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 which 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 additional 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 ("initc_" [1000] 1000) where
  "Initc n ≡ Init (CID n)"

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

abbreviation Sendc ("sendc_" [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)
fun ′v agval ⇒ nat ("theN") where
"theN n = n"

fun ′v agval ⇒ ′v agval ("fstT") where
"fstT (x,y)T = x"

fun ′v agval ⇒ ′v agval ("sndT") where
"sndT (x,y)T = y"

fun ′v agval ⇒ ′v agval ("_ #L _") where
"x #L xsL = (x # xs)L"

fun ′v agval ⇒ ′v agval ("hdL") where
"hdL (a#as)L = a"

fun ′v agval ⇒ ′v agval ("tlL") where
"tlL (a#as)L = asL"

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

abbreviation AgentID’ ("_A" [1000] 1000) where
"xA ≡ 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

3
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 \( "MSG_{OFFER} \equiv 0" \)
**definition** \( MSG_{REQ\_OFFER} \) where \( "MSG_{REQ\_OFFER} \equiv 1" \)
**definition** \( MSG_{BUY} \) where \( "MSG_{BUY} \equiv 2" \)

**definition** \( ca_{\tau evt} :: \"nat \Rightarrow (agentid, val agval) event set\" \)
where \( "ca_{\tau evt} n \equiv let \ a = CID n in \ \{ \tau_a \ 1, \ \tau_a \ 2, \ \tau_a \ 3, \ \tau_a \ 4, \ \tau_a \ 5 \} \" \)

**definition** \( ca_{in\_evt} :: \"nat \Rightarrow (agentid, val agval) event set\" \)
where \( "ca_{in\_evt} n \equiv let \ a = CID n in \ \{ e. \ 3msg. \ init_a \ msg = e \} \bigcup \ \{ \text{start}_a \} \bigcup \ \{ e. \ 3b \ p. \ \text{recv}_a \ b \ (Msg \ MSG_{OFFER} \ (b_A, p_N)T) = e \land b \neq a \} \" \)

**definition** \( ca_{out\_evt} :: \"nat \Rightarrow (agentid, val agval) event set\" \)
where \( "ca_{out\_evt} n \equiv let \ a = CID n in \ \{ e. \ 3b. \ \text{send}_a \ b \ (Msg \ MSG_{REQ\_OFFER} \ (0_N)) = e \land b \neq a \} \bigcup \ \{ e. \ 3b \ p. \ \text{send}_a \ b \ (Msg \ MSG_{BUY} \ (b_A, p_N)T) = e \land b \neq a \} \" \)

**definition** \( ca_{evt} :: \"nat \Rightarrow (agentid, val agval) event set\" \)
where \( "ca_{evt} n \equiv ca_{\tau evt} n \bigcup ca_{in\_evt} n \bigcup ca_{out\_evt} n \" \)

**definition** \( v_{cmas} \) where \( "v_{cmas} \equiv Var \ 2" \)
**definition** \( v_{mas} \) where \( "v_{mas} \equiv Var \ 3" \)
**definition** \( v_{ma} \) where \( "v_{ma} \equiv Var \ 4" \)
**definition** \( v_{os} \) where \( "v_{os} \equiv Var \ 5" \)

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** \( T_{ca} :: \"nat \Rightarrow (val agval agstate \times (agentid, val agval) event) \times val agval agstate\" \)
**set**
for \( n :: \"nat\" \)
where
\[ T_{init}:: [m \ pc = NN \ 0; \ m \ run = BB \ False; \ m' = m(pc:=1_N); \ a = CID n] \]
\[ \implies (m, \ init_a \ msg, m') \in T_{ca} n \]
| \[ T_{start}:: [m \ pc = NN \ 1; \ m \ run = BB \ False; \ m' = m(run:=True_B, \ pc:=2_N); \ a = CID n] \]
\[ \implies (m, \ start_a, m') \in T_{ca} n \]
| T_copy: "\[ \text{m pc = NN 2; m run = BB True; m' = m(pc:=NN 3); m v_mas = (x_A#xs)_L; a = CID n} \] \implies (m, \tau_a 1, m') \in T_{ca} n" |
| T_selM: "\[ \text{m pc = NN 3; m run = BB True; m' = m(v_mas:=xs_L, pc:=4N); m v_mas = (x_A#xs)_L; m' v_ma = x_A; a = CID n} \] \implies (m, \tau_a 2, m') \in T_{ca} n" |
| T_req0: "\[ \text{m pc = NN 4; m run = BB True; m' = m(pc:=NN 2); m v_mas = (x_A#xs)_L; a = CID n} \] \implies (m, \tau_a 3, m') \in T_{ca} n" |
| T_noreq: "\[ \text{m pc = NN 2; m run = BB True; m' = m(pc:=NN 5); m v_mas = [ ]; a = CID n} \] \implies (m, \tau_a 4, m') \in T_{ca} n" |
| T_best: "\[ \text{m pc = NN 6; m run = BB True; m' = m(pc:=NN 7); m v_os = [(ma_A,price_N)_T]_L; a \neq ma; a = CID n} \] \implies (m, \tau_a 5, m') \in T_{ca} n" |
| T_buy: "\[ \text{m pc = NN 7; m run = BB True; m' = m(pc:=NN 8); m v_os = [(ma_A,price_N)_T]_L; a \neq ma; a = CID n} \] \implies (m, send_a ma (Msg MSG_BUY (ma_A,price_N)_T), m') \in T_{ca} n" |

inductive_cases T_initE[elim]: "(m,init_a msg,m') \in T_{ca} n" |
inductive_cases T_startE[elim]: "(m,start_a a,m') \in T_{ca} n" |
inductive_cases T_copyE[elim]: "(m,\tau_a 1,m') \in T_{ca} n" |
inductive_cases T_selME[elim]: "(m,\tau_a 2,m') \in T_{ca} n" |
inductive_cases T_req0E[elim]: "(m,send_a b (Msg MSG_REQ_OFFER (NN 0)),m') \in T_{ca} n" |
inductive_cases T_noreqE[elim]: "(m,\tau_a 3,m') \in T_{ca} n" |
inductive_cases T_rcvOE[elim]: "(m,recv_a ma (Msg MSG_OFFER (ma_A,price_N)_T),m') \in T_{ca} n" |
inductive_cases T_readyE[elim]: "(m,\tau_a 4,m') \in T_{ca} n" |
inductive_cases T_bestE[elim]: "(m,\tau_a 5,m') \in T_{ca} n" |
inductive_cases T_buyE[elim]: "(m, send_a ma (Msg MSG_BUY (ma_A,price_N)_T), m') \in T_{ca} n" |

lemma T_ca_fun[intro]: "[(m,e,m') \in T_{ca} n; (m,e,m'') \in T_{ca} n] \implies m' = m''" |

fun f_ca :: "nat \Rightarrow val agval agstate \Rightarrow (agentid, val agval) event \Rightarrow val agval agstate" where |
"f_ca n s e = (if (\exists s'. (s,e,s') \in T_{ca} n) then Some (SOME s'. (s,e,s') \in T_{ca} n) else None)" |

definition cust_agent :: "nat \Rightarrow ((agentid,val agval) event, ..."
val agval agstate) SES_record" where

"cust_agent n = (S_SES=UNIV,
  s0_SES=(λx. (0N)) (v_ma := [(MID 0)A, (MID 1)A]L, run:=False),
  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. 3msg. init_a msg = e} ∪ (start_a) ∪
  {e. 3b. recv_a b (Msg MSG_REQ_OFFER 0N) = e ∧ b ≠ a} ∪
  {e. 3b p. recv_a b (Msg MSG_BUY (aA,pN)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. 3b p. send_a b (Msg MSG_OFFER (aA,pN)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:=agsL);
          m v_req = (agA#ags)L; m v_of = (aA,priceN)T; a ≠ ag;
          a = MID n"
\[
\Rightarrow (m, \text{send}_a \text{ ag} (\text{Msg MSG\_OFFER } (a_A, \text{priceN})_T), m') \in T_{\text{ma}} n \\
\]

| T_rreq: "[m pc = NN 2; m run = BB True; m' = m(v\_req:= (ag_A\#ags)_L); 
\quad m v\_req = ags_L; a \neq ag; a = MID n] 
\Rightarrow (m, \text{recv}_a \text{ ag} (\text{Msg MSG\_REQ\_OFFER } 0_N), m') \in T_{\text{ma}} n" \\
| T_rbuy: "[m pc = NN 2; m run = BB True; m v\_of = (a_A, \text{priceN})_T; a \neq b; 
\quad a = MID n] 
\Rightarrow (m, \text{recv}_a \text{ b} (\text{Msg MSG\_BUY } (a_A, \text{priceN})_T), m) \in T_{\text{ma}} n"

inductive cases T_initE2[elim]: "(m, \text{init}_a \text{ msg}, m') \in T_{\text{ma}} n"
inductive cases T_startE2[elim]: "(m, \text{start}_a, m') \in T_{\text{ma}} n"
inductive cases T_sendE[elim]: 
"(m, \text{send}_a \text{ b } (\text{Msg MSG\_OFFER } (a_A, \text{priceN})_T), m') \in T_{\text{ma}} n"
inductive cases T_rreqE[elim]: 
"(m, \text{recv}_a \text{ b } (\text{Msg MSG\_REQ\_OFFER } 0_N), m') \in T_{\text{ma}} n"
inductive cases T_rbuyE[elim]: 
"(m, \text{recv}_a \text{ b } (\text{Msg MSG\_BUY } (a_A, \text{priceN})_T), m') \in T_{\text{ma}} n"

lemma T_ma_inj[intro]: 
"[(m, e, m') \in T_{\text{ma}} n; (m, e, m'') \in T_{\text{ma}} n] \Rightarrow m' = m''"

fun f_ma :: "nat \Rightarrow \text{val agval agstate} \Rightarrow (\text{agentid}, \text{val agval}) \text{ event} 
\Rightarrow \text{val agval agstate}" where
"f_{\text{ma}} n s e = (if (\exists s'. (s, e, s') \in T_{\text{ma}} n) 
\quad \text{then Some (SOME s'. (s, e, s') \in T_{\text{ma}} n) else None})"

definition merch_agent :: "nat \Rightarrow ((\text{agentid},\text{val agval}) \text{ event}, 
\text{val agval agstate}) \text{ SES\_record}" where
"merch_agent n = (\{\text{S\_SES=UNIV,} 
\quad \text{s0\_SES=} (\lambda x. (\text{NN 0})) (\text{run:=BB False, pc:=0_N, v\_of:=}((\text{MID n})_A, \text{100N})_T), 
\quad \text{E\_SES=ma\_evt n, I\_SES=ma\_in\_evt n, O\_SES=ma\_out\_evt n, T\_SES=f_{\text{ma}} n\})"

interpretation ma: StateEventSystem "merch_agent n"

2.1.3 Platform

definition pl_tau_evt :: "(\text{agentid},\text{val agval}) \text{ event set}" 
\where "pl\_tau\_evt \equiv \{\}

definition pl_in_evt :: "(\text{agentid},\text{val agval}) \text{ event set}" 
\where "pl\_in\_evt \equiv \{e. \exists a b msg. \text{send}_a b msg = e \land b \neq a\}"

definition pl_out_evt :: "(\text{agentid},\text{val agval}) \text{ event set}" 
\where "pl\_out\_evt \equiv \{e. \exists a b msg. \text{recv}_a b msg = e \land b \neq a\}"

definition pl_evt :: "(\text{agentid},\text{val agval}) \text{ 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 = (|$SES=UNIV, s0_SES=(λ x. (0 N))(run:=False),
  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 system.

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 $C_{omc} :: \text{agentid} \Rightarrow \text{agentid} \Rightarrow (\text{agentid}, \text{val agval}) \text{ event set}$
where
$C_{omc} \text{ ma ca} \equiv$
$(\{x. \exists \text{price } m. x = \text{send}_m \text{ ca (Msg MSG_OFFER (m_A,price_N)_T)} \land m \neq \text{ma}\})$
fun $V_{om} :: \text{agentid} \Rightarrow (\text{agentid}, \text{val agval}) \text{ event set}$
where
$V_{om} (\text{MID ma}) = \text{ma_evt ma}$

definition $\nu_g :: \text{agentid} \Rightarrow \text{agentid} \Rightarrow (\text{agentid}, \text{val agval}) \text{ event View}$
($\nu_g$)
where
$\nu_g \text{ ma ca} \equiv (|V=V_{om} \text{ ma}, N=\text{UNIV} - (V_{om} \text{ ma} \cup C_{omc} \text{ ma ca}),$
$C=C_{omc} \text{ ma ca}|)$

The view for customer agents:
definition $C_{ca} :: \text{agentid} \Rightarrow \text{agentid} \Rightarrow (\text{agentid}, \text{val agval}) \text{ event set}$
where
$C_{ca} \text{ ma ca} \equiv$
$(\{x. \exists \text{a } \text{price } m. x = \text{recv}_m \text{ ca (Msg MSG_OFFER (a_A,price_N)_T)} \land m \neq \text{ma}\}$
$\cup (\{x. \exists \text{a } \text{price } m. x = \text{send}_m \text{ ca (Msg MSG_BUY (a_A,price_N)_T)} \land m \neq \text{ma}\})$
fun $\nu_{ca} :: \text{agentid} \Rightarrow \text{agentid} \Rightarrow (\text{agentid}, \text{val agval}) \text{ event View"} (\nu_{ca})$
where
$\nu_{ca} \text{ ma (CID ca)} = (|V=\text{ca_evt ca} - (\text{ca_tau_evt ca} \cup C_{ca} \text{ ma (CID ca)}),$n$C=C_{ca} \text{ ma (CID ca)}|)$

The view for merchant agents:
definition $C_{ma} :: \text{agentid} \Rightarrow \text{agentid} \Rightarrow (\text{agentid}, \text{val agval}) \text{ event set}$
where
$C_{ma} \text{ ma ca} \equiv$
$(\{x. \exists \text{a } \text{price } m. x = \text{send}_m \text{ ca (Msg MSG_OFFER (a_A,price_N)_T)} \land m \neq \text{ma}\}$
$\cup (\{x. \exists \text{a } \text{price } m. x = \text{recv}_m \text{ ca (Msg MSG_BUY (a_A,price_N)_T)} \land m \neq \text{ma}\})$
fun $\nu_{ma} :: \text{agentid} \Rightarrow \text{agentid} \Rightarrow (\text{agentid}, \text{val agval}) \text{ event View"} (\nu_{ma})$
where
$\nu_{ma} \text{ ma (CID ca)} = (|V=\text{ca_evt ca} - (\text{ca_tau_evt ca} \cup C_{ca} \text{ ma (CID ca)}),$n$N=\text{ca_evt ca}, C=C_{ma} \text{ ma (CID ca)}|)$
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:

\textbf{type synonym} actorid = "nat"

The following type describes the internal state of an actor.

\textbf{type synonym} 'v acstate = "var \rightarrow 'v"

We need two special variables for assertions.

\textbf{definition} v_par :: "var" where "v_par = Var 2"

\textbf{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.

\textbf{type synonym} 'v handlerses = "'v acstate \times (actorid \times 'v msg) list"

\textbf{type synonym} 'v handler = "actorid \Rightarrow 'v \Rightarrow 'v acstate \Rightarrow 'v handlerses"

\textbf{type synonym} 'v actorclass = "msgid \Rightarrow 'v handler"

'v handlerses 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 implement-
ation 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"
    aerect :: "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 ⇒
```
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**

```plaintext
type synonym 'v spec_env = "var ⇒ 'v"
```

**Record**

```plaintext
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**

```plaintext
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**

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

**Notation**

```plaintext
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**: 
  
  \[
  \begin{array}{l}
  g \text{ aid1} = \text{Some ac1}; \ g \text{ aid2} = \text{Some ac2}; \\
  \text{out_msgs ac1} = \text{om1} @ \text{amsg} \# \text{om2}; \\
  \text{amsg} = (\text{DstActor aid2, msg}); \\
  g' = (g(\text{aid1→ac1(out_msgs:=om1 @ om2)}, \text{aid2→ac2(in_msgs:=(in_msgs ac2)@[msg]})) \\
  \Rightarrow \text{actors_step cfg g [mkAEvent aid1 aid2 msg]} g' \\
  \end{array}
  \]

- **actors_step_actor**: 
  
  \[
  \begin{array}{l}
  g \text{ aid} = \text{Some ac}; \ \text{out_msgs ac} = []; \\
  \text{actor_step aid ac el ac'} \\
  \Rightarrow \text{actors_step cfg g [] (g(aid→ac'))} \\
  \end{array}
  \]

- **actors_new_actor**: 
  
  \[
  \begin{array}{l}
  \text{canCreate cfg g ac} \\
  \Rightarrow \text{actors_step cfg g [] (g(free_aid g→mkActor ac [] []) empty)} \\
  \end{array}
  \]

- **actors_step_refl**: 
  
  \[
  \begin{array}{l}
  \text{actors_step cfg g eml g'; actors_step cfg g' eml' g''} \\
  \Rightarrow \text{actors_step cfg g (eml @ eml')} g'' \\
  \end{array}
  \]

**lemma global_range_stays_finite[intro]**: 

\[
\text{actors_step cfg g eml g' } \Rightarrow \text{finite (range g) } \Rightarrow \text{finite (range g')} \\
\]

**lemma actor_ex_trans**: 

\[
\begin{array}{l}
(\exists \text{ac'}. \text{actor_step self ac ml ac'} \land (\exists \text{ac''}. \text{actor_step self ac'} [m] \text{ ac''})) \\
\Rightarrow (\exists \text{ac'}. \text{actor_step self ac (ml @ [m]) ac''}) \\
\end{array}
\]

**fun acTr :: \"\nat ⇒ \'v actor ⇒ \'v aevent list set\"**

where

\[
\text{acTr aid ac} = \{\text{tr. } \exists \text{ac'}. \text{actor_step aid ac tr ac'}\}
\]

**fun gTr :: \"\'v syscfg ⇒ \'v aevent list set\"**

where

\[
\text{gTr cfg} = \{\text{tr. } \exists \ g' . \text{actors_step cfg g tr g'}\}
\]

**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 and provide 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 = lL,
   postcond = λL self par S. (S v_mas) = Some par ∨ (S v_recv = Some False)
     ∨ (S v_boff = None) ∨ (S v_midx = Some 0N),
   emits = λL self par S. [(self, Msg MSG_SEL 0N)])"

fun cust_init :: "acval handler"
where
"cust_init self l1 s =
(s(v_boff:=None)(v_midx:=0N,v_mas:=l1,v_recv:=FalseB),
  [(self, Msg MSG_SEL 0N)])"
| "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 \( "L_x \equiv \text{Var 1}" \)

**definition** \( L_y \) where \( "L_y \equiv \text{Var 2}" \)

**definition** \( L_z \) where \( "L_z \equiv \text{Var 3}" \)

**definition** \( \text{cust_start_spec} :: \ "\text{acval acspec}\" \) where

\[
\text{cust_start_spec} \equiv \lambda L \text{ self } \text{ par } S. \\
\text{(|precond = } \lambda L \text{ self } \text{ par } S. \\
\exists ml \text{ midx. } S \text{ v_mas = Some ml} \land L L_x = ml \land \notag \\
\land S \text{ v_midx = Some midx} \land L L_y = midx, \land \notag \\
\text{postcond = } \lambda L \text{ self } \text{ par } S. \\
(\exists ml \text{ midx. } (L L_x = ml \land L L_y = midx) \land \notag \\
\text{if midx < length ml then } S \text{ v_recv = Some True} \land \notag \\
\text{else True}), \land \notag \\
\text{emits = } \lambda L \text{ self } \text{ par } S. \\
\text{(|case L L_y of midx } \Rightarrow \notag \\
\text{| case L L_x of ml } \Rightarrow \notag \\
\text{| if midx < length ml then } \{(\text{self, Msg MSG_SEND 0N})\} \notag \\
\text{| else } \{(\text{system, Msg MSG_START_TIMER selfN})\} \notag \\
\text{| _ } \Rightarrow \{\} \notag \\
\text{| _ } \Rightarrow \{\})\))
\]

**fun** \( \text{cust_start} :: \ "\text{acval handler}\" \) where

\[
\text{cust_start self _ s } = \notag \\
\text{(|case s v_mas of Some ml } \Rightarrow \notag \\
\text{| case s v_midx of Some midx } \Rightarrow \notag \\
\text{| if midx < length ml then } \{(\text{self, Msg MSG_SEND 0N})\} \notag \\
\text{| else } \{(\text{system, Msg MSG_START_TIMER selfN})\} \notag \\
\text{| _ } \Rightarrow \{(\), | _ } \Rightarrow \{s, [\})\}
\]

**lemma** \( \text{cust_start_valid:} \notag \\
\"\text{spec_valid cust_start_spec cust_start}\" \)

**definition** \( \text{cust_send_spec} :: \ "\text{acval acspec}\" \) where

\[
\text{cust_send_spec} \equiv \lambda L \text{ self } \text{ par } S. \\
\exists ml \text{ midx. } S \text{ v_mas = Some ml} \land L L_x = ml \land \notag \\
\land S \text{ v_midx = Some midx} \land L L_y = midx, \land \notag \\
\text{postcond = } \lambda L \text{ self } \text{ par } S. \text{ True,} \notag \\
\text{emits = } \lambda L \text{ self } \text{ par } S. \text{ (case L L_x of ml } \Rightarrow \notag \\
\text{| case L L_y of midx } \Rightarrow \notag \\
\text{| case ml!midx of mN } \Rightarrow \notag \\
\text{| (m, Msg MSG_REQ_OFFER selfN)}, \notag \\
\text{| (self, Msg MSG_SEL 0N)})
\]

15
fun cust_send :: "acval handler"
where "cust_send self _ s =
  (case s v_mas of Some ml \Rightarrow
   (case s v_midx of Some midx \Rightarrow
    (case ml!midx of mN \Rightarrow
     (s,[(m, Msg MSG_REQ_OFFER self_N),
        (self, Msg MSG_SEL 0_N)]))
     | _ \Rightarrow (s,[])) | _ \Rightarrow (s,[])) | _ \Rightarrow (s,[]))\)

lemma cust_send_valid:
"spec_valid cust_send_spec cust_send"

definition cust_recv_spec :: "acval acspec"
where "cust_recv_spec \equiv
  (|precond = \\lambda self par S. \exists sender price recv offs.
                  par = (sender_N,price_N)T
                 \land L L_x = (sender_N,price_N)T
                 \land S v_recv = Some recv \land L L_y = recv
                 \land S v_boff = Some offs \land L L_z = offs,
   postcond =
    \\lambda self par S. \exists sender price. L L_x = (sender_N,price_N)T \land
                 S = (case L L_y of TrueB \Rightarrow
                      (case L L_z of (b_senderN,b_priceN)T
                              \Rightarrow (if price < b_price
                                             then S(v_boff\rightarrow(b_senderN,b_priceN)T) else S)
                         | _ \Rightarrow (S(v_boff\rightarrow(b_senderN,b_priceN)T))))
                 | _ \Rightarrow S),
   emits = \\lambda self par S. [{}])\)

fun cust_recv :: "acval handler"
where "cust_recv self (senderN,priceN)T s =
  (case s v_recv of Some TrueB \Rightarrow
   (case s v_boff of Some (b_senderN,b_priceN)T
          \Rightarrow (if price < b_price
                  then s(v_boff\rightarrow(b_senderN,b_priceN)T) else s,[])
          | _ \Rightarrow (s(v_boff\rightarrow(b_senderN,b_priceN)T),[]))
          | _ \Rightarrow (s,[]))\)

lemma cust_recv_valid:
"spec_valid cust_recv_spec cust_recv"
"cust_timer_spec ≡
  (\precond = \lambda L \text{ self par } S. \text{s v_recv = Some (L L_x)},
   \text{postcond = } \lambda L \text{ self par } S.
   S = \begin{cases}
       \text{case } L \text{ L_x of True_B } \Rightarrow (S(\text{v_recv} \mapsto \text{False_B})) \\
       \_ \Rightarrow S
   \end{cases},
   \text{emits = } \lambda L \text{ self par } S.
   \begin{cases}
       \text{case } L \text{ L_x of True_B } \Rightarrow \{(\text{self, Msg MSG_FIND_BEST 0_N})\} \\
       \_ \Rightarrow [\]}
  )\)"

fun cust_timer :: "acval handler"
where
"cust_timer self _ s =
  (case s \text{ v_recv of Some True_B } \Rightarrow (s(\text{v_recv} \mapsto \text{False_B}),
        [\{(\text{self, Msg MSG_FIND_BEST 0_N})\}]
        \_ \Rightarrow [\]})\)"

lemma cust_timer_valid:
"spec_valid cust_timer_spec cust_timer"

definition cust_find_spec :: "acval acspec"
where
"cust_find_spec ≡
  (\precond = \lambda L \text{ self par } S. \text{s v_boff = Some (L L_x)},
   \text{postcond = } \lambda L \text{ self par } S. \text{True},
   \text{emits = } \lambda L \text{ self par } S.
   \begin{cases}
       \text{case } L \text{ L_x of (sender_N, price_N)_T } \Rightarrow \{(\text{sender, Msg MSG_BUY price_N})\} \\
       \_ \Rightarrow (s,[])\})\)"

fun cust_find :: "acval handler"
where
"cust_find self _ s =
  (case s \text{ v_boff of Some (sender_N, price_N)_T } \Rightarrow
    (s,[\{(\text{sender, Msg MSG_BUY price_N})\}]
    \_ \Rightarrow (s,[])\})\)"

lemma cust_find_valid:
"spec_valid cust_find_spec cust_find"

definition cust_class where
"cust_class ≡ ((\lambda x. emptyHnd)(\text{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
2.2.2 Merchant Actor

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

**definition** \( v\_price \) where "\( v\_price \equiv \text{Var} \ 2 \)"

**definition** \( \text{merc\_init\_spec} :: \ "\text{acval acspec}" \) where

"\( \text{merc\_init\_spec} \equiv \)

\[
\begin{align*}
& (| \text{precond} = \lambda L \ self \ par \ S. \ \exists \text{price}. \ par = \text{price}N, \\
& \quad \text{postcond} = \lambda L \ self \ par \ S. \ \exists \text{price}. \ par = \text{price}N \land \\
& \quad S \ v\_price = \text{Some} \ \text{price}N, \\
& \quad \text{emits} = \lambda L \ self \ par \ S. \ [\] | )
\end{align*}
\]

**fun** \( \text{merc\_init} :: \ "\text{acval handler}" \) where

"\( \text{merc\_init\ self priceN} \ s = (s(v\_price \mapsto priceN),[]) \)"

**lemma** \( \text{merc\_init\_valid} \): "\( \text{spec\_valid merc\_init\_spec merc\_init} \)"

**definition** \( \text{merc\_req\_spec} :: \ "\text{acval acspec}" \) where

"\( \text{merc\_req\_spec} \equiv \)

\[
\begin{align*}
& (| \text{precond} = \lambda L \ self \ par \ S. \ \exists \text{sender}. \ par = \text{sender}N, \\
& \quad \text{postcond} = \lambda L \ self \ par \ S. \ \text{True}, \\
& \quad \text{emits} = \lambda L \ self \ par \ S. \ \{\text{case} \ \text{par} \ \text{of} \ \text{sender}N \Rightarrow \\
& \quad (\text{case} \ S \ v\_price \ \text{of} \ \text{Some} \ \text{price} \Rightarrow \\
& \quad \ [(\text{sender}, \text{Msg MSG\_OFFER} \ \text{price})] \\
& \quad | _ \Rightarrow [\])\} \} | )
\end{align*}
\]

**fun** \( \text{merc\_req} :: \ "\text{acval handler}" \) where

"\( \text{merc\_req\ self senderN} \ s = \\
\quad (\text{case} \ s \ v\_price \ \text{of} \ \text{Some} \ \text{price} \Rightarrow (s,[(\text{sender}, \text{Msg MSG\_OFFER} \ \text{price})]) \\
\quad | _ \Rightarrow (s,[])) \)"

**lemma** \( \text{merc\_req\_valid} \): "\( \text{spec\_valid merc\_req\_spec merc\_req} \)"

**definition** \( \text{merc\_buy\_spec} :: \ "\text{acval acspec}" \) where

"\( \text{merc\_buy\_spec} \equiv \)"
fun merc_buy :: "acval handler"
where
"merc_buy self price N s =
(case s v_price of Some price' N \Rightarrow
(if price = price' then (s,[]) else (s,[]))
| _ \Rightarrow (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\mapsto\mkActor merc_class [Msg MSG_INIT MERC_PRICE N] [] empty,
1\mapsto\mkActor merc_class [Msg MSG_INIT MERC_PRICE N] [] empty,
2\mapsto\mkActor merc_class [Msg MSG_INIT MERC_PRICE N] [] empty,
3\mapsto\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 \neq ma ∧ m \neq 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\* subscripts apply to merchant agents or customer agents, N=UNIV - (V subscripts ma or ca) and C=omc ma ca subscripts apply to merchant agents or customer agents.

**Lemma νvalid:**
valid_View (ν subscripts ma ca) UNIV

**View for customer agents:**

**Definition** C subscripts ca :: "actorid \(\Rightarrow\) actorid \(\Rightarrow\) acval aevent set" where
\*C subscripts ca ma ca\* ≡ 
\{x. ∃\ a\ price\ m. x = mkAEvent m ca (Msg MSG_OFFER (aN,priceN)T) ∧
\m ≠ ma ∧ m ≠ ca}\)
∪ \{x. ∃\ a\ price\ m. x = mkAEvent ca m (Msg MSG_BUY (aN,priceN)T) ∧
\m ≠ ma ∧ m ≠ ca}\)

**Definition** N subscripts ca :: "actorid \(\Rightarrow\) acval aevent set" where
\*N subscripts ca ca\* ≡ \{x. ∃\ msg\ v. x = mkAEvent ca ca (Msg msg v)\}

**Definition** E subscripts ca :: "actorid \(\Rightarrow\) acval aevent set" where
\*E subscripts ca ca\* ≡ \{x. ∃\ r\ msg\ v. x = mkAEvent ca r (Msg msg v)\}

**Fun** ν subscripts ca :: "actorid \(\Rightarrow\) actorid \(\Rightarrow\) acval aevent View" ("ν subscripts ca") where
\*ν subscripts ca ma ca\* = \{V=UNIV - (N subscripts ca ca ∪ C subscripts ca ma ca),
N=N subscripts ca ca, C=C subscripts ca ma ca\}

**Lemma νcavalid:**
valid_View (ν subscripts ca ma ca) UNIV

---

**View for merchant agents:**

**Definition** C subscripts ma :: "actorid \(\Rightarrow\) actorid \(\Rightarrow\) acval aevent set" where
\*C subscripts ma ma ca\* ≡ 
\{x. ∃\ a\ price. x = mkAEvent ma ca (Msg MSG_OFFER (aN,priceN)T) ∧
\ma ≠ ma ∧ \m ≠ ca\}
∪ \{x. ∃\ a\ price. x = mkAEvent ca ma (Msg MSG_BUY (aN,priceN)T) ∧
\ma ≠ ma ∧ \m ≠ ca\}

**Definition** N subscripts ma :: "actorid \(\Rightarrow\) acval aevent set" where
\*N subscripts ma ma\* ≡ \{x. ∃\ msg\ v. x = mkAEvent ma ma (Msg msg v)\}

**Fun** ν subscripts ma :: "actorid \(\Rightarrow\) actorid \(\Rightarrow\) acval aevent View" ("ν subscripts ma") where
\*ν subscripts ma ma ca\* = \{V=UNIV - (N subscripts ma ma ∪ C subscripts ma ca),
N=N subscripts ma ma, C=C subscripts ma ca\}

**Lemma νmavvalid:**
valid_View (ν subscripts 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 program 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.

```java
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) {
```

21
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)
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).

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 cl0 :: clientid where "cl0 ≡ 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"
```

Figure 1: Model of the e-voting system
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).

```plaintext
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"
```

26
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 \( v_0 \) for a constant client \( c_0 \). 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. \exists cid\, v. e = Enc cid v \} ∪ \{ e. \exists cid\, s. e = CRetEnc cid s \} ∪ \{ e. \exists s. e = Dec s \}"

definition if_out :: "event set"
where
"if_out ≡ \{ e. \exists cid\, v. e = CEnc cid v \} ∪ \{ e. \exists cid\, s. e = RetEnc cid s \} ∪ \{ e. \exists 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 \}"

27
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"

"inet.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


