Specifying Access Control in Event-B

Specifying Access Control in Event-B

Thai Son Hoang, David Basin, and Jean-Raymond Abrial

Department of Computer Science
Swiss Federal Institute of Technology, Zurich (ETH Zurich)

Abstract. We investigate the idea of developing access control systems in Event-B by specifying separately the "insecure" target system and the security authorisation, then combining them together in order to construct a secure system. This is based on the work by Basin et. al. [6] where the chosen language is CSP-OZ. Moreover, in order to verify the secure system against some safety temporal properties, we propose an approach of constructing several abstract models corresponding to these properties, and using refinement to prove that the final system satisfies these properties.
Keyword: Event-B, access control, security, temporal property.

1 Introduction

We investigate the idea of developing access control systems in Event-B. Our approach is based on the work of Basin et. al. [6] where the chosen language is CSP-OZ. Our first contribution is the methodology to develop this type of systems by specifying separately the insecure target system and the security authorisation, then combine them together in order to construct an secure system. When developing the insecure system, developers can concentrate on the functional requirements of the actual system without any knowledge of the security features. On the other hand, when developing the security authorisation, the developers do not need to know about the details that are irrelevant to the model of the authorisation itself. Later the combine secure system contain properties of both of them: the functional requirements of the insecure system and the security aspects of the authorisation.

Our second contribution is the idea of verifying system against safety temporal properties using refinement. In our approach, we propose for each safety property to construct an abstract model correspondingly, and prove that our system refines these models. The proof of refinement guarantees that our model satisfies these temporal properties. Safety properties which are expressed in LTL or by some regular expressions can be translated easily into Event-B models. The proofs of refinement depend on the glueing invariants between these "test" models and the system under verification.

As for tool support, we have used the "Parallel Composition Plug-in" [7] for supporting our combination of insecure model and the security authorisation. Our approach is in fact a special case of using shared event composition. Moreover, we also require a tool to support "multiple" refinements since we usually want to verify our secure model against several properties which are expressed as abstract models. This is not supported by standard Event-B and the Rodin Platform. However, somewhat surprisingly, the parallel composition plug-in provides a workaround for us. For each property, we create a composition machine which include only the final secure model, but we can specified the abstract model and the corresponding invariant for the secure model to refine this abstract model. This is a bonus for us since supporting multiple refinement is not the main purpose of the composition plug-in.

The structure of our report as follows. We give a brief overview of the Event-B modelling methods in Section 2. Section 3 discusses about the development of an insecure bank, without any security measure. Section 4 formalises separately a security authorisation system. Section 5 describes our approach of combining the two developed systems to create a secure bank. In Section 6, we verify the established secure system against some safety properties using refinement. Finally, Section 7 gives some conclusion and future work.

2 Event-B Modelling Methods

Event-B [3], unlike classical B [2], does not have a fixed syntax. Instead, it is a collection of modelling elements that are stored in a repository. Still, we present the basic notation for Event-B using some syntax. We proceed like this to improve legibility and help the reader remembering the different constructs of Event-B. The syntax should be understood as a convention for presenting Event-B models in textual form rather than defining a language.

Event-B models are described in terms of the two basic constructs: *contexts* and *machines*. Contexts contain the static part of a model whereas machines contain the dynamic part. Contexts may contain *carrier sets*, *constants*, *axioms*, where carrier sets are similar to types [4]. In this article, we simply assume that there is some context and do not mention it explicitly. Machines are presented in Section 2.1, and machine refinement in Section 2.2.

2.1 Machines

Machines provide behavioural properties of Event-B models. Machines may contain *variables*, *invariants*, and *events*. Variables v define the state of a machine. They are constrained by invariants $I(v)$. Possible state changes are described by means of events. Each event is composed of a *guard* $G(v)$ and an *action* $S(v)$[1] . The guard states the necessary condition under which an event may occur, and the action describes how the state variables evolve when the event occurs. An event can be represented by the following form

$$\mathsf{evt} \ \widehat{=}\ \textbf{when } G(v) \textbf{ then } S(v) \textbf{ end} \tag{1}$$

The short form

$$\mathsf{evt} \ \widehat{=}\ \textbf{begin } S(v) \textbf{ end} \tag{2}$$

[1] For simplicity, we do not treat events with *parameters*.

is used if the guard equals *true*. A dedicated event of the form (2) is used for *initialisation*.

The action of an event is composed of several *assignments* of the form

$$x \;:=\; E(v) \tag{3}$$
$$x \;:\in\; E(v) \tag{4}$$
$$x \;:\mid\; Q(v, x') \quad , \tag{5}$$

where x are some variables, $E(v)$ expressions, and $Q(v, x')$ a predicate. Assignment form (3) is *deterministic*, the other two forms are *non-deterministic*. Form (4) assigns x to an element of a set, and form (5) assigns to x a value satisfying a predicate. The effect of each assignments can also be described by a before-after predicate:

$$BAP\big(x \;:=\; E(v)\big) \quad \widehat{=} \quad x' \;=\; E(v) \tag{6}$$
$$BAP\big(x \;:\in\; E(v)\big) \quad \widehat{=} \quad x' \;\in\; E(v) \tag{7}$$
$$BAP\big(x \;:\mid\; Q(v, x')\big) \quad \widehat{=} \quad Q(v, x') \quad . \tag{8}$$

A before-after predicate describes the relationship between the state just before an assignment has occurred (represented by un-primed variable names x) and the state just after the assignment has occurred (represented by primed variable names x'). All assignments of an action $S(v)$ occur simultaneously which is expressed by conjoining their before-after predicates, yielding a predicate $A(v, x')$. Variables y that do not appear on the left-hand side of an assignment of an action are not changed by the action. Formally, this is achieved by conjoining $A(v, x')$ with $y' = y$, yielding the before-after predicate of the action:

$$BAP\big(S(v)\big) \quad \widehat{=} \quad A(v, x') \;\wedge\; y' = y \quad . \tag{9}$$

In proof obligations we represent the before-after predicate $BAP\big(S(v)\big)$ of an action $S(v)$ directly by the predicate

$$\boldsymbol{S}(v, v') \quad .$$

Proof obligations serve to verify certain properties of a machine. Here a proof obligation is presented in the form of a sequent: "hypotheses" $\vdash$ "goal". The intuitive meaning of this sequent is that under the assumption of the *hypotheses*, prove the *goal*.

For each event of a machine, the following proof obligation which guarantees *feasibility* must be proved.

$$\boxed{\begin{array}{ll|c} & I(v) & \\ & G(v) & \textbf{FIS} \\ \vdash & & \\ & \exists v' \cdot \boldsymbol{S}(v, v') & \end{array}}$$

By proving feasibility, we achieve that $\boldsymbol{S}(v, v')$ provides an after state whenever $G(v)$ holds. This means that the guard indeed represents the enabling condition of the event.

Invariants are supposed to hold whenever variable values change. Obviously, this does not hold a priori for any combination of events and invariants and, thus, needs to be proved. The corresponding proof obligation is called *invariant preservation*:

$$\boxed{\begin{array}{ll|c} & I(v) & \\ & G(v) & \\ & \boldsymbol{S}(v,v') & \textbf{INV} \\ \vdash & & \\ & I(v') & \end{array}}$$

Similar proof obligations are associated with the initialisation event of a machine. The only difference is that the invariant and guard do not appear in the antecedent of the proof obligations (**FIS**) and (**INV**).

2.2 Machine Refinement

Machine refinement provides a means to introduce more details about the dynamic properties of a model [4]. For more on the well-known theory of refinement, we refer to the Action System formalism that has inspired the development of Event-B [5]. We present some important proof obligations for machine refinement. As mentioned before, the user of Event-B is not presented with a behavioural model but only with proof obligations. The proof obligations describe the semantics of Event-B models.

A machine CM can refine at most one other machine AM. We call AM the *abstract* machine and CM a *concrete* machine. The state of the abstract machine is related to the state of the concrete machine by a *glueing invariant* $J(v, w)$, where v are the variables of the abstract machine and w the variables of the concrete machine.

Each event ea of the abstract machine is *refined* by one or more concrete events ec. Let abstract event ea and concrete event ec be:

$$\mathsf{ea} \;\widehat{=}\; \textbf{when } G(v) \textbf{ then } S(v) \textbf{ end}$$

$$\mathsf{ra} \;\widehat{=}\; \textbf{when } H(w) \textbf{ then } T(w) \textbf{ end}$$

Somewhat simplified, we can say that ec refines ea if the following conditions hold.

1. The concrete event is feasible. This is formalised by the following proof obligation.

$$\boxed{\begin{array}{ll|c} & I(v) & \\ & J(v,w) & \\ & H(w) & \textbf{FIS} \\ \vdash & & \\ & \exists w' \cdot \boldsymbol{T}(w,w') & \end{array}}$$

2. The guard of *ec* is stronger than the guard of *ea*. This is formalised by the following proof obligation.

$$\boxed{\begin{array}{ll|c} & I(v) & \\ & J(v,w) & \\ & H(w) & \textbf{GRD\_REF} \\ \vdash & & \\ & G(v) & \end{array}}$$

3. The abstract event can always "simulate" the concrete event and preserve the glueing (concrete) invariant. This is formalised by the following proof obligation.

$$\boxed{\begin{array}{ll|c} & I(v) & \\ & J(v,w) & \\ & H(w) & \textbf{SIM} \\ & \boldsymbol{T}(w,w') & \\ \vdash & & \\ & \exists v' \cdot \begin{pmatrix} \boldsymbol{S}(v,v') \wedge \\ J(v',w') \end{pmatrix} & \end{array}}$$

For the initialisation, the corresponding proof obligations are obvious. The proofs of these above obligations guarantee the correctness of the refinement model with respect to the abstract model and the glueing invariant between them.

In the course of refinement, often *new events ec* are introduced into a model. New events must be proved to refine the implicit abstract event *skip* that does nothing.

$$\mathsf{skip} \;\widehat{=}\; \textbf{begin}\ \textsc{skip}\ \textbf{end} \tag{10}$$

Moreover, it may be proved that new events do not collectively diverge, but this is not relevant here.

3 An Insecure Bank

In this section, we develop a simple bank system maintains the balances of some accounts without any knowledge about security measure. In our simple system, we even assume that set of customers and accounts are constants, but this should not effect the generality of our approach.

The following context information about customers and accounts are also shared with the development of security authorisation later.

carrier sets: $USER, ACCOUNT$

constants: $customers, accounts, initBal$

axioms:
$customers \subseteq USER$
$accounts \subseteq ACCOUNT$
$accounts \neq \varnothing$
$initBal \in accounts \rightarrow \mathbb{N}$

We have two carrier sets *USER* and *ACCOUNT* to denote the set of all possible users and accounts. Constant *customers* denotes the current set of registered customers of the bank, whereas constant *accounts* formalises the set of existing accounts within the bank. We also required that this there must be some existing accounts in the system. Since we are going only model only the query operation about the balance of an account and transferring operation between two account, other operations such as deposit, withdraw are left out. As a result, we will need to have some initial balance in our accounts to start with and this is model by constant *initBal*.

Moreover, to denote different states of the bank, we use the following enumerated set *BANK_STATE*, with the elements corresponding to the initial state, operation state (after login), transfer state (between transfer request and transfer execution).

carrier sets: $BANK\_STATE$

constants: $INIT, OP, TRANS$

axioms:
$BANK\_STATE = \{INIT, OP, TRANS\}$
$\neg INIT = OP$
$\neg INIT = TRANS$
$\neg OP = TRANS$

The model of our insecure bank is as follows. We have two variables to keep track of the current state of the bank, and the balance of the existing accounts within the bank. Every account must have some balance which is a natural number associated with. The initialisation for these two variables are trivial.

variables: $bankState, balance$

invariants:
$bankState \in BANK\_STATE$
$balance \in accounts \rightarrow \mathbb{N}$

```
init
  begin
    $bankState := INIT$
    $balance := initBal$
  end
```

We first model the login operation as two separate events to reflect the different outcome of this operation, either successful or not. Here we use a convention for modelling the output of an operation by parameters of the event which are written in capital letters. In the following two events, OK is the corresponding output to the user of the system. In the case of unsuccessful, the bank state is still at the initial state.

```
loginTrue
  any  $u, OK$  where
    $bankState = INIT$
    $u \in customers$
    $OK = \mathrm{TRUE}$
  then
    $bankState := OP$
  end
```

```
loginFalse
  any  $u, OK$  where
    $bankState = INIT$
    $u \notin customers$
    $OK = \mathrm{FALSE}$
  then
    SKIP
  end
```

The event logout is modelled as follows

```
logout
  when
    $bankState = OP$
  then
    $bankState := INIT$
  end
```

The user can logout when the bank is in "operation" state, but not between the execution of the transfer operation. Note that our bank here is single threaded, i.e. at a time at most one user can log in and use the system.

The query operation about balance of an account is straight forward with the use of event parameters as outputs for the operation. We have a parameter a to indicate the account under query.

```
balance
  any  $a, VAL$  where
    $bankState = OP$
    $a \in accounts$
    $VAL = balance(a)$
  then
    SKIP
  end
```

The transfer operation are carried out in two steps: The user first make a request and then transfer is then executed or aborted depending on the users' input. As a result, the bank need to keep the information that the user enters in the first step in order to use these information later in executing the transfer. We introduce the following additional variables for this purpose.

variables: $from, to, amount, transferOK$

invariants:
$from \in accounts$
$to \in accounts$
$amount \in \mathbb{N}1$
$transferOK = \mathrm{TRUE} \Rightarrow from \neq to \land amount \leq balance(from)$

The variable *transferOK* to keep the status of the transfer request if it is valid. We have an invariant to state that if this is the case then the source account must be different from the destination account and there must be sufficient fund in the source account for the transfer.

The transfer request operation is modelled as the following event. It is straight forward to see that the event maintains the invariants presented above.

```
transferRequest
  any  $a1, a2, s, OK$  where
    $bankState = OP$
    $a1 \in accounts$
    $a2 \in accounts$
    $s \in \mathbb{N}1$
    $OK = \mathrm{bool}(a1 \neq a2 \land s \leq balance(a1))$
  then
    $bankState := TRANS$
    $transferOK := OK$
    $from := a1$
    $to := a2$
    $amount := s$
  end
```

Depending on the validity of the request, the bank either carries out the transfer or aborts the transfer operation. This is modelled by the following pair of events.

```
transferExecTrue
  when
    $bankState = TRANS transferOK = \mathrm{TRUE}$
  then
    $bankState := OP$
    $transferOK = \mathrm{FALSE}$
    $balance := balance \Lleftarrow$
        $\{from \mapsto balance(from) - amount, to \mapsto balance(to) + amount\}$
  end
```

```
transferExecFalse
  when
    $bankState = TRANS$
    $transferOK = \mathrm{FALSE}$
  then
    $bankState := OP$
  end
```

4 A Security Authorisation

In this section, we look at the model of a security authorisation which is independent of the insecure bank that we have modelled in the last section. First, we define some context that will be used for the model of the authorisation.

We use a enumerated set SEC_STATE for denoting different states of the authorisation. We leave out the trivial axioms to specify that the elements of this set are different. They corresponding to the following state: initial, entering PIN, operation, entering TAN, transferring state (between transfer request and transfer execution).

carrier sets: SEC_STATE

constants: $SEC\_INIT, SEC\_PIN, SEC\_OP, SEC\_TAN, SEC\_TRANS$

The authorisation uses several security methods: PINs, TANs (Transaction Authorisation Numbers) and privileges. They are defined as some carrier sets and constants as follows.

carrier sets: $PIN, ACTION, TAN$

constants: $credentials, BALANCE, TRANSFER, privileges, tanlist$

```
axioms:
    $credentials \in customers \rightarrow PIN$
    $ACTION = \{BALANCE, TRANSFER\}$
    $\neg BALANCE = TRANSFER$
    $privileges \in \mathbb{P}(USER \times ACCOUNT \times ACTION)$
    $tanlist \in customers \rightarrow (\mathbb{N} \rightarrow TAN)$
```

- We model the set of PINs by a carrier set and each customer needs to have a pin.
- The authorisation monitors the customers' privileges. This is the relationship between a users, an account and the action that the user can perform on this account. For our simple bank, the action is either checking the balance or transferring money from this account.
- We model the set of possible TANs by a carrier set and for each customer, we assume that there is an infinite sequence of TANs associated with them. An infinite sequence of TANs is modelled by a total function from $\mathbb{N}$ to TAN.

The model of the authorisation contains the variables representing the current state, the current user who logged into the system and current index in TAN lists for each customer.

```
variables:   $secState, user, tid$
```

```
invariants:
    $secState \neq SEC\_INIT \Rightarrow user \in customers$
    $tid \in customers \rightarrow \mathbb{N}$
```

```
init
  begin
    $secState := SEC\_INIT$
    $user :\in USER$
    $tid := customers \times \{0\}$
  end
```

An important invariant is that if the authorisation is not in the initial state, i.e. some user has already attempted to log into the system, then that user must be an existing customer of the bank. Initially, the index in the tan list for each customer starts from the beginning of their corresponding sequence, i.e. with index 0.

The login operation is modelled similarly by two events as in the insecure bank. The differences are that the authorisation keeps track of the user logging into the system and there is no need for an output.

```
loginTrue
  any  $u$  where
    $secState = SEC\_INIT$
    $u \in customers$
  then
    $bankState := OP$
    $user := u$
  end
```

```
loginFalse
  any  $u$  where
    $secState = SEC\_INIT$
    $u \notin customers$
  then
    SKIP
  end
```

For the authorisation, the user can log out of the system when in either two state *SEC_OP* or *SEC_TAN*.

```
logout
  when
    $secState \in \{SEC\_OP, SEC\_TAN\}$
  then
    $secState := SEC\_INIT$
  end
```

After enter the user name, the user of the system will be ask to enter his PIN number. Again, we separate the operation into two events depend on the outcome. Both of them has output to indicate if the operation is successful or not.

```
pinTrue
  any  $p, OK$  where
    $secState = SEC\_IDENT$
    $user \mapsto p \in credentials$
    $OK = \mathrm{TRUE}$
  then
    $secState := SEC\_OP$
  end
```

```
pinTrue
  any  $p, OK$  where
    $secState = SEC\_IDENT$
    $user \mapsto p \notin credentials$
    $OK = \mathrm{FALSE}$
  then
    SKIP
  end
```

The authorisation for the balance operation does not concern with the actual functional requirement operation, but with the privilege of the user. The user need to have a privilege for checking the balance of the account in order to perform the actual operation.

```
balance
  any  $a$  where
    $secState = SEC\_OP$
    $user \mapsto a \mapsto BALANCE \in privileges$
  then
    SKIP
  end
```

Similarly for the transfer request operation, the authorisation does not relate to the amount that will be transferred or the destination account. Instead, the authorisation only relates the privilege of the user on the source account.

```
transferRequest
  any  $a1$  where
    $secState = SEC\_OP$
    $user \mapsto a1 \mapsto TRANSFER \in privileges$
  then
    $secState := SEC\_TAN$
  end
```

After the user makes a transfer request, the bank requires the user to enter the TAN number corresponding to transfer operation. This must be the TAN corresponding to the current number in the TAN sequence of this user. We separate this operation into two events for different outcome of this operation. In the case the user gives the correct tan number, the authorisation continues to the next state in order to execute the transfer, and at the same time, the index in the tan list corresponding to the user is advanced by 1. On the other hand, in the case of unsuccessful, the state of the authorisation comes back to operation state, i.e. the transfer process is cancelled.

```
tanTrue
  any  $t, OK$  where
    $secState = SEC\_TAN$
    $t = tanlist(user)(tid(user))$
    $OK = \mathrm{TRUE}$
  then
    $secState := SEC\_TRANS$
    $tid := tid \mathbin{\lhd\!\!-} \{user \mapsto tid(user) + 1\}$
  end
```

```
tanFalse
  any  $t, OK$  where
    $secState = SEC\_TAN$
    $t \neq tanlist(user)(tid(user))$
    $OK = \mathrm{FALSE}$
  then
    $secState := SEC\_OP$
  end
```

In contrast with the insecure system, the transfer execution for authorisation just changes its internal state from *SEC_TRANS* to *SEC_OP*.

```
transferExec
  when
    $secState = SEC\_TRANS$
  then
    $secState := SEC\_OP$
  end
```

5 A Secure Bank

In this section, we see how the two systems, i.e. insecure bank and security authorisation, combining together to create the secure bank. We merge the two systems as follows.

- The states of the two machines are merged.
- The following events are merged

Insecure Bank	Sec. Auth.	Secure Bank
loginTrue	loginTrue	loginTrue
loginFalse	loginFalse	loginFalse
logout	logout	logout
balance	balance	balance
transferRequest	transferRequest	transferRequest
transferExecTrue	transferExec	transferExecTrue
transferExecFalse	transferExec	transferExecFalse

- The following events are copied from Security Authorisation: pinTrue, pinFalse, tanTrue, tanFalse.

We use the parallel composition plug-in [7] to create a "composition machine" according to the above merging information. The resulting Event-B machine is created automatically by the plug-in. For example, the combining event balance is as follows.

```
balance
  any  $a, VAL$  where
    $bankState = OP$
    $a \in accounts$
    $VAL = balance(a)$
    $secState = SEC\_OP$
    $user \mapsto a \mapsto BALANCE \in privileges$
  then
    SKIP
  end
```

Another example is the event transferRequest, which is combined as follows.

```
transferRequest
  any  $a1, a2, s, OK$  where
    $bankState = OP$
    $a1 \in accounts$
    $a2 \in accounts$
    $s \in \mathbb{N}1$
    $OK = bool(a1 \neq a2 \land s \leq balance(a1))$
    $secState = SEC\_OP$
    $user \mapsto a1 \mapsto TRANSFER \in privileges$
  then
    $bankState := TRANS$
    $transferOK := OK$
    $from := a1$
    $to := a2$
    $amount := s$
    $secState := SEC\_TAN$
  end
```

As one can see, the combination of the two systems contains the properties of both, the functional requirements from the insecure systems is guarded by the security authorisation to prevent improper usages.

Here, we present the approach where the two systems are developed separately and combining later together. We could develop the secure system as a superposition refinement of the insecure bank by adding information about security authorisation. By specifying the two sub-systems separately, we have separation of concerns when developing these systems.

6 Analysis of Safety Properties

Since we have developed the secure system, we would like to verify our secure system against some safety properties. We consider here two properties as described in [6].

1. No *balance* check occurs before a sequence of successful *login* and *pin* entries.
2. No *transferExec* occurs before a successful *tan* entry.

These properties can be expressed by a CSP processes. For example the second property can be specified as follows. (From now on, we use the second property for our illustration).

$$\begin{aligned}
INIT &= tanTrue \rightarrow TRANS \\
&\quad \Box (\Box\, x : Other \cdot x \rightarrow INIT) \\
TRANS &= transferExec \rightarrow INIT \\
&\quad \Box (\Box\, x : Other \cdot x \rightarrow INIT)
\end{aligned}$$

where *Other* is the set of other events rather than tanTrue and transferExec. Notice that in our model, we have two different events corresponding to execution of transfer, namely transferExecTrue and transferExecFalse.

On the other hand, we this property can be represented by a simple automaton as in Figure 1.

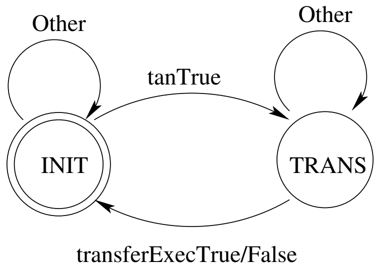

Fig. 1. Automaton representation of the second property

The corresponding Event-B model of this automaton is by having a variable to represent the current state. First, we defined a carrier set to represent the possible state of this automaton.

carrier sets: $TEST2\_STATE$

constants: $TEST2\_INIT, TEST2\_TRANS$

axioms:
  $\neg\ TEST2\_INIT = TEST2\_TRANS$

The variable *test2State* represents the state and is initialised to *TEST2_INIT* as follows.

variables: *test2State*

invariants:
  $test2State \in TEST2\_STATE$

```
init
  begin
    $test2State := TEST2\_INIT$
  end
```

The transition system of the automaton is modelled by three events corresponding to the different transitions. The event other is the same as the skip event.

```
tanTrue
  when
    $test2State = TEST2\_INIT$
  then
    $test2State := TEST2\_TRANS$
  end
```

```
transferExec
  when
    $test2State = TEST2\_TRANS$
  then
    $test2State := TEST2\_INIT$
  end
```

```
other
  begin
    SKIP
  end
```

We need to prove that our secure system refines the above test model. We need the glueing invariant as follows. The glueing invariant related the state of the test model, i.e., *test2State* and the state of the secure system, here, represented by variable *secState*.

invariants:

$test2State = TEST2\_TRANS \Leftrightarrow secState = SEC\_TRANS$

We also need to specify the corresponding refinement relationship between events: tanTrue event of the secure model refines tanTrue event of the test model, events transferExecTrue and transferExecFalse both refine transferExec event of the test model. All other events in the secure model correspond to other in the abstract test model.

We also used the parallel composition plug-in [7] in order to verify this refinement relationship. In Event-B, a model can be a refinement of at most one abstract model. The reason is that the invariant in a model can represent at most one glueing invariant with an abstract model. In our case, we would like to verify our model against different abstract models representing different properties. In other words, we would like to be able to state the following: our system refines the first test model with some glueing invariants and also refines the second test model with some other glueing invariants. The plug-in provides us with a workaround by constructing a special composition machine contains just the secure system, and provides information that this composition machine refines the test model with the specify glueing invariants. This can be illustrate by the diagram in Figure 2. Since the parallel composition plug-in is not designed for this purpose, the fact we can use the plug-in for multiple refinement is an extra bonus for us.

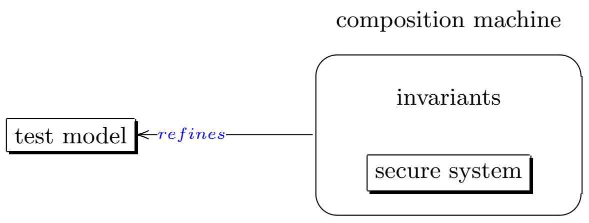

Fig. 2. Using composition plug-in for testing refinement

Using this technique, we verify that our secure bank system satisfies both properties as expected.

7 Conclusion

We presented here our approach for developing access control system in Event-B. The method that we use is to specify separately the insecure system and the security authorisation and later to combine them to construct secure system. Moreover, in order to verify our final system against some safety temporal properties, we propose an approach of constructing abstract models corresponding to these properties and proving the refinement relationship between the final model and these abstract model. We have used the parallel composition plug-in as a tool support.

As for future work, we are going to investigate the example from SAP as to see the applicability of our approach to this particular domain. We have confident that our method is well suited to the particular mini-pilot for access control. This will be our basis for further investigation of security aspect applying to the context of business communication.

As for our work related to safety temporal properties, our approach is promising. Proving refinement is more general than proving trace refinement using some model checker. The construction of the test abstract model is straight forward and could be partially automatised. More interesting question which need to be investigate further is about the preservation of these temporal properties with refinement. There has been some work along line, notably by Abadi and Lamport [1]. We would like to investigate to know which kind of temporal properties is preserved with Event-B refinement. Otherwise, what are the extra proof obligations necessarily for preserving other kind of temporal properties.

References

1. Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.

2. Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.
3. Jean-Raymond Abrial. *Modeling in Event-B: System and Software Design.* Cambridge University Press, 2009. To appear.
4. Jean-Raymond Abrial and Stefan Hallerstede. Refinement, Decomposition and Instantiation of Discrete Models: Application to Event-B. *Fundamentae Informatica*, 2006.
5. Ralph-Johan Back. Refinement Calculus II: Parallel and Reactive Programs. In J. W. deBakker, W. P. deRoever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 67–93, Mook, The Netherlands, May 1989. Springer-Verlag.
6. David A. Basin, Ernst-Rüdiger Olderog, and Paul E. Sevinç. Specifying and analyzing security automata using csp-oz. In Feng Bao and Steven Miller, editors, *ASIACCS*, pages 70–81. ACM, 2007.
7. Renato Silva. Parallel composition using event-b. `http://wiki.event-b.org/index.php/Parallel_Composition_using_Event-B`.