Towards relating security properties on the model layer to implementations

The virtual mall and e-voting case studies

Christoph Feller

April 7, 2012

1 Introduction

In a software development process security properties can be shown on the model or directly on the implemented code. The first, model-based, approach is used for example in the MAKS[5] framework or UMLsec[4]. An example for the language-based approach is the Key-Tool[1]. Both approaches have certain shortcomings. Model-based approaches make few to no guarantees about the actual code while language-based approaches can become quite difficult to handle for larger projects.

In the MoVeSPAcI project wich is part of the DFG project cluster RS3 we try to combine the relative ease of proving security properties with actual guarantees about the code. We want to be able to model a system with its security properties, proof those and show that they are also valid for the implementation. We identified two major differences between models and concrete implementations. First there is usually more indeterminism in the model than there is in the code. Second there is a lot of additonal functionality in actual implementations that was abstracted from in the model; these additional functions might be for example related to GUIs, memory-(de-)allocation or inter-application communication (sockets, RMI, ...).

As one can see our project's goal is not easy to achieve and so we formulated two example scenario to study possible solutions. We have a virtual mall (an example taken from [3]) where merchants offer certain goods but aren't allowed to learn about the offers of others. Then we have an e-voting scenario which arose directly from the project cluster of RS3; here we want to keep the vote confidential. This papers' goal is to describe the scenarios and present possible solutions for the problems they present.

2 Virtual Mall

This example involving a virtual mall is taken from [3]. There an model is given using the methods of the MAKS framework, i.e. the system parts are modelled as state event systems. Also a security property is given and the authors prove it correct again using

the tools provided by the MAKS framework. The system presented contains three types of so-called agents. There are merchants and customers and there is also a platform that models the communication between the agents.

2.1 Agent model

We try to model the agent-layer close to the model given in [3]. We use the Isabelle/HOL and the I-MAKS framework, an adaptation of the MAKS framework for Isabelle/HOL developed by members of Heiko Mantel's group at TU Darmstadt.

```
theory AgentModelEx imports "I-MAKS" CommonData
begin
```

This theory contains all definitions that combine to the virtual mall example.

The type *agentid* contains our agent-identifiers. These are important to denote the sender and receiver of an event.

```
datatype agentid = CID "nat" | MID "nat" | PLATFORM
```

Every agent has a state that we simply model as a function from the variables to some generic values.

```
type_synonym 'v agstate = "var ⇒ 'v"
```

We structure the events the same way as they are given in [3]. That is, we divide the events into the five categories below:

```
datatype ('a,'v) event =
  Init 'a "var ⇒ 'v" ("init_ _" [1000,1000] 1000)
| Start 'a ("start_" [1000] 1000)
| Send 'a 'a "'v msg" ("send_ _ _" [1000,1000,1000] 1000)
| Recv 'a 'a "'v msg" ("recv_ _ _" [1000,1000,1000] 1000)
| Tau 'a nat ("τ_ _" [1000,1000] 1000)

abbreviation Initc ("init_c_" [1000] 1000) where
"Initc n ≡ Init (CID n)"

abbreviation Startc ("start_c_" [1000] 1000) where
"Startc n ≡ Start (CID n)"

abbreviation Sendc ("send_c_" [1000] 1000) where
"Sendc n ≡ Send (CID n)"
```

The type *agval* contains the values used in our different agents' states. We allow natural numbers, boolean values, tuples and lists. We also allow some genericity with the fifth parameter where we can add arbitrary scenario-specific types. We will do this below with the datatype *val*.

```
datatype 'v agval = NN "nat" ("_N" [1000] 1000)
                  | BB "bool" ("_B" [1000] 1000)
                  | TP "'v agval × 'v agval" ("_T" [1000] 1000)
                  | VL "'v agval list" ("_L" [1000] 1000)
```

```
                  | OV "'v"

fun NN_the :: "'v agval ⇒ nat" ("the_N") where
"the_N n_N = n"

fun TP_fst :: "'v agval ⇒ 'v agval" ("fst_T") where
"fst_T (x,y)_T = x"

fun TP_snd :: "'v agval ⇒ 'v agval" ("snd_T") where
"snd_T (x,y)_T = y"

fun VL_Cons :: "'v agval ⇒ 'v agval ⇒ 'v agval" ("_ #_L _") where
"x #_L xs_L = (x # xs)_L"

fun VL_hd :: "'v agval ⇒ 'v agval" ("hd_L") where
"hd_L (a#as)_L = a"

fun VL_tr :: "'v agval ⇒ 'v agval" ("tl_L") where
"tl_L (a#as)_L = as_L"

datatype val = AgentID "agentid"
             | Buffers' "agentid ⇒ ((agentid × agentid × (val agval msg))
list)"

abbreviation AgentID' ("_A" [1000] 1000) where
"x_A ≡ OV (AgentID x)"

abbreviation Buffers where
"Buffers bufs ≡ OV (Buffers' bufs)"
```

We define two variables that are used in customer *and* merchant agents.

```
definition run where "run ≡ Var 0"
definition pc where "pc ≡ Var 1"
```

2.1.1 Customer Agent

We will now define our agents as state event systems. According to the MAKS framework those systems consist of six components:

- the set of states the agent can be in
- the initial state
- the events that can happen in the agent
- the part of these events that are the input events
- analogously the part that are the output events
- a state transition function that takes a state and an event and returns a state as the result

We parameterize the events in the id of the agent; i.e. each agent has its own distinct set of events.

For the customer there are three relevant messages, that we specify below. From these we derive the customer's input and output events. There are also five internal events defined by the constant *ca_tau_evt*.

```
definition MSG_OFFER where "MSG_OFFER ≡ 0"
definition MSG_REQ_OFFER where "MSG_REQ_OFFER ≡ 1"
definition MSG_BUY where "MSG_BUY ≡ 2"

definition ca_tau_evt :: "nat ⇒ (agentid,val agval) event set"
where
"ca_tau_evt n ≡ let a = CID n in {τa 1, τa 2, τa 3, τa 4, τa 5}"

definition ca_in_evt :: "nat ⇒ (agentid,val agval) event set"
where
"ca_in_evt n ≡ let a = CID n in
               {e. ∃msg. inita msg  = e} ∪ {starta} ∪
               {e. ∃b p. recva b (Msg MSG_OFFER (bA,pN)T) = e ∧ b ≠ a}"

definition ca_out_evt :: "nat ⇒ (agentid,val agval) event set"
where
"ca_out_evt n ≡
       let a = CID n in
       {e. ∃b. senda b (Msg MSG_REQ_OFFER (0N)) = e ∧ b ≠ a} ∪
       {e. ∃b p. senda b (Msg MSG_BUY (bA,pN)T) = e ∧ b ≠ a}"

definition ca_evt :: "nat ⇒ (agentid,val agval) event set"
where
"ca_evt n ≡ ca_tau_evt n ∪ ca_in_evt n ∪ ca_out_evt n"


definition v_cmas where "v_cmas ≡ Var 2"
definition v_mas where "v_mas ≡ Var 3"
definition v_ma where "v_ma ≡ Var 4"
definition v_os where "v_os ≡ Var 5"
```

We define the state transition function first as a relation to make the definition a bit clearer. We then show that this relation is indeed a function.

```
inductive_set T_ca :: "nat ⇒
      (val agval agstate × (agentid,val agval) event × val agval agstate)
set"
for n :: "nat"
where
  T_init: "⟦m pc = NN 0; m run = BB False; m' = m(pc:=1N); a = CID n⟧
   ⟹ (m, inita msg, m') ∈ T_ca n"
| T_start: "⟦m pc = NN 1; m run = BB False; m' = m(run:=TrueB, pc:=2N);
             a = CID n⟧
   ⟹ (m, starta, m') ∈ T_ca n"
```

```
| T_copy: "⟦m pc = NN 2; m run = BB True; m' = m(pc:=NN 3);
          m v_mas = (x_A#xs)_L; a = CID n⟧
   ⟹ (m, τ_a 1, m') ∈ T_ca n"
| T_selM: "⟦m pc = NN 3; m run = BB True; m' = m(v_mas:=xs_L, pc:=4_N);
           m v_mas = (x_A#xs)_L; m' v_ma = x_A; a = CID n⟧
   ⟹ (m, τ_a 2, m') ∈ T_ca n"
| T_reqO: "⟦m pc = NN 4; m run = BB True; m' = m(pc:=NN 2);
           m v_ma = b_A; a ≠ b; a = CID n⟧ ⟹
   (m, send_a b (Msg MSG_REQ_OFFER (NN 0)), m') ∈ T_ca n"
| T_noreq: "⟦m pc = NN 2; m run = BB True; m' = m(pc:=NN 5);
            m v_mas = []_L; a = CID n⟧
   ⟹ (m, τ_a 3, m) ∈ T_ca n"
| T_rcvO: "⟦m pc = NN 5; m run = BB True; m' = m(v_os:=((ma_A,price_N)_T#os)_L);
           m v_os = os_L; ma ≠ a; a = CID n⟧
   ⟹ (m,recv_a ma (Msg MSG_OFFER (ma_A,price_N)_T), m') ∈ T_ca n"
| T_ready: "⟦m pc = NN 5; m run = BB True; m' = m(pc:=NN 6); a = CID n⟧
   ⟹ (m, τ_a 4, m') ∈ T_ca n"
| T_best: "⟦m pc = NN 6; m run = BB True;
           m' = m(v_os := (ma_A,price_N)_T ,pc:=NN 7);
           m v_os = os_L; (ma_A,price_N)_T ∈ set os; a = CID n;
           ∀offer'. offer' ∈ set os ⟶ price < the_N (snd_T offer')⟧
   ⟹ (m, τ_a 5, m') ∈ T_ca n"
| T_buy: "⟦m pc = NN 7; m run = BB True; m' = m(pc:=NN 8);
          m v_os = [(ma_A,price_N)_T]_L; a ≠ ma; a = CID n⟧
   ⟹ (m, send_a ma (Msg MSG_BUY (ma_A,price_N)_T), m') ∈ T_ca n"

inductive_cases T_initE[elim]: "(m,init_a msg,m') ∈ T_ca n"
inductive_cases T_startE[elim]: "(m,start_a,m') ∈ T_ca n"
inductive_cases T_copyE[elim]: "(m,τ_a 1,m') ∈ T_ca n"
inductive_cases T_selME[elim]: "(m,τ_a 2,m') ∈ T_ca n"
inductive_cases T_req0E[elim]:
    "(m,send_a b (Msg MSG_REQ_OFFER (NN 0)),m') ∈ T_ca n"
inductive_cases T_noreqE[elim]: "(m,τ_a 3,m') ∈ T_ca n"
inductive_cases T_rcv0E[elim]:
    "(m,recv_a b (Msg MSG_OFFER (ma_A,price_N)_T),m') ∈ T_ca n"
inductive_cases T_readyE[elim]: "(m, τ_a 4, m') ∈ T_ca n"
inductive_cases T_bestE[elim]: "(m, τ_a 5, m') ∈ T_ca n"
inductive_cases T_buyE[elim]:
    "(m, send_a ma (Msg MSG_BUY (ma_A,price_N)_T), m') ∈ T_ca n"

lemma T_ca_fun[intro]:
"⟦(m,e,m') ∈ T_ca n; (m,e,m'') ∈ T_ca n⟧ ⟹ m' = m''"

fun f_ca :: "nat ⇒ val agval agstate ⇒ (agentid, val agval) event
                  ⇀ val agval agstate" where
"f_ca n s e = (if (∃s'. (s,e,s') ∈ T_ca n)
               then Some (SOME s'. (s,e,s') ∈ T_ca n) else None)"

definition cust_agent :: "nat ⇒ ((agentid,val agval) event,
```

```
                                        val agval agstate) SES_record" where
"cust_agent n = (|S_SES=UNIV,
      s0_SES=(λx. (0_N))(v_ma := [(MID 0)_A,(MID 1)_A]_L,run:=False_B),
      E_SES=ca_evt n, I_SES=ca_in_evt n, O_SES=ca_out_evt n, T_SES=f_ca
n|)"

interpretation ca: StateEventSystem "cust_agent n"
```

2.1.2 Merchant agent

```
definition ma_tau_evt :: "nat ⇒ (agentid,val agval) event set"
where
"ma_tau_evt n ≡ {}"

definition ma_in_evt :: "nat ⇒ (agentid,val agval) event set"
where
"ma_in_evt n ≡ let a = MID n in
               {e. ∃msg. init_a msg  = e} ∪ {start_a} ∪
               {e. ∃b. recv_a b (Msg MSG_REQ_OFFER 0_N) = e ∧ b ≠ a}  ∪
               {e. ∃b p. recv_a b (Msg MSG_BUY (a_A,p_N)_T) = e ∧ b ≠ a}"

definition ma_out_evt :: "nat ⇒ (agentid,val agval) event set"
where
"ma_out_evt n ≡
       let a = MID n in
       {e. ∃b p. send_a b (Msg MSG_OFFER (a_A,p_N)_T) = e ∧ b ≠ a}"

definition ma_evt :: "nat ⇒ (agentid,val agval) event set"
where
"ma_evt n ≡ ma_tau_evt n ∪ ma_in_evt n ∪ ma_out_evt n"

definition v_req where "v_req ≡ Var 2"
definition v_of where "v_of ≡ Var 3"

inductive_set T_ma :: "nat ⇒
       (val agval agstate × (agentid,val agval) event × val agval agstate)
set"
for n :: "nat"
where
  T_init: "⟦m pc = NN 0; m run = BB False; m' = m(pc:=NN 1); a = MID n⟧

   ⟹ (m, init_a msg, m') ∈ T_ma n"
| T_start: "⟦m pc = NN 1; m run = BB False;
             m' = m(run:=BB True, pc:=NN 2); a = MID n⟧
   ⟹ (m, start_a, m') ∈ T_ma n"
| T_send: "⟦m pc = NN 2; m run = BB True; m' = m(v_req:=ags_L);
            m v_req = (ag_A#ags)_L; m v_of = (a_A,price_N)_T; a ≠ ag;
            a = MID n⟧
```

```
   ⟹ (m,send_a ag (Msg MSG_OFFER (a_A,price_N)_T),m') ∈ T_ma n"
| T_rreq: "⟦m pc = NN 2; m run = BB True; m' = m(v_req:= (ag_A#ags)_L);
           m v_req = ags_L; a ≠ ag; a = MID n⟧
   ⟹ (m,recv_a ag (Msg MSG_REQ_OFFER 0_N), m') ∈ T_ma n"
| T_rbuy: "⟦m pc = NN 2; m run = BB True; m v_of = (a_A,price_N)_T; a ≠ b;
           a = MID n⟧
   ⟹ (m,recv_a b (Msg MSG_BUY (a_A,price_N)_T),m) ∈ T_ma n"

inductive_cases T_initE2[elim]: "(m,init_a msg,m') ∈ T_ma n"
inductive_cases T_startE2[elim]: "(m,start_a,m') ∈ T_ma n"
inductive_cases T_sendE[elim]:
    "(m,send_a b (Msg MSG_OFFER (a_A,price_N)_T),m') ∈ T_ma n"
inductive_cases T_rreqE[elim]:
    "(m,recv_a b (Msg MSG_REQ_OFFER 0_N), m') ∈ T_ma n"
inductive_cases T_rbuyE[elim]:
    "(m,recv_a b (Msg MSG_BUY (a_A,price_N)_T),m') ∈ T_ma n"

lemma T_ma_inj[intro]:
"⟦(m,e,m') ∈ T_ma n; (m,e,m'') ∈ T_ma n⟧ ⟹ m' = m''"

fun f_ma :: "nat ⇒ val agval agstate ⇒ (agentid, val agval) event
                 ⇀ val agval agstate" where
"f_ma n s e = (if (∃s'. (s,e,s') ∈ T_ma n)
               then Some (SOME s'. (s,e,s') ∈ T_ma n) else None)"

definition merch_agent :: "nat ⇒ ((agentid,val agval) event,
                                  val agval agstate) SES_record" where
"merch_agent n = (|S_SES=UNIV,
  s0_SES=(λx. (NN 0))(run:=BB False, pc:=0_N, v_of:=((MID n)_A,100_N)_T),
  E_SES=ma_evt n, I_SES=ma_in_evt n, O_SES=ma_out_evt n, T_SES=f_ma n|)"

interpretation ma: StateEventSystem "merch_agent n"
```

2.1.3 Platform

```
definition pl_tau_evt :: "(agentid,val agval) event set"
where
"pl_tau_evt ≡ {}"

definition pl_in_evt :: "(agentid,val agval) event set"
where
"pl_in_evt ≡ {e. ∃a b msg. send_a b msg = e ∧ b ≠ a}"

definition pl_out_evt :: "(agentid,val agval) event set"
where
"pl_out_evt ≡ {e. ∃a b msg. recv_a b msg = e ∧ b ≠ a}"

definition pl_evt :: "(agentid,val agval) event set"
```

where
"pl_evt ≡ pl_tau_evt ∪ pl_in_evt ∪ pl_out_evt"

definition *v_buf* **where** *"v_buf ≡ Var 2"*

inductive_set *T_pl ::*
"(val agval agstate × (agentid,val agval) event × val agval agstate) set"
where
T_plsend: "⟦m v_buf = Buffers buf; buf a = (b,a,msg)#msgs; b ≠ a; m' = m(v_buf:=Buffers (buf(a:=msgs)))⟧ ⟹ (m, recv$_a$ b msg, m') ∈ T_pl"
| T_plrecv: "⟦m v_buf = Buffers buf; buf b = msgs; b ≠ a; m' = m(v_buf:=Buffers (buf(b:=(a,b,msg)#msgs)))⟧ ⟹ (m, send$_a$ b msg, m') ∈ T_pl"

fun *f_pl :: "val agval agstate ⇒ (agentid, val agval) event ⇀ val agval agstate"* **where**
"f_pl s e = (if (∃s'. (s,e,s') ∈ T_pl) then Some (SOME s'. (s,e,s') ∈ T_pl) else None)"

definition *platform :: "((agentid,val agval) event, val agval agstate) SES_record"* **where**
"platform = (|S_SES=UNIV, s0_SES=(λx. (0$_N$))(run:=False$_B$), E_SES=pl_evt, I_SES=pl_in_evt, O_SES=pl_out_evt, T_SES=f_pl|)"

interpretation *pl: StateEventSystem "platform"*

2.1.4 Composition of the system

Here we show that we can compose two (different) merchants, a customer and the platform and get a valid state event systen.

interpretation *mes1 : ComposedEventSystem "ma.induced_ES 0" "ma.induced_ES (Suc 0)"*

interpretation *cmes1: ComposedEventSystem "ca.induced_ES 0" "ma.induced_ES 0 ‖ ma.induced_ES (Suc 0)"*

interpretation *system: ComposedEventSystem "pl.induced_ES" "ca.induced_ES 0 ‖ (ma.induced_ES 0 ‖ ma.induced_ES (Suc 0))"*

The view for the complete system:

definition *C_omc :: "agentid ⇒ agentid ⇒ (agentid, val agval) event set"*

where
"C_omc ma ca ≡
{x. ∃price m. x = send$_m$ ca (Msg MSG_OFFER (m$_A$,price$_N$)$_T$) ∧ m ≠ ma}"

fun *V_om :: "agentid ⇒ (agentid, val agval) event set"*
where
"V_om (MID ma) = ma_evt ma"

definition *ν_g :: "agentid ⇒ agentid ⇒ (agentid, val agval) event View"*
("ν$_g$")
where
"ν$_g$ ma ca ≡ (|V=V_om ma, N=UNIV − (V_om ma ∪ C_omc ma ca),
C=C_omc ma ca|)"

The view for customer agents:

definition *C_ca :: "agentid ⇒ agentid ⇒ (agentid, val agval) event set"*

where
"C_ca ma ca ≡
{x. ∃a price m. x = recv$_m$ ca (Msg MSG_OFFER (a$_A$,price$_N$)$_T$) ∧ m ≠ ma}
∪ {x. ∃a price m. x = send$_m$ ca (Msg MSG_BUY (a$_A$,price$_N$)$_T$) ∧ m ≠ ma}"

fun *ν_ca :: "agentid ⇒ agentid ⇒ (agentid, val agval) event View" ("ν$_{ca}$")*

where
"ν$_{ca}$ ma (CID ca) = (|V=ca_evt ca − (ca_tau_evt ca ∪ C_ca ma (CID ca)),
N=ca_evt ca, C=C_ca ma (CID ca)|)"

The view for merchant agents:

definition *C_ma :: "agentid ⇒ agentid ⇒ (agentid, val agval) event set"*

where
"C_ma ma ca ≡
{x. ∃a price m. x = send$_m$ ca (Msg MSG_OFFER (a$_A$,price$_N$)$_T$) ∧ m ≠ ma}
∪ {x. ∃a price m. x = recv$_m$ ca (Msg MSG_BUY (a$_A$,price$_N$)$_T$) ∧ m ≠ ma}"

fun *ν_ma :: "agentid ⇒ agentid ⇒ (agentid, val agval) event View" ("ν$_{ma}$")*

where
"ν$_{ma}$ ma (CID ca) = (|V=ca_evt ca − (ca_tau_evt ca ∪ C_ca ma (CID ca)),
N=ca_evt ca, C=C_ma ma (CID ca)|)"

end

2.2 Actor model

The actor model is also defined in Isabelle/HOL. Its aim is to bridge the gap between the abstract (agent-)model and the concrete implementation. We assume that some security property has been given and proven for the agent model. As the actor model is defined formally and with the same formalism we are able and intend to prove the preservance of said security property to the actor model.

Design space We want the actor model to be close to the implementation. There should be object that are equivalent to methods so that we can write specifications for them. Additionally it should have trace-based semantics so that we can reuse the security properties of the MAKS framework. Apart from this we have a lot of freedom in the design of our actors. So we can, for example, choose the message passing mechanism. We choose to model incoming and outgoing event queues for the actors. The theory *ActorModel* will detail our design decisions.

theory *ActorModel* **imports** *CommonData BSPs*
begin

In this theory we introduce our actor model in general. We explain what we mean when we talk about an actor and how the semantics of such an actor looks.

We introduce unique identifiers for actors:

type_synonym *actorid = "nat"*

The following type describes the internal state of an actor.

type_synonym *'v acstate = "var ⇀ 'v"*

We need two special variables for assertions.

definition *v_par :: "var"* **where** *"v_par = Var 2"*

definition *v_self :: "var"* **where** *"v_self = Var 3"*

We want actors to be able to create new actors. There are different ways to allow that (e.g. via cloning). As we try to define our actors with a certain similarity to Java objects we introduce the analog to Java classes namely actor classes.

So here we define actor classes. An actor class defines how an actor of that class reacts when it receives messages with a certain message id.

type_synonym *'v handleres = "'v acstate × (actorid × 'v msg) list"*
type_synonym *'v handler = "actorid ⇒ 'v ⇒ 'v acstate ⇒ 'v handleres"*
type_synonym *'v actorclass = "msgid ⇒ 'v handler"*

'v handleres contains results of the handling of a message: a new state and a list of messages to be sent.

The type *'v handler* contains the functions that produce such a result. These function take three parameters: the actors id, a generic parameter value and the current

(local) state. Note that these message handlers correspond to methods on the implementation level.

'v actorclass contains all actor classes i.e. definitions on how to handle different messages.

But some classes should never receive some message ids. We have to specify what happens then. We have basically three options:

- Ignore unhandled message ids
- Let the execution get stuck
- Introduce some error state or exception handling

We choose the first option because it's easy to implement.

So we define an handler that does nothing:

```
definition emptyHnd :: "actorid ⇒ 'v ⇒ 'v acstate ⇒ 'v handleres"
where
"emptyHnd self param s ≡ (s,[])"
```

Now an actor consist of its actor class, a list of queued incoming messages with their sender a list of outgoing messages and some internal state.

```
record 'v actor =
  aclass :: "'v actorclass"
  in_msgs :: "'v msg list"
  out_msgs :: "(actorid × 'v msg) list"
  state :: "'v acstate"

notation make ("mkActor")
```

A global state is a partial function mapping actorids to the respective actors.

```
— global state
type_synonym 'v gstate = "actorid ⇀ 'v actor"

definition free_aid :: "'v gstate ⇒ actorid"
where
"free_aid g = (LEAST aid. g aid = None)"

record 'v aevent =
  aesender :: "actorid"
  aerecv :: "actorid"
  aemsg :: "'v msg"

notation make ("mkAEvent")
```

The following inductive set defines the step that a single actor can do and the trace that results from said step.

```
inductive actor_step :: "actorid ⇒ 'v actor ⇒ 'v aevent list ⇒
```

```
                        'v actor ⇒ bool"
where
  actor_emit[intro]:
  "⟦ac = mkActor acl (mla @ (Msg msgid val) # mlb) smsg s;
    acl msgid self val s = (t,rl)⟧
   ⟹ actor_step self ac (map (λ(recv,m). mkAEvent aid recv m) rl)
                  (ac(|in_msgs := mla @ mlb, out_msgs:=rl@smsg, state:=t|))"
| actor_recv[intro]:
  "⟦in_msgs ac = ml⟧
   ⟹ actor_step self ac [mkAEvent sender aid msg] (ac(|in_msgs:=msg#ml|))"
| actor_refl[intro]:
  "actor_step self ac [] ac"
| actor_trans[intro]:
  "⟦actor_step self ac el ac'; actor_step self ac' el' ac''⟧
   ⟹ actor_step self ac (el @ el') ac''"

lemma actor_recv2[intro]:
"⟦in_msgs ac = ml; ac' = ac(|in_msgs:=msg#ml|)⟧
   ⟹ actor_step self ac [mkAEvent sender aid msg] ac'"
```

We also want to be able to specify the effects of a single incoming message. We do this by having said specifications as shallow embeddings; i.e. every specification is consists of three predicates that represent precondition, postcondition and the set of outgoing messages.

```
type_synonym 'v spec_env = "var ⇒ 'v"

record 'v acspec =
  precond :: "'v spec_env ⇒ actorid ⇒ 'v ⇒ 'v acstate ⇒ bool"
  postcond :: "'v spec_env ⇒ actorid ⇒ 'v ⇒ 'v acstate ⇒ bool"
  emits :: "'v spec_env ⇒ actorid ⇒ 'v ⇒ 'v acstate ⇒
                                        (actorid × 'v msg) list"

definition spec_valid :: "'v acspec ⇒ 'v handler ⇒ bool" where
"spec_valid spec hnd ≡
   ∀L self par S S' l. precond spec L self par S ∧ hnd self par S = (S',l)
         ⟶ postcond spec L self par S' ∧ emits spec L self par S = l"
```

The type *syscfg* represents a system's initial state together with a predicate that tells us at which point which actors can be created.

```
record 'v syscfg =
  init :: "'v gstate"
  canCreate :: "'v gstate ⇒ 'v actorclass ⇒ bool"

notation make ("mkSyscfg")
```

The next relation describes a system of multiple actors. It relates states where messages get taken from the outgoing queue of actors to the incoming queue of the receiver. Messages the were emitted in one *actor step* can be send this way in arbitrary order.

But an actor can only do another step if its outgoing queue is empty.

```
inductive actors_step :: "'v syscfg ⇒ 'v gstate ⇒
                          ('v aevent) list ⇒ 'v gstate ⇒ bool"
where
  actors_step_send_actor:
  "⟦g aid1 = Some ac1; g aid2 = Some ac2;
    out_msgs ac1 = om1 @ amsg # om2;
    amsg = (DstActor aid2, msg);
    g' = (g(aid1↦ac1(|out_msgs:=om1 @ om2|), aid2↦ac2(|in_msgs:=(in_msgs
ac2)@[msg]|)))⟧
   ⟹ actors_step cfg g [mkAEvent aid1 aid2 msg] g'"
| actors_step_actor:
  "⟦g aid = Some ac; out_msgs ac = [];
    actor_step aid ac el ac'⟧
   ⟹ actors_step cfg g [] (g(aid↦ac'))"
| actors_new_actor:
  "⟦canCreate cfg g ac⟧
   ⟹ actors_step cfg g [] (g(free_aid g↦mkActor ac [] [] empty))"
| actors_step_refl:
  "actors_step cfg g [] g"
| actors_step_trans:
  "⟦actors_step cfg g eml g'; actors_step cfg g' eml' g''⟧
   ⟹ actors_step cfg g (eml @ eml') g''"

lemma global_range_stays_finite[intro]:
"actors_step cfg g eml g' ⟹ finite (range g) ⟹ finite (range g')"


lemma actor_ex_trans:
"(∃ac'. actor_step self ac ml ac' ∧ (∃ac''. actor_step self ac' [m] ac''))
 ⟹ (∃ac''. actor_step self ac (ml @ [m]) ac'')"

fun acTr :: "nat ⇒ 'v actor ⇒ 'v aevent list set"
where
"acTr aid ac = {tr. ∃ac'. actor_step aid ac tr ac'}"

fun gTr :: "'v syscfg ⇒ 'v aevent list set"
where
"gTr cfg = {tr. ∃ g g'. actors_step cfg g tr g'}"

end
```

Now that we have choosen our actor model we can use it to define the virtual mall example.

```
theory ActorModelEx imports ActorModel
begin
```

Now we define actors (hopefully) equivalent to the agents further above. Note that we do not specify a platform here. We assume that the function of the platform is already subsumed in our actor semantics.

```
definition system :: "actorid" where "system ≡ 0"

datatype acval = NN "nat" ("_N" [1000] 1000)
               | BB "bool" ("_B" [1000] 1000)
               | TP "acval × acval" ("_T" [1000] 1000)
               | VL "acval list" ("_L" [1000] 1000)

definition MSG_INIT where "MSG_INIT ≡ 1"
definition MSG_SEL where "MSG_SEL ≡ 2"
definition MSG_SEND where "MSG_SEND ≡ 3"
definition MSG_REQ_OFFER where "MSG_REQ_OFFER ≡ 4"
definition MSG_OFFER where "MSG_OFFER ≡ 5"
definition MSG_START_TIMER where "MSG_START_TIMER ≡ 6"
definition MSG_ON_TIMER where "MSG_ON_TIMER ≡ 7"
definition MSG_FIND_BEST where "MSG_FIND_BEST ≡ 8"
definition MSG_BUY where "MSG_BUY ≡ 9"
definition MSG_MYID where "MSG_MYID ≡ 10"
definition MSG_INIT_RET where "MSG_INIT_RET ≡ 11"
```

2.2.1 Customer Actor

Now we can begin to specify the customer. After some variables we declare this actor's message handlers andprovide specifications for said handlers. We also prove that our specifications are valid.

```
definition v_mas where "v_mas ≡ Var 10"
definition v_midx where "v_midx ≡ Var 11"
definition v_recv where "v_recv ≡ Var 12"
definition v_boff where "v_boff ≡ Var 13"

definition cust_init_spec :: "acval acspec"
where
"cust_init_spec ≡
  (|precond = λL self par S. ∃l. par = l_L,
    postcond = λL self par S. (S v_mas) = Some par ∧ (S v_recv = Some False_B)
                            ∧ (S v_boff = None) ∧ (S v_midx = Some 0_N),
    emits = λL self par S. [(self, Msg MSG_SEL 0_N)]|)"

fun cust_init :: "acval handler"
where
  "cust_init self l_L s =
          (s(v_boff:=None)(v_midx↦0_N,v_mas↦l_L,v_recv↦False_B),
                                              [(self, Msg MSG_SEL 0_N)])"
| "cust_init self par s = (s,[])"

lemma cust_init_valid:
"spec_valid cust_init_spec cust_init"
```

For some of our specifications we need logical variables to carry information from the

pre- to the postcondition. We denote them with the prefix L_.

```
definition L_x where "L_x ≡ Var 1"
definition L_y where "L_y ≡ Var 2"
definition L_z where "L_z ≡ Var 3"

definition cust_start_spec :: "acval acspec" where
"cust_start_spec ≡
 (|precond = λL self par S.
                ∃ml midx. S v_mas = Some ml_L ∧  L L_x = ml_L
                        ∧ S v_midx = Some midx_N ∧ L L_y = midx_N,
   postcond = λL self par S.
                (∃ml midx. L L_x = ml_L ∧ L L_y = midx_N ∧
                           (if midx < length ml
                              then S v_recv = Some True_B
                              else True)),
   emits = λL self par S.
                (case L L_y of midx_N ⇒
                  (case L L_x of ml_L ⇒
                    (if midx < length ml
                       then [(self, Msg MSG_SEND 0_N)]
                       else [(system,Msg MSG_START_TIMER self_N)])
                     | _ ⇒ [])
                   | _ ⇒ []) |)"

fun cust_start :: "acval handler"
where
"cust_start self _ s =
    (case s v_mas of Some ml_L ⇒
      (case s v_midx of Some midx_N ⇒
         (if midx < length ml
            then (s(v_recv↦True_B),[(self,Msg MSG_SEND 0_N)])
            else (s,[(system,Msg MSG_START_TIMER self_N)]))
              | _ ⇒ (s,[])) | _ ⇒ (s,[]))"

lemma cust_start_valid:
"spec_valid cust_start_spec cust_start"

definition cust_send_spec :: "acval acspec"
where
"cust_send_spec ≡
  (|precond = λL self par S.
                ∃ml midx. S v_mas = Some ml_L ∧ L L_x = ml_L ∧
                          S v_midx = Some midx_N ∧ L L_y = midx_N,
    postcond = λL self par S. True,
    emits = λL self par S. (case L L_x of ml_L ⇒
                             (case L L_y of midx_N ⇒
                               (case ml!midx of m_N ⇒
                                     [(m, Msg MSG_REQ_OFFER self_N),
                                      (self, Msg MSG_SEL 0_N)]
```

```
                    | _ ⇒ []) | _ ⇒ []) | _ ⇒ []) |)"

fun cust_send :: "acval handler"
where
"cust_send self _ s =
   (case s v_mas of Some ml_L ⇒
     (case s v_midx of Some midx_N ⇒
       (case ml!midx of m_N ⇒ (s,[(m, Msg MSG_REQ_OFFER self_N),
                                  (self, Msg MSG_SEL 0_N)])
         | _ ⇒ (s,[])) | _ ⇒ (s,[])) | _ ⇒ (s,[]))"

lemma cust_send_valid:
"spec_valid cust_send_spec cust_send"

definition cust_recv_spec :: "acval acspec"
where
"cust_recv_spec ≡
  (|precond = λL self par S. ∃sender price recv offs.
                                  par = (sender_N,price_N)_T
                                ∧ L L_x = (sender_N,price_N)_T
                                ∧ S v_recv = Some recv ∧ L L_y = recv
                                ∧ S v_boff = Some offs ∧ L L_z = offs,
    postcond =
       λL self par S. ∃sender price. L L_x = (sender_N,price_N)_T ∧
          S = (case L L_y of True_B ⇒
                (case L L_z of (bsender_N,bprice_N)_T
                      ⇒ (if price < bprice
                            then S(v_boff↦(sender_N,price_N)_T) else S)
                  | _ ⇒ (S(v_boff↦(sender_N,price_N)_T)))
              | _ ⇒ S),
    emits = λL self par S. []|)"

fun cust_recv :: "acval handler"
where
"cust_recv self (sender_N,price_N)_T s =
  (case s v_recv of Some True_B ⇒
    (case s v_boff of Some (bsender_N,bprice_N)_T
          ⇒ (if price < bprice
                then s(v_boff↦(sender_N,price_N)_T) else s,[])
      | _ ⇒ (s(v_boff↦(sender_N,price_N)_T),[]))
   | _ ⇒ (s,[]))"
| "cust_recv self par s = (s,[])"

lemma cust_recv_valid:
"spec_valid cust_recv_spec cust_recv"


definition cust_timer_spec :: "acval acspec"
where
```

```
"cust_timer_spec ≡
  (|precond = λL self par S. S v_recv = Some (L L_x),
    postcond = λL self par S.
         S = (case L L_x of True_B ⇒ (S(v_recv↦False_B))
                          | _ ⇒ S),
    emits = λL self par S.
        (case L L_x of True_B ⇒ [(self,Msg MSG_FIND_BEST 0_N)]
                     | _ ⇒ [])|)"

fun cust_timer :: "acval handler"
where
"cust_timer self _ s =
   (case s v_recv of Some True_B ⇒ (s(v_recv↦False_B),
                                    [(self,Msg MSG_FIND_BEST 0_N)])
                   | _ ⇒ (s,[]))"

lemma cust_timer_valid:
"spec_valid cust_timer_spec cust_timer"



definition cust_find_spec :: "acval acspec"
where
"cust_find_spec ≡
  (|precond = λL self par S. S v_boff = Some (L L_x),
    postcond = λL self par S. True,
    emits = λL self par S.
       (case L L_x of (sender_N,price_N)_T ⇒ [(sender, Msg MSG_BUY price_N)]
                    | _ ⇒ [])|)"

fun cust_find :: "acval handler"
where
"cust_find self _ s =
   (case s v_boff of Some (sender_N,price_N)_T ⇒
                          (s,[(sender, Msg MSG_BUY price_N)])
                   | _ ⇒ (s,[]))"

lemma cust_find_valid:
"spec_valid cust_find_spec cust_find"

definition cust_class where
"cust_class ≡ ((λx. emptyHnd)(MSG_INIT:=cust_init,
                              MSG_SEL:=cust_start,
                              MSG_SEND:=cust_send,
                              MSG_OFFER:=cust_recv,
                              MSG_ON_TIMER:=cust_timer,
                              MSG_FIND_BEST:=cust_find))"

definition class_cust_spec :: "acval actorclass ⇒ bool"  where
```

```
"class_cust_spec cc ≡ spec_valid cust_init_spec (cc MSG_INIT)
                    ∧ spec_valid cust_start_spec (cc MSG_SEL)
                    ∧ spec_valid cust_send_spec (cc MSG_SEND)
                    ∧ spec_valid cust_recv_spec (cc MSG_OFFER)
                    ∧ spec_valid cust_timer_spec (cc MSG_ON_TIMER)
                    ∧ spec_valid cust_find_spec (cc MSG_FIND_BEST)"
```

2.2.2 Merchant Actor

We define the merchant the same way we did the customer.

```
definition v_price where "v_price ≡ Var 2"

definition merc_init_spec :: "acval acspec"
where
"merc_init_spec ≡
  (|precond = λL self par S. ∃price. par = price_N,
    postcond = λL self par S. ∃price. par = price_N ∧
                                        S v_price = Some price_N,
    emits = λL self par S. [] |)"

fun merc_init :: "acval handler"
where
"merc_init self price_N s = (s(v_price↦price_N),[])"

lemma merc_init_valid:
"spec_valid merc_init_spec merc_init"

definition merc_req_spec :: "acval acspec"
where
"merc_req_spec ≡
  (| precond = λL self par S. ∃sender. par = sender_N,
    postcond = λL self par S. True,
    emits = λL self par S. (case par of sender_N ⇒
                              (case S v_price of Some price ⇒
                                    [(sender,Msg MSG_OFFER price)]
                                                | _ ⇒ [])) |)"

fun merc_req :: "acval handler"
where
"merc_req self sender_N s =
   (case s v_price  of Some price ⇒ (s,[(sender, Msg MSG_OFFER price)])
                              | _ ⇒ (s,[]))"

lemma merc_req_valid:
"spec_valid merc_req_spec merc_req"

definition merc_buy_spec :: "acval acspec"
where
"merc_buy_spec ≡
```

```
  (|precond = λL self par S. ∃price. par = price_N,
    postcond = λL self par S. True,
    emits = λL self par S. [] |)"

fun merc_buy :: "acval handler"
where
"merc_buy self price_N s =
  (case s v_price of Some price'_N ⇒
     (if price = price' then (s,[]) else (s,[]))
                              | _ ⇒ (s,[]))"

lemma merc_buy_valid:
"spec_valid merc_buy_spec merc_buy"

definition merc_class where
"merc_class ≡ ((λx. emptyHnd)(MSG_INIT:=merc_init,
                              MSG_REQ_OFFER:=merc_req,
                              MSG_BUY:=merc_buy))"
```

2.2.3 Composition and Security Properties

We create a system configuration that consists of three merchants and one customer. We make the customer aware of the merchants.

```
definition MERC_PRICE :: "nat" where "MERC_PRICE ≡ 100"

definition static_state :: "acval gstate" where
"static_state ≡
  empty(0↦mkActor merc_class [Msg MSG_INIT MERC_PRICE_N] [] empty,
        1↦mkActor merc_class [Msg MSG_INIT MERC_PRICE_N] [] empty,
        2↦mkActor merc_class [Msg MSG_INIT MERC_PRICE_N] [] empty,
        3↦mkActor cust_class [Msg MSG_INIT [0_N,1_N,2_N]_L] [] empty)"

definition static_cfg :: "acval syscfg"  where
"static_cfg ≡
   (|init = static_state, canCreate = (λstate aclass. False)|)"
```

As our actors have trace semantics we can use the Views and BSPs from the MAKS framwork to specify security properties on the actor level, too.

View for the complete system:

```
definition C_omc :: "actorid ⇒ actorid ⇒ (acval) aevent set" where
"C_omc ma ca ≡
  {x. ∃a price m. x = mkAEvent m ca (Msg MSG_OFFER (a_N,price_N)_T) ∧
                  m ≠ ma ∧ m ≠ ca}"

fun V_om :: "actorid ⇒ acval aevent set" where
"V_om ma = {x. ∃r msg v. x = mkAEvent ma r (Msg msg v)}"

definition ν_g :: "actorid ⇒ actorid ⇒ acval aevent View" ("ν_g") where
```

```
"ν_g ma ca ≡ (|V=V_om ma, N=UNIV - (V_om ma ∪ C_omc ma ca),C=C_omc ma ca|)"

lemma νgvalid:
"valid_View (ν_g ma ca) UNIV"
```

View for customer agents:

```
definition C_ca :: "actorid ⇒ actorid ⇒ acval aevent set" where
"C_ca ma ca ≡
  {x. ∃a price m. x = mkAEvent m ca (Msg MSG_OFFER (a_N,price_N)_T) ∧
                  m ≠ ma ∧ m ≠ ca}
∪ {x. ∃a price m. x = mkAEvent ca m (Msg MSG_BUY (a_N,price_N)_T) ∧
                  m ≠ ma ∧ m ≠ ca}"

definition N_ca :: "actorid ⇒ acval aevent set" where
"N_ca ca ≡ {x. ∃msg v. x = mkAEvent ca ca (Msg msg v)}"

definition E_ca :: "actorid ⇒ acval aevent set" where
"E_ca ca ≡ {x. ∃r msg v. x = mkAEvent ca r (Msg msg v)}"

fun ν_ca :: "actorid ⇒ actorid ⇒ acval aevent View" ("ν_ca") where
"ν_ca ma ca = (|V=UNIV - (N_ca ca ∪ C_ca ma ca),
              N=N_ca ca, C=C_ca ma ca|)"

lemma νcavalid:
"valid_View (ν_ca ma ca) UNIV"
```

View for merchant agents:

```
definition C_ma :: "actorid ⇒ actorid ⇒ acval aevent set" where
"C_ma ma ca ≡
  {x. ∃a price. x = mkAEvent ma ca (Msg MSG_OFFER (a_N,price_N)_T) ∧
                 ma ≠ ca}
∪ {x. ∃a price. x = mkAEvent ca ma (Msg MSG_BUY (a_N,price_N)_T) ∧
                 ma ≠ ca}"

definition N_ma :: "actorid ⇒ acval aevent set" where
"N_ma ma ≡ {x. ∃msg v. x = mkAEvent ma ma (Msg msg v)}"

fun ν_ma :: "actorid ⇒ actorid ⇒ acval aevent View" ("ν_ma") where
"ν_ma ma ca = (|V=UNIV - (N_ma ma ∪ C_ma ma ca),
              N=N_ma ma, C=C_ma ma ca|)"

lemma νmavalid:
"valid_View (ν_ma ma ca) UNIV"

end
```

2.3 Implementation

We want to have an implementation that resembles actor oriented programming. We have multiple possibilities to accomplish that. We can use JCoBox [6] an extension for Java that allows to designate classes as actors and adds syntax and semantics for an asynchronous method call. We can also use RMI to emphasize the distributed aspect of actors.

Finally we decided for pure Java code. This means that we have to emulate asynchronous message calls and do not get real distributed actors at all but it has an important advantage: we are now able to examine the programm with the KeY tool [1]. We think that the advantage of being able to use formal methods on the implementation level, too, outweighs potential difficulties.

```
class Offer {

  private int price;
  private Merchant source;

  public Offer(int price, Merchant source) {
    this.price = price;
    this.source = source;
  }

  public int getPrice() {
    return price;
  }

  public Merchant getSource() {
    return source;
  }
}


class Merchant {
  private String name;
  private int price;
  private boolean silent = false;

  Merchant(String name, int price) {
    this.name = name;
    this.price = price;
  }

  /*@ requires True
    @ ensures True
    @ emits {}
    @*/
    void buy(Customer customer, Offer offer) {
    if (!silent) {
```

```
      System.out.println(String.format("%s␣buys␣ITEM␣for␣%d.\n",
          this.name,offer.getPrice()));
    }
  }

  /*@ requires True
    @ ensures True
    @ emits customer!addOffer(
    @*/
  void requestOffer(Customer customer) {
    if (!silent)
      System.out.println(String.format("%s␣offers␣ITEM.", this.name));
    customer.addOffer(new Offer(price, this));
  }

  void setPrice(int price) {
    this.price = price;
  }

  void setSilent(boolean silent) {
    this.silent = silent;
  }

}

class Customer {
  static enum State { WORKING, SEARCHING};

  private String name;
  private Merchant[] merchants;
  private int merch_idx;
  private State state;
  private Offer bestOffer;
  private int counter;

  /*@ requires True
    @ ensures merchants == this.merchants && merch_idx == 0
    @ emits {}
    @*/
  Customer(String name, Merchant[] merchants) {
    this.name = name;
    this.merchants = merchants;
    this.state = State.WORKING;
    merch_idx = 0;
  }

  /*@ requires X == \par && Y = state && Z == bestOffer
    @ ensures if (X == State.SEARCHING && Z != null &&
                  X.getPrice() < Z.getPrice)
```

```
            then bestOffer == X else True
  @ emits {}
  @*/
void addOffer(Offer offer) {
  System.out.println(state);
  if (state == State.SEARCHING) {
    System.out.println(String.format("%s␣got␣offer␣for␣%d.",
                                     this.name, offer.getPrice()));
    if (bestOffer == null || offer.getPrice() < bestOffer.getPrice()) {
      bestOffer = offer;
    }
  }
}

/*@ requires X = merchants && Y = merch_idx
  @ ensures if (Y >= X.length) then True
                               else state = State.SEARCHING
  @ emits if (Y >= X.length) then {system!startTimer(this)}
                             else {this!send}
  @*/
void start() {
  if (merch_idx >= merchants.length) {
    VMall.system.startTimer(this);
  } else {
    state = State.SEARCHING;
    this.send();
  }
}

/*@ requires X = state
  @ ensures if (X == State.SEARCHING)
            then state = State.Working else true
  @ emits if (X == State.SEARCHING)
          then {this!findBest()} else {}
  @*/
void onTimer() {
  if (state == State.SEARCHING) {
    state = State.WORKING;
    this.findBest();
  }
}

/*@ requires X = merchants && Y = merch_idx
  @ ensures True
  @ emits {X[Y]!requestOffer(this),this!start()}
  @*/
void send() {
  merchants[merch_idx].requestOffer(this);
  merch_idx++;
```

```
    this.start();
  }

  /*@ requires X = bestOffer
    @ ensures true
    @ emits if (X != null) then {X.getSource()!buy(this,bestOffer)} else {}
    @*/
  void findBest() {
    if (bestOffer == null) {
      System.out.println(this.name + "␣got␣no␣offer␣for␣ITEM.");
    } else {
      bestOffer.getSource().buy(this,bestOffer);
    }
  }

}

class MSystem {
  void startTimer(Customer cust) {
    cust.onTimer();
  }
}

public class VMall {

  static MSystem system = new MSystem();

  public static void main(String[] args)
  {
    System.out.println("Start␣main");
    Merchant m1 = new Merchant("m1", 100);
    Merchant m2 = new Merchant("m2", 200);
    Customer c1 = new Customer("c1",new Merchant[]{m1,m2});
    c1.start();
    System.out.println("Stop␣main");
  }
}
```

3 E-Voting

This theory models the e-voting system that was developed by the "information security and cryptography" group by Prof.-Küsters (University of Trier).

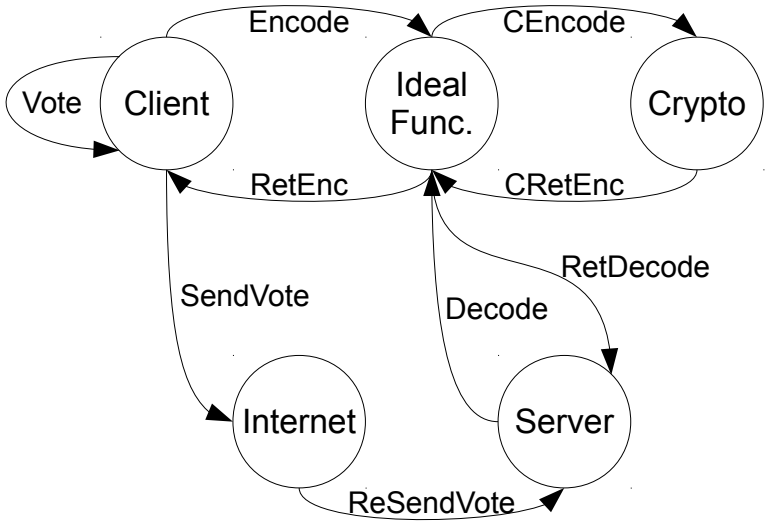

Figure 1: Model of the e-voting system

In fig. 1 we see the structure of the system: all agents and their exchanged messages.

```
theory EVoting imports XView "I-MAKS"
begin

notation Some ("_!" [1000] 1000)
```

In this model one can only vote for three alternatives that are modeled with the type *vote*.

```
datatype vote = V1 | V2 | V3

type_synonym clientid = "nat"


definition c10 :: clientid where "c10 ≡ 0"
definition v0 :: vote where "v0 ≡ V1"
```

The datatype *event* models all events (with data) that can occur in the system (c.f. fig. 1).

```
datatype event = DecideVote "clientid" "vote"
               | Enc "clientid" "vote"
               | RetEnc "clientid" "string"
               | Dec "string"
```

```
| RetDec "clientid" "vote"
| CEnc "clientid" "vote"
| CRetEnc "clientid" "string"
| SendVote "string"
| ResendVote "string"
```

3.1 Client

The client decides on how to vote then asks the the ideal functionality to encode his vote. It waits for the response and sends that to the internet (in the hope that it will be delivered to the server).

```
datatype clstate = ClInit | Decided "vote" | ClWaiting | Encrypted "string"
| ClTerm

definition client_in :: "clientid ⇒ event set"
where
"client_in cid ≡ {e. ∃s. e = RetEnc cid s}"

definition client_out :: "clientid ⇒ event set"
where
"client_out cid ≡ {e. ∃v. e = Enc cid v} ∪ {e. ∃s. e = SendVote s}"

fun client_f :: "clientid ⇒ clstate ⇒ event ⇀ clstate"
where
  "client_f cid ClInit (DecideVote cid' v) =
                (if cid = cid' then Some (Decided v) else None)"
| "client_f cid (Decided v) (Enc cid' v') =
                (if cid = cid' ∧ v = v' then Some ClWaiting else None)"
| "client_f cid Waiting (RetEnc cid' s) =
                (if cid = cid' then (Encrypted s)¦ else None)"
| "client_f cid (Encrypted s) (SendVote s') =
                (if s = s' then Some (ClTerm) else None)"
| "client_f cid _ _ = None"

definition client_rec :: "clientid ⇒ (event, clstate) SES_record"
where
"client_rec cid = (| S_SES = UNIV,
                     s0_SES = ClInit,
                     E_SES = client_out cid ∪ client_in cid ∪
                             {e. ∃v. e = DecideVote cid v},
                     I_SES = client_in cid,
                     O_SES = client_out cid,
                     T_SES = client_f cid |)"

interpretation clientn : StateEventSystem "client_rec n"
```

3.2 Ideal Functionality

In its initial state the ideal functionality waits for encoding or decoding requests. When it gets the request for an encoding it asks the cryptographic component to encode some constant vote *v0* for a constant client *c0*. It then returns the result of that encoding but saves that result together with the original vote and client id.

If the ideal functionality receives a request to decode some message it tries to look it up in its internal state. If it finds something it returns the decoded vote and client id else it does not do anything.

```
datatype ifstate =
  IFListen "(clientid × vote × string) list"
| IFSending "clientid" "vote" "((clientid × vote × string) list)"
| IFWaiting "clientid" "vote" "((clientid × vote × string) list)"
| IFReplying "clientid" "vote" "((clientid × vote × string) list)"
| IFDecReplying "string" "(clientid × vote × string) list"

definition if_in :: "event set"
where
"if_in ≡ {e. ∃cid v. e = Enc cid v} ∪ {e. ∃cid s. e = CRetEnc cid s}
       ∪ {e. ∃s. e = Dec s}"

definition if_out :: "event set"
where
"if_out ≡ {e. ∃cid v. e = CEnc cid v} ∪ {e. ∃cid s. e = RetEnc cid s}
        ∪ {e. ∃cid v. e = RetDec cid v}"

fun if_f :: "ifstate ⇒ event ⇀ ifstate"
where
  "if_f (IFListen l) (Enc cid v) = (IFSending cid v l)¡"
| "if_f (IFSending cid v l) (CEnc cid' v') =
              (if cid' = cl0 ∧ v = v0 then (IFWaiting cid v l)¡ else None)"
| "if_f (IFWaiting cid v l) (CRetEnc cid' s) =
              (if cid = cid'
                 then (IFReplying cid v ((cid,v,s)#l))¡ else None)"
| "if_f (IFReplying cid v l) (RetEnc cid' s) =
              (if cid = cid' then (IFListen l)¡ else None)"
| "if_f (IFListen l) (Dec s) = (IFDecReplying s l)¡"
| "if_f (IFDecReplying s l) (RetDec cid v) =
           (case filter {(cid',v',s'). s' = s ∧ v = v' ∧ cid = cid'} l
of
                          [] ⇒ None
                    | (x#xs) ⇒ (IFListen l)¡)"
| "if_f _ _ = None"

definition if_rec :: "(event, ifstate) SES_record"
where
"if_rec = (| S_SES = UNIV, s0_SES = IFListen [], E_SES = if_in ∪ if_out,
             I_SES = if_in, O_SES = if_out, T_SES = if_f |)"
```

interpretation *idealf : StateEventSystem "if_rec"*

3.3 Cryptographic Component

The cryptographic component waits for encoding requests and upon such requests returns a random string.

datatype *crstate = CRListen | CRWorking "clientid" "vote"*

definition *crypto_in :: "event set"*
where
"crypto_in ≡ {e. ∃c v. e = CEnc c v}"

definition *crypto_out :: "event set"*
where
"crypto_out ≡ {e. ∃c s. e = CRetEnc c s}"

fun *crypto_f :: "crstate ⇒ event ⇒ crstate option"*
where
"crypto_f CRListen (CEnc c v) = (CRWorking c v)¡"
| "crypto_f (CRWorking cid v) (CRetEnc cid' s) =
(if cid = cid' then CRListen¡ else None)"
| "crypto_f _ _ = None"

definition *crypto_rec :: "(event,crstate) SES_record"*
where
"crypto_rec = (| S_SES = UNIV, s0_SES = CRListen, E_SES = crypto_in ∪ crypto_out,
I_SES = crypto_in, O_SES = crypto_out, T_SES = crypto_f
|)"

interpretation *crypto : StateEventSystem "crypto_rec"*

3.4 Internet

In this scenario we assume that the attacker has control over the internet. We model this by making the internet's behaviour completely nondeterministic.

type_synonym *inetstate = unit*

definition *inet_in :: "event set"*
where
"inet_in ≡ {e. ∃s. e = SendVote s}"

definition *inet_out :: "event set"*
where
"inet_out ≡ {e. ∃s. e = ResendVote s}"

```
fun inet_f :: "inetstate ⇒ event ⇀ inetstate"
where
  "inet_f _ (SendVote _) = ()!"
| "inet_f _ (ResendVote _) = ()!"
| "inet_f _ _ = None"

definition inet_rec :: "(event,inetstate) SES_record"
where
"inet_rec = (| S_SES = UNIV, s0_SES = (), E_SES = inet_in ∪ inet_out,
               I_SES = inet_in, O_SES = inet_out, T_SES = inet_f |)"

interpretation inet : StateEventSystem "inet_rec"
```

3.5 Server

The server listens for new (encrypted) votes. It sends the votes to the ideal functionality. As soon as it gets the decrypted reply it counts the vote.

```
datatype srvstate = Listen "vote list" "clientid list"
                  | Sending "vote list" "clientid list" "string"
                  | Waiting "vote list" "clientid list"

definition srv_in :: "event set"
where
"srv_in ≡ {e. ∃s. e = ResendVote s} ∪ {e. ∃cid v. e = RetDec cid v}"

definition srv_out :: "event set"
where
"srv_out ≡ {e. ∃s. e = Dec s}"

fun srv_f :: "srvstate ⇒ event ⇀ srvstate"
where
  "srv_f (Listen vl cl) (ResendVote s) = (Sending vl cl s)!"
| "srv_f (Sending vl cl s) (Dec s') =
                   (if s = s' then (Waiting vl cl)! else None)"
| "srv_f (Waiting vl cl) (RetDec cid v) = (Listen (v#vl) (cid#cl))!"
| "srv_f _ _ = None"

definition srv_rec :: "(event,srvstate) SES_record"
where
"srv_rec = (| S_SES = UNIV, s0_SES = Listen [] [], E_SES = srv_in ∪ srv_out,
              I_SES = srv_in, O_SES = srv_out, T_SES = srv_f |)"

interpretation server : StateEventSystem "srv_rec"
```

3.6 Composition

Finally we have to show that we can compose the agents defined above, i.e. the composed system is a state event system, too.

interpretation *clif_ES : ComposedEventSystem*
"clientn.induced_ES 0" "idealf.induced_ES"

interpretation *clifcr_ES : ComposedEventSystem*
"clientn.induced_ES 0 ‖ idealf.induced_ES" "crypto.induced_ES"

interpretation *clifcrsrv_ES : ComposedEventSystem*
"(clientn.induced_ES 0 ‖ idealf.induced_ES) ‖ crypto.induced_ES"
"server.induced_ES"

interpretation *complete_ES : ComposedEventSystem*
"((clientn.induced_ES 0 ‖ idealf.induced_ES) ‖ crypto.induced_ES)
‖ server.induced_ES"
"inet.induced_ES"

end

4 Conclusions

These examples can be used to study the relation of the three layers: agent-, actor- and implementation-layer. This is especially important because we do not only want to research interesting formal/theoretical results (c.f. [2]) but we also want to be able to apply these result in concrete scenarios like these examples.

References

[1] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach.* LNCS 4334. Springer-Verlag, 2007.

[2] Christoph Feller. Formal investigation of refinement steps and security properties in distributed systems. Technical report, University of Kaiserslautern, March 2012.

[3] Dieter Hutter, Heiko Mantel, Ina Schaefer, and Axel Schairer. Security of multi-agent systems: A case study on comparison shopping. *Journal of Applied Logic*, 5(2):303 – 332, 2007. Logic-Based Agent Verification.

[4] Jan Jürjens. *Secure Systems Development with UML.* Springer Berlin / Heidelberg, 2005.

[5] Heiko Mantel. *A Uniform Framework for the Formal Specification and Verification of Information Flow Security.* PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, Juli 2003.

[6] Jan Schäfer and Arnd Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *24th European Conference on Object-Oriented Programming (ECOOP 2010)*, LNCS. Springer, June 2010.