

Architecting Security with Paradigm

S. Andova¹, L.P.J. Groenewegen², J.H.S. Verschuren², and E.P. de Vink¹

¹ Department of Mathematics and Computer Science
Technische Universiteit Eindhoven, The Netherlands

² LIACS, Leiden University, The Netherlands

Abstract. For large security systems a clear separation of concerns is achieved through architecting. Particularly the dynamic consistency between the architectural components should be addressed, in addition to individual component behaviour. In this paper, relevant dynamic consistency is specified through Paradigm, a coordination modeling language based on dynamic constraints. As it is argued, this fits well with security issues. A smaller example introduces the architectural approach towards implementing security policies. A larger casestudy illustrates the use of Paradigm in analyzing the FOO voting scheme. In addition, translating the Paradigm models into process algebra brings model checking within reach. Security properties of the examples discussed, are formally verified with the model checker `mCRL2`.

1 Introduction

Characteristic for software security problems is, all details matter [2]. Such details fall into several categories centered round the software that must be secure. First: computational details, purely internal to a single component of the software. Second: interaction details between the various components of the software. Third: interaction details between the software and relevant other application software. Fourth: interaction details between the software and the lower level inner world of machine software and hardware. Fifth: interaction details between the software and the outer world of its human stakeholders. Finally, in a recurrent manner: sets of these categories can be found again, centered round relevant other pieces of software, hardware or stakeholders.

In general software development, it has become standard to integrate such diverse categories of computation and interaction coherently within one model, at least at a global level. To that aim, architecture description languages and architectural frameworks are used, cf. [16, 24, 28], comprising not only the software application level and technical infrastructure level, but also the organization level constituting the habitat of the software. An architectural model in such a language succeeds in giving a clear and coherent overview of the problem situation at hand, but its interaction details remain declarative mainly, as e.g. service-oriented architectures do not usually go into the details of an orchestration or choreography of a service.

In the context of security, where all details matter, an architectural approach might be seriously hampering insight into the quality of the solution. By its

global nature, an architectural description does not readily express every detail of the security problem situation. And even worse, architectures are weak in clarifying operationally how interaction occurs and what behavioural consequences may arise. This is not amazing as even in detailed UML models for software design, dynamic consistency is a problem far from being solved within the UML language [27]. However, interaction categories dominate the listing given above. So, behavioural interaction details are of the greatest importance.

Very often, coordination languages are successfully used for interaction issues, also in relation to architectures. In the context of security it is argued however, coordination solutions are not so easy to apply. Where security is generally oriented towards restriction of dynamics by effectively prohibiting and preventing unwelcome behaviour of participants and intruders, coordination is rather more oriented towards broadening dynamics by efficiently establishing a larger behavioural scope through collaboration, see [44].

For security problems of a larger size, the above disadvantages of architectures or coordination languages, are even more prominent: more details that matter, more coordination directed to even more restrictive dynamics. Nevertheless, we propose the coordination modeling language Paradigm (see e.g. [19, 18, 4]) both for architecting and for coordinating solutions to larger security problems. Paradigm's architecting is done by splitting the problem situation into well-chosen collaborations, characterized each through a separate protocol. Protocol dynamics, although global, are kept dynamically consistent with detailed dynamics of collaborating participants, through well-defined roles. Paradigm's specification of coordination solutions is in terms of consistency rules forming protocols, typically formulated as constraint orchestration or as constraint choreography. Thus, additional effective restriction-on-purpose of collaboration protocol dynamics towards a solution for security issues, fits well with Paradigm's usual orchestration or choreography of constraints.

To underpin the above claims, the sequel is organized as follows. In Section 2 we introduce Paradigm by means of a small secure email example, with a light architectural flavor already present. Section 3 presents Paradigm's constraint architecting by using a larger e-voting example, the FOO voting scheme. Here it is not only specified how one individual voter is to be handled, but also how potential voters are to be hoarded as an ensemble. Section 4 addresses formal verification of the Paradigm models, through model checking on the basis of a process algebraic translation. Related work is discussed in Section 5. Section 6 gives conclusions.

2 An email example: Paradigm explained

In this section we briefly explain the key notions of the coordination modeling language Paradigm. A simple case dealing with security policies for encryption of email messages is used as a running example. The exposition should provide sufficient understanding of Paradigm for the subsequent sections, in particular for Section 3, where we discuss a more extensive voting scheme. For more detailed introductions to Paradigm we refer to [17, 19, 43].

Let us consider the following situation. In the R&D laboratory of a company, confidential research is taking place. A document security policy applies to email communication. It states that email addressed to colleague researchers may be signed and encrypted, dependent on the security label of the email content or attachments. Additionally, mail directed to recipients outside the lab are mandatorily encrypted. To support the cryptographic algorithms used, a public key infrastructure has been set up.

Some workers use a PDA for email communication. For this, a location-dependent security policy is in place, demanding all email traffic to be encrypted, when sent using the PDA outside of the premises of the laboratory. Antennas at the exits of the lab send a signal to the PDA, caught by the security module on the PDA, upon entering or leaving the grounds. The email client of the PDA will automatically encrypt all messages when being outside, and provides optional encryption when being inside. However, encryption in particular, substantially consumes battery power. Therefore, as an exception to the rule, for email of a low security label, the PDA owner may override the obligation to encrypt when being outside. Upon completion of sending an email, the PDA switches back to the default mode of encryption, optional or mandatory encryption when inside, mandatory when outside, whichever applies.

In view of modeling coordination solutions in terms of behavior influencing, Paradigm has five key notions: STD, phase, (connecting) trap, role and consistency rule. We shall introduce them guided by the R&D lab example.

Every Paradigm model is built from STDs, purely sequential behavioral units.

- A *state-transition diagram (STD)* is a triple $\langle \text{ST}, \text{AC}, \text{TS} \rangle$ with ST the set of states, AC the set of actions and $\text{TS} \subseteq \text{ST} \times \text{AC} \times \text{ST}$ the set of transitions or steps. A step $(x, a, x') \in \text{TS}$ is said to be a step from x to x' , notation $x \xrightarrow{a} x'$.

Thus, an STD is just a labeled transition system (LTS), rather like a very simple, purely sequential state machine in UML.

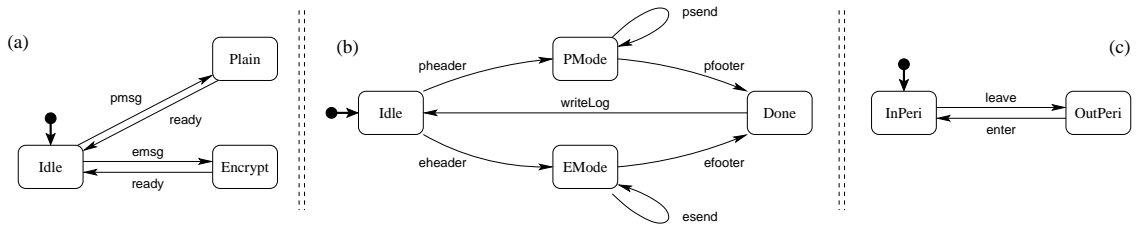


Fig. 1. Basic STDs: (a) PDA, (b) email client EMC, (c) security module SM.

Modeling the above R&D lab encryption aspects with Paradigm entails, three components are being distinguished. Their dynamics are modeled through one STD each: a PDA, an email client on the PDA, a security module on the PDA. See Figure 1abc visualizing the three respective STDs in UML-style: rounded rectangles as states and a black-dot-and-arrow pointing to a starting state. The

user of the PDA can send a message either in plain mode by moving to state `Plain` or in encryption mode by moving to state `Encrypt`. After the message has been sent, PDA returns to the starting state `Idle`. (Below we will refine this.) The email client EMC, when asked to send a message, splits the message into blocks and transmits them with additional header and footer. It does so, either in plain mode, state `PMode`, or in encryption mode, state `EMode`. After arrival in state `Done`, the email client returns to its state `Idle`. The security module SM shuttles between the two states `InPeri` and `OutPeri`, registering whether the PDA is inside or outside the security perimeter.

The main coordination modeling issue is, to organize the mutual influencing of the components such that the security policies are respected. For example, while PDA resides in state `Encrypt`, the email client EMC should remain restricted to taking steps where sending of the header, of the separate blocks and of the footer occurs in encrypted mode only. To that aim, Paradigm provides the notions of a *phase* of an STD and of a *trap* of a phase, both notions serving as temporary constraint on the STD's dynamics, i.e. on the choice there is for taking steps.

- A *phase* of STD $\langle \text{ST}, \text{AC}, \text{TS} \rangle$ is an STD $S = \langle \text{st}, \text{ac}, \text{ts} \rangle$ such that $\text{st} \subseteq \text{ST}$, $\text{ac} \subseteq \text{AC}$ and $\text{ts} \subseteq \{ (x, a, x') \in \text{TS} \mid x, x' \in \text{st}, a \in \text{ac} \}$.

A phase of an STD is itself an STD, actually a subSTD of the STD it is a phase of. As such, a phase of an STD is meant to express a temporarily valid dynamic constraint *imposed* on the STD it is a phase of. Visualized, a phase is an STD-like fragment of the original, larger STD preserving the form of the original in the fragment. See Figure 2abc. Apart from the extra rectangles to be discussed below, each figure part represents one phase of the full EMC in Figure 1b, viz. `PSend`, `ESend` and `Finished`. `PSend` gives the behavior needed for plain sending; `ESend` singles out the behavior needed for encrypted sending; `Finished` represent the behavior needed for getting prepared for whatever next sending after having closed the last sending properly.

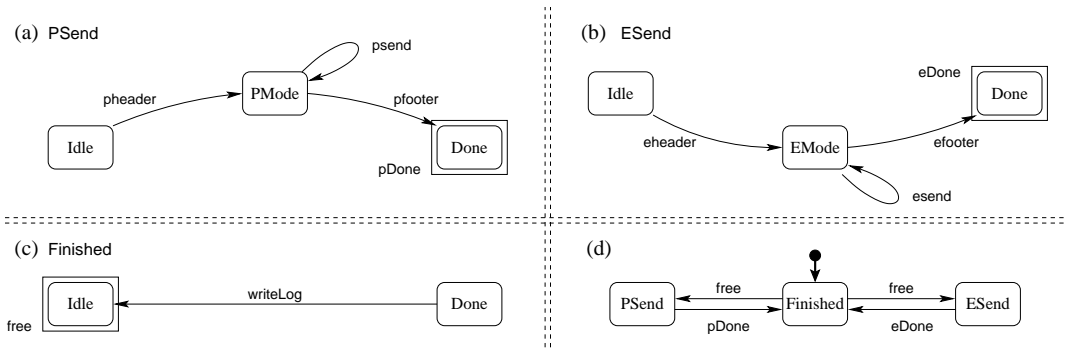


Fig. 2. (a–c) phases `PSend`, `ESend` and `Finished`, (d) role `EMC(DoPo)`.

Traps, the other dynamic constraint notion of Paradigm, are stepping stones for switching from one phase to another.

- A *trap* t of phase $S = \langle \mathbf{st}, \mathbf{ac}, \mathbf{ts} \rangle$ is a non-empty set of states $t \subseteq \mathbf{st}$ such that $x \in t$ and $x \xrightarrow{a} x' \in \mathbf{ts}$ imply $x' \in t$. The trap t of S *connects* phase S to another phase $S' = \langle \mathbf{st}', \mathbf{ac}', \mathbf{ts}' \rangle$, notation $S \xrightarrow{t} S'$, if $t \subseteq \mathbf{st}'$. This is called a *phase transfer*. If $t = \mathbf{st}$, t is called the *trivial* trap of S .

A trap of a phase is a subset of the states of the phase, such that once entered, the subset cannot be left as long as the phase remains imposed. A trap represents a second type of dynamic constraints, *committed to* by a phase, through its own dynamics: within a phase, the entering of a trap is irrevocable, thus marking the beginning of a final stage of the phase. A trap often serves as a guard for a phase change of the basic STD, i.e. as a guard for changing the constraint currently imposed into a constraint imposed next. In such a case, the trap has to be *connecting* to the phase aimed at next.

A trap is visualized as a polygon surrounding the states belonging to the trap. More concretely, the three small rectangles in each of the diagrams in Figure 2a–c represent a trap of the particular phase, named `pDone`, `eDone` and `free`, respectively. Below we shall see examples of larger (connecting) traps. Normally, as is the case here, trivial traps are not drawn, unless serving as connecting trap. Note, trap `pDone` is connecting from `PSend` to `Finished`, trap `eDone` is connecting from `ESend` to again `Finished` and trap `free` is connecting from `Finished` to `PSend` as well as to `ESend`. Hence, phase transfers $\text{PSend} \xrightarrow{\text{pDone}} \text{Finished}$, $\text{ESend} \xrightarrow{\text{eDone}} \text{Finished}$, $\text{Finished} \xrightarrow{\text{free}} \text{PSend}$ and $\text{Finished} \xrightarrow{\text{free}} \text{ESend}$ are well-defined.

From phase transfers a concrete role STD can be constructed. Roles are generally defined in terms of phases and of traps thereof, belonging each to a well-chosen set, referred to as a partition.

- A *partition* $\pi = \{ (S_i, T_i) \mid i \in I \}$ of an STD Z is a set of phases S_i of Z and a set T_i of traps of S_i . The *role* or *global STD* at the level of partition π is an STD $Z(\pi) = \langle \mathbf{GST}, \mathbf{GAC}, \mathbf{GTS} \rangle$ with $\mathbf{GST} \subseteq \{ S_i \mid i \in I \}$, $\mathbf{GAC} \subseteq \bigcup_{i \in I} T_i$ and $\mathbf{GTS} \subseteq \{ S_i \xrightarrow{t} S_j \mid i, j \in I, t \in \mathbf{GAC} \}$ a set of phase transfers. Z is called the *detailed* STD underlying *global* STD $Z(\pi)$, the π -role of Z .

Thus, a role of an STD is based on a partition, a particular set of phases of the STD and of connecting traps between them. Here, a connecting trap marks the readiness of a phase to be changed into another phase within the role. The role is to provide a consistent and global view on the ongoing detailed dynamics of the underlying STD. If phases and traps have been chosen well, such a global view expresses precisely the dynamics essential for coordinating the underlying STD via its role. On the one hand, the current state of the role STD, being a phase, imposes the constraint being relevant for the coordination at that moment, on the underlying detailed dynamics. On the other hand, the current detailed state belonging to a trap, is a commit towards the ongoing coordination: the detailed STD shall stay within the trap until a next phase is imposed. In this manner, a role remains dynamically consistent with the underlying detailed STD.

More concretely, Figure 2d presents the role EMC(DoPo) for implementing the coordination consequences of the document security policy for the email client EMC. The role in Figure 2d has the three phases PSend, ESend and Finished as its states and it has the three connecting traps pDone, eDone and free as its actions. Thus, partition DoPo consists of the phases and traps from Figure 2a–c, together with the three trivial traps (not drawn).

The key idea is, at each point in time a component not only is in one of the states of its detailed STD, but for every role the component has, at each point in time the component is also in one of the phases of that role. Therefore, to maintain consistency, in Paradigm a detailed transition can only be made, if allowed by every current phase of its roles, compliant with the current constraints imposed. In addition, in Paradigm a global transition can only be made, if the component's current detailed state belongs to the trap labeling the global transition, currently entered and hence committed to. For example, if the email client is in detailed state *Idle* as well as in global state *ESend*, the detailed STD cannot take the transition from *Idle* to *PMode*. Hence, from *Idle*, sooner or later it is to take the step to *EMode*, if any, and possibly much later the step from *EMode* to *Done*. Only then a connecting trap is entered, viz. trap *eDone* connecting from *ESend* to *Finished*, whereupon at the role level sooner or later the global transition labeled *eDone* is to be taken from phase *ESend* to phase *Finished*.

The control of actually taking a role step, is governed by the consistency rules. Via a consistency rule other roles are taken into account, relating the behavior of individual components depending on the coordination one wants to achieve. A consistency rule synchronizes single steps of detailed and global STDs as follows: per consistency rule at most one detailed step and arbitrarily many global steps from different roles. As general consistency rule format we use:

detailed transition * global transition , ... , global transition

Relating to a consistency rule format we use the following terminology (cf. [43]).

- *protocol step*: consistency rule with at least one global transition
- *orchestration step*: protocol step with a detailed transition
- *choreography step*: protocol step without a detailed transition
- *protocol*: a set of protocol steps
- *choreography*: a protocol with choreography steps only
- *orchestration*: a protocol with at least one orchestration step
- protocol *conductor*: detailed STD with a transition occurring in the protocol
- protocol *participant*: detailed STD having a role in the protocol.

We have four consistency rules, together called the `PlainOrEncrypt` protocol, that define the `DoPo` role of the email client `EMC`. (Below we shall refine it).

$$\begin{aligned}
& \text{PDA: Idle} \xrightarrow{\text{pmsg}} \text{Plain} \quad * \quad \text{EMC(DoPo): Finished} \xrightarrow{\text{free}} \text{PSend} \\
& \text{PDA: Plain} \xrightarrow{\text{ready}} \text{Idle} \quad * \quad \text{EMC(DoPo): PSend} \xrightarrow{\text{pDone}} \text{Finished} \\
& \text{PDA: Idle} \xrightarrow{\text{emsg}} \text{Encrypt} \quad * \quad \text{EMC(DoPo): Finished} \xrightarrow{\text{free}} \text{ESend} \\
& \text{PDA: Encrypt} \xrightarrow{\text{ready}} \text{Idle} \quad * \quad \text{EMC(DoPo): ESend} \xrightarrow{\text{eDone}} \text{Finished}
\end{aligned}$$

This protocol certainly is an *orchestration*. `PDA` is present as conductor in every consistency rule of it. The first rule, for example, is operationally interpreted as follows: `PDA`, when in state `Idle` and if allowed to do so by every role of it, can make a `pmsg` transition to state `Plain`, if also `EMC`, residing in phase `Finished` of role `DoPo`, has reached trap `free` and thus can make a transfer to phase `PSend`. This way, the detailed step of conductor `PDA` is coupled to the phase transfer or global step of participant `EMC` in role `DoPo`. The four consistency rules specify: `PDA` is conducting `EMC` in sending either in plain or encrypted as well as in preparing for sending again; `EMC` is notifying `PDA` when such conducting has led to the result aimed at.

We want the security module to conduct `PDA`. The alternative of the security module conducting the email client directly is possible too, but not done here. Thus, we have the following two consistency rules, collectively referred to as the `InOrOut` protocol. (A refined version of the protocol is given below.)

$$\begin{aligned}
& \text{SM: InPeri} \xrightarrow{\text{leave}} \text{OutPeri} \quad * \quad \text{PDA(LoPo): EncryptSome} \xrightarrow{\text{triv}} \text{EncryptAll} \\
& \text{SM: OutPeri} \xrightarrow{\text{enter}} \text{InPeri} \quad * \quad \text{PDA(LoPo): EncryptAll} \xrightarrow{\text{triv}} \text{EncryptSome}
\end{aligned}$$

The `PDA` is supposed to have a role `LoPo`, to deal with the location security policy. Within this role there are two phases, `EncryptSome` and `EncryptAll`, each with the trivial trap comprising all states of the phase; furthermore, both traps `triv` are connecting to the other phase. Partition `LoPo` and role `PDA(LoPo)` at its level are depicted in Figure 3.

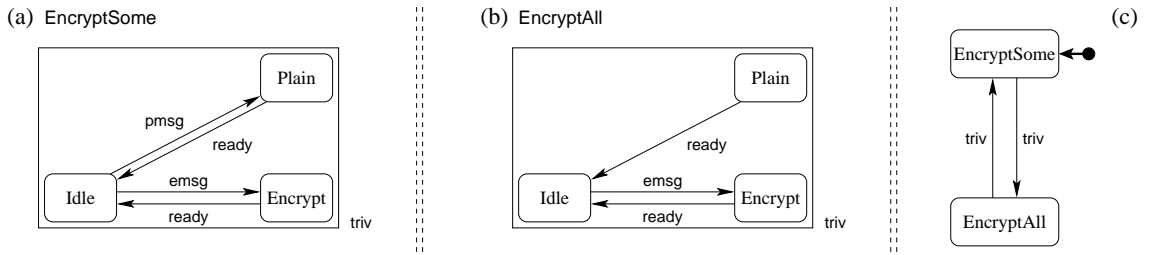


Fig. 3. (a–b) Phases and traps, (c) corresponding role `PDA(LoPo)`

To complete the picture, the overall collaboration involving the two protocols `InOrOut` and `PlainOrEncrypt` of `PDA`, email client and security module, is drawn

in Figure 4. In `InOrOut`, the security module conducts PDA in its role `PDA(LoPo)`; in the `PlainOrEncrypt` protocol, PDA conducts the email client in its role `EMC(DoPo)`. In the figure conducting is indicated by thin boxes.

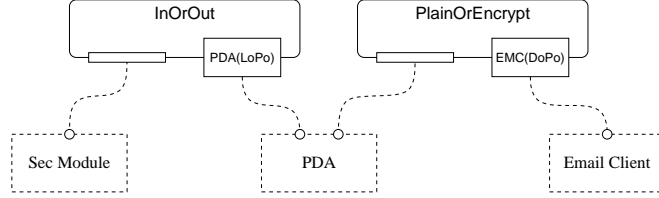


Fig. 4. Collaboration: protocols `InOrOut` and `PlainOrEncrypt`.

Clearly, the above `InOrOut` protocol does not model the possibility for the PDA user to override the location security policy. As depicted in Figure 5, we extend the detailed STD of PDA with a new state `Override` and two transitions (a), and we add a new phase (d) to PDA's role `LoPo` (e). Furthermore, we redefine the original two phases, in view of the addition of state `Override` (b,c). The protocol `InOrOut` is extended with two consistency rules as choreography steps. Also, a consistency rule as orchestration step is added, dealing with the mobility of PDA conducted by security module SM.

$$\begin{aligned}
 & * \text{PDA(LoPo)}: \text{EncryptAll} \xrightarrow{\text{escape}} \text{EscEncrypt} \\
 & * \text{PDA(LoPo)}: \text{EscEncrypt} \xrightarrow{\text{escdone}} \text{EncryptAll} \\
 & \text{SM}: \text{OutPeri} \xrightarrow{\text{enter}} \text{InPeri} * \text{PDA(LoPo)}: \text{EscEncrypt} \xrightarrow{\text{triv}} \text{EncryptSome}
 \end{aligned}$$

In the first choreography step, the PDA, once in the trap `escape`, can transfer unconductedly to phase `EscEncrypt` in which a `pmsg`-transition to the state `Plain` is available. The second choreography step transfers the PDA unconductedly to phase `EncryptAll` once trap `escDone` has been entered. However, to assure no restrictions apply any longer in case the PDA has returned into the security perimeter while sending the message, the consistency rule conducted by SM is added.

As can be seen from Figure 5c for phase `EncryptAll`, the inner trap `escape` contains the new state `Override`. The trap `escape` is used to catch the PDA user's wish to override the standard encryption regulation. The outer trap `triv` is still needed, viz. for the former consistency rule transferring the PDA to phase `EncryptSome`. The new phase `EscEncrypt` in the role `LoPo` of the PDA, Figure 5d, has a transition labelled `pmsg` to the state `Plain`. Note, neither phase `EncryptAll` nor `EscEncrypt` have a `pmsg`-transition leaving from state `Idle`. Only in the special case of overriding, the sending of plain messages is allowed. In order for the email client to stay consistent with this transition, the consistency rule

$$\text{PDA}: \text{Override} \xrightarrow{\text{pmsg}} \text{Plain} * \text{EMC(DoPo)}: \text{Finished} \xrightarrow{\text{free}} \text{PSend}$$

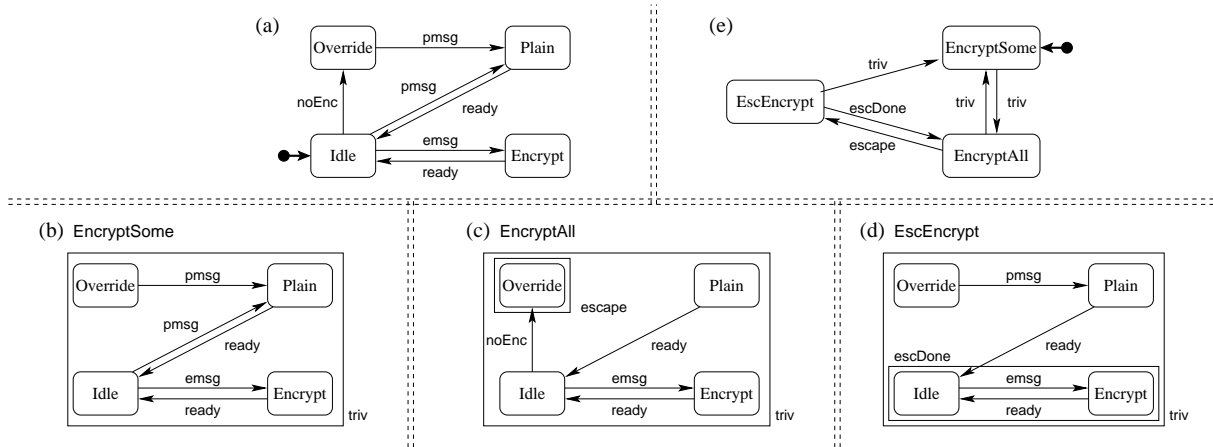


Fig. 5. Overriding: (a) adapted STD for PDA, (b–d) phases, (e) role PDA(LoPo).

is to be added to the PlainOrEncrypt protocol. Note, the STD of the email client itself is not changed. The overriding of the location security policy ends once trap `escDone` of phase `EscEncrypt` is reached, a signal caught by the second choreography rule above.

Using the example of security policies regulating plain or encrypted sending of email, we have illustrated Paradigm’s key notions of STD, phase, (connecting) trap, role and consistency rule as well as the terminology of protocol, orchestration and choreography. In the next section, we shall exploit Paradigm notions and terminology in describing the well-known FOO voting scheme.

3 A voting example: architecting interaction

In this section we address a substantially larger security protocol, the so-called FOO voting scheme proposed by Fujioka, Okamoto and Ohta [14]. The example is small enough for the size of this chapter, but also large enough to underpin our architectural ideas concerning security systems. It comes down to the following. We take a security problem as an interaction situation, where specific interactions are controlled via dynamic constraint regulations. This is modeled within Paradigm via suitable groupings of collaborating components into UML-like collaborations, each responsible for a certain aspect of the overall interaction. Each collaboration then can be taken as a separate architectural unit of a security concern, to be analyzed and understood in relative isolation, resulting in a dedicated specification of a solution for that concern. Via the consistency inherently provided by Paradigm, well-separated concern solutions can be re-united and integrated into a complete solution for the security situation. Based on the architectural organization into separate concerns, the complete solution can be overseen and remains manageable in terms of partial solutions.

For the purpose of this paper, an abstract description of the FOO voting scheme suffices. See [25, 34] for more details. The scheme distinguishes between three main stages. As a first step, the **Organizer** of the election makes public that an election will take place. During the first stage of the election process, **Humans** register with the so-called **Administrator**. During this stage, the **Human** contacts the **Administrator**, identifies himself and sends a blinded message containing his encrypted vote to the **Administrator**. Due to the blinding, the **Administrator** cannot determine the message of the **Human**. In case the **Human** is entitled to vote, the **Administrator** will sign the **Human**'s blinded message and return it. At a certain moment the first stage will end, meaning that **Humans** entitled cannot obtain the possibility to vote any more.

During the second stage a **Human** can send his encrypted vote signed by the **Administrator** to a so-called anonymous channel. This **Channel** collects all received encrypted votes and signatures and sends these to the **Counter** in an arbitrary order. For simplicity, we assume that **Channel** first collects all encrypted votes and then forwards them in bulk. Thus, the output message of the **Channel** cannot be related to a specific **Human**, providing anonymity of votes. The **Channel** does not check the correctness of the messages it obtains from the **Humans**, it only reorders messages. The second stage, encrypted voting, needs to take place within a certain period specified in advance: **Humans** wanting to cast their votes, need to send their encrypted votes to the **Channel** in time.

During the third stage, each voter uncovers his vote anonymously. To that aim, each voter sends his uncovering, i.e. the key he used for encrypting his vote, to the **Counter** via the **Channel**. As the output of the **Channel** hides the sender of the output message, the privacy is protected. Each voter in the scheme makes use of the **Channel** twice. First, the **Channel** collects all encrypted votes. After this, the **Channel** outputs them in an arbitrary order. Analogously, the anonymous channel first collects all keys for uncovering and after that outputs these to the **Counter** in arbitrary order. The set-up, according to the voting scheme, guarantees full anonymity by strict separation of subsequent stages: (i) administering and encrypted voting, (ii) first bulk output, (iii) uncovering, (iv) second bulk output and counting.

As we see from the description, the FOO voting scheme has five different types of components: **Human**, **Organizer**, **Administrator**, **Channel** and **Counter**. The number of the **Human** components is undetermined, $n \in \mathbb{N}$ say. Of the remaining four types there is exactly one of each. Given the above, we differentiate between two major activities: **ElectionOrganizing** covering the overall voting procedure; **VoteHandling** covering the individual handling of voters and votes.

Figure 6 presents the roles of each component grouped into two separate collaborations. No role is given for **Organizer**, as he only conducts the protocol of collaboration **ElectionOrganizing**, as indicated by the thin unlabeled box. Indeed, **ElectionOrganizing** will be coordinated by orchestration. Each **Human_i**, $1 \leq i \leq n$, participates in **ElectionOrganizing** and in **VoteHandling** in its two roles **Human_i(InElection)** and **Human_i(AsVoter)**, for brevity written as **Hum_i(InElec)**

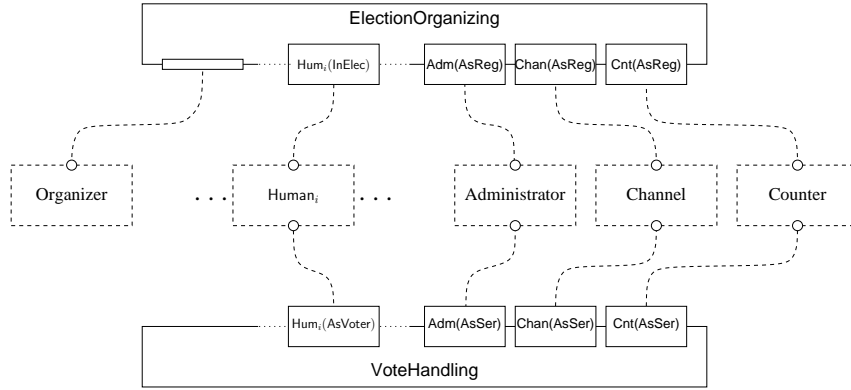


Fig. 6. Collaborations ElectionOrganizing and VoteHandling.

and $\text{Hum}_i(\text{AsVoter})$, respectively. Using the shorter names *Adm*, *Chan* and *Cnt* for the *Administrator*, *Channel* and *Counter*, their cooperation within the collaborations is via their roles *AsRegulator* and *AsServer*, written as *AsReg* and *AsSer* in the figure. Note, the protocol of collaboration *VoteHandling* has no conductor; no thin unlabeled box is present. The protocol will be coordinated choreographically. Except for *Organizer*, all components are contributing to both protocols, but via two different roles, one for each protocol exclusively. Components do not belong themselves to a collaboration, but their roles do so instead. In view thereof the components are visualized in dotted form.

We proceed by explaining the dynamics of the orchestration of the collaboration *ElectionOrganizing*. Because of its overall guiding character, it is easier to explain than the choreographic *VoteHandling*. The dynamics of the latter will be addressed thereafter.

In its five parts, Figure 7 visualizes the detailed STDs of the five components. *Organizer* clearly takes four fixed consecutive actions. The first action *announce* allows all *Human* components to perform their first two main voting activities of administering and encrypted voting. The second action *start* allows the *Administrator* to become active. From then, *Humans* can be handled by *Administrator* and subsequently by *Channel*. The third action *proceed* is done only after the encrypted votes have been received by the *Counter*. The fourth action *declare* is done after all vote uncoverings have been processed and counted. Only then the result of the election is made known.

Any *Human* can try to behave as a voter by doing the eight actions leading from starting state *Idle* to state *Voted*. In state *Idle*, after having heard he may or may not vote, such a *Human* can choose to do action *hear* or *no-Hear*. Similarly, in state *Invited*, he may choose to do action *askForm* or *no-Ask*. In state *ToForm* he does action *getForm* if and only if *Administrator* allows so. In state *WithForm* he can choose action *complete* or *no-Complete*. In state *Filled*, while being served by *Channel*, he does action *sendEnc*. In state *EncSent* he chooses to do action *waitUncover* or *no-Wait*. In state *Waiting*, he can choose to wait until it is allowed

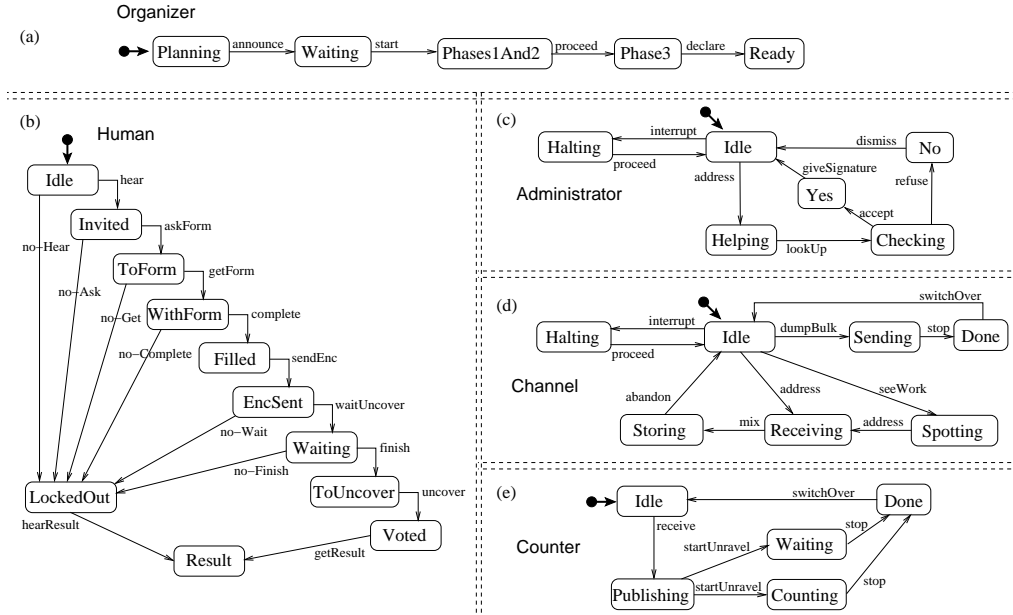


Fig. 7. Detailed STDs: Organizer, Human Administrator, Channel, Counter.

to uncover by doing action `finish`, or he can quit earlier by doing action `no-Finish`. In state `ToUncover`, while again being served by `Channel`, he does action `uncover`. In state `Voted`, he waits until the election outcome is made known, upon which he does action `getResult`. Similarly, in state `LockedOut`, he waits until the election outcome is made known, upon which he does action `hearResult`.

Figure 7c presents `Administrator`. Starting from state `Idle`, any `Human` asking for a form, can be handled individually by `Administrator` as follows. First he does action `address`, then action `lookUp` for a particular `Human`. Depending on the result thereof, he either continues by doing action `accept` followed by `giveSignature`, or he continues by doing `refuse` followed by `dismiss`. After both continuations, `Administrator` is back in state `Idle`, ready to handle another `Human`. In addition, `Administrator` has another possibility in `Idle` as a kind of closing-time policy: first doing action `interrupt` followed by doing action `proceed`, thus returning to `Idle` once more. This action pair is done exactly once, as we shall see. Upon returning to `Idle` each `Human` not yet having asked for a form, cannot do so any longer, but all those that already did, will be handled, one after another indeed.

`Channel` takes over individual `Human` handling from `Administrator`. Moreover, it actually does so two times, first for the encrypted votes they cast. `Channel` explicitly closes encrypted vote handling, for every `Human` involved, by going to state `Done` for the first time. After that, it continues `Human` handling for their uncoverings. Again, it explicitly closes this handling by going to state `Done` for the second time. Starting from state `Idle`, each time a `Human` turns up, `Channel` does action `address`, thus receiving the encrypted vote. Then it does

action `mix`, scheduling the encrypted vote to be delivered later. Finally it does action `abandon`, thus ending service to the particular `Human` and returning to state `Idle`.

Like `Administrator` in state `Idle`, `Channel` has, as closing-time policy, the possibility of doing action `interrupt` followed by `proceed`, thus returning to `Idle` once more. Upon returning to `Idle` each `Human` not yet having turned up, cannot do so any longer, but all those that already did, will be served, one after another indeed. To that aim, action `seeWork` leading to state `Spotting` is chosen to address any such waiting `Human` individually, whereupon actions `address`, `mix` and `abandon` lead back to `Idle`. As soon as no other `Human` is waiting to be served, it takes action `dumpBulk`, thereby forwarding all encrypted votes together to `Counter` for further handling. The action `stop` leads to state `Done`. By taking action `switchOver` returning to state `Idle` occurs. Subsequently, `Channel` displays highly similar behaviour once more, now for handling any `Human`'s uncovering in the same manner as it handled his encrypted vote.

Like `Channel`, `Counter` as given in Figure 7e is involved twice in vote handling: first, receiving all encrypted votes together and publishing them, be it still encrypted, and, second, receiving all uncoverings together and applying these, thereby doing vote counting too. Starting from state `Idle`, `Counter` takes action `receive`, thus receiving all encrypted votes as one bulk. Then, by taking action `startUnravel`, it goes to state `Waiting`, where all encrypted votes are being published. To finish encrypted vote handling, `Counter` takes action `stop` towards state `Done`. By taking step `switchOver` it returns to `Idle` from where it repeats the three actions `receive`, `startUnravel` and `stop` returning to state `Done` again, but in this case via state `Counting`. By doing so, it first receives the uncoverings, it then processes all uncoverings received, thereby counting the votes too. After all uncoverings have been applied and the votes have been counted, it stops in `Done` at last.

The STDs discussed above are detailed STDs. Only one of these, `Organizer`, participates as such in collaboration `ElectionOrganizing`. The other four STDs do participate, but more indirectly, via their respective roles `InElection` and `AsRegulator`, see Figure 6. Each such role is a global STD, whose states and actions are phases and connecting traps, respectively, from a particular partition. These partitions as well as the roles built from them, are visualized in Figure 8 for `Human` and in Figure 9 for `Administrator`, `Channel` and `Counter`.

Note how the four phases of partition `InElection` of the role `Human(InElection)`, subsequently allow a `Human` more freedom: first it can do nothing in phase `PreElection`, second it can go as far as sending its encrypted vote in phase `HalfWayVoting`, third it can even do the uncovering in phase `FinalVoting`, and last it can hear the outcome of the voting in phase `Outcome`. Trivial traps here facilitate phase transfers independently from `Human` behaviour within a phase.

Figure 9a depicts how `Administrator` first can do nothing in phase `Passive` and, second, can do things unrestrictedly in phase `Active`. Slightly less simple, Figure 9b expresses how `Channel`, in phases `Active1` and `Active2`, can do things

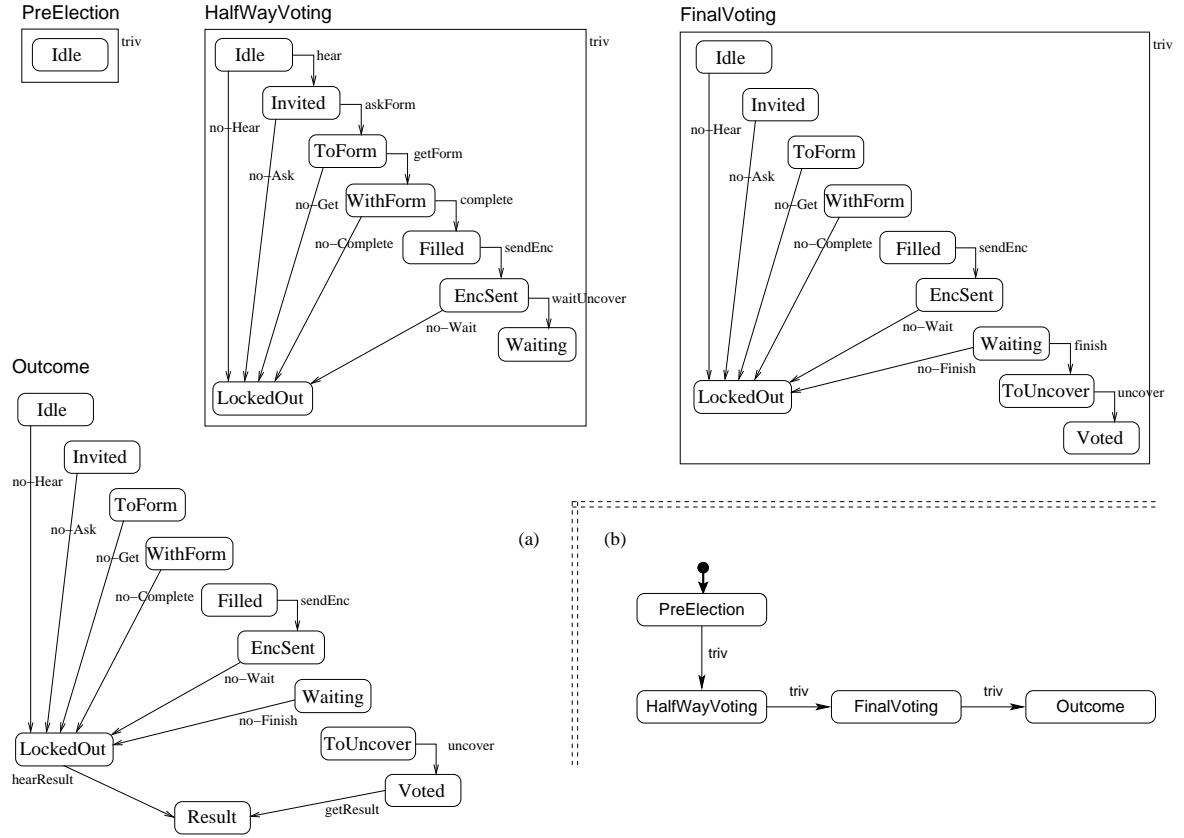


Fig. 8. (a) Partition InElection of $Human_i$, (b) corresponding role $Human_i$ (InElection).

unrestrictedly between state Idle and state Done. Phase Active1 covers all encrypted vote handling by Channel, phase Active2 covers all uncovering handling by Channel. Their difference is, in phase Active2 component Channel can no longer resume its activities after having arrived in state Done.

Figure 9c expresses similar phases for Counter, where phase ShowingEncs is the analogue of Active1 above, precisely during phase HalfWayVoting of all Humans together. Similarly, phase ShowingVotes is the analogue of Active2 above too, but in this case during phase FinalVoting of all Human together, where vote uncoverings are being published and counted accordingly. The phase Resumption bridges the behavioral gap between the two.

It is via these coarse-grained constraints, Organizer conducts the election procedure. By action announce he unleashes all Humans into phase HalfWayVoting while still keeping Administrator in phase Passive. It is only by Organizer's second action start, he puts Administrator into phase Active. By his third action proceed, on the basis of Counter having closed encrypted vote handling, Organizer conducts all Humans into the third phase FinalVoting. In addition, both Channel and Counter are conducted to their next phases, Active2 and Resumption, respectively.

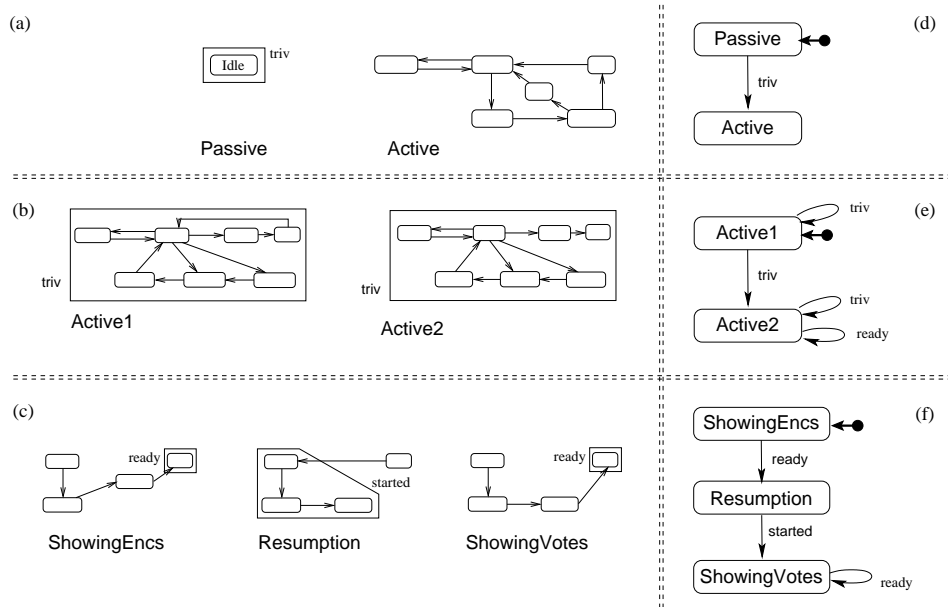


Fig. 9. (a–c) Partitions AsRegulator of Administrator, Channel and Counter, (d–f) corresponding roles.

Note, this is to happen only, if both Counter in its role AsRegulator1 and Channel in its other role AsServer have made enough progress. The combined condition for such progress is: trap ready of phase ShowingEncs (for Counter) as well as trap ready of phase Finishing (for Channel) are the two relevant, nontrivial traps currently entered. In the case of the latter trap ready, its having been entered should be *observed* only, since phase Finishing is not to be transferred –such dynamics belong to Channel’s role AsServer in other protocol VoteHandling. Finally, on the basis of Counter having closed vote uncovering, Organizer takes his last action declare, conducting all Humans into the last phase Outcome. The consistency rules specify this precisely. From their format one can directly recognize the orchestration character of the protocol of collaboration ElectionOrganizing. Note, the use of the universal quantifier to abbreviate the notation in three of the rules, thus establishing a broadcast to each $Human_i$ via his InElection role: one synchronized role step for all n Humans together.

$$\begin{aligned}
 & \text{Organizer : Planning} \xrightarrow{\text{announce}} \text{Waiting} * \\
 & \quad \forall i \in I [\text{Human}_i(\text{InElection}) : \text{PreElection} \xrightarrow{\text{triv}} \text{HalfWayVoting}] \\
 \\
 & \text{Organizer : Waiting} \xrightarrow{\text{start}} \text{Phases1And2} * \\
 & \quad \text{Administrator(AsRegulator) : Passive} \xrightarrow{\text{triv}} \text{Active} \\
 \\
 & \text{Organizer : Phases1And2} \xrightarrow{\text{proceed}} \text{Phase3} * \\
 & \quad \forall i \in I [\text{Human}_i(\text{InElection}) : \text{HalfWayVoting} \xrightarrow{\text{triv}} \text{FinalVoting}], \\
 & \quad \text{Channel(AsRegulator) : Active1} \xrightarrow{\text{triv}} \text{Active2},
 \end{aligned}$$

$\text{Channel(AsServer)} : \text{Finishing} \xrightarrow{\text{ready}} \text{Finishing},$
 $\text{Counter(AsRegulator)} : \text{ShowingEncs} \xrightarrow{\text{ready}} \text{Resumption}$
 $\text{Organizer} : \text{Phase3} \xrightarrow{\text{declare}} \text{Ready} *$
 $\forall i \in I [\text{Human}_i(\text{InElection}) : \text{FinalVoting} \xrightarrow{\text{triv}} \text{Outcome}],$
 $\text{Counter(AsRegulator)} : \text{ShowingVotes} \xrightarrow{\text{ready}} \text{ShowingVotes}$
 $* \text{Counter(AsRegulator)} : \text{Resumption} \xrightarrow{\text{started}} \text{ShowingVotes}$

The last consistency rule is a choreography step assuring, Counter swaps from phase Resumption to phase ShowingVotes after trap started has been entered. This concludes the coordination of collaboration ElectionOrganizing addressing the overall conducting.

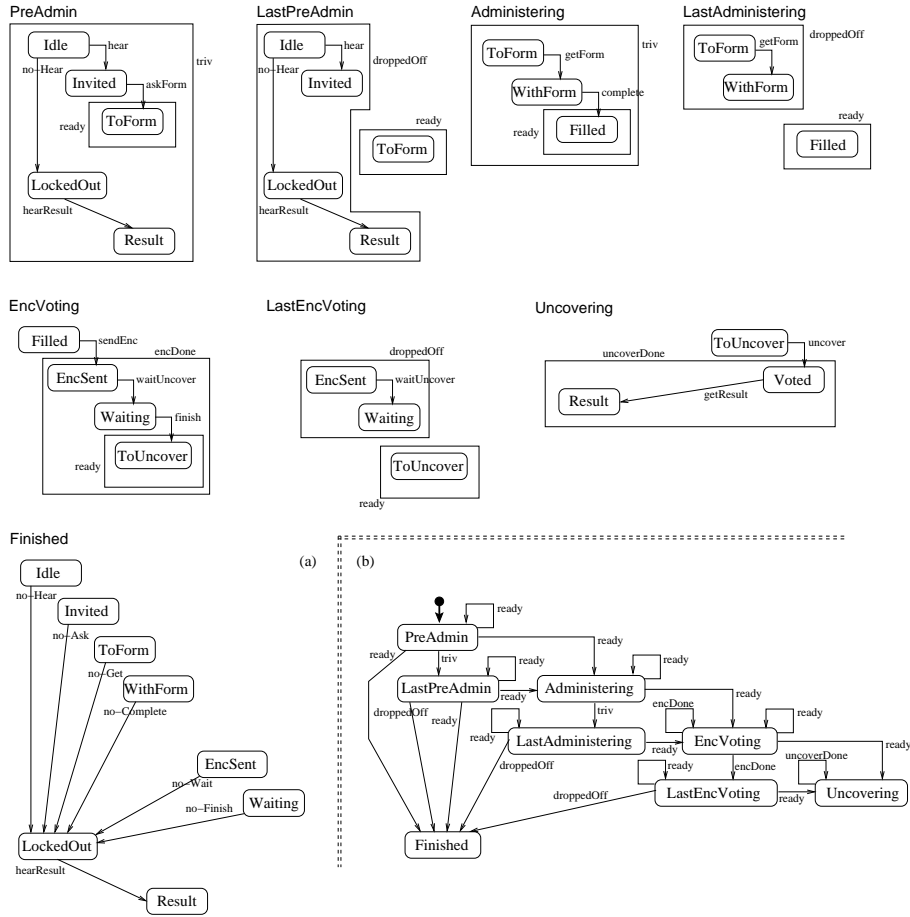


Fig. 10. (a) Partition AsVoter of Human, (b) role Human(AsVoter).

For the remainder of this section we direct our attention towards the collaboration `VoteHandling`. Figure 10a gives partition `AsVoter` of `Human`, Figure 10b gives the corresponding role. We briefly discuss phases, traps and role, first by concentrating on successful, normal voting. Four phases have been defined: `PreAdmin`, `Administering`, `EncVoting` and `Uncovering`. First, phase `PreAdmin` where the role starts by allowing any `Human` to ask for the signature providing service of `Administrator`; second, phase `Administering` where the signed form is given to the voter; third, phase `EncVoting` where encrypted voting is done and a voter can ask to uncover his vote; fourth and last, phase `Uncovering` where uncovering of a voter's encrypted vote is done.

Note, within phase `PreAdmin` any human indeed has the choice between trying to get a signed voting form and not trying to, independent from having the right to vote. So, a prohibiting outcome, different from getting the normal next phase `Administering` imposed, should be possible, even if trap `ready` within phase `PreAdmin` has been entered. The prohibiting outcome is represented by phase `Finished`. As we shall specify below, this is handled by `Administrator` via its role `AsRegulator`. Thus, `Administrator` is involved in any `Human`'s phase transfer from `PreAdmin` either to `Administering` or to `Finished`, both via connecting trap `ready`. In relation to the other normal phase transfers, we have omitted the prohibiting outcome as a possibility, for space reasons only. Thus, similar but simpler, `Channel` is involved in any `Human`'s phase transfer from `Administering` to `EncVoting` and, again, `Channel` is involved in any `Human`'s phase transfer from `EncVoting` to `Uncovering`. So, two components are pipeline-wise involved in the three subsequent, successful, normal phase transfers of any `Human`.

However, there is some time pressure too. So, on the basis of trap `triv` of phase `PreAdmin`, or of `Administering` or on the basis of trap `encDone` having been entered of phase `EncVoting`, the particular phase can be interrupted. In case a `Human` turns out to have entered trap `droppedOff` instead of trap `ready`, the responsible component is not going to serve that `Human` and the `Human` transfers to `Finished`, as he was too late in asking for the next service needed.

According to the above explanation, the non-`Human` component `Counter` is not responsible for any `Human`'s phase transfers. The reason is, `Counter` cooperates with `Channel` exclusively and only twice. The first time is, the sending of all encrypted votes together; the second time is, the sending of all uncoverings together. We shall clarify this point later, after first having explained the phases of both `Administrator` and `Channel` relevant for the pipeline-wise guiding of each `Human` component through his role `AsVoter`. To that aim we refer to Figures 11 and 12. They visualize the phases from partition `AsServer` of the two components `Administrator` and `Channel`, together with the corresponding roles.

In Figure 11, we depict how `Administrator` starts in phase `Finishing`, where it finishes the signature providing service to a `Human` by entering trap `ready` and where it cannot start a next service. In phase `Handlingi`, $1 \leq i \leq n$, it can start doing so for `Humani` only, as follows from the consistency rules below. Serving proceeds up to either giving the signature or refusing to give it: by entering trap `signature` or trap `noSignature` respectively. Via these traps it returns to phase

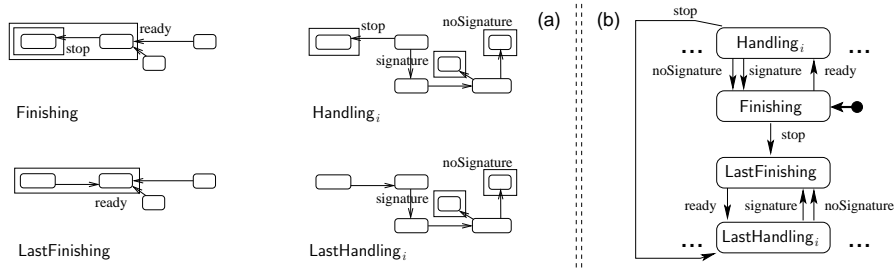


Fig. 11. (a) Partition AsServer of Administrator, (b) role Administrator(AsServer).

Finishing, where it will get ready for handling another Human asking for the signature. In all $n + 1$ phases discussed so far it has the additional possibility to enter trap **stop**, which it does at closing time. From **stop** in Finishing, it continues in **LastFinishing** and from **stop** in **Handling_i** it continues in **LastHandling_i**. Phase transfers between **LastFinishing** and **LastHandling_i** are exactly similar as discussed above between **Finishing** and **Handling_i**. The difference is in the Human components, however. From now on no new Human components can start asking for the signature providing service. This means, only those who were already asking for it *before* trap **stop** was entered by Administrator, have to be served. Below, consistency rules specify this, serving Humans one by one.

Figure 12 visualizes similar phases and role of Channel in view of providing to a Human the service of anonymously sending his encrypted vote or his uncovering thereof. The role starts in phase **Finishing**, where it waits inside trap **ready** to start a new service turn. Until further notice, this service is the anonymous bulk sending of encrypted votes only. After having been asked by Human_i, it provides a fresh service turn, exclusively to Human_i, by going to phase **Handling_i** where it enters trap **next**, via which it returns to phase **Finishing**. Similar as above, the additional trap **stop** is used in view of the closing time policy. Via trap **stop** a swap is made to phase **LastFinishing** or to phase **LastHandling_i**. Via these two phases, any Human that had asked for the anonymous encrypted vote sending service in time, is being served. Only then, all encrypted votes together are being sent to Counter. It is via trap **bulkSent** of **LastFinishing** the role returns to **Finishing**, where it restarts providing the anonymous sending service to any Human asking for it, this time with respect to vote uncoverings. So it returns to phase **Finishing** where it can reenter trap **ready** soon enough. This is particularly relevant for the last consistency rule of the ElectionOrganizing protocol starting on page 15.

The roles given in Figures 10, 11, 12, are synchronized through the following consistency rules. Their synchronization constitutes the main part of the protocol for collaboration **VoteHandling**; the remaining part will be discussed separately, through the additional AsServer role of Counter. Note, none of the protocol steps, being the consistency rules, has a conductor, in line with the choreographic character of the protocol. To facilitate the discussion, we split the presentation of the rules into three groups.

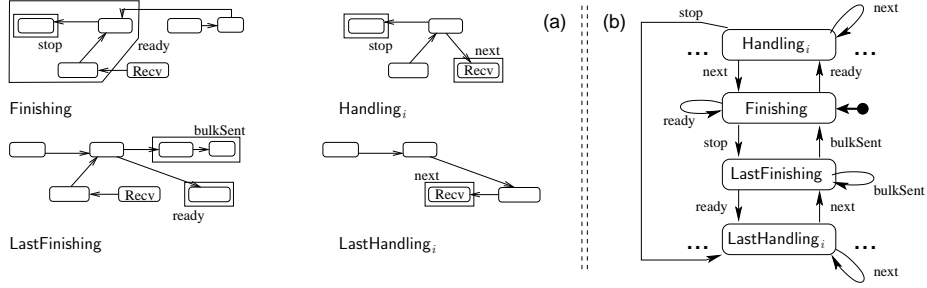


Fig. 12. (a) Partition AsServer of Channel, (b) role Channel(AsServer).

The first group of eight rules specifies how Administrator serves a single Human_{*i*} in his role AsVoter, thereby transferring him from phase PreAdmin either to Administering or to Finishing, possibly via LastPreAdmin in view of the closing time policy. The first three rules of the group address how a Human_{*i*} in two steps is being transferred from PreAdmin to Administering or to Finished. The last three rules address a similar transfer from LastPreAdmin. The two middle rules address phase transfers needed for the closing time policy, transferring all relevant Humans together from PreAdmin to LastPreAdmin. Note, the universal quantifier ranges over those Humans still being in PreAdmin.

- * Human_{*i*}(AsVoter) : PreAdmin $\xrightarrow{\text{ready}}$ PreAdmin ,
Administrator(AsServer) : Finishing $\xrightarrow{\text{ready}}$ Handling_{*i*}
- * Human_{*i*}(AsVoter) : PreAdmin $\xrightarrow{\text{ready}}$ Administering ,
Administrator(AsServer) : Handling_{*i*} $\xrightarrow{\text{signature}}$ Finishing
- * Human_{*i*}(AsVoter) : PreAdmin $\xrightarrow{\text{ready}}$ Finished ,
Administrator(AsServer) : Handling_{*i*} $\xrightarrow{\text{noSignature}}$ Finishing
- * $\forall i \in I$ [Human_{*i*}(AsVoter) : PreAdmin $\xrightarrow{\text{triv}}$ LastPreAdmin],
Administrator(AsServer) : Finishing $\xrightarrow{\text{stop}}$ LastFinishing
- * $\forall i \in I$ [Human_{*i*}(AsVoter) : PreAdmin $\xrightarrow{\text{triv}}$ LastPreAdmin],
Administrator(AsServer) : Handling_{*i*} $\xrightarrow{\text{stop}}$ LastHandling_{*i*}
- * Human_{*i*}(AsVoter) : LastPreAdmin $\xrightarrow{\text{ready}}$ LastPreAdmin ,
Administrator(AsServer) : LastFinishing $\xrightarrow{\text{ready}}$ LastHandling_{*i*}
- * Human_{*i*}(AsVoter) : LastPreAdmin $\xrightarrow{\text{ready}}$ Administering ,
Administrator(AsServer) : LastHandling_{*i*} $\xrightarrow{\text{signature}}$ LastFinishing
- * Human_{*i*}(AsVoter) : LastPreAdmin $\xrightarrow{\text{ready}}$ Finished ,
Administrator(AsServer) : LastHandling_{*i*} $\xrightarrow{\text{noSignature}}$ LastFinishing

The above rules do not express, what happens to a Human trapped in droppedOff of phase LastPreAdmin. This is covered by the second group of three consistency rules, additionally expressing what happens to a Human similarly trapped in droppedOff of phase LastAdministering or of phase LastEncVoting. Note, in these

three cases the particular Human_i only is mentioned. Administrator and Channel are not involved.

- * $\text{Human}_i(\text{AsVoter}) : \text{LastPreAdmin} \xrightarrow{\text{droppedOff}} \text{Finished}$
- * $\text{Human}_i(\text{AsVoter}) : \text{LastAdministering} \xrightarrow{\text{droppedOff}} \text{Finished}$
- * $\text{Human}_i(\text{AsVoter}) : \text{LastEncVoting} \xrightarrow{\text{droppedOff}} \text{Finished}$

The third group of rules is resembling the first group, specifying how Channel serves a particular Human_i in his role AsVoter . Thereby, Channel firstly transfers him from phase Administering to EncVoting and secondly transfers him, much later, from EncVoting to Uncovering . In view of closing time policy, the first transfer may take place via LastAdministering and the second transfer via LastEncVoting . As we did not take into account the possibility of a refusal, the rules are more simple in that respect than those from the first group. A slightly less simple detail, however, arises in the service offering by Channel when in trap next. Channel has to remain inside phase Handling_i until it indeed has received the encrypted vote from Human_i , before it can schedule the vote for output. The same detail is observable in other rules as well. Again, the closing time policy is addressed. Here, we need an additional detail concerning discriminating between handling encrypted votes and handling uncoverings. So, an inspection of its own current phase in the other role AsRegulator has been added to the rules, about being in trap triv either of phase Active1 or of phase Active2 .

- * $\text{Human}_i(\text{AsVoter}) : \text{Administering} \xrightarrow{\text{ready}} \text{Administering},$
 $\text{Channel}(\text{AsServer}) : \text{Finishing} \xrightarrow{\text{ready}} \text{Handling}_i,$
- * $\text{Human}_i(\text{AsVoter}) : \text{Administering} \xrightarrow{\text{ready}} \text{EncVoting},$
 $\text{Channel}(\text{AsServer}) : \text{Handling}_i \xrightarrow{\text{next}} \text{Handling}_i,$
- * $\text{Human}_i(\text{AsVoter}) : \text{EncVoting} \xrightarrow{\text{encDone}} \text{EncVoting},$
 $\text{Channel}(\text{AsServer}) : \text{Handling}_i \xrightarrow{\text{next}} \text{Finishing}$
- * $\forall i \in I [\text{Human}_i(\text{AsVoter}) : \text{Administering} \xrightarrow{\text{triv}} \text{LastAdministering}],$
 $\text{Channel}(\text{AsServer}) : \text{Finishing} \xrightarrow{\text{stop}} \text{LastFinishing},$
 $\text{Channel}(\text{AsRegulator}) : \text{Active1} \xrightarrow{\text{triv}} \text{Active1}$
- * $\forall i \in I [\text{Human}_i(\text{AsVoter}) : \text{Administering} \xrightarrow{\text{triv}} \text{LastAdministering}],$
 $\text{Channel}(\text{AsServer}) : \text{Handling}_i \xrightarrow{\text{stop}} \text{LastHandling}_i,$
 $\text{Channel}(\text{AsRegulator}) : \text{Active1} \xrightarrow{\text{triv}} \text{Active1}$
- * $\text{Human}_i(\text{AsVoter}) : \text{LastAdministering} \xrightarrow{\text{ready}} \text{LastAdministering},$
 $\text{Channel}(\text{AsServer}) : \text{LastFinishing} \xrightarrow{\text{ready}} \text{LastHandling}_i$
- * $\text{Human}_i(\text{AsVoter}) : \text{LastAdministering} \xrightarrow{\text{ready}} \text{EncVoting},$
 $\text{Channel}(\text{AsServer}) : \text{LastHandling}_i \xrightarrow{\text{next}} \text{LastHandling}_i$
- * $\text{Human}_i(\text{AsVoter}) : \text{EncVoting} \xrightarrow{\text{encDone}} \text{EncVoting},$
 $\text{Channel}(\text{AsServer}) : \text{LastHandling}_i \xrightarrow{\text{next}} \text{LastFinishing}$

- * $\text{Human}_i(\text{AsVoter}) : \text{EncVoting} \xrightarrow{\text{ready}} \text{EncVoting},$
 $\text{Channel}(\text{AsServer}) : \text{Finishing} \xrightarrow{\text{ready}} \text{Handling}_i,$
- * $\text{Human}_i(\text{AsVoter}) : \text{EncVoting} \xrightarrow{\text{ready}} \text{Uncovering},$
 $\text{Channel}(\text{AsServer}) : \text{Handling}_i \xrightarrow{\text{next}} \text{Handling}_i,$
- * $\text{Human}_i(\text{AsVoter}) : \text{Uncovering} \xrightarrow{\text{uncoverDone}} \text{Uncovering},$
 $\text{Channel}(\text{AsServer}) : \text{Handling}_i \xrightarrow{\text{next}} \text{Finishing}$
- * $\forall i \in I [\text{Human}_i(\text{AsVoter}) : \text{EncVoting} \xrightarrow{\text{encDone}} \text{LastEncVoting}],$
 $\text{Channel}(\text{AsServer}) : \text{Finishing} \xrightarrow{\text{stop}} \text{LastFinishing},$
 $\text{Channel}(\text{AsRegulator}) : \text{Active2} \xrightarrow{\text{triv}} \text{Active2}$
- * $\forall i \in I [\text{Human}_i(\text{AsVoter}) : \text{EncVoting} \xrightarrow{\text{encDone}} \text{LastEncVoting}],$
 $\text{Channel}(\text{AsServer}) : \text{Handling}_i \xrightarrow{\text{stop}} \text{LastHandling}_i,$
 $\text{Channel}(\text{AsRegulator}) : \text{Active2} \xrightarrow{\text{triv}} \text{Active2}$
- * $\text{Human}_i(\text{AsVoter}) : \text{LastEncVoting} \xrightarrow{\text{ready}} \text{LastEncVoting},$
 $\text{Channel}(\text{AsServer}) : \text{LastFinishing} \xrightarrow{\text{ready}} \text{LastHandling}_i,$
- * $\text{Human}_i(\text{AsVoter}) : \text{LastEncVoting} \xrightarrow{\text{ready}} \text{Uncovering},$
 $\text{Channel}(\text{AsServer}) : \text{LastHandling}_i \xrightarrow{\text{next}} \text{LastHandling}_i,$
- * $\text{Human}_i(\text{AsVoter}) : \text{Uncovering} \xrightarrow{\text{uncoverDone}} \text{Uncovering},$
 $\text{Channel}(\text{AsServer}) : \text{LastHandling}_i \xrightarrow{\text{next}} \text{LastFinishing}$

So far we have discussed the consistency rules coupling the AsVoter role of Human and AsServer roles of Administrator and of Channel. In the remainder of this section we explain the AsServer role of Counter and how it is coupled to the AsServer role of Channel. Figure 13 presents partition AsServer of Counter and the corresponding role. The role starts in phase Finishing, where in trap ready it is waiting for the first bulk to arrive. Such a bulk gets handled in phase Beginning, where via the large trap started the role returns to phase Finishing, but the actual handling newly started just continues within Finishing to its regular end.

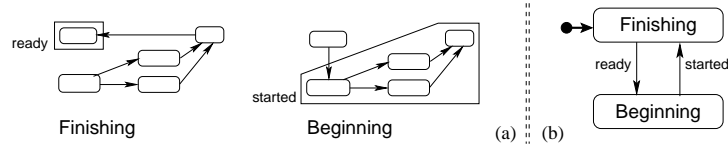


Fig. 13. (a) Partition AsServer of Counter, (b) role Counter(AsServer).

The consistency rules coupling the AsServer role of Counter from Figure 13b and the AsServer role of Channel from Figure 12d are the fourth and last group of rules specifying the protocol for collaboration VoteHandling. It has two rules only. The first rule couples trap bulkSent of Channel having been entered in its phase LastFinishing, to a transfer to phase Beginning of Counter. Thus the end of Channel's service providing during either all Humans' simultaneous phase HalfWayVoting or their simultaneous phase FinalVoting has been reached. Hence,

encrypted vote publishing or uncovering results are to be initiated by **Counter**, uncovering combined with the counting the votes. Recall, the actual restart of **Counter** is done by **Organizer** in protocol **ElectionOrganizing**.

- * Channel(AsServer) : LastFinishing $\xrightarrow{\text{bulkSent}}$ Finishing ,
- Counter(AsServer) : Finishing $\xrightarrow{\text{ready}}$ Beginning
- * Counter(AsServer) : Beginning $\xrightarrow{\text{started}}$ Finishing

As a final remark we like to observe, it is the clear separation of two concerns, achieved through the two collaborations chosen, which has been instrumental in constructing the Paradigm model and in subsequently explaining it. One concern is the overall voting procedure for all **Human** components together, the other concern is the individual handling of each **Human_i** separately, but sufficiently well in line with the overall procedure. Additional features then turned up, like closing time, which we incorporated completely, and like malicious behaviour, which we addressed superficially only, via the two possible outcomes of the service provided on an individual basis by **Administrator**. How to handle truly malicious and unintended behavior is a topic of ongoing research however. This is also relevant outside the field of security; e.g. in business process modeling and in computer supported collaborative work.

4 Model checking safety and security properties

General Paradigm models, in particular security architecture models as above, can be translated into the process language of the model checker **mCRL2** [22, 21]. This way one can formally verify whether an architectural description satisfies certain requirements or whether it exhibits specific undesired behaviour.

Characteristic for **mCRL2** are the support for abstract data types and the use of parametrized boolean equation systems for symbolic model checking [23]. A process description P and the property φ to be checked together yield such an equation system. Solving the equation system provides the answer whether system P satisfies property φ . In addition, **mCRL2** provides extensive support, e.g. for the generation and visualization of LTSs.³

In outline, the translation of Paradigm into **mCRL2** is as follows. A component is represented by the parallel composition of its detailed STD and all its roles. It expresses the component's current local state as well as the current phases for all its roles. Within the parallel construct, state information is communicated from the detailed STD to the global ones, allowing them to update their trap information. Vice versa, according to the Paradigm semantics, a transition to be taken at the detailed level requires the transition to be allowed by all the phases the component is currently in. By a proper synchronization of the actions involved in state updates and transition requests, consistency between detailed STD and global roles is dynamically guaranteed.

We have specified the Paradigm models of both variants of the email example from Section 2 in **mCRL2** and verified a number of properties using the **mCRL2**

³ See www.mcr12.org.

toolset. Here, we list some of them. Note, the translation requires the double occurrence of label `pmsg` in the adapted STD for PDA to be distinguished. We use `pmsg1` from `Idle` to `Plain`, and `pmsg2` from `Override` to `Plain`.

(a) It is not possible to send a plain message while being outside the security perimeter. This property is expressed as

$$[\text{true}^* . \text{sync}(\text{leave}, \text{triv}) . (!\text{sync}(\text{enter}, \text{triv}))^* . \text{sync}(\text{pmsg1}, \text{free})] \text{false}$$

Here, `sync(pmsg1, free)` denotes an initiation of sending a message in plain mode, not triggered by overriding. `sync(leave, triv)` and `sync(enter, triv)` represent synchronization of the security module and the PDA in its role `PDA(LoPo)` captured by the consistency rules on page 7. The above formula states that a sequence of actions in which `sync(leave, triv)` is followed by `sync(pmsg1, free)` is impossible if no action `sync(enter, triv)` is in between. The term `(!sync(enter, triv))^*` expresses a sequence of actions different from `sync(pmsg1, free)`. The tool reported this formula to be valid for both versions of the email example.

(b) For the second version of the email example, we have checked that a message in plain mode can be issued while the PDA is outside the security perimeter only if overriding has been requested. We use the synchronization action `sync(pmsg2, free)` to denote the request to override, and the local EMC action `psend` to denote the sending of a message in plain mode:

$$[\text{true}^* . \text{sync1}(\text{leave}, \text{triv}) . \\ (!(\text{sync}(\text{pmsg2}, \text{free}) \parallel \text{psend} \parallel \text{sync}(\text{enter}, \text{triv})))^* . \text{sync}(\text{pmsg1}, \text{free}) . \\ (!(\text{sync}(\text{pmsg2}, \text{free}) \parallel \text{psend} \parallel \text{sync}(\text{enter}, \text{triv})))^* . \text{psend}] \text{false}$$

A sequence of actions in which `sync(leave, triv)` is followed by `psend`, and in between `sync(pmsg1, free)` occurs, but neither `sync(pmsg2, free)`, nor `sync(enter, triv)` nor `psend` occur, is impossible. In `mCRL2`, `||` denotes disjunction.

(c) If a sending of a message in a certain mode is initiated, assuming fairness, the sending event is executed eventually. For instance, an initiation of sending a message in plain mode, denoted by `sync(pmsg1, free)`, will be followed by a sending event in plain mode, `psend`, i.e.

$$[\text{true}^* . \text{sync}(\text{pmsg2}, \text{free}) . (!\text{psend})^*] \langle \text{psend} \rangle \text{true}$$

Similar formulas are checked for the other modes. Note, the 1-1-correspondence of initiation and actual sending, is guaranteed by the previous property.

We have translated the Paradigm model of the voting scheme of Section 3 into `mCRL2` as well.⁴ For the base case of a single voter the generated LTS has about 130.000 states and 565.000 transitions. Specific tuning of the BES-solver was needed to cope with state space explosion in the case of multiple voters. However, the overall architecture, as represented in Paradigm, with its clear separation of phases and roles per component in each voting phase, allowed us to investigate certain properties of the protocol as a whole, by localizing them on the relevant components. Thus, we have been able to verify, by modularization, a number of security properties, as discussed below.

⁴ See www.win.tue.nl/~andova/research/mcrl2-experiments/VotingExample/.

(a) A voter without a signature from the administrator is not allowed to vote. And, if a voter has not been registered or has not voted, he cannot uncover. Partition `AsVoter` captures the behaviour of a voter and using the relevant trap information we express these properties as

```
[ true* . ( !ready_complete )* . encDone ] false  &&
  [ true* . ( !(ready_complete || encDone ) )* . uncoverDone ] false
```

(b) If the voter for any reason is locked out, he cannot cast his vote. Using the synchronization between `Human(AsVoter)` and `Administrator(AsServer)`, `sync(ready,nosignature)` and the voter action `droppedOff`, we can express this by the property

```
[ true* . sync(ready,nosignature) . (!encDone)* . encDone ] false  &&
  [ true* . droppedOff . (!encDone)* . encDone ] false
```

(c) The next property reflects the coordination of the components of the scheme in the election phase, driven by the organizer. It states that: as soon as the first two phases are closed, i.e. no voter can be registered by the administrator, no voting is allowed anymore, neither he can cast his vote. As the closing of the voting phases is orchestrated by the organizer performing action `proceed`, we can express the property as

```
[ true* . sync(proceed,triv,triv,ready) . (!encDone)* . encDone ] false
```

(d) The last property we consider is that no voter will be allowed to vote more than once. The property `[true*. sendEnci . (sendEnci)*. sendEnci] false` is confirmed by the model checker. This means that a sequence of actions in which there are at least two occurrences of action `sendEnci` is not possible.

We briefly discuss malicious voter behavior in the FOO voting scheme. Any malicious activity a voter wants to perform, can be modeled as a local action in the detailed STD of `Human`, Figure 7. For instance, an additional outgoing transition from state `WithForm` or `Filled` back to state `Invited` means that the voter has an option to ask for a form more than once. Or, a transition from `LockedOut` back to `Idle` would mean that the voter may attempt to start the voting process again after he has been dismissed and put in state `LockedOut`. However, in the Paradigm model, the phases and traps chosen constrain the voter's global behaviour and prevent a dishonest voter to proceed from one to another voting phase as soon as he does not follow the voting policy and timely executes the steps required. For instance, take transition `askFormAgain` from state `ToForm` to state `Invited`. In the partition `Human(AsVoter)`, we find that this transition may be possibly permitted in three phases. However, in the phases `PreAdmin` and `LastPreAdmin` the state `ToForm` forms a trap, thus missing any outgoing transitions. Furthermore, in phase `Finished`, the transition added does not play any role as the phase itself does not have any outgoing transitions. Thus, the transition `askFormAgain` does not add any behaviour to the voting scheme. Similarly, an additional transition from `LockedOut` to `Invited` at the detailed level does not add any global behaviour either.

Intuitively this means that as long as the voter executes these actions according to the voting policy, dishonest activities of the voter are irrelevant and are not a threat to the voting scheme. More precisely, by being *in accordance* to the

voting policy, we actually mean to be *branching bisimilar* to the behaviour of the honest voter. In other words, if the relevant actions listed above are observed and all other local actions are hidden, both from the original model and from an extended or adapted model of a dishonest voter, then branching bisimilarity of the two detailed voter behaviours implies that both voters will show the same overall behaviour in the voting process. This, in fact, provides a security proof of the dishonest voter model with respect to the honest voter model, based on equivalence checking.

5 Related work

Since the seminal paper [31], tool-supported security protocol analysis has been flourishing. The tool Casper provides a high-level language for describing security protocols and secrecy or authentication properties together with a front-end for the CSP-based model checker FDR. For strand spaces, a framework of reconciling complete and partial protocol runs, the Athena tool [42] as well as the constraint solving approach of [36] are available for computer assistance. Another tool for the verification of security protocols is ProVerif [9]. More recent high-performance security checkers include the on-the-fly model checker [7] and the Scyther tool [12]. Main focus of these approaches is not so much the overall architecture, but rather secrecy and authentication in the small, the verification of secrecy and authentication properties of specific security protocols.

In the setting of formal description and analysis methods, anonymity of security protocols goes back to [38] dealing with the Chaum’s famous Dining Cryptographers problem and proposing a notion of anonymity based on trace-equivalence and invariance of permutation of agent names. The use of modal logics, to keep track dynamically of knowledge of principals underlies the approach of [35, 29, 13], for example, for automated anonymity checking. Network anonymity, with the Crowds network as leading example, has been addressed in [15, 33, 40]. For the later case study, the Prism probabilistic model checker has been used. Anonymity for π -calculus has been proposed in [8], in combination with information hiding in [39]. In the present paper, following [33], anonymity is implied by a behavioral property, viz. strict separation of the stages of administering and encrypting from the stage of uncovering.

Access control policies can be integrated in UML models in the approach of [6], called Model Driven Security. The SecureUML proposed, supports various modeling techniques and transformation functions for the construction of access control structures. In [1], a framework is presented for programming distributed computer-supported cooperative work with regulation of role-based access control. In the RW framework [45] for generation and evaluation of access control policies a dedicated model checker can be used to assess policy compliance. Access control and security policies, as illustrated by the email example, can be modeled easily in Paradigm, but is not supported directly. Generic formalisms for architectural modeling, such as the higher-order architectural connectors of [30], can be instantiated to deal with security issues.

Coordination languages can be divided into three main categories: data-based, flow-based and transition-based. Some security issues, in particular role-based access control and trust management, have been addressed in the context of data-based and transition-based coordination languages. Confidentiality in data-based, Linda-like coordination languages is mainly achieved via encryption of tuples and access control on the tuple space. In the context of agent systems, [37] proposes a framework for dynamically establishing security policies. Other approaches to tuple space security include SecOS [44] and SecSpace [11], which also come equipped with a process algebraic semantics. Role-based access control can be statically achieved in transition-based coordination settings via dedicated components. Dynamicity is much more subtle, cf. [10] for a π -calculus dialect. A calculus for ubiquitous computing dealing with trust is proposed in [26], a framework allowing LTL model checking. An instance of a formal approach based on the actor model dealing with trust management is [41]. Paradigm, also a transition-based coordination language, provides an architectural view on secure coordination, unlike the other example formalisms mentioned.

The separation of computation and coordination has been seen as a valuable concept. Flow-based coordination languages strictly follow this distinction: components comprise computation, streams and manipulations thereof through channels comprise coordination. Perhaps, such strong separation causes a gap difficult to bridge. For security problems at least, it is not so clear how honest or malicious dynamics within one component, via the flows the component brings about, must lead to or cannot lead to certain dynamics within another component. The problem then becomes, how to guarantee that component dynamics and the flows between them are coupled right and secure indeed.

6 Concluding Remarks

In brief, the approach outlined above constitutes a modeling suite for design and architecture of security solutions. The suite comprises Paradigm, process algebra and model checking. Paradigm provides the means to factorize security issues and other aspects into focused collaborations and protocols. In process algebra the reformulation of Paradigm models can be further molded using hiding and abstraction, relying on appropriate notions of process equivalence. The state-of-art `mCRL2` toolset supports the analysis and formal verification of security properties and system requirements. The following observations particularly highlight the relevance of the modeling suite.

Separation of concerns helps greatly. As we have demonstrated, not only in the larger voting example, but also in the small email example, it worked out well to split the interaction into different protocols, yet remaining sufficiently consistent. This is complementary to the usual all-detail-matters attitude seen in security protocol analysis.

In the Paradigm visualizations above we did not try to follow UML 2.0 closely. We could have done much better in this respect, however, if not space restrictions had prevented us to do so. E.g. collaboration diagrams and activity diagrams could have been used. For instance, for each of the four protocols in Figures 4

and 6, we could have used a collaboration diagram. Also, for the consistency rules constituting one protocol, we could have used an activity diagram with swimlanes per role and per conductor. Although nicely clarifying, particularly the latter are not so small. See [3, 20] for such a stronger UML flavor. Additional sequence diagrams illustrating protocol interaction in greater detail, would have underlined similarities between our solution and the one in [32] visually too.

Translation of Paradigm models into process algebra [4] preserves the dynamic constraint compositions. Thus, phase and trap constraints as well as consistency rules remain embedded in the synchronization accordingly. By model checking with the mCRL2 tool set, we have succeeded in formally analyzing and verifying properties of an email security system as well as of the FOO e-voting scheme. In this manner, we have established validity of the original Paradigm models and their quality for security systems.

Future work includes the analysis of malicious behaviour. If in a setting the relevant interaction of a malicious agent is essentially the same as that of a honest one, i.e. if after hiding of internal actions their observed behaviour is branching bisimilar, the system is secure regarding dishonest principals. It is noted, such an approach can very well be combined with appropriate modularization guided by the Paradigm architecture, resulting in smaller state spaces to explore.

In a more general setting than security and architecture, other future work will be directed towards developing a tool suite supporting these ideas, amongst others. The tool suite aims at providing an integrated environment for editing and animating/running Paradigm models (cf. [43]), for reproducing and animating their dynamics in UML (cf. [20]), for translating them into PA and subsequently analyzing and model checking the resulting processes. As already touched upon in [4, 5], we particularly aim at feeding unwanted verification results back into the editing and animating parts of the tool suite.

Acknowledgments. We are grateful to Tim Willemse for advice on the use of the mCRL2 toolkit and to the ADS6-reviewers for their constructive comments.

References

1. T. Ahmed and A.R. Tripahti. Specification and verification of security requirements in a programming model for decentralized CSCW systems. *Transactions on Information and Systems Security*, 10, 2007. Article 7.
2. R. Anderson. *A Guide to Building Dependable Distributed Systems*. Wiley, 2001.
3. S. Andova, L. Groenewegen, and E. de Vink. System evolution by migration coordination. In A. Serebrenik, editor, *Proc. BENEVOLE 2008*, pages 18–21, Eindhoven, 2008. Technische Universiteit Eindhoven.
4. S. Andova, L.P.J. Groenewegen, and E.P. de Vink. Dynamic consistency in process algebra: From Paradigm to ACP. In C. Canal, P. Poizat, and M. Sirjani, editors, *Proc. FOCLASA '08*. ENTCS, to appear. 19pp, extended version submitted.
5. S. Andova, L.P.J. Groenewegen, and E.P. de Vink. Formalizing adaptation on-the-fly. In G. Salaün and M. Sirjani, editors, *Proc. FOCLASA '09*. ENTCS, to appear.
6. D. Basin, J. Doser, and T. Lodderstedt. Model driven security: UML models to access control infrastructures. *Transactions on Software Engineering and Methodology*, 15:39–91, 2006.

7. D.A. Basin, S. Mödersheim, and L. Viganò. OFMC: A symbolic model checker for security protocols. *Journal of Information Security*, 4:181–208, 2005.
8. M. Bhargava and C. Palamidessi. Probabilistic anonymity. In M. Abadi and L. de Alfaro, editors, *Proc. CONCUR*, pages 171–185. LNCS 3653, 2005.
9. B. Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *Proc. CSFW, Cape Breton*, pages 82–96. IEEE, 2001.
10. C. Braghin, D. Gorla, and V. Sassone. Role-based access control for a distributed calculus. *Journal of Computer Security*, 14:113–155, 2006.
11. M. Bravetti, N. Busi, R. Gorrieri, R. Lucchi, and G. Zavattaro. Security issues in the tuple-space coordination model. In T. Dimitrakos and F. Martinelli, editors, *Proc. FAST'04*. Kluwer, 2005. 13pp.
12. C.J.F. Cremers. *Scyther: Semantics and Verification of Security Protocols*. PhD thesis, Technische Universiteit Eindhoven, 2006.
13. J. van Eijck and S. Orzan. Epistemic verification of anonymity. In M. ter Beek and F. Cadducci, editors, *Proc. VODCA 2006*, pages 159–174. ENTCS 168, 2007.
14. A. Fujioka, T. Okamoto, and K. Ohta. A practical secret voting scheme for large scale elections. In J. Seberry and Y. Zheng, editors, *Proc. ASIACRYPT'92*, pages 244–251. LNCS 718, 1992.
15. F.D. Garcia, I. Hasuo, W. Pieters, and P. van Rossum. Provable anonymity. In V. Atluri, P. Samarati, R. Küsters, and J.C. Mitchell, editors, *Proc. FMSE, Fairfax*, pages 63–72. ACM, 2005.
16. D. Garlan. Software architecture: a roadmap. In *Proc. ICSE 200, Limerick*, pages 91–101. ACM, 2000.
17. L. Groenewegen and E. de Vink. Operational semantics for coordination in Paradigm. In F. Arbab and C. Talcott, editors, *Proc. COORDINATION 2002*, pages 191–206. LNCS 2315, 2002.
18. L. Groenewegen and E. de Vink. Evolution on-the-fly with Paradigm. In P. Ciancarini and H. Wiklicky, editors, *Proc. COORDINATION 2006*, pages 97–112. LNCS 4038, 2006.
19. L.P.J. Groenewegen, A.W. Stam, P.J. Toussaint, and E.P. de Vink. Paradigm as organization-oriented coordination language. In L. van de Torre and G. Boella, editors, *Proc. CoOrg 2005*, pages 93–113. ENTCS 150(3), 2005.
20. L.P.J. Groenewegen and E.P. de Vink. Dynamic system adaptation by constraint orchestration. Technical Report CSR 08/29, Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, 2008. 20pp, arXiv:0811.3492v1.
21. J.F. Groote, A.H.J. Mathijssen, M.A. Reniers, Y.S. Usenko, and M.J. van Weerdenburg. The formal specification language mCRL2. In E. Brinksma, D. Harel, A. Mader, P. Stevens, and R. Wieringa, editors, *Methods for Modelling Software Systems*. IBFI, Schloss Dagstuhl, 2007. 34 pages.
22. J.F. Groote and M.A. Reniers. Algebraic process verification. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, pages 1151–1208. Elsevier, 2001.
23. J.F. Groote and T. Willemse. Parameterised boolean equation systems. In *Theoretical Computer Science*, volume 343, pages 332–369, 2005.
24. J. Kramer, J. Magee, and S. Uchitel. Software architecture modeling & analysis: A rigorous approach. In M. Bernardo and P. Inverardi, editors, *SFM 2003*, pages 44–51. LNCS 2804, 2003.
25. S. Kremer and M. Ryan. Analysis of an electronic voting protocol in the applied Pi calculus. In S. Sagiv, editor, *Proc. ESOP*, pages 186–200. LNCS 3444, 2005.

26. K. Krukow, M. Nielsen, and V. Sassone. A logical framework for history-based access control and reputation systems. *Journal of Computer Security*, 16:63–101, 2008.
27. J. Küster. *Consistency Management of Object-Oriented Behavioral Models*. PhD thesis, University of Paderborn, 2004.
28. M. Lankhorst, editor. *Enterprise Architecture at Work: Modelling, Communication and Analysis*. Springer, 2005.
29. A. Lomuscio and F. Raimondi. MCMAS: A model checker for multi-agent systems. In H. Hermanns and J. Palsberg, editors, *Proc. TACAS*, pages 450–454. LNCS 3920, 2006.
30. A. Lopes, M. Wermelinger, and J.L. Fiadeiro. Higher-order architectural connectors. *Transactions on Software Engineering and Methodology*, 12:64–104, 2003.
31. G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In T. Margaria and B. Steffen, editors, *Proc. TACAS*, pages 147–166. LNCS 1055, 1996.
32. S. Mauw, J. Verschuren, and E.P. de Vink. Data anonymity in the FOO voting scheme. In M. ter Beek and F. Gadducci, editors, *Proc. VODCA 2006*, pages 5–28. ENTCS 168, 2007.
33. S. Mauw, J. Verschuren, and E.P. de Vink. A formalization of anonymity and onion routing. In P. Samarati, P. Ryan, D. Gollmann, and R. Molva, editors, *Proc. ESORICS 2004*, pages 109–124, Sophia Antipolis, 2004. LNCS 3193.
34. S. Mauw, J. Verschuren, and E.P. de Vink. Data anonymity in the FOO voting scheme. In M. ter Beek and F. Gadducci, editors, *Proc. VODCA 2006*, pages 5–28. ENTCS 168, 2007.
35. R. van der Meyden and K. Su. Symbolic model checking the knowledge of the dining cryptographers. In *Proc. CSFW, Pacific Grove*, pages 280–291. IEEE, 2004.
36. J.K. Millen and V. Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In *Proc. CCS, Philadelphia*, pages 166–175. ACM, 2001.
37. A. Omicini. Towards a notion of agent coordination context. In D.C. Marinescu and C. Lee, editors, *Process Coordination and Ubiquitous Computing*, chapter 12, pages 187–200. CRC Press, 2002.
38. S. Schneider and A. Sidiropoulos. CSP and anonymity. In E. Bertino, H. Kurth, G. Martella, and E. Montolivo, editors, *Proc. ESORICS*, pages 198–218. LNCS 1146, 1996.
39. A. Serjantov and G. Danezis. Towards an information theoretic metric for anonymity. In R. Dingledine and P.F. Syverson, editors, *Proc. PET*, pages 41–53. LNCS 2482, 2002.
40. V. Shmatikov. Probabilistic analysis of an anonymity system. *Journal of Computer Security*, 12:355–377, 2004.
41. V. Shmatikov and C.L. Talcott. Reputation-based trust management. *Journal of Computer Security*, 13:167–190, 2005.
42. D.X. Song, S. Berezin, and A. Perrig. Athena: A novel approach to efficient automatic security protocol analysis. *Journal of Computer Security*, 9:47–74, 2001.
43. A.W. Stam. *Interaction Protocols in PARADIGM*. PhD thesis, LIACS, Leiden University, 2009. Forthcoming.
44. J. Vitek, C. Bryce, and M. Oriol. Coordinating processes with secure spaces. *Science of Computer Programming*, 46:163–193, 2003.
45. N. Zhang, M. Ryan, and D.P. Guelev. Synthesising verified access control systems through model checking. *Journal of Computer Security*, 16:1–61, 2008.