$
$$(\forall m{:}\mathbb{N}.val(numDishonestVoters \leq m \wedge m < numSigners) \rightarrow$$
$$(\forall d{:}\mathbb{D}.[val(isDishonestList(d)) \wedge bcast(m,d)]$$
$$val(scnt < vcnt - 1 \Rightarrow containsItem(voter(n), dList(d)) \wedge$$
$$containsItem(voter(n), dList(d)) \Rightarrow scnt < vcnt - 1))))$$

Here again we have the same issue as with formula `soundFinalTally`. We want to test whether *voter(i) ∈ dList(d)*, but *pbes2bool* can again not rewrite this expression. Therefore have constructed the function *containsItem*, which acts as a replacement function.

n	#equations	time	solution
2	2	0m01s	true
3	234582	0m58s	true
4	1313238	7m30s	true

Unicity

Using modal formulae we can also check whether the model of the protocol satisfies the unicity property. In order to verify this property we need to adapt the model, since it is not possible to check this property in the original model. In the original model it is not possible to see how many ballots there are cast per voter and how many ballots per voter there are in the final tally. We adapt the model such that this can be checked, but by doing this the privacy property is not satisfied anymore and the adapted model should therefore only be used to check the unicity property.

The model is adapted by adding the identity of a voter to each ballot that he sends. Since the tally is formed out of a number of ballots, the tally also contains the identities of the voters.

The model is changed as follows. In the regular (dishonest) model a ballot is modelled as a list of signatures combined with an order that is used to indicate the order in which the ballot is sent.

```
ballot(listc: List(Data), order: Nat)?isBallot
```

In the model for unicity we add a Data parameter that represents the identity of the voter. For a given ballot d, the identity can be extracted using projection function *bsender*.

```
ballot(bsender: Data, listc: List(Data), order: Nat)?isBallot
```

When a ballot is sent, the identity of the voter that sends the ballot is added to the ballot. As an example the case where an honest voter V_i casts his first (and only) ballot. In the regular model it is modelled by constructing the ballot from a list of signatures and adding an order. After that the ballot is broadcasted anonymously using the `abcast` action and the $Voters$ process is called with the appropriate parameters (not shown here). In the regular model this is modelled as follows.

```
abcast(ballot(removeDishonestSignatures(kTimesDishonest(0, k, alldcList.i), filterSigByNonce(0,
sigList.i)), order))
```

In the model for unicity we add the identity of the voter to the ballot as follows.

```
abcast(ballot(voter(i), removeDishonestSignatures(kTimesDishonest(0, k, alldcList.i),
filterSigByNonce(0, sigList.i)), order))
```

The full list of differences can be found in Appendix A.5. This includes the other cases where ballots are cast by voters, changes in the functions *smaller* and *addOrderToNoncesTally* and a change in the rename file.

Using the adapted model we can check whether the model satisfies the unicity property. We check this by verifying whether there is (at most) one ballot in the final tally for each voter that has sent at least one blinded vote. Since dishonest voters do not always send a ballot after they have sent a blinded vote, it is possible that there is no ballot from a dishonest voter in the final tally. For that reason we check whether the number of ballots in the final tally per voter is less than or equal to one if the voter is a dishonest voter and equal to one when the voter is honest. We can express this using the following formula, where the function *countBallotsInTally* counts the number of ballots for a specific voter.

$$
\begin{array}{l}
\forall n{:}\mathbb{N}.val(n < numVoters) \rightarrow \\
\quad (\forall d{:}\mathbb{D}.([true^{\star}.bcast(n, blindmsg(n,0)).true^{\star}.finalTally(d)] \\
\qquad val((n < numDishonestVoters \Rightarrow countBallotsInTally(voter(n), tallyc(d)) \leq 1) \wedge \\
\qquad\quad (n \geq numDishonestVoters \Rightarrow countBallotsInTally(voter(n), tallyc(d)) = 1))))
\end{array}
$$

However, when trying to check this formula on the adapted model using the tool *pbes2bool* it seems that the universal quantification over Data elements d is a problem. The tool starts by trying to eliminate the universal quantifier for at least ten minutes without results (for the model with $n = 3$). From the semantics of μ-calculus formulae [GW05] it follows that it is allowed to move quantifiers that are outside a box or diamond modality whenever possible, that is if the same variables are still bound to the same quantifiers. For example the following equality holds.

$$\forall d{:}\mathbb{D}.[true^{\star}.a(d)]false = [true^{\star}]\forall d{:}\mathbb{D}.[a(d)]false$$

When we do this here, we get the following formula, which can be checked using *pbes2bool* within six minutes, for the model with $n = 3$. We have not tested the formula for $n = 4$ since we have not adapted the strong synchronicity model.

$$
\begin{array}{l}
\forall n{:}\mathbb{N}.val(n < numVoters) \rightarrow \\
\quad ([true^{\star}.bcast(n, blindmsg(n,0)).true^{\star}]\forall d{:}\mathbb{D}.[finalTally(d)] \\
\qquad val((n < numDishonestVoters \Rightarrow countBallotsInTally(voter(n), tallyc(d)) \leq 1) \wedge \\
\qquad\quad (n \geq numDishonestVoters \Rightarrow countBallotsInTally(voter(n), tallyc(d)) = 1)))
\end{array}
$$

n	#equations	time	solution
2	451	0m01s	true
3	1410025	5m32s	true

Extra functions

Since a number of standard operations on lists do not rewrite using the tool *pbes2bool* as discussed in the previous sections, we have replaced them by functions that do the same. Note that where the standard operations are polymorphic, our replacement functions are not since it is not possible to define your own polymorphic function in mCRL2. Our replacement functions thus only work for certain data types, which gives no problems in the way we apply them. Furthermore we have added an extra function that is needed for the verification of the unicity property. We have added the following three functions: *countItems*, *containsItem* and *countBallotsInTally*. Function *countItems* is a replacement function for the length operator # on lists, function *containsItem* is a replacement function for element test on lists and function *countBallotsInTally* is needed for the verification of the unicity property and counts for a given voter the number of ballots in the tally.

The functions are rather trivial, so we do not elaborate on them (they can be found in the models in Appendix A.3 and Appendix A.5). The first two functions, *countItems* and *containsItem*, are included in the regular model. The third function, *countBallotsInTally*, is only included in the adapted model.

3.6.3 Observations

We have shown that the privacy property, the unicity property and some less general properties hold for the regular dishonest model for up to three voters. We have also shown that these properties (except the unicity property) hold for four voters in the model with strong synchronicity. All formulae used for model checking are parameterized, that way larger models can easily be checked – given that there is enough time and space – by only generating a new model.

Unfortunately we have not been able to do model checking on a model with five voters. The reason for this is that the models with five voters are too large to model check in our environment. This is mainly due to the number of messages in our protocol and then particularly the messages introduced by signing the blinded votes. In a naïve approach this leads to $n \cdot s$ messages for the signatures in an honest model, where s is the number of signers (typically close to n) and n the number of voters. This leads to roughly $\mathcal{O}((ns)!)$ different orders in which the messages can be sent (it is not exactly $(ns)!$ since signatures for a certain blinded vote can only be sent when the blinded vote has been cast).

We have reduced the number of orders by not letting signers broadcast the signature of their own blinded vote and by using a blind-then-sign approach. Since this did not help enough we further reduced the number by applying a strong synchronicity approach. Using that approach the number of orders is (for the honest case) reduced from $\mathcal{O}((ns)!)$ in the naïve approach to $\mathcal{O}(n! \cdot s!)$. Although this did enable us to do model checking with four voters, it is still not possible to do model checking with five voters. It should be noted that the complexity of the model not just comes from sending the large number of messages, it also comes from processing these messages. For example, every signature that is sent is received by $n - 1$ voters and can often be processed in various orders.

A way to further reduce the number of orders is to put an ordering on the messages. For example when a blinded vote is sent, first voter 0 may send its signature, then voter 1 and so on. We have not investigated this since we feel that this restricts the model too much and since it is likely only a relatively small reduction on the total state space because we are still stuck with the large number of messages.

Chapter 4

Proving privacy in the TD-1 protocol

In the previous chapter we have shown how we have verified properties for a model of one our protocols. Unfortunately this was only possible for a small number of voters since the state space had to be generated (to check equivalence relations) or because all paths had to be traversed (for model-checking). Although such approach can find errors in the model and may give confidence that the protocol is also correct for a larger number of voters, it is impossible to know for sure whether it really is correct for an arbitrary number of voters.

In this chapter we prove that the privacy property is satisfied for an arbitrary number of voters n in the protocol of Section 2.4, which we call the TD-1 protocol. Recent work [CPvdPW07, OW08, OWW09] has made it possible to determine equivalence relations on models where n is kept variable. We have chosen for this relatively simple protocol since the techniques we use for the proof need to be applied by hand, involve a large number of equations and can be quite complex.

Before we start explaining the techniques, we first construct a model of the protocol. In order to apply the technique it is needed that the model is in the form of an LPE (introduced in Section 3.1). We have chosen to construct a model directly as an LPE, however it would have also been possible to construct a model using parallel processes and then to linearize it using the mCRL2 toolset.

4.1 Model

In order to keep the model small and clear, we have modelled the most basic setting. Trusted devices are *not* shared: each user has its own trusted device and therefore explicit voter registration can be left out. For the encryption of messages a static key is used instead of a dynamic key. Furthermore we assume that the trusted devices do not cache any broadcast messages: there is only one message on the broadcast channel that can be processed by the trusted devices.

The model is quite straightforward and can be found in Figure 4.1 (only the process $Voters$). The full model is included in Appendix B.1 and the rename file in Appendix B.1.1. Function *insListNat* inserts a natural number in a sorted list of natural numbers. Function *updSingleList* is similar, it inserts a ballot in a sorted list that is in a list of lists. Then, function *sanitize* takes care of the sanitization of the ballots: it removes the voter identities of the ballots such that the privacy property is not violated. The process $Voters$ works as follows, *castList* contains a list of voters that have cast their vote and is used to take care that voters can only vote once. After a ballot has been cast, the process is called with an empty *procList*. The parameter *procList* contains a list of voters that have processed the message that is currently on the broadcast channel (represented by parameter *bc*). Only when all trusted devices have processed that message, it is possible that a new ballot is cast. Since all trusted devices are honest, it suffices to only check the tally of one of them. We do this by inspecting the tally of trusted device with identity 0, which is always

in the protocol for realistic instantiations ($numVoters > 0$). Since all trusted devices can only perform honest actions we can choose a single device to read out the tally. The model has only two parameters that are used to change the model: *numVoters*, which specifies the number of voters (and thus trusted devices) in the protocol and *votes*, which contains the distribution of the votes.

```
proc Voters(N: Nat, bc: Data, votes: List(Bool), castList: List(Nat), key: Data,
            tallyList: List(List(Data)), procList: List(Nat)) =
  sum i: Nat . (0 <= i && i < N) -> (
    (!(i in castList) && #(procList) == N) ->
      bcast(i, encrypt(ballot(i, votes.i), key)) .
        Voters(N, encrypt(ballot(i, votes.i), key), votes, insListNat(i,
               castList), key, tallyList, [])
    +
    (!(i in procList)) -> (
      (decrypt(bc, key) != err) ->
        storedMessage(i, decrypt(bc, key)) .
          Voters(N, bc, votes, castList, key, updSingleList(i, decrypt(bc, key),
                 tallyList), insListNat(i, procList))
      <>
        invalidMessage(i, bc) .
          Voters(N, bc, votes, castList, key, tallyList, insListNat(i,
                 procList))
    )
    +
    (#(tallyList.i)==N && i== 0) ->
      ftally(sanitize(tallyList.i)) . delta
);
```

Figure 4.1: mCRL2 model of TD-1 protocol (main process only).

To call the process initially, the following process call should be used, where *numVoters* is the number of voters in the protocol, *votesVector* is a list of Boolean votes, *eKey* is a static key of type Data, *initList(numVoters)* initializes a list containing *numVoters* empty lists and *initProcList(numVoters)* generates the list $[0, \ldots, numVoters - 1]$.

```
Voters(numVoters, null, votesVector, [], eKey, initlist(numVoters), initProcList(numVoters))
```

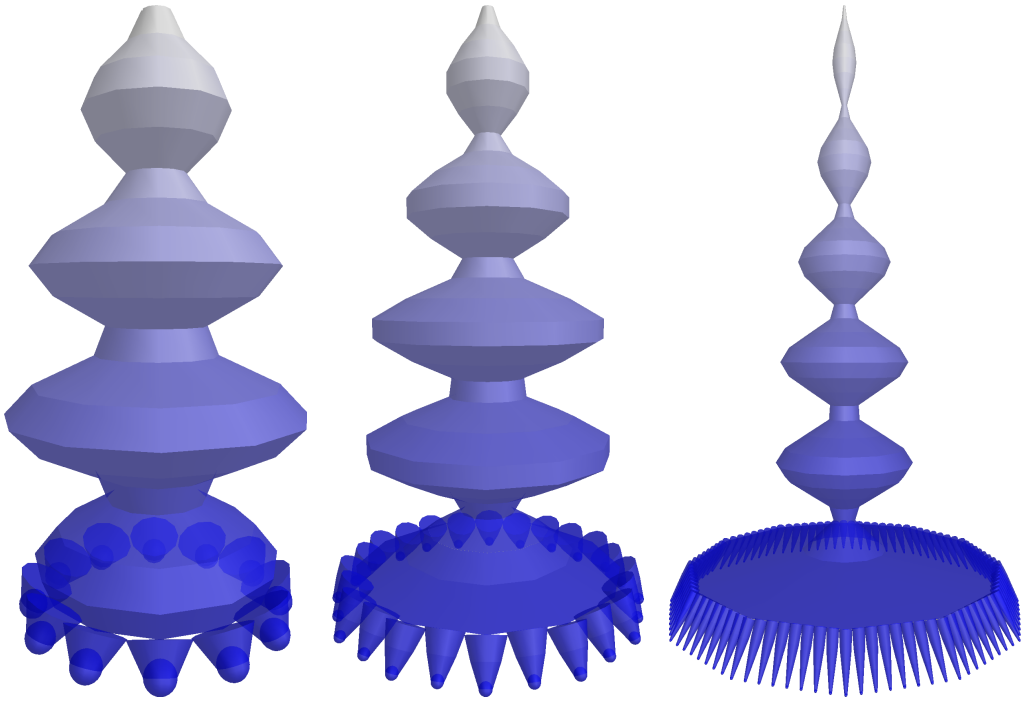
In the initial process the value *null* is on the broadcast channel. This is done since we have to define some value of type Data for the initial call. We however do not want that this message is processed, therefore we initialize the parameter *procList* with the list containing the identities of all voters. In order to generate the state space of the model we first need to generate an LPS. To construct an LPS from the model (included in Appendix B.1) and the rename file (included in Appendix B.1.1) we use the following toolchain.

```
mcrl22lps --delta <name>.mcrl | lpsconstelm | lpsactionrename
  --renamefile=<renfilename>.ren | lpssuminst | lpsconstelm > <name>.lps
```

We have generated the state space of the model for a number of parameters. The results are presented in the table below. The state spaces are generated using the tool *lps2lts* without parameters for $n \leq 6$. For $n > 6$ the parameter *rewriter=jittyc* is added to speed up state space generation. The column *time* contains the time it takes to generate a certain LTS.

n	$\langle \mathbf{v} \rangle$	time	#states	#transitions
1	F	0m01s	4	3
2	F,T	0m01s	18	24
3	F,T,F	0m01s	106	187
4	F,T,F,F	0m01s	738	1626
5	F,T,F,F,T	0m01s	4450	11721
6	F,T,F,F,T,F	0m24s	37122	115380
7	F,T,F,F,T,F,T	0m50s	255874	917651
8	F,T,F,F,T,F,T,F	12m37s	2583042	10535270

We have constructed figures of some of the LTSs using the tool *ltsview*, these figures can be found in Figure 4.2. In these figures we can see number the broadcast messages: after a ballot has been broadcast there follows a bulb. This is because there are a number of different orders in which broadcast messages can be processed. In the model it is only checked whether the number of ballots in the tally of trusted device with identity 0 is equal to the number of trusted devices before publishing the tally. It can very well be the case that a number of other trusted devices has not yet processed the last broadcast message; this is represented by the various nodes at the bottom of the figures. When we would check whether all tallies contain enough ballots, then these would be replaced by a single state at the bottom of the figure.



(a) $n = 4$, $\mathbf{v} = \langle$F,T,F,F$\rangle$ (b) $n = 5$, $\mathbf{v} = \langle$F,T,F,F,T$\rangle$ (c) $n = 6$, $\mathbf{v} = \langle$F,T,F,F,T,F$\rangle$

Figure 4.2: LTSs for model of TD-1 protocol (scaled).

We have used the tool *ltscompare* on some instantiations in order to get some intuition about whether the model is strongly bisimilar for different vote vectors. For example, we have generated two instantiations of the model with $n = 6$ and votes=[F,T,F,F,T,F] and votes=[F,F,F,T,F,T]. Using the tool *ltscompare* with parameter *equivalence=bisim* we have validated in one second that the two models are strongly bisimilar. We do not know anything about whether the equivalence holds for arbitrary n, but if we would have found out here that the equivalence did not hold for $n = 6$ we would not need to spend time trying to prove the model for arbitrary n.

4.2 PBES theory

Now that we have constructed a model of the protocol we proceed with explaining the techniques we use to verify the privacy property. Please note that this is established theory, therefore we use in this section (slightly adapted) definitions, lemmas and theorems from earlier work [CPvdPW07, OW08, OWW09] for some established concepts.

Parameterized Boolean Equation Systems

To verify the privacy property we need to check whether two instantiations of the model of the protocol are strongly bisimilar (as is explained in Section 3.6.1). As means to express the equivalence relation we use Parameterized Boolean Equation Systems (PBESs). In work of Chen et al. [CPvdPW07] it is explained how a number of equivalence relations can be expressed as a PBES. Let us first explain what a PBES is before we proceed with explaining the technique. Since a PBES contains a number of equations consisting of predicate formulae, we first define those.

Definition 1. *A* predicate formula *is a formula ϕ in positive form, defined by the following grammar:*

$$\phi ::= b \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \forall d{:}D.\ \phi \mid \exists d{:}D.\ \phi \mid X(\mathbf{e})$$

where b is a data term of Boolean sort $\mathbb{B}$, possibly containing data variables $d \in \mathcal{D}$. Furthermore, X (taken from some set $\mathcal{P}$ of predicate variables) is a (parameterized) predicate variable and $\mathbf{e}$ is a vector of data terms.

The set of all predicate formulae is denoted by Pred and a predicate formula is called a *simple* predicate formula if it does not contain predicates variables. Then some things have to be noted here. First, predicate variable X in the above definition of a predicate formula is associated with a vector $\mathbf{d_X}$ of sort $\mathbf{D_X}$. We however use, without loss of generality [OW08], a single variable d_X of sort D_X. Then we would like to introduce some shorthand notation. Since Definition 1 does not contain negation, we use for Boolean terms b, $b \implies \phi$ as a shorthand for $\neg b \vee \phi$. This is allowed since negation is an operator on data terms. Likewise, for Boolean terms b_1, b_2, we define $b_1 \iff b_2$ as a shorthand for $(\neg b_1 \vee b_2) \wedge (\neg b_2 \vee b_1)$.

Predicate formulae can contain both bound (by a universal or existential quantifier) and free data variables. We assume that a data variable is not both bound and free in the same predicate formula, that is that the sets of bound variables and the set of free variables are disjoint. Furthermore we assume an interpretation function $[\![\_]\!]$ over closed data terms (data terms that do not contain free variables). This function maps closed term e to the semantic data element $[\![e]\!]$ it represents. For open terms (terms that do contain free variables) we use a data environment ε that maps each variable from $\mathcal{D}$ to a data variable of the right sort. The interpretation of an open term e is noted as $[\![e]\!]\varepsilon$.

Definition 2. *Let θ be a* predicate environment *assigning a function of type $D_X \to \mathbb{B}$ to every predicate variable X, and let ε be a* data environment *assigning a value from domain D to every variable d of sort D. The interpretation $[\![\_]\!]\theta\varepsilon$ of a predicate formula in the context of environment θ and ε is either true or false, determined by the following induction:*

$$\begin{array}{lll}
[\![b]\!]\theta\varepsilon & =_{def} & [\![b]\!]\varepsilon \\
[\![\phi_1 \wedge \phi_2]\!]\theta\varepsilon & =_{def} & [\![\phi_1]\!]\theta\varepsilon \text{ and } [\![\phi_2]\!]\theta\varepsilon \\
[\![\phi_1 \vee \phi_2]\!]\theta\varepsilon & =_{def} & [\![\phi_1]\!]\theta\varepsilon \text{ or } [\![\phi_2]\!]\theta\varepsilon \\
[\![\forall d{:}D.\ \phi]\!]\theta\varepsilon & =_{def} & \text{for all } v \in D,\ [\![\phi]\!]\theta(\varepsilon[v/d]) \\
[\![\exists d{:}D.\ \phi]\!]\theta\varepsilon & =_{def} & \text{for some } v \in D,\ [\![\phi]\!]\theta(\varepsilon[v/d]) \\
[\![X(e)]\!]\theta\varepsilon & =_{def} & \text{true if } \theta(X)([\![e]\!]\varepsilon) \text{ and false otherwise}
\end{array}$$

Furthermore we use the notation $\phi \rightarrow \psi$ to denote that the interpretation of predicate formula ϕ implies the interpretation of predicate formula ψ.

Definition 3. *Let ϕ and ψ be predicate formulae. We write $\phi \rightarrow \psi$ iff for all predicate environments θ and all data environments ε, $[\![\phi]\!]\theta\varepsilon$ implies $[\![\psi]\!]\theta\varepsilon$.*

A *Parameterized Boolean Equation System* (PBES) is a sequence of fixed point equations. Each equation in a PBES is of the form $\sigma X(d_X{:}D_X) = \phi$, where X is a predicate variable, ϕ a predicate formula and where σ denotes the least fix point (μ) or largest fixed point (ν). Variable d_X is of sort D_X and may occur in predicate formula ϕ. The empty PBES is denoted by ϵ. In the remainder of this thesis, we use the terms Parameterized Boolean Equation System, PBES and equation system interchangeably.

An equation system is called *closed* if for all equations it contains it holds that all predicate variables at the right hand side of some equation also occur at the left hand side of some (possibly different) equation. When an equation is not closed it is called *open*. For an equation system $\mathcal{E}$ the set $\mathsf{bnd}(\mathcal{E})$ is defined as the set of predicate variables at the left hand sides of the equations. The predicate variables at the right hand sides are collected in the set $\mathsf{occ}(\mathcal{E})$.

The solution of a PBES is defined in the context of a predicate environment. It assigns to each predicate variable X a function of type $D_X \rightarrow \mathbb{B}$. This way it can be decided for every predicate variable whether the solution for the corresponding equation is true or false for a given data environment.

Expressing bisimilarity

Now that we have defined what a PBES is we can explain how we can generate a PBES expressing strong bisimilarity between two specifications S and M. Assume that these specifications S and M are given by the following LPEs, where $\mathcal{A}_\tau =_{def} \mathcal{A} \cup \{\tau\}$:

$$\begin{aligned} M(d{:}D^M) &= \sum_{\mathsf{a}\in\mathcal{A}_\tau} \sum_{e_\mathsf{a}:E^M_\mathsf{a}} c^M_\mathsf{a}(d,e) \Longrightarrow \mathsf{a}(f^M_\mathsf{a}(d,e)) \cdot M(g^M_\mathsf{a}(d,e)) \\ S(d{:}D^S) &= \sum_{\mathsf{a}\in\mathcal{A}_\tau} \sum_{e_\mathsf{a}:E^S_\mathsf{a}} c^S_\mathsf{a}(d,e) \Longrightarrow \mathsf{a}(f^S_\mathsf{a}(d,e)) \cdot S(g^S_\mathsf{a}(d,e)) \end{aligned}$$

We can use Algorithm 1 [CPvdPW07] to generate a PBES that expresses strong bisimulation between LPEs M and S. The resulting PBES resolves to true for a given data environment if and only if the processes M and S are strongly bisimilar in that environment.

Algorithm 1 Generation of a PBES encoding Strong Bisimilarity between LPEs M and S.

$\mathsf{sbisim} = \nu E$, **where**

$$\begin{aligned} E := \{ & X^{M,S}(d{:}D^M, d'{:}D^S) = \mathsf{match}^{M,S}(d,d') \wedge \mathsf{match}^{S,M}(d',d) \ , \\ & X^{S,M}(d'{:}D^S, d{:}D^M) = X^{M,S}(d,d') \ \} \end{aligned}$$

Where (for all $\mathsf{a} \in Act \wedge (p,q) \in \{(M,S),(S,M)\}$) we use the following abbreviations:

$$\mathsf{match}^{p,q}(d{:}D^p, d'{:}D^q) = \textstyle\bigwedge_{\mathsf{a}\in Act} \forall e{:}E^p_\mathsf{a}.(c^p_\mathsf{a}(d,e) \implies \mathsf{step}^{p,q}_\mathsf{a}(d,d',e));$$

$$\mathsf{step}^{p,q}_\mathsf{a}(d{:}D^p, d'{:}D^q, e{:}E^p_\mathsf{a}) = \exists e'{:}E^q_\mathsf{a}.c^q_\mathsf{a}(d',e') \wedge (f^p_\mathsf{a}(d,e) = f^q_\mathsf{a}(d',e')) \wedge X^{p,q}(g^p_\mathsf{a}(d,e), g^q_\mathsf{a}(d',e'));$$

The following lemma (which is a shortened version of a lemma by Orzan and Willemse [OW08]) says that substitution does not affect the solution of an equation system. This lemma can for example be used to remove occurrences of a certain predicate variable in the equation of another predicate variable.

Lemma 1. *Let $\mathcal{E}_0, \mathcal{E}_1, \mathcal{E}_2$ be arbitrary equation systems and let X, Y be predicate variables with $X, Y \notin \mathsf{bnd}((\mathcal{E}_i))$ for $i = 0..2$. Then:*

(Substitution) Let ϕ and ψ be arbitrary predicate formulae. Let

$$\begin{array}{lll} \mathcal{E} :\equiv & \mathcal{E}_0\ (\sigma X(d_X{:}D_X) = \phi)\ \mathcal{E}_1\ (\sigma' Y(d_Y{:}D_Y) = \psi)\ \mathcal{E}_2 & \textit{and} \\ \mathcal{F} :\equiv & \mathcal{E}_0\ (\sigma X(d_X{:}D_X) = \phi[\psi_{\langle d_Y \rangle}/Y])\ \mathcal{E}_1\ (\sigma' Y(d_Y{:}D_Y) = \psi)\ \mathcal{E}_2 & \end{array}$$

where substitution of predicate function $\psi_{\langle d_X \rangle}$ for a predicate variable X in a predicate formula ϕ is defined as follows.

$$\begin{array}{rll} b[\psi_{\langle d_X \rangle}/X] & =_{def} & b \\ Y(e)[\psi_{\langle d_X \rangle}/X] & =_{def} & \begin{cases} \psi[e/d_X] & \textit{if } Y = X \\ Y(e) & \textit{otherwise} \end{cases} \\ (\phi_1 \wedge \phi_2)[\psi_{\langle d_X \rangle}/X] & =_{def} & \phi_1[\psi_{\langle d_X \rangle}/X] \wedge \phi_2[\psi_{\langle d_X \rangle}/X] \\ (\phi_1 \vee \phi_2)[\psi_{\langle d_X \rangle}/X] & =_{def} & \phi_1[\psi_{\langle d_X \rangle}/X] \vee \phi_2[\psi_{\langle d_X \rangle}/X] \\ (\forall d{:}D.\ \phi)[\psi_{\langle d_X \rangle}/X] & =_{def} & \forall d{:}D.\ \phi[\psi_{\langle d_X \rangle}/X] \\ (\exists d{:}D.\ \phi)[\psi_{\langle d_X \rangle}/X] & =_{def} & \exists d{:}D.\ \phi[\psi_{\langle d_X \rangle}/X] \end{array}$$

Then $\mathcal{E}$ and $\mathcal{F}$ have the same solution, regardless of the predicate environments and data environments that are used.

Invariants

Invariants on PBESs are relations on the data variables of a PBES. They are predicates that are valid for certain parts of the 'parameter space' of an equation system. We use the concept of global invariants on PBESs [OW08], but before we bring the definition of that concept we first give the definition of a simple function. A function $f : V \to \mathsf{Pred}$, where V is a vector of predicate variables, is a *simple function* if for all predicate variables $X \in V$ the predicate $f(X)$ is simple.

Definition 4. *The simple function $f{:}V \to \mathsf{Pred}$ is said to be a* global invariant *for an equation system $\mathcal{E}$ iff $V \supseteq \mathsf{bnd}(\mathcal{E})$ and for each $(\sigma X(d_X{:}D_X) = \phi)$ occurring in $\mathcal{E}$, we have:*

$$f(X) \wedge \phi \ \leftrightarrow\ (f(X) \wedge \phi)\ \left[_{X_i \in V}(f(X_i) \wedge X_i(d_{X_i}))_{\langle d_{X_i} \rangle}/X_i\right]$$

where $\phi\ \left[_{X_i \in V}\phi_{i\langle d_{X_i} \rangle}/X_i\right]$ denotes consecutive substitution *for predicate formula ϕ. This is inductively defined as:*

$$\begin{array}{ll} \phi\ \left[_{X_i \in \langle\rangle}\phi_{i\langle d_{X_i} \rangle}/X_i\right] & =_{def} \phi \\ \phi\ \left[_{X_i \in \langle X_1, \ldots, X_n \rangle}\phi_{i\langle d_{X_i} \rangle}/X_i\right] & =_{def} (\phi[\phi_{1\langle d_{X_1} \rangle}/X_1])\ \left[_{X_i \in \langle X_2, \ldots, X_n \rangle}\phi_{i\langle d_{X_i} \rangle}/X_i\right] \end{array}$$

A number of properties in the global invariants theory are only defined on predicate formulae in *Predicate Formula Normal Form* (PFNF) [OWW09], which is defined as follows.

Definition 5. *A predicate formula is said to be in* Predicate Formula Normal Form *(PFNF) if it has the following form:*

$$\mathsf{Q}_1\, v_1{:}V_1. \cdots \mathsf{Q}_n\, v_n{:}V_n.h \wedge \bigwedge_{i \in I}(g_i \Longrightarrow \bigvee_{j \in J_i} X^j(\mathbf{e}^j))$$

where $X^j \in \mathcal{X}$ (the domain of predicate variables), $\mathsf{Q}_i \in \{\forall, \exists\}$, I is a (possible empty) finite index set, each J_i is a non-empty finite index set, and h and every g_i are simple formulae.

In order to prove that a simple function $f{:}V \to \mathsf{Pred}$ is a global invariant we use the following theorem [OWW09].

Theorem 2. *Let $\mathcal{E}$ be an equation system where every equation k is in PFNF:*

$$(\sigma_k X_k(\mathbf{d}_{X_k}{:}\mathbf{D}_k) = \mathsf{Q}_1^k\, v_1. \cdots \mathsf{Q}_{n_k}^k\, v_{n_k}.(h^k \wedge \bigwedge_{i \in I_k} (g_i^k \implies \bigvee_{j \in J_i} X^j(\mathbf{e}^j))))$$

Then the simple function $f{:}V \to \mathsf{Pred}$ is a global invariant for $\mathcal{E}$ if for each k:

$$\bigwedge_{i \in I_k} \bigwedge_{j \in J_i} ((f(X_k) \wedge h^k \wedge g_i^k) \to f(X^j)[\mathbf{e}^j/\mathbf{d}_{X^j}])$$

The conjunction of two global invariants is, as one might expect, again a global invariant. Formally this is stated by the following property [OW08].

Property 3. *Let $f, g{:}V \to \mathsf{Pred}$ be global invariants for an equation system $\mathcal{E}$. Then also $f \wedge g$ and $f \vee g$ are global invariants for $\mathcal{E}$.*

Using a global invariant that is strong enough to satisfy certain conditions, it is possible that the solution of predicate variable can be expressed in terms of the global invariant. This is formally expressed in the following proposition [OW08].

Proposition 1. *Let $\mathcal{E}$ be an equation system. Let f be a global invariant for $\mathcal{E}$ and assume $\mathcal{E}$ contains an equation for X of the form:*

$$(\nu X(d{:}D) = f(X) \wedge \bigwedge_{i \in I} \mathsf{Q}_1\, e_i^1{:}E_i^1 \ldots \mathsf{Q}_{m_i}\, e_i^{m_i}{:}E_i^{m_i}.\ \psi_i \implies X(g_i(d, e_i^1, \ldots, e_i^{m_i})))$$

where $\mathsf{Q}_j \in \{\forall, \exists\}$ for any j, and for all i, ψ_i are simple predicate formulae and g_i is a data term that depends only on the values of d and $e_i^1, \ldots, e_i^{m_i}$. Then X has the solution $f(X)$.

4.3 Proving privacy

In this section we apply the techniques from the previous section to prove the privacy property of the TD-1 protocol. As discussed before, we can do this by checking whether two instantiations of the model of the protocol are strongly bisimilar. Here we do *not* instantiate the model with constants like in Section 4.1, instead we prove that the equivalence relation holds for arbitrary n. What we want to show is that the processes $Voters(n, null, V, \emptyset, [\emptyset, \emptyset, \ldots, \emptyset], \{0, 1, \ldots n-1\})$ and $Voters(n, null, \pi(V), \emptyset, [\emptyset, \emptyset, \ldots, \emptyset], \{0, 1, \ldots n-1\})$ are strongly bisimilar, where V is a list of votes, $\pi(V)$ a permutation of that list and where $[\emptyset, \emptyset, \ldots, \emptyset]$ denotes a list of n empty sets. In the remainder of this section we work with a mathematical model, where we define the set of Booleans as $\mathbb{B} = \{\bot, \top\}$, where $\bot$ denotes false and $\top$ denotes true.

Since we want to compute a PBES expressing strong bisimilarity by hand, we want to have a model that is as small as possible. Therefore we have constructed the following simplified model, where $\mathbb{D}$ denotes type Data. In this model we have already applied the hiding and renaming operators. Note that in the original model there was a check whether the broadcast message could properly be decrypted. This check is removed here because it always evaluates to true since no undecryptable messages are broadcasted.

$$\begin{aligned}
&Voters(N{:}\mathbb{N}, bc{:}\mathbb{D}, votes{:}List(\mathbb{B}), cast{:}Set(\mathbb{N}), tally{:}List(Set(\mathbb{D})), proc{:}Set(\mathbb{N})) = \\
&\quad \textstyle\sum_{i=0}^{N-1}((i \notin cast \wedge \#(proc) = N) \implies \\
&\qquad\qquad bcast(i, cmsg(i)) \cdot Voters(N, enc(blt(i, votes.i)), votes, \{i\} \cup cast, tally, \emptyset) \\
&\qquad + i \notin proc \implies \tau \cdot Voters(N, bc, votes, cast, updSet(i, dec(bc), tally), \{i\} \cup proc) \\
&\qquad + (\#(tally.i) = N \wedge i = 0) \implies ftally(count(\top, tally.i), count(\bot, tally.i)) \cdot \delta \\
&\quad)
\end{aligned}$$

Furthermore we have replaced function *sanitize* by a tuple containing the number of votes containing $\bot$ and the number of votes containing $\top$. These can be computed using the function *count*:

$$
\begin{array}{ll}
\textbf{map} & count : \mathbb{B} \times List(\mathbb{D}) \to \mathbb{N}; \\
\textbf{var} & b : \mathbb{B}; \\
 & x : \mathbb{D}; \\
 & xs : List(\mathbb{D}); \\
\textbf{eqn} & count(b, []) = 0; \\
 & count(b, x \triangleright xs) = b \approx cand(x) \to 1 + count(b, xs) \diamond count(b, xs);
\end{array}
$$

We have done this since it is easier to reason about this tuple than about the less intuitive function *sanitize*. The action $ftally(sanitize(tally.0))$ is then replaced by the action $ftally(count(\top, tally.0), count(\bot, tally.0))$. Both functions produce a similar result; function *sanitize* produces a list of $count(\bot, tally.0)$ $\bot$ items concatenated with a list consisting of $count(\top, tally.0))$ $\top$ items whereas the tuple $(count(\top, tally.0), count(\bot, tally.0))$ only contains these numbers.

There is another minor difference from this model and the mCRL model. Since sets are not properly supported in revision 6517-shared (as explained in Section 3.2), we have used lists in the mCRL model where we would have wanted to use sets. Now that we are working with a mathematical model, we can use sets in the protocol description above. Therefore we replace the function *updateSingleList* by the function *updSet* that is defined as follows.

$$
\begin{array}{ll}
\textbf{map} & updSet : \mathbb{N} \times \mathbb{D} \times List(Set(\mathbb{D})) \to List(Set(\mathbb{D})); \\
\textbf{var} & i : \mathbb{N}; \\
 & x : Set(\mathbb{D}); \\
 & xs : List(Set(\mathbb{D})); \\
\textbf{eqn} & updSet(i, m, x \triangleright xs) = \\
 & \quad i > 0 \to x \triangleright updSet(i - 1, m, xs) \diamond (\{m\} \cup x) \triangleright xs);
\end{array}
$$

Since the simplified model is already an LPE, we can use Algorithm 1 to construct a PBES expressing strong bisimilarity between the model and the model where the votes are permuted. When we apply the algorithm on our model, we get the equation system in Table 4.1. In that equation system we can simplify some conjuncts using logical rewriting. Consider the following conjunct.

$$
\begin{array}{l}
\forall_{i:[0,N)}.((i \notin cast \wedge \#proc = N) \Longrightarrow \\
\quad \exists_{j:[0,N')}.(j \notin cast' \wedge \#proc' = N' \wedge (i = j \wedge cmsg(i) = cmsg(j)) \wedge \\
\qquad X(N, enc(blt(i, votes.i)), votes, \{i\} \cup cast, tally, \emptyset, N', enc(blt(j, votes'.j)), votes', \\
\qquad\quad \{j\} \cup cast', tally', \emptyset)
\end{array}
$$

Since $i = j$ needs to hold for the variable j we can eliminate the existential quantification over j, which results in the following equation. The same simplification can be made on the primed version, that is the equation where the variables with a prime mark are replaced by variables without a prime mark and vice versa.

$$
\begin{array}{l}
\forall_{i:[0,N)}.((i \notin cast \wedge \#proc = N) \Longrightarrow \\
\quad i \notin cast' \wedge \#proc' = N' \wedge \\
\qquad X(N, enc(blt(i, votes.i)), votes, \{i\} \cup cast, tally, \emptyset, N', enc(blt(i, votes'.i)), votes', \\
\qquad\quad \{i\} \cup cast', tally', \emptyset)
\end{array}
$$

Furthermore we can simplify the following equation (again, the same can be done for the primed version):

$$
\begin{array}{l}
\forall_{i:[0,N)}.((\#tally.i = N \wedge i = 0) \Longrightarrow \\
\quad \exists_{j:[0,N')}.\#tally'.j = N' \wedge j = 0 \wedge (cnt(\top, tally.0) = cnt(\top, tally'.0) \wedge \\
\qquad cnt(\bot, tally.0) = cnt(\bot, tally'.0))
\end{array}
$$

Since i and j are bound to 0, we can rewrite this equation to:

$$\#tally.0 = N \Longrightarrow \\ \quad \#tally'.0 = N' \wedge (cnt(\top, tally.0) = cnt(\top, tally'.0) \wedge cnt(\bot, tally.0) = cnt(\bot, tally'.0))$$

Using the result from Lemma 1 we can apply the following substitutions that remove the occurrences of predicate variable X' from the equations of X. The first occurrence of predicate variable X' is the following.

$$X'(N', enc(blt(i, votes'.i)), votes', \{i\} \cup cast', tally', \emptyset, N, enc(blt(i, votes.i)), votes, \\ \{i\} \cup cast, tally, \emptyset)$$

Using Lemma 1 this occurrence of predicate variable X' can be substituted by the following predicate formula containing predicate variable X instead of X'.

$$X(N, enc(blt(i, votes.i)), votes, \{i\} \cup cast, tally, \emptyset, N', enc(blt(i, votes'.i)), votes', \\ \{i\} \cup cast', tally', \emptyset)$$

$$\nu X(N{:}\mathbb{N}, bc{:}\mathbb{D}, votes{:}List(\mathbb{B}), cast{:}Set(\mathbb{N}), tally{:}List(List(\mathbb{D})), proc{:}Set(\mathbb{N}), \\ N'{:}\mathbb{N}, bc'{:}\mathbb{D}, votes'{:}List(\mathbb{B}), cast'{:}Set(\mathbb{N}), tally'{:}List(List(\mathbb{D})), proc'{:}Set(\mathbb{N}) = \\ \forall_{i:[0,N)}.((i \notin cast \wedge \#proc = N) \Longrightarrow \\ \quad \exists_{j:[0,N')}.(j \notin cast' \wedge \#proc' = N' \wedge (i = j \wedge cmsg(i) = cmsg(j)) \wedge \\ \quad\quad X(N, enc(blt(i, votes.i)), votes, \{i\} \cup cast, tally, \emptyset, N', enc(blt(j, votes'.j)), votes', \\ \quad\quad\quad \{j\} \cup cast', tally', \emptyset) \\ \wedge \\ \forall_{i:[0,N)}.(i \notin proc \Longrightarrow \\ \quad \exists_{j:[0,N')}.j \notin proc' \wedge \\ \quad\quad X(N, bc, votes, cast, updSet(i, dec(bc), tally), \{i\} \cup proc, N', bc', votes', cast', \\ \quad\quad\quad updSet(j, dec(bc'), tally'), \{j\} \cup proc') \\ \wedge \\ \forall_{i:[0,N)}.((\#tally.i = N \wedge i = 0) \Longrightarrow \\ \quad \exists_{j:[0,N')}.\#tally'.j = N' \wedge j = 0 \wedge (cnt(\top, tally.0) = cnt(\top, tally'.0) \wedge \\ \quad\quad\quad cnt(\bot, tally.0) = cnt(\bot, tally'.0)) \\ \wedge \\ \forall_{i:[0,N')}.((i \notin cast' \wedge \#proc' = N') \Longrightarrow \\ \quad \exists_{j:[0,N)}.(j \notin cast \wedge \#proc = N \wedge (i = j \wedge cmsg(i) = cmsg(j)) \wedge \\ \quad\quad X'(N', enc(blt(i, votes'.i)), votes', \{i\} \cup cast', tally', \emptyset, N, enc(blt(j, votes.j)), \\ \quad\quad\quad votes, \{j\} \cup cast, tally, \emptyset) \\ \wedge \\ \forall_{i:[0,N')}.(i \notin proc' \Longrightarrow \\ \quad \exists_{j:[0,N)}.j \notin proc \wedge \\ \quad\quad X'(N', bc', votes', cast', updSet(i, dec(bc'), tally'), \{i\} \cup proc', N, bc, votes, cast, \\ \quad\quad\quad updSet(j, dec(bc), tally), \{j\} \cup proc) \\ \wedge \\ \forall_{i:[0,N')}.((\#tally'.i = N' \wedge i = 0) \Longrightarrow \\ \quad \exists_{j:[0,N)}.\#tally.j = N \wedge j = 0 \wedge (cnt(\top, tally.0) = cnt(\top, tally'.0) \wedge \\ \quad\quad\quad cnt(\bot, tally.0) = cnt(\bot, tally'.0))$$

$$\nu X'(N'{:}\mathbb{N}, bc'{:}\mathbb{D}, votes'{:}List(\mathbb{B}), cast'{:}Set(\mathbb{N}), tally'{:}List(List(\mathbb{D})), proc'{:}Set(\mathbb{N}), \\ N{:}\mathbb{N}, bc{:}\mathbb{D}, votes{:}List(\mathbb{B}), cast{:}Set(\mathbb{N}), tally{:}List(List(\mathbb{D})), proc{:}Set(\mathbb{N})) = \\ X(N, bc, votes, cast, tally, proc, N', bc', votes', cast', tally', proc')$$

Table 4.1: Encoding of strong bisimilarity between two Voters processes.

The second occurrence of predicate variable X' is the following.

$$X'(N', bc', votes', cast', updSet(i, dec(bc'), tally'), \{i\} \cup proc', N, bc, votes, cast, \\ updSet(j, dec(bc), tally), \{j\} \cup proc)$$

Using Lemma 1 this occurrence can be replaced by the following predicate formula.

$$X(N, bc, votes, cast, updSet(j, dec(bc), tally), \{j\} \cup proc, N', bc', votes', cast', \\ updSet(i, dec(bc'), tally'), \{i\} \cup proc')$$

When we apply these simplifications and substitutions we get the equation system in Table 4.2. There the equation for predicate variable X' is removed since it is not used in the equation for predicate variable X and it is thus not needed for the computation of the solution of X.

$$\begin{array}{l}
\nu X(N{:}\mathbb{N}, bc{:}\mathbb{D}, votes{:}List(\mathbb{B}), cast{:}Set(\mathbb{N}), tally{:}List(List(\mathbb{D})), proc{:}Set(\mathbb{N}), \\
\quad N'{:}\mathbb{N}, bc'{:}\mathbb{D}, votes'{:}List(\mathbb{B}), cast'{:}Set(\mathbb{N}), tally'{:}List(List(\mathbb{D})), proc'{:}Set(\mathbb{N}) = \\
\quad \forall_{i:[0,N)}.((i \notin cast \wedge \#proc = N) \Longrightarrow \\
\qquad i \notin cast' \wedge \#proc' = N' \wedge \\
\qquad\quad X(N, enc(blt(i, votes.i)), votes, \{i\} \cup cast, tally, \emptyset, N', enc(blt(i, votes'.i)), votes', \\
\qquad\qquad \{i\} \cup cast', tally', \emptyset) \\
\quad \wedge \\
\quad \forall_{i:[0,N)}.(i \notin proc \Longrightarrow \\
\qquad \exists_{j:[0,N')}.j \notin proc' \wedge \\
\qquad\quad X(N, bc, votes, cast, updSet(i, dec(bc), tally), \{i\} \cup proc, N', bc', votes', cast', \\
\qquad\qquad updSet(j, dec(bc'), tally'), \{j\} \cup proc') \\
\quad \wedge \\
\quad \#tally.0 = N \Longrightarrow \\
\qquad \#tally'.0 = N' \wedge cnt(\top, tally.0) = cnt(\top, tally'.0) \wedge cnt(\bot, tally.0) = cnt(\bot, tally'.0) \\
\quad \wedge \\
\quad \forall_{i:[0,N')}.((i \notin cast' \wedge \#proc' = N') \Longrightarrow \\
\qquad i \notin cast \wedge \#proc = N \wedge \\
\qquad\quad X(N, enc(blt(i, votes.i)), votes, \{i\} \cup cast, tally, \emptyset, N', enc(blt(i, votes'.i)), votes', \\
\qquad\qquad \{i\} \cup cast', tally', \emptyset) \\
\quad \wedge \\
\quad \forall_{i:[0,N')}.(i \notin proc' \Longrightarrow \\
\qquad \exists_{j:[0,N)}.j \notin proc \wedge \\
\qquad\quad X(N, bc, votes, cast, updSet(j, dec(bc), tally), \{j\} \cup proc, N', bc', votes', cast', \\
\qquad\qquad updSet(i, dec(bc'), tally'), \{i\} \cup proc') \\
\quad \wedge \\
\quad \#tally'.0 = N' \Longrightarrow \\
\qquad \#tally.0 = N \wedge cnt(\top, tally.0) = cnt(\top, tally'.0) \wedge cnt(\bot, tally.0) = cnt(\bot, tally'.0)
\end{array}$$

Table 4.2: Encoding of strong bisimilarity between two Voters processes after reduction.

Now we need to solve this equation system in order to find a solution for which we can check whether it holds in the initial process call. Before we can apply the techniques from the previous section, we need to bring the predicate formula of our equation system in PFNF. This predicate formula consists of six conjuncts, which we split into six separate predicate formulae and then bring these individually in PFNF. This is done because we can reason more easily on these separate parts that on one large formula. The conjunction of these formula in PFNF can easily be brought in PFNF. Of six of the conjuncts three are very similar, therefore we only consider the first three conjuncts.

The first conjunct in the predicate formula can be written in PFNF as follows.

$$\forall_{i:[0,N)}.(((i \notin cast \wedge \#proc = N) \implies (i \notin cast' \wedge \#proc' = N')) \wedge ((i \notin cast \wedge \#proc = N) \implies X(N, enc(blt(i, votes.i)), votes, \{i\} \cup cast, tally, \emptyset, N', enc(blt(i, votes'.i)), votes', \{i\} \cup cast', tally', \emptyset)))$$

The second conjunct in the predicate formula can be written in PFNF as follows.

$$\forall_{i:[0,N)}.\exists_{j:[0,N)}.(i \notin proc \implies j \notin proc') \wedge (i \notin proc \implies X(N, bc, votes, cast, updSet(i, dec(bc), tally), \{i\} \cup proc, N', bc', votes', cast', updSet(j, dec(bc'), tally'), \{j\} \cup proc'))$$

The third conjunct is already in PFNF.

$$\#tally.0 = N \implies \#tally'.0 = N' \wedge (cnt(\top, tally.0) = cnt(\top, tally'.0) \wedge cnt(\bot, tally.0) = cnt(\bot, tally'.0))$$

We have defined a number of invariants in Table 4.3. For these invariants we need to prove that each of them is a global invariant for the equation system. The proofs can be found in Appendix C, where we also explain each invariant in words.

In predicates involving summation, $\top$ (or true) is counted as one and $\bot$ (or false) as zero.

$$\begin{aligned}
\iota_1 &: \quad N = N' \\
\iota_2 &: \quad \forall_{k,l:[0,N)}.k \notin cast \implies (blt(k, votes.k) \notin tally.l \wedge blt(k, votes.k) \neq dec(bc)) \\
\iota_3 &: \quad \forall_{k:[0,N)}.(k \in proc \iff dec(bc) \in tally.k) \\
\iota_4 &: \quad (cand(dec(bc)) \wedge dec(bc) \notin tally.0) + count(\top, tally.0) + \textstyle\sum_{k:[0,N) \wedge k \notin cast} votes.k = \\
&\qquad (cand(dec(bc')) \wedge dec(bc') \notin tally'.0) + count(\top, tally'.0) + \textstyle\sum_{k:[0,N') \wedge k \notin cast'} votes'.k \\
\iota_5 &: \quad (cand(dec(bc)) \wedge dec(bc) \notin tally.0) + count(\top, tally.0) + count(\bot, tally.0) = \\
&\qquad (cand(dec(bc')) \wedge dec(bc') \notin tally'.0) + count(\top, tally'.0) + count(\bot, tally'.0) \\
\iota_6 &: \quad \#proc = N \iff proc = \{0, \ldots, N-1\} \\
\iota_7 &: \quad (dec(bc) \notin tally.0 + \#tally.0) = N \iff cast = \{0, \ldots, N-1\} \\
\iota_8 &: \quad proc \subseteq \{0, \ldots, N-1\} \\
\iota_9 &: \quad cast = cast' \\
\iota_{10} &: \quad \#tally.0 = N \implies dec(bc) \in tally.0 \\
\iota_{11} &: \quad cast \subseteq \{0, \ldots, N-1\} \\
\iota_{12} &: \quad dec(bc) \notin tally.0 + \#tally.0 = \#cast \\
\iota_{13} &: \quad \#tally.0 \leq N \\
\\
\iota_{2'} &: \quad \forall_{k,l:[0,N')}.k \notin cast' \implies (blt(k, votes'.k) \notin tally'.l \wedge blt(k, votes'.k) \neq dec(bc')) \\
\iota_{3'} &: \quad \forall_{k:[0,N')}.(k \in proc' \iff dec(bc') \in tally'.k) \\
\iota_{6'} &: \quad \#proc' = N' \iff proc' = \{0, \ldots, N'-1\} \\
\iota_{7'} &: \quad (dec(bc') \notin tally'.0 + \#tally'.0) = N' \iff cast' = \{0, \ldots, N'-1\} \\
\iota_{8'} &: \quad proc' \subseteq \{0, \ldots, N'-1\} \\
\iota_{10'} &: \quad \#tally'.0 = N' \implies dec(bc') \in tally'.0 \\
\iota_{11'} &: \quad cast' \subseteq \{0, \ldots, N'-1\} \\
\iota_{12'} &: \quad dec(bc') \notin tally'.0 + \#tally'.0 = \#cast' \\
\iota_{13'} &: \quad \#tally'.0 \leq N'
\end{aligned}$$

Table 4.3: List of invariants for the equation system in Table 4.2.

Since we have proven that all invariants from Table 4.3 are global invariants for predicate variable X, we define predicate ι as the conjunction of all these invariants.

$$\iota = \left(\bigwedge_{i=1}^{13} \iota_i\right) \wedge \iota_{2'} \wedge \iota_{3'} \wedge \iota_{6'} \wedge \iota_{7'} \wedge \iota_{8'} \wedge \iota_{10'} \wedge \iota_{11'} \wedge \iota_{12'} \wedge \iota_{13'} \tag{4.1}$$

By using Property 3 on invariants $\iota_1, \iota_2, \ldots, \iota_{13}, \iota_{2'}, \iota_{3'}, \iota_{6'}, \iota_{7'}, \iota_{8'}, \iota_{10'}, \iota_{11'}, \iota_{12'}, \iota_{13'}$ it follows that ι, as defined in Equation 4.1, is a global invariant for our equation system. Therefore we can use this predicate ι to strengthen the equation system.

Lemma 4. *For equation X of the equation system in Table 4.2 and predicate ι as defined in Equation 4.1, it holds that:*

$$\iota \rightarrow \#tally.0 = N \implies \\ \#tally'.0 = N' \wedge count(\top, tally.0) = cnt(\top, tally'.0) \wedge count(\bot, tally.0) = count(\bot, tally'.0)$$

Proof. From ι_1 it follows that $N = N'$. Then we split the proof obligation in two parts, which we prove separately.

1. $\#tally.0 = N \implies \#tally'.0 = N$
   From $\#tally.0 = N$ in combination with invariant ι_{10} it follows that $dec(bc) \in tally.0$. The same holds for the primed version, that is from $\#tally'.0 = N$ in combination with invariant ι_{10} it follows that $dec(bc') \in tally.0$. Then from invariants ι_9 and ι_{12} it follows that $\#tally.0 = \#tally'.0$. Hence $\#tally.0 = N \implies \#tally'.0 = N$.

2. $\#tally.0 = N \implies count(\top, tally.0) = cnt(\top, tally'.0) \wedge count(\bot, tally.0) = count(\bot, tally'.0)$
   Here we can use what we have already derived: $dec(bc) \in tally.0 \wedge dec(bc') \in tally'.0 \wedge \#tally'.0 = N'$. From invariants $\iota_7, \iota_{7'}$ and ι_9 it follows that $cast = cast' = \{0, \ldots, N-1\}$. Then from invariant ι_4 it follows that $count(\top, tally.0) = count(\top, tally'.0)$. Finally, from invariant ι_5 it follows what needs to be proven, that is $count(\top, tally.0) = cnt(\top, tally'.0) \wedge count(\bot, tally.0) = count(\bot, tally'.0)$.

□

Now we can derive the solution of our equation system. This is, as shown below, the global invariant ι.

Lemma 5. *Global invariant ι, as defined in Equation 4.1, is a solution to the equation system in Table 4.2.*

Proof. From Lemma 4 it follows that the following predicate is valid.

$$\iota \leftrightarrow \iota \wedge (\#tally.0 = N \implies \#tally'.0 = N' \wedge count(\top, tally.0) = cnt(\top, tally'.0) \wedge \\ count(\bot, tally.0) = count(\bot, tally'.0))$$

Now we can use global invariant ι to strengthen the equation system. Then according to Proposition 1 it follows that global invariant ι is a solution for the equation system. □

Now we need to check whether the global invariant ι is also valid in the initial state.

Lemma 6. *The global invariant ι, as defined in Equation 4.1 holds in the initial states of the following two instantiations of the process: $Voters(n, null, V, \emptyset, [\emptyset, \emptyset, \ldots, \emptyset], \{0, 1, \ldots n-1\})$ and $Voters(n, null, \pi(V), \emptyset, [\emptyset, \emptyset, \ldots, \emptyset], \{0, 1, \ldots n-1\})$, where V is a list of votes, $\pi(V)$ a permutation of that list and where $[\emptyset, \emptyset, \ldots, \emptyset]$ denotes a list of n empty sets.*

Proof. In order to check whether the global invariant ι holds in the initial states we check per invariant from Table 4.3 whether it holds for the given instantiation. An overview of how the variables in the invariants are instantiated is given below. We only consider processes where $n > 0$. Parameters bc and bc' are instantiated with the value $null$, for this value we define that $dec(null) = err$, $cand(null) = \bot$ and $cand(err) = \bot$.

N:	n	N':	n
bc:	$null$	bc':	$null$
$votes$:	V	$votes'$:	$\pi(V)$
$cast$:	$\emptyset$	$cast'$:	$\emptyset$
$tally$:	$[\emptyset, \emptyset, \ldots, \emptyset]$	N':	$[\emptyset, \emptyset, \ldots, \emptyset]$
$proc$:	$\{0, 1, \ldots, n-1\}$	N':	$\{0, 1, \ldots n-1\}$

Below follows a list of the invariants after they have been instantiated. It must be shown that all these predicates hold.

$$
\begin{array}{ll}
\iota_1: & n = n \\
\iota_2: & \forall_{k,l:[0,n)}.k \notin \emptyset \implies (blt(k, V.k) \notin [\emptyset, \emptyset, \ldots, \emptyset].l \wedge blt(k, V.k) \neq dec(null)) \\
\iota_3: & \forall_{k:[0,n)}.(k \in \{0, 1, \ldots, n-1\} \iff dec(null) \in [\emptyset, \emptyset, \ldots, \emptyset].k) \\
\iota_4: & (cand(dec(null)) \wedge dec(null) \notin [\emptyset, \emptyset, \ldots, \emptyset].0) + count(\top, [\emptyset, \emptyset, \ldots, \emptyset].0) + \\
& \quad \sum_{k:[0,n) \wedge k \notin \emptyset} V.k = (cand(dec(null)) \wedge dec(null) \notin [\emptyset, \emptyset, \ldots, \emptyset].0) + \\
& \quad count(\top, [\emptyset, \emptyset, \ldots, \emptyset].0) + \sum_{k:[0,n) \wedge k \notin \emptyset} \pi(V).k \\
\iota_5: & (cand(dec(null)) \wedge dec(null) \notin [\emptyset, \emptyset, \ldots, \emptyset].0) + \\
& \quad count(\top, [\emptyset, \emptyset, \ldots, \emptyset].0) + count(\bot, [\emptyset, \emptyset, \ldots, \emptyset].0) = (cand(dec(null)) \wedge \\
& \quad dec(null) \notin [\emptyset, \emptyset, \ldots, \emptyset].0) + count(\top, [\emptyset, \emptyset, \ldots, \emptyset].0) + count(\bot, [\emptyset, \emptyset, \ldots, \emptyset].0) \\
\iota_6: & \#\{0, 1, \ldots, n-1\} = n \iff \{0, 1, \ldots, n-1\} = \{0, \ldots, n-1\} \\
\iota_7: & (dec(null) \notin [\emptyset, \emptyset, \ldots, \emptyset].0 + \#[\emptyset, \emptyset, \ldots, \emptyset].0) = n \iff \emptyset = \{0, \ldots, n-1\} \\
\iota_8: & \{0, 1, \ldots, n-1\} \subseteq \{0, \ldots, n-1\} \\
\iota_9: & \emptyset = \emptyset \\
\iota_{10}: & \#[\emptyset, \emptyset, \ldots, \emptyset].0 = n \implies dec(null) \in [\emptyset, \emptyset, \ldots, \emptyset].0 \\
\iota_{11}: & \emptyset \subseteq \{0, \ldots, n-1\} \\
\iota_{12}: & dec(null) \notin [\emptyset, \emptyset, \ldots, \emptyset].0 + \#[\emptyset, \emptyset, \ldots, \emptyset].0 = \#\emptyset \\
\iota_{13}: & \#[\emptyset, \emptyset, \ldots, \emptyset].0 \leq n
\end{array}
$$

Invariants 1, 6, 8, 9, 11, 13 Trivial.

Invariant 2 For all $k \in [0, N)$ we need to prove that $blt(k, V.k) \notin [\emptyset, \emptyset, \ldots, \emptyset]$ which is true and that $blt(k, V.k) \neq dec(null)$ which is also true since $dec(null)$ rewrites to the error value err.

Invariant 3 This invariant does not hold for the given instantiation. This is because $\forall_{k:[0,N)}.k \in \{0, 1, \ldots, n-1\}$ evaluates to true, whereas $\forall_{k:[0,N)}.dec(null) \in [\emptyset, \emptyset, \ldots \emptyset].k$ evaluates to false. So, although it is an invariant for the process, it does not hold for our initial set of parameters. This is desired behaviour, since if the invariant would hold for this initial set, then this would mean that for all $k \in [0, n]$ the value $dec(null) = err$ is in $tally.k$. We can unfortunately not change the set of initial parameters: since bc is of type $Data$ we have to give some value as initial parameter and we have chosen for the value $null$. We however do not want that this message is actually processed, therefore we set $proc$ to $\{0, 1, \ldots, N-1\}$. In order to fix this problem we strengthen our invariant ι_3 to $bc \neq null \implies \forall_{k:[0,N)}.(k \in proc \iff dec(bc) \in tally.k)$. It is easy to see that this strengthened invariant holds for the initial state. The proof of invariant ι_3 remains the same: since value $null$ is never used inside the process, $bc \neq null$ always evaluates to true.

Invariant 4 Since $cand(dec(null)) = cand(err)$ is defined as $\bot$, we can rewrite the proof obligation to $\sum_{k:[0,n)} V.k = \sum_{k:[0,n)} \pi(V).k$. This holds since $\pi(V)$ is a permutation of V.

Invariant 5 Since $cand(dec(null))$ is defined as $\bot$, we can rewrite the proof obligation to the following predicate: $count(\top, [\emptyset, \emptyset, \ldots, \emptyset].0) + count(\bot, [\emptyset, \emptyset, \ldots, \emptyset].0) = count(\top, [\emptyset, \emptyset, \ldots, \emptyset].0) + count(\bot, [\emptyset, \emptyset, \ldots, \emptyset].0)$, which holds since $count(b, \emptyset) = 0$ for all $b \in \mathbb{B}$.

Invariant 7 Term $dec(null) \notin [\emptyset, \emptyset, \ldots, \emptyset].0$ rewrites to one and term $\#[\emptyset, \emptyset, \ldots, \emptyset].0$ to zero. Therefore this invariant does not hold for $n = 1$. We strengthen this invariant in the same way as is done for invariant ι_3. That is, $bc \neq null \implies (dec(bc) \notin tally.0 + \#tally.0) = N \iff cast = \{0, \ldots, N-1\}$. Then it is easy to see that the invariant holds.

Invariant 10 Term $\#[\emptyset, \emptyset, \ldots, \emptyset].0 = n$ rewrites to false for $n > 0$, hence the implication is true.

Invariant 12 This invariant also does not hold due to the fact that bc is instantiated with $null$. Therefore we also strengthen this invariant in the same way as is done for invariants ι_3 and ι_7. The invariant then becomes: $bc \neq null \implies dec(bc) \notin tally.0 + \#tally.0 = \#cast$.

The primed invariants $\iota_{2'}, \iota_{3'}, \iota_{6'}, \iota_{7'}, \iota_{8'}, \iota_{10'}, \iota_{11'}, \iota_{12'}, \iota_{13'}$ can be proven to hold in the same way. Since we have now shown that all invariants hold for the initial set of parameters, it follows that the conjunction of all those invariants, the global invariant ι, also holds for the initial set of parameters. □

From the results in this section we can conclude that the process instantiations are strongly bisimilar and therefore we can conclude that privacy is guaranteed.

Theorem 7. *The processes* $Voters(n, null, V, \emptyset, [\emptyset, \emptyset, \ldots, \emptyset], \{0, 1, \ldots n-1\})$ *and* $Voters(n, null, \pi(V), \emptyset, [\emptyset, \emptyset, \ldots, \emptyset], \{0, 1, \ldots n-1\})$ *are strongly bisimilar for* $n > 0$*, where* V *is a list of votes,* $\pi(V)$ *a permutation of that list and where* $[\emptyset, \emptyset, \ldots, \emptyset]$ *denotes a list of* n *empty sets.*

Proof. The equation system in Table 4.2 expresses that process $Voters(n, null, V, \emptyset, [\emptyset, \emptyset, \ldots, \emptyset], \{0, 1, \ldots n-1\})$ and $Voters(n, null, \pi(V), \emptyset, [\emptyset, \emptyset, \ldots, \emptyset], \{0, 1, \ldots n-1\})$ are strongly bisimilar. From Property 3 and the invariants in Table 4.3 it follows that ι, as defined in Equation 4.1, is a global invariant for the equation system. From Lemma 4 and Lemma 5 it follows that ι is a solution for the equation system. Then Lemma 6 states that the invariant also holds for the specific instantiation we want to prove. Hence the two processes are strongly bisimilar. Therefore it is for an external observer impossible to distinguish between the two, and thus privacy is guaranteed for all $n > 0$. □

Chapter 5

Conclusion

In this thesis we have introduced the concept of ad-hoc e-voting. For this setting we have added two properties to the standard list of properties that a voting protocol ideally satisfies, namely the on-line property and the walk-away property. It turns out that the ad-hoc e-voting setting, although simpler than the traditional one, does not allow a fundamentally simpler solution for e-voting. This is because the main e-voting puzzles remain: how to satisfy both the privacy and unicity property, where to place trust and how to establish this trust. However, anonymous broadcast and trusted devices are two instruments that are better imaginable in a mobile devices context than in a traditional wired network and they do simplify e-voting in practice.

We have developed four voting protocols for the ad-hoc e-voting setting: two using regular mobile devices and two using trusted devices. For each of these protocols we have using informal analysis determined which properties are likely satisfied. Furthermore we have used formal verification methods to verify some properties: we have verified for small numbers of voters that the poor man's protocol satisfies the privacy and unicity property and we have verified the privacy property of the TD-1 protocol for an arbitrary number of voters. We have summarized these results in the following table, where properties that are formally verified are marked with an asterisk.

	Poor man's	Shuffling	TD-1	TD (2)
general properties				
fairness	*partial*	✓	✓	✓
privacy	✓*	✓	✓*	✓
receipt-freeness	✗	✗	✓	✓
individual verifiability	✓	✓	✗	✗
universal verifiability	✓	*partial*	✗	✗
unicity	✓*	✓	✓	✓
coercion-resistance	✗	✗	✗	✗
ad-hoc properties				
on-line property	*partial*	*partial*	✓	✓
walk-away property	*partial*	*partial*	✓	✓

In this table we see that the protocols using regular mobile devices only partially support the on-line and walk-away property, whereas the protocols using trusted devices fully support these properties. This is due to the fact that trusted devices provide a safe method to create and store ballots that regular devices can not provide. On the trusted devices we have studied it is not possible to illegally alter data, therefore we know for example that when a value for the secret sharing method (second protocol using trusted devices) is published, that this value is correct. Using regular devices, voters can alter data and thus construct invalid values, which can often not easily be identified among valid values. To prevent this type of dishonesty, in many cases complex

commitment schemes and zero-knowledge proofs have to be introduced. This is not wanted in an ad-hoc setting since this introduces a number of extra messages and often adds computational intensive cryptography. The choice for an ad-hoc e-voting protocol thus depends on the situation: only when trusted devices are available we can choose a protocol that fully satisfies the ad-hoc properties.

Verification of e-voting protocols remains a huge challenge. We have used different verification methods: we have constructed a model of the poor man's protocol in mCRL2 and we have used PBESs to prove an equivalence relation on a model of the TD-1 protocol. The mCRL2 specification language is an expressive language that is well-suited for modelling protocol details. It is supported by a toolset that allows various manipulations, such that user understanding can be properly exploited in the verification process. For example, in Section 3.3 we give some intelligent optimizations and Section 3.4 contains a reasonable dishonesty model. A general unguided dishonesty model would not be tractable by the toolset, even with our model it is only possible to verify properties for a small number of voters, as can be seen in the results in Section 3.6. The technique using invariants on PBESs requires a model with a high level of abstraction, but comes with a major advantage over methods on state spaces: it can be used to verify the privacy property for an arbitrary number of voters. But even a relatively simple protocol like the TD-1 protocol needs a large number of invariants in order to prove an equivalence relation, as we have shown in Section 4.3.

We encountered some minor technical limitations of the mCRL2 toolset: the implementation of sets is not fully completed, which we have solved by using sorted lists instead, and the tool *lpsactionrename* does not properly import functions from a rename file, which we have solved by including these functions in the model (both discussed in Section 3.2). Furthermore the tool *pbes2bool* does not support the standard operators on lists, which we have solved by constructing replacement functions (discussed in Section 3.6.2). All in all, there were only some small technical problems that we were able to solve easily.

Bibliography

[ABF07] Martín Abadi, Bruno Blanchet, and Cédric Fournet. Just fast keying in the pi calculus. *ACM Transactions on Information and System Security (TISSEC)*, 10(3):1–59, July 2007.

[AGGV05] Gildas Avoine, Felix C. Gärtner, Rachid Guerraoui, and Marko Vukolic. Gracefully degrading fair exchange with security modules. In *Proceedings EDCC'05*, volume 3463 of *Lecture Notes in Computer Science*, pages 55–71, 2005.

[BCE+97] Frazer Bennett, David Clarke, Joseph B. Evans, Andy Hopper, Alan Jones, and David Leask. Piconet: Embedded mobile networking. *IEEE Personal Communications*, 4(5):8–15, 1997.

[BCP07] Emmanuel Bresson, Olivier Chevassut, and David Pointcheval. Provably-secure authenticated group Diffie-Hellman key exchange. *ACM Transactions on Information and System Security*, 10(3):10, July 2007.

[Ben06] Josh Benaloh. Simple verifiable elections. In *EVT'06: Proceedings of the USENIX/Accurate Electronic Voting Technology Workshop 2006 on Electronic Voting Technology Workshop*, Berkeley, CA, USA, 2006. USENIX Association.

[BFHB05] Jeremy P. Birnholtz, Thomas A. Finholt, Daniel B. Horn, and Sung Joo Bae. Grounding needs: achieving common ground via lightweight chat in large, distributed, ad-hoc groups. In *Proceedings of ACM CHI 2005 Conference on Human Factors in Computing Systems*, volume 1 of *Large communities*, pages 21–30, 2005.

[Bla01] Bruno Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *14th IEEE Computer Security Foundations Workshop (CSFW '01)*, pages 82–96, Washington - Brussels - Tokyo, June 2001. IEEE.

[BOGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, STOC'88 (Chicago, IL, May 2-4, 1988)*, pages 1–10, New York, 1988. ACM, ACM Press.

[Bra06] Felix Brandt. Efficient cryptographic protocol design based on distributed El Gamal encryption. In Dongho Won and Seungjoo Kim, editors, *International Conference on Information Security and Cryptology*, volume 3935 of *Lecture Notes in Computer Science*, pages 32–47. Springer, 2006.

[BT94] Josh Benaloh and Dwight Tuinstra. Receipt-free secret-ballot elections. In *Proceedings of the 26th Annual Symposium on the Theory of Computing*, pages 544–553, New York, 1994. ACM Press.

[BT07] Anne Broadbent and Alain Tapp. Information-theoretic security without an honest majority. In Kaoru Kurosawa, editor, *ASIACRYPT*, volume 4833 of *Lecture Notes in Computer Science*, pages 410–426. Springer, June 13 2007.

[BW90] Jos Baeten and Peter Weijland. *Process Algebra*, volume 18 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, UK, 1990.

[Cet08] Orhan Cetinkaya. Analysis of security requirements for cryptographic voting protocols (extended abstract). In *ARES*, pages 1451–1456. IEEE Computer Society, 2008.

[Cha81] David Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.

[Cha88] David Chaum. Elections with unconditionally- secret ballots and disruption equivalent to breaking RSA. In *Advances in Cryptology (EUROCRYPT '88)*, volume 330, pages 177–182, Berlin - Heidelberg - New York, May 1988. Springer.

[CPvdPW07] Taolue Chen, Bas Ploeger, Jaco van de Pol, and Tim A. C. Willemse. Equivalence checking for infinite systems using parameterized boolean equation systems. In Luís Caires and Vasco Thudichum Vasconcelos, editors, *CONCUR*, volume 4703 of *Lecture Notes in Computer Science*, pages 120–135. Springer, 2007. Extended version: CS-Report 07-14 of TU/e.

[DH76] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE Transactions on information Theory*, 22(6):644–654, 1976.

[DiM02] Joan Morris DiMicco. Mobile ad hoc voting. In *Proceedings of CHI Workshop on Mobile Ad-Hoc Collaboration*, 2002.

[DJ01] Ivan Damgård and Mads Jurik. A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In Kwangjo Kim, editor, *Public Key Cryptography*, volume 1992 of *Lecture Notes in Computer Science*, pages 119–136. Springer, 2001.

[DJ03] Ivan Damgård and Mads Jurik. A length-flexible threshold cryptosystem with applications. In Reihaneh Safavi-Naini and Jennifer Seberry, editors, *Information Security and Privacy, 8th Australasian Conference, ACISP 2003, Wollongong, Australia, July 9-11, 2003, Proceedings*, volume 2727 of *Lecture Notes in Computer Science*, pages 350–364. Springer, 2003.

[FOO92] Atsushi Fujioka, Tatsuaki Okamoto, and Kazuo Ohta. A practical secret voting scheme for large scale elections. In Jennifer Seberry and Yuliang Zheng, editors, *Advances in Cryptology—AUSCRYPT '92*, volume 718 of *Lecture Notes in Computer Science*, pages 244–251, Gold Coast, Queensland, Australia, December 1992. Springer-Verlag.

[FPS01] Pierre-Alain Fouque, Guillaume Poupard, and Jacques Stern. Sharing decryption in the context of voting or lotteries. In *FC '00: Proceedings of the 4th International Conference on Financial Cryptography*, pages 90–104, London, UK, 2001. Springer-Verlag.

[GMR85] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. In *Proceedings of the 17th Annual Symposium on Theory of Computing (STOC)*, pages 291–304, Providence, RI USA, 1985. ACM Press.

[GMR+06] Jan Friso Groote, Aad Mathijssen, Michel Reniers, Yaroslav Usenko, and Muck van Weerdenburg. The formal specification language mCRL2. In Ed Brinksma, David Harel, Angelika Mader, Perdita Stevens, and Roel Wieringa, editors, *MMOSS*, volume 06351 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play ANY mental game. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, STOC'87 (New York City, May 25–27, 1987)*, pages 218–229, New York, 1987. ACM, ACM Press.

[GP95] Jan Friso Groote and Alban Ponse. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef, and S. F. M. van Vlijmen, editors, *Algebra of Communicating Processes '94*, Workshops in Computing Series, pages 26–62. Springer-Verlag, 1995.

[Gro04] Jens Groth. Efficient maximal privacy in boardroom voting and anonymous broadcast. In Ari Juels, editor, *Financial Cryptography, 8th International Conference, FC 2004, Key West, FL, USA, February 9-12, 2004. Revised Papers*, volume 3110 of *Lecture Notes in Computer Science*, pages 90–104. Springer, 2004.

[GW05] Jan Friso Groote and Tim A.C. Willemse. Model-checking processes with data. *Science of Computer Programming*, 56(3):251–274, 2005.

[HS00] Martin Hirt and Kazue Sako. Efficient receipt-free voting based on homomorphic encryption. In Bart Preneel, editor, *Advances in Cryptology – EUROCRYPT ' 2000*, volume 1807 of *Lecture Notes in Computer Science*, pages 539–556. Springer-Verlag, Berlin Germany, 2000.

[HZ06] Feng Hao and Piotr Zieliński. A 2-round anonymous veto protocol. In *the 14th International Workshop on Security Protocols*, 2006.

[JRSW04] David Jefferson, Aviel D. Rubin, Barbara Simons, and David Wagner. Analyzing internet voting security. *Communications of the ACM*, 47(10):59–64, 2004.

[KGH83] Ehud D. Karnin, Jonathan W. Greene, and Martin E. Hellman. On secret sharing systems. *IEEE Transactions on Information Theory*, 29(1):35–41, 1983.

[KKL01] Jinho Kim, Kwangjo Kim, and Chulsoo Lee. An efficient and provably secure threshold blind signature. In Kwangjo Kim, editor, *ICISC*, volume 2288 of *Lecture Notes in Computer Science*, pages 318–327. Springer, 2001.

[KR05] Steve Kremer and Mark Ryan. Analysis of an electronic voting protocol in the applied pi calculus. In Shmuel Sagiv, editor, *ESOP: 14th European Symposium on Programming*, volume 3444 of *Lecture Notes in Computer Science*, pages 186–200. Springer, 2005.

[KY02] Aggelos Kiayias and Moti Yung. Self-tallying elections and perfect ballot secrecy. In David Naccache and Pascal Paillier, editors, *Public Key Cryptography, 5th International Workshop on Practice and Theory in Public Key Cryptosystems, PKC 2002, Paris, France, February 12-14, 2002, Proceedings*, volume 2274 of *Lecture Notes in Computer Science*, pages 141–158. Springer, 2002.

[KY03] Aggelos Kiayias and Moti Yung. Non-interactive zero-sharing with applications to private distributed decision making. In Rebecca N. Wright, editor, *Financial Cryptography*, volume 2742 of *Lecture Notes in Computer Science*, pages 303–320. Springer, 2003.

[LCPD07] Zhengqin Luo, Xiaojuan Cai, Jun Pang, and Yuxin Deng. Analyzing an electronic cash protocol using applied pi calculus. In Jonathan Katz and Moti Yung, editors, *ACNS*, volume 4521 of *Lecture Notes in Computer Science*, pages 87–103. Springer, 2007.

[LJY02] Chin-Laung Lei, Wen-Shenq Juang, and Pei-Ling Yu. Provably secure blind threshold signatures based on discrete logarithm. *Journal of Information Science and Engineering*, 18(1):23–39, 2002.

[MM06] Ülle Madise and Tarvi Martens. E-voting in Estonia 2005. the first practice of country-wide binding internet voting in the world. In Robert Krimmer, editor, *Electronic Voting*, volume 86 of *Lecture Notes in Informatics*, pages 15–26. GI, 2006.

[MR07] Aybek Mukhamedov and Mark Ryan. Anonymity protocol with identity escrow and analysis in the applied π-calculus. In Gilles Barthe and Cédric Fournet, editors, *TGC*, volume 4912 of *Lecture Notes in Computer Science*, pages 330–346. Springer, 2007.

[OMA+99] Miyako Ohkubo, Fumiaki Miura, Masayuki Abe, Atsushi Fujioka, and Tatsuaki Okamoto. An improvement on a practical secret voting scheme. In *ISW '99: Proceedings of the Second International Workshop on Information Security*, pages 225–234, London, UK, 1999. Springer-Verlag.

[OU98] Tatsuaki Okamoto and Shigenori Uchiyama. A new public-key cryptosystem as secure as factoring. In Kaisa Nyberg, editor, *Advances in Cryptology – EUROCRYPT ' 98*, volume 1403 of *Lecture Notes in Computer Science*, pages 308–318. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 1998.

[OW08] Simona Orzan and Tim A. C. Willemse. Invariants for parameterised boolean equation systems. In Franck van Breugel and Marsha Chechik, editors, *CONCUR*, volume 5201 of *Lecture Notes in Computer Science*, pages 187–202. Springer, 2008. Extended version: CS-Report 08-17 of TU/e.

[OWW09] Simona Orzan, Wieger Wesselink, and Tim A.C. Willemse. Static analysis techniques for parameterised boolean equation systems. In S. Kowalewski and A. Philippou, editors, *TACAS*, volume 5505 of *Lecture Notes in Computer Science*, pages 230–245. Springer, 2009.

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *Advances in Cryptology – EUROCRYPT ' 99*, volume 1599 of *Lecture Notes in Computer Science*, pages 223–238. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 1999.

[Sch99] Berry Schoenmakers. A simple publicly verifiable secret sharing scheme and its application to electronic voting. In Michael Wiener, editor, *Advances in Cryptology – CRYPTO ' 99*, volume 1666 of *Lecture Notes in Computer Science*, pages 148–164. International Association for Cryptologic Research, Springer-Verlag, Berlin Germany, 1999.

[Sha79] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, November 1979.

[SNS88] Jennifer G. Steiner, Clifford Neuman, and Jeffrey I. Schiller. Kerberos: An authentication service for open network systems. In USENIX Association, editor, *USENIX Conference Proceedings (Dallas, TX, USA)*, pages 191–202. USENIX, Winter 1988.

[wpa] mCRL2 wiki page. User manual: FAQ. `http://www.mcrl2.org/mcrl2/wiki/index.php/User_manual/FAQ#Support_for_sets_and_bags`. Accessed June 11th, 2009.

[wpb] mCRL2 wiki page. User manual: lpsactionrename. `http://www.mcrl2.org/mcrl2/wiki/index.php/User_manual/lpsactionrename`. Accessed June 11th, 2009.

[wpc] mCRL2 wiki page. User manual: Toolset overview. `http://www.mcrl2.org/mcrl2/wiki/index.php/User_manual/Toolset_overview`. Accessed June 11th, 2009.

Appendix A

Poor man's model

All models in this appendix need to be transformed into an LPS as follows.

```
mcrl22lps --delta <name>.mcrl | lpsconstelm | lpsactionrename
  --renamefile=<renfilename>.ren | lpssuminst | lpsconstelm > <name>.lps
```

To construct an LTS from this LPS the tool *lps2lts* is used as follows.

```
lps2lts -v [--rewriter=jittyc] <name>.lps <name>.svc
```

A.1 Honest parallel model

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% SORTS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Definition of a nonce.
sort Nonce = struct nonce?isPlainNonce |
                    ordNonce(order: Nat)?isOrdNonce;

%General datatype.
sort  Data =
  struct sign(signer: Nat, smsg: Data)?isSign |
         blind(blinder: Nat, bmsg: Data)?isBlind |
         vote(cand: Bool, nonce: Nonce)?isVote |
         ballot(listc: List(Data), order: Nat)?isBallot|
         blindmsg(blinderm: Nat)?isBlindMsg |
         err?isErr; % error value

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% ACTIONS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
act recv: Nat#Data;
act give: Nat#Data;
act bcast: Nat#Data;
act request: Nat#Data;
act send: Nat#Data;
act retrieve: Nat#Data;
act selfSigned: Nat#Data;
act unblinded: Nat#Data;
act cannotUnblind: Nat#Data;
act receivedBallot: Nat#Data;
%anon
act done;
act sdone;
act rdone;
act asend: Data;
act arecv: Data;
act abcast: Data;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% FUNCTIONS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Define an order on Data elements (only for the ones that are needed).
```

```
map smaller: Data#Data -> Bool;
var d,e: Data;
    i,j: Nat;
    l,m: List(Data);
eqn smaller(sign(i,d), sign(j,e)) = i < j;
    smaller(ballot(l,i), ballot(m,j)) = i < j;

% unblind: set of equations indicating what Data constructs can be unblinded. If
% a certain construct can not be unblinded, the term reduces to the error value
% 'err'.
map unblind: Nat#Data -> Data;
var i,j,k: Nat;
    x: Data;
eqn unblind(i, sign(k, blind(j, x))) = if(i==j, sign(k, x), err);
    unblind(i, blind(j, x)) = if(i==j, x, err);
    !(isBlind(x) || isSign(x)) -> unblind(i,x) = err;
    !(isBlind(x)) -> unblind(i, sign(j, x)) = err;

% insListData: function that inserts a Data item in a sorted list.
map insListData: Data#List(Data) -> List(Data);
var x,d: Data;
    xs: List(Data);
eqn insListData(d, []) = [d];
    insListData(d, x |> xs) = if(smaller(d, x),
                                 d |> (x |> xs),
                                 x |> insListData(d, xs));

% addOrderToNonces; Function that replaces the generic 'plain' nonce : Nonce in
% each signed (vote, nonce) pair by a nonce which has an order. This is done in
% order to be able to distinguish between different ballots. If the plain nonces
% were kept, one can not properly distinguish between two ballots. The only dif-
% ference would then be the order that is in ballot(listc: List(Data), order:
% Nat). However, this order can easily be adapted since it is plain text in
% the tuple. Hence a voter can abuse this to add extra ballots.
%
% The meaning of ordNonce(i) is that it is the i^th nonce observed on the anon-
% ymous broadcast channel.
map addOrderToNonces: List(Data)#Nat -> List(Data);
var x: Data;
    xs: List(Data);
    order: Nat;
eqn addOrderToNonces([], order) = [];
    isSign(x) ->
        addOrderToNonces(x |> xs, order) =
              if(isVote(smsg(x)),
                 sign(signer(x), vote(cand(smsg(x)), ordNonce(order))) |>
                      addOrderToNonces(xs, order),
                 x |> addOrderToNonces(xs, order));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% PROCESSES %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% bchannel: The broadcast channel. This process models both the authenticated
% as well as the anonymous broadcast channel. Although it only accepts messages
% on the anonymous broadcast channel after it has received a 'done' action from
% all n voters.
%
% Note that there is some intelligence on the anonymous part of the broadcast
% channel. The anonymous broadcast channel only opens when N voters have
% performed a sdone action. Then, if the channel is opened the channel only
% receives ballots. This is done (for now) to add an ordering on the ballots.
proc bChannel(l: List(Data), N: Nat, order: Nat) =
  sum d:Data, i:Nat . recv(i, d) . bChannel(l <| d, N, order) +
  sum i: Nat . (i < #(l)) -> give(i, l.i) . bChannel(l, N, order) +
  (N > 0) -> rdone . bChannel(l, Int2Nat(N-1), order) +
  (N == 0) -> sum d:List(Data) .
                arecv(ballot(d, order)) .
                  bChannel(l <| ballot(d, order), N, order+1);
```

```
%Voter registration process
proc regVoter(i: Nat, v: Bool, N: Nat) =
    send(i, blind(i, vote(v, nonce))) .
      Voter(i, [sign(i, vote(v, nonce))], [], 0, N);

%Voter process
proc Voter(i: Nat, sigList: List(Data), tally: List(Data), j: Nat, N: Nat) =
  sum d: Data. request(j, d) . (
      isBlind(d) ->
        ((blinder(d) != i) ->
          send(i, sign(i, d)) .
            Voter(i, sigList, tally, j+1, N)
       <> selfSigned(i, d) .
            Voter(i, sigList, tally, j+1, N))
    + isSign(d) ->
        (unblind(i, d) != err) ->
          unblinded(i, d) .
            Voter(i, insListData(unblind(i, d), sigList), tally, j+1, N)
       <> cannotUnblind(i, d) .
            Voter(i, sigList, tally, j+1, N)
    + isBallot(d) ->
        receivedBallot(i, d) .
          Voter(i, sigList, insListData(d, tally), j+1, N)
    )
    + (#sigList == N) -> sdone .
        sum o: Nat . asend(ballot(sigList, o)) .
          Voter(i, sigList, tally, j, N);

% numVoters: defines the number of voters in the protocol.
map numVoters: Nat;
eqn numVoters = 2;

init  hide({retrieve, unblinded, cannotUnblind, receivedBallot, selfSigned,
            done},
      allow({bcast, retrieve, unblinded, cannotUnblind, receivedBallot,
             selfSigned, abcast, done},
      comm({recv|send->bcast, request|give->retrieve, rdone|sdone-> done,
            asend|arecv->abcast},
          bChannel([], numVoters, 0) ||
          regVoter(0, true, numVoters) ||
          regVoter(1, false, numVoters)
      )
    )
  );
```

A.1.1 Honest parallel model rename file

```
var i: Nat;
    d: Data;

%rename functions
rename
% bcast(i,d) gets rewritten to bcast(i, blindmsg(j)) iff d is a blinded message
% and where j is the index of the voter who blinded the message
isBlind(d) -> bcast(i, d) => bcast(i, blindmsg(blinder(d)));
% bcast(i,d) gets rewritten to bcast(i, sign(j, blindmsg(k))) iff d is a signed
% message containing a blinded message. Identifier j represents the index of the
% voter who signed the message (which is needed to sign the rewritten message)
% and index k represents the index of the voter who blinded the message.
isSign(d) && isBlind(smsg(d)) -> bcast(i, d) =>
    bcast(i, sign(signer(d), blindmsg(blinder(smsg(d)))));

% abcast(d) gets rewritten to abcast(d') when d is a ballot. Data item d' con-
% tains the signature list and the order-number of ballot d, where in the list
% of signatures every nonce is converted to an ordered nonce.
isBallot(d) ->
  abcast(d) => abcast(ballot(addOrderToNonces(listc(d), order(d)), order(d)));
```

A.2 Honest linear model

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% SORTS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Definition of a nonce.
sort Nonce = struct nonce?isPlainNonce |
                    ordNonce(order: Nat)?isOrdNonce;

%General datatype.
sort  Data =
  struct sign(signer: Nat, smsg: Data)?isSign |
         blind(blinder: Nat, bmsg: Data)?isBlind |
         vote(cand: Bool, nonce: Nonce)?isVote |
         ballot(listc: List(Data), order: Nat)?isBallot |
         blindmsg(blinderm: Nat)?isBlindMsg |
         err?isErr; % error value

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% ACTIONS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
act abcast: Data;              %anonymous broadcast
act bcast: Nat#Data;           %authenticated broadcast
act cannotUnblind: Nat#Data;   %action when incoming message cannot be unblinded
act unblinded: Nat#Data;       %action when incoming message is unblinded
act receivedBallot: Nat#Data; %action when a ballot is received

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% FUNCTIONS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Define an order on Data elements (only for the ones that are needed).
map smaller: Data#Data -> Bool;
var d,e: Data;
    i,j: Nat;
    l,m: List(Data);
eqn smaller(sign(i,d), sign(j,e)) = i < j;
    smaller(ballot(l,i), ballot(m,j)) = i < j;

% updSingleList: Function to add a data item to a local list.
%
% parameters:
% m: the data item to be added
% i: the index of the local list in the the List of Lists
% x |> xs: the List of Lists, containg all local lists
%
% pre: value of Nat parameter is smaller than length of List(List(Data) param.
map updSingleList: Nat#Data#List(List(Data)) -> List(List(Data));
var m: Data;
    x: List(Data);
    xs: List(List(Data));
    i: Nat;
eqn updSingleList(i, m, x |> xs) =
      if(i>0,
         (x |> updSingleList(Int2Nat(i-1), m, xs)),
         insListData(m, x) |> xs);

% updBuf: Function to update all buffers after a broadcast (mimicking the
% reception of the broadcast message).
map updBuf: Data#List(List(Data)) -> List(List(Data));
var m: Data;
    x: List(Data);
    xs: List(List(Data));
eqn updBuf(m, []) = [];
    updBuf(m, x |> xs) = (x <| m) |> updBuf(m, xs);

% removeHead: Removes the the head of the list with the index as given by the
% first parameter. Used for example when a message has been processed by a given
% voter and can thus be removed from its input buffer.
% pre: value of Nat parameter is smaller than length of List(List(Data) param.
map removeHead: Nat#List(List(Data)) -> List(List(Data));
var x: List(Data);
    xs: List(List(Data));
    i: Nat;
```

```
eqn removeHead(i, x |> xs) = if(i>0,
                                x |> removeHead(Int2Nat(i-1), xs),
                                tail(x) |> xs);

% removeToe: Removes the toe (last element of a list) of the list with the
% index as given by the first parameter. Only used (for now) to remove a
% broadcasted message from a voters own buffer. Done in order to reduce state
% space.
% pre: value of Nat parameter is smaller than length of List(List(Data)) param.
map removeToe: Nat#List(List(Data)) -> List(List(Data));
var x: List(Data);
    xs: List(List(Data));
    i: Nat;
eqn removeToe(i, x |> xs) = if(i>0,
                              x |> removeToe(Int2Nat(i-1), xs),
                              rtail(x) |> xs);

% unblind: Set of equations indicating what Data constructs can be unblinded.
% If a certain construct can not be unblinded, the term reduces to the error
% value 'err'.
map unblind: Nat#Data -> Data;
var i,j,k: Nat;
    x: Data;
eqn unblind(i, sign(k, blind(j, x))) = if(i==j, sign(k, x), err);
    unblind(i, blind(j, x)) = if(i==j, x, err);
    !(isBlind(x) || isSign(x)) -> unblind(i,x) = err;
    !(isBlind(x)) -> unblind(i, sign(j, x)) = err;

% reachedDeadline: Function to check whether the first deadline has been
% reached. The parameter of this function is the list of all local buffers. When
% all these buffers are empty, the deadline is reached. Pruning is used by
% inspecting the head of a non-empty local list. When this head is either
% a ballot or a dishonestList, then the deadline must have been passed.
map reachedDeadline: List(List(Data)) -> Bool;
var x: List(Data);
    xs: List(List(Data));
eqn reachedDeadline([]) = true;
    reachedDeadline(x |> xs) =
      if(#x==0,
        reachedDeadline(xs),
        if(isBallot(head(x)),
           true,
           false));

% insListData: Function that inserts a Data item in a sorted list.
map insListData: Data#List(Data) -> List(Data);
var x,d: Data;
    xs: List(Data);
eqn insListData(d, []) = [d];
    insListData(d, x |> xs) = if(smaller(d, x),
                                 d |> (x |> xs),
                                 x |> insListData(d, xs));

map insListNat: Nat#List(Nat) -> List(Nat);
var x, n: Nat;
    xs: List(Nat);
eqn insListNat(n, []) = [n];
    insListNat(n, x |> xs) = if(n < x,
                                n |> (x |> xs),
                                x |> insListNat(n, xs));

% initList: Function to initialize Lists of n empty Lists of Data.
map initList: Nat -> List(List(Data));
var n: Nat;
eqn  initList(n) = if(n>1,
                      [] |> initList(Int2Nat(n-1)),
                      [[]]);
```

```
% addOrderToNonces; Function that replaces the generic 'plain' nonce : Nonce in
% each signed (vote, nonce) pair by a nonce which has an order. This is done in
% order to be able to distinguish between different ballots. If the plain nonces
% were kept, one can not properly distinguish between two ballots. The only dif-
% ference would then be the order that is in ballot(listc: List(Data), order:
% Nat). However, this order can easily be adapted since it is plain text in
% the tuple. Hence a voter can abuse this to add extra ballots.
%
% The meaning of ordNonce(i) is that it is the i^th nonce observed on the anon-
% ymous broadcast channel.
map addOrderToNonces: List(Data)#Nat -> List(Data);
var x: Data;
    xs: List(Data);
    order: Nat;
eqn addOrderToNonces([], order) = [];
    isSign(x) ->
        addOrderToNonces(x |> xs, order) =
              if(isVote(smsg(x)),
                 sign(signer(x), vote(cand(smsg(x)), ordNonce(order))) |>
                      addOrderToNonces(xs, order),
                 x |> addOrderToNonces(xs, order));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% PROCESSES %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%Voters process
% parameters:
%   N:         Number of voters.
%   inBuf:     The 'broadcast channel'; buffer where messages get exchanged on.
%   regList:   List of registered voters, i.e. voters who have broadcasted a
%                blinded vote. For now, this list is modelled as a global list.
%   sigList:   List of list of signatures.
%   votes:     List of votes.
%   tally:     List of list of ballots.
%   order:     Order in which the ballots are cast, used to sort them internally.
proc Voters(N: Nat, inBuf: List(List(Data)), regList: List(Nat), sigList:
  List(List(Data)), votes: List(Bool), tally: List(List(Data)), order: Nat,
  castList: List(Nat)) =
  sum i:Nat . (0 <= i && i < N) ->
    % If voter i has not yet cast its blinded vote, he can do this.
    !(i in regList) ->
      bcast(i, blind(i, vote(votes.i, nonce))) .
        Voters(N, removeToe(i, updBuf(blind(i, vote(votes.i,
               nonce)), inBuf)), insListNat(i, regList),
               updSingleList(i, sign(i, vote(votes.i, nonce)), sigList),
               votes, tally, order, castList)
    <>
    ( % If voter i has cast its blinded vote, he can not cast another blinded
      % vote. He can only process incoming messages now, until the deadline is
      % reached. After the deadline has been reached, he can still process
      % messages.
    (#(inBuf.i)>0) -> (
      isBlind(head(inBuf.i)) ->
          bcast(i, sign(i, head(inBuf.i))).
            Voters(N, removeToe(i, removeHead(i, updBuf(sign(i,
                  head(inBuf.i)), inBuf))), regList, sigList, votes,
                  tally, order, castList)
      + isSign(head(inBuf.i)) -> (
        (unblind(i, head(inBuf.i)) != err) ->
          unblinded(i, head(inBuf.i)) .
            Voters(N, removeHead(i, inBuf), regList, updSingleList(i,
                   unblind(i, head(inBuf.i)), sigList), votes, tally,
                   order, castList)
        <> cannotUnblind(i, head(inBuf.i)) .
             Voters(N, removeHead(i, inBuf), regList, sigList, votes, tally,
                   order, castList)
        )
```

```
        + isBallot(head(inBuf.i)) ->
          receivedBallot(i, head(inBuf.i)) .
            Voters(N, removeHead(i, inBuf), regList, sigList, votes,
                   updSingleList(i, head(inBuf.i), tally), order, castList)
        )
        % Once the deadline has been reached, voter i may cast his ballot.
      + (reachedDeadline(inBuf)) ->
        % Check whether voter has already cast a ballot or not to avoid that a
        % voter sends an infinite number of ballots.
         !(i in castList) ->
          abcast(ballot(sigList.i, order)) .
            Voters(N, removeToe(i, updBuf(ballot(sigList.i, order), inBuf)),
                   regList, sigList, votes, updSingleList(i, ballot(sigList.i,
                   order), tally), order+1, insListNat(i, castList))

    );

    % numVoters: Defines the number of voters in the protocol.
    map numVoters: Nat;
    eqn numVoters = 3;

    map votesVector: List(Bool);
    eqn votesVector = [false, true, false];

    init hide({cannotUnblind, unblinded, receivedBallot}, Voters(numVoters,
              initList(numVoters), [], initList(numVoters), votesVector,
              initList(numVoters), 0, []));
```

A.3 Dishonest model

```
  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% SORTS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %Definition of a nonce.
    % * nonce(n) is used locally in the construction of a blinded vote
    % * ordNonce(n) is used to differentiate between nonce on the anonymous
    %     broadcast channel
    sort Nonce = struct nonce(ord: Nat)?isPlainNonce |
                        ordNonce(order: Nat)?isOrdNonce;

    %General datatype.
    sort Data =
      struct sign(signer: Nat, smsg: Data)?isSign | % signed message
             blind(blinder: Nat, bmsg: Data)?isBlind | % blinded message
             vote(cand: Bool, nonce: Nonce)?isVote | % vote
             voter(id: Nat)?isVoter | % identity of a voter
             ballot(listc: List(Data), order: Nat)?isBallot | % ballot message
             blindmsg(blinderm: Nat, norder: Nat)?isBlindMsg | % blinded message
               % (externally observed)
             ballotIdx(index: Nat)?isBallotIdx | % index of a ballot
             dishonestList(dList: List(Data))?isDishonestList |
               % list of dishonest signers
             signed(signerf: Nat, signedf: Nat)?isSigned | % relation expressing
               % that voter signerf signed the first vote of voter signedf
             vtally(tallyc: List(Data))?isTally | % tally of a voter
             regInfo(voterid: Nat, tries: Nat, bindex: Nat)?isRegInfo |
               %registration information
             null?isNull | % null value
             err?isErr; % error value

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% ACTIONS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    act abcast: Data;              % anonymous broadcast
    act bcast: Nat#Data;           % authenticated broadcast
    act cannotUnblind: Nat#Data;   % action when incoming message cannot be unblinded
    act unblinded: Nat#Data;       % action when incoming message is unblinded
    act receivedBallot: Nat#Data; % action when a ballot is received
    act receivedDishonestList: Nat#Data; %action when a voter receives a dishonest
                                   % list from a signer
    act emptyDishonestList: Nat;  % action that is performed when a voter does not
```

```
                                  %  cast its dishonesty list because it is empty
   act noBlind;                   % action when dishonest voter decides not to send
                                  %  a blinded vote
40 act noBallot;                  % action when dishonest voter decides not to send
                                  %  a ballot
   act notSign: Nat#Data;         % action when dishonest signer decides not to sign
                                  %  a blinded vote
   act finalTally: Data;          % observer action, used to see what the final
45                                %  tally is

   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% FUNCTIONS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   % smaller: Function that defines an order on Data elements (only for the ones
   % that are used in sorted lists).
50 map smaller: Data#Data -> Bool;
   var d, e: Data;
       i, j, i2, j2: Nat;
       l, m: List(Data);
   eqn smaller(sign(i, d), sign(j, e)) = i < j;
55     smaller(ballot(l, i), ballot(m, j)) = i < j;
       smaller(ballotIdx(i), ballotIdx(j)) = i < j;
       smaller(voter(i), voter(j)) = i < j;
       smaller(signed(i, j), signed(i2, j2)) = ((i == i2) && (j < j2)) || (i < i2);

60 % updSingleList: Function to add a data item to a local sorted list, given a
   % list of local lists.
   %
   % parameters:
   % 1st parameter i: Nat; the index of the local list in the list of lists
65 % 2nd parameter m: Data; the data item to be added
   % 3rd parameter x |> xs: List(List(Data)); a list of lists, containing all
   %   local lists
   %
   % pre: value of i parameter is smaller than length of the list of lists
70 map updSingleList: Nat#Data#List(List(Data)) -> List(List(Data));
   var m: Data;
       x: List(Data);
       xs: List(List(Data));
       i: Nat;
75 eqn updSingleList(i, m, x |> xs) = if(i>0,
                                        (x |> updSingleList(Int2Nat(i-1), m, xs)),
                                        insListData(m, x) |> xs);

   % emptySingleList: Function to empty a local list, given a list of local lists.
80 %
   % parameters:
   % 1st parameter i: Nat; the index of the local list in the list of lists
   % 2nd parameter x |> xs: List(List(Data)); a list of lists, containing all local
   %   lists
85 %
   % pre: value of i parameter is smaller than length of the list of lists
   map emptySingleList: Nat#List(List(Data)) -> List(List(Data));
   var x: List(Data);
       xs: List(List(Data));
90     i: Nat;
   eqn emptySingleList(i, x |> xs) = if(i>0,
                                       (x |> emptySingleList(Int2Nat(i-1), xs)),
                                       [] |> xs);

95 % nullSingleList: Function that replaces a specific local list by a list
   % containing only the value null, given a list of local lists.
   %
   % parameters:
   % 1st parameter i: Nat; the index of the local list in the list of lists
100 % 2nd parameter x |> xs: List(List(Data)); a list of lists, containing all local
   %   lists
   %
   % pre: value of i parameter is smaller than length of the list of lists
```

```
map nullSingleList: Nat#List(List(Data)) -> List(List(Data));
var x: List(Data);
    xs: List(List(Data));
    i: Nat;
eqn nullSingleList(i, x |> xs) = if(i>0,
                                   x |> nullSingleList(Int2Nat(i-1), xs),
                                   [null] |> xs);

% updBuf: Function to update all buffers with a new message (mimicking the
% reception of the broadcast message). All buffers can be seen as a queue, the
% new message is thus placed at the end of the buffers.
%
% parameters;
% 1st parameter m: Data; the data item containing the message
% 2nd parameter x |> xs: List(List(Data)); a list of lists, containing all local
%   lists
map updBuf: Data#List(List(Data)) -> List(List(Data));
var m: Data;
    x: List(Data);
    xs: List(List(Data));
eqn updBuf(m, []) = [];
    updBuf(m, x |> xs) = (x <| m) |> updBuf(m, xs);

% removeHead: Removes the the head of the list with the index as given by the
% first parameter. Used for example when a message has been processed by a given
% voter and can thus be removed from its input buffer.
%
% parameters:
% 1st parameter i: Nat; index of the list of which the head has to be removed
% 2nd parameter x |> xs: List(List(Data)); a list of lists, containing all local
%   lists
%
% pre: value of Nat parameter is smaller than length of the list of lists
map removeHead: Nat#List(List(Data)) -> List(List(Data));
var x: List(Data);
    xs: List(List(Data));
    i: Nat;
eqn removeHead(i, x |> xs) = if(i>0,
                                x |> removeHead(Int2Nat(i-1), xs),
                                tail(x) |> xs);

% removeToe: Removes the toe (last element of a list) of the list with the
% index as given by the first parameter. Used to remove a broadcasted message
% from a voters own buffer. Done in order to reduce the state space.
%
% parameters:
% 1st parameter i: Nat; index of the list of which the head has to be removed
% 2nd parameter x |> xs: List(List(Data)); a list of lists, containing all local
%   lists
%
% pre: value of Nat parameter is smaller than length of the list of lists
map removeToe: Nat#List(List(Data)) -> List(List(Data));
var x: List(Data);
    xs: List(List(Data));
    i: Nat;
eqn removeToe(i, x |> xs) = if(i>0,
                               x |> removeToe(Int2Nat(i-1), xs),
                               rtail(x) |> xs);

% unblind: Set of equations indicating what Data constructs can be unblinded.
% If a certain construct can not be unblinded, the term reduces to the error
% value 'err'.
map unblind: Nat#Data -> Data;
var i,j,k: Nat;
    x: Data;
eqn unblind(i, sign(k, blind(j, x))) = if(i==j, sign(k, x), err);
    unblind(i, blind(j, x)) = if(i==j, x, err);
```

```
    !(isBlind(x) || isSign(x)) -> unblind(i, x) = err;
    !(isBlind(x)) -> unblind(i, sign(j, x)) = err;

% reachedDeadline: Function to check whether the first deadline has been
% reached. The parameter of this function is the list of all local buffers. When
% all these buffers are empty, the deadline is reached. Pruning is used by
% inspecting the head of a non-empty local list. When this head is either
% a ballot or a dishonestList, then the deadline must have been passed.
%
% parameters:
% 1st parameter x |> xs: List(List(Data)); list of local lists
map reachedDeadline: List(List(Data)) -> Bool;
var x: List(Data);
    xs: List(List(Data));
eqn reachedDeadline([]) = true;
    reachedDeadline(x |> xs) =
      if(#x==0,
        reachedDeadline(xs),
        if((isBallot(head(x)) || isDishonestList(head(x))),
           true,
           false));

% reachedDeadline2: Function to check whether the second deadline has been
% reached. The deadline is reached when all local buffers are empty and when
% all dishonesty lists have value [null]. This means that all dishonesty lists
% are processed, thus the deadline is reached. Pruning is again applied.
%
% parameters:
% 1st parameter x |> xs : List(List(Data)); list of all local buffers
% 2nd parameter y |> ys : List(List(Data)); list of all local dishonesty lists
map reachedDeadline2: List(List(Data))#List(List(Data)) -> Bool;
var x, y: List(Data);
    xs, ys: List(List(Data));
eqn reachedDeadline2([], []) = true;
    reachedDeadline2([], y |> ys) =
      if(y == [null],
         reachedDeadline2([], ys),
         false);
    reachedDeadline2(x |> xs, ys) =
      if(#x==0,
         reachedDeadline2(xs, ys),
         if(isBallot(head(x)),
            true,
            false));

% reachedDeadline3: Function to check whether the third deadline has been
% reached. The deadline is reached when all voters have cast (or for the
% dishonest voters: skipped) the ballots they have prepared and when the local
% buffers are empty.
%
% parameters:
% 1st parameter d: Nat; number of dishonest voters
% 2nd parameter limit: Nat; limit value
% 3rd parameter x |> xs: List(List(Data)); list of local cast lists, containing
%   identifiers for ballots that have been cast
% 4th parameter y |> ys: List(List(Data)); list of local buffers
map reachedDeadline3: Nat#Nat#List(List(Data))#List(List(Data)) -> Bool;
var d, limit: Nat;
    x, y: List(Data);
    xs, ys: List(List(Data));
eqn reachedDeadline3(d, limit, [], []) = true;
    reachedDeadline3(d, limit, [], y |> ys) =
      if(#y==0,
         reachedDeadline3(d, limit, [], ys),
         false);
    reachedDeadline3(d, limit, x |> xs, ys) =
      % i < d, hence first d > 0 indices need to be checked
```

```
        if(d > 0,
           if(#x == limit,
              % current list ok, proceed to next one
              reachedDeadline3(Int2Nat(d-1), limit, xs, ys),
              % not all ballots cast
              false),
           if(#x == 1,
              reachedDeadline3(d, limit, xs, ys),
              false));

   % insListData: Function that inserts a Data item in a sorted list.
   %
   % parameters:
   % 1st parameter d: Data; data item that needs to be inserted
   % 2nd parameter x |> xs: List(Data); list of data items in which the item needs
   %   to be inserted
   map insListData: Data#List(Data) -> List(Data);
   var x, d: Data;
       xs: List(Data);
   eqn insListData(d, []) = [d];
       insListData(d, x |> xs) = if(smaller(d, x),
                                    d |> (x |> xs),
                                    x |> insListData(d, xs));

   % insListListData: Function that that inserts the elements of a list of Data
   %  items in (another) sorted list of Data.
   %
   % parameters:
   % 1st parameter x |> xs: List(Data); list of data items to be inserted
   % 2nd parameter ys: List(Data); list in which the data items need to be inserted
   map insListListData: List(Data)#List(Data) -> List(Data);
   var x: Data;
       xs, ys: List(Data);
   eqn insListListData([], ys) = ys;
       insListListData(x |> xs, ys) = insListData(x, insListListData(xs, ys));

   % initList: Function to initialize lists containing a specific number of empty
   % lists.
   %
   % parameters:
   % 1st parameter n: Nat; the number of lists that need to be initialized.
   map initList: Nat -> List(List(Data));
   var n: Nat;
   eqn initList(n) = if(n>1,
                        [] |> initList(Int2Nat(n-1)),
                        [[]]);

   % count: Function that counts the number of occurrences of a item in a list.
   %
   % parameters:
   % 1st parameter d: Data; data item which is searched for
   % 2nd parameter x |> xs: List(Data); list of data items to be searched in
   map count: Data#List(Data) -> Nat;
   var d, x: Data;
       xs: List(Data);
   eqn count(d, []) = 0;
       count(d, x |> xs) = if(d == x,
                              1 + count(d, xs),
                              count(d, xs));

   % uniq: Function that removes duplicate entries from a list.
   %
   % parameters:
   % 1st parameter x |> xs: List(Data); list of data items
   map uniq: List(Data) -> List(Data);
   var x: Data;
       xs: List(Data);
```

```
eqn uniq([]) = [];
    uniq(x |> xs) = if(!(x in xs),
                       x |> uniq(xs),
                       uniq(xs));

% remove: Function that removes all occurrences of a given item from a list.
%
% parameters:
% 1st parameter x |> xs: List(Data); list of data items
map remove: Data#List(Data) -> List(Data);
var m, x: Data;
    xs: List(Data);
eqn remove(m, []) = [];
    remove(m, x |> xs) = if(m == x,
                            remove(m, xs),
                            x |> remove(m, xs));

% filterSigByNonce: Function that given a list of signatures and the order of a
% nonce returns all signatures which contain that order.
%
% parameters:
% 1st parameter n: Nat; order of a nonce
% 2nd parameter x |> xs: List(Data); list of signatures
map filterSigByNonce: Nat#List(Data) -> List(Data);
var x: Data;
    xs: List(Data);
    n: Nat;
eqn filterSigByNonce(n, []) = [];
    filterSigByNonce(n, x |> xs) = if(isSign(x),
                                      if(ord(nonce(smsg(x)))==n,
                                         x |> filterSigByNonce(n, xs),
                                         filterSigByNonce(n, xs)),
                                      filterSigByNonce(n, xs));

% updDCList: Function that updates a local list containing who signed the
% votes of which other voters.
% All voters have to sign the first blinded vote that is cast by another voter.
% This vote can be identified by the order of its nonce, this has to be 0. If
% the given signature contains a blinded message containing a nonce with order
% 0, then the signer is honest and a data item describing that that voter
% signed a vote from a specific other voter is added to the list.
%
% parameters:
% 1st parameter i: Nat; identifier for the local list
% 2nd parameter m: Data; the signature that needs to be processed
% 3rd parameter x |> xs: List(List(Data)); a list of local lists containing
% who signed the votes of which other voters.
map updDCList: Nat#Data#List(List(Data)) -> List(List(Data));
var m: Data;
    x: List(Data);
    xs: List(List(Data));
    i: Nat;
eqn updDCList(i, m, x |> xs) =
      if(i>0,
         (x |> updDCList(Int2Nat(i-1), m, xs)),
         if(ord(nonce(bmsg(smsg(m)))) == 0,
           insListData(signed(signer(m), blinder(smsg(m))), x) |> xs,
           x |> xs)
        );

% getListOfCorrectSigned: Helper function for the function
% identifyDishonestSigners. Constructs a list of voters for which a specific
% signer has signed a blinded vote.
%
% parameters:
% 1st parameter v: Nat; the identity of a signer
% 2nd parameter x |> xs: List(Data); a list of pairs that say which
```

```
%   signer signed the blinded votes of which voters.
map getListOfCorrectSigned: Nat#List(Data) -> List(Data);
var x: Data;
    xs: List(Data);
    v: Nat;
eqn getListOfCorrectSigned(v, []) = [];
    getListOfCorrectSigned(v, x |> xs) =
      if(signerf(x) == v,
         insListData(voter(signedf(x)), getListOfCorrectSigned(v, xs)),
         getListOfCorrectSigned(v, xs));

% identifyDishonestSigners: Function that compares per signer the list of
% registered voters with a list of signers and the senders of the messages
% they signed. When these lists are equal, the signer is honest, when these
% are not equal, the signer has skipped at least one message. Therefore he is
% put on the list of dishonest signers.
%
% parameters:
% 1st parameter i: Nat; iterator;
% 2nd parameter signersList: List(Data); list of signers
% 3rd parameter regList: List(Data); list of registered voters
% 4th parameter dcList: List(Data); a list of pairs that say which
%   signer signed the blinded votes of which voters.
map identifyDishonestSigners: Nat#List(Data)#List(Data)#List(Data)-> List(Data);
var i: Nat;
    signersList, regList, dcList: List(Data);
eqn i == #(signersList) ->
      identifyDishonestSigners(i, signersList, regList, dcList) = [];
    i < #(signersList) ->
      identifyDishonestSigners(i, signersList, regList, dcList) =
         if(getListOfCorrectSigned(id(signersList.i),
            dcList) == remove(signersList.i, regList),
              %everything ok
              identifyDishonestSigners(i+1, signersList, regList, dcList),
              %skipped at least one signature => dishonest
              insListData(signersList.i, identifyDishonestSigners(i+1,
                          signersList, regList, dcList))
         );

% constructSignersList: Function that takes a list of voters and removes the
% voters with an index larger than a given number.
%
% parameters:
% 1st parameter s: Nat; number of signers (used as iterator)
map constructSignersList: Nat -> List(Data);
var i: Nat;
eqn i > 0 -> constructSignersList(i) =
      constructSignersList(Int2Nat(i-1)) <| voter(Int2Nat(i-1));
    i == 0 -> constructSignersList(i) = [];

% updADC: Function that adds the elements of a given dishonesty list to a list
% containing the items of all received dishonesty lists.
%
% parameters:
% 1st parameter i: Nat; the index of the local list in the list of lists
% 2nd parameter m: Data; the dishonesty list to be added
% 3rd parameter x |> xs: List(List(Data)); a list of lists, containing all
%   local lists
%
% pre: value of i parameter is smaller than length of the list of lists
map updADC: Nat#Data#List(List(Data)) -> List(List(Data));
var i: Nat;
    m: Data;
    x: List(Data);
    xs: List(List(Data));
eqn updADC(i, m, x |> xs) =
      if(i>0,
```

```
          (x |> updADC(Int2Nat(i-1), m, xs)),
          insListListData(dList(m), x) |> xs);

  % kTimesDishonest: Function that, given a threshold value and a list of
  % possibly dishonest signers, determines which voters are really dishonest. That
  % is, which signers are accused at least k times of dishonesty.
  %
  % parameters:
  % 1st parameter i: Nat; iterator
  % 2nd parameter k: Nat; threshold value
  % 3rd parameter adcList: List(Data); all received dishonesty lists from a
  %   specific voter
  map kTimesDishonest: Nat#Nat#List(Data) -> List(Data);
  var i, k: Nat;
      adcList: List(Data);
  eqn i == #(uniq(adcList)) -> kTimesDishonest(i, k, adcList) = [];
      i <  #(uniq(adcList)) ->
        kTimesDishonest(i, k, adcList) =
          if(count((uniq(adcList)).i, adcList) >= k,
             insListData((uniq(adcList)).i, kTimesDishonest(i+1, k, adcList)),
             kTimesDishonest(i+1, k, adcList));

  % removeDishonestSignatures: Function that given a list of dishonest signers
  % and a list of signatures removes the signatures that are signed by any of the
  % dishonest signers.
  %
  % parameters:
  % 1st parameter dList: List(Data); list of dishonest signers
  % 2nd parameter s |> sigList: List(Data); a list of signatures, all
  %   corresponding to the same vote (filtered by nonce)
  map removeDishonestSignatures: List(Data)#List(Data) -> List(Data);
  var d, s: Data;
      dList, sigList: List(Data);
  eqn removeDishonestSignatures(dList, []) = [];
      removeDishonestSignatures(dList, s |> sigList) =
        if(voter(signer(s)) in dList,
           % then signature is from a dishonest signer and should be removed
           removeDishonestSignatures(dList, sigList),
           % else signature is from an honest signer and should be kept
           insListData(s, removeDishonestSignatures(dList, sigList)));

  % containsDishonestSignatures: Function that checks whether a list of signatures
  % contains signatures of dishonest signers.
  %
  % parameters:
  % 1st parameter x |> xs: List(Data); list of dishonest signers
  % 2nd parameter ys: List(Data); list of signatures
  map containsDishonestSignatures: List(Data)#List(Data) -> Bool;
  var y: Data;
      xs, ys: List(Data);
  eqn containsDishonestSignatures(xs, []) = false;
      containsDishonestSignatures(xs, y |> ys) =
        if(voter(signer(y)) in xs,
           true,
           containsDishonestSignatures(xs, ys));

  % extractTally: Function that extracts the tally given a list of ballots and a
  % list of dishonest signers.
  %
  % parameters:
  % 1st parameter k: Nat; threshold value
  % 2nd parameter x |> xs: List(Data): list of ballots
  % 3rd parameter alldcList: List(Data): list of dishonest signers
  map extractTally: Nat#List(Data)#List(Data) -> List(Data);
  var k: Nat;
      x: Data;
      xs, alldcList: List(Data);
```

```
eqn extractTally(k, [], alldcList) = [];
    extractTally(k, x |> xs, alldcList) =
    if(isBallot(x),
      if((#(uniq(listc(x))) >= k &&
         !(containsDishonestSignatures(kTimesDishonest(0, k, alldcList),
         listc(x)))),
           x |> extractTally(k, xs, alldcList),
           extractTally(k, xs, alldcList)),
      extractTally(k, xs, alldcList));

% filterDCList: Function that given a list of data items, constructs a
% single sorted list containing all elements of type voter. This function is
% used by the external observer to extract the dishonest voters from its input
% buffer.
%
% parameters:
% 1st parameter x |> xs: List(Data); List of data, contains dishonesty lists
map filterDCList: List(Data) -> List(Data);
var x: Data;
    xs: List(Data);
eqn filterDCList([]) = [];
    filterDCList(x |> xs) =
      if(isVoter(x),
         insListData(x, filterDCList(xs)),
         filterDCList(xs));

% addOneBindex: Function that given a list containing registration information
% for the voters, adds one to the index of the blinded votes (bindex) for a
% specific voter.
%
% parameters:
% 1st parameter i: Nat; the index of the regInfo in the list of data items
%   containing regInfo
% 2nd parameter: x |> xs: List(Data); list of data items containing regInfo
%
% pre: value of i parameter is smaller than length of the list of lists
map addOneBindex: Nat#List(Data)->List(Data);
var i: Nat;
    x: Data;
    xs: List(Data);
eqn addOneBindex(i, x |> xs) =
      if(i>0,
         x |> addOneBindex(Int2Nat(i-1), xs),
         regInfo(voterid(x), tries(x), bindex(x)+1) |> xs);

% addOneBindex: Function that given a list containing registration information
% for the voters, adds one to the tries for a specific voter.
%
% parameters:
% 1st parameter i: Nat; the index of the regInfo in the list of data items
%   containing regInfo
% 2nd parameter: x |> xs: List(Data); list of data items containing regInfo
%
% pre: value of i parameter is smaller than length of the list of lists
map addOneTries: Nat#List(Data)->List(Data);
var i: Nat;
    x: Data;
    xs: List(Data);
eqn addOneTries(i, x |> xs) =
      if(i>0,
         x |> addOneTries(Int2Nat(i-1), xs),
         regInfo(voterid(x), tries(x)+1, bindex(x)) |> xs);

% getListOfRegisteredVoters: Function that extracts a list of voters that have
% cast at least one blinded vote from a list of data items containing regInfo.
%
% parameters:
```

```
% 1st parameter: x |> xs: List(Data); list of data items containing regInfo
map getListOfRegisteredVoters: List(Data) -> List(Data);
var x: Data;
    xs: List(Data);
eqn getListOfRegisteredVoters([]) = [];
    getListOfRegisteredVoters(x |> xs) =
      if(bindex(x) > 0,
         voter(voterid(x)) |> getListOfRegisteredVoters(xs),
         getListOfRegisteredVoters(xs));

% initRegList: Function that initializes a list data items containing initial
% regInfo. That is, a unique voter identity, tries = 0 and bindex = 0.
%
% parameters
% 1st parameter n: Nat; number of voters
map initRegList: Nat -> List(Data);
var n: Nat;
eqn initRegList(n) =
      if(n > 0,
         initRegList(Int2Nat(n-1)) <| regInfo(Int2Nat(n-1), 0, 0),
         []);

% allRegistered: Function that checks whether all voters are done registering
% their blinded votes.
%
% parameters
% 1st parameter d: Nat; the number of dishonest voters
% 2nd parameter limit: Nat; the limit value of dishonest actions
% 3rd parameter x |> xs: List(Data);  list of data items containing regInfo
map allRegistered: Nat#Nat#List(Data)->Bool;
var d, limit: Nat;
    x: Data;
    xs: List(Data);
eqn allRegistered(d, limit, []) = true;
    allRegistered(d, limit, x |> xs) =
      if(voterid(x) < d,
         if(tries(x) == limit,
            allRegistered(d, limit, xs),
            false),
         if(tries(x) == 1,
            allRegistered(d, limit, xs),
            false));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% EXTRA FUNCTIONS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Some extra functions that are not directly used in the model, but are needed
% for some tools that take the model as input.

% addOrderToNonces: Function that replaces the generic 'plain' nonce : Nonce in
% each signed (vote, nonce) pair by a nonce which has an order. This is done in
% order to be able to distinguish between different ballots. If the plain nonces
% were kept, one can not properly distinguish between two ballots. The only dif-
% ference would then be the order that is in ballot(listc: List(Data), order:
% Nat). However, this order can easily be adapted since it is plain text in
% the tuple. Hence a voter could abuse this to add extra ballots.
%
% The meaning of ordNonce(i) is that it is the i^th nonce observed on the anon-
% ymous broadcast channel.
%
% parameters:
% 1st parameter x |> xs: List(Data); list of signatures
% 2nd parameter order: Nat; order that is used in the ordNonce
map addOrderToNonces: List(Data)#Nat -> List(Data);
var x: Data;
    xs: List(Data);
    order: Nat;
eqn addOrderToNonces([], order) = [];
    isSign(x) ->
```

```
640         addOrderToNonces(x |> xs, order) =
                if(isVote(smsg(x)),
                   sign(signer(x), vote(cand(smsg(x)), ordNonce(order))) |>
                        addOrderToNonces(xs, order),
                   x |> addOrderToNonces(xs, order));
645
    % addOrderToNoncesTally: Function that calls function addOrderToNonces for all
    % ballots in a tally.
    %
    % parameters:
650 % 1st parameter x |> xs: List(Data); list of ballots (tally)
    map addOrderToNoncesTally: List(Data)->List(Data);
    var x: Data;
        xs: List(Data);
    eqn addOrderToNoncesTally([]) = [];
655     addOrderToNoncesTally(x |> xs) =
          ballot(addOrderToNonces(listc(x), order(x)), order(x))
            |> addOrderToNoncesTally(xs);


660 % countItems: Replacement function for the length operator #. This replacement
    % function is needed for formula checking with PBES since #(list) does not work
    % to count the items of a list when using the tool pbes2bool (on revision
    % 6517-shared).
    map countItems: List(Data) -> Nat;
665 var x: Data;
        xs: List(Data);
    eqn countItems([]) = 0;
        countItems(x |> xs) = 1 + countItems(xs);

670 % countItems: Replacement function for the element test operator _in_. This
    % replacement function is needed for formula checking with PBES since x in xs
    % does not work to count the items of a list when using the tool pbes2bool
    % (on revision 6517-shared).
    map containsItem: Data#List(Data) -> Bool;
675 var d, x: Data;
        xs: List(Data);
    eqn containsItem(d, []) = false;
        containsItem(d, x |> xs) = if(d==x,
                                      true,
680                                   containsItem(d, xs));

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% PROCESSES %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %
    %Voters process
685 % parameters:
    % N:           Number of voters.
    % k:           Threshold value.
    % d:           Number of dishonest voter.
    % s:           Number of signers.
690 % inBuf:       The 'broadcast channel'; buffer where messages get exchanged on.
    % regList:     List of registered voters, i.e. voters who have broadcasted a
    %                blinded vote. For now, this list is modelled as a global list.
    % sigList:     List of lists (per voter) of signatures.
    % votes:       List of votes.
695 % tally:       List of lists (per voter) of ballots.
    % order:       Order in which the ballots are cast, used to sort them internally.
    % castList:    List of lists (per voter) of ballots. Contains for each voter
    %                the indices of the ballots he has cast.
    % limit:       The maximum number of times a voter can perform a dishonest action
700 %              (including the honest actions).
    % signedList:  List of lists (per voter) that contain the identities of the
    %                voters of which a voter has signed a message.
    % dcList:      List of lists (per voter) that contain a list of pairs that say
    %                which signer signed the blinded votes of which voters
705 % alldcList:   List of lists (per voter) that contain the received dishonest
    %                lists.
```

```
  % inBufObs:   The buffer of the broadcast channel for the observer.
  proc Voters(N: Nat, k: Nat, d: Nat, s: Nat, inBuf: List(List(Data)), regList:
    List(Data), sigList: List(List(Data)), votes: List(Bool), tally:
710 List(List(Data)), order: Nat, castList: List(List(Data)), limit: Nat,
    signedList: List(List(Data)), dcList: List(List(Data)), alldcList:
    List(List(Data)), inBufObs: List(Data)) =
    sum i:Nat . (0 <= i && i < N) -> (
      % If voter i has not yet cast its blinded vote, he can do this.
715     % check whether voter i has already cast a blinded vote
        ((i >= d) && (bindex(regList.i) < 1)) -> (
          (i < s) -> % voter is also a signer, hence he can sign his own vote
            bcast(i, blind(i, vote(votes.i, nonce(0)))) .
              Voters(N, k, d, s, removeToe(i, updBuf(blind(i, vote(votes.i,
720                 nonce(0))), inBuf)), addOneTries(i, addOneBindex(i,
                    regList)), updSingleList(i, sign(i, vote(votes.i,
                    nonce(0))), sigList), votes, tally, order, castList, limit,
                    signedList, dcList, alldcList, inBufObs)
          <> % voter is not a signer, and can hence not sign his own vote
725         bcast(i, blind(i, vote(votes.i, nonce(0)))) .
              Voters(N, k, d, s, removeToe(i, updBuf(blind(i, vote(votes.i,
                    nonce(0))), inBuf)), addOneTries(i, addOneBindex(i,
                    regList)), sigList, votes, tally, order, castList, limit,
                    signedList, dcList, alldcList, inBufObs)
730     )
        <>
        % dishonest voter can cast 'limit' blinded votes
        ((i < d) && (tries(regList.i) < limit)) -> ( %has not yet reached limit
          % internal decision: casting a blinded vote
735       bcast(i, blind(i, vote(votes.i, nonce(bindex(regList.i))))) .
            Voters(N, k, d, s, removeToe(i, updBuf(blind(i, vote(votes.i,
                  nonce(bindex(regList.i)))), inBuf)),
                  addOneTries(i, addOneBindex(i, regList)), updSingleList(i,
                  sign(i, vote(votes.i, nonce(bindex(regList.i)))),
740               sigList), votes, tally, order, castList, limit, signedList,
                  dcList, alldcList, inBufObs)
        + % or not casting a blinded vote
          noBlind . Voters(N, k, d, s, inBuf, addOneTries(i, regList),
                           sigList, votes, tally, order, castList, limit,
745                        signedList, dcList, alldcList, inBufObs)
        )
      <>
      ( % If voter i has processed all its blinded votes (an honest voter can only
        % cast a single blinded vote, a dishonest gets 'limit' blinded votes,
750     % which he can skip or cast), he can only process incoming messages or
        % send signatures (if he is a signer) until the deadline is reached.
      (#(inBuf.i)>0) -> (
        isBlind(head(inBuf.i)) ->
          (i >= d) -> ( %honest voters
755         % Honest signers check whether they have already signed a blinded vote
            % by the sender of the current blinded vote. If it is not the case,
            % then sign the blinded vote.
            (!(voter(blinder(head(inBuf.i))) in signedList.i) && (i < s)) ->
              bcast(i, sign(i, head(inBuf.i))) .
760             Voters(N, k, d, s, removeToe(i, removeHead(i, updBuf(sign(i,
                      head(inBuf.i)), inBuf))), regList, sigList, votes,
                      tally, order, castList, limit, updSingleList(i,
                      voter(blinder(head(inBuf.i))), signedList), dcList,
                      alldcList, inBufObs)
765       <> % If it is the case, do not sign the blinded vote. Also do not sign
             % the blinded vote if the voter is not a signer.
              notSign(i, head(inBuf.i)) .
                Voters(N, k, d, s, removeHead(i, inBuf), regList, sigList, votes,
                      tally, order, castList, limit, signedList, dcList,
770                   alldcList, inBufObs)
          )
          <> %i < d; dishonest voters
          (
```

```
        % Dishonest signers sign a blinded vote without checking whether it
        % was the first blinded vote that was sent by the sender.
        bcast(i, sign(i, head(inBuf.i))).
          Voters(N, k, d, s, removeToe(i, removeHead(i, updBuf(sign(i,
                 head(inBuf.i)), inBuf))), regList, sigList, votes, tally,
                 order, castList, limit, signedList, dcList, alldcList,
                 inBufObs)
        +
        % A dishonest voter can decide to behave dishonest by not signing a
        % blinded vote.
        notSign(i, head(inBuf.i)) .
          Voters(N, k, d, s, removeHead(i, inBuf), regList, sigList, votes,
                 tally, order, castList, limit, signedList, dcList, alldcList,
                 inBufObs)
      )
    + isSign(head(inBuf.i)) -> (
        (i < s) -> ( % Signers need to store all incoming signatures to
                     % construct the dishonesty lists.
          (unblind(i, head(inBuf.i)) != err) ->
            unblinded(i, head(inBuf.i)) .
              Voters(N, k, d, s, removeHead(i, inBuf), regList, updSingleList(
                     i, unblind(i, head(inBuf.i)), sigList), votes, tally,
                     order, castList, limit, signedList, updDCList(i,
                     head(inBuf.i), dcList), alldcList, inBufObs)
          <> cannotUnblind(i, head(inBuf.i)) .
               Voters(N, k, d, s, removeHead(i, inBuf), regList, sigList,
                      votes, tally, order, castList, limit, signedList,
                      updDCList(i, head(inBuf.i), dcList), alldcList,
                      inBufObs)
        )
        <>
        ( %Voters that are not signers do not need to store all signatures.
          (unblind(i, head(inBuf.i)) != err) ->
            unblinded(i, head(inBuf.i)) .
              Voters(N, k, d, s, removeHead(i, inBuf), regList, updSingleList(
                     i, unblind(i, head(inBuf.i)), sigList), votes, tally,
                     order, castList, limit, signedList, dcList, alldcList,
                     inBufObs)
          <> cannotUnblind(i, head(inBuf.i)) .
               Voters(N, k, d, s, removeHead(i, inBuf), regList, sigList,
                      votes, tally, order, castList, limit, signedList,
                      dcList, alldcList, inBufObs)
        )
      )
    + isBallot(head(inBuf.i)) ->
        ((#(uniq(listc(head(inBuf.i)))) < k ||
          containsDishonestSignatures(kTimesDishonest(0, k, alldcList.i),
          listc(head(inBuf.i)))) && !(head(inBuf.i) in tally.i) && i >= d) ->
        % If a ballot contains insufficient signatures or signatures of
        % dishonest signers or if the ballot is already received (these are
        % called invalid ballots), then do not store the ballot (only done
        % by honest voters).
        receivedBallot(i, head(inBuf.i)) .
          Voters(N, k, d, s, removeHead(i, inBuf), regList, sigList, votes,
                 tally, order, castList, limit, signedList, dcList, alldcList,
                 inBufObs)
        <> % Otherwise the ballot is stored. In case the voter is dishonest,
           % he also stores invalid ballots in order to try to get them
           % accepted.
        receivedBallot(i, head(inBuf.i)) .
          Voters(N, k, d, s, removeHead(i, inBuf), regList, sigList, votes,
                 updSingleList(i, head(inBuf.i), tally), order, castList,
                 limit, signedList, dcList, alldcList, inBufObs)
    + isDishonestList(head(inBuf.i)) ->
        receivedDishonestList(i, head(inBuf.i)) .
          Voters(N, k, d, s, removeHead(i, inBuf), regList, sigList, votes,
                 tally, order, castList, limit, signedList, dcList, updADC(i,
```

```
                head(inBuf.i), alldcList), inBufObs)
      )
      % Once the deadline has been reached, dishonesty lists can be constructed
      % and sent. Note that dishonesty lists are also added to the buffer of the
      % external observer (represented by parameter inBufObs) since these are
      % needed to construct the final tally.
    + (reachedDeadline(inBuf) && allRegistered(d, limit, regList)) -> (
      % construct and send dishonesty lists
      ((dcList.i != [null]) && (i < s)) -> (
        (identifyDishonestSigners(0, remove(voter(i),
        constructSignersList(s)), getListOfRegisteredVoters(regList),
        dcList.i) != []) -> (
          bcast(i, dishonestList(identifyDishonestSigners(0, remove(voter(i),
                constructSignersList(s)), getListOfRegisteredVoters(regList),
                dcList.i))) .
            Voters(N, k, d, s, removeToe(i, updBuf(dishonestList(
                  identifyDishonestSigners(0, remove(voter(i),
                  constructSignersList(s)), getListOfRegisteredVoters(
                  regList), dcList.i)), inBuf)), regList, sigList, votes,
                  tally, order, castList, limit, signedList,
                  nullSingleList(i, dcList), updADC(i, dishonestList(
                  identifyDishonestSigners(0, remove(voter(i),
                  constructSignersList(s)), getListOfRegisteredVoters(
                  regList), dcList.i)), alldcList), insListListData(
                  identifyDishonestSigners(0, remove(voter(i),
                  constructSignersList(s)), getListOfRegisteredVoters(
                  regList), dcList.i), inBufObs))
        )
        <>
        ( % empty dishonesty list -> no need to broadcast dishonesty list
          emptyDishonestList(i) .
            Voters(N, k, d, s, inBuf, regList, sigList, votes, tally, order,
                  castList, limit, signedList, nullSingleList(i, dcList),
                  alldcList, inBufObs)
        )
        % dishonest: mark all other signers as dishonest voters
        + (i < d) -> (
        bcast(i, dishonestList(remove(voter(i), constructSignersList(s)))) .
          Voters(N, k, d, s, removeToe(i, updBuf(dishonestList(
                remove(voter(i), constructSignersList(s))), inBuf)), regList,
                sigList, votes, tally, order, castList, limit, signedList,
                nullSingleList(i, dcList), updADC(i,
                dishonestList(remove(voter(i), constructSignersList(s))),
                alldcList), insListListData(remove(voter(i),
                constructSignersList(s)), inBufObs))
        )
      )
      <>
      (reachedDeadline2(inBuf, dcList)) -> (
      % Once the second deadline has been reached, the ballots can be cast.
      % Note that the ballots are also added to the buffer of the external
      % observer.
      % An honest voter can only cast a single ballot.
      ((i >= d) && (#(castList.i) < 1)) ->
            abcast(ballot(removeDishonestSignatures(kTimesDishonest(0, k,
                  alldcList.i), uniq(sigList.i)),
                  order)) .
              Voters(N, k, d, s, removeToe(i, updBuf(ballot(
                    removeDishonestSignatures(kTimesDishonest(0, k,
                    alldcList.i), uniq(sigList.i)), order), inBuf)),
                    regList, sigList, votes, updSingleList(i,
                    ballot(removeDishonestSignatures(kTimesDishonest(0, k,
                    alldcList.i), uniq(sigList.i)), order), tally),
                    order+1, updSingleList(i, ballotIdx(0), castList),
                    limit, signedList, dcList, alldcList, inBufObs <|
                    ballot(removeDishonestSignatures(kTimesDishonest(0, k,
                    alldcList.i), uniq(sigList.i)), order))
```

```
          <> (i < d) -> (
          % A dishonest voter has the choice to broadcast 'limit' ballots.
            sum j: Nat . (0 <= j && j < limit) ->
              !(ballotIdx(j) in castList.i) ->
                ( % Sending a ballot.
                  (removeDishonestSignatures(kTimesDishonest(0, k, alldcList.i),
                   uniq(filterSigByNonce(j, sigList.i))) != []) -> (
                    abcast(ballot(removeDishonestSignatures(kTimesDishonest(0, k,
                           alldcList.i), uniq(filterSigByNonce(j, sigList.i))),
                           order)) .
                      Voters(N, k, d, s, removeToe(i, updBuf(ballot(
                             removeDishonestSignatures(kTimesDishonest(0, k,
                             alldcList.i), uniq(filterSigByNonce(j, sigList.i))),
                             order), inBuf)), regList, sigList, votes,
                             updSingleList(i, ballot(removeDishonestSignatures(
                             kTimesDishonest(0, k, alldcList.i), uniq(
                             filterSigByNonce(j, sigList.i))), order), tally),
                             order+1, updSingleList(i, ballotIdx(j), castList),
                             limit, signedList, dcList, alldcList, inBufObs <|
                             ballot(removeDishonestSignatures(kTimesDishonest(0,
                             k, alldcList.i), uniq(filterSigByNonce(j,
                             sigList.i))), order))
                  )
                  + % Sending no ballot.
                    noBallot .
                      Voters(N, k, d, s, inBuf, regList, sigList, votes, tally,
                             order, updSingleList(i, ballotIdx(j), castList),
                             limit, signedList, dcList, alldcList, inBufObs)
                )
            )
          )
        )
      )
      % External objective observer; extracts tally from message observed from
      % authenticated broadcast channel (dishonesty lists) and anonymous broadcast
      % channel (ballots). Makes this tally observable using function finalTally.
    + reachedDeadline3(d, limit, castList, inBuf) -> (
        finalTally(vtally(extractTally(k, inBufObs, filterDCList(inBufObs)))) .
          delta
      )
    ;

    % numVoters: Defines the number of voters in the protocol.
    map numVoters: Nat;
    eqn numVoters = 3;

    % threshold: Defines the threshold value.
    map threshold: Nat;
    eqn threshold = 2;

    % numDishonestVoters: Defines the number of dishonest voters.
    % Maximum: threshold-1.
    map numDishonestVoters: Nat;
    % Default: maximum number of dishonest voters.
    eqn numDishonestVoters = Int2Nat(threshold-1);

    % numSigners: Defines the number of signers.
    % Minimum: 2*threshold-1.
    map numSigners: Nat;
    % Default: minimum number of signers.
    eqn numSigners = Int2Nat(2*threshold-1);

    % votesVector: Defines the votes of the voters, votesVector.i is voter i's vote.
    map votesVector: List(Bool);
    eqn votesVector = [false, true, false];
```

```
% limit: The maximum number of times a voter can perform a dishonest action
% (including the honest actions).
map limit: Nat;
eqn limit = 2;

init hide({cannotUnblind, unblinded, receivedBallot, receivedDishonestList,
           emptyDishonestList, notSign, noBlind, noBallot},
             Voters(
             numVoters,                  %parameter N
             threshold,                  %parameter k
             numDishonestVoters,         %parameter d
             numSigners,                 %parameter s
             initList(numVoters),        %parameter inBuf
             initRegList(numVoters),     %parameter regList
             initList(numVoters),        %parameter sigList
             votesVector,                %parameter votes
             initList(numVoters),        %parameter tally
             0,                          %parameter order
             initList(numVoters),        %parameter castList
             limit,                      %parameter limit
             initList(numVoters),        %parameter signedList
             initList(numSigners),       %parameter dcList
             initList(numVoters),        %parameter alldcList
             [])                         %parameter inBufObs
         );
```

A.3.1 Dishonest model rename file

```
var i: Nat;
    d: Data;
    l: List(Data);
rename
% bcast(i,d) gets rewritten to bcast(i, blindmsg(j)) iff d is a blinded message
% and where j is the index of the voter who blinded the message
isBlind(d) ->
  bcast(i, d) => bcast(i, blindmsg(blinder(d), ord(nonce(bmsg(d)))));
% bcast(i,d) gets rewritten to bcast(i, sign(j, blindmsg(k))) iff d is a signed
% message containing a blinded message. Identifier j represents the index of the
% voter who signed the message (which is needed to sign the rewritten message)
% and index k represents the index of the voter who blinded the message.
isSign(d) && isBlind(smsg(d)) -> bcast(i, d) =>
    bcast(i, sign(signer(d),
      blindmsg(blinder(smsg(d)), ord(nonce(bmsg(smsg(d)))))));

% abcast(d) gets rewritten to abcast(d') when d is a ballot. Data item d' con-
% tains the signature list and the order-number of ballot d, except that in the
% list of signatures every nonce is converted to an ordered nonce.
isBallot(d) ->
  abcast(d) => abcast(ballot(addOrderToNonces(listc(d), order(d)), order(d)));

% finalTally(d) gets rewritten to finalTally(d') when d is a tally. Data item d'
% is a tally containg a list of ballots where an order is added on the nonces
% of the ballot.
isTally(d) ->
  finalTally(d) => finalTally(vtally(addOrderToNoncesTally(tallyc(d))));
```

A.4 Model using strong synchronicity

Only the process *Voters* and its initialization is included. All sorts, actions and functions are as in the dishonest model, which can be found in Appendix A.3.

```
proc Voters(N: Nat, k: Nat, d: Nat, s: Nat, inBuf: List(List(Data)), regList:
  List(Data), sigList: List(List(Data)), votes: List(Bool), tally:
  List(List(Data)), order: Nat, castList: List(List(Data)), limit: Nat,
  signedList: List(List(Data)), dcList: List(List(Data)), alldcList:
  List(List(Data)), inBufObs: List(Data), semaphore: Nat) =
  sum i:Nat . (0 <= i && i < N) -> (
```

```
    % Check whether blinded vote may be cast (i.e. whether semaphore == 0).
    (semaphore == 0) -> ( % Semaphore is set to N-1 when a blinded vote is sent.
      ((i >= d) && (bindex(regList.i) < 1)) -> (
        (i < s) ->
10        bcast(i, blind(i, vote(votes.i, nonce(0)))) .
            Voters(N, k, d, s, removeToe(i, updBuf(blind(i, vote(votes.i,
                  nonce(0))), inBuf)), addOneTries(i, addOneBindex(i,
                  regList)), updSingleList(i, sign(i, vote(votes.i,
                  nonce(0))), sigList), votes, tally, order, castList, limit,
15                signedList, dcList, alldcList, inBufObs, Int2Nat(N-1))
        <>
          bcast(i, blind(i, vote(votes.i, nonce(0)))) .
            Voters(N, k, d, s, removeToe(i, updBuf(blind(i, vote(votes.i,
                  nonce(0))), inBuf)), addOneTries(i, addOneBindex(i,
20                regList)), sigList, votes, tally, order, castList, limit,
                  signedList, dcList, alldcList, inBufObs, Int2Nat(N-1))
      )
      <>
      ((i < d) && (tries(regList.i) < limit)) -> (
25      bcast(i, blind(i, vote(votes.i, nonce(bindex(regList.i))))) .
          Voters(N, k, d, s, removeToe(i, updBuf(blind(i, vote(votes.i,
                nonce(bindex(regList.i)))), inBuf)),
                addOneTries(i, addOneBindex(i, regList)), updSingleList(i,
                sign(i, vote(votes.i, nonce(bindex(regList.i)))),
30              sigList), votes, tally, order, castList, limit, signedList,
                dcList, alldcList, inBufObs, Int2Nat(N-1))
      + % No blinded vote is sent, thus do not change semaphore.
        noBlind . Voters(N, k, d, s, inBuf, addOneTries(i, regList),
                         sigList, votes, tally, order, castList, limit,
35                       signedList, dcList, alldcList, inBufObs, semaphore)
      )
    )
    +
    (
40  (#(inBuf.i)>0) -> (
      isBlind(head(inBuf.i)) ->
        (i >= d) -> ( % Lower sempahore by one if blinded vote is processed.
          (!(voter(blinder(head(inBuf.i))) in signedList.i) && (i < s)) ->
            bcast(i, sign(i, head(inBuf.i))) .
45            Voters(N, k, d, s, removeToe(i, removeHead(i, updBuf(sign(i,
                    head(inBuf.i)), inBuf))), regList, sigList, votes,
                    tally, order, castList, limit, updSingleList(i,
                    voter(blinder(head(inBuf.i))), signedList), dcList,
                    alldcList, inBufObs, Int2Nat(semaphore-1))
50        <>
              notSign(i, head(inBuf.i)) .
                Voters(N, k, d, s, removeHead(i, inBuf), regList, sigList, votes,
                       tally, order, castList, limit, signedList, dcList,
                       alldcList, inBufObs, Int2Nat(semaphore-1))
55      )
        <>
        (
          bcast(i, sign(i, head(inBuf.i))).
            Voters(N, k, d, s, removeToe(i, removeHead(i, updBuf(sign(i,
60                  head(inBuf.i)), inBuf))), regList, sigList, votes, tally,
                    order, castList, limit, signedList, dcList, alldcList,
                    inBufObs, Int2Nat(semaphore-1))
          +
          notSign(i, head(inBuf.i)) .
65          Voters(N, k, d, s, removeHead(i, inBuf), regList, sigList, votes,
                    tally, order, castList, limit, signedList, dcList, alldcList,
                    inBufObs, Int2Nat(semaphore-1))
        )
      + isSign(head(inBuf.i)) -> (
70        (i < s) -> (
            (unblind(i, head(inBuf.i)) != err) ->
              unblinded(i, head(inBuf.i)) .
```

```
                Voters(N, k, d, s, removeHead(i, inBuf), regList, updSingleList(
                       i, unblind(i, head(inBuf.i)), sigList), votes, tally,
                       order, castList, limit, signedList, updDCList(i,
                       head(inBuf.i), dcList), alldcList, inBufObs, semaphore)
          <> cannotUnblind(i, head(inBuf.i)) .
               Voters(N, k, d, s, removeHead(i, inBuf), regList, sigList,
                      votes, tally, order, castList, limit, signedList,
                      updDCList(i, head(inBuf.i), dcList), alldcList,
                      inBufObs, semaphore)
        )
        <>
        (
          (unblind(i, head(inBuf.i)) != err) ->
            unblinded(i, head(inBuf.i)) .
              Voters(N, k, d, s, removeHead(i, inBuf), regList, updSingleList(
                     i, unblind(i, head(inBuf.i)), sigList), votes, tally,
                     order, castList, limit, signedList, dcList, alldcList,
                     inBufObs, semaphore)
          <> cannotUnblind(i, head(inBuf.i)) .
               Voters(N, k, d, s, removeHead(i, inBuf), regList, sigList,
                      votes, tally, order, castList, limit, signedList,
                      dcList, alldcList, inBufObs, semaphore)
        )
      )
    + isBallot(head(inBuf.i)) ->
        ((#(uniq(listc(head(inBuf.i)))) < k ||
          containsDishonestSignatures(kTimesDishonest(0, k, alldcList.i),
          listc(head(inBuf.i)))) && !(head(inBuf.i) in tally.i) && i >= d) ->
        receivedBallot(i, head(inBuf.i)) .
          Voters(N, k, d, s, removeHead(i, inBuf), regList, sigList, votes,
                 tally, order, castList, limit, signedList, dcList, alldcList,
                 inBufObs, semaphore)
        <>
        receivedBallot(i, head(inBuf.i)) .
          Voters(N, k, d, s, removeHead(i, inBuf), regList, sigList, votes,
                 updSingleList(i, head(inBuf.i), tally), order, castList,
                 limit, signedList, dcList, alldcList, inBufObs, semaphore)
    + isDishonestList(head(inBuf.i)) ->
        receivedDishonestList(i, head(inBuf.i)) .
          Voters(N, k, d, s, removeHead(i, inBuf), regList, sigList, votes,
                 tally, order, castList, limit, signedList, dcList, updADC(i,
                 head(inBuf.i), alldcList), inBufObs, semaphore)
    )
  + (reachedDeadline(inBuf) && allRegistered(d, limit, regList)) -> (
      ((dcList.i != [null]) && (i < s)) -> (
        (identifyDishonestSigners(0, remove(voter(i),
        constructSignersList(s)), getListOfRegisteredVoters(regList),
        dcList.i) != []) -> (
          bcast(i, dishonestList(identifyDishonestSigners(0, remove(voter(i),
                constructSignersList(s)), getListOfRegisteredVoters(regList),
                dcList.i))) .
            Voters(N, k, d, s, removeToe(i, updBuf(dishonestList(
                   identifyDishonestSigners(0, remove(voter(i),
                   constructSignersList(s)), getListOfRegisteredVoters(
                   regList), dcList.i)), inBuf)), regList, sigList, votes,
                   tally, order, castList, limit, signedList,
                   nullSingleList(i, dcList), updADC(i, dishonestList(
                   identifyDishonestSigners(0, remove(voter(i),
                   constructSignersList(s)), getListOfRegisteredVoters(
                   regList), dcList.i)), alldcList), insListListData(
                   identifyDishonestSigners(0, remove(voter(i),
                   constructSignersList(s)), getListOfRegisteredVoters(
                   regList), dcList.i), inBufObs), semaphore)
        )
        <>
        (
          emptyDishonestList(i) .
```

```
140                 Voters(N, k, d, s, inBuf, regList, sigList, votes, tally, order,
                           castList, limit, signedList, nullSingleList(i, dcList),
                           alldcList, inBufObs, semaphore)
              )
              + (i < d) -> (
145           bcast(i, dishonestList(remove(voter(i), constructSignersList(s)))) .
                Voters(N, k, d, s, removeToe(i, updBuf(dishonestList(
                       remove(voter(i), constructSignersList(s))), inBuf)), regList,
                       sigList, votes, tally, order, castList, limit, signedList,
                       nullSingleList(i, dcList), updADC(i,
150                    dishonestList(remove(voter(i), constructSignersList(s))),
                       alldcList), insListListData(remove(voter(i),
                       constructSignersList(s)), inBufObs), semaphore)
              )
            )
155         <>
            (reachedDeadline2(inBuf, dcList)) -> (
            ((i >= d) && (#(castList.i) < 1)) ->
                    abcast(ballot(removeDishonestSignatures(kTimesDishonest(0, k,
                           alldcList.i), uniq(sigList.i)),
160                        order)) .
                      Voters(N, k, d, s, removeToe(i, updBuf(ballot(
                             removeDishonestSignatures(kTimesDishonest(0, k,
                             alldcList.i), uniq(sigList.i)), order), inBuf)),
                             regList, sigList, votes, updSingleList(i,
165                          ballot(removeDishonestSignatures(kTimesDishonest(0, k,
                             alldcList.i), uniq(sigList.i)), order), tally),
                             order+1, updSingleList(i, ballotIdx(0), castList),
                             limit, signedList, dcList, alldcList, inBufObs <|
                             ballot(removeDishonestSignatures(kTimesDishonest(0, k,
170                          alldcList.i), uniq(sigList.i)), order), semaphore)
            <> (i < d) -> (
              sum j: Nat . (0 <= j && j < limit) ->
                !(ballotIdx(j) in castList.i) ->
                  (
175                 (removeDishonestSignatures(kTimesDishonest(0, k, alldcList.i),
                     uniq(filterSigByNonce(j, sigList.i))) != []) -> (
                      abcast(ballot(removeDishonestSignatures(kTimesDishonest(0, k,
                             alldcList.i), uniq(filterSigByNonce(j, sigList.i))),
                             order)) .
180                     Voters(N, k, d, s, removeToe(i, updBuf(ballot(
                               removeDishonestSignatures(kTimesDishonest(0, k,
                               alldcList.i), uniq(filterSigByNonce(j, sigList.i))),
                               order), inBuf)), regList, sigList, votes,
                               updSingleList(i, ballot(removeDishonestSignatures(
185                            kTimesDishonest(0, k, alldcList.i), uniq(
                               filterSigByNonce(j, sigList.i))), order), tally),
                               order+1, updSingleList(i, ballotIdx(j), castList),
                               limit, signedList, dcList, alldcList, inBufObs <|
                               ballot(removeDishonestSignatures(kTimesDishonest(0,
190                            k, alldcList.i), uniq(filterSigByNonce(j,
                               sigList.i))), order), semaphore)
                    )
                    +
                      noBallot .
195                     Voters(N, k, d, s, inBuf, regList, sigList, votes, tally,
                               order, updSingleList(i, ballotIdx(j), castList),
                               limit, signedList, dcList, alldcList, inBufObs,
                               semaphore)
                  )
200           )
            )
          )
        )
205     + reachedDeadline3(d, limit, castList, inBuf) -> (
          finalTally(vtally(extractTally(k, inBufObs, filterDCList(inBufObs)))) .
```

```
        delta
    )
;

init hide({cannotUnblind, unblinded, receivedBallot, receivedDishonestList,
           emptyDishonestList, notSign, noBlind, noBallot},
             Voters(
             numVoters,                  %parameter N
             threshold,                  %parameter k
             numDishonestVoters,         %parameter d
             numSigners,                 %parameter s
             initList(numVoters),        %parameter inBuf
             initRegList(numVoters),     %parameter regList
             initList(numVoters),        %parameter sigList
             votesVector,                %parameter votes
             initList(numVoters),        %parameter tally
             0,                          %parameter order
             initList(numVoters),        %parameter castList
             limit,                      %parameter limit
             initList(numVoters),        %parameter signedList
             initList(numSigners),       %parameter dcList
             initList(numVoters),        %parameter alldcList
             [],                         %parameter inBufObs
             0)                          %parameter semaphore
         );
```

A.5 Unicity model

In this section we describe the full list of changes to the dishonest model (as can be found in Appendix A.3) and the rename file for the dishonest model (as can be found in Appendix A.3.1) in order to construct a model for which we can check the unicity property.

First we start by adding the parameter *bsender* to the ballot constructor. In the dishonest model we have a constructor that creates a ballot of type Data as follows.

```
blindmsg(blinderm: Nat, norder: Nat)?isBlindMsg
```

In the model for unicity we add the parameter *bsender* of type Data as follows.

```
blindmsg(blinderm: Nat, norder: Nat)?isBlindMsg
```

This has as result that we need to add the parameter to every occurrence of the ballot constructor. The first occurrence of in the *Voters* process is when a ballot is broadcast by an honest voter. In the dishonest model this is modelled as follows.

```
0       abcast(ballot(removeDishonestSignatures(kTimesDishonest(0, k,
               alldcList.i), uniq(sigList.i)), order)) .
          Voters(N, k, d, s, removeToe(i, updBuf(ballot(
                 removeDishonestSignatures(kTimesDishonest(0, k,
                 alldcList.i), uniq(sigList.i)), order), inBuf)),
5                regList, sigList, votes, updSingleList(i,
                 ballot(removeDishonestSignatures(kTimesDishonest(0, k,
                 alldcList.i), uniq(sigList.i)), order), tally),
                 order+1, updSingleList(i, ballotIdx(0), castList),
                 limit, signedList, dcList, alldcList, inBufObs <|
10               ballot(removeDishonestSignatures(kTimesDishonest(0, k,
                 alldcList.i), uniq(sigList.i)), order))
```

In the model for unicity we give the parameter *bsender* the value *voter(i)*, this is done as follows.

```
0       abcast(ballot(voter(i),
               removeDishonestSignatures(kTimesDishonest(0, k,
               alldcList.i), uniq(sigList.i)), order)) .
          Voters(N, k, d, s, removeToe(i, updBuf(ballot(voter(i),
                 removeDishonestSignatures(kTimesDishonest(0, k,
```

```
                alldcList.i), uniq(sigList.i)), order), inBuf)),
                regList, sigList, votes, updSingleList(i,
                ballot(voter(i), removeDishonestSignatures(
                kTimesDishonest(0, k, alldcList.i), uniq(sigList.i)),
                order), tally), order+1, updSingleList(i,
                ballotIdx(0), castList), limit, signedList, dcList,
                alldcList, inBufObs <| ballot(voter(i),
                removeDishonestSignatures(kTimesDishonest(0, k,
                alldcList.i), uniq(sigList.i)), order))
```

The second and last occurrence is the case where a dishonest voter casts its ballot. In the dishonest model this is modelled as follows.

```
        abcast(ballot(removeDishonestSignatures(kTimesDishonest(0, k,
               alldcList.i), uniq(filterSigByNonce(j, sigList.i))),
               order)) .
          Voters(N, k, d, s, removeToe(i, updBuf(ballot(
                 removeDishonestSignatures(kTimesDishonest(0, k,
                 alldcList.i), uniq(filterSigByNonce(j, sigList.i))),
                 order), inBuf)), regList, sigList, votes,
                 updSingleList(i, ballot(removeDishonestSignatures(
                 kTimesDishonest(0, k, alldcList.i), uniq(
                 filterSigByNonce(j, sigList.i))), order), tally),
                 order+1, updSingleList(i, ballotIdx(j), castList),
                 limit, signedList, dcList, alldcList, inBufObs <|
                 ballot(removeDishonestSignatures(kTimesDishonest(0,
                 k, alldcList.i), uniq(filterSigByNonce(j,
                 sigList.i))), order))
```

In the model for unicity we again add tern *voter(i)*.

```
        abcast(ballot(voter(i),
               removeDishonestSignatures(kTimesDishonest(0, k,
               alldcList.i), uniq(filterSigByNonce(j, sigList.i))),
               order)) .
          Voters(N, k, d, s, removeToe(i, updBuf(ballot(voter(i),
                 removeDishonestSignatures(kTimesDishonest(0, k,
                 alldcList.i), uniq(filterSigByNonce(j, sigList.i))),
                 order), inBuf)), regList, sigList, votes,
                 updSingleList(i, ballot(voter(i),
                 removeDishonestSignatures(kTimesDishonest(0, k,
                 alldcList.i), uniq(filterSigByNonce(j, sigList.i))),
                 order), tally), order+1, updSingleList(i,
                 ballotIdx(j), castList), limit, signedList, dcList,
                 alldcList, inBufObs <| ballot(voter(i),
                 removeDishonestSignatures(kTimesDishonest(0,
                 k, alldcList.i), uniq(filterSigByNonce(j,
                 sigList.i))), order))
```

Furthermore, we need to update some functions involving ballots. The first function is the function *smaller* that defines an order on ballots. In the dishonest model we have the following equation expressing that.

```
smaller(ballot(l, i), ballot(m, j))= i < j;
```

In the model for unicity we replace this equation by the following, where d, e are of type Data.

```
smaller(ballot(d, l, i), ballot(e, m, j))= i < j;
```

Then we also need to update function *addOrderToNoncesTally*. In the dishonest model we have the following equation.

```
    addOrderToNoncesTally(x |> xs) =
     ballot(addOrderToNonces(listc(x), order(x)), order(x))
       |> addOrderToNoncesTally(xs);
```

This equation is replaced in the model for unicity by the following equation.

```
    addOrderToNoncesTally(x |> xs) =
      ballot(bsender(x), addOrderToNonces(listc(x), order(x)), order(x))
        |> addOrderToNoncesTally(xs);
```

Finally, we need to update the rename file. In the rename file we only need to update a single rename rule. This is the following one.

```
isBallot(d)-> abcast(d)=> abcast(ballot(addOrderToNonces(listc(d), order(d)), order(d)));
```

This is replaced by the following rename rule.

```
isBallot(d)-> abcast(d)=> abcast(ballot(bsender(d), addOrderToNonces(listc(d), order(d)), order(
d)));
```

Finally, we need to add function *countBallotsInTally*, which is needed in one of the modal μ-formulae. This function is as follows.

```
map countBallotsInTally: Data#List(Data)->Nat;
var d, x: Data;
    xs: List(Data);
eqn countBallotsInTally(d, []) = 0;
    countBallotsInTally(d, x |> xs) = if(d == bsender(x),
                                        1 + countBallotsInTally(d, xs),
                                        countBallotsInTally(d, xs));
```

When all these changes are applied, we have a model on which we can verify the unicity property.

A.6 Modal formulae

In this section all formulae of Section 3.6.2 are given in the syntax used by *lps2pbes*.

File name:	`alwaysFinalTally.mcf`
In words:	For every path, eventually the action `finalTally` occurs.
Formula:	`[(!(exists d: Data . finalTally(d)))*]<true* . exists d: Data . finalTally(d)>true`

File name:	`atLeastOneSignature.mcf`
In words:	Every blinded vote is at least signed once by one of the signers.
Formula:	`forall n: Nat . val(n < numVoters) => [true* . bcast(n, blindmsg(n, 0))]<true* . exists m : Nat . bcast(m, sign(m, blindmsg(n, 0)))>true`

File name:	`atLeastKSignatures.mcf`
In words:	Every blinded vote is signed at least k times.
Formula:	`forall n: Nat . val(n < numVoters) => ([true* . bcast(n, blindmsg(n, 0))](nu X (cnt: Nat = 0) . [!(exists m: Nat . bcast(m, sign(m, blindmsg(n, 0))))] X(cnt) && [exists m: Nat . bcast(m, sign(m, blindmsg(n, 0)))] X(cnt+1) && (forall d: Data . [finalTally(d)](val((n < numSigners => (cnt+1 >= threshold)) && (n >= numSigners => (cnt >= threshold))))))`

File name:	`noSignerSignsTwice.mcf`
In words:	A signer does not sign the same blinded message twice.
Formula:	`forall n: Nat . val(n < numSigners) => forall blinder: Nat . val(blinder < numVoters) => forall order: Nat . val(order < limit) => [true* . bcast(n, sign(n, blindmsg(blinder, order))) . true* . bcast(n, sign(n, blindmsg(blinder, order)))]false`

File name:	`soundFinalTally.mcf`
In words:	The number of ballots in the final tally is greater than or equal to the number of blinded votes that are cast by honest voters.
Formula:	`nu X (cnt: Nat = 0) . ([!(exists n: Nat . val(n >= numDishonestVoters) && bcast(n, blindmsg(n, 0)))] X(cnt) && [exists n: Nat . val(n >= numDishonestVoters) && bcast(n, blindmsg(n, 0))] X(cnt+1) && forall d: Data . ([finalTally(d)](val(cnt <= countItems(tallyc(d))))))`

File name:	`putOnDishonestList.mcf`
In words:	A dishonest signer is put on the list of dishonest signers of every honest signer when it has not signed all first blinded messages of all voters.
Formula:	`forall n: Nat . val(n < numDishonestVoters) => nu X (scnt: Nat = 0, vcnt: Nat = 0) . ([!(exists m: Nat . bcast(n, sign(n, blindmsg(m, 0))) || exists m: Nat . bcast(m, blindmsg(m, 0)))] X(scnt, vcnt) && [exists m: Nat . bcast(n, sign(n, blindmsg(m, 0)))] X(scnt+1, vcnt) && [exists m: Nat . bcast(m, blindmsg(m, 0))] X(scnt, vcnt+1) && (forall m: Nat . val(numDishonestVoters <= m && m < numSigners) => (forall d: Data . [val(isDishonestList(d)) && bcast(m, d)](val(scnt < vcnt-1 => containsItem(voter(n), dList(d)) && containsItem(voter(n), dList(d)) => scnt < vcnt-1)))))`

File name:	`unicity.mcf`
In words:	A voter can only vote once.
Formula:	`forall n: Nat . val(n < numVoters) => ([true* . bcast(n, blindmsg(n, 0)) . true*] forall d: Data . [finalTally(d)](val((n < numDishonestVoters => (countBallotsInTally(voter(n), tallyc(d)) <= 1)) && (n >= numDishonestVoters => (countBallotsInTally(voter(n), tallyc(d)) == 1)))))`

Appendix B

TD-1 model

B.1 mCRL model

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% SORTS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sort Data = struct ballot(voter: Nat, cand: Bool)?isBallot |
                   encrypt(msg: Data, key: Data)?isCrypted |
                   cryptedMsg(sender: Nat)?isCryptedMsg |
                   eKey?isKey |
                   null?isNull |
                   err?isErr;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% ACTIONS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% action regarding putting data on the broadcast channel
act bcast: Nat#Data;
% actions regarding data
act storedMessage, invalidMessage: Nat#Data;
% action regarding publishing of tally
act ftally: List(Bool);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% FUNCTIONS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% insListBallot: Function that inserts a ballot in a sorted list of ballots.
map insListBallot: Data#List(Data) -> List(Data);
var x, d: Data;
    xs: List(Data);
eqn insListBallot(d, []) = [d];
    insListBallot(d, x |> xs) = if(cand(d) < cand(x),
                                   d |> (x |> xs),
                                   x |> insListBallot(d, xs));

% insListNat: Function that inserts a value of type nat in a sorted list.
map insListNat: Nat#List(Nat) -> List(Nat);
var x, d: Nat;
    xs: List(Nat);
eqn insListNat(d, []) = [d];
    insListNat(d, x |> xs) = if(d < x,
                                   d |> (x |> xs),
                                   x |> insListNat(d, xs));

% updSingleList: Function to add a ballot to a local sorted list, given a
% list of local lists.
%
% parameters:
% 1st parameter i: Nat; the index of the local list in the list of lists
% 2nd parameter m: Data; the ballot to be added
% 3rd parameter x |> xs: List(List(Data)); a list of lists, containing all
%   local lists
%
% pre: value of i parameter is smaller than length of the list of lists
map updSingleList: Nat#Data#List(List(Data)) -> List(List(Data));
var m: Data;
```

```
      x: List(Data);
      xs: List(List(Data));
      i: Nat;
  eqn updSingleList(i, m, x |> xs) = if(i>0,
                                        (x |> updSingleList(Int2Nat(i-1), m, xs)),
                                        insListBallot(m, x) |> xs);

  % decrypt: Function to decrypt messages given a key. When the message can be
  % decrypted using the given key, the message is returned. If it can not, err is
  % returned.
  map decrypt: Data#Data -> Data;
  var d, k: Data;
  eqn decrypt(d, k) =
    if(isCrypted(d),
       if(k==key(d),
          msg(d),
          err),
       err);

  % sanitize: Function that removes voter identities from ballots.
  map sanitize: List(Data) -> List(Bool);
  var v: Nat;
      c: Bool;
      x: Data;
      xs: List(Data);
  eqn sanitize([]) = [];
      sanitize(x |> xs) = cand(x) |> sanitize(xs);

  % initProcList: Function that generates a list [0, 1, ... n-1], where n is given
  % as a parameter.
  map initProcList: Nat -> List(Nat);
  var i: Nat;
  eqn initProcList(i) = if(i > 0,
                           initProcList(Int2Nat(i-1)) <| Int2Nat(i-1),
                           []);

  % initList: Function to initialize lists containing a specific number of empty
  % lists.
  %
  % parameters:
  % 1st parameter n: Nat; the number of lists that need to be initialized.
  map initlist: Nat -> List(List(Data));
  var n: Nat;
  eqn initlist(n) = if(n>1,
                       [] |> initlist(Int2Nat(n-1)),
                       [[]]);

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% PROCESSES %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  proc Voters(N: Nat, bc: Data, votes: List(Bool), castList: List(Nat), key: Data,
              tallyList: List(List(Data)), procList: List(Nat)) =
    sum i: Nat . (0 <= i && i < N) -> (
      (!(i in castList) && #(procList) == N) ->
        bcast(i, encrypt(ballot(i, votes.i), key)) .
          Voters(N, encrypt(ballot(i, votes.i), key), votes, insListNat(i,
                 castList), key, tallyList, [])
      +
      (!(i in procList)) -> (
        (decrypt(bc, key) != err) ->
          storedMessage(i, decrypt(bc, key)) .
            Voters(N, bc, votes, castList, key, updSingleList(i, decrypt(bc, key),
                   tallyList), insListNat(i, procList))
        <>
          invalidMessage(i, bc) .
            Voters(N, bc, votes, castList, key, tallyList, insListNat(i,
                   procList))
      )
      +
```

```
      (#(tallyList.i)==N && i== 0) ->
        ftally(sanitize(tallyList.i)) . delta
);

map numVoters: Nat;
eqn numVoters = 6;

map votesVector: List(Bool);
eqn votesVector = [false, true, false, false, true, false];

init  hide({storedMessage, invalidMessage},
           Voters(numVoters, null, votesVector, [], eKey, initlist(numVoters),
                  initProcList(numVoters))
      );
```

B.1.1 Rename file

```
var
d: Data;
i: Nat;
rename
isCrypted(d) -> bcast(i, d) => bcast(i, cryptedMsg(i));
```

Appendix C

Proofs of invariants

In this chapter we provide proofs for the invariants in Table 4.3. We prove the invariants one by one, but since we (as is shown in Section 4.3) need that the conjunct of the invariants of Table 4.3 hold in the equation system, we are allowed to use other invariants in the proof of a certain invariant as long as the proof of those invariants does not rely on the validity of the invariant we are proving. The predicate formula in the equation system of Table 4.2 contains six conjuncts. Due to similarities (as discussed in Section 4.3), we only consider the first three. Since the third conjunct does not contain a predicate variable, we only need to prove that the invariant holds for the first two conjuncts. For clarity we prove each of these conjuncts separately.

Invariant 1 The first invariant says that the number of voters in both systems should be the same. Formally, $\iota_1 : N = N'$.

1. First conjunct, to prove:

   $$\begin{array}{l}\forall_{i:[0,N)}\cdot \\ \quad ((N = N')\wedge \\ \quad ((i \notin cast \wedge \#proc = N) \implies (i \notin cast' \wedge \#proc' = N')) \wedge i \notin cast \wedge \#proc = N) \implies \\ \quad (N = N')\end{array}$$

   Which is trivially true.

2. Second conjunct, to prove:

   $$\forall_{i,j:[0,N)}.((N = N') \wedge (i \notin proc \implies j \notin proc') \wedge i \notin proc) \implies (N = N')$$

   Which is also trivially true.

Therefore we can conclude that ι_1 is a global invariant for our equation system.

Invariant 2 The second invariant says that when a certain voter has not yet cast his vote, his ballot is not in a tally of a voter and is also not on the broadcast channel. Formally, $\iota_2 : \forall_{k,l:[0,N)}.k \notin cast \implies (blt(k, votes.k) \notin tally.l \wedge blt(k, votes.k) \neq dec(bc))$.

1. First conjunct, to prove:

   $$\begin{array}{l}\forall_{i:[0,N)}\cdot \\ \quad (\forall_{k,l:[0,N)}.k \notin cast \implies (blt(k, votes.k) \notin tally.l \wedge blt(k, votes.k) \neq dec(bc))\wedge \\ \quad\quad ((i \notin cast \wedge \#proc = N) \implies (i \notin cast' \wedge \#proc' = N')) \wedge i \notin cast \wedge \#proc = N) \implies \\ \quad (\forall_{k,l:[0,N)}.k \notin (\{i\} \cup cast) \implies \\ \quad\quad (blt(k, votes.k) \notin tally.l \wedge blt(k, votes.k) \neq dec(enc(blt(i, votes.i)))))\end{array}$$

   For the proof we start with the following case distinction.

(a) $i = k$
Term $k \notin (\{i\} \cup cast)$ evaluates to false when $i = k$. Since false implies everything, the invariant holds for the case $i = k$.

(b) $i \neq k$
We again need to make a case distinction.

i. $k \in cast$
If $k \in cast$ then also $k \in (\{i\} \cup cast)$. Then $k \notin (\{i\} \cup cast)$ is again false and the invariant thus also holds when $i \neq k$ and $k \in cast$.

ii. $k \notin cast$
If $k \notin cast$, we know also that $\forall_{l:[0,N)}.(blt(k, votes.k) \notin tally.l \wedge blt(k, votes.k) \neq dec(bc))$. Since $k \notin cast$ and $i \neq k$, it holds that $k \notin (\{i\} \cup cast)$. Therefore we need to prove that the following holds: $\forall_{l:[0,N)}.(blt(k, votes.k) \notin tally.l \wedge blt(k, votes.k) \neq dec(enc(blt(i, votes.i))))$. The first conjunct, $blt(k, votes.k) \notin tally.l$, follows from the assumptions (that is, the left hand side of the implication). The second conjunct, $blt(k, votes.k) \neq dec(enc(blt(i, votes.i)))$, follows from the fact that $i \neq k$. Hence the invariant also holds when $i \neq k$ and $k \notin cast$.

The invariant thus holds for the first conjunct.

2. Second conjunct, to prove:

$$\begin{array}{l}\forall_{i,j:[0,N)}.\\ \quad (\forall_{k,l:[0,N)}.k \notin cast \implies (blt(k, votes.k) \notin tally.l \wedge blt(k, votes.k) \neq dec(bc)) \wedge\\ \quad (i \notin proc \implies j \notin proc') \wedge i \notin proc) \implies\\ \quad (\forall_{k,l:[0,N)}.k \notin cast \implies (blt(k, votes.k) \notin updSet(i, dec(bc), tally).l \wedge\\ \quad\quad blt(k, votes.k) \neq dec(bc)))\end{array}$$

Note that function $updSet(i, dec(bc), tally)$ inserts the value $dec(bc)$ in the set $tally.i$. For the proof we start with the following case distinction.

(a) $i = l$

i. $k \in cast$
Trivial, since $k \notin cast$ evaluates to false.

ii. $k \notin cast$
If $k \notin cast$, we need to prove that: $(blt(k, votes.k) \notin updSet(i, dec(bc), tally).l \wedge blt(k, votes.k) \neq dec(bc))$. The first conjunct follows from the assumptions: it holds that $blt(k, votes.k) \notin (dec(bc) \cup tally.l)$ since $blt(k, votes.k) \notin tally.l$ and $blt(k, votes.k) \neq dec(bc)$. The second conjunct follows directly from the assumptions.

(b) $i \neq l$
Since $updSet(i, dec(bc), tally)$ only updates $tally.i$, all tallies $tally.j$ for $j \neq i$ remain unchanged. For those i, what needs to be proven follows directly from the assumptions, since no parameters are changed for $i \neq l$.

The invariant ι_2 is thus a global invariant for our equation system.

Invariant 3 The third invariant says that if and only if a trusted device has processed the message that is on the broadcast channel, the decryption of that message is in its tally. Formally, $\iota_3 : \forall_{k:[0,N)}.(k \in proc \iff dec(bc) \in tally.k)$.

1. First conjunct, to prove:

$$\begin{array}{l}\forall_{i:[0,N)}.\\ \quad (\forall_{k:[0,N)}.(k \in proc \iff dec(bc) \in tally.k) \wedge\\ \quad\quad ((i \notin cast \wedge \#proc = N) \implies (i \notin cast' \wedge \#proc' = N')) \wedge i \notin cast \wedge \#proc = N) \implies\\ \quad (\forall_{k:[0,N)}.(k \in \emptyset \iff dec(enc(blt(i, votes.i))) \in tally.k))\end{array}$$

Term $k \in \emptyset$ evaluates to false for all $k \in [0, N)$. Now we need to prove that for all $k \in [0, N)$ the term $dec(enc(blt(i, votes.i))) \in tally.k$ also evaluates to false. Hence we need to prove that $blt(i, votes.i) \notin tally.k$ for all $k \in [0, N)$. From our assumptions we know that $i \notin cast$. In combination with invariant ι_2 it follows that for all $l \in [0, N)$ it holds that $blt(i, votes.i) \notin tally.l \wedge blt(i, votes.i) \neq dec(bc)$. Since no tally is changed, it follows from the assumptions in combination with invariant ι_2 that $blt(i, votes.i) \notin tally.k$ for all $k \in [0, n)$. Hence the invariant is true for the first conjunct.

2. Second conjunct, to prove:

$$\begin{array}{l}\forall_{i,j:[0,N)}\cdot \\ \quad (\forall_{k:[0,N)}.(k \in proc \iff dec(bc) \in tally.k) \wedge \\ \quad (i \notin proc \implies j \notin proc') \wedge i \notin proc) \implies \\ \quad (\forall_{k:[0,N)}.(k \in (\{i\} \cup proc) \iff dec(bc) \in updSet(i, dec(bc), tally).k))\end{array}$$

For the proof we start with the following case distinction.

(a) $i = k$
If $i = k$ then $k \in (\{i\} \cup proc)$ is true. Then we need to prove that $dec(bc) \in updSet(i, dec(bc), tally).k$ is also true. Function $updSet(i, dec(bc), tally).k$ returns, for $i = k$, $\{dec(bc)\} \cup tally.i$. Hence $dec(bc) \in updSet(i, dec(bc), tally).k$.

(b) $i \neq k$
Term $updSet(i, dec(bc), tally).k$ rewrites to $tally.k$ for all $k \in [0, N) \wedge k \neq i$. Therefore, what we need to prove is already in the assumptions.

The invariant ι_3 is thus also a global invariant for our equation system.

Invariant 4 The fourth invariant says that the lists of votes in the two processes are permutations of each other. Formally, $\iota_4 : (cand(dec(bc)) \wedge dec(bc) \notin tally.0) + count(\top, tally.0) + \sum_{k:[0,N) \wedge k \notin cast} votes.k = (cand(dec(bc')) \wedge dec(bc') \notin tally'.0) + count(\top, tally'.0) + \sum_{k:[0,N') \wedge k \notin cast'} votes'.k$. Recall that $\top$ is counted as one in a summation and $\bot$ as zero.

1. First conjunct, to prove:

$$\begin{array}{l}\forall_{i:[0,N)}\cdot \\ \quad ((cand(dec(bc)) \wedge dec(bc) \notin tally.0) + count(\top, tally.0) + \sum_{k:[0,N) \wedge k \notin cast} votes.k = \\ \quad\quad (cand(dec(bc')) \wedge dec(bc') \notin tally'.0) + count(\top, tally'.0) + \sum_{k:[0,N') \wedge k \notin cast'} votes'.k \\ \quad\quad \wedge((i \notin cast \wedge \#proc = N) \implies (i \notin cast' \wedge \#proc = N)) \wedge i \notin cast \wedge \#proc = N) \implies \\ \quad (((cand(dec(enc(blt(i, votes.i)))) \wedge dec(enc(blt(i, votes.i))) \notin tally.0) + \\ \quad\quad count(\top, tally.0) + \sum_{k:[0,N) \wedge k \notin (\{i\} \cup cast)} votes.k = (cand(dec(enc(blt(i, votes'.i)))) \wedge \\ \quad\quad dec(enc(blt(i, votes'.i))) \notin tally'.0) + count(\top, tally'.0) + \\ \quad\quad \sum_{k:[0,N') \wedge k \notin (\{i\} \cup cast')} votes'.k)\end{array}$$

First, we reduce the right hand side of the implication, which gives us the following term: $(votes.i \wedge blt(i, votes.i) \notin tally.0) + cnt(\top, tally.0) + \sum_{k:[0,N) \wedge k \notin (\{i\} \cup cast)} votes.k = (votes'.i \wedge blt(i, votes'.i) \notin tally'.0) + cnt(\top, tally'.0) + \sum_{k:[0,N') \wedge k \notin (\{i\} \cup cast')} votes'.k$. From the assumptions it holds that $\#proc = N$. In combination with invariant ι_6 it follows that $proc = \{0, \ldots, N-1\}$. Then, when we combine that with invariant ι_3 it follows that for all $i \in [0, N)$ it holds that $dec(bc) \in tally.i$. Hence $cand(dec(bc)) \wedge dec(bc) \notin tally.0$ (in the left hand side of the implication) reduces to zero.

From the assumptions it also holds that $i \notin cast$. In combination with invariant ι_2 it follows that for all $l \in [0, N)$, $blt(k, votes.k) \notin tally.l$. Therefore the conjunction $(votes.i \wedge blt(i, votes.i) \notin tally.0)$ reduces to $votes.i$. Since, given that $i \notin cast$ it holds that:

$$votes.i + \sum_{\substack{k:[0,N) \wedge \\ k \notin (\{i\} \cup cast)}} votes.k = \sum_{\substack{k:[0,N) \\ \wedge k \notin cast}} votes.k$$

The same holds for the primed version (which can be checked in the same way). Hence it follows that the invariant holds for the first conjunct.

2. Second conjunct, to prove:

$$\begin{array}{l}\forall_{i,j:[0,N)}\cdot\\ \quad((cand(dec(bc)) \wedge dec(bc) \notin tally.0) + count(\top, tally.0) + \sum_{k:[0,N)\wedge k\notin cast} votes.k =\\ \quad\quad(cand(dec(bc')) \wedge dec(bc') \notin tally'.0) + count(\top, tally'.0) + \sum_{k:[0,N')\wedge k\notin cast'} votes'.k\\ \quad\wedge(i \notin proc \implies j \notin proc') \wedge i \notin proc) \implies\\ \quad((cand(dec(bc)) \wedge dec(bc) \notin updSet(i, dec(bc), tally).0) + count(\top,\\ \quad\quad updSet(i, dec(bc), tally).0) + \sum_{k:[0,N)\wedge k\notin cast} votes.k = (cand(dec(bc')) \wedge dec(bc') \notin\\ \quad\quad updSet(j, dec(bc'), tally).0) + count(\top, updSet(j, dec(bc'), tally').0)+\\ \quad\quad \sum_{k:[0,N')\wedge k\notin cast'} votes'.k)\end{array}$$

From $i \notin proc$ (and thus also $j \notin proc'$) it follows in combination with invariant ι_3 that $dec(bc) \notin tally.i$) (respectively $dec(bc') \notin tally'.0$). Then we proceed by making the following case distinction.

(a) $i = 0$
If $i = 0$, it follows from invariant ι_3 that $dec(bc) \notin tally.0$, hence on the LHS of the implication $(cand(dec(bc)) \wedge dec(bc) \notin tally.0)$ rewrites to $cand(dec(bc))$. On the RHS of the implication $(cand(dec(bc)) \wedge dec(bc) \notin updSet(i, dec(bc), tally).0)$ rewrites to zero. Since $dec(bc) \notin updSet(i, dec(bc), tally).0$ for rewrites for $i = 0$ to $dec(bc) \cup tally.0$ it follows that $count(\top, updSet(i, dec(bc), tally).0$ rewrites to the same term as $count(\top, tally.0) + cand(dec(bc))$. Since $\sum_{k:[0,N)\wedge k\notin cast} votes.k$ is both at the LHS as well as the RHS of the implication, it follows that the LHS of the equation remains constant. That is, the LHS of the equation at the LHS of the implication has the same value as the LHS of the equation at the RHS of the implication.

(b) $i \neq 0$
When $i \neq 0$, the LHS of the equation is the same for the LHS and the RHS of the implication.

The same case distinction can be made for j. Then it follows that the LHS of the invariant is always constant, as well as the RHS of the invariant. Therefore the invariant holds.

Since the invariant holds for both conjuncts we conclude that invariant ι_5 holds for our equation system.

Invariant 5 The fifth invariant says that the total number of votes expressed in both processes should be the same. Formally, $\iota_5 : (cand(dec(bc)) \wedge dec(bc) \notin tally.0) + count(\top, tally.0) + count(\bot, tally.0) = (cand(dec(bc')) \wedge dec(bc') \notin tally'.0) + count(\top, tally'.0) + count(\bot, tally'.0)$

1. First conjunct, to prove:

$$\begin{array}{l}\forall_{i:[0,N)}\cdot\\ \quad((cand(dec(bc)) \wedge dec(bc) \notin tally.0) + count(\top, tally.0) + count(\bot, tally.0) =\\ \quad\quad(cand(dec(bc')) \wedge dec(bc') \notin tally'.0) + count(\top, tally'.0) + count(\bot, tally'.0)\wedge\\ \quad\quad((i \notin cast \wedge \#proc = N) \implies (i \notin cast' \wedge \#proc' = N')) \wedge i \notin cast \wedge \#proc = N) \implies\\ \quad((cand(dec(enc(blt(i, votes.i)))) \wedge dec(enc(blt(i, votes.i))) \notin tally.0)+\\ \quad\quad count(\top, tally.0) + count(\bot, tally.0) = (cand(dec(enc(blt(i, votes'.i))))\wedge\\ \quad\quad dec(enc(blt(i, votes'.i))) \notin tally'.0) + count(\top, tally'.0) + count(\bot, tally'.0))\end{array}$$

We can rewrite the RHS of the implication to the following term: $(votes.i \wedge blt(i, votes.i) \notin tally.0) + count(\top, tally.0) + count(\bot, tally.0) = (votes'.i \wedge blt(i, votes'.i) \notin tally'.0) + count(\top, tally'.0) + count(\bot, tally'.0)$. On the LHS of the implication term $(cand(dec(bc)) \wedge$

$dec(bc) \notin tally.0$) rewrites to zero since $\#proc = N$ in combination with invariants ι_3 and ι_6. On the RHS of the implication the term ($votes.i \wedge blt(i, votes.i) \notin tally.0$) rewrites to zero since $i \notin cast$ in combination with invariant ι_2. The same holds for the primed version. Hence the invariant holds for this case.

2. Second conjunct, to prove:

$$
\begin{array}{l}
\forall_{i,j:[0,N)} \cdot \\
\quad ((cand(dec(bc)) \wedge dec(bc) \notin tally.0) + count(\top, tally.0) + count(\bot, tally.0) = \\
\qquad (cand(dec(bc')) \wedge dec(bc') \notin tally'.0) + count(\top, tally'.0) + count(\bot, tally'.0) \wedge \\
\quad (i \notin proc \implies j \notin proc') \wedge i \notin proc) \implies \\
\quad ((cand(dec(bc)) \wedge dec(bc) \notin updSet(i, dec(bc), tally).0) + \\
\qquad count(\top, updSet(i, dec(bc), tally).0) + count(\bot, updSet(i, dec(bc), tally).0) = \\
\qquad (cand(dec(bc')) \wedge dec(bc') \notin updSet(j, dec(bc'), tally').0) + \\
\qquad count(\top, updSet(j, dec(bc'), tally').0) + count(\bot, updSet(j, dec(bc'), tally').0))
\end{array}
$$

   For the proof we start with the following case distinction.

   (a) $i = 0$
   Since $i \notin proc$ in combination with invariant ι_3 it follows that $cand(dec(bc)) \wedge dec(bc) \notin tally.0$ rewrites to $cand(dec(bc))$. Furthermore, the term $cand(dec(bc)) \wedge dec(bc) \notin updSet(i, dec(bc), tally).0$ rewrites to zero. Note that the following equality holds for all $b \in \mathbb{B}$: $count(b, updSet(i, dec(bc), tally).0) = count(b, tally.0) + (b \iff cand(dec(bc)))$. Hence we can make the following derivation:

$$
\begin{array}{ll}
 & count(\top, updSet(i, dec(bc), tally).0) + count(\bot, updSet(i, dec(bc), tally).0) \\
= & \\
 & count(\top, tally.0) + count(\bot, tally.0) + (\top \iff cand(dec(bc))) + \\
 & \qquad (\bot \iff cand(dec(bc))) \\
= & \\
 & count(\top, tally.0) + count(\bot, tally.0) + cand(dec(bc))
\end{array}
$$

   Hence the LHS of the equation remains constant.

   (b) $i \neq 0$
   For $i \neq 0$ nothing changes on the RHS of the implication since the tally for $i \neq 0$ is not inspected.

   The same case distinction can be made for j. Then it follows that the LHS of the invariant is always constant, as well as the RHS of the invariant. There the invariant holds.

Since the invariant holds for both conjuncts we conclude that also invariant ι_6 holds for our equation system.

Invariant 6 The sixth invariant says that if and only if all trusted devices have processed a certain message, the length of the set indicating which devices have processed a message is equal to the number of trusted devices. Formally, $\iota_6 : \#proc = N \iff proc = \{0, \ldots, N-1\}$. This invariant is needed to reason about sets in the proof of other invariants, for example invariants ι_4 and ι_5.

1. First conjunct, to prove:

$$
\begin{array}{l}
\forall_{i:[0,N)} \cdot \\
\quad (\#proc = N \iff proc = \{0, \ldots, N-1\} \wedge \\
\qquad ((i \notin cast \wedge \#proc = N) \implies (i \notin cast' \wedge \#proc' = N')) \wedge i \notin cast \wedge \#proc = N) \implies \\
\quad (\#\emptyset = N \iff \emptyset = \{0, \ldots, N-1\})
\end{array}
$$

   For the proof we make the following case distinction.

(a) $N = 0$
Both the LHS and the RHS of the bi-implication evaluate to true.

(b) $N > 0$
Both the LHS and the RHS of the bi-implication evaluate to false.

Hence for this case the invariant holds.

2. Second conjunct, to prove:

$$\forall_{i,j:[0,N)} \cdot \\ (\#proc = N \iff proc = \{0, \ldots, N-1\} \wedge \\ (i \notin proc \implies j \notin proc') \wedge i \notin proc) \implies \\ (\#(\{i\} \cup proc) = N \iff (\{i\} \cup proc) = \{0, \ldots, N-1\})$$

For the proof we start with the following case distinction.

(a) $\#(\{i\} \cup proc) = N$

i. $(\#(\{i\} \cup proc) = N \implies (\{i\} \cup proc) = \{0, \ldots, N-1\})$
From invariant ι_8 we know that $proc \subseteq \{0, \ldots, N-1\}$. Since $i \in [0, N)$ it also holds that $(\{i\} \cup proc) \subseteq \{0, \ldots, N-1\}$. Note that $\#(\{i\} \cup proc) = N = \#\{0, \ldots, N-1\}$. Then, according to Theorem 8 it holds that $(\{i\} \cup proc) = \{0, \ldots, N-1\}$.

ii. $(\#(\{i\} \cup proc) = N \impliedby (\{i\} \cup proc) = \{0, \ldots, N-1\})$
This implication holds since $\#(\{0, \ldots, N-1\})$.

Hence the invariant holds for this case.

(b) $\#(\{i\} \cup proc) \neq N$

i. $\#(\{i\} \cup proc) \neq N \implies (\{i\} \cup proc) \neq \{0, \ldots, N-1\}$
Since $\#(\{i\} \cup proc) \neq N$ and $\#\{0, \ldots, N-1\} = N$ it follows that $\#(\{i\} \cup proc) \neq \#\{0, \ldots, N-1\}$. Then, by Theorem 8 it follows that $(\{i\} \cup proc) \neq \{0, \ldots, N-1\}$.

ii. $\#(\{i\} \cup proc) \neq N \impliedby (\{i\} \cup proc) \neq \{0, \ldots, N-1\}$
From invariant ι_8 we know that $proc \subseteq \{0, \ldots, N-1\}$ and we also known that $i \in [0, N)$, thus $(\{i\} \cup proc) \subseteq \{0, \ldots, N-1\}$. Using Theorem 8 it follows that that: $\neg((\{i\} \cup proc) \subseteq \{0, \ldots, N-1\}) \vee \neg(\#(\{i\} \cup proc) = \#(\{0, \ldots, N-1\}))$. Since we have proven that $(\{i\} \cup proc) \subseteq \{0, \ldots, N-1\}$ it follows that $\#(\{i\} \cup proc) \neq N$.

Hence the invariant also holds for this case.

Therefore we can conclude that invariant ι_6 holds for our equation system.

Theorem 8. *Given two sets A and B it holds that:*

$$A \subseteq B \wedge \#A = \#B \Leftrightarrow A = B$$

Proof. We proof the bi-implication by proving the two implications separately.

1. $A \subseteq B \wedge \#A = \#B \Rightarrow A = B$
To prove $A = B$ we need to prove $A \subseteq B$ and $B \subseteq A$.

(a) $A \subseteq B \wedge \#A = \#B \Rightarrow A \subseteq B$
Trivial.

(b) $A \subseteq B \wedge \#A = \#B \Rightarrow B \subseteq A$
We can rewrite the proof obligation to: $\neg(A \subseteq B) \vee \neg(\#A = \#B) \vee B \subseteq A$. Assume the negation of that: $A \subseteq B \wedge \#A = \#B \wedge \neg(B \subseteq A)$. Since $\neg(B \subseteq A)$ there has to be an $x \in B$ for which $x \notin A$. When that is the case it follows that $A \subseteq (B \setminus \{x\})$. This means that $\#A \leq \#(B \setminus \{x\})$ and thus (given that $x \in B$) $\#A \leq \#B - 1$ hence $\#A < \#B$. This contradicts with $\#A = \#B$. Hence $A \subseteq B \wedge \#A = \#B \Rightarrow B \subseteq A$.

2. $A \subseteq B \wedge \#A = \#B \Leftarrow A = B$
   By definition.

□

Invariant 7 The seventh invariant says that the number of items in the tally, plus one when the current message on the broadcast channel is not processed, is equal to the number of voters if and only if all ballots are cast. Formally, $\iota_7 : (dec(bc) \notin tally.0 + \#tally.0) = N \iff cast = \{0, \ldots, N-1\}$.

1. First conjunct, to prove:

   $$\begin{array}{l}\forall_{i:[0,N)} \cdot \\ \quad ((dec(bc) \notin tally.0 + \#tally.0) = N \iff cast = \{0, \ldots, N-1\} \wedge \\ \qquad ((i \notin cast \wedge \#proc = N) \implies (i \notin cast' \wedge \#proc' = N')) \wedge i \notin cast \wedge \#proc = N) \implies \\ \quad ((dec(enc(blt(i, votes.i))) \notin tally.0 + \#tally.0) = N \iff (\{i\} \cup cast) = \{0, \ldots, N-1\})\end{array}$$

   We can rewrite the RHS of the implication to the following term: $(blt(i, votes.i) \notin tally.0 + \#tally.0) = N \iff (\{i\} \cup cast) = \{0, \ldots, N-1\}$. From $i \notin cast$ follows by invariant ι_2 that for all $l \in [0, N)$ that $blt(i, votes.i) \notin tally.l \wedge blt(i, votes.i) \neq dec(bc)$. Hence $blt(i, votes.i) \notin tally.0$ rewrites to one. Using $\#proc = N$ in combination with invariants ι_6 and ι_3 it follows that $dec(bc) \in tally.0$. Hence $dec(bc) \notin tally.0$ rewrites to zero and $blt(i, votes.i) \notin tally.0$ to one. Then we continue the proof by making the following case distinction.

   (a) $blt(i, votes.i) \notin tally.0 + \#tally.0 = \#tally.0 + 1 = N$
   To prove: $(\{i\} \cup cast) = \{0, \ldots, N-1\}$. From invariant ι_{11} we have $cast \subseteq \{0, \ldots, N-1\}$, furthermore we know $i \in [0, N)$. Hence we have $(\{i\} \cup cast) \subseteq \{0, \ldots, N-1\}$. From invariant ι_{12} we know that $dec(bc) \notin tally.0 + \#tally.0 = N - 1 = \#cast$. Now we can apply Theorem 8 to conclude that $(\{i\} \cup cast) = \{0, \ldots, N-1\}$.

   (b) $dec(bc) \notin tally.0 + \#tally.0 = \#tally.0 = N$
   When $\#tally.0 = N$ it holds that $cast = \{0, \ldots, N-1\}$. Therefore there is no $i \in [0, N)$ for which $i \notin cast$. Thus we have a universal quantification over an empty domain. Hence the invariant holds for this case.

   (c) $\#tally.0 \neq N \wedge \#tally.0 + 1 \neq N$
   From $\#tally.0 \neq N \wedge \#tally.0 + 1 \neq N$ in combination with invariant ι_{13} it follows that $\#tally.0 \leq N - 2$. From invariant ι_{12} it follows then that $\#cast \leq N - 2$. Since $\#tally.0 \neq N$ it follows that $cast \neq \{0, \ldots, N-1\}$. We have to prove: $(\{i\} \cup cast) \neq \{0, \ldots, N-1\}$. From what we know it follows that $\#(\{i\} \cup cast) \leq N-1$ and $\{0, \ldots, N-1\} = N$. Now we can apply Theorem 8 to conclude that $(\{i\} \cup cast) \neq \{0, \ldots, N-1\}$.

2. Second conjunct, to prove:

   $$\begin{array}{l}\forall_{i,j:[0,N)} \cdot \\ \quad ((dec(bc) \notin tally.0 + \#tally.0) = N \iff cast = \{0, \ldots, N-1\} \wedge \\ \quad (i \notin proc \implies j \notin proc') \wedge i \notin proc) \implies \\ \quad ((dec(bc) \notin updSet(i, dec(bc), tally).0 + \#updSet(i, dec(bc), tally).0) = N \iff \\ \qquad cast = \{0, \ldots, N-1\})\end{array}$$

   For the proof we start with the following case distinction.

   (a) $i = 0$
   From $i \notin proc$ we know using invariant ι_3 that $dec(bc) \notin tally.i$. The term $dec(bc) \notin tally.0$ therefore rewrites to one. The term $(dec(bc) \notin updSet(i, dec(bc), tally).0$ rewrites to zero, but $\#updSet(i, dec(bc), tally).0$ rewrites to $\#tally.0 + 1$. Hence the LHS of the equation remains constant and the invariant holds for this case.

(b) $i \neq 0$
Nothing changes since tally.i for $i \neq 0$ is not inspected. Hence the invariant holds also for this case.

Therefore we can conclude that invariant ι_7 holds for our equation system.

Invariant 8 The eighth invariant says that the set of trusted devices that processed a certain broadcast message is a subset of the the set of all trusted devices. Formally, $\iota_8 : proc \subseteq \{0, \ldots, N-1\}$.

1. First conjunct, to prove:

$$\begin{array}{l}\forall_{i:[0,N)} \cdot \\ \quad (proc \subseteq \{0, \ldots, N-1\} \wedge \\ \qquad ((i \notin cast \wedge \#proc = N) \implies (i \notin cast' \wedge \#proc' = N')) \wedge i \notin cast \wedge \#proc = N) \implies \\ \quad (\emptyset \subseteq \{0, \ldots, N-1\})\end{array}$$

   The empty set is a subset of all sets. Hence for this case the invariant holds.

2. Second conjunct, to prove:

$$\begin{array}{l}\forall_{i,j:[0,N)} \cdot \\ \quad (proc \subseteq \{0, \ldots, N-1\} \wedge (i \notin proc \implies j \notin proc') \wedge i \notin proc) \implies \\ \quad ((\{i\} \cup proc) \subseteq \{0, \ldots, N-1\})\end{array}$$

   Since $i \in [0, N)$ and $proc \subseteq \{0, \ldots, N-1\}$ it holds that $(\{i\} \cup proc) \subseteq \{0, \ldots, N-1\}$.

Since the invariant holds for both conjuncts we conclude that the invariant ι_8 holds for our equation system.

Invariant 9 The ninth invariant says that the number of votes that has been cast in both systems should be the same. Formally, $\iota_9 : cast = cast'$.

1. First conjunct, to prove:

$$\begin{array}{l}\forall_{i:[0,N)} \cdot \\ \quad (cast = cast' \wedge \\ \qquad ((i \notin cast \wedge \#proc = N) \implies (i \notin cast' \wedge \#proc' = N')) \wedge i \notin cast \wedge \#proc = N) \implies \\ \quad ((\{i\} \cup cast) = (\{i\} \cup cast'))\end{array}$$

   Holds trivially.

2. Second conjunct, to prove:

$$\begin{array}{l}\forall_{i,j:[0,N)} \cdot \\ \quad (cast = cast' \wedge (i \notin proc \implies j \notin proc') \wedge i \notin proc) \implies (cast = cast')\end{array}$$

   Holds also trivially.

Therefore we can conclude that invariant ι_9 holds for our equation system.

Invariant 10 The tenth invariant says that when all ballots are in the tally, it must be that the ballot on the broadcast channel has been processed. Formally, $\iota_{10} : \#tally.0 = N \implies dec(bc) \in tally.0$.

1. First conjunct, to prove:

   $$\forall_{i:[0,N)} \cdot \\ (\#tally.0 = N \implies dec(bc) \in tally.0 \wedge \\ ((i \notin cast \wedge \#proc = N) \implies (i \notin cast' \wedge \#proc' = N')) \wedge i \notin cast \wedge \#proc = N) \implies \\ (\#tally.0 = N \implies dec(enc(blt(i, votes.i))) \in tally.0)$$

   We start the proof by making the following case distinction.

   (a) $\#tally.0 \neq N$
   Trivial, false implies all.

   (b) $\#tally.0 = N$
   We strengthen the assumptions with invariant $\iota_7 : (dec(bc) \notin tally.0 + \#tally.0) = N \iff cast = \{0, \ldots, N-1\}$. Since $\#tally.0 = N$ and therefore $dec(bc) \in tally.0$, it follows from invariant ι_7 that for all $k \in [0, N)$ it holds that $k \in cast$. Hence there is no i for for which $i \notin cast$. Thus we have a universal quantification over an empty domain. Therefore it follows that the invariant holds.

2. Second conjunct, to prove:

   $$\forall_{i,j:[0,N)} \cdot \\ (\#tally.0 = N \implies dec(bc) \in tally.0 \wedge (i \notin proc \implies j \notin proc') \wedge i \notin proc) \implies \\ (\#updSet(i, dec(bc), tally).0 = N \implies dec(bc) \in updSet(i, dec(bc), tally).0)$$

   We make the following case distinction.

   (a) $i \neq 0$
   Holds since $tally.0$ is not updated when $i \neq 0$.

   (b) $i = 0$

      i. $\#updSet(i, dec(bc), tally).0 \neq N$
      Trivial since false implies all.

      ii. $\#updSet(i, dec(bc), tally).0 = N$
      To prove: $dec(bc) \in updSet(i, dec(bc), tally).0)$ holds given the assumptions. This holds since $updSet(i, dec(bc), tally).0)$ is equal to $\{dec(bc)\} \cup tally.0$

Therefore we can conclude that invariant ι_{10} also holds for our equation system.

Invariant 11 The eleventh invariant says that the set of trusted devices that have cast a ballot is a subset of all trusted devices. Formally, $\iota_{11} : cast \subseteq \{0, \ldots, N-1\}$.

1. First conjunct, to prove:

   $$\forall_{i:[0,N)} \cdot \\ (cast \subseteq \{0, \ldots, N-1\} \wedge \\ ((i \notin cast \wedge \#proc = N) \implies (i \notin cast' \wedge \#proc' = N')) \wedge i \notin cast \wedge \#proc = N) \implies \\ ((\{i\} \cup cast) \subseteq \{0, \ldots, N-1\})$$

   Holds since $cast \subseteq \{0, \ldots, N-1\}$ and $i \in [0, N)$. Therefore it follows that $(\{i\} \cup cast) \subseteq \{0, \ldots, N-1\}$.

2. Second conjunct, to prove:

$$\begin{array}{l}\forall_{i,j:[0,N)}\cdot\\ \quad (cast \subseteq \{0,\ldots,N-1\}\wedge\\ \quad (i \notin proc \Longrightarrow j \notin proc') \wedge i \notin proc) \Longrightarrow\\ \quad (cast \subseteq \{0,\ldots,N-1\})\end{array}$$

Holds trivially.

Therefore we can conclude that invariant ι_{11} holds for our equation system.

Invariant 12 The twelfth invariant says that the number of ballots cast is equal to the number of ballots in the tally plus one for the ballot in the broadcast channel if that one has not been processed yet. Formally, $\iota_{12} : dec(bc) \notin tally.0 + \#tally.0 = \#cast$.

1. First conjunct, to prove:

$$\begin{array}{l}\forall_{i:[0,N)}\cdot\\ \quad (dec(bc) \notin tally.0 + \#tally.0 = \#cast\wedge\\ \quad\quad ((i \notin cast \wedge \#proc = N) \Longrightarrow (i \notin cast' \wedge \#proc' = N')) \wedge i \notin cast \wedge \#proc = N) \Longrightarrow\\ \quad (dec(enc(blt(i, votes.i))) \notin tally.0 + \#tally.0 = \#(\{i\} \cup cast))\end{array}$$

The RHS of the implication can be rewritten to $blt(i, votes.i) \notin tally.0 + \#tally.0 = \#(\{i\} \cup cast)$. From $i \notin cast$ and invariant ι_2 it follows that for all $l \in [0, N)$ that $dec(bc) \notin tally.l$. Hence $blt(i, votes.i) \notin tally.0$ rewrites to zero. From $\#proc = N$ in combination with invariants ι_6 and ι_3 it follows that $dec(bc) \notin tally.0$. The invariant holds for this case since $\#tally.0 = \#cast$ implies $\#tally.0 + 1 = \#(\{i\} \cup cast) = \#cast + 1$.

2. Second conjunct, to prove:

$$\begin{array}{l}\forall_{i,j:[0,N)}\cdot\\ \quad (dec(bc) \notin tally.0 + \#tally.0 = \#cast\wedge\\ \quad (i \notin proc \Longrightarrow j \notin proc') \wedge i \notin proc) \Longrightarrow\\ \quad (dec(bc) \notin updSet(i, dec(bc), tally).0 + \#updSet(i, dec(bc), tally).0 = \#cast)\end{array}$$

   (a) $i = 0$
   From $i \in proc$ it follows using invariant ι_{10} that $dec(bc) \notin tally.i$. Term $dec(bc) \notin tally.0$ rewrites thus to one and therefore $\#tally.0 + 1 = \#cast$. Term $dec(bc) \notin updSet(i, dec(bc), tally).0$ rewrites to zero. It holds that $\#updSet(i, dec(bc), tally).0 = \#tally.0 + 1$ and therefore it follows that the invariant holds for this case.

   (b) $i \neq 0$
   When $i \neq 0$ $tally.0$ is not updated. Therefore it follows that the invariant holds.

Therefore we can conclude that invariant ι_{12} holds for our equation system.

Invariant 13 The thirteenth invariant says that the number of ballots in the tally can never exceed the number of trusted devices in the protocol. Formally, $\iota_{13} : \#tally.0 \leq N$.

1. First conjunct, to prove:

$$\begin{array}{l}\forall_{i:[0,N)}\cdot\\ \quad (\#tally.0 \leq N\wedge\\ \quad\quad ((i \notin cast \wedge \#proc = N) \Longrightarrow (i \notin cast' \wedge \#proc' = N')) \wedge i \notin cast \wedge \#proc = N) \Longrightarrow\\ \quad (\#tally.0 \leq N)\end{array}$$

For this conjunct the invariant holds trivially.

2. Second conjunct, to prove:

$$\forall_{i,j:[0,N)} \cdot \\ (\#tally.0 \leq N \wedge \\ (i \notin proc \implies j \notin proc') \wedge i \notin proc) \implies \\ (\#updSet(i, dec(bc), tally).0 \leq N)$$

From $i \notin proc$ in combination with invariant ι_3 it follows that $dec(bc) \notin tally.i$. Then we proceed the proof with the following case distinction.

(a) $i \neq 0$
Invariant holds since $tally.0$ is not updated.

(b) $i = 0$

i. $\#tally.0 = N$
If $\#tally.0 = N$ it follows using invariant ι_{10} that $dec(bc) \in tally.0$ This contradicts with $dec(bc) \notin tally.0$. Hence we have a universal quantifier over an empty domain, thus the invariant holds for this case.

ii. $\#tally.0 < N$
Since $\#updSet(i, dec(bc), tally).0$ rewrites to $\#tally.0 + 1$ it follows that the invariant holds because $\#tally.0 + 1 \leq N$.

Therefore we can conclude that invariant ι_{13} holds for our equation system.