Under consideration for publication in Formal Aspects of Computing

Cryptographic Protocols with Everyday Objects

James Heather[1], Steve Schneider[2] and Vanessa Teague[3]

[1]Department of Computing, University of Surrey, Guildford, Surrey GU2 7XH, UK j.heather@surrey.ac.uk
[2]Department of Computing, University of Surrey, Guildford, Surrey GU2 7XH, UK s.schneider@surrey.ac.uk
[3]Dept. Computer Science and Software Engineering, University of Melbourne vjteague@unimelb.edu.au

Abstract. Most security protocols appearing in the literature make use of cryptographic primitives that assume that the participants have access to some sort of computational device.

However, there are times when there is need for a security mechanism to evaluate some result without leaking sensitive information, but computational devices are unavailable. We discuss here various protocols for solving cryptographic problems using everyday objects: coins, dice, cards, and envelopes.

1. Introduction

The last forty years have seen an explosion in the number of cryptographic techniques we have at our disposal: as well as symmetric encryption, we now have asymmetric encryption, hash functions, Diffie-Hellman key exchange, zero-knowledge proofs, homomorphic encryption, threshold encryption, digital signatures, blinding, and many others. This has led to a wide range of applications that would previously have been impossible: electronic cash, electronic voting, electronic contract signing, encrypted searches on encrypted data, and so on.

These techniques are entirely appropriate when there is a demand for high security and computational power is readily available. However, sometimes the need arises for something more mundane: a security protocol that can be run on the spot, with no advance planning, and with no electronic computational capabilities available. This paper deals with such situations. We describe mechanisms for running simple cryptographic protocols using everyday objects.

Usually in such situations the security requirements are less strict than in an electronic setting. For this reason, some of the protocols that we will consider will be treated under the *honest-but-curious* model, in which all participants are assumed to obey the rules of the protocol, but will happily use all information available to deduce others' secrets wherever possible. In other words, such participants operate exactly as honest participants, except that they cannot be trusted not to learn extra information whenever they can do so undetectably. Send such an agent an encrypted secret for which he has the decryption key, and he will decrypt it; pass a playing card face down to such an agent and then look the other way, and the agent will not swap it for another card, but will certainly take a sneaky peek. Some of the protocols, however, will be secure even under a stronger adversarial model.

It should be noted that there are similar protocols in real-world use that operate even in environments where one would hope that security is paramount. For instance, some military aircraft are rumoured to have

Correspondence and offprint requests to: James Heather (j.heather@surrey.ac.uk) mechanical interlocks that enable one and only one of several war plans to be released to the onboard crew, in order to prevent the crew from learning information that does not need to be disclosed to them about other war plans. Postal voting relies on a protocol whereby the voter fills in an anonymous ballot paper, inserts it into an unmarked envelope, inserts this into a second envelope and signs the outer one, so that the electoral officials can verify who has voted (from the outer envelopes), but so that, after shuffling the inner envelopes, individual votes cannot be linked to individual voters.

The protocols presented here are all information-theoretically secure; that is, they do not rely on computational assumptions about attacker capabilities or on problems that are assumed to be hard.

As well as describing the protocols and their properties, we will show how to construct a formal proof that the protocols meet their claimed properties, by modelling the protocols in CSP [Ros10, Sch99a].

The paper is arranged as follows. Section 2 contains a brief introduction to CSP, which will be needed for the formal modelling later in the paper. In Section 4 we discuss Chaum's dining cryptographers protocol that uses coins, and give an extension of it. Section 5 then considers a dating protocol attributed to Bennett that uses playing cards, and extends it to a three-player unanimity protocol; we then use CSP to prove that these protocols provide the desired properties. Then in Section 6 we look at the Secret Santa problem, and give a solution that uses envelopes; we also prove this protocol secure using CSP. This section then continues by constructing a veto protocol, and then a threshold voting protocol, also using envelopes; the desired properties of these protocols are proven to hold using CSP too. Finally in Section 7 we sum up and draw conclusions.

2. CSP background

CSP describes systems in terms of *processes*, which interact by means of synchronising on *events*. The set of all events that a process P can engage in is called its *alphabet*, written αP. Events can be structured, for example $place.i.x$ may represent a player placing a card x at position i; in general, $c.v$ represents value v passing on channel c. The set $\{|c|\}$ denotes the set of all events of the form $c.v$. The set of all events is denoted Σ.

The CSP language is used to describe processes. $Stop_A$ represents the process with alphabet A that cannot engage in any events; A may be omitted when it is clear from the context. Process $a \to P$ initially performs a, and subsequently behaves as P. The input process $c?x : S \to P(x)$ can receive a value $x \in S$ on channel c, and then behave as $P(x)$. The set S can also be expressed as a predicate. The output process $c!v \to P$ outputs value v along channel c, and then behaves as P. $\bigsqcap_i P_i$ is an internal choice: the process decides which of the P_i to behave as. It also admits a binary form $P \sqcap Q$. $P \,\Box\, Q$ offers an external choice: it is prepared to behave as either P or Q, and the choice is resolved by its environment (as long as the initial events offered by P and those offered by Q are distinct, as will be the case for all examples in this paper). $\|_i^X P_i$ is the parallel composition of the P_i, in which occurrence of any event $a \in X$ requires the synchronous participation of all processes, but events outside X can be performed independently. This operator also has a binary form $P \underset{X}{\|} Q$. One important special case of interface parallel is where $X = \emptyset$, in which case we write it as '$P \,|||\, Q$'; here P and Q operate entirely independently, with no synchronization. $P \setminus A$ behaves as P but with all events from the set A hidden and performed internally. The process $P[[_X \setminus {}^{x'}]]$ behaves as P, except that any event in X is renamed to x'; we can also use $P[[_{X,Y} \setminus {}^{x',y'}]]$ to introduce multiple renamings at the same time. (CSP allows one-to-many renamings, but we will not use them in this paper.) Process definitions take the form $N = P$, and can be recursive (i.e., the definition of P contains N). N can also be parameterised.

CSP has several semantic models, which are appropriate for capturing different aspects of process behaviour. In this paper, we use the traces model, because we are interested only in whether an observer can learn secret information by seeing events that occur as the protocol proceeds. None of the protocols will require consideration of refusals or failures.

3. Modelling in CSP

The protocols in the rest of the paper will make use of various things from the physical world that can be thought of as primitives: coins, dice, cards, envelopes and signatures. We consider them to have the following properties:

Dice are a mechanism for choosing a random number from 1 to (usually) 6 in such a way that

- the operator can be seen not to have influenced the result;
- the result can be revealed to, or hidden from, those in close proximity.

Coins are essentially two-sided dice (whose faces are typically labelled HEADS and TAILS).

A deck of **cards** contains (usually) 52 cards, each with two faces; one side (the back) is the same on all cards, and the other side (the front) contains a unique image for each card. The cards can be passed around face down (so that no one can see which card is which) or face up (so that each can be identified). Additionally, the cards can be held so that one participant can see the fronts and one participant can see the backs. An ordered sequence of cards can be *shuffled* (so that the order is randomized) or *cut* (cycled around) so that an observer can tell that the operator has shuffled or cut correctly, but cannot learn anything about the new ordering of the cards.

Envelopes have two distinguishable faces, initially blank, but which can be written on in secret or in public; envelopes, like cards, can be passed around either face up or face down. Envelopes can also have *contents*, which are not visible when inserted into the envelope, but are visible when taken out. Additionally, if the contents are pieces of paper or card, they can be slid out to reveal one face but not both. A stack of envelopes can be shuffled or cut in the same way as a deck of cards. Finally, envelopes contain a *seal*: they can be shut so that the contents cannot be disclosed without visibly breaking the seal.

Signatures can be written onto envelopes and pieces of paper. Each participant has a unique signature that none of the others can produce; anyone can verify any other participant's signature. Signing a piece of card or an envelope therefore allows it to be authenticated back to the signer at a later time.

For the most part, the protocols presented here are secure only under the honest-but-curious model, in which the participants are assumed to follow the protocol correctly, but will happily infer whatever secrets they can from what they observe. Our modelling will prove that they do not, in fact, learn information that they should not learn. Modelling this scenario in CSP will mean modelling honest participants who follow the protocol; defining an appropriate abstraction that represents the participants' view of the model; and demonstrating that secret information cannot be learnt from this view.

Since the focus is on whether sensitive information can leak out, the general approach will be as follows. Checking the protocols for correctness requires both a CSP model and an appropriate specification. We first construct a model of the protocol and its participants, including events to represent the actions of the participants (for instance, an event to indicate that a particular playing card has been placed at a particular place on the table). We then construct a specification that claims that, from the point of view of either one of the participants or an observer, two states that should be indistinguishable are in fact indistinguishable. This is done by first abstracting the view of the model to hide events that are not visible to that participant, and then performing various refinement checks in FDR to confirm that, under this abstraction, two different states can no longer be distinguished.

We will use three different techniques for hiding things from the view of a participant or observer. The simplest is to use the CSP hiding operator: $P \setminus X$ acts as P, but events in H are hidden from view. For a more selective abstraction, we will at times use renaming to model cases where one part of an event is hidden; so, for instance, we might rename $lay.Alice.Bob$ to $lay.X.X$ (where X is some dummy value, roughly corresponding to $\perp$ in a functional language). This will allow the occurrence of lay to stay visible, but abstract its contents away. Finally, we can hide events in the specification itself: rather than asserting that $SPEC \sqsubseteq_T IMPL$, we can assert

$$SPEC \mid\mid\mid CHAOS(H) \sqsubseteq_T IMPL \mid\mid\mid CHAOS(H)$$

Since $CHAOS(H)$ can perform events from H at any time, this has the effect of abstracting such events away: we can no longer tell whether they occurred as part of $IMPL$ or as actions of $CHAOS(H)$.

One of the protocols is secure under a stronger adversarial model, in which a dishonest participant can manipulate a stack of envelopes under the table. Modelling this requires us to add the extra possible actions of the adversary into the CSP process controlling his activity. The most efficient and comprehensive way of doing so is to allow these nefarious actions to occur at any appropriate time and in any order; many such combinations will not be of use to the adversary, but since all combinations are allowed by the model, any that prove to be successful will be picked up by the model checker.

4. Coins and Dice: Dining Cryptographers

One of the most discussed protocols that uses everyday objects is Chaum's solution to the *dining cryptographers* problem [Cha88]; it has become a standard test case for formal analysis [SS96, vdMS04, KLN$^+$06].

There are n cryptographers having dinner ($n > 2$) round a circular table, and their company has either chosen one of them to pay the bill, or decided to cover the bill itself. If the former, then the 'lucky' cryptographer has been notified. The cryptographers want to determine whether the company is paying or not, without revealing any information about which cryptographer is paying in the event that the company is not.

Chaum's solution runs as follows. Each pair of adjacent cryptographers tosses a coin so that those two can see the result but the others cannot. Each cryptographer then reports to the group whether the two coins he saw gave the same result; he is required to tell the truth if he is not paying the bill, but to lie if he is paying.

A parity argument tells us that the number of cryptographers reporting that the coins are different will be even if all tell the truth. If the cryptographers are honest (that is, if they obey the protocol), then the number reporting a difference will be odd if and only if one cryptographer is paying the bill.

4.1. Extending to multiple payers

Let us suppose that the company may have chosen more than one cryptographer to share paying the bill. How can they extend the protocol to allow for this?

If they have dice available, this can be easily accomplished. The dice are simply a means of choosing a number from a uniform distribution over $\mathbb{Z}_6$, just as a coin chooses uniformly from $\mathbb{Z}_2$; we will describe the protocol using six-sided dice and make the assumption that the number paying is no more than 5, but if this threshold is not enough then the random numbers could be chosen using, for example, a pack of cards.

To extend the protocol, each pair of adjacent cryptographers should throw a die, again so that only they can see the result. Each cryptographer reports the result of $L - R + p$ modulo 6, where L is the die he sees on the left, R is the die he sees to his right, and p is 0 if he is not paying, and 1 if he is. Each die will be added once and subtracted once, so that the sum of the reported values, modulo 6, gives the number of cryptographers who are paying.

Like Chaum's original protocol, this protocol assumes that the cryptographers are honest but curious: if they are able to determine more than they should from the information available to them, then they will do so; but they will always follow the protocol correctly.

5. Cards: Unanimity and Vetoes

Charles Bennett [Sin11] describes a *dating protocol*, to be used perhaps by participants at a speed-dating evening who need to decide whether to pursue things afterwards. Alice and Bob wish to determine whether they both want to go on a date; but they want to avoid the embarrassing situation in which one of them does not want to go on a date, but knows that the other would have liked to do so. Essentially they need a two-player veto protocol: they want to compute whether at least one has vetoed the date, without revealing any further information.

Bennett's solution uses playing cards. It relies on having five cards, all indistinguishable from the back; when revealed, the five cards should divide into one set of identical triplets and one pair of identical twins. Normal packs of cards are therefore not well suited to this. We shall use playing cards in this section, partly for ease of exposition, and partly because Bennett's protocol was first described in these terms; we will need to assume that we have three of the Queen of Clubs and two of the King of Hearts, all with the same back. However, it is worth noting that the same effect can be easily achieved using five identical business cards.

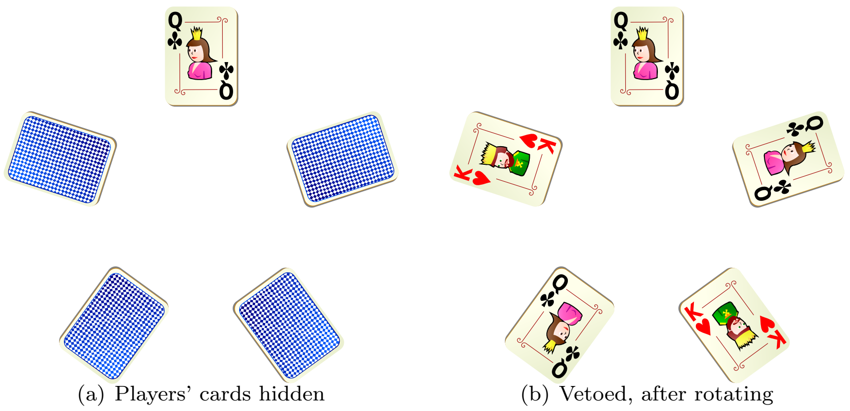
(a) Players' cards hidden (b) Vetoed, after rotating

Fig. 1. Bennett's dating protocol

They can be turned into three triplets either by carefully marking the front of three of them in one corner, or by using the orientation of the card to encode what we will call 'King' or 'Queen'. Needless to say, for this latter option, one needs to pick up and lay down the cards carefully to avoid destroying the orientation.

Figure 1(a) shows the starting point. A Queen is placed on the table, face up; Alice lays two cards face down in the spaces on the left, and Bob lays two cards face down in the spaces on the right. Each has a King and a Queen; to veto, the King is placed next to the fixed Queen, and to vote in favour, the Queen is placed next to the fixed Queen so that the King is at the bottom of the circle. The key point to observe is that if neither has vetoed then the two Kings are adjacent to each other, but if either has vetoed then they are not.

The fixed Queen is now turned face down. They now need to rotate the circle of cards, keeping the order intact, so that neither of them knows how much it has been rotated. The easiest way to do this is for one of them to pick the cards up in a clockwise fashion, arranging them into a deck; each of them can then repeatedly cut the deck until they are both satisfied that it has been sufficiently rotated; then the cards can be laid out again, and turned face up.

There are only two possible results: either the two Kings are adjacent (in which case neither of them vetoed) or the two Kings are not adjacent (in which case at least one of them vetoed). Figure 1(b) shows the result following a veto.

5.1. Unanimity for three players

We show how to use a similar idea to construct a unanimity protocol for three players. They will use the protocol to determine whether they all agree on the point at issue; if they do not all agree, then they will not discover who stands alone against the other two. This is useful in situations where eventual consensus is required, but there is a danger that voters might be unduly influenced by knowledge of the other votes.

The protocol itself will not distinguish unanimous assent from unanimous dissent; but, of course, once unanimity has been established, each player will know which way the unanimity went, because he knows his own vote.

In this case, we need six cards, in two sets of identical triplets. We will continue to use playing cards to describe the protocol, but remind the reader that, in practice, business cards may be easier to come by.

Figure 2(a) shows the starting point. The cards should all be indistinguishable when face down, but we have drawn different patterns on the backs to show which player uses which cards.

The three players are sitting at 6 o'clock, 10 o'clock and 2 o'clock. Each controls the card in front of him, and the card diametrically opposite. To vote in favour, he places the King face down immediately in front of him and the Queen face down opposite; to vote against, he places the Queen in front and the King opposite.

As in the dating protocol, the cards are collected, repeatedly cut and laid out again, maintaining the order but applying a random rotation.

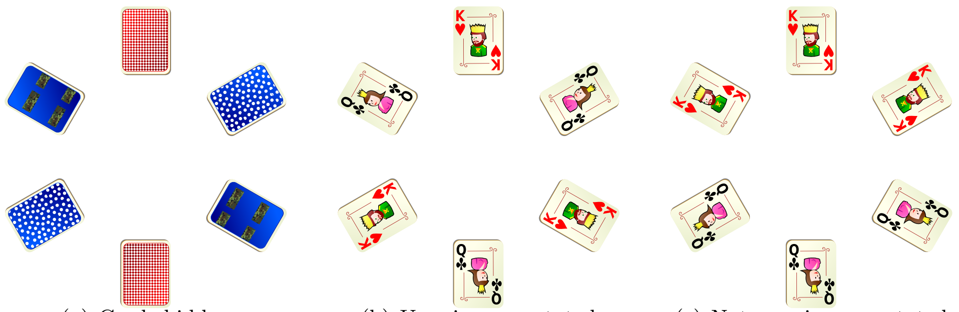
(a) Cards hidden (b) Unanimous, rotated (c) Not unanimous, rotated

Fig. 2. Three-player unanimity protocol

Again, there are only two possible results[1]. Either the cards alternate, in which case there was unanimity (everyone voted in favour, or everyone voted against); or the three Kings are next to each other and so are the three Queens, in which case there was no unanimity.

The dating protocol and the unanimity protocol do not require honesty on the part of the participants, on the assumption that they cannot cheat undetectably when cutting the deck. They do require careful phrasing of their properties, however. In the dating protocol, for instance, we can conclude that Alice gains no information about Bob's vote if she decides to veto; but this is not quite the same as saying that Alice gains no information about whether Bob wants to go on a date if she does not want to, because their votes might not express their true preferences.

5.2. Formal modelling

We show here how to construct a formal proof of correctness of the dating protocol and the unanimity protocol. We have modelled both protocols in CSP, and proven them correct using the FDR model checker [For97]. The full code for both appears in Appendix A; what follows is an overview. Figure 3 contains the CSP for the most important parts of the model of the dating protocol; the unanimity protocol is very similar and uses much of the same code.

Each player will decide whether to *accept* or *veto*, and will then place his cards down accordingly. The places are numbered anticlockwise, with 0 being the fixed Queen; player 1 uses places 1 and 2, and player 2 uses places 3 and 4. (In the machine-readable code, we have constructed the player processes from generic parametric player processes so as to make as much code as possible common to the two protocols, but for clarity we here give the expanded version of the player processes.)

The card collection process is defined in terms of a helper process *COLLECTING* that takes three parameters: the cards already on the table, the number of cards to pick up, and the starting position of the first pick-up. The details are in Appendix A; the process will eventually build a sequence of five cards, and then pass that sequence to the *ANNOUNCE* process.

ANNOUNCE is also defined in terms of a helper process whose full details appear in the appendix. It first constructs all possible rotated versions of the input sequence; then it chooses one non-deterministically, and outputs it on channel *announce*. Thus $ANNOUNCE(\langle Q, K, K\rangle)$ may output any of $\langle Q, K, K\rangle$, $\langle K, Q, K\rangle$, and $\langle K, K, Q\rangle$.

The whole system is then the interleaved combination of the two players, put in parallel with the card collector.

We will check that if player 1 vetoes, he cannot tell whether player 2 also vetoed. We start by constructing player 1's view of the system, by hiding the events that should be invisible to player 1—that is, by hiding player 2's card placement events and veto or acceptance events. We then set things up using a helper process *CONTROLS* that allows us to specify how the two players vote. This helper process is used to construct

[1] There is a third possible arrangement of three Kings and three Queens: KKQQKQ. However, this state is not reachable, since it necessitates having two Kings opposite each other and two Queens opposite each other, implying that two players did not follow the protocol.

$$
\begin{aligned}
P1\_DATE &= accept.1 \rightarrow place.1.Q \rightarrow place.2.K \rightarrow Stop \\
&\quad \Box\ veto.1 \rightarrow place.1.K \rightarrow place.2.Q \rightarrow Stop \\
P2\_DATE &= accept.2 \rightarrow place.3.K \rightarrow place.4.Q \rightarrow Stop \\
&\quad \Box\ veto.2 \rightarrow place.3.Q \rightarrow place.4.K \rightarrow Stop
\end{aligned}
$$

$$COLLECT\_CARDS\_DATE = COLLECTING(\langle Q\rangle, 4, 1)$$

$$ANNOUNCE(xs) = ANNOUNCE\_FROM(allrots(xs))$$

$$
\begin{aligned}
DATE\_SYSTEM = {} & (P1\_DATE \mathbin{|||} P2\_DATE) \\
& \underset{\{|place.x|x\in\{1..4\}|\}}{\parallel} \\
& COLLECT\_CARDS\_DATE
\end{aligned}
$$

$$
\begin{aligned}
P2\_EVENTS &= \{|accept.2, veto.2, place.3, place.4|\} \\
P1\_DATE\_VIEW(choices) &= (DATE\_SYSTEM \underset{\{|accept,veto|\}}{\parallel} CONTROLS(choices)) \\
&\quad \setminus P2\_EVENTS
\end{aligned}
$$

Fig. 3. CSP code for the dating protocol

$P1\_DATE\_VIEW(\langle x, y\rangle)$, which represents player 1's view of the system when player 1 and 2 cast votes x and y respectively, where 0 represents acceptance and 1 represents a veto.

The check that the system satisfies the required property is then

$$P1\_DATE\_VIEW(\langle 1,0\rangle) =_T P1\_DATE\_VIEW(\langle 1,1\rangle)$$

which FDR confirms to be the case.

The model of the unanimity protocol is similar, but we have chosen this time to analyse it from the point of view of an external observer, who should be able to tell the difference between unanimity and lack of it, but not distinguish unanimous assent from unanimous dissent. By a similar line of reasoning, we construct a process EXT_UNANIM_VIEW, and verify using FDR that

$$EXT\_UNANIM\_VIEW(\langle 0,0,0\rangle) =_T EXT\_UNANIM\_VIEW(\langle 1,1,1\rangle)$$

and that

$$EXT\_UNANIM\_VIEW(\langle a,b,c\rangle) =_T EXT\_UNANIM\_VIEW(\langle d,e,f\rangle)$$

whenever $\{a,b,c\} = \{d,e,f\} = \{0,1\}$.

A final check that

$$EXT\_UNANIM\_VIEW(\langle 0,0,0\rangle) \neq_T EXT\_UNANIM\_VIEW(\langle 0,0,1\rangle)$$

shows that the the two equivalence classes are different. This is essentially a sanity check that the protocol is not pointless: it does reveal one bit of information, namely, whether there was unanimity among the participants.

6. Envelopes: Secret Santa and Vetoes

Another promising avenue of exploration for performing simple cryptography with everyday objects is that of envelopes. In this section, we show how to use envelopes to solve the Secret Santa problem (Section 6.1) and to provide a more comprehensive veto protocol and threshold voting protocol (Section 6.2) than we were able to achieve with cards.

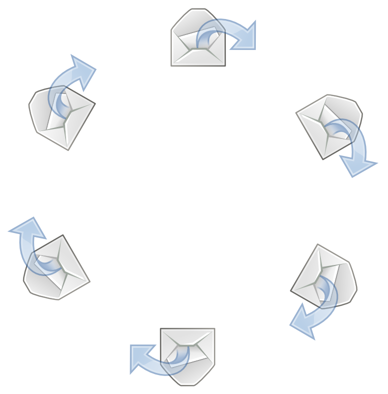

Fig. 4. Father Cryptmas: moving the pieces of card

6.1. The Secret Santa problem

The *Secret Santa* problem centres round a group of friends who wish to allocate gift-buying responsibilities so that each friend gives a gift to one other friend, and receives a gift from one other friend. They want to arrange these responsibilities so that no one knows anything about the assignments other than the name of the friend for which he should be buying a gift. They require a mechanism that ensures that:

1. every participant is allocated another participant for whom to buy a present;
2. the allocation is bijective, irreflexive and random;
3. every participant learns for whom he must buy a present;
4. no participant learns any other allocations.

It is the last of these requirements that we shall concentrate on here, though the others will be ensured by construction.

One traditional way of assigning these responsibilities is by drawing names out of a hat. However, this can be tedious because of the requirement that no one should have to buy a present for himself; every time someone draws his own name from the hat, the protocol needs to be restarted.

Mauw, Radomirović and Ryan discuss this problem, and provide a solution [MRR10]. Their algorithm gives an arbitrary derangement, so that it is guaranteed that no one is assigned to buy a gift for himself; it does not make any further restrictions on the nature of the assignments. It also adds one further security requirement, to the effect that if anyone does not receive a present, the cheating participant can be revealed.

However, one of their protocols requires ElGamal [ElG85], and the other requires a fully homomorphic encryption system. Their protocols are thus suitable for those with some planning, some expertise and access to computational assistance, but not to a group of friends without access to computers.

Here we present an alternative solution, which does not lower the security requirements, and does not require anything beyond card, envelopes and pens.

6.1.1. The Father Cryptmas Protocol

Each participant should be given a piece of card and an envelope. The card and envelope should be thick enough that something can be written on the front without being visible from the back.

The participants each sign the piece of card, and sign the front of the envelope, and then insert the card into the envelope so that the name is facing forward, and close the envelope without sealing it. The envelopes are collected, face down, and shuffled until everyone is happy that the envelopes are in a random order known to no one. The envelopes are then laid out, face down, in a circle, and carefully opened without revealing the names on the envelopes or on the card. Each piece of card is then slid out of the envelope, and moved round one place in a clockwise direction, and inserted into its new envelope (see Figure 4). The envelopes are now sealed, collected again, and shuffled again.

Now each envelope is given to the participant whose signature appears on the front. Each participant buys a gift for the person whose signature appears on the card inside.

This is secure under the honest-but-curious model. In fact, it is also secure even with dishonest participants, provided that the physical signatures can be verified, and provided that participants are observed in sliding the cards in and out of the envelopes. Its only disadvantage over cryptographic solutions is that it produces a cycle rather than an arbitrary derangement; for most purposes, and especially Secret Santa set-ups in which the donor's name is revealed at present-giving time, this may in fact be an advantage (since it avoids cycles of length 2, with consequent annoyance on the part of the one who bought the better present).

6.1.2. Formal modelling

We show in this section how to model and verify the Father Cryptmas protocol of Section 6.1.1 in CSP. Again, the code presented here is an overview; we have constructed a full model and formally verified the protocol using FDR, and the full code appears in Appendix B.

Figure 5 contains the most important elements of the model.

Each person i will *offer* an envelope, having signed both the front of the envelope and the card inside. The envelopes are shuffled, and the person will then see an envelope *laid* in front of him, containing a signature on the front and a signature on the card inside. (Neither of these will be visible; we will mask these values later.) When everyone has received a new envelope (by synchronization on *sync*), he will then *pass* the card inside the envelope to the next person in the circle, and be passed a card from the previous person. After all cards have been moved, he will *return* the envelope, with its new card inside, for reshuffling; finally, he will *receive* a new envelope with his name on the front and someone else's name inside. He will then buy a *gift* for that person.

These *PERSON* processes are then combined and put in parallel with processes that handle the the two shuffles of the envelopes, along with two control processes that constrain the order of *pass* and *gift* events to prevent state space explosion.

We start with a sanity check on the model, specifying that gift-buying should be irreflexive; in terms of the model, this means that we should never see an event of the form *gift.x.x*. We first construct the set of such events, and then specify that they should never occur. The most efficient way of doing this is to specify that if we hide all other events, we end up with *STOP*:

$$\mathit{REFLEXIVE} = \{\mathit{gift}.x.x \mid x \in \mathit{PEOPLE}\}$$

$$\mathit{STOP} \sqsubseteq_T \mathit{SYSTEM} \setminus (\Sigma \setminus \mathit{REFLEXIVE})$$

For the security checks, we need a mechanism for controlling who will end up buying a gift for whom. We thus define a *FORCE_GIFTS* process that takes a sequence of pairs of names, and for each pair $\langle n_1, n_2\rangle$, ensures that n_1 buys a gift for n_2:

$$\mathit{SYSTEM\_WITH}(\mathit{Pairs}) = \mathit{SYSTEM} \underset{\{|\mathit{lay}|\}}{\parallel} \mathit{FORCE\_GIFTS}(\mathit{Pairs})$$

We will consider the system from Alice's perspective. We hide the events that do not concern her, and mask the names that she cannot see:

$$\begin{aligned}\mathit{SYSTEM\_ALICE\_WITH}(\mathit{Pairs}) = (&\mathit{SYSTEM\_WITH}(\mathit{Pairs}) \setminus \mathit{Non\_Alice\_Events})\\ &[[{}_{\mathit{lay}.i.n_1.n_2, \mathit{pass}.i.j.n_1, \mathit{return}.i.n_1.n_2} \setminus {}^{\mathit{lay}.i.X.X, \mathit{pass}.i.j.X, \mathit{return}.i.X.X}\\ &\quad \mid i, j \in \mathit{NUMS}, n_1 \in \mathit{Ident}, n_2 \in \mathit{Ident}]]\end{aligned}$$

Now we test the crucial properties. The traces model is the most natural choice for these checks, since what we are interested in is whether Alice can infer anything that ought to be secret by observing the events that take place around the table.

First, we specify that all behaviours of a system where Alice buys a gift for Bob are possible behaviours of a system where Charlie additionally buys a gift for Dawn (with Charlie and Dawn chosen without loss of generality). This has the effect of saying that whatever Alice observes is consistent with any gift assignments for those other than Alice and Bob, and so Alice cannot infer anything about the gift patterns. In fact, the refinement holds in both directions, giving us equality in the traces model:

$$\begin{aligned}&\mathit{SYSTEM\_ALICE\_WITH}(\langle\langle \mathit{Charlie}, \mathit{Dawn}\rangle, \langle \mathit{Alice}, \mathit{Bob}\rangle\rangle)\\ &\qquad =_T \mathit{SYSTEM\_ALICE\_WITH}(\langle\langle \mathit{Alice}, \mathit{Bob}\rangle\rangle)\end{aligned}$$

$$\begin{aligned}
\mathit{PERSON}(i) &= \mathit{offer}.i.\mathit{nameOf}(i).\mathit{nameOf}(i) \rightarrow \mathit{sync} \rightarrow \mathit{lay}.i?n_1?n_2 \\
&\rightarrow \mathit{sync} \rightarrow \mathit{PERSON\_SWAP}(i, n_1, n_2, \mathit{nxt}(i), \mathit{prev}(i)) \\
\mathit{PERSON\_SWAP}(i, n_1, n_2, \mathit{to}, \mathit{frm}) &= i < \mathit{frm}\ \&\ \mathit{pass}.i.\mathit{to}.n_2 \rightarrow \mathit{pass}.\mathit{frm}.i?n_3 \\
&\qquad \rightarrow \mathit{PERSON\_END}(i, n_1, n_3) \\
&\Box\ \mathit{frm} < i\ \&\ \mathit{pass}.\mathit{frm}.i?n_3 \rightarrow \mathit{pass}.i.\mathit{to}.n_2 \\
&\qquad \rightarrow \mathit{PERSON\_END}(i, n_1, n_3) \\
\mathit{PERSON\_END}(i, n_1, n_3) &= \mathit{sync} \rightarrow \mathit{return}.i.n_1.n_3 \rightarrow \mathit{sync} \\
&\rightarrow \mathit{receive}.i.\mathit{nameOf}(i)?n_4 \rightarrow \mathit{sync} \rightarrow \mathit{gift}.\mathit{nameOf}(i).n_4 \rightarrow \mathit{Stop}
\end{aligned}$$

$$\begin{aligned}
\mathit{SHUFFLE\_ME}(i) &= \mathit{offer}.i?n_1?n_2 \\
&\rightarrow \mathit{lay}?j!n_1!n_2 \rightarrow \mathit{Stop} \\
\mathit{COUNT\_SEQ}(c, 0) &= \mathit{Stop} \\
\mathit{COUNT\_SEQ}(c, n) &= c.(\mathit{SIZE} - n)?\_ \rightarrow \mathit{COUNT\_SEQ}(c, n-1)
\end{aligned}$$

$$\begin{aligned}
\mathit{SHUFFLING} &= (\ \mathop{|||}_{i \in NUMS}\ \mathit{SHUFFLE\_ME}(i)) \\
&\mathop{\|}_{\{|\mathit{offer},\mathit{lay}|\}}\ (\mathit{COUNT\_SEQ}(\mathit{offer}, \mathit{SIZE})\ |||\ \mathit{COUNT\_SEQ}(\mathit{lay}, \mathit{SIZE}))
\end{aligned}$$

$$\mathit{RESHUFFLING} = \mathit{SHUFFLING}[[_{\mathit{offer},\mathit{lay}} \setminus {}^{\mathit{return},\mathit{receive}}]]$$

$$\begin{aligned}
\mathit{ORDER\_GIFTS}(0) &= \mathit{Stop} \\
\mathit{ORDER\_GIFTS}(n) &= \mathit{gift.nth(SIZE\text{-}n,PEOPLESEQ)}?\_ \rightarrow \mathit{ORDER\_GIFTS}(n-1)
\end{aligned}$$

$$\begin{aligned}
\mathit{SYSTEM} &= ((\ \mathop{\|}_{i \in NUMS}^{AlphaPerson(i)}\ \mathit{PERSON}(i)) \setminus \{\mathit{sync}\}) \\
&\qquad \mathop{\|}_{\{|\mathit{offer},\mathit{lay},\mathit{return},\mathit{receive},\mathit{pass},\mathit{receive}|\}} \\
&(\mathit{SHUFFLING}\ |||\ \mathit{RESHUFFLING} \\
&\quad |||\ \mathit{COUNT\_SEQ}(\mathit{pass}, \mathit{SIZE})\ |||\ \mathit{ORDER\_GIFTS}(\mathit{SIZE}))
\end{aligned}$$

Fig. 5. CSP code for the Father Cryptmas protocol

FDR confirms that this holds.

Secondly, we check that if Alice buys for Bob then she can infer nothing about who buys her a gift:

$$SYSTEM\_ALICE\_WITH(\langle\langle Charlie, Alice\rangle, \langle Alice, Bob\rangle\rangle) \\ =_T SYSTEM\_ALICE\_WITH(\langle\langle Dawn, Alice\rangle, \langle Alice, Bob\rangle\rangle)$$

Again, FDR confirms the equality.

Thirdly, we check that Alice can, nonetheless, infer a small amount about who is buying her a gift. If we specify that she can never distinguish between a case where Charlie buys her a gift and a case where Dawn buys her a gift, the refinement fails:

$$SYSTEM\_ALICE\_WITH(\langle\langle Charlie, Alice\rangle\rangle) \sqsubseteq_T SYSTEM\_ALICE\_WITH(\langle\langle Dawn, Alice\rangle\rangle)$$

FDR returns a trace in which Alice is assigned to buy a gift for Charlie. This demonstrates that it is in the case where Alice buys a gift for Charlie that she is able to distinguish between a setup where Charlie buys her a present and a setup where Dawn buys her a present: the former, given Alice's knowledge that she is buying for Charlie, is impossible (and so is not a trace of the left-hand side); the latter is possible (and so is a trace of the right-hand side).

6.2. Vetoing with envelopes

In Section 5, we discussed a two-player veto protocol, and showed how to construct a three-player unanimity protocol. Here, we use envelopes to construct a veto protocol that works for any number of participants. Initially, we will describe a protocol that determines whether at least one participant has cast a veto, without revealing how many have done so; we will then discuss how to extend this to a threshold protocol that determines, without revealing any further information, whether at least k participants have vetoed the motion. The threshold version comes at a price: the security assumptions are a little stronger.

6.2.1. Single Veto Protocol

For n participants, the protocol requires $n + 1$ indistinguishable thick envelopes, $n + 1$ indistinguishable pieces of card (for example, business cards), and a large, opaque bag. One of the pieces of card should have 'NO VETO' written across it; the others will be designated as veto cards, and they should be left blank. (It would be more natural, in one sense, to leave the non-veto card blank and write 'VETO' on the others; but it is harder to keep the veto cards indistinguishable if they have all been written on.)

Each piece of card is put into its own envelope, so that everyone knows which envelope contains the non-veto, and is satisfied that each other envelope contains a veto card. The envelopes are sealed, and the non-veto envelope is placed into the bag. The n veto envelopes are left in a pile in the centre of the table.

Each participant in turn takes a veto envelope, and puts his hand into the bag. He then withdraws his hand, either still holding the veto envelope, or having swapped it for the envelope that is already in the bag. (This must be done so that the others cannot tell whether he has swapped the envelopes or not; it is for this reason that the bag needs to be large and opaque.) The one that he removes should be placed, unopened, on a discard pile for later.

When all participants have done this, either at least one person vetoed, in which case there will be a veto envelope in the bag and the non-veto envelope will be somewhere in the discard pile, or no one vetoed, in which case the non-veto envelope will be still in the bag and the discard pile will contain only veto envelopes.

The envelope inside the bag is tipped out, and opened in view of everyone. If it contains the non-veto card, then the motion is carried. In this case, the discard pile will contain only veto envelopes, so the ordering of the discard pile reveals no further information. If the envelope tipped out of the bag contains a veto card, then the motion is defeated; in this case, the location of the non-veto card in the discard pile will reveal the identity of the first participant to veto, and so the discard pile must be shuffled or destroyed.

This protocol is secure even against dishonest participants, as long as they are unable to place identifying marks on any of the cards or envelopes. If participants can track individual envelopes, then they will be able to infer additional information about the others' behaviour. This assumption seems reasonable in the low security contexts in which it might be deployed.

It should be noted that the whole protocol can be run in eco-mode, with zero wastage. The envelopes can be closed but not sealed; and if business cards are used, then the non-veto card can be inserted front-facing into the envelope, and the veto card can be inserted back-facing. This means that no envelopes need be ripped, and no cards need be written on. Not sealing the envelopes does give slightly more opportunity for manipulating the cards and envelopes inside the bag when casting a vote, however.

6.2.2. Threshold Voting Protocol

It is possible to adapt this protocol to allow for a threshold version: we now wish to determine whether at least k participants have objected to the motion, without revealing any further information. This can be achieved still under the honest-but-curious model, although the consequences of breaking that assumption will be greater. We first describe the protocol, and then some possible improvements to mitigate this.

The word 'veto' is inappropriate in a threshold context, since it implies that lone action can defeat the motion; so we will use 'NO' and 'YES' envelopes in place of the veto and non-veto envelopes of the previous version, and we will ask participants to *accept* or *reject* the motion.

We need $k+n$ indistinguishable thick envelopes, $k+n$ pieces of indistinguishable card, and a large, opaque cloth (for instance, a tablecloth or a bath towel). As before, a pile of n indistinguishable 'NO' envelopes is placed in the centre of the table. Next to it is placed a pile of k 'YES' envelopes. This second pile effectively implements a FIFO queue: items will be added to the top, and removed from the bottom.

The first participant takes a 'NO' envelope, and publicly places it on the top of the pile representing the FIFO queue. The cloth is now carefully laid over this pile. The same participant reaches under the cloth with one hand, and removes either the top envelope (if he wishes to accept the motion), or the bottom envelope (if he wishes to reject). In the first case, he has performed a null operation; in the second, he has added a 'NO' to the back of the queue, and removed an item from the front of the queue. The envelope he has removed should now be placed unopened on a discard pile, and the cloth removed.

Each participant in turn follows this procedure. By the end, if at least k people have rejected the motion, then the bottom envelope will be a 'NO'; if not, it will be a 'YES'. It can be publicly opened to determine the result.

The distribution of the remaining envelopes reveals information, and so, as before, they must be shuffled or destroyed. This can also be run in eco-mode.

The main security weakness is with the cloth. It needs to be thick enough so that no one can tell whether the top envelope or bottom envelope was removed. In the honest-but-curious model, where participants correctly follow the protocol, this is sufficient; for a stronger adversarial model in which participants may break the protocol, the cloth also needs to be thin enough to make it detectable if anyone rearranges the envelopes or takes anything other than the top or bottom envelope. If the last participant, for instance, can manipulate the pile, then he can cause the motion to fail regardless of how the others have voted.

There are strategies one might adopt to mitigate this. Perhaps the best option is to attach Velcro tabs to the top and bottom of each envelope before the protocol begins. This means that the FIFO queue now becomes a pile of envelopes that are stuck together all down the pile. Each participant publicly attaches a 'NO' envelope to the top, and then takes the whole pile and hides it under the table. He will be unable to manipulate the pile because each time two adjacent envelopes are separated there will be an audible 'ripping' sound from the Velcro tabs. If only one 'rip' is heard, and the result is a pile of $k-1$ envelopes and a single envelope, it must have been either the top envelope or the bottom envelope that was removed.

6.2.3. Formal modelling

Here we give a CSP model of these protocols, and prove that they meet their specifications using FDR. Since the veto protocol is equivalent to the threshold protocol using a threshold of 1, we model the threshold version. The highlights are given in Figure 6; the full code is to be found in Appendix C.

The key process is the one controlling the stack. When a player *casts* a vote, a 'NO' envelope will be *added* to the top of the stack. Then, under the table, the player removes either the envelope he just added or the envelope at the bottom of the stack. This removal elicits a *rip* event, the significance of which will become clear later.

The stack has two other important behavioural characteristics. First, a dishonest player may try to *manipulate* the stack under the table; we shall return to this when we introduce a dishonest player. Secondly, the envelope at the bottom of the stack may be opened, in which case the protocol terminates.

$$\begin{aligned}
STACK(\langle x\rangle \frown xs) &= cast?p.Y \rightarrow add.N \rightarrow rip \rightarrow STACK\_DISCARD(\langle N\rangle, \langle x\rangle \frown xs)\\
&\Box\ cast?p.N \rightarrow add.N \rightarrow rip \rightarrow STACK\_DISCARD(\langle x\rangle, xs \frown \langle N\rangle)\\
&\Box\ manipulate?p?v \rightarrow add.N \rightarrow SPLIT\_STACK(\langle\langle x\rangle \frown xs \frown \langle N\rangle\rangle)\\
&\Box\ open!x \rightarrow Stop\\
STACK\_DISCARD(\langle x\rangle, xs) &= discard!x \rightarrow STACK(xs)\\
SPLIT\_STACK(xss) &= (\#last(xss) == 1)\ \&\\
&finished \rightarrow STACK\_DISCARD(last(xss), concat(init(xss)))\\
&\quad\Box\\
&(\Box_{\substack{i\in\{1..\#xss\}\\ j\in\{1..\#xss[i]-1\}}}\ split.i.j \rightarrow rip \rightarrow SPLIT\_STACK(split\_at(xss,i,j)))\\
&\quad\Box\\
&(\Box_{\substack{i\in\{1..\#xss-1\}\\ j\in\{i+1..\#xss\}}}\ swap.i.j \rightarrow SPLIT\_STACK(swapped(xss,i,j)))\\
PLAYER(n) &= \bigsqcap_{e\in ENV}\ cast.n.e \rightarrow Stop
\end{aligned}$$

$$\begin{aligned}
COUNT(0) &= open?x \rightarrow Stop\\
COUNT(n) &= cast.n?e \rightarrow COUNT(n-1)\\
&\Box\ manipulate.n?e \rightarrow COUNT(n-1)
\end{aligned}$$

$$\begin{aligned}
SYSTEM = ((&\ \big|\big|\big|_{n\in\{1..PLAYERS\}}\ PLAYER(n))\\
&\underset{\{|cast,manipulate|\}}{\|}\\
&(STACK(STARTSTACK)\ \underset{\{|cast,open|\}}{\|}\ COUNT(PLAYERS)))\\
&\setminus \{|discard, manipulate, finished|\}
\end{aligned}$$

Fig. 6. CSP code for the Threshold Voting Protocol

A player process needs to do no more than choose a vote value and *cast* the relevant vote. The stack ensures that the appropriate envelopes are added and then removed.

To ensure that the envelope is not opened until the end, we introduce a counting process that keeps track of the number of votes that have been cast, and allows the opening of the bottom envelope only when all votes are in. It also counts a *manipulate* event, since a dishonest player will use that instead of casting.

The whole system, composed, for the moment, of honest players, is then the interleaving of the processes controlling the players, synchronized with the stack (initialised with *STARTSTACK*, a sequence of k 'YES' envelopes) and the counting process over the relevant events (initialised with *PLAYERS*, the number of participants). We hide the internal events representing stack operations, except for the dishonest actions, which will be abstracted away later in the specification.

To check that this satisfies the right property in the honest-but-curious model, we define a specification that counts the votes, and checks that the contents of the bottom envelope correspond to the correct group outcome[2]:

$$\begin{aligned}
\mathit{SPEC} &= \mathit{SPECCOUNT}(\mathit{PLAYERS}, \mathit{THRESHOLD}) \\
&\ ||| \ \mathit{CHAOS}(\{|\mathit{rip}, \mathit{swap}, \mathit{split}|\}) \\
\mathit{SPECCOUNT}(0, \mathit{credit}) &= \mathit{credit} \leqslant 0 \,\&\, \mathit{open}.N \to \mathit{Stop} \\
&\ \Box\ \mathit{credit} > 0 \,\&\, \mathit{open}.Y \to \mathit{Stop} \\
\mathit{SPECCOUNT}(\mathit{unvoted}, \mathit{credit}) &= \mathit{unvoted} > 0 \,\& \\
&\quad (\mathit{cast}.\mathit{unvoted}.N \to \mathit{add}.N \to \mathit{SPECCOUNT}(\mathit{unvoted}-1, \mathit{credit}-1) \\
&\ \sqcap\ \ \mathit{cast}.\mathit{unvoted}.Y \to \mathit{add}.N \to \mathit{SPECCOUNT}(\mathit{unvoted}-1, \mathit{credit}))
\end{aligned}$$

FDR assures us that the system meets its specification:

$$\mathit{SPEC} \sqsubseteq_T \mathit{SYSTEM}$$

The focus here is on whether it is possible for the wrong result to emerge; for this reason, we leave the *cast* events visible. This is not because they are naturally visible to an observer, but because we consider this type of specification from a 'God's eye' perspective: we want to know whether the declared result corresponds to the votes that were in fact cast.

To model a dishonest player, we take advantage of the stack's capabilities for being manipulated. The stack has operations for *splitting* it into several substacks, and for *swapping* any two substacks. In this way, it is possible to produce any permutation of the stack. (We assume that the dishonest player has nothing 'up his sleeve', as it were: the envelopes that he takes under the table also come back up again.) Rather than *casting* a vote, then, the dishonest player decides to *manipulate* the stack; in doing so, he passes his player number (so that the counting process can keep track of who has had a turn), and the result that he is attempting to produce. The purpose of this declaration of intent is that we do not want to consider it an attack if the dishonest player simply disenfranchises himself: the question will be whether he can produce a result that he should not be able to produce. We will thus check later whether he has been able to manipulate the result in a particular direction.

After he has chosen to *manipulate*, the stack will then *add* a 'NO' envelope (since this is done in public view). The dishonest player then takes the stack under the table. After that, he may engage in any number of *splits* and *swaps*. When he finishes, he must have got it to the point where the bottom envelope is separated from the rest of the stack; he can then *finish*, and publicly discard the bottom envelope. For the moment, we will assume that no one can tell anything about his actions hidden under the table (though they can see that he is not looking at the envelopes).

In order to ensure that the dishonest player is not disadvantaged by sitting at the wrong seat around the table, we allow him to start by choosing his seat. He chooses a value of i, and then assumes that seat; he is

[2] The purpose of interleaving with $\mathit{CHAOS}(\{|\mathit{rip}, \mathit{swap}, \mathit{split}|\})$ is to abstract these events away: they can now happen at any time. The alternative would be to hide these events in the system; however, this would produce a divergent system in what follows when we introduce the dishonest player. Although this would not technically cause problems, it is perhaps stylistically undesirable.

then placed in combination with honest players at other seats:

$$\mathit{WITH\_BADDIE} = \bigsqcap_{i \in \{1..\mathit{PLAYERS}\}} (\mathit{BADDIE}(i) \mathbin{|||} (\mathop{\Big|\Big|\Big|}_{n \neq i} \mathit{PLAYER}(n)))$$

$$\begin{aligned}
\mathit{BADDIE}(i) &= \mathit{manipulate}.i?v \to \mathit{CHEATING}(i) \\
\mathit{CHEATING}(i) &= \mathit{swap}?x?y \to \mathit{CHEATING}(i) \\
&\quad \Box\ \mathit{split}?x?y \to \mathit{rip} \to \mathit{CHEATING}(i) \\
&\quad \Box\ \mathit{finished} \to \mathit{discard}?x \to \mathit{Stop}
\end{aligned}$$

The new system is exactly as before, but with the dishonest player added:

$$\begin{aligned}
\mathit{CHEATSYSTEM} = (\,&\mathit{WITH\_BADDIE} \underset{\{|\mathit{cast},\mathit{manipulate}|\}}{\|} \\
&(\mathit{STACK}(\mathit{STARTSTACK}) \underset{\{|\mathit{cast},\mathit{open},\mathit{manipulate}|\}}{\|} \mathit{COUNT}(\mathit{PLAYERS}))) \\
&\setminus \{|\mathit{discard}, \mathit{finished}|\}
\end{aligned}$$

The specification needs a little more care. We keep track of votes for and against—or, more precisely, we keep track of the number of participants yet to vote (*unv*) and the number of 'no' votes that must still come in to reach the threshold (*cred*)—and also of the dishonest player's intentions (on the *manipulate* channel). The specification states that the bottom envelope may contain either the result that the dishonest player did not want, or the correct result based on taking the dishonest player's intention as his vote. The only way that the 'incorrect' result is allowed, then, is if the dishonest player announces his intention, and then acts inconsistently with that intention, thus disenfranchising himself:

$$\begin{aligned}
\mathit{CHEATSPEC} &= \mathit{CHEATCOUNT}(\mathit{PLAYERS}, \mathit{THRESHOLD}) \\
&\quad \mathbin{|||} \mathit{CHAOS}(\{|\mathit{rip}, \mathit{swap}, \mathit{split}|\}) \\
\mathit{CHEATDONE}(0, \mathit{cred}, e) &= \mathit{cred} \leqslant 0 \,\&\, \mathit{open}.N \to \mathit{Stop} \\
&\quad \Box\ \mathit{cred} > 0 \,\&\, \mathit{open}.Y \to \mathit{Stop} \\
&\quad \Box\ \mathit{open}.e \to \mathit{Stop} \\
\mathit{CHEATDONE}(\mathit{unv}, \mathit{cred}, e) &= \mathit{unv} > 0 \,\& \\
&\quad (\mathit{cast}.\mathit{unv}.N \to \mathit{add}.N \to \mathit{CHEATDONE}(\mathit{unv}-1, \mathit{cred}-1, e) \\
&\quad \Box\ \mathit{cast}.\mathit{unv}.Y \to \mathit{add}.N \to \mathit{CHEATDONE}(\mathit{unv}-1, \mathit{cred}, e)) \\
\mathit{CHEATCOUNT}(\mathit{unv}, \mathit{cred}) &= \mathit{unv} > 0 \,\& \\
&\quad (\mathit{cast}.\mathit{unv}.N \to \mathit{add}.N \to \mathit{CHEATCOUNT}(\mathit{unv}-1, \mathit{cred}-1) \\
&\quad \Box\ \mathit{cast}.\mathit{unv}.Y \to \mathit{add}.N \to \mathit{CHEATCOUNT}(\mathit{unv}-1, \mathit{cred}) \\
&\quad \Box\ \mathit{manipulate}.\mathit{unv}.N \to \mathit{add}.N \to \mathit{CHEATDONE}(\mathit{unv}-1, \mathit{cred}-1, Y) \\
&\quad \Box\ \mathit{manipulate}.\mathit{unv}.Y \to \mathit{add}.N \to \mathit{CHEATDONE}(\mathit{unv}-1, \mathit{cred}, N))
\end{aligned}$$

When we ask FDR to check whether $\mathit{CHEATSPEC} \sqsubseteq_T \mathit{CHEATSYSTEM}$, we find that the refinement fails. This is as it should be: if the dishonest player is able to manipulate the stack at will under the table, then he cannot be stopped. For instance, he can take the last seat at the table, and force the motion to be turned down by placing his own 'NO' envelope at the bottom of the stack.

To strengthen the assumptions, we introduce the Velcro tabs discussed in Section 6.2.2. This is the purpose of the *rip* event: it occurs every time the stack is split. By adding one more control process, we can model the honest players' ability to hear the Velcro being separated. Each player must publicly attach his veto envelope to the top of the stack; he is then allowed one 'rip' under the table to produce a discard.

We introduce a process that ensures that there is only one *rip* after each *cast* or *manipulate* event:

$$\begin{aligned} VELCROCOUNT &= cast?\_ \to rip \to VELCROCOUNT \\ &\quad \Box\ manipulate?\_ \to rip \to VELCROCOUNT \end{aligned}$$

Now we check that the system meets the specification when it is put in parallel with this process:

$$CHEATSPEC \sqsubseteq_T CHEATSYSTEM \underset{\{|cast, manipulate, rip|\}}{\parallel} VELCROCOUNT$$

Running this through FDR demonstrates that the new system does indeed meet the specification.

7. Conclusion and future work

In this paper, we have presented several cryptographic protocols that can be performed using everyday objects and no electronic assistance. These protocols are suitable for situations in which the need for such a protocol arises with no advance planning, and no computational power available. They are intended for situations in which some security properties are desirable, but a high level of security is not deemed essential, because they rely on physical properties that are hard to guarantee absolutely (for instance, the participants' inability to substitute one envelope for another by sleight of hand).

We have also shown how to construct a formal proof of correctness of the two playing-card protocols, the envelope veto protocols and the Father Cryptmas protocol using CSP and FDR. Such formal proofs give confidence that the protocols provide the guarantees they are intended to provide.

It should be noted that the analysis in this paper does not model probabilities. It allows us to conclude that, for instance, in the Secret Santa setup, Alice learns nothing (non-trivial) about who is buying for whom in the limited sense that any observation she can make is consistent with any allocation, but this does not rule out the possibility that she learns probabilistic information from her observation. Similarly, although we have proven that, in the dating protocol, a vetoing participant does not discover whether the other participant chose to accept or veto, the formal analysis leaves open the (formal) possibility that the protocol leaks information about the relative probabilities of the other participant's decision. It seems clear at an informal level that the protocols are indeed information theoretically secure; but our formal analysis in this paper does not cover this. There are probabilistic variants of CSP [MMSS96], and it would be quite feasible to model these protocols using these variants; however, the tool support is currently rather limited and development appears to have stalled [GW05]. For the purposes of this paper, we have chosen to model the protocols using classical CSP to facilitate automated analysis with FDR; however, future work will involve adapting the models to include probabilities, with hand proofs of correctness.

It would be interesting to find ways of generalizing the results of Section 5 to include an arbitrary number of players. This seems difficult: our attempts to increase the number to four seem to give too many possible equivalence classes of states, and thus too much information leakage. There are possible ways round this that involve techniques to fuse some of the equivalence classes. One promising line is to use cards with arrows drawn on the underside: this gives a new primitive operation of reversing the arrow without reading it.

Extending the dining cryptographers protocols to cover vetoes is another interesting angle of enquiry. One method here is to ask each player to report $L - R$ if not vetoing, and a random $x \in \mathbb{Z}_6$ if wanting to veto. If at least one person vetoes, this will return a non-zero group result with probability $\frac{5}{6}$; if the group result is zero then this could have come from random submissions that cancel, so repeated runs are needed to give increasing assurance that the result really is zero. However, this does leak information: if a player vetoes by submitting x, and the group result is not x, then he knows that someone else also cast a veto.

The formal models in this paper are finite, and so the formal verification of properties gives guarantees only about the finite setups we have run through FDR. The cards protocols are designed for two or three participants, but the Secret Santa protocol and the threshold veto protocol could in principle be run with larger numbers; it is interesting to ask whether the finite models that we have verified here can be used to infer results about larger systems with an unbounded number of participants. This is not a particularly crucial practical question for these protocols, since there is a practical limit to the number of people who can sit round in a circle to slide pieces of card about for the Secret Santa problem, or who can sit round a table to construct a threshold veto using envelopes; and we have successfully verified Secret Santa setups with ten participants and veto setups with twelve. However, it is a useful theoretical question to ask.

Increasing the number of participants would be most naturally done by using hierarchical compression in

FDR, as exemplified in [RGG+95], where the technique is applied to the dining philosophers problem. The approach for our protocols would be to show that if we take two consecutive participants in the threshold voting scenario, for instance, and hide the events involved as the stack passes from one to the other, and if at most one of them casts a veto, then we end up with a process that is semantically equivalent to a single participant. An inductive argument then allows as many non-vetoing participants as necessary to be added without affecting the result. A similar line of reasoning could be used to deal with the Secret Santa protocol, though there is the added complexity that each participant has two successors: one successor in the original circle, and one successor in the gift-buying cycle. We will tackle this in future work.

One interesting piece of work that is tangentially related to the work presented in this paper is Schneier's *Solitaire* [Sch99b], a cryptosystem popularised in [Ste00] that uses a pack of cards to generate a keystream in the form of a pseudo-random sequence of alphabet characters. This can be used to encrypt (or decrypt) any message in the usual way: by adding (or subtracting) each element of the keystream and the message modulo 26. The shared key is the initial configuration of cards. The scheme is designed to be secure against cryptanalysis, though in fact some bias has been found in the pseudo random stream, where values repeat more often than they should by chance.

Acknowledgements

This work is a much expanded and improved version of a previous paper presented at the CryptoForma Workshop (Limerick, June 2011). We are grateful both to the anonymous referees on the CryptoForma paper, and to the anonymous referees on this version, for comments that led to major improvements in content and style.

The research presented here was conducted while the first-named author was a Royal Academy of Engineering/Leverhulme Trust Senior Research Fellow.

References

[Cha88] David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *J. Cryptology*, 1(1):65–75, 1988.

[ElG85] T. ElGamal. A public-key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31:469–472, 1985.

[For97] Formal Systems (Europe) Ltd. Failures-Divergence Refinement—FDR 2 user manual, 1997. Available from Formal Systems' web site at `http://www.formal.demon.co.uk/FDR2.html`.

[GW05] Michael Goldsmith and Paul Whittaker. A CSP Frontend for probabilistic tools. Technical report, The FORWARD project, June 2005. Available from `http://forward-project.org.uk/PDF_Files/D14.pdf`, last accessed 14 August 2012.

[KLN+06] Magdalena Kacprzak, Alessio Lomuscio, Artur Niewiadomski, Wojciech Penczek, Franco Raimondi, and Maciej Szreter. Comparing BDD and SAT Based Techniques for Model Checking Chaum's Dining Cryptographers Protocol. *Fundamentica Informaticae*, 72(1-3):215–234, April 2006.

[MMSS96] Carroll Morgan, Annabelle Mciver, Karen Seidel, and J. W. Sanders. Refinement-oriented probability for CSP. *Formal Aspects of Computing*, V8(6):617–647, November 1996.

[MRR10] S. Mauw, S. Radomirović, and P.Y. Ryan. Security protocols for Secret Santa. In *Proc. 18th Security Protocols Workshop*, Lecture Notes in Computer Science. Springer-Verlag, March 24-26 2010.

[RGG+95] A. W. Roscoe, Paul H. B. Gardiner, Michael Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood. Hierarchical Compression for Model-Checking CSP or How to Check 10^{20} Dining Philosophers for Deadlock. In Ed Brinksma, Rance Cleaveland, Kim Guldstrand Larsen, Tiziana Margaria, and Bernhard Steffen, editors, *TACAS*, volume 1019 of *Lecture Notes in Computer Science*, pages 133–152. Springer, 1995.

[Ros10] A.W. Roscoe. *Understanding Concurrent Systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1st edition, 2010.

[Sch99a] Steve A. Schneider. *Concurrent and Real-time Systems: The CSP approach*. Wiley, 1999.

[Sch99b] Bruce Schneier. The solitaire encryption algorithm. `http://www.schneier.com/solitaire.html`, last accessed 22 March 2011, 1999.

[Sin11] Simon Singh. Personal communication, March 2011.

[SS96] Steve Schneider and Abraham Sidiropoulos. CSP and anonymity. In Elisa Bertino, Helmut Kurth, Giancarlo Martella, and Emilio Montolivo, editors, *European Symposium on Research Into Computer Security (ESORICS) 96*, volume 1146 of *Lecture Notes in Computer Science*, pages 198–218. Springer Berlin / Heidelberg, 1996.

[Ste00] N. Stephenson. *Cryptonomicon*. Arrow Books, 2000.

[vdMS04] Ron van der Meyden and Kaile Su. Symbolic Model Checking the Knowledge of the Dining Cryptographers. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop (CSFW)*, June 2004.

A. Code for the dating and unanimity protocols

```
-- The maximum number of cards we'll be using
MAXLEN = 6

-- Some functional programming to allow us to construct
-- equivalence classes under rotation
inseq(x,<>) = false
inseq(x,<y>^ys) = if x==y then true else inseq(x,ys)

rotateseq(<>) = <>
rotateseq(<x>^xs) = xs^<x>

makerots(<xs>^xss) =
  let newxs = rotateseq(xs) within
    if inseq(newxs,<xs>^xss) then
      <xs>^xss
    else
      makerots(<newxs>^<xs>^xss)

-- Now allrots(<K,Q,Q,K,K>) gives the set of all sequences
-- that are equivalent to KQQKK under rotation
allrots(xs) = makerots(<xs>)

-- SEQS is the set of all seqs of K/Q up to max length
makeseqs(S,0) = S
makeseqs(S,n) = makeseqs({<K>^xs,<Q>^xs,xs | xs <- S},n-1)
SEQS = makeseqs({<>},MAXLEN)

datatype CARD = K | Q
PLAYERS = {0..MAXLEN}
PLACES = {0..MAXLEN}

channel announce: SEQS
channel rotate
channel place:PLACES.CARD
channel accept:PLAYERS
channel veto:PLAYERS

-- ANNOUNCE will non-deterministically choose something
-- from all sequences in the same equivalence class as xs,
-- and announce it
ANNOUNCE(xs) = ANNOUNCE_FROM(allrots(xs))
ANNOUNCE_FROM(xss) = |~| xs:set(xss) @ announce!xs -> STOP

COLLECTING(xs,0,cur) = rotate -> ANNOUNCE(xs)
COLLECTING(xs,remain,cur) =
        place.cur?X -> COLLECTING(xs^<X>,remain-1,cur+1)

-- We collect up the cards, and announce the result
COLLECT_CARDS_DATE = COLLECTING(<Q>,4,1)
COLLECT_CARDS_UNANIM = COLLECTING(<>,6,0)

-- A generic player accepts by a=K, b=Q or vetoes by a=Q, b=K
-- We need to make sure that the cards are placed in the
-- right order for collection
GENERIC_PLAYER_ORDERED(n,a,b,<x,y>) =
             accept.n -> place.a.x -> place.b.y -> STOP
          [] veto.n   -> place.a.y -> place.b.x -> STOP
GENERIC_PLAYER(n,a,b) =
        if a<b then
          GENERIC_PLAYER_ORDERED(n,a,b,<K,Q>)
        else
          GENERIC_PLAYER_ORDERED(n,b,a,<Q,K>)

-- The players in the dating protocol
P1_DATE = GENERIC_PLAYER(1,2,1)
```

```
P2_DATE = GENERIC_PLAYER(2,3,4)

-- The overall system running the dating protocol
DATE_SYSTEM = (P1_DATE ||| P2_DATE)
                   [|{|place.x | x <- {1..4}|}|]
              COLLECT_CARDS_DATE

-- CONTROL(n,x) gives a way of forcing Pn to accept or veto
-- CONTROLS(xs) determines the votes for all players
CONTROL(n,0) = accept.n -> STOP
CONTROL(n,1) = veto.n -> STOP
CONTROL_BUILD(_,<>) = STOP
CONTROL_BUILD(n,<x>^xs) =
    CONTROL(n,x) ||| CONTROL_BUILD(n+1,xs)
CONTROLS(xs) = CONTROL_BUILD(1,xs)

-- This is P1's view of the dating protocol
P2_EVENTS = {|accept.2,veto.2,place.3,place.4|}
P1_DATE_VIEW(choices) =
    (DATE_SYSTEM [|{|accept,veto|}|] CONTROLS(choices))
        \ P2_EVENTS

-- P1's view, if he vetoes, should be the same
-- regardless of what P2 does
-- These checks succeed
assert P1_DATE_VIEW(<1,0>) [T= P1_DATE_VIEW(<1,1>)
assert P1_DATE_VIEW(<1,1>) [T= P1_DATE_VIEW(<1,0>)

-- On the other hand, if he accepts, he can tell
-- the difference
-- These checks fail
assert P1_DATE_VIEW(<0,0>) [T= P1_DATE_VIEW(<0,1>)
assert P1_DATE_VIEW(<0,1>) [T= P1_DATE_VIEW(<0,0>)

-- The players in the three-party unanimity protocol
P1_UNANIM = GENERIC_PLAYER(1,0,3)
P2_UNANIM = GENERIC_PLAYER(2,2,5)
P3_UNANIM = GENERIC_PLAYER(3,4,1)

-- The overall system running the unanimity protocol
UNANIM_SYSTEM = (P1_UNANIM ||| P2_UNANIM ||| P3_UNANIM)
                    [|{|place|}|]
                COLLECT_CARDS_UNANIM

-- We'll analyze this from the view of an external observer
-- Only the rotation and result are visible
EXT_UNANIM_VIEW(choices) =
    (UNANIM_SYSTEM [|{|accept,veto|}|] CONTROLS(choices))
        \ {|accept,veto,place|}

-- We want to say that unanimity looks the same,
-- and everything non-unanimous looks the same.
-- There are two unanimous possibilities
-- These checks succeed
assert EXT_UNANIM_VIEW(<0,0,0>) [T= EXT_UNANIM_VIEW(<1,1,1>)
assert EXT_UNANIM_VIEW(<1,1,1>) [T= EXT_UNANIM_VIEW(<0,0,0>)

-- The six non-unanimous possibilities can be put in a
-- refinement cycle
-- These checks show that all the processes are equivalent
-- These checks succeed
assert EXT_UNANIM_VIEW(<0,0,1>) [T= EXT_UNANIM_VIEW(<0,1,0>)
assert EXT_UNANIM_VIEW(<0,1,0>) [T= EXT_UNANIM_VIEW(<0,1,1>)
assert EXT_UNANIM_VIEW(<0,1,1>) [T= EXT_UNANIM_VIEW(<1,0,0>)
assert EXT_UNANIM_VIEW(<1,0,0>) [T= EXT_UNANIM_VIEW(<1,0,1>)
assert EXT_UNANIM_VIEW(<1,0,1>) [T= EXT_UNANIM_VIEW(<1,1,0>)
assert EXT_UNANIM_VIEW(<1,1,0>) [T= EXT_UNANIM_VIEW(<0,0,1>)
```

```
-- Finally, the following assertion shows that the two
-- equivalence classes are different
-- Unanimity and non-unanimity look different
-- These two checks fail
assert EXT_UNANIM_VIEW(<0,0,0>) [T= EXT_UNANIM_VIEW(<0,0,1>)
assert EXT_UNANIM_VIEW(<0,0,1>) [T= EXT_UNANIM_VIEW(<0,0,0>)
```

B. Code for the Father Cryptmas protocol

```
-- We'll have five people in our Secret Santa. The X is there
-- to denote a masked identity: we know someone's name is there
-- but we don't know whose.

datatype Ident = Alice | Bob | Charlie | Dawn | Erica | X
--

-- The order in which they'll sit around the table

PEOPLESEQ = <Alice, Bob, Charlie, Dawn, Erica>

-- and some constants to give us a handle on who is playing

PEOPLE = set(PEOPLESEQ)
SIZE = #PEOPLESEQ
NUMS = {0..SIZE-1}

-- Some standard functions that we will need

nameOf(i) = nth(i,PEOPLESEQ)
nth(0,<x>^xs) = x
nth(n,<x>^xs) = nth(n-1,xs)

nxt(i) = (i+1)%SIZE
prev(i) = (i-1)%SIZE

channel offer,lay,return,receive:NUMS.Ident.Ident
channel pass:NUMS.NUMS.Ident
channel gift:Ident.Ident
channel sync

-- Alice (etc.) will 'offer' an envelope with her name written
-- on the outside and her name written on the card inside. She
-- will then see a random envelope 'laid' on the table in front
-- of her. When everyone has got an envelope (the 'sync' event)
-- then she will 'pass' the piece of card to the next person at
-- the table; someone else will also 'pass' her a card. These
-- pieces of card go into their new envelopes. Alice will then
-- 'return' the envelope she was given, with its new piece of
-- card. She'll finally 'receive' the envelope that has her
-- name on the outside, and she'll open it, read the name, and
-- buy a 'gift' for the person named on the card.

PERSON(i) = offer.i.nameOf(i).nameOf(i) -> sync
          -> lay.i?n1?n2 -> sync
          -> PERSON_SWAP(i,n1,n2,nxt(i),prev(i))
PERSON_SWAP(i,n1,n2,to,frm) = i<frm & pass.i.to.n2
                    -> pass.frm.i?n3 -> PERSON_END(i,n1,n3)
          [] frm<i & pass.frm.i?n3 -> pass.i.to.n2
                    -> PERSON_END(i,n1,n3)
PERSON_END(i,n1,n3) = sync -> return.i.n1.n3 -> sync
          -> receive.i.nameOf(i)?n4 -> sync
          -> gift.nameOf(i).n4 -> STOP
AlphaPerson(i) = {|offer.i,lay.i,pass.i.nxt(i),pass.prev(i).i,
                        return.i,receive.i,gift.nameOf(i),sync|}

-- A process to shuffle the envelopes. Each envelope gets taken
-- in, and then laid at any free space on the table. The sync
-- events ensure that all the envelopes are taken in before
-- anything is laid out again.

SHUFFLE_ME(i) = offer.i?n1?n2 -> lay?j!n1!n2 -> STOP
COUNT_SEQ(c,0) = STOP
COUNT_SEQ(c,n) = c.(SIZE-n)?_ -> COUNT_SEQ(c,n-1)

SHUFFLING = (||| i:NUMS @ SHUFFLE_ME(i))
                  [|{|offer,lay|}|]
                    (COUNT_SEQ(offer,SIZE) ||| COUNT_SEQ(lay,SIZE))

-- The envelopes need shuffling a second time, after the cards
-- have been moved round. That shuffle looks exactly the same,
-- except that it uses different events.

RESHUFFLING = SHUFFLING[[offer <- return, lay <- receive]]

-- The basic system is the synchronised participants in parallel
-- with the two shuffling processes, plus something to constrain
-- the order in which the cards are passed round, and something
-- similar for the order of gift announcements (to keep the state
-- space under control).

ORDER_GIFTS(0) = STOP
ORDER_GIFTS(n) = gift.nth(SIZE-n,PEOPLESEQ)?_ -> ORDER_GIFTS(n-1)

SYSTEM = ((|| i:NUMS @ [AlphaPerson(i)] PERSON(i))\{sync})
               [|{|offer,lay,return,receive,pass,gift|}|]
          (SHUFFLING ||| RESHUFFLING
                     ||| COUNT_SEQ(pass,SIZE) ||| ORDER_GIFTS(SIZE))

Sigma = Union({AlphaPerson(i) | i <- NUMS})
Non_Alice_Events = diff(Sigma,AlphaPerson(0))

-- This sanity check says that no one should end up buying
-- him or herself a gift. It should (and does) succeed.

-- Specify the events that shouldn't ever happen.

REFLEXIVE = {gift.x.x | x <- PEOPLE}

-- If we remove everything else from SYSTEM, we have nothing left.

assert STOP [T= SYSTEM \ diff(Sigma,REFLEXIVE)

-- We need to specify properties by looking at who gives gifts
-- to whom. The FORCE_GIFT process allows us to specify some
-- particular arrangements: it takes two names, n1 and n2, and
-- ensures that n1 will end up buying a gift for n2. By putting
-- several such processes in parallel, we can create more
-- complex arrangements.

FORCE_GIFT(n1,n2) = FORCE_GIFT_OPEN(n1,n2)
FORCE_GIFT_OPEN(n1,n2) = ([] n:diff(PEOPLE,{n1,n2}) @
                                lay?i!n!n -> FORCE_GIFT_OPEN(n1,n2))
                       [] lay?i!n2!n2 -> FORCE_GIFT_NODE(n1,n2,i)
                       [] lay?i!n1!n1 -> FORCE_GIFT_NODE(n1,n2,prev(i))
FORCE_GIFT_NODE(n1,n2,i) = ([] n:diff(PEOPLE,{n1,n2}) @
                                lay?j:diff(NUMS,{i,nxt(i)})!n!n
                                -> FORCE_GIFT_NODE(n1,n2,i))
                       [] lay.i.n2.n2 -> FORCE_GIFT_NODE(n1,n2,i)
                       [] lay.nxt(i).n1.n1 -> FORCE_GIFT_NODE(n1,n2,i)

FORCE_GIFTS(<<x,y>>) = FORCE_GIFT(x,y)
FORCE_GIFTS(<<x,y>>^xs) = FORCE_GIFT(x,y) [|{|lay|}|] FORCE_GIFTS(xs)
```

```
-- SYSTEM_WITH now allows us to constrain the system to meet a
-- particular arrangement. It takes a sequence of pairs, where
-- the presence of <x,y> in the sequence of pairs means that
-- x gives a gift to y.

SYSTEM_WITH(Pairs) = SYSTEM
                       [|{|lay|}|]
                     FORCE_GIFTS(Pairs)

-- SYSTEM_ALICE_WITH allows us to constrain the system, but also
-- considers things from Alice's perspective. We hide all events
-- she can't see; we also mask all identites that are hidden
-- from her (because the envelope is face down, or the card is
-- face down or inside the envelope).

SYSTEM_ALICE_WITH(Pairs) = (SYSTEM_WITH(Pairs)\Non_Alice_Events)
         [[lay.i.n1.n2 <- lay.i.X.X,
           pass.i.j.n1 <- pass.i.j.X,
           return.i.n1.n2 <- return.i.X.X
         | i <- NUMS, j <- NUMS, n1 <- Ident, n2 <- Ident]]

-- These are the critical checks. They assume the honest-but-
-- curious model: Alice obeys the protocol, but we want to know
-- if she can infer anything secret from what she sees. The
-- first says that all paths that lead to Alice's giving a gift
-- to Bob are consistent with Charlie's giving a gift to Dawn.
-- Since they are chosen arbitrarily, if Alice were able to
-- infer anything about others' gifts, there would be a path
-- that would enable her to infer that Charlie had not bought a
-- gift for Dawn. So this check should (and does) succeed.

assert SYSTEM_ALICE_WITH(<<Charlie,Dawn>,<Alice,Bob>>)
          [T=
       SYSTEM_ALICE_WITH(<<Alice,Bob>>)

-- The refinement also holds the other way round, giving us
-- equality between the processes (in the traces model).

assert SYSTEM_ALICE_WITH(<<Alice,Bob>>)
          [T=
       SYSTEM_ALICE_WITH(<<Charlie,Dawn>,<Alice,Bob>>)

-- The next check says that if Alice gives to Bob then she will
-- not be able to distinguish between a case where Dawn gives
-- her a gift and a case where Charlie gives her a gift. This
-- also should (and does) succeed.

assert SYSTEM_ALICE_WITH(<<Charlie,Alice>,<Alice,Bob>>)
          [T=
       SYSTEM_ALICE_WITH(<<Dawn,Alice>,<Alice,Bob>>)

-- Of course, this holds in the other direction too.

assert SYSTEM_ALICE_WITH(<<Dawn,Alice>,<Alice,Bob>>)
          [T=
       SYSTEM_ALICE_WITH(<<Charlie,Alice>,<Alice,Bob>>)

-- Finally, we assert that, with no restrictions on Alice's
-- recipient, she should not be able to distinguish between a
-- case where Dawn gives her a gift and a case where Charlie
-- gives her a gift. This quite rightly fails: if Alice is
-- buying for Charlie, then Charlie cannot be buying for her.

assert SYSTEM_ALICE_WITH(<<Charlie,Alice>>)
          [T=
       SYSTEM_ALICE_WITH(<<Dawn,Alice>>)
```

C. Code for the envelope veto protocols

```
-- This file models the envelope protocols.

-- Since the first protocol is equivalent to the second one
-- with a threshold of 1, we give just one model with a
-- THRESHOLD parameter (k in the paper) that can be varied.

-- constants for the number of participants and the veto
-- threshold (motion fails if THRESHOLD or more vetoes)

PLAYERS = 7
THRESHOLD = 3

-- a datatype for votes: 'Y' (accept) or 'N' (reject)

datatype ENV = Y | N

-- We'll have cast.Y to say yes, and cast.N to say no
-- A dishonest agent might 'split' the stack to manipulate it.

channel open,add,discard:ENV
channel cast,manipulate:{1..PLAYERS}.ENV
channel split:{1..THRESHOLD+1}.{1..THRESHOLD}
channel swap:{1..THRESHOLD+1}.{1..THRESHOLD+1}
channel finished,rip

-- The pile at the start is a sequence of 'YES' envelopes

STARTSTACK = <Y | i <- <1..THRESHOLD>>

-- The pile is a sequence of envelopes acting as a FIFO queue
-- We add to the top, and then either remove from the top
-- (null op, to vote Y) or remove from the bottom (to vote N)
-- The end of the sequence represents the top of the queue

-- The manipulate event is there for a dishonest player to use
-- to change the stack by splitting and swapping

-- Honest and dishonest players must finish by producing a
-- single discard envelope and a stack of THRESHOLD envelopes

-- The 'rip' event occurs whenever anyone splits the stack
-- We'll use this later to model the Velcro tabs

STACK(<x>^xs) =
          cast?p.Y -> add.N -> rip -> STACK_DISCARD(<N>,<x>^xs)
       [] cast?p.N -> add.N -> rip -> STACK_DISCARD(<x>,xs^<N>)
       [] manipulate?p?v -> add.N -> SPLIT_STACK(<<x>^xs^<N>>)
       [] open!x -> STOP
STACK_DISCARD(<x>,xs) = discard!x -> STACK(xs)

SPLIT_STACK(xss) =
    #last(xss)==1 &
          finished -> STACK_DISCARD(last(xss),concat(init(xss)))
     []
    ([]i:{1..#xss} @ ([]j:{1..#nth(xss,i)-1} @
       split.i.j -> rip -> SPLIT_STACK(split_at(xss,i,j))))
                []
    ([]i:{1..#xss-1} @ ([]j:{i+1..#xss} @
       swap.i.j -> SPLIT_STACK(swapped(xss,i,j))))

-- Some standard functions

nth(<x>^xs,1) = x
nth(<x>^xs,n) = nth(xs,n-1)
```

```
last(<x>) = x
last(<x>^xs) = last(xs)

init(<x>) = <>
init(<x>^<y>^xs) = <x>^init(<y>^xs)

-- Some helper functions to split sequences
-- split_at splits the ith sequence at the jth point
-- split_seq splits a single sequence

split_at(<xs>^xss,1,j) = split_seq(<>,xs,j)^xss
split_at(<xs>^xss,n,j) = <xs>^split_at(xss,n-1,j)
split_seq(ys,<x>^xs,1) = <ys^<x>,xs>
split_seq(ys,<x>^xs,n) = split_seq(ys^<x>,xs,n-1)

-- swapped(xss,i,j) is sequence xss but with the ith and jth
-- entries swapped over

remap(k,i,j) = if k==i then j else if k==j then i else k
swapped(xss,i,j) = <nth(xss,remap(k,i,j)) | k <- <1..#xss>>

-- A player chooses whether to accept, and casts accordingly.
-- The players are numbered, and play in decreasing order.

PLAYER(n) = |~| e:ENV @ cast.n.e -> STOP

-- We need to check whether everyone has voted, and not open
-- the envelope yet if not. The players play in order, from
-- highest number to lowest. A dishonest player will
-- attempt to manipulate the vote instead of casting a normal
-- vote.

COUNT(0) = open?x -> STOP
COUNT(n) = cast.n?e -> COUNT(n-1)
        [] manipulate.n?e -> COUNT(n-1)

-- The system is then some players, an initial stack, and
-- a counter.

-- We hide the adding and discarding; the cast events are left
-- because the specification needs to check how many rejections
-- there were. The dishonest events are left in for now, but
-- we will abstract them away in the specification.

SYSTEM = ((||| n:{1..PLAYERS} @ PLAYER(n))
              [|{|cast,manipulate|}|]
       (STACK(STARTSTACK) [|{|cast,open|}|] COUNT(PLAYERS)))
                \ {|discard,manipulate,finished|}

-- This spec says that:
--   if not enough players have voted then keep voting
--   otherwise, if k or more have rejected then we should be
--   able to open a 'NO', else not.
-- 'unvoted' is the number of players still to vote
-- 'credit' is how many more players will need to reject for
-- the motion to fail

-- The effect of the interleaving with CHAOS is to abstract away
-- the rip, swap and split events: they can occur at any time,
-- and be blocked at any time.

SPEC = SPECCOUNT(PLAYERS,THRESHOLD) ||| CHAOS({|rip,swap,split|})
SPECCOUNT(0,credit) = credit<=0 & open.N -> STOP
                   [] credit>0 & open.Y -> STOP
SPECCOUNT(unvoted,credit) =
  unvoted>0 & (cast.unvoted.N -> add.N -> SPECCOUNT(unvoted-1,credit-1)
           |~| cast.unvoted.Y -> add.N -> SPECCOUNT(unvoted-1,credit))
```

```
-- FDR confirms that the system meets the specification.

assert SPEC [T= SYSTEM

-- The WITH_BADDIE process represents all the players, one of
-- whom is dishonest. Effectively, this represents a dishonest
-- player who starts by choosing where he wants to sit at the
-- table.

WITH_BADDIE = |~| i:{1..PLAYERS} @
      (BADDIE(i) ||| (||| n:diff({1..PLAYERS},{i}) @ PLAYER(n)))

-- A dishonest player will try to manipulate the vote to come
-- out at a particular result (so v=Y or v=N). The stack will then
-- add a 'NO' envelope (since this is done in public); thereafter,
-- he might engage in any number of splits and swaps. The stack
-- process will not allow him to finish unless the stack ends up
-- with the first part being a singleton (for discard).

BADDIE(i) = manipulate.i?v -> CHEATING(i)
CHEATING(i) = swap?x?y -> CHEATING(i)
           [] split?x?y -> rip -> CHEATING(i)
           [] finished -> discard?x -> STOP

-- The cheating system is then as before, but with the dishonest
-- player involved.

CHEATSYSTEM = (WITH_BADDIE [|{|cast,manipulate|}|]
    (STACK(STARTSTACK)
                [|{|cast,open,manipulate|}|]
                                       COUNT(PLAYERS)))
                           \ {|discard,finished|}

-- The specification is now more complicated. We want to know
-- whether the dishonest player has successfully manipulated
-- it in his favour; so we keep track of votes for and votes
-- against, but also what the dishonest player is trying to
-- achieve. If the envelope turns out to defeat his plans then
-- that's OK; the problem comes only if we get the wrong
-- result, and it's the result he asked for.

CHEATSPEC = CHEATSPECCOUNT(PLAYERS,THRESHOLD)
                                   ||| CHAOS({|rip,swap,split|})
CHEATSPECDONE(0,credit,e) = credit<=0 & open.N -> STOP
                          [] credit>0 & open.Y -> STOP
                          [] open.e -> STOP
CHEATSPECDONE(unvoted,credit,e) = unvoted>0 &
          (cast.unvoted.N -> add.N -> CHEATSPECDONE(unvoted-1,credit-1,e)
        [] cast.unvoted.Y -> add.N -> CHEATSPECDONE(unvoted-1,credit,e))
CHEATSPECCOUNT(unvoted,credit) = unvoted>0 &
    (cast.unvoted.N -> add.N -> CHEATSPECCOUNT(unvoted-1,credit-1)
  [] cast.unvoted.Y -> add.N -> CHEATSPECCOUNT(unvoted-1,credit)
  [] manipulate.unvoted.N -> add.N -> CHEATSPECDONE(unvoted-1,credit-1,Y)
  [] manipulate.unvoted.Y -> add.N -> CHEATSPECDONE(unvoted-1,credit,N))

-- As things stand, we are assuming that no one can tell what
-- the dishonest player does with the envelopes under the table
-- (except that he cannot see their contents).

-- The system does not meet this specification. The dishonest
-- player can manipulate the envelopes under the table to his
-- heart's content.

assert CHEATSPEC [T= CHEATSYSTEM

-- Now we add some Velcro tabs. We do so by allowing the other
-- players to hear 'rip' events every time the stack is split.
-- This control process ensures that no player can produce more
-- than one 'rip' event. This prevents the dishonest player
-- from making alterations to the stack that would change the
-- result.

VELCROCOUNT = cast?_ -> rip -> VELCROCOUNT
           [] manipulate?_ -> rip -> VELCROCOUNT

-- FDR confirms that the new system, with the Velcro in place,
-- meets the new specification.

assert CHEATSPEC
          [T=
       CHEATSYSTEM [|{|cast,manipulate,rip|}|] VELCROCOUNT
```