```
AR IN ARObj
SUCH THAT
AR.Status<>None;
ACTION
I:=AR;

% The RIMO is sent for the first time: it takes
% the 'Issued' state.
EVENT RIMO1st(RIMO; I)
GUARD
RIMO IN RIMOObj
SUCH THAT
RIMO.Status=None;
ACTION
I:=RIMO;
I.Status:=Issued;

% The RIMO is an old RIMO with the state changed.
% The new state becomes effective.
EVENT NewRIMO(RIMO; I,RIMOOut)
GUARD
RIMO IN RIMOObj
SUCH THAT
RIMO.Status<>None;
ACTION
I:=RIMO;

% If the item sent is a Verification & Validation Report
% for a configuration item, the VVR must be evaluated
% to determine the next state of the reviewed item.
EVENT IsVVR(IFVT; VVR)
GUARD
IFVT IN ItemForVVRTest
SUCH THAT
IFVT.Kind=VVR;
ACTION
VVR:=IFVT;
```

```
% Registers the event of state transition
% into the Event Journal.
EVENT RegisterStateTransition(ITS; D2,IR,IFVT)
GUARD
ITS IN ItemToStore;
ACTION
% Tool invocation.
EJRegisterTool.Invoke(ITS,ITS.Status);
D2:=EJRegisterTool.ToolOK;
IR:=ITS;
IFVT:=ITS;

% Stores the item.
EVENT StoreItem(IR,D1; IS)
GUARD
IR IN ItemRegistered,
D2 IN Done2;
ACTION
% Tool invocation.
FilingTool.Invoke(I);
IS:=FilingTool.ToolResult;

% The item was not formally correct.
EVENT Refused(D1,IR; RC)
GUARD
IR IN ItemRefused,
D1 IN Done1;
ACTION
% Builds the output token.
RC.ItemObj:=IR;
RC.ItemRef:=NIL;
RC.OpOK:=TRUE;
RC.Accepted:=FALSE;

% Transfers a item which is not a VVR out of
% activity body. The only operations that must
```

```
% be performed before event's firing are item
% storing and transition registering.
EVENT TransferSimpleItem1(IS; RC)
GUARD
IS IN ItemStored
ACTION
% Builds the output token.
RC.ItemObj:=IS;
RC.ItemRef:=IS.ID;
RC.OpOK:=TRUE;
RC.Accepted:=TRUE;

% Transfers a VVR out of activity body. In addition
% to operations described before, the event can fire
% only after VVR has been evaluated to determine
% state transitions associated.
EVENT TransferVVR(IS,E; RC)
GUARD
IS IN ItemStored
ACTION
% Builds the output token.
RC.ItemObj:=IS;
RC.ItemRef:=IS.ID;
RC.OpOK:=TRUE;
RC.Accepted:=TRUE;
```

The previous activity contains an invocation to **CheckItem Activity**, which is shown in Fig. A.14. This relatively simple activity is quite interesting since it uses the object-oriented features of polymorphism and late binding. In fact, event PerformCheck contains the invocation of the method CheckFormalCorrectness binded with the token object contained in RequestsInfo. Every kind of item has its own implementation of this method, as described in Section A.2: consequently, the correct check is selected on the basis of polymorphism rules. The formal description is the following:

```
% Performs a formal correctness check
% on the received item.
```

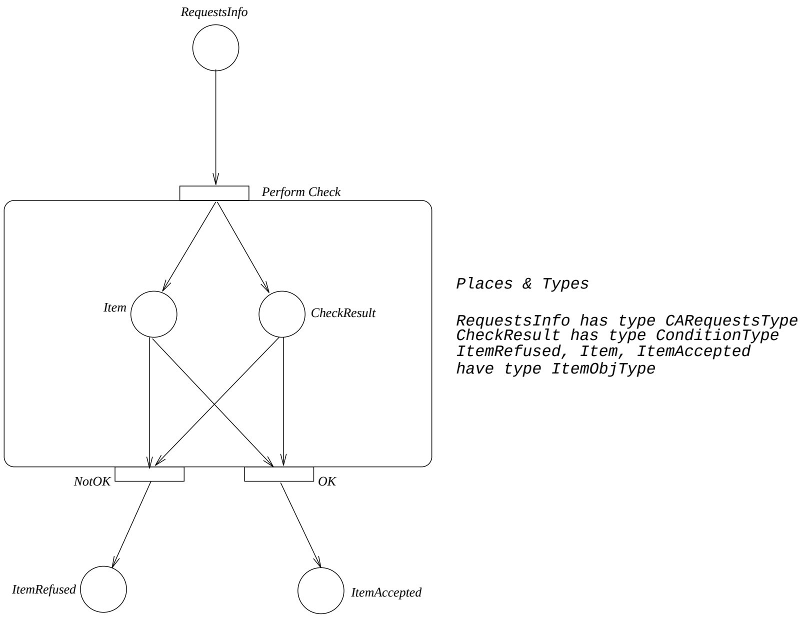


Figura A.14: **CheckItemOp Activity.** The implementation.

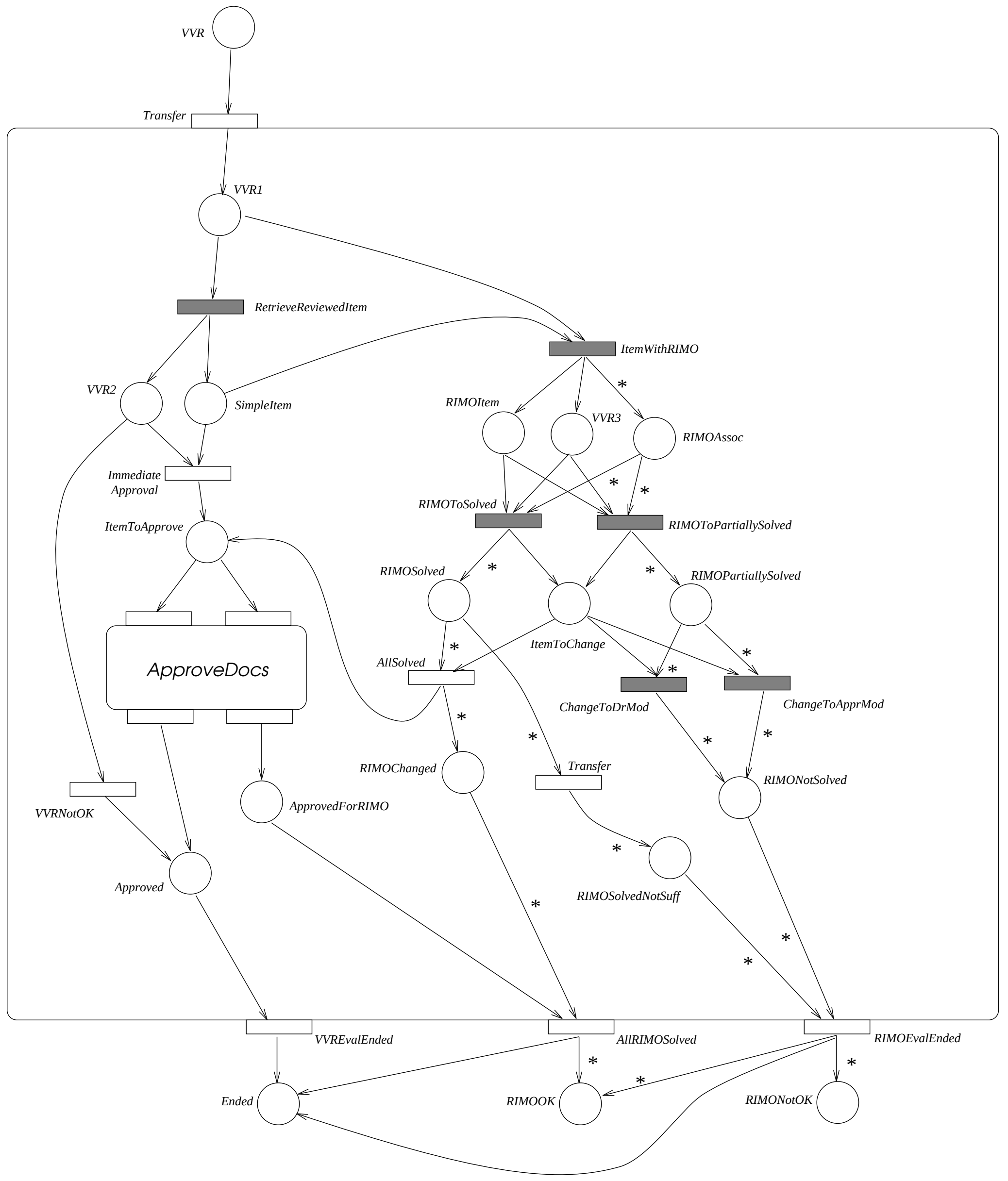
```
EVENT PerformCheck(RI; I,CR)
GUARD
RI IN RequestsInfo
SUCH THAT
RI.OpReq=CheckItem;
ACTION
% Method invocation. The check is performed only through
% formal checking of the module contents. The correct method to
% be employed is selected through polymorphism and late binding
% according to the input token's type.
CR:=RI.ItemObj.CheckFormalCorrectness;
I:=RI.ItemObj;

% The item is correct.
EVENT OK(CR,I; IA)
GUARD
CR IN CheckResult
SUCH THAT
```

```
CR=OK;
ACTION
IA:=I;

% The item is not correct.
EVENT NotOK(CR; IR)
GUARD
CR IN CheckResult
SUCH THAT
CR=NotOK;
ACTION
IR:=I;
```

SendItemOp invokes an activity deeply linked with software anomaly management, namely **EvaluateVVR**. This activity takes as input a Verification & Validation Report, analyzes the outcome of review activity and determines state transitions for reviewed item, related objects, and RIMOs. Different operations are performed according to context that originated the review, as review on an item sent for the first time to Configuration Management or review on an item with RIMO. To determine which case has to be followed, the item reviewed is retrieved from Configuration Archive and the associated set of RIMO is examined. If the first case is satisfied, there are two possible subcases:

- *Review was negative.* No further action is required and activity ends. An Anomaly Report is issued by review team, and this will start the anomaly solving when the AR will be received by Configuration Management.
- *Review was positive.* Activity ApproveDocs is invoked to determine and perform state transitions over related items.

If the second case is satisfied, RIMOs associated with the reviewed item are retrieved and separated accordingly to the what indicated in VVR: as described in Section A.2, `VVRFormType` has two fields that are filled, during review, with the sets of RIMOs to be considered solved and the ones that lack some feature. Accordingly to these indications, the appropriate state transitions are performed.




```
Places & Types

VVR has type VVRFormType
SimpleItem, ItemToApprove, RIMOItem, ItemToChange have type ItemObjType
Approved, ApprovedForRIMO, Ended have type ConditionType
RIMOChanged, RIMOAssoc, RIMOSolved, RIMOPartiallySolved, RIMONotSolved, RIMOOK, RIMONotOK have type RIMOFormType
```

Figura A.15: **EvalVVR Activity.** The implementation.

If all RIMOs are solved, the item must be approved, consequently, the same approval iter described before must be followed. If some RIMO is in PartiallySolved state, the associated item's state is updated accordingly to its previous state, in either 'DraftModified' or 'ApprovedModified'. Then, the remaining transitions take care of correctly transfer RIMOs objects in intermediate places; note that these results are used in anomaly solving to determine if further updates are needed on items or not.

It must be pointed out that if the review for an item that owns some RIMOs was not positive, no AR is issued, but RIMO are re-sent (through RIMONotOK place) to implementor, for update completion. Here follows formal definition:

```
% Transfers the VVR token into activity body.
EVENT Transfer(VVR; VVR1)
GUARD
VVR IN VVR;
ACTION
VVR1:=VVR;

% Retrieves the item reviewed.
EVENT RetrieveReviewedItem(VVR1; SI,VVR2)
GUARD
VVR1 IN VVR1;
ACTION
RetrieveTool.Invoke(VVR1.RevItem);
SI:=RetrieveTool.ToolResult;
VVR2:=VV1;

% If the item does not have any RIMO associated
% it takes suddenly the 'Approved' state.
EVENT ImmediateApproval(SI,VVR2; ITA)
GUARD
SI IN SimpleItem
SUCH THAT
CARD(SI.RIMO)=0 AND VVR.RevOutcome=OK;
ACTION
ITA:=SI;
```

```
% The item has some RIMO, so further work must
% be performed before approval.
EVENT ItemWithRIMO(VVR1,SI; RI,VVR3,RIMOA)
LOCAL t: RIMOFormType;
GUARD
VVR1 IN VVR1,
SI IN SimpleItem
SUCH THAT
CARD(SI.RIMO)<>0 AND
(CARD(VVR1.RIMOOK)<>0 OR CARD(VVR1.RIMONotOK)<>0);
ACTION
FOR t IN VVR.RIMOOK
RetrieveTool.Invoke(t);
RIMOA:=RetrieveTool.ToolResult;
END
FOR t IN VVR1.RIMONotOK
RetrieveTool.Invoke(t);
RIMOA:=RetrieveTool.ToolResult;
END
RI:=SI;
VVR3:=VVR1;

% Extracts RIMO that have been implemented
% effectively and makes state transition
% to the 'Solved' state.
EVENT RIMOToSolved(VVR3,RI,RIMOA; ITC,RIMOS)
LOCAL t: RIMOFormType;
GUARD
VVR3 IN VVR3,
RI IN RIMOItem,
RIMOA IN RIMOAssoc
SUCH THAT
RIMOA.ID IN VVR3.RIMOOK;
ACTION
% Tool invocation.
FOR t IN RIMOA
```

```
ChangeStateTool.Invoke(RIMOA.ID,Solved);
EJRegisterTool.Invoke(RIMOA.ID,Solved);
RIMOS:=ChangeStateTool.ToolResult;
END
ITC:=RI;

% Extracts RIMO that have not been implemented
% effectively and makes state transition
% to the 'PartiallySolved' state.
EVENT RIMOToPartiallySolved(VVR3,RI,RIMOA; ITC,RIMOPS)
LOCAL t: RIMOFormType;
GUARD
VVR3 IN VVR3,
RI IN RIMOItem,
RIMOA IN RIMOAssoc
SUCH THAT
RIMOA.ID IN VVR3.RIMONotOK;
ACTION
% Tool invocation.
FOR t IN RIMOA
ChangeStateTool.Invoke(RIMOA.ID,PartiallySolved);
EJRegisterTool.Invoke(RIMOA.ID,PartiallySolved);
RIMOPS:=ChangeStateTool.ToolResult;
END
ITC:=RI;

% If there are RIMO in PartiallySolved state,
% makes the appropriate state transition for the
% related item.
EVENT ChangeToDrMod(ITC,RIMOPS; RIMONS)
GUARD
RIMOPS IN RIMOPartiallySolved,
ITC IN ItemToChange
SUCH THAT
ITC.Status=Draft;
ACTION
ChangeStateTool.Invoke(ITC.ID,DraftModified);
```

```
EJRegisterTool.Invoke(ITC.ID,DraftModified);
RIMONS:=RIMONS UNION RIMOPS;

% If there are RIMO in PartiallySolved state,
% makes the appropriate state transition for the
% related item.
EVENT ChangeToApprMod(ITC,RIMOPS; RIMONS)
GUARD
RIMOPS IN RIMOPartiallySolved,
ITC IN ItemToChange
SUCH THAT
ITC.Status=ApprovedReview;
ACTION
ChangeStateTool.Invoke(ITC.ID,ApprovedModified);
EJRegisterTool.Invoke(ITC.ID,ApprovedModified);
RIMONS:=RIMONS UNION RIMOPS;

% If all RIMO has been successfully solved,
% the associated item must be approved.
EVENT AllSolved(RIMOS,ITC; ITA,RIMOC)
GUARD
ITC IN ItemToChange,
ANY RIMOS IN RIMOSolved
SUCH THAT
CARD(RIMOS)=CARD(ITC.RIMO);
ACTION
RIMOC:=RIMOS;
ITA:=ITC;

% Transfers RIMOs solved when there are also
% other RIMOs in PartiallySolved state.
EVENT Transfer(RIMOS; RIMOSNS)
GUARD
ANY RIMOS IN RIMOSolved
SUCH THAT
CARD(RIMOS)<>CARD(I.RIMO);
ACTION
```

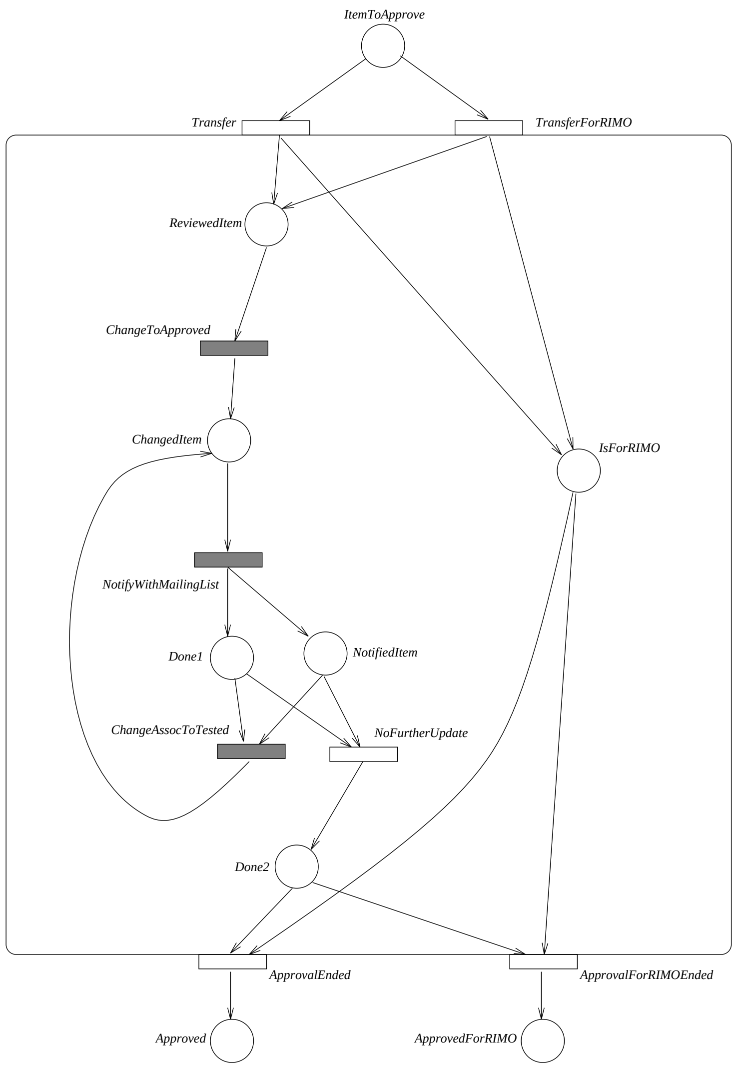
```
RIMOSNS:=RIMOS;

% The review gave a negative outcome.
% Further state transition are managed into
% callee activity.
EVENT VVRNotOK(VVR2; A)
GUARD
VVR2 IN VVR2
SUCH THAT
VVR2.RevOutcome=NotOK;
ACTION
A:=NotOK;

% The evaluation of a VVR issued after a 'normal'
% review is now ended.
EVENT VVREvalEnded(A; E)
GUARD
A IN Approved;
ACTION
E:=A;

% The evaluation ended with a the approval of an
% item that had RIMOs associated.
EVENT AllRIMOSolved(AFR,RIMOC; E,RIMOOK)
GUARD
AFR IN ApprovedForRIMO,
RIMOC IN RIMOChanged;
ACTION
RIMOOK:=RIMOOK UNION RIMOC;
E:=AFR;

% The evaluation of a VVR for an item that has
% associated RIMOs shows that some RIMO are not
% properly solved.
EVENT RIMOEvalEnded(RIMOSNS,RIMOPS; E,RIMOOK,RIMONOK)
GUARD
RIMOSNS IN RIMOSolvedNotSuff,
```

```
RIMOPS IN RIMOPartiallySolved;
ACTION
RIMOOK:=RIMOOK UNION RIMOSNS;
RIMONOK:=RIMONOK UNION RIMOPS;
E:=NotOK;
```

The activity **ApproveDocs** is invoked by EvalVVR and performs the necessary state transition on the item associated with the VVR. People interested in state transitions of the item are notified through the mailing list mechanism described in Section A.2 and eventually, a different output is provided when the state transition is performed for an item that owns RIMOs. Note that if the item reviewed is a Test Launch Report, the state of tested item must be updated too: this is obtained activating a second thread through the previous chunk of net.

Formal specification of activity's transitions is the following:

```
% Transfers the item into activity body,
% keeping track that the item must be
% approved after a simple review, with no
% RIMO involved.
EVENT Transfer(ITA; RI,IFR)
GUARD
ITA IN ItemToApprove
SUCH THAT
CARD(ITA.RIMO)=0;
ACTION
RI:=ITA;
IFR:=FALSE;

% Transfers the item into activity body,
% keeping track that the item has some
% RIMOs attached.
EVENT TransferForRIMO(ITA; RI,IFR)
GUARD
ITA IN ItemToApprove
SUCH THAT
CARD(ITA.RIMO)<>0;
```



Places & Types

ItemToApprove, ReviewedItem have type ItemObjType
IsForRIMO has type BOOLEAN
Done1, Done2, Approved, ApprovedForRIMO
have type ConditionType

Figura A.16: **ApproveDocs Activity.** The implementation.

```
ACTION
RI:=ITA;
IFR:=TRUE;

% Performs a state transition for the
% item specified, into 'Approved' state.
EVENT ChangeToApproved(RI; CI)
GUARD
RI IN ReviewedItem;
ACTION
% Tool invocation.
ChangeStateTool.Invoke(RI.ID,Approved);
EJRegisterTool.Invoke(RI.ID,Approved);
CI:=ChangeStateTool.ToolResult;

% Sends notifications to people interested
% in item's modifications using the associated
% mailing list.
EVENT NotifyWithMailingList(CI; D1,NI)
GUARD
CI IN ChangedItem;
ACTION
CI.Distrib:=CI.FillDistribution;
% Tool&method invocation.
NotifyTool.Invoke(CI.Distrib,'Item approved');
D1:=NotifyTool.ToolOK;
NI:=CI;

% If the item approved is a TLR, the item
% associated becomes 'Tested' and a further
% notification is needed.
EVENT ChangeAssocToTested(NI,D1; CI)
GUARD
D1 IN Done1,
NI IN NotifiedItem
SUCH THAT
NI.Kind=TLR;
```

```
ACTION
% Tool invocation.
ChangeStateTool.Invoke(NI.ItemTested,Tested);
EJRegisterTool.Invoke(NI.ItemTested,Tested);
NI:=ChangeStateTool.ToolResult;

% The item is not a TLR, so the evaluation
% is now ended.
EVENT NoFurtherUpdate(NI,D1; D2)
GUARD
D1 IN Done1,
NI IN NotifiedItem
SUCH THAT
NI.Kind<>TLR;
ACTION
D2:=AToken;

% Approval for a simple review is now ended.
EVENT ApprovalEnded(D2,IFR; A)
GUARD
D2 IN Done2,
IFR IN IsForRIMO
SUCH THAT
IFR=FALSE;
ACTION
A:=D2;

% Approval for an item carrying RIMOs
% is now ended.
EVENT ApprovalForRIMOEnded(D2,IFR; AFR)
GUARD
D2 IN Done2,
IFR IN IsForRIMO
SUCH THAT
IFR=TRUE;
ACTION
AFR:=D2;
```

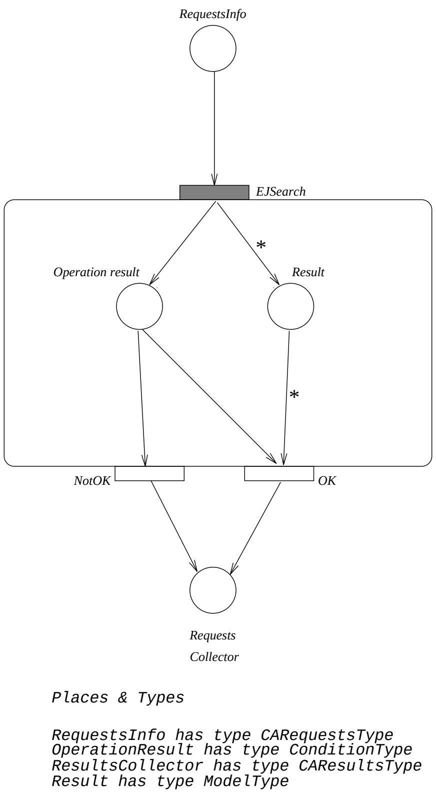


Figura A.17: **EventJournalSearchOp Activity.** The implementation.

EventJournalSearchOp Activity takes as input a predicate specified in some kind of language (dependent on the tool used to implement the Event Journal, and not relevant here) that allows extraction of relevant information from the Event Journal, which organized on a strictly temporal basis. For example, it is used in our specification to retrieve the number of solved RIMOs since the AR related to those RIMOs has been issued. The activity implementation is similar to Duplicate and Retrieve: a major difference is that the result of tool invocation maybe a set of objects, whose type is statically undefined, and consequently set to `ModelType`. This is due to the unawareness about how Event Journal is actually implemented. Transitions are defined as follows:

```
% Invokes the search on Event Journal.
EVENT EJSearch(RI; R,OR)
```

```
GUARD
RI IN RequestsInfo
SUCH THAT
RI.Op=EJSearch;
ACTION
% Tool invocation.
EJSearchTool.Invoke(RI.Pred);
R:=EJSearchTool.ToolResult;
OR:=EJSearchTool.ToolOK;

EVENT OK(OR,R; RC)
GUARD
R IN Result,
OR IN OperationResult
SUCH THAT
OR2.Result=OK;
ACTION
% Builds the output token.
RC.ItemObj:=R;
RC.ItemRef:=NIL;
RC.OpOK:=TRUE;
RC.Accepted:=NIL;

EVENT NotOK(OR; RC)
GUARD
OR IN OperationResult
SUCH THAT
OR=NotOK;
ACTION
% Builds the output token.
RC.ItemObj:=NIL;
RC.ItemRef:=NIL;
RC.OpOK:=FALSE;
RC.Accepted:=NIL;
```

Summing up, the most important result achieved by this modelization effort is that no SGMR group is formally needed. If we make the assumption that all the tools used within Configuration Management specification are available or feasible (and this hypothesis is actually verified), the whole organizational activity is fully automatized and codified in a formal process description.

A.5 Software Anomaly Management

In this section we explain the specification of the Software Anomaly Management, that constitutes our initial and primary specification goal. The specification is based both on original Quality Manual description of actual process and on the updates performed on this description on the basis of our suggestions and collaboration with Italtel. For example, the finite state machines integrated into our specification are not the ones described in the manual, but they are new automata defined in conjunction with the Italtel BUTD process expert.

Description of SLANG activities

Figure A.18 shows the static tree of **ManageSoftwareAnomalies**. By converse, the activity's high-level view is illustrated by Fig. A.19; the activity can be started by a unique event corresponding to the presence in Configuration Archive of an Anomaly Report.

Note that the activity box is not made of a solid line, but it is drawn using a dotted line: this can be viewed as a user-defined macro. In our intention, this notation distinguish a *module* from an *activity*. The module concept is not defined in the original SLANG definition, but it is our opinion that it is an important one when developing specifications of real software processes. A module is composed of an interface part and an implementation part as an activity is, and moreover it owns an activity static tree. The unique difference with respect to activity is in the management of input tokens:

- When an activity is started by the firing of an input tuple, this tuple of tokens will be managed by a dedicated instance of the activity. In other words, if the input event fires n times, there will be n activity instances managing corresponding tuples.

- When a module is started by the firing of an input tuple no instance is created but every input tuple fired is managed directly by the module

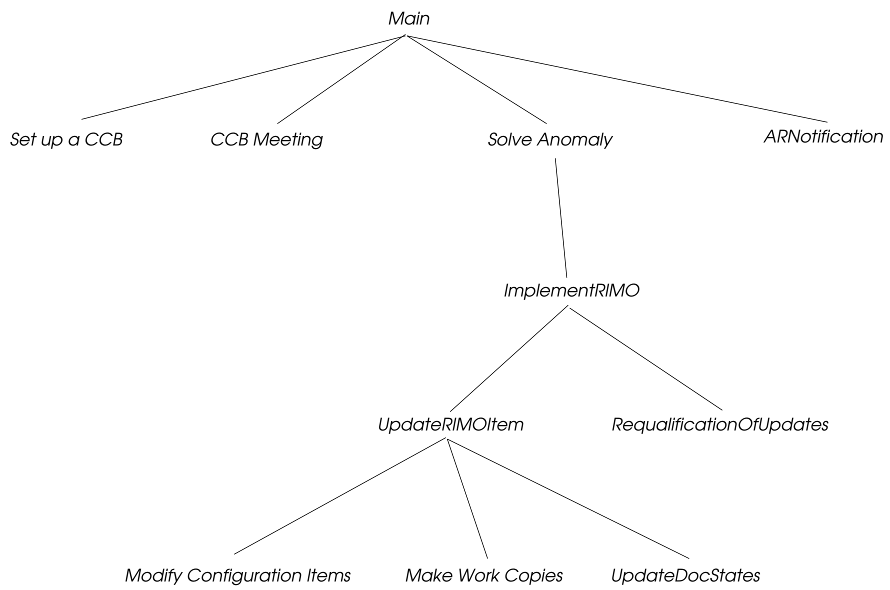


Figura A.18: The activity static tree relative to software anomalies management.

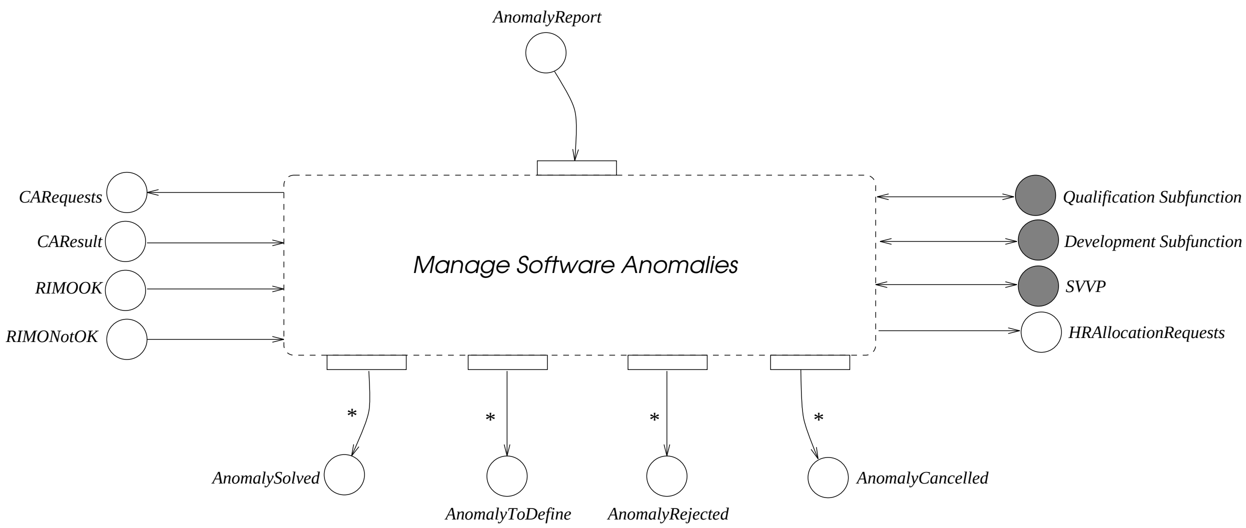
implementation. This means that if the input event fires n times, there will be only one instance of the module managing the tuples, possibly with n threads.

This allows a unique piece of net to manage all the events that compete to it, without distributing them on a set of activities' instances, without losing important features like modularization and information hiding. The need for this addition raised from our trial to remove a major assumption made in our previous version of specification. In that document, we assumed the following steps to be performed after Anomaly Report is notified:

- SVVP analyzes the AR notified and issues convocation letters for the CCB to be summoned.
- CCB members examine the single AR notified and determine the new AR state and RIMO to be executed.

Instead, real events to be executed are the following:

- SVVP has a list of all ARs that are waiting for a CCB examination: this includes ARs in 'Notified', ARs in 'Suspended' state, and ARs




```
Places & Types
AnomalyReport, AnomalySolved, AnomalyToDefine,
AnomalyCancelled have type ARModType
QualificationSubgroup, DevelopmentSubgroup, SVVP have type PersonType
CARequests has type CARequestsType
CAResults has type CAResultsType
RIMOOK, RIMONotOK have type RIMOFormType
HRAllocationRequests has type TaskAssignmentType
```

Figura A.19: High-level view of the main activity for Software Anomaly Management.

that are requested to be cancelled or suspended. When SVVP receives a new AR, it examine it together with other ARs waiting for a CCB summoning. When the number, or the importance, of ARs present is sufficient CCB must be summoned and convocation letters are sent.

- CCB members answer to convocation letters confirming their availability: only if all CCB members are available CCB can be actually started. If not, SVVP must reschedule the CCB.

- CCB starts to examine *all* ARs waiting.

In our first draft ManageSoftwareAnomalies was an activity. It is easily understood that removing the approximation of considering an AR at the time, we have no mean (unless to make complex "support" nets) to centralize the AR management by SVVP and CCB, since the activities corresponding to SVVP task and CCB meeting would be local to this main activity and so they would work only on the AR that enabled the starting event. Another solution would be to cut SVVP tasks and CCB meeting from the main anomaly activity and access it. This a suitable solution from a "technical" point of view, but it does not map well the real process: these tasks, in fact, are specific and intimately related only with software anomaly management, and there is no reason for defining them as independent activities providing services to other activities.

Nothing is added to SLANG: what we call a module it is only a syntactic notation to delimit a piece of SLANG net evidencing interaction with the rest of specification.

ManageSoftwareAnomalies can end with four mutually exclusive outcomes:

- the anomaly is cancelled, or

- the anomaly is rejected, or

- an anomaly is recognized but further information is needed, or

- the anomaly has been solved.

All these situations are modeled by the activity ending events.

ManageSoftwareAnomalies must access Configuration Management in order to allow updating and retrieving of software documentation: this is modeled by the interface places on the left of the activity box, which are the input and output places for the activity ManageConfigurationArchive, already described in Section A.4 and the places containing the sets of RIMOs to be considered respectively solved and not solved yet. Moreover, on the right side of activity box, are shown some places brown-colored, whose type is `PersonType`: we called them **Human places** and their color is brown to distinguish them from "traditional" places. The reason for the brown color instead of normal white color is to evidence that the place contains a human resource, that as it is known, plays a decisive role into software process enaction. Note that they are not a SLANG construct: they can be regarded as another macro, built over the SLANG kernel, in order to better suit modeler needs and taste. It must be pointed out that it is part of SLANG underlying philosophy to offer a well-defined kernel of basic constructs to a user that can build its own macro and constructs over these.

Figure A.20 shows the net underlying the module interface described before. The first action performed within the net is the invocation of ARNotification activity. This will perform necessary notifications and make the anomaly to get the new state of 'Notified'. The Anomaly Report with new state is transferred into the ARToBeExamined, that corresponds to the list of ARs described before. SVVP tasks can be activated by these set of ARs, or by the presence of cancel/suspend requests from process actors or by the combination of these two conditions.

SVVP analyzes the ARs in activity SetUpACCB, determining if the CCB must be summoned or not. If not, ARs are re-filled in ARToBeExamined for a later analysis. By converse, CCB summoning is described by CCBMeeting activation; CCB ends by filling the place UpdatedARs with the new copy of ARs, modified by CCB by changing the anomaly state and/or cancelling some RIMOs, and possibly by filling also the place CancelRIMO. This one is an intermediate place for the SolveAnomaly activity, which performs the actual solution of the anomaly, and it is used to selectively erase from the process the implementation of the specified RIMO. The mechanism is very simple: the only operation requested to cancel a RIMO is to fill the place CancelRIMO with the RIMO to be cancelled.

In any case, the tokens in UpdatedARs may enable one of these transitions, in mutual exclusion:

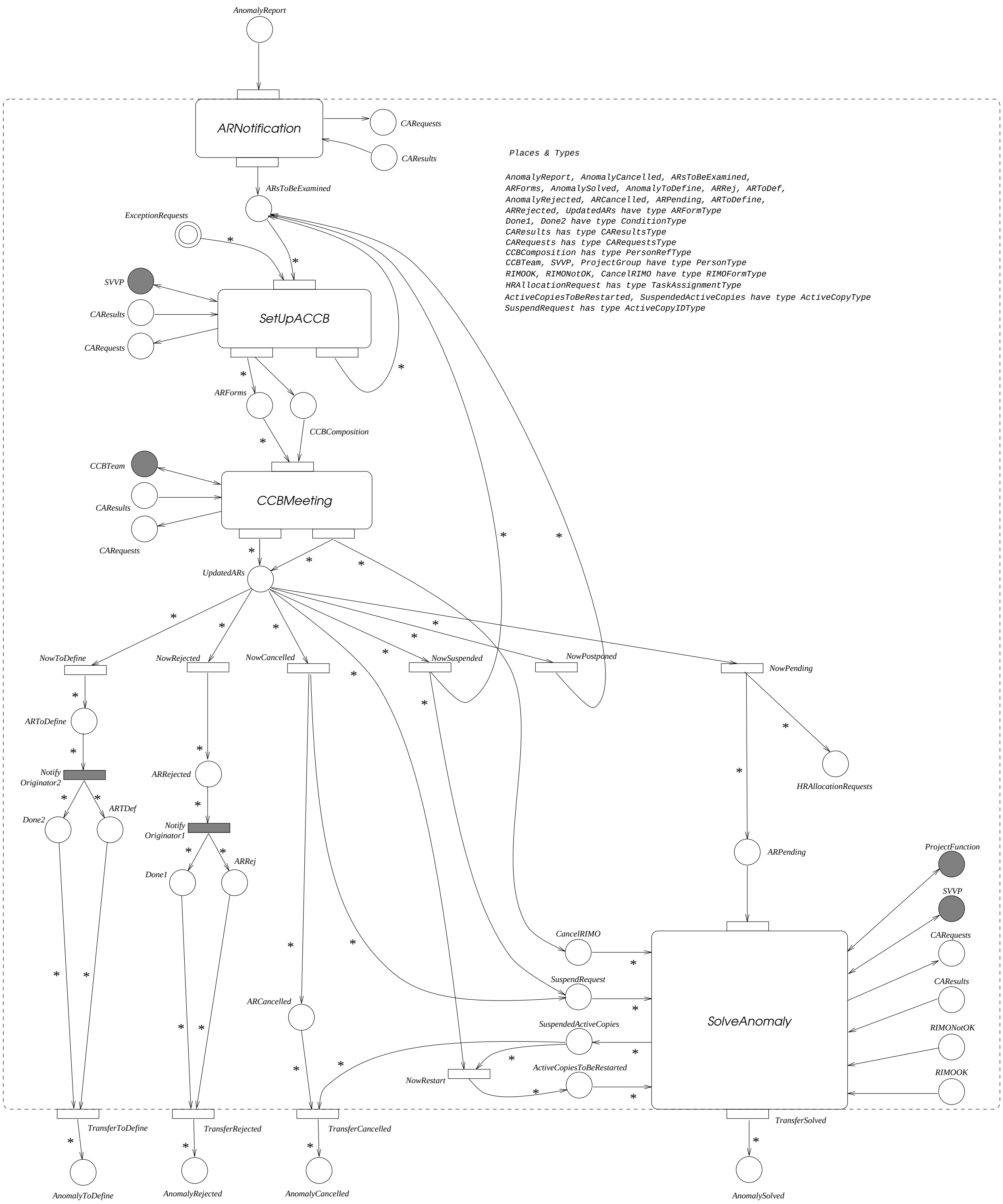


Figura A.20: **ManageSoftwareAnomalies.** The implementation.

- NowToDefine. The anomaly needs further information to be evaluated by the CCB. ARs with these state are used to notify their originator of CCB decision and later to produce the activity output.

- NowRejected. The anomaly is not recognized as a proper anomaly by the CCB. ARs with these state are used to notify their originator of CCB decision and later to produce the activity output.

- NowCancelled. The ARs with this state must be cancelled, i.e. the solution of the anomaly must be aborted. This is obtained by issuing a suspension request to the system for the instance associated to that anomaly, and then erasing the active copy from the system. The first step is obtained by filling the SolveAnomaly's built-in place SuspendRequest with the active copy identifier contained in the field Instance.ID of the AR; the second step is performed by taking the suspended active copy by the SolveAnomaly's built-in place SuspendedActiveCopies and consuming it in the output event TransferCancelled.

- NowRestart. This event occurs when some anomaly, previously suspended, needs to be restarted. This means that the anomaly will have either the 'PartiallySolved' or the 'Pending' state (as shown by the anomaly finite state machine), and that its active copy, contained in the Instance place of the AR, must be contained in SolveAnomaly's SuspendedActiveCopies. If all this is verified, the active copy is put into the built-in place ActiveCopiesToBeRestarted, provoking the restart of the activity.

- NowSuspended. In this case CCB has determined that the anomaly solution must be suspended. A suspension request is issued in SuspendRequest and the AR is put into ARToBeExamined, waiting for a new CCB that will possibly restart it.

- NowPending. The CCB decides to start anomaly's solution, *for the first time.* This means that the anomaly was not previously started and then suspended. This is checked with the SuspendedActiveCopies place.

Here follows the transitions' formal description:

```
% Spawns rejected anomalies.
EVENT NowRejected(UAR; ARR)
LOCAL t: ARFormType;
GUARD
UAR IN UpdatedAR
SUCH THAT
FORALL t IN UAR t.State=Rejected;
ACTION
ARR:=ARR UNION UAR;

% Spawns suspended anomalies that are
% already in process.
EVENT NowSuspended(UAR; ARTBE,SR)
LOCAL t: ARFormType;
GUARD
UAR IN UpdatedAR
SUCH THAT
FORALL t IN UAR
(t.State=Suspended AND t.Instance<>NIL);
ACTION
ARTBE:=ARTBE UNION UAR;
% Issues suspend request for the active
% copies actually managing the anomaly
% solution.
FOR t IN UAR
SR:=SR UNION t.Instance.ID;
END

% Spawns suspended anomalies whose solution
% has not been started yet.
EVENT NowPostponed(UAR; ARTBE)
LOCAL t: ARFormType;
GUARD
UAR IN UpdatedAR
SUCH THAT
FORALL t IN UAR
(t.State=Suspended AND t.Instance=NIL);
```

```
ACTION
ARTBE:=ARTBE UNION UAR;

% Spawns anomalies to be defined.
EVENT NowToDefine(UAR; ARTD)
LOCAL t: ARFormType;
GUARD
UAR IN UpdatedAR
SUCH THAT
FORALL t IN UAR t.State=ToDefine;
ACTION
ARTD:=ARTD UNION UAR;

% Spawns pending anomalies.
EVENT NowPending(UAR; ARP,HRAR)
LOCAL
t: ARFormType;
r: RIMOFormType;
h: TaskAssignmentType;
GUARD
UAR IN UpdatedAR,
HRFR IN HRForRIMO,
SUCH THAT
FORALL t IN UAR
(t.State=Pending AND t.Instance=NIL);
ACTION
ARP:=ARP UNION UAR;
FOR r IN UAR.RIMO
h.Actor:=r.Actor;
h.Deadline:=r.Deadline;
HRAR:=HRAR UNION h;
END

% Spawns cancelled anomalies.
EVENT NowCancelled(UAR; ARC,SR)
LOCAL t: ARFormType;
GUARD
```

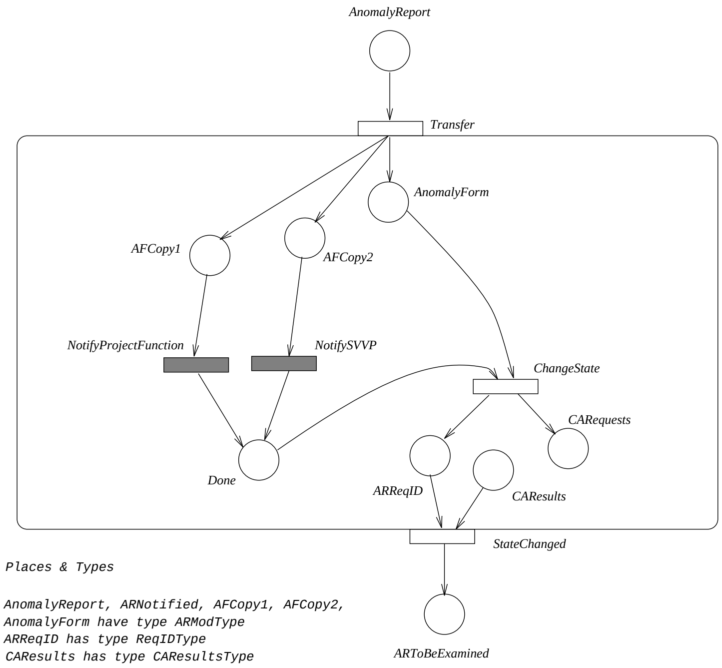
```
UAR IN UpdatedAR
SUCH THAT
FORALL t IN UAR t.State=Cancelled;
% Issues suspension requests for active
% copies to be later cancelled.
FOR t IN UAR
SR:=SR UNION t.Instance.ID;
END

% Restarts active copies of anomalies
% previously in 'Suspended' state and now
% running again.
EVENT NowRestart(UAR,SAC; ACTBR)
LOCAL
t: ARFormType;
s: ActiveCopyType;
GUARD
UAR IN UpdatedARs,
SAC IN SuspendedActiveCopies
SUCH THAT
FORALL t IN UAR, s IN SAC
(t.Status=Pending OR t.Status=PartiallySolved)
AND (t.Instance.ID=s.ID);
ACTION
ARTBR:=ARTBR UNION SAC;

% Sends notification to the anomaly originator
% about anomaly's rejection by CCB.
EVENT NotifyToOriginator1(ARR; D,AR)
LOCAL t: ARFormType;
GUARD
AR IN ARRejected;
ACTION
AR:=AR UNION ARR;
% Tool invocation.
FOR t IN ARR
NotifyTool.Invoke(t.Drafter,t,'AnomalyRejected');
```

```
D:=NotifyTool.ToolOK;
END;

% Sends notification to the anomaly originator
% that more information is needed to evaluate
% anomaly.
EVENT NotifyToOriginator2(ARTD; D,ART)
LOCAL t: ARFormType;
GUARD
ARTD IN ARToDefine;
ACTION
ART:=ART UNION ARTD;
% Tool invocation.
FOR t IN ARTD
NotifyTool.Invoke(t.Drafter,t,'AnomalyToDefine');
D:=NotifyTool.ToolOK;
END;

% Transfers to output places AR rejected.
EVENT TransferRejected(ARR,D1; AR)
GUARD
ARR IN ARRejected,
D IN Done1
SUCH THAT
CARD(ARR)=CARD(D1);
ACTION
AR:=AR UNION ARR;

% Transfers to output places AR cancelled.
EVENT TransferCancelled(ARC,SAC; AC)
LOCAL
t: ARFormType;
s: ActiveCopyType;
GUARD
ARC IN ARCancelled,
SAC IN SuspendedActiveCopies
SUCH THAT
```

```
FORALL t IN ARC, s IN SAC
t.Instance.ID=s.ID;
ACTION
AC:=AC UNION ARC;

% When notifications are performed, transfers
% tokens carrying ARs in state 'ToDefine' to
% output place.
EVENT TransferToDefine(ARTD,D2; ATD)
GUARD
ARTD IN AnomalyToDefine,
D2 IN Done2
SUCH THAT
CARD(ARTD)=CARD(D2);
ACTION
ATD:=ATD UNION ARTD;
```

ARNotification Activity, shown in Fig. A.21 performs the anomaly notification. It takes an AR from its input place and notifies both Project Function and SVVP that a new AR is present in Configuration Archive. At the same time, it asks Configuration Management for a state change on the considered AR taking the AR form the 'Originated' to the 'Notified' state. The activity ends filling the output place with the updated Anomaly Report. Here follows the formal description of transition:

```
% Transfers the AR to be notified into the
% activity body.
EVENT Transfer(AR; AF,AFC1,AFC2)
GUARD
AR IN AnomalyReport;
ACTION
AF:=AR;
AFC1:=AR;
AFC2:=AR;

% Sends notification to Project Function that
% an AR has been sent to Configuration Management.
EVENT NotifyProjectFunction(AFC1; D)
```




```
Places & Types

AnomalyReport, ARNotified, AFCopy1, AFCopy2,
AnomalyForm have type ARModType
ARReqID has type ReqIDType
CAResults has type CAResultsType
CARequests has type CARequestsType
Done has type ConditionType
```

Figura A.21: **ARNotification Activity.** The implementation.

```
GUARD
AFC1 IN AFCopy1;
ACTION
% Tool invocation.
NotifyTool.Invoke(ProjectFunction,AFC1,'New Anomaly Report');
D:=NotifyTool.ToolOK;

% Sends notification to SVVP that an AR
% has been sent to Configuration Management.
EVENT NotifySVVP(AFC2; D)
GUARD
AFC2 IN AFCopy2;
ACTION
% Tool invocation.
NotifyTool.Invoke(SVVP,AFC2,'New Anomaly Report');
D:=NotifyTool.ToolOK;

% Issues a request to change the anomaly
% state to 'Notified'.
EVENT ChangeState(AM,D; AR,CARq)
LOCAL c: CARequestsType;
GUARD
AM IN AnomalyModule,
D IN Done
SUCH THAT
CARD(Done)=2;
ACTION
c.OpReq:=ChangeState;
c.ItemRef:=AM.ID;
c.State:=Notified;
c.ReqID:=c.GenReqID();
AR:=c.ReqID;
CARq:=CARq UNION c;

% The state change is done.
EVENT StateChanged(AR,CARs; ARTBE)
GUARD
```

```
AR IN ARReqID,
CARs IN CAResults
SUCH THAT
AR=CARs.ReqID;
ACTION
ARTBE:=ARTBE UNION CARs.ItemObj;
```

SetupCCB activity, shown in Figure A.22, selects human resources to employ within the CCB. This activity is not properly modeled with SLANG, since there is no information available explaining procedures to execute during this activity: in other words, the task must be accomplished in a non-automatizable way by human resources. Thus, we take the real work performed by SVVP as a black-box: we model only the retrieving of documents needed by SVVP to perform its task. This kind of mechanism will be found repeatedly in the rest of the specification. Looking at the net, the upper piece of the net is devoted to documentation retrieving, by means of the Configuration Management access mechanism described in Section A.2. All the documentation is collected by the WaitForSVVP1 transition: the name of this event means that after the transition is fired the net gives the control to external actors (SVVP) and suspends itself, waiting for results of external operations. These results are communicated to the net by means of user interface places, like the SummonCCB place. Note that the SVVP is informed of the availability of needed documentation through a notification. The WaitForSVVP1 transition allocates human resources contained in the human place SVVP: since this place is shared with the rest of the net, human resources taken from this place are no more available to other tasks. The place SummonCCB contains the information about the decision to summon (or not) CCB. If the CCB is not to be summoned, the SVVP human resources are reallocated and the set of ARs examined is left in the waiting list. By converse, if the CCB must be summoned, SVVP sends letters to CCB members to verify their availability for the selected date. If some of CCB member is not available, SVVP reschedules the meeting and re-sends the letters. If not, the CCBComposition and the set of ARs to be examined are output to be used by the CCBMeeting activity and human resources are reallocated. The following is the formal specification of activity's transitions:

```
% Transfers any AR issued to SVVP into
% activity body and retrieves QualityPlan
```

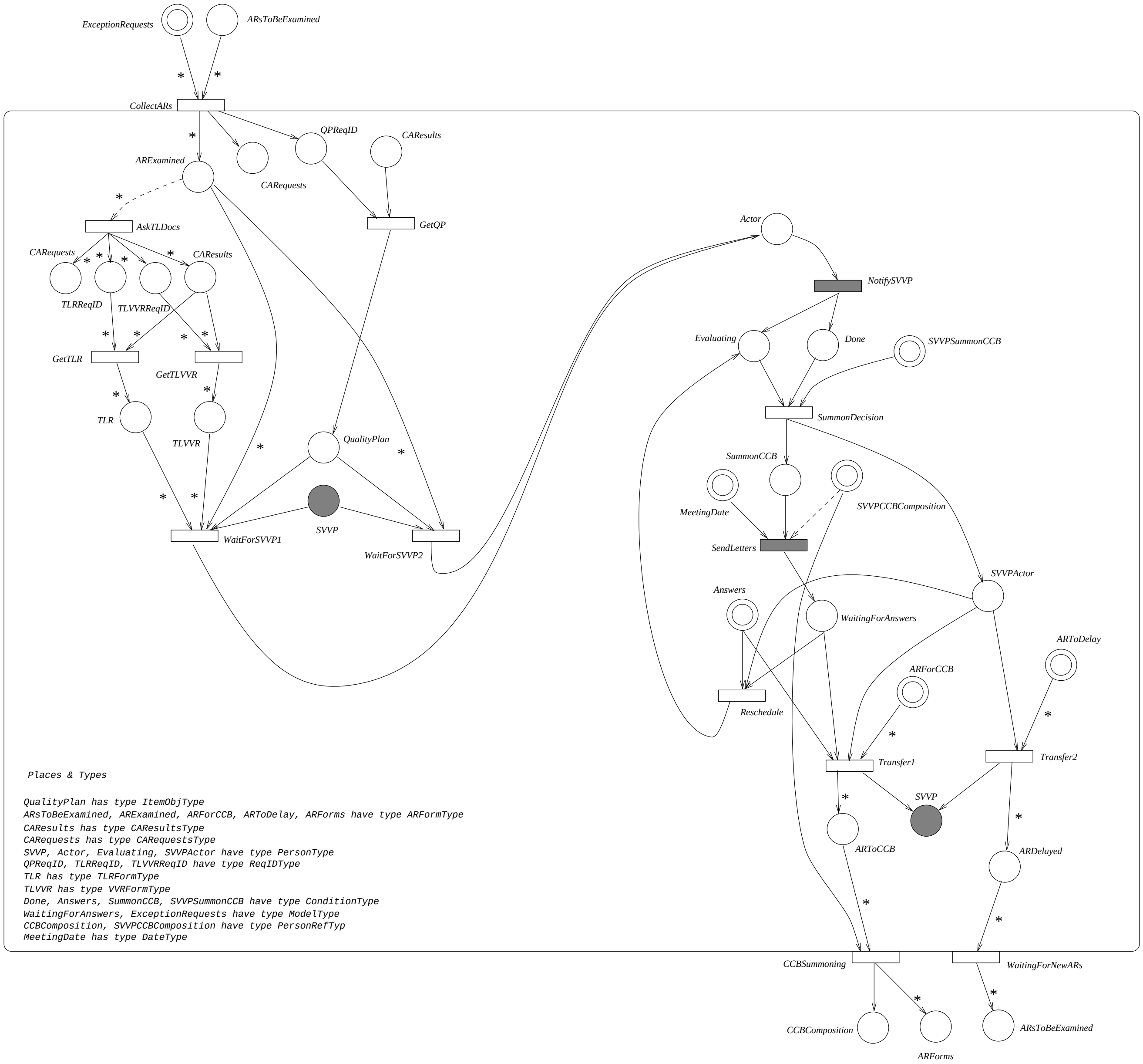


Figura A.22: **SetupCCB Activity.** The implementation.

```
% from Configuration Management.
EVENT CollectARs(ER,ARTBE; ARE,QPid,CARq)
LOCAL c: CARequestsType;
GUARD
ER IN ExceptionRequests,
ARTBE IN ARsToBeExamined
SUCH THAT
CARD(ER)<>0 OR CARD(ER)=0;
ACTION
ARE:=ARTBE;
c.ItemRef:=QualityPlanRef;
c.OpReq:=Retrieve;
c.ReqID:=c.GenReqID;
QPid:=c.ReqID;
CARq:=CARq UNION c;

% Issues requests to Configuration Management to
% retrieve TLRs and TLVVR associated with ARs.
EVENT AskTLDocs(ARE; CARq,TLRid,TLVVRid)
LOCAL
a: ARFormType;
t: TLRFormType;
v: VVRFormType;
c: CARequests;
GUARD
ARE IN ARExamined
SUCH THAT
FORALL t IN ARE
(t.TLRRef<>NIL OR t.TLVVRRef<>NIL);
ACTION
FOR a IN ARE
FOR t IN a.TLRRef
c.ItemRef:=t;
c.OpReq:=Retrieve;
c.ReqID:=c.GenReqID;
TLRid:=c.ReqID;
CARq:=CARq UNION c;
```

```
END
FOR v IN ARE.TLVVRRef
c.ItemRef:=v;
c.OpReq:=Retrieve;
c.ReqID:=c.GenReqID;
TLVVRid:=c.ReqID;
CARq:=CARq UNION c;
END
END

% Takes QualityPlan from Configuration Management.
EVENT GetQP(QPid,CARs; QP)
GUARD
QPid IN QPReqID,
CARs IN CAResults
SUCH THAT
QPid=CARs.ReqID;
ACTION
QP:=CARs.ItemObj;

% Takes TLRs associated to ARs from
% Configuration Management.
EVENT GetTLR(TLRid,CARs; TLR)
LOCAL
t: ReqIDType;
c: CAResultsType;
GUARD
TLRid IN TLRReqID,
CARs IN CAResults
SUCH THAT
FORALL t IN TLRid, c IN CARs TLRid=CARs.ReqID;
ACTION
TLR:=TLR UNION CARs.ItemObj;

% Takes TLVVRs associated to ARs from
% Configuration Management.
EVENT GetTLVVR(TLVVRid,CARs; TLVVR)
```

```
LOCAL
t: ReqIDType;
c: CAResultsType;
GUARD
TLVVRid IN TLVVRReqID,
CARs IN CAResults
SUCH THAT
FORALL t IN TLVVRid, c IN CARs TLVVRid=CARs.ReqID;
ACTION
TLVVR:=TLVVR UNION CARs.ItemObj;

% Provides SVVP with docs to start preliminary
% anomaly evaluation, in the case with TLRs.
EVENT WaitForSVVP1(ARE,QP,SVVP,TLR,TLVVR; A)
LOCAL t: ARFormType;
GUARD
ARE IN ARExamined,
QP IN QualityPlan,
SVVP IN SVVP,
TLR IN TLR,
TLRVVR IN TLVVR
SUCH THAT
FORALL t IN ARE
(CARD(t.TLRRef)=CARD(TLR) AND (t.TLRRef=TLR.ID)) AND
(CARD(t.TLVVRRef)=CARD(TLVVR) AND (t.TLVVRRef=TLVVR.ID));
ACTION
A:=SVVP;

% Provides SVVP with docs to start preliminary
% anomaly evaluation, in the case without TLRs.
EVENT WaitForSVVP2(ARE,QP,SVVP; A)
LOCAL t: ARFormType
GUARD
ARE IN ARExamined,
QP IN QualityPlan,
SVVP IN SVVP
SUCH THAT
```

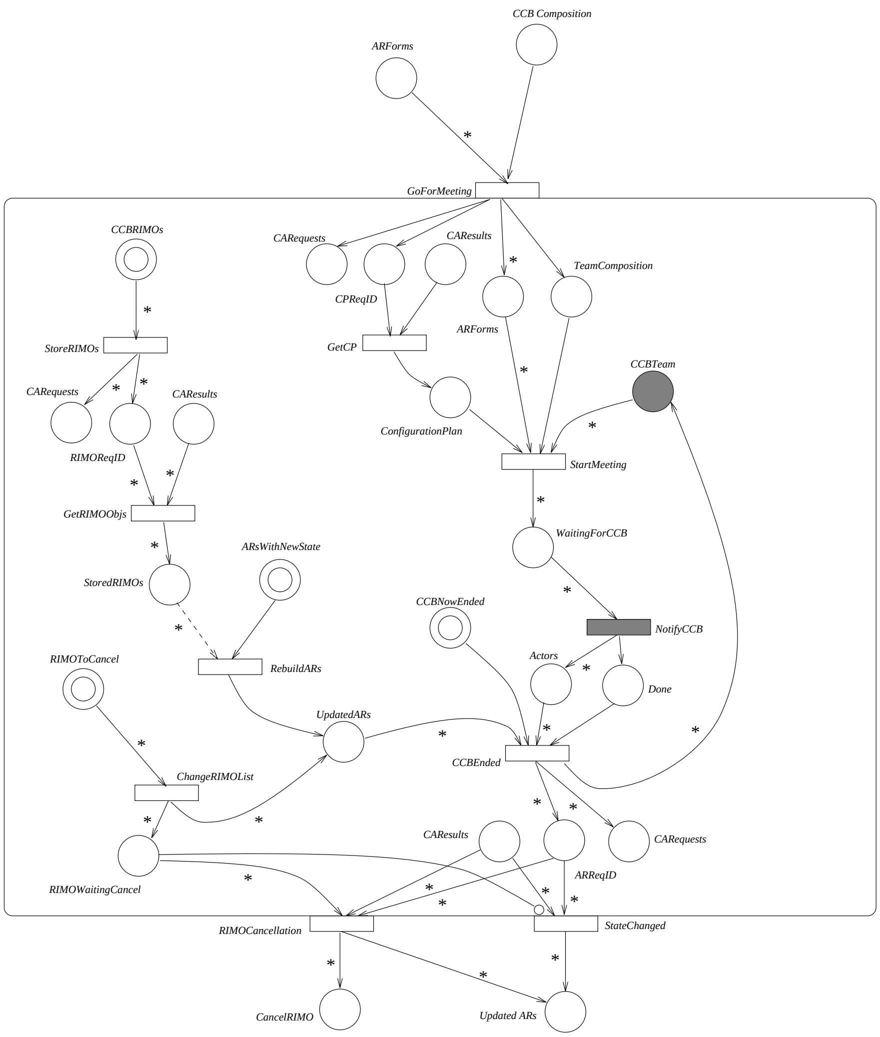
```
FORALL t IN ARE
t.TLRRef=NIL AND t.TLVVRRef=NIL;
ACTION
A:=SVVP;

% Notifies SVVP that documentation is available.
EVENT NotifySVVP (A; D)
GUARD
A IN Actor,
D IN Done;
ACTION
% Tool invocation.
NotifyTool.Invoke(A,'CCB documentation available');
D:=NotifyTool.ToolOK;
E:=A;

% SVVP has determined if CCB must be
% summoned or not.
EVENT SummonDecision(E,D,S; SOut,SA)
GUARD
E IN Evaluating,
D IN Done,
S IN SVVPSummonCCB;
ACTION
SOut:=S;
SA:=E;

% Sends convocation letters to CCB members.
EVENT SendLetters(MD,S,SC; W)
GUARD
MD IN MeetingDate,
S IN SummonCCB,
C IN SVVPCCBComposition
SUCH THAT
S=OK;
ACTION
NotifyTool.Invoke(SC,'CCB Meeting',MD);
```

```
W:=NotifyTool.ToolOK;

% Not all the members are available for
% the meeting. Dates must be rescheduled.
EVENT Reschedule(A,W,SA; E)
GUARD
A IN Answers,
W IN WaitingForAnswers
SUCH THAT
A=NotOK;
ACTION
E:=SA;

% CCB must be summoned.
EVENT Transfer1(A,W,ARFC,S; ARTC,SVVP)
GUARD
A IN Answers,
W IN WaitingForAnswers,
ARFC IN ARForCCB,
S IN SVVPActor
SUCH THAT
A=OK;
ACTION
ARTC:=ARFC;
SVVP:=S;

% CCB must not be summoned.
EVENT Transfer2(S,ARTD; SVVP,ARD)
GUARD
ARTD IN ARToDelay;
ACTION
SVVP:=S;
ARD:=ARTD;

% Docs transfer for CCB.
EVENT CCBSummoning(ARTC,C; COut, ARF)
GUARD
```

```
ARTC IN ARToCCB,
C IN CCBComposition;
ACTION
COut:=C;
ARF:=ARF UNION ARTC;

% Loops waiting new ARs.
EVENT WaitingForNewARs(ARD; ARTBE)
GUARD
ARD IN ARDelayed;
ACTION
ARTBE:=ARTBE UNION ARD;
```

The **CCBMeeting activity** (Fig. A.23) employs the same mechanism described for previous activity, waiting for external task completion. Documentation generated by CCB, i.e. RIMOs and updated ARs, is communicated to the net by user interface places: the contents of these places are sent to Configuration Management for storing and then sent to the event StateChanged. When the change is performed, the updated ARs are output, possibly together with RIMOs whose cancellation is needed. The user interface CCBNowEnded is needed to determine when all ARs and RIMOs are collected. Transitions are so specified:

```
% Starts activity by transferring AR forms and
% composition of CCB; moreover, ask Configuration
% Management for Configuration Plan.
EVENT GoForMeeting(ARF,C; AR,TC,CARq,CPid)
LOCAL c: CARequestsType;
GUARD
ARF IN ARForms,
C IN CCBComposition,
ACTION
AR:=ARF;
TC:=C;
c.ItemRef:=ConfigurationPlanRef;
c.OpReq:=Retrieve;
c.ReqID:=c.GenReqID;
CPid:=c.ReqID;
```




```
Places & Types

ARForms, UpdatedARs,
ARsWithNewState have type ARFormType
CCBRIMOs, StoredRIMOs, RIMOToCancellation, RIMOWaitingCancel, CancelRIMO have type RIMOFormType
Done has type ConditionType
CAResults has type CAResultsType
CARequests has type CARequestsType
CCBComposition,TeamComposition have type PersonRefType
CCBTeam, WaitingForCCB, Actors have type PersonType
ARReqID, RIMOReqID, CPReqID have type ReqIDType
ConfigurationPlan has type ItemObjType
CCBNowEnded has type ModelType
```

Figura A.23: **CCBMeeting Activity.** The implementation.

```
CARq:=CARq UNION c;

% Gets Configuration Plan from CM.
EVENT GetCP(CPid,CARs; CP)
GUARD
CPid IN CPReqID,
CARs IN CAResults
SUCH THAT
CARs.ReqID=CPid;
ACTION
CP:=CARs.ItemObj;

% Suspends the net waiting for CCB results.
EVENT StartMeeting(AR,TC,CP,CCBT; W)
GUARD
AR IN ARs,
C IN TeamComposition,
CP IN ConfigurationPlan,
CCBT IN CCBTeam
SUCH THAT
CARD(TC)=CARD(CCBT);
ACTION
W:=CCBT;

% Sends RIMOs issued by CCB to
% Configuration Management.
EVENT StoreRIMOs(RIMO; CARq,RIMOid)
LOCAL
t: RIMOFormType;
c: CARequestsType;
GUARD
RIMO IN CCBRIMOs;
ACTION
FOR t IN RIMO
c.ItemObj:=t;
c.OpReq:=Send;
c.ReqID:=c.GenReqID;
```

```
RIMOid:=c.ReqID;
CARq:=CARq UNION c;
END

% Gets RIMO objects after Configuration
% Management has stored them.
EVENT GetRIMOObjs(CARs,RIMOid; RIMOObj)
LOCAL
c: CAResultsType;
r: ReqIDType;
GUARD
CARs IN CAResults,
RIMOid IN RIMOReqID
SUCH THAT
FORALL c IN CAResults, r IN RIMOid RIMOid=CARs.ReqID;
ACTION
FOR c IN CARs
RIMOObj:=RIMOObj UNION c.ItemObj;
END

% Appends RIMO references to modified ARs.
EVENT RebuildARs(RIMO,ARWNS; UAR)
LOCAL t: RIMOFormType;
GUARD
ARWST IN ARsWithNewState,
RIMO IN StoredRIMOs
SUCH THAT
FORALL t IN RIMO ARWST.ID=t.AR;
ACTION
UAR:=UAR UNION ARWST;
UAR.RIMO:=UAR.RIMO UNION t.ID;

% Notifies CCB that the needed documentation
% is available.
EVENT NotifyToCCB(W; D,A)
GUARD
W IN Waiting;
```

```
ACTION
% Tool invocation.
NotifyTool.Invoke(W,'Documentation ready');
D:=NotifyTool.ToolOK;
Actors:=W;

% Reallocates CCB members to previous tasks
% and sends the updated AR to Configuration
% Management, for Event Journal updating.
% In addition, prepares HR allocation requests
% for RIMO implementing.
EVENT CCBEnded(UAR,A,D,NCE; CARq,ARid,CCBT,HRFR)
LOCAL c: CARequestsType;
GUARD
UAR IN UpdatedAR,
A IN Actors,
NCE IN NowCCBEnded,
D IN Done;
ACTION
FOR t IN UAR
c.OpReq:=Send;
c.Status:=t.Status;
c.ItemRef:=t.ItemRef;
c.ReqID:=c.GenReqID;
ARid:=c.ReqID;
CARq:=CARq UNION c;
END
CCBT:=A;

% The change has been performed.
EVENT StateChanged(ARid,CARs; UAR)
LOCAL
t: ReqIDType;
c: CAResultsType;
GUARD
ARid IN ARReqID,
CARs IN CAResults
```

```
SUCH THAT
FORALL t IN ARid, c IN CARs (t=c.ReqID AND c.OpOK=TRUE);
ACTION
FOR c IN CARs
UAR:=UAR UNION c.ItemObj;
END

% Updates the ARs by removing the RIMOs
% that must be cancelled.
EVENT ChangeRIMOList(RTC; RWC,UAR)
LOCAL t: RIMOFormType;
GUARD
RTC IN RIMOToCancel;
ACTION
FOR t IN RTC
UAR:=t.AR;
UAR.RIMO:=UAR.RIMO SUBTRACT t;
END

% Transfers RIMOs in the interface place of
% SolveAnomaly activity, to perform actual
% cancellation of RIMO solving.
EVENT RIMOCancellation(RWC,CARs,ARid; CR)
LOCAL
t: ReqIDType;
c: CAResultsType;
GUARD
ARid IN ARReqID,
CARs IN CAResults,
RWC IN RIMOWaitingCancellation
SUCH THAT
FORALL t IN ARid, c IN CARs (t=c.ReqID AND c.OpOK=TRUE);
ACTION
FOR c IN CARs
UAR:=UAR UNION c.ItemObj;
END
CR:=CR UNION RWC;
```

The activity **SolveAnomaly**, shown in Fig. A.24, contains the steps performed by Project Function to update and re-qualificate the items involved in anomaly solution.

Note that the activity starts getting only an AR from its input place, consequently, the solving process involves only items and RIMOs relative to that anomaly.

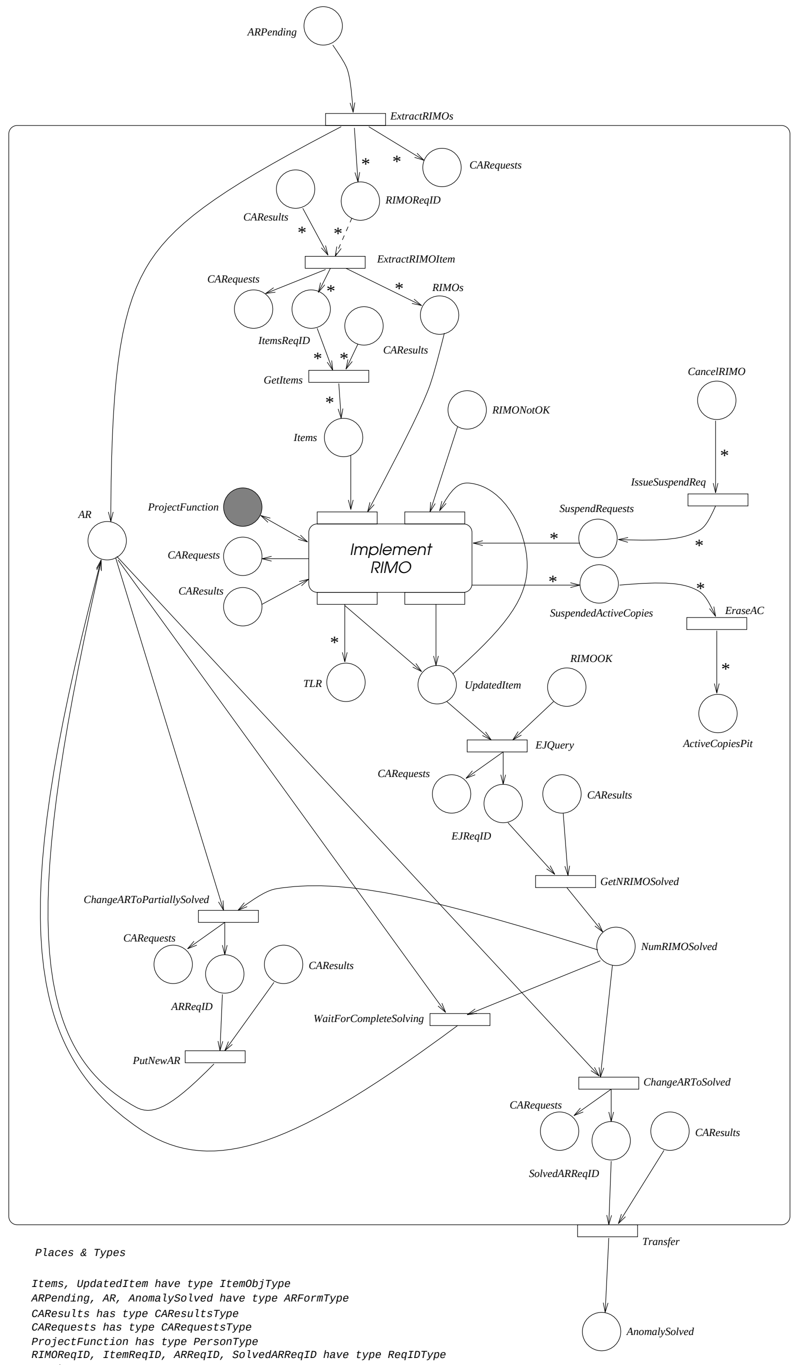
First of all, RIMOs associated to the AR are retrieved form Configuration Archive, together with items for which RIMOs has been issued. They can be possibly sets of tokens. SolvingAnomaly provides a couple item-RIMO associated to the activity ImplementRIMO, which performes the real implementation of updates. This activity outputs the updated item and possibly a set of Test Launch Reports generated by requalification, if expected. This activity takes care for sending items to Configuration Archive, consequently enacting the review process. This will generate a Verification & Validation Report that will be evaluated by Configuration Management and will generate some tokens in (possibly both) RIMOOK and RIMONotOK. If some of RIMOs contained in RIMONotOK are relative to the item in UpdatedItem, the implementation process restarts to fit incompleteness in RIMO solution. By converse, if the RIMO matching with the updated item is in RIMOOK, a query in Event Journal is performed to determine the number of RIMOs currently solved. This number is compared to the number of RIMOs associated to AR, determining three possible cases:

- All RIMOs have been solved. The AR becomes 'Solved' too, and activity ends.

- Not all RIMOs are solved, but it is the first time this check is performed, consequently the AR must change its state from 'Pending' to 'PartiallySolved'.

- Not all RIMOs are solved and the AR is already 'PartiallySolved': no action is needed except for waiting for new successfully updated items.

The intermediate place CancelRIMO is used to determine the cancellation of the implementation of specified RIMO, as described before. The cancellation is actually performed by filling in SuspendRequest the identifiers of the active copies associated to the RIMOs. When the active copies appear in SuspendedActiveCopies, meaning that the related instances have been




```
Places & Types

Items, UpdatedItem have type ItemObjType
ARPending, AR, AnomalySolved have type ARFormType
CAResults has type CAResultsType
CARequests has type CARequestsType
ProjectFunction has type PersonType
RIMOReqID, ItemReqID, ARReqID, SolvedARReqID have type ReqIDType
TLR has type TLRFormType
RIMOOK, RIMONotOK have type RIMOFormType
NumRIMOSolved has type INTEGER
SuspendRequest, SuspendedActiveCopies, CancelRIMO,
ActiveCopiesPit have type ActiveCopyType
```

Figura A.24: **SolveAnomaly Activity.** The implementation.

really suspended, they are removed by the EraseAC event, which prevent the system from restarting those activities. These "garbage" active copies are collected in the ActiveCopiesPit, and will be definitely removed from the system when the activity SolveAnomaly will end. Here follows transitions' SLANG implementation:

```
% Extracts RIMOs associated to ARs in process.
EVENT ExtractRIMOs(ARP; RIMOid,CARq,AR)
LOCAL
c: CARequestsType;
t: RIMOFormType;
GUARD
ARP IN ARPending;
ACTION
FOR t IN ARP.RIMO
c.ItemRef:=t;
c.OpReq:=Retrieve;
c.ReqID:=c.GenReqID;
RIMOid:=c.ReqID;
CARq:=CARq UNION c;
END
AR:=ARP;

% Extracts the configuration item associated
% with the RIMO.
EVENT ExtractRIMOItem(CARs,RIMOid; RIMO,CARq,Iid)
LOCAL
r: CAResultsType;
t: ReqIDType;
c: CARequestsType;
GUARD
CARs IN CAResults,
RIMOid IN RIMOReqID
SUCH THAT
FORALL r IN CARs, t IN RIMOid r.ReqID=t;
ACTION
FOR r IN CARs
```

```
RIMO:=RIMO UNION r.ItemObj;
c.ItemRef:=r.Item;
c.OpReq:=Retrieve;
c.ReqID:=c.GenReqID;
Iid:=c.ReqID;
CARq:=CARq UNION c;
END

% Gets items to update from Configuration
% Management.
EVENT GetItems(Iid,CARs; I)
LOCAL
c: CAResultsType;
t: ReqIDType;
GUARD
Iid IN ItemsReqID,
CARs IN CAResults
SUCH THAT
FORALL c IN CARs, t IN RIMOid t=c.ReqID;
ACTION
FOR c IN CARs
I:=I UNION c.ItemObj;
END

% Every time an updated item is ready (after
% being sent to Configuration Management) and its
% RIMO is correctly implemented, it is performed
% a check to determine if all RIMOs associated to
% an AR have been solved.
EVENT EJQuery(UI,ROK; CARq,EJid)
LOCAL
c: CARequestsType;
u: ItemObjType;
r: RIMOFormType;
GUARD
UI IN UpdatedItem,
ROK IN RIMOOK
```

```
SUCH THAT
FORALL r IN ROK, u IN UI r.ID IN u.RIMO;
ACTION
c.OpReq:=EJSearch;
c.Pred:='Number of RIMOs associated to AR that
    are already in Solved state';
c.ReqID:=c.GenReqID;
EJid:=c.ReqID;
CARq:=CARq UNION c;

% Gets the number of RIMOs solved associated
% with an AR.
EVENT GetNRIMOSolved(EJid,CARs; NRS)
GUARD
CARs IN CAResults,
EJid IN EJReqID
SUCH THAT
CARs.ReqID=EJid;
ACTION
NRS:=CARs.EJResults;

% The AR is not in PartiallySolved state and
% has some RIMO not yet solved, consequently
% the next state must be PartiallySolved.
EVENT ChangeARToPartiallySolved(AR,NRS; CARq,ARid)
LOCAL c: CARequestsType;
GUARD
AR IN AR,
NRS IN NumRIMSolved
SUCH THAT
CARD(AR.RIMO)<>NRS AND AR.Status<>PartiallySolved;
ACTION
c.ItemRef:=AR.ID;
c.OpReq:=ChangeState;
c.Status:=PartiallySolved;
c.ReqID:=c.GenReqID;
ARid:=c.ReqID;
```

```
CARq:=CARq UNION c;

% The new AR is available for next check.
EVENT PutNewAR(ARid,CARs; AR)
GUARD
CARs IN CAResults,
ARid IN ARReqID
SUCH THAT
ARid=CARs.ReqID;
ACTION
AR:=CARs.ItemObj;

% The AR is already in the PartiallySolved state,
% consequently, the only operation to perform is to
% wait for the accomplishment of the last RIMOs.
EVENT WaitForCompleteSolving(AR,NRS; AROut)
GUARD
AR IN AR,
NRS IN NumRIMSolved
SUCH THAT
CARD(AR.RIMO)<>NRS AND AR.Status=PartiallySolved;
ACTION
AROut:=AR;

% All RIMO associated with the AR have been solved,
% so the new state for AR is Solved.
EVENT ChangeARToSolved(AR,NRS; CARq,SARid)
LOCAL c: CARequestsType;
GUARD
AR IN AR,
NRS IN NumRIMSolved
SUCH THAT
CARD(AR.RIMO)=NRS;
ACTION
c.ItemRef:=AR.ID;
c.OpReq:=ChangeState;
c.Status:=Solved;
```

```
c.ReqID:=c.GenReqID;
SARid:=c.ReqID;
CARq:=CARq UNION c;

% Solved AR is transferred out off activity body.
EVENT Transfer(SARid,CARs; AS)
GUARD
SARid IN SolvedARReqID,
CARs IN CAResults
SUCH THAT
CARs.ReqID=SARid;
ACTION
AS:=AS UNION CARs.ItemObj;

% Used to suspend a RIMO solution.
% The RIMO will be cancelled by subsequent
% transitions.
EVENT IssueSuspendReq(CR; SR)
LOCAL t: RIMOFormType;
GUARD
CR IN CancelRIMO;
ACTION
FOR t IN CR
SR:=SR UNION t.Instance.ID;
END

% Used to cancel a RIMO solution.
% The cancelling is actually implemented by
% removing the active copy associated with the RIMO
% from SuspendedActiveCopies, so erasing the link
% between the SPADE interpreter and the reference to
% the active copy. The token is then put into a 'pit',
% where garbage active copies are collected. It will
% be erased automatically by the system when the
% instance of SolveAnomaly is terminated.
EVENT Erase(SAC; ACP)
GUARD
```

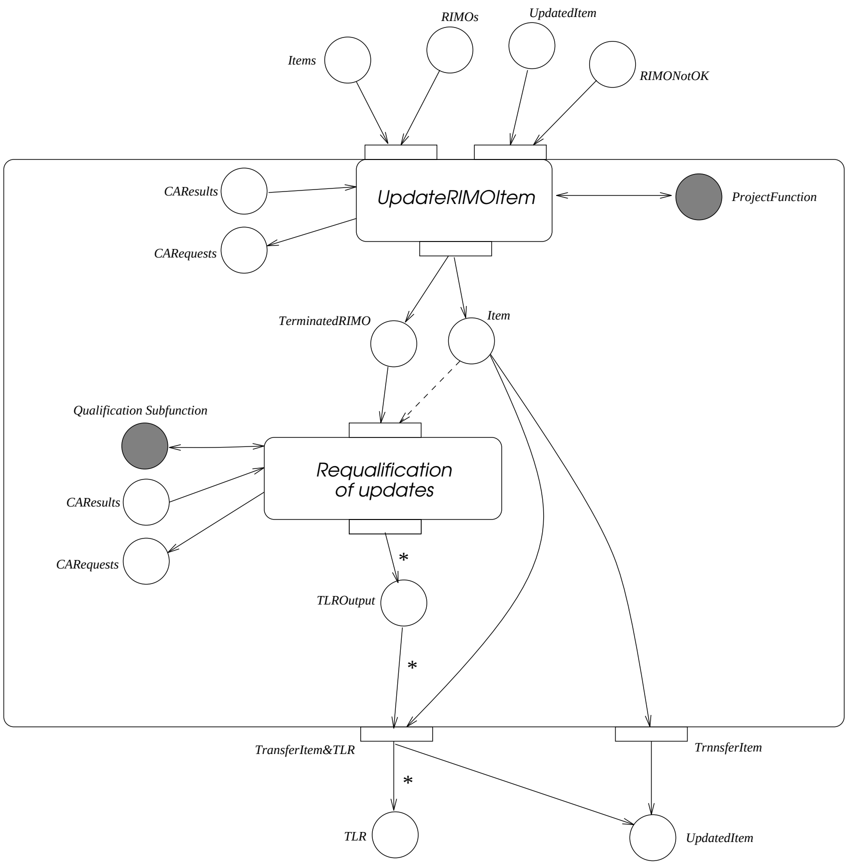
```
SAC IN SuspendedActiveCopies;
ACTION
ACP:=ACP UNION SAC;
```

The **ImplementRIMO activity**, shown in Fig. A.25, contains the subtask of implementation and requalification of updates requested. This activity can start two ways:

- Normal implementation of updates. An item and the related RIMO are fetched from input places Items and RIMOs, which contain all the items and RIMOs currently employed to solve the anomaly.
- Completion of RIMO implementation. Started when review evidenced the lack of some feature required in the updated item. RIMOs not completely implemented are present in RIMONotOK place, which is an intermediate place of ManageConfigurationArchive activity. One of such RIMOs is fetched together with the updated item.

Note that the requalification is performed only for those configuration items for which the testing is requested: this information is contained in the field `ToBeTested` of `ItemObjType` and we suppose it is properly filled by some of the other activities we have not specify yet. Only if `ToBeTested=TRUE` the activity RequalificationOfUpdates is entered, otherwise it is skipped and the activity ImplementRIMO ends without any TLR output.

```
% If the item is subject to testing, both TLR
% and item are transferred.
EVENT TransferItem&TLR(TLRO,I; TLR,UI)
LOCAL
t: TLRFormType;
GUARD
TLRO IN TLROutput,
I IN Item
SUCH THAT
FORALL t IN TLRO
I.ToBeTested=TRUE AND t.Item=I.ID;
ACTION
TLR:=TLR UNION TLRO;
```



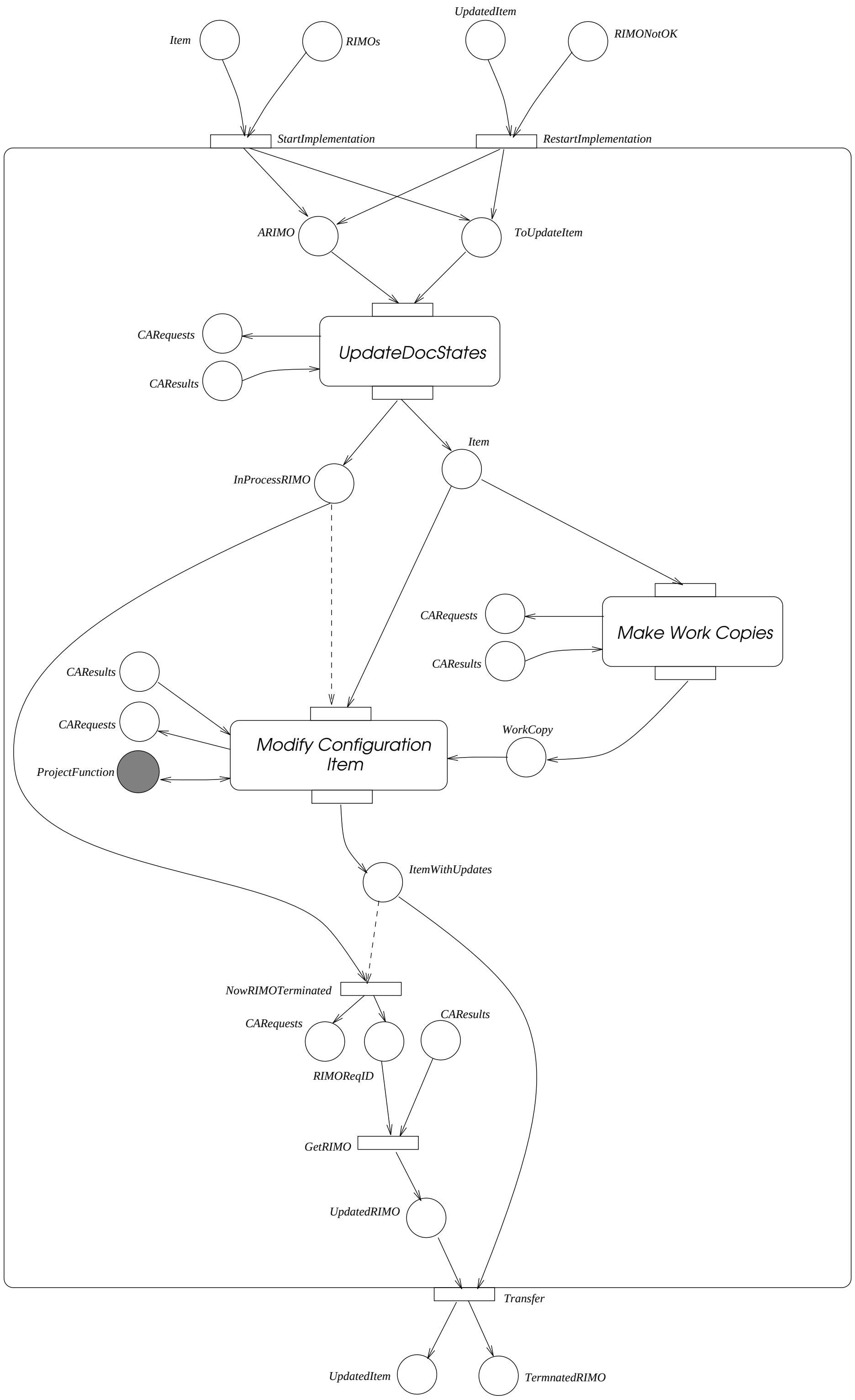

```
Places & Types

Items, Item, UpdatedItem have type ItemObjType
CAResults has type CAResultsType
CARequests has type CARequestsType
ProjectFunction, QualificationSubfunction have type PersonType
RIMOReqID, ItemReqID, ARReqID have type ReqIDType
TLR, TLROutput have type TLRFormType
RIMOOK, RIMONotOK, RIMOs, TerminatedRIMO have type RIMOFormType
```

Figura A.25: **ImplementRIMO Activity.** The implementation.

```
UI:=UI UNION I;

% If the item is not subject to testing, only
% the item must be transferred.
EVENT TransferItem(I; UI)
GUARD
I IN Item
SUCH THAT
I.ToBeTested=FALSE;
ACTION
UI:=I;
```

Fig. A.26 shows the implementation of **UpdateRIMOItem activity**. As described before, this activity can be started directly after RIMO issuing or restarted after review, due to the lack of some feature requested in updated item. First of all, item and RIMO's states are updated according to the current processing by UpdateDocStates activity. After this activity has output the version of object carrying a new state, real implementation can actually start. This is carried on in ModifyConfigurationItem activity, which can work in parallel together with MakeWorkCopies. The rationale under this is that implementor can start working by reading RIMOs and item to update while waiting for Configuration Management to provide the the item copy in Development Archive to be updated. When the updated item is output, the Configuration Management is asked for a state transition over the RIMO, carrying it into 'Terminated' state. After this is done, the activity ends by transferring the new versions of the item and related RIMO.

```
% Starts the implementation process on an
% item that owns at least a RIMO. This is
% the first implementation attempt.
EVENT StartImplementation(I,RIMO; ARIMO)
GUARD
I IN Item,
RIMO IN RIMOs
SUCH THAT
RIMO.Item=I.ID;
ACTION
```




```
Places & Types

RIMOReqID has type ReqIDType
ProjectFunction has type PersonType
RIMOs, ARIMO, InProcessRIMO, UpdatedRIMO, TerminatedRIMO, RIMONotOK have type RIMOFormType
Item, UpdatedItem, ToUpdateItem, ItemWithUpdates have type ItemObjType
CAResults has type CAResultsType
CARequests has type CARequestsType
WorkCopies have type ItemObjType
```

Figura A.26: **UpdateRIMOItem Activity.** The implementation.

```
ARIMO:=RIMO;
TUI:=I;

% Restart the implementation process because
% a review has evidenced that not all RIMO
% have been correctly solved. Nevertheless,
% this fact does not need an AR issuing.
EVENT RestartImplementation(UI,ROK; ARIMO,TUI)
GUARD
UI IN UpdatedItem,
ROK IN RIMOOK
SUCH THAT
ROK.Item=UI.ID;
ACTION
TUI:=UI;
ARIMO:=ROK;

% Implementation of updates is ended, and
% the RIMO must take the 'Terminated' new state.
EVENT NowRIMOTerminated(IWU,IPRIMO; CARq,Rid)
LOCAL c: CARequestsType;
GUARD
IWU IN ItemWithUpdates,
IPRIMO IN InProcessRIMO;
ACTION
c.ItemRef:=IPRIMO;
c.OpReq:=ChangeState;
c.Status:=Terminated;
c.ReqID:=c.GenReqID;
Rid:=c.ReqID;
CARq:=CARq UNION c;

% Takes the updated object from
% Configuration Archive.
EVENT GetRIMO(Rid,CARs; URIMO)
GUARD
Rid IN RIMOReqID,
```

```
CARs IN CAResults
SUCH THAT
CARs.ReqID=Rid;
ACTION
URIMO:=CARs.ItemObj;

% Transfers both the item with updates and
% the RIMO with the new state out of activity
% body, available for further operations.
EVENT Transfer(URIMO,IWU; UI,TRIMO)
GUARD
URIMO IN UpdatedRIMO,
IWU IN ItemWithUpdates;
ACTION
TRIMO:=URIMO;
UI:=IWU;
```

The task of requalificating a configuration item is different according with the nature of the item, as shown in **RequalificationOfUpdates activity**:

- If the item with RIMOs is an SCU or a design, the related tests must be re-executed.
- If the item with RIMOs is a Test Code, this test must be re-executed over all the items that have this test among the set of associated tests.

This is what is actually performed by the activity implementation. The associated tests or associated items are retrieved from Configuration Archive and executed by tool invocation. TLRs so generated are sent to Configuration Management (which will start the review process) and output to outer activities.

```
% Transfers a couple item-RIMO into
% the activity body, if the item
% needs requalification.
EVENT Transfer(UI,TRIMO; I,RIMO)
UI IN UpdatedItem,
TRIMO IN TerminatedRIMO
```

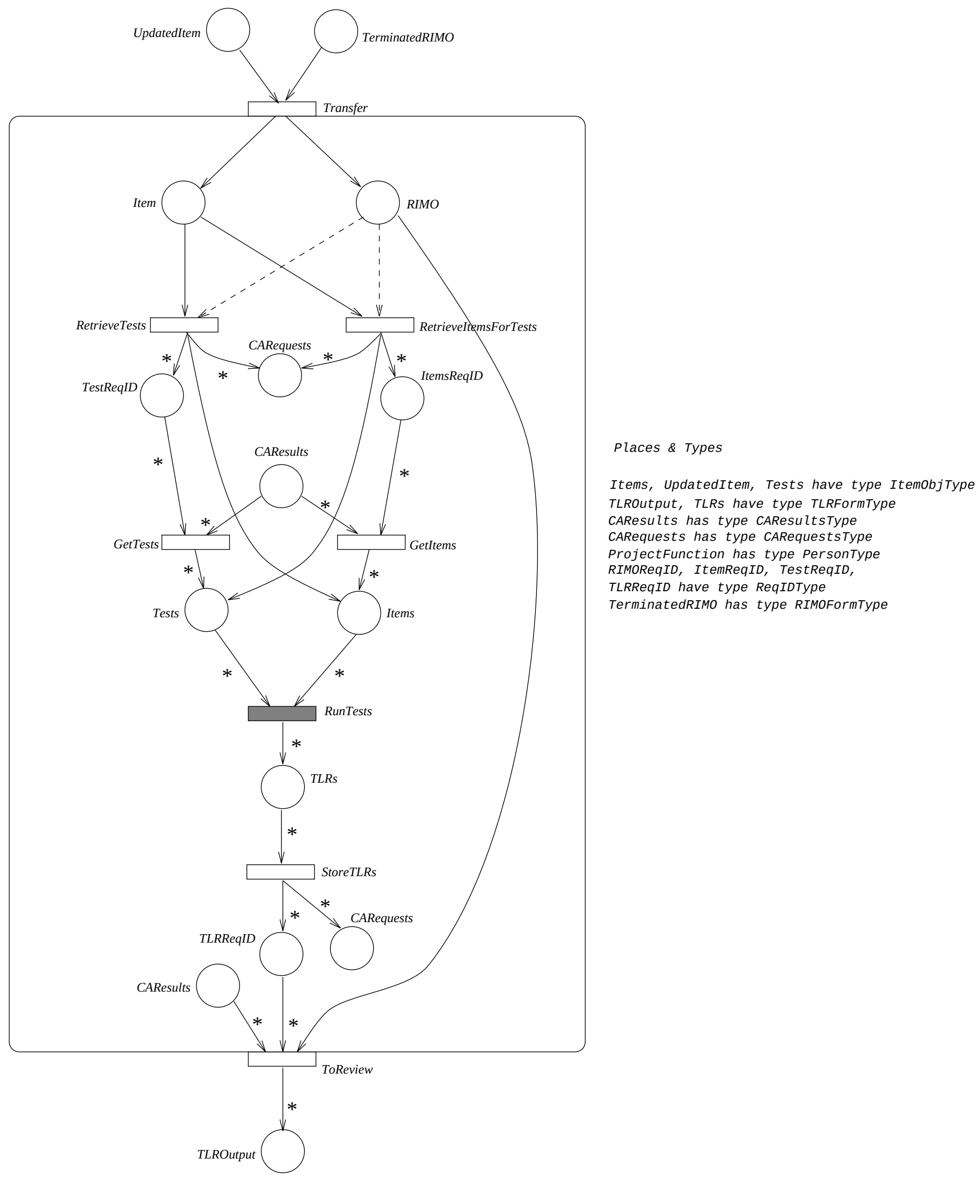


Figura A.27: **RequalificationOfUpdates Activity.** The implementation.

```
SUCH THAT
UI.ID=TRIMO.Item AND UI.Kind=Normal;
ACTION
I:=UI;
RIMO:=RIMO;

% Retrieves the tests associated with the item
% and specified by the RIMO.
EVENT RetrieveTests(I,RIMO; Tid,CARq,IOut)
LOCAL
c: CARequestsType;
t: RIMOFormType;
GUARD
I IN Item,
RIMO IN RIMO
SUCH THAT
I.ToBeTested=FALSE;
ACTION
FOR t IN RIMO.Requisite
c.ItemRef:=t;
c.OpReq:=Retrieve;
c.ReqID:=c.GenReqID;
Tid:=c.ReqID;
CARq:=CARq UNION c;
END
IOut:=I;

% Retrieves the tests associated with the item
% and specified by the RIMO.
EVENT RetrieveItemsForTests(I,RIMO; Iid,CARq,T)
LOCAL
c: CARequestsType;
t: TLRFormType;
GUARD
I IN Item,
RIMO IN RIMO
SUCH THAT
```

```
I.ToBeTested=TRUE;
ACTION
FOR t IN RIMO.Requisite
c.ItemRef:=t;
c.OpReq:=Retrieve;
c.ReqID:=c.GenReqID;
Iid:=c.ReqID;
CARq:=CARq UNION c;
T:=I;

% Gets tests from Configuration Management.
EVENT GetTests(CARs,Tid; T)
LOCAL
c: CAResultsType;
t: ReqIDType;
GUARD
CARs IN CAResults,
Tid IN TestsReqID
SUCH THAT
FORALL c IN CARs, t IN Tid c.ReqID=t;
ACTION
FOR c IN CARs
T:=T UNION c.ItemObj;
END

% Gets items from Configuration Management.
EVENT GetItems(CARs,Iid; I)
LOCAL
c: CAResultsType;
t: ReqIDType;
GUARD
CARs IN CAResults,
Iid IN ItemReqID
SUCH THAT
FORALL c IN CARs, t IN Iid c.ReqID=t;
ACTION
FOR c IN CARs
```

```
I:=I UNION c.ItemObj;
END

% Runs tests.
EVENT RunTests(T,I; TLR)
GUARD
T IN Tests,
I IN Items,
ACTION
% Tool invocation.
RunTestTool.Invoke(T,I);
TLR:=ToolResult;

% Sends TLRs produced by test running
% to Configuration Management.
EVENT StoreTLRs(TLR; CARq,TLRid)
LOCAL
c: CARequestsType;
t: TLRFormType;
GUARD
TLR IN TLRs;
ACTION
FOR t IN TLR
c.OpReq:=Send;
c.ItemObj:=t;
c.ReqID:=c.GenReqID;
CARq:=CARq UNION c;
END

% Transfers TLRs and RIMO to review activity,
% as support documentation.
EVENT ToReview(CARs,TLRid,RIMO; TLR)
LOCAL
c: CAResultsType;
t: ReqIDType;
GUARD
CARs IN CAResults,
```

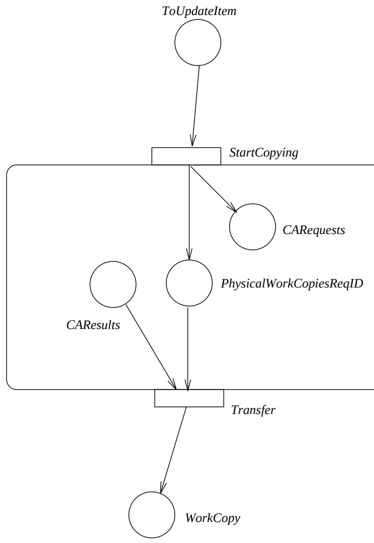


Figura A.28: **MakeWorkCopies Activity.** The implementation.

```
TLRid IN TLRReqID
RIMO IN RIMO
SUCH THAT
FORALL c IN CARs, t IN TLRid c.ReqID=t;
ACTION
FOR c IN CARs
TLR:=c.ItemObj;
END
```

The activity **MakeWorkCopies** (Fig A.28) is fully automatized: its purpose is to provide a work copy of the configuration item to be updated into Development Archive. Its structure is very simple: it contains only an access to Configuration Management to retrieve the copies needed. The textual specification is the following:

```
% Issues the copy request to Configuration Management.
EVENT StartCopying(TUI; PWCid,CARq)
```

```
LOCAL c: CARequestsType;
GUARD
TUI IN ToUpdateItem;
ACTION
c.OpReq:=Duplicate;
c.Destination:='Work Path';
c.ItemRef:=TUI.ID;
c.ReqID:=c.GenReqID;
CARq:=CARq UNION c;
PWCid:=ReqID;

% Gets the physical object.
EVENT Transfer(PWCid,CARs; WC)
GUARD
PWC IN PhysicalWorkCopies,
CARs IN CAResults
SUCH THAT
CARs.ReqID=PWCid AND CARs.OpOK=TRUE;
ACTION
WC:=CARs.ItemObj;
```

ModifyConfigurationItems is the central implementing activity: it relies on the same waiting mechanism described above. In fact, we have no information about tools and techniques used to implement software, consequently we model it as an external activity. Input for this activity is a couple item-RIMO, which are provided to implementor. He/she starts examining them, until the work copy is ready, provided by the activity MakeWorkCopies. After the copy is delivered, the net suspends itself waiting for the new version of the item to be updated, which will appear in the user interface place UpdatedItem. When the event EndUpdates fires, the process actor who made the updates is reallocated to previous work and the new item is sent to Configuration Management.

The activity handles also the case of a missed deadline: the event NowTimedOut, in fact, is fired when such a situation occurs, provoking a state transition on the RIMO, towards the 'TimedOut' state. The formal specification:

```
% Transfers RIMO and related configuration item
```

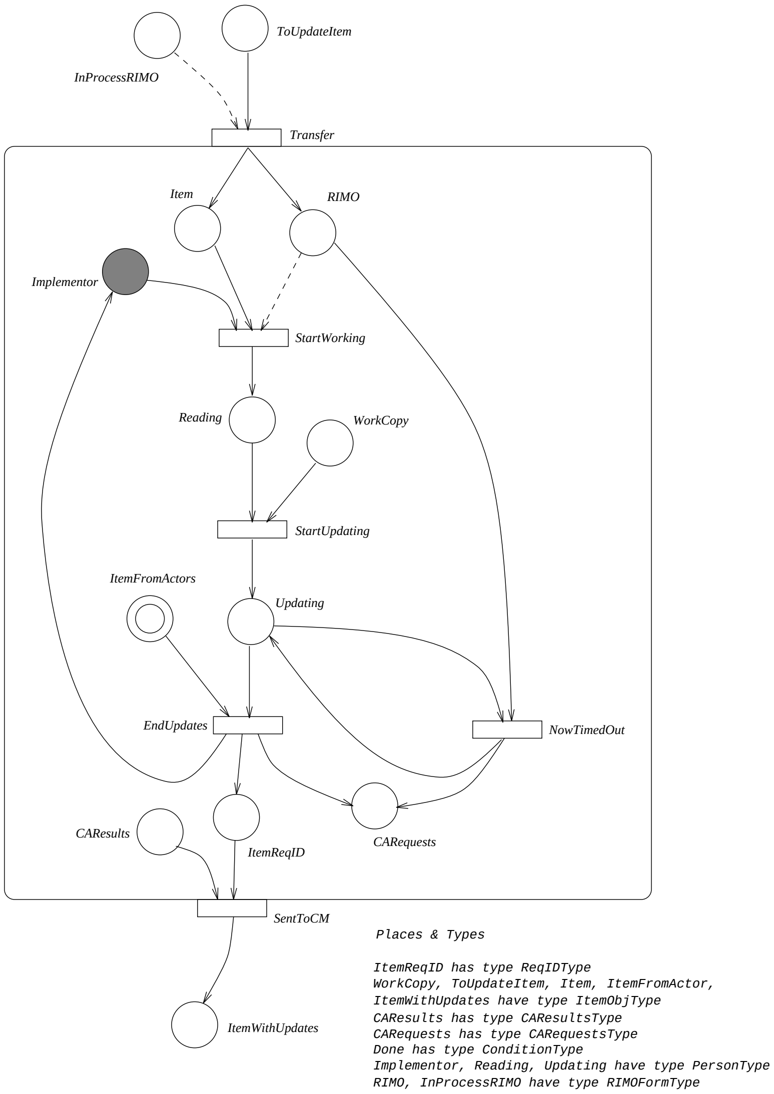
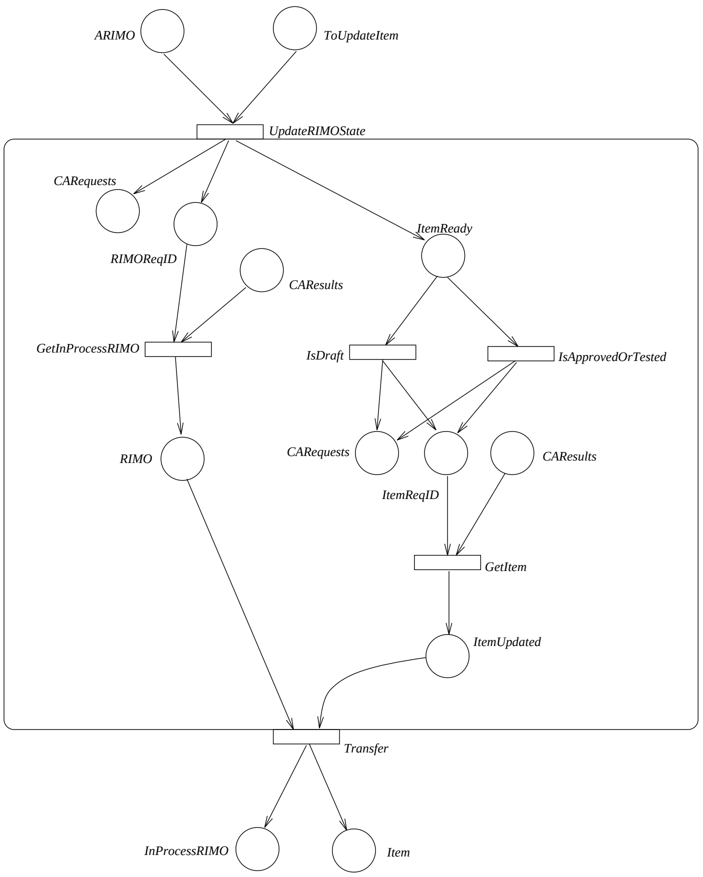


Figura A.29: **ModifyConfigurationItems Activity.** The implementation.

```
% into activity body.
EVENT Transfer(IPRIMO,TUI; RIMO,I)
GUARD
IPRIMO IN InProcessRIMO,
TUI IN ToUpdateItem;
ACTION
RIMO:=IPRIMO;
I:=TUI;

% Starts implementor's work. No modification
% is allowed.
EVENT StartWorking(RIMO,I,Im; R)
GUARD
RIMO IN RIMO,
I IN Item,
Im IN Implementor;
ACTION
R:=Im;

% Implementor's work is going on, with
% modifications allowed on work copies.
EVENT StartUpdating(R,WC; U)
GUARD
R IN Reading,
WC IN WorkCopy;
ACTION
U:=R;

% The new version of configuration item under
% update is ready to be delivered to
% Configuration Management.
EVENT EndUpdates(IFA,U; I,CARq,Iid)
LOCAL c: CARequestsType;
GUARD
IFA IN ItemFromActor,
U IN Updating;
ACTION
```

```
c.ItemObj:=UI;
c.OpReq:=Send;
c.ReqID:=c.GenReqID;
Iid:=c.ReqID;
CARq:=CARq UNION c;
I:=U;

% This transition is fired when the deadline
% for RIMO implementation is missed: it
% simply change the status of the RIMO to
% 'TimedOut'. This allows subsequent analysis
% and retrieving of late RIMOs.
EVENT NowTimedOut(U,RIMO; UOut,CARq)
LOCAL c: CARequestsType;
GUARD
U IN Updating,
RIMO IN RIMO
ASAP AFTER RIMO.Deadline;
ACTION
c.ItemRef:=RIMO.ID;
c.OpReq:=ChangeState;
c.Status:=TimedOut;
CARq:=CARq UNION c;
UOut:=U;

% Gets the item back from Configuration
% Management and outputs it to the outer
% activity.
EVENT SentToCM(CARs,Iid; IWU)
GUARD
CARs IN CAResults,
Iid IN ItemReqID
SUCH THAT
CARs.ReqID=Iid;
ACTION
IWU:=CARs.ItemObj;
```



Places & Types

Item, ToUpdateItem, ItemUpdated, ItemReady have type ItemObjType
CAResults has type CAResultsType
CARequests has type CARequestsType
RIMOReqID, ItemReqID have type ReqIDType
RIMO, ARIMO, InProcessRIMO have type RIMOFormType

Figura A.30: **UpdateDocStates Activity.** The implementation.

The activity **AnomUpdateDocStates** updates states of item to be updated and related RIMO. In fact, when a RIMO is actually implemented by Project Function, it must be in the 'InProcess' state. Moreover, the item must be either in 'DraftModified' or 'ApprovedModified' state. After these state transitions have been requested to and performed by Configuration Management, the activity ends transferring the updated objects out of activity body. All this is formally specified as follows:

```
% When the implementation phase start,
% the RIMO is always in 'InProcess' state.
EVENT UpdateRIMOState(ARIMO,TUI; CARq,Rid,IR)
LOCAL c: CARequestsType;
GUARD
RIMO IN RIMO,
TUI IN ToUpdateItem;
ACTION
c.ItemRef:=ARIMO.ID;
c.OpReq:=ChangeState;
c.Status:=InProcess;
c.ReqID:=c.GenReqID;
Rid:=c.ReqID;
CARq:=CARq UNION c;
IR:=TUI;

% Gets the updated RIMO from
% Configuration Management.
EVENT GetInProcessRIMO(Rid,CARs; RIMO)
GUARD
CARs IN CAResults,
Rid IN RIMOReqID
SUCH THAT
Rid=CARs.ReqID;
ACTION
RIMO:=CARs.ItemObj;

% If the item associated with the RIMO
% has the 'Draft' state, it must take the
```

```
% 'DraftModified' one.
EVENT IsDraft(IR; CARq,Iid)
LOCAL c: CARequestsType;
GUARD
IR IN ItemReady
SUCH THAT
IR.Status=Draft;
ACTION
c.ItemRef:=IR.ID;
c.OpReq:=ChangeStatus;
c.Status:=DraftModified
c.ReqID:=c.GenReqID;
CARq:=CARq UNION c;
Iid:=c.ReqID;

% If the item associated with the RIMO
% has either the  'Approved' or 'Tested' state,
% it must take the 'ApprovedModified' one.
EVENT IsApprovedOrTested(IR; CARq,Iid)
LOCAL c: CARequestsType;
GUARD
IR IN ItemReady
SUCH THAT
IR.Status=Approved OR IR.Status=Tested;
ACTION
c.ItemRef:=IR.ID;
c.OpReq:=ChangeStatus;
c.Status:=ApprovedModified
c.ReqID:=c.GenReqID;
CARq:=CARq UNION c;
Iid:=c.ReqID;

% Takes the item with new state from
% Configuration Management.
EVENT GetItem(Iid,CARs; IU)
GUARD
CARs IN CAResults,
```

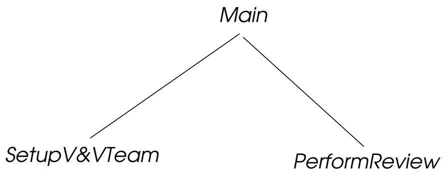


Figura A.31: The activity static tree relative to software review management.

```
Iid IN ItemReqID
SUCH THAT
Iid=CARs.ReqID;
ACTION
IU:=CARs.ItemObj;

% Transfers objects with modified
% state out of activity body.
EVENT Transfer(RIMO,IU; IPRIMO,I)
GUARD
RIMO IN RIMO,
IU IN ItemUpdated;
ACTION
IPRIMO:=RIMO;
I:=IU;
```

A.6 Software Review Management

Software review is an activity that is performed in a parallel to the normal life-cycle of the product. It is of major interest in ensuring the quality of final software product. Italtel BUTD makes an extensive use of review procedures (as already described elsewhere), which are applied not only to objects directly involved in software production as components of the final product, but also to procedures and standards used to develop it (Audits). As an example, the Quality Manual itself is subject to an annual review performed by Quality Function.

The static tree of ManageSoftwareReview is shown in Fig. A.31: it must be pointed out that we did not model review management in all its details, since this, although well documented and formalized, was too extended and

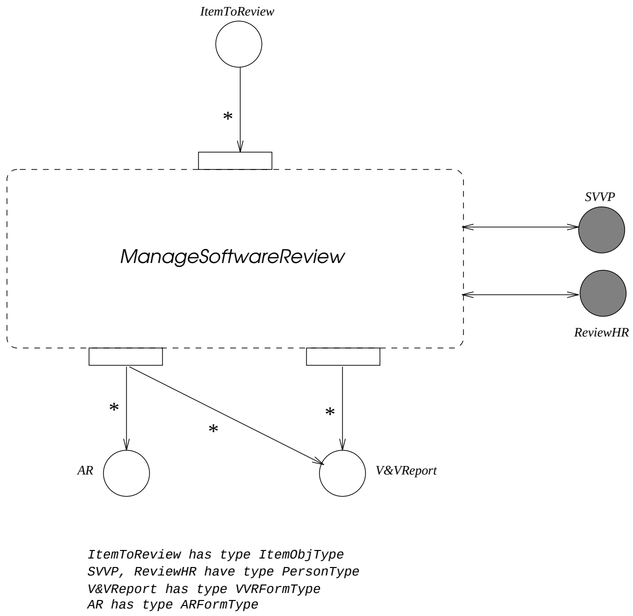


Figura A.32: **ManageSoftwareReview Activity.** The high-level view of the activity describing software review.

low significant with respect to other activities involved in anomaly management. Maybe it will modeled in further works at Italtel or CEFRIEL.

We take the real review work performed by the Review Team as a "black-box". Our modelization uses the modularization constructs provided by SLANG to abstract from details and focus on interfaces and general structure of activity: so unspecified details are collected in the activity PerformReview, though its interface with other activities is well specified.

Description of SLANG activities

ManageSoftwareReview, shown in Fig. A.32 is not a proper activity as it was the case for ManageSoftwareAnomalies: by converse, it is what we called a *module*. It has a single input place, collecting items that are waiting for a review. The activity must always end with a Verification & Validation Report issuing, but it may also issue Anomaly Reports, in the case of negative review. These Anomaly Reports will be sent to Configuration Management, consequently starting the anomaly solving process. Human Resources involved are SVVP and people which is authorized to participate in Review Teams.

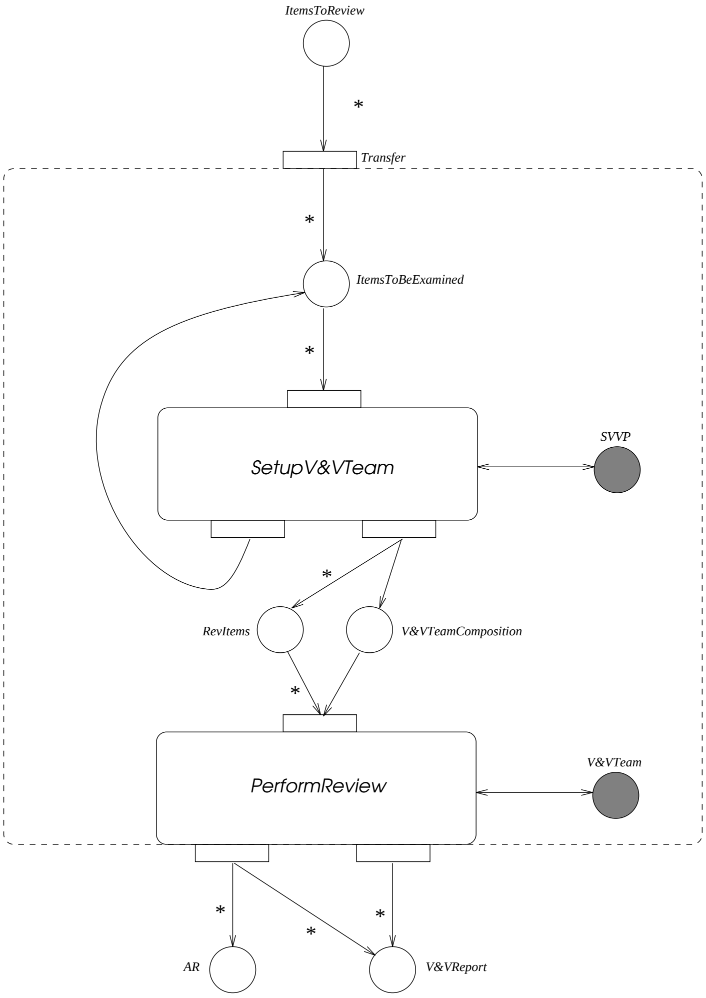
These two groups of people involved in review perform different tasks, as evidenced by the implementation of ManageSoftwareReview, shown in Fig. A.33; SVVP examines items submitted for review and determines if the Review Team must be summoned or not. Depending on this decision the SetupV&VTeam activity can end either providing the set of items and team composition to the activity that actually performs the review task, or refill the input place, waiting for more items.

The formal specification is reduced to a single transition:

```
% Transfers tokens into activity body.
EVENT Transfer(ITR; ITBE)
GUARD
ITR IN ItemToReview;
ACTION
ITBE:=ITBE UNION ITR;
```

Activity **SetupV&VTeam** (Fig. A.34) is very similar to the SetupCCB found in anomaly management, so we will not explain further it: we only provide formal specification:

```
% Transfers input objects into activity body.
EVENT GatherInfo(ITBE; I)
GUARD
ITBE IN ItemToBeExamined;
ACTION
I:=ITR;

% Starts waiting for SVVP decisions.
EVENT WaitForSVVP(SVVP,I; A)
GUARD
SVVP IN SVVP,
I IN Item;
ACTION
A:=SVVP;

% Notifies SVVP that documentation is ready.
```




```
ItemToReview, ItemToBeExamined, RevItems have type ItemObjType
SVVP, V&VTeam have type PersonType
V&VReport has type VVRFormType
AR has type ARFormType
V&VTeamComposition has type PersonRefType
```

Figura A.33: **ManageSoftwareReview.** The implementation.

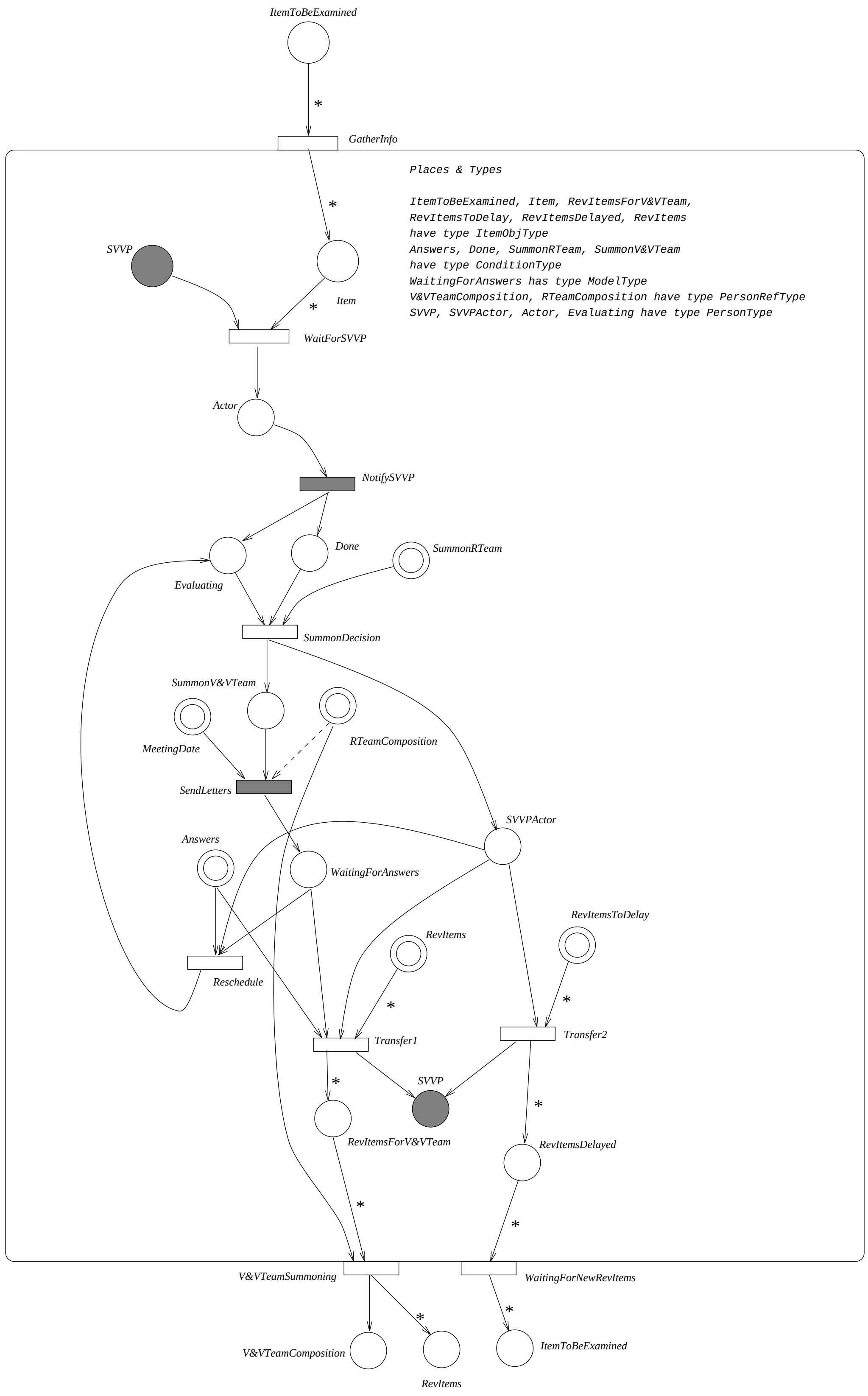


Figura A.34: **SetupV&VTeam** The implementation.

```
EVENT NotifySVVP(A; D)
GUARD
A IN Actor;
ACTION
% Tool invocation.
NotifyTool.Invoke(A,'Documentation for VV ready');
D:=NotifyTool.ToolOK;
E:=A;

% SVVP has determined if ReviewTeam
% must be summoned or not.
EVENT SummonDecision(E,D,S; SOut,SA)
GUARD
E IN Evaluating,
D IN Done,
S IN SummonRTeam;
ACTION
SOut:=S;
SA:=E;

% Sends convocation letters to Review Team members.
EVENT SendLetters(MD,S,RTC; WFA)
GUARD
MD IN MeetingDate,
S IN SummonVVTeam,
RTC IN RTeamComposition
SUCH THAT
S=OK;
ACTION
NotifyTool.Invoke(RTC,'Review Meeting',MD);
WFA:=NotifyTool.ToolOK;

% Not all the members are available for
% the meeting. Dates must be rescheduled.
EVENT Reschedule(A,WFA,SA; E)
GUARD
A IN Answers,
```

```
W IN WaitingForAnswers
SUCH THAT
A=NotOK;
ACTION
E:=SA;

% V&VTeam must be summoned.
EVENT Transfer1(A,W,RI,S; RIFVT,SVVP)
GUARD
A IN Answers,
W IN WaitingForAnswers,
RI IN RevItems,
S IN SVVPActor
SUCH THAT
A=OK;
ACTION
RIFVT:=RI;
SVVP:=S;

% V&VTeam must not be summoned.
EVENT Transfer2(S,RITD; SVVP,RITDOut)
GUARD
RITD IN RevItemsToDelay;
ACTION
SVVP:=S;
RITDOut:=RITD;

% Docs transfer for V&VTeam.
EVENT V&VTeamSummoning(RIFVT,RTC; VVTC,RI)
GUARD
RIFVT IN RevItemsForV&VTeam,
RTC IN RTeamComposition;
ACTION
VVTC:=RTC;
RI:=RI UNION RIFVT;

% Loops waiting new ARs.
```

```
EVENT WaitingForNewARs(RID; ITBE)
GUARD
RID IN RevItemsDelayed;
ACTION
ITBE:=ITBE UNION RID;
```

As we said before, **PerformReview** is not implemented yet: it takes a set of items to be reviewed and a list of person that have to participate the meeting and returns a VVR and possibly an Anomaly Report, which are directly output.

A.7 Human Resource Allocation Management

Human Resources Management is a major issue in software process management, and we included it in this last version of our specification.

The view we developed onto human resource management is quite limited: we considered only the problem of human resource allocation, and only in the special although frequent case of anomaly management.

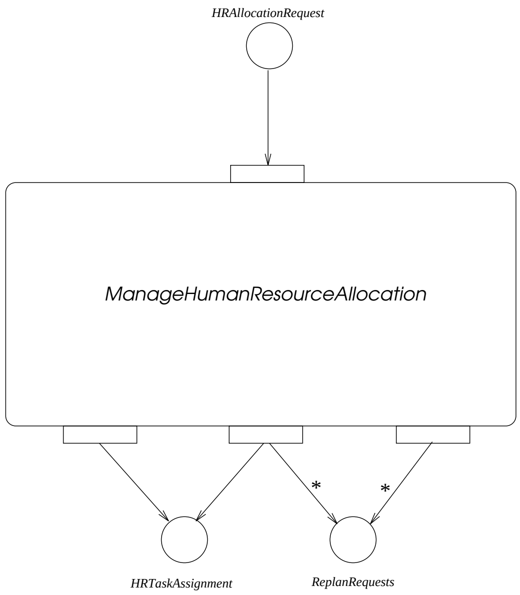
Every process actor has a task queue, possibly with priorities associated. A task is defined (for scheduling) by a duration, a deadline and possibly a priority. Priority is used only to solve (the most frequent) cases when constraints imposed by durations and deadlines do not fit themselves.

Description of SLANG activities

The activity implementing human resource allocation is **ManageHuman-Resources**, shown in Fig. A.35.

This activity takes an allocation request from its input place: the allocation request is a token whose complex type is `TaskAssignmentType`, carrying a task description and an actor suggested for performing the task. This is what is actually done in current Italtel BUTD process, and described in specification; when the CCB issues RIMOs to solve an AR, together with task description there is also the name of the actor that will implement the necessary updates.

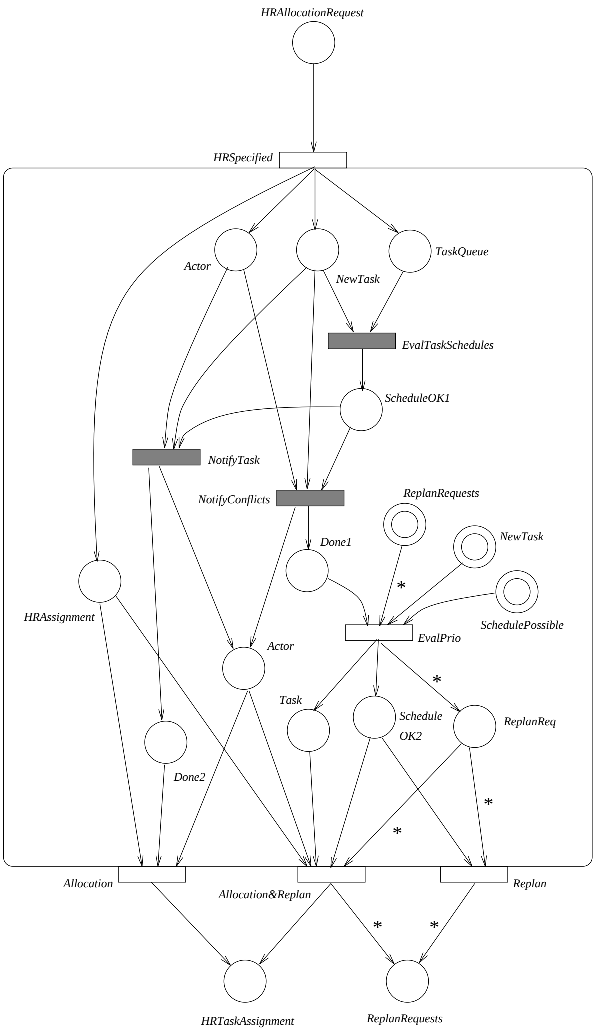
The activity outputs the real task assignment, that can differ from the original in deadline or duration, or priority, and possibly some replan request to be delivered typically to SVVP.



Places & Types

HRAllocationRequest, HRTaskAssignment
have type TaskAssignmentType
ReplanRequest has type SVVPRequestType
Project'sHumanResources has type PersonType

Figura A.35: **ManageHumanResources Activity.** The high-level view of the activity describing human resource allocation mechanism.




```
Places & Types

HRAllocationRequests, HRAssignment, HRTaskAssignment have type TaskAssignment
Actor has type PersonRefType;
NewTask, Task have type TaskType;
Done1, Done2, ScheduleOK1, ScheduleOK2, SchedulePossible have type ConditionType;
ReplanRequests, ReplanReq have type SVVPRequestType;
```

Figura A.36: **ManageHumanResources Activity.** The implementation.

Fig. A.36 shows the "explosion" of previous description. The complex input token is decomposed into the designed process actor, the new task, the task queue and the desired task assignment.

First of all the task attributes are evaluated in conjunction with those of other tasks present in the task queue, using an "ad hoc" tool. Depending on the evaluation results, different notification are sent to the process actor:

- If the task is not in conflict with tasks already allocated, the suggested task assignment is taken as effective and output.

- If the task raises some conflict, notification is given to the actor, who tries to solve conflicts by assigning priorities, in agreement with a superior (typically the Project Manager).

If priorities must be determined, two cases are possible:

- The schedule is possible. Some task of less importance will be delayed, so there is a need for replanning such tasks, but the original assignment of new task can be accepted and immediately made operative.

- The schedule is not possible, not even with priorities. A request for replanning the new task is issued, and no assignment is output, since there is no way to fit a schedule for the specified task (e.g. because it has a low priority).

All these considerations are so formalized:

```
% Starts the HR managing by decomposing the
% request into its components.
EVENT HRSpecified(HRAR; HRA,A,NT,TQ)
GUARD
HRAR IN HRAllocationRequest;
ACTION
HRA:=HRAR;
A:=HRAR.Actor;
NT:=HRAR.Task;
TQ:=HRAR.Actor.TQ;

% Invokes an automated tool to evaluate the
```

```
% new task in relation to other tasks already
% assigned to the specified actor.
EVENT EvalTaskSchedules(NT,TQ; S)
GUARD
NT IN NewTask,
TQ IN TaskQueue;
ACTION
% Tool invocation.
EvalScheduleTool.Invoke(NT,TQ);
S:=EvalScheduleTool.ToolResult;

% Notifies the actor of the new task assignment.
% This is performed only if an appropriate schedule
% can be found for the new task.
EVENT NotifyTask(A,NT,S; D,AOut)
GUARD
A IN Actor,
NT IN NewTask,
S IN ScheduleOK1
SUCH THAT
S=OK;
ACTION
NotifyTool.Invoke(A,'New Task Assigned',NT);
D:=NotifyTool.ToolOK;
AOut:=A;

% Notifies the actor that he/she is requested to
% perform a task that has schedules conflicting with
% other schedules already assigned. A priority assignment
% is needed, to be determined in conjunction with the
% project chief.
EVENT NotifyConflicts(A,NT,S; D,AOut)
GUARD
A IN Actor,
NT IN NewTask,
S IN ScheduleOK1
SUCH THAT
```

```
S=NotOK;
ACTION
NotifyTool.Invoke(A,'New Task has conflicts',NT);
D:=NotifyTool.ToolOK;
AOut:=A;

% This event is fired when it is already defined a
% new priority for the task (if possible) and replan
% requests for tasks with lower priorities.
EVENT EvalPrio(D1,RR,NT,S; T,SOut,RROut)
GUARD
D1 IN Done1,
RR IN ReplanRequests,
NT IN NewTasks,
S IN SchedulePossible;
ACTION
T:=NT;
RROut:=RR;
SOut:=S;

% The new task is allocated succesfully with no
% schedule overhead.
EVENT Allocation(HRA,D2; HRTA)
LOCAL h: TaskAssignment;
GUARD
HRA IN HRAssignment,
D2 IN Done2;
ACTION
h:=HRA;
h.Actor.TaskQueue:=h.Actor.TaskQueue UNION h.Task;
HRTA:=HRTA UNION h;


% A priority has been defined for the task and
% replanning is needed in order to redefine schedules
% for tasks with lower priorities.
EVENT Allocation&Replan(HRA,T,S,RR; HRTA,RROut)
```

```
LOCAL h: TaskAssignment;
GUARD
HRA IN HRAssignment,
T IN Task,
S IN ScheduleOK2,
RR IN ReplanRequests
SUCH THAT
S=OK;
ACTION
h:=HRA;
h.Actor.TaskQueue:=h.Actor.TaskQueue UNION h.Task;
HRTA:=HRTA UNION h;
RROut:=RROut UNION RR;

% No schedule is feasible for the new task, consequently
% replan requests are issued.
EVENT Replan(S,RR; RROut)
GUARD
S IN ScheduleOK2,
RR IN ReplanRequests
SUCH THAT
S=NotOK;
ACTION
RROut:=RROut UNION RR;
```

Appendice B

La sintassi impiegata per la modellazione

Il linguaggio SLANG, per quanto abbia ormai raggiunto un livello di maturità tale da poter essere utilizzato per la modellazione di un caso complesso come quello affrontato in questo lavoro non è ancora stato dotato di una definizione formale, in termini di sintassi e semantica. Nel seguito, con l'aiuto di grammatiche BNF-estese, si fornisce la sintassi utilizzata nella scrittura delle parti testuali della specifica.

Tale sintassi ricalca quella utilizzata negli articoli citati in bibliografia, ed è stata sviluppata a solo scopo operativo, pertanto potrà presentare differenze rilevanti rispetto alla versione definitiva, che sarà definita da un lavoro di tesi [Lip93].

Guardie & Azioni

$< SLANGTransition > \rightarrow$
$< EventClause >$ $[< LocalDeclClause >]$
$< GuardClause >< ActionClause >$

$< EventClause > \rightarrow$
`EVENT` $< EventName >$ $(< InputPlaceVar > (, < InputPlaceVar >)^*;$
$< OutputPlaceVar > (, < OutputPlaceVar >)^*)$

$< LocalDeclClause > \rightarrow$

$\texttt{LOCAL}(< VarName > : < TypeName > ;)^+$

$< GuardClause > \rightarrow$
$\texttt{GUARD}$
$< VarDecl > (, < VarDecl >)^*$
$[\texttt{SUCH THAT} < Predicate > (, < Predicate >)^*];$

$< ActionClause > \rightarrow \texttt{ACTION} < Statement > (, < Statement > ;)^*$

$< VarDecl > \rightarrow < VarName > \texttt{IN} < PlaceName >$

$< Statement > \rightarrow$
$< Assignment > \mid < Iteration > \mid < MethodInvocation >$

$< Iteration > \rightarrow$
$\texttt{FOR} < IteratorName > \texttt{IN} < SetName >$
$(< Statement > ;)^+$
$\texttt{END}$

I non-terminali *InputPlaceVar,OutputPlaceVar*, *VarName*, *EventName*, *TypeName*, *IteratorName* sono identificatori e si intendono costruiti secondo le consuete regole seguite per i linguaggi di programmazione, mentre *MethodInvocation* è di fatto la chiamata di un metodo associato ad un tipo.

Il non-terminale *Predicate* può essere espanso con un predicato qualsiasi, analogo a quelli incontrati nei linguaggi di programmazione. In particolare, si può utilizzare la funzione `CARD()` che ritorna la cardinalità di un insieme e il costrutto `FORALL`, il quale possiede la seguente sintassi:

$< ForallClause > \rightarrow$
$\texttt{FORALL} < IteratorName > \texttt{IN} < SetName >$
$(, < IteratorName > \texttt{IN} < SetName >)^* < Predicate >$

il quale ritorna `TRUE` se il predicato descritto all'interno del non-terminale *Predicate* è vero per ogni elemento nell'insieme *SetName*.

Il non-terminale *Assignment* descrive la sintassi dell'assegnamento ad una variabile: all'interno di esso è possibile invocare metodi, utilizzare operazioni aritmetiche e quanto in genere disponibile nei linguaggi di programmazione.

In aggiunta, è possibile utilizzare il costrutto `UNION` che realizza l'unione tra due insiemi, senza ripetizioni, secondo la seguente sintassi:

$< UnionClause > \rightarrow < SetName1 >$ `UNION` $< SetName2 >$

Type Declarations

$< TypeDecl > \rightarrow$
$< TypeName >$ `SUBTYPE OF` $< TypeName >$ $(, < TypeName >)^*$
$(< AttributeDecl > ;)^*$
$(< MethodDecl > ;)^*$

$< AttributeDecl > \rightarrow < AttributeName > : < TypeName >$

$< MethodDecl > \rightarrow$
`PROCEDURE` $< MethodName > [((< ParDecl >;)^*)]$ $[: < TypeName >];$

$< ParDecl > \rightarrow < ParName > : < TypeName >$

$< PlaceDecl > \rightarrow < PlaceName > : < TypeName >$

$< TypeName > \rightarrow < TypeString > \mid$ `SET`$[< TypeName >]$

I non-terminali *AttributeName*, *MethodName*, *ParName*, *TypeString* sono identificatori e seguono le comuni regole usate per i linguaggi di programmazione.

Appendice C

Concetti generali delle ER Nets

Le reti Ambienti/Relazioni (*Environment/Relationship Nets*) sono una classe di reti di Petri ad alto livello, simili alle reti Predicati/Transizioni [Gen87] e alle Coloured Nets [Jen87].

Esse sono state introdotte in [GMMP91], come un formalismo che integra dati, controllo, funzionalità ed aspetti temporali. Il linguaggio è stato pensato come un nucleo formale per linguaggi di specifica a livello ancora più alto. Le motivazioni principali di un simile linguaggio erano collocate originariamente all'interno dell'area dei sistemi real-time. Nel seguito viene fornita una breve ed informale introduzione al formalismo.

Le reti ER sono reti di Petri i cui token contengono informazioni e le transizioni sono estese mediante predicati ed azioni. I *predicati* selezionano i token che effettivamente abilitano la transizione, a seconda dell'informazione contenuta in essi. Le *azioni* descrivono la trasformazione che passa dai token in input alla transizioni a quelli in output, in seguito allo scatto della transizione. Le informazioni associate ai token sono descritte in termini di variabili e relativi valori. I predicati e le azioni si riferiscono a tali informazioni mediante i nomi delle variabili e dei posti che contengono i token.

La Fig. C.1 mostra una rete ER: il predicato associato alla transizione t richiede che il valore della variabile x, associata al token nel posto p_1, sia maggiore del valore della variabile y, associata al token nel posto p_2. L'azione associata alla transizione t afferma che il token prodotto nel posto p_3 dallo

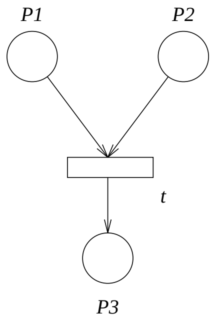


Figura C.1: Un esempio di rete ER, dove il predicato associato alla transizione è $p_1.x > p_2.y$ e l'azione $p_3.y = p_1.x + p_2.y \wedge p_1.x < p_3.z < p_2.y$

scatto della transizione è associato alle variabili y e z. Il valore della variabile y, associata al token prodotto in output, è dato dalla somma dei valori di x, associata al token prelevato da p_1 ed y, associata al token rimosso da p_2. Il valore di z, associata al token in uscita dalla transizione t, è un qualsiasi valore maggiore del valore della variabile x associata al token rimosso da p_1 e minore del valore di y, associata al token prelevato da p_2. Ad esempio, se p_1 contiene due token:

$$\tau_1' : \{x - 0\}$$
$$\tau_1'' : \{x - 7\}$$

e p_2 contiene il token:

$$\tau_2' : \{y - 6\}$$

la transizione t è abilitata dalla coppia $< \tau_1', \tau_2' >$. Infatti, il valore (0) della variabile x associata al token in p_1 è minore del valore (6) della variabile y associata al token in p_2, pertanto il predicato associato a t è soddisfatto. Al contrario, la coppia di token $< \tau_1', \tau_2' >$ non abilita la transizione t. Infatti, il valore (7) associato al token nel posto p_1 è maggiore del valore (6) della variabile y associata al token in p_2, pertanto il predicato non è soddisfatto. Lo scatto della transizione t rimuove i token abilitanti dai posti p_1 e p_2 e produce un nuovo token nel posto p_3 associato a due variabili, y e z. Il valore di y in p_3 è (6), mentre il valore di z in p_3 può essere un qualsiasi valore compreso tra 0 e 6, per esempio 4.

Nessun tipo di estensione è necessaria per modellare gli aspetti temporali: il tempo può essere rappresentato semplicemente aggiungendo all'insieme di variabili associate ai token una variabile *Time* che rappresenti l'istante temporale nel quale i token sono stati creati. Il valore di *tempo* è determinato da predicati ed azioni associati alle transizioni in maniera analoga a quanto avviene per ogni altra variabile associata ai token. Il valore di *Time* associato con i tokens prodotti dallo scatto di una transizione rappresenta l'*istante di scatto* della transizione (assumiamo che tutti i token prodotti da uno scatto siano associati ad un medesimo valore di *Time*). Le azioni possono rappresentare una qualsiasi relazione tra i valori delle variabili associate ai token rimossi dallo scatto (inclusi i valori delle variabili *tempo)* e i valori delle variabili prodotte dallo scatto, fornendo così un modello generale del tempo.

Per esempio, le reti ER possono modellare il caso in cui l'istante di scatto di una transizione rappresentante la trasmissione di un pacchetto in una rete di comunicazione, può dipendere dalla lunghezza del pacchetto. In [GMMP91], si mostra come le ER nets siano più generali dei formalismi di estensione temporale delle reti di Petri proposti in letteratura. Sono proposti inoltre assiomi che vincolano la variabile *Time* in modo da assicurare una rapppresentazione intuitiva del concetto di tempo, il quale è non decrescente rispetto alla sequenza degli scatti. A seconda degli insiemi di assiomi possono essere elaborate due diverse semantiche temporali: una semantica *debole* ed una *forte*. La differenza principale tra i due tipi di semantiche è che nella semantica debole una transizione abilitata *può* scattare all'interno del proprio intervallo temporale, nella maniera specificata dall'azione. La transizione, comunque, non è obbligata ad eseguire lo scatto: se non lo fa all'interno dell'intervallo permesso perde il diritto a scattare. Secondo la semantica forte, invece, una transizione abilitata *deve* scattare all'interno dell'intervallo concesso, a meno che venga disabilitata dallo scatto di un'altra transizione in conflitto.

Elenco delle figure

Elenco delle tabelle

Riferimenti bibliografici

[ABGM92] Pasquale Armenise, Sergio Bandinelli, Carlo Ghezzi, and Angelo Morzenti. Software Process Representation Languages: Survey and Assessment. In *Proceedings of the 4th International Conference on Software Engineering and Knowledge Engineering*, pages 455–462, Capri (Italy), June 1992. IEEE.

[AQA87] AQAP-13. *NATO Software Control System Requirements*. 1987.

[Bal90] R. Balzer. What we do and don't know about software process. CQS-European Observatory on Software Engineering, Milano, October 1990.

[BCF93] Sergio Bandinelli, Stefano Ceri, and Miguel Felder. TANGO: A notation for describing advanced transaction models. Technical report, Politecnico di Milano, 1993.

[BF93] Sergio Bandinelli and Alfonso Fuggetta. Computational Reflection in Software Process Modeling: the SLANG Approach. In *Proceedings of 15th International Conference on Software Engineering–ICSE 15*, Baltimore (MD), May 1993.

[BFG91] Sergio Bandinelli, Alfonso Fuggetta, and Carlo Ghezzi. Software Processes as Real-time Systems: a Case Study Using High-level Petri nets. In Alfonso Fuggetta, Reidar Conradi, and Vincenzo Ambriola, editors, *Proceedings of the First European Workshop on Software Process Modeling*, pages 203–226, Milano (Italy), May 1991. AICA–Italian National Association for Computer Science.

[BFG93a] Sergio Bandinelli, Alfonso Fuggetta, and Carlo Ghezzi. Software Process Model Evolution in the SPADE Environment. Technical

Report, CEFRIEL, Via Emanueli, 15 - 20126 Milano (Italy), May 1993. Submitted to the IEEE TSE Special Section on the Evolution of Software Processes (September 1993).

[BFG93b] Sergio Bandinelli, Alfonso Fuggetta, and Sandro Grigolli. Process Modeling in-the-large with SLANG. In *Proceedings of the 2nd International Conference on the Software Process*, Berlin (Germany), February 1993.

[BFGG92] Sergio Bandinelli, Alfonso Fuggetta, Carlo Ghezzi, and Sergio Grigolli. Process Enactment in SPADE. In *Proceedings of the Second European Workshop on Software Process Technology*, Trondheim (Norway), September 1992. Springer-Verlag.

[BK91] Naser S. Barghouti and Gail E. Kaiser. Scaling up rule-based software development environments. In Axel van Lamsweerde and Alfonso Fuggetta, editors, *Proceedings of ESEC '91–Third European Software Engineering Conference*, volume 550 of *Lecture Notes on Computer Science*, Milano (Italy), October 1991. Springer-Verlag.

[Boe88] Barry W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, pages 61–72, May 1988.

[BR88] Victor R. Basili and H. Dieter Rombach. The TAME Project: Towards Improvement-Oriented Software Environments. *IEEE Transactions on Software Engineering*, 14(6), June 1988.

[BS93] Paolo Battiston and MArco Signori. L'architettura di spade-1: un ambiente di supporto al processo di sviluppo del software. Master's thesis, Politecnico di Milano, July 1993.

[CKO92] Bill Curtis, Marc Kellner, and Jim Over. Process modeling. *Communications of the ACM*, 35(9):75–90, September 1992.

[CR91] Panos K. Chrysanthis and Krithi Ramamritham. ACTA: The SAGA Continues. In A.K. Elmagarmid, editor, *Transaction Models for Advanced Database Applications*. Morgan Kaufmann Publishers, 1991.

[DOD88] DOD-STD-2167a. *Defense System Software Development*. 1988.

[FGM+91] Alfonso Fuggetta, Carlo Ghezzi, Angelo Morzenti, Sandro Morasca, and Mauro Pezzé. *Ingegneria del Software*. Mondadori Informatica, 1991.

[Gen87] H. J. Genrich. Predicate transition nets. In *Advances in Petri Nets 1986 LNCS 254-255*. Springer-Verlag, 1987.

[GJM91] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Englewood Cliffs (NJ), 1991.

[GMMP91] Carlo Ghezzi, Dino Mandrioli, Sandro Morasca, and Mauro Pezzé. A Unified High-level Petri Net Formalism for Time-critical Systems. *IEEE Transactions on Software Engineering*, February 1991.

[HJP+90] H. Hünnekens, G. Junkermann, B. Peuschel, W. Schäfer, and J. Vagts. A Step Towards Knowledge-based Software Process Modeling. In *Proceedings of the First Conference on System Development Environments and Factories*, London (UK), 1990. Pitman Publishing.

[HKG91] Watts S. Humphrey, David H. Kitson, and Julia Gale. A Comparison of U.S. and Japanese Software Process Maturity. Technical report, Software Engineering Institute, 1991.

[Hum89] Watts S. Humphrey. *Managing the Software Process*. SEI Series in Software Engineering. Addison-Wesley, 1989.

[ISO91] ISO 9000. *Quality Management and Quality Assurance standards*. ISO–International Organization for Standardization, first edition, 1991.

[Ita88] Italtel BUTD. *Manuale del Sistema di Qualità Software*, 1988.

[Jen87] F. Jensen. Coloured petri nets. In *Advances in Petri Nets 1986 LNCS 254-255*, pages 254–255. Springer-Verlag, 1987.

[K+90] M. Kellner et al. Ispw-6 software process example. In *Proc. of the 6th. International Software Process Workshop*, Hakodate (Japan), October 1990.

[Kel89] Mark I. Kellner. Software Process Modeling: Value and Experience. In *SEI Technical Review*, pages 23–54, SEI, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213 (USA), 1989.

[KFP88] Gail E. Kaiser, Peter H. Feiler, and Steven S. Popovich. Intellingent assistance for software development and maintenance. *IEEE Software*, pages 40–49, May 1988.

[KH89a] Marc I. Kellner and Gregory A. Hansen. Software process modeling: A case study. In Bruce D. Shriver, editor, *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences, Vol. II - Software Track*. IEEE, January 1989.

[KH89b] Marc I. Kellner and Watts S. Humphrey. Software process modeling: Principles of entity process models. In *Proceedings of the 11th International Conference on Software Engineering*, pages 331–342. IEEE, May 1989.

[KR90] Marc I. Kellner and H. Dieter Rombach. Comparison of software process description. In *Proceedings of the 6th International Software Process Workshop: Support for the Software Process*, Hakodate (Japan), October 1990. ACM.

[Lip93] Patrizia Lippi. Definizione del linguaggio slang per la specifica di processi software. Master's thesis, Politecnico di Milano, July 1993.

[Mad92] Nazim H. Madhavji. Environment evolution: The prism model of changes. *IEEE Transactions on Software Engineering*, 18(5), May 1992.

[MS90] Peiwei Mi and Walt Scacchi. A Knowledge-Based environment for modeling and simulating software engineering processes. *IEEE Transactions on Knowledge and Data Engineering*, 2(3), September 1990.

[Ost87] Leon Osterweil. Software processes are software too. In *Proceedings of the Ninth International Conference on Software Engineering*. IEEE, 1987.

[PCC+91] Mark C. Paulk, Bill Curtis, Mary Beth Chrissis, et al. Capability Maturity Model for Software. Technical Report CMU/SEI-91-TR-24, Software Engineering Institute, SEI, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213 (USA), August 1991.

[Roy70] W. W. Royce. Managing the development of large software systems: concept and techniques. In *Proceedings of WesCon*, August 1970.

[TSY+88] R.N. Taylor, R.W. Selby, M. Young, F.C. Belz, L.A. Clark, J.C. Wileden, L. Osterweil, and A.L. Wolf. Foundations of the Arcadia environment architecture. In *Proceedings of the Third ACM SIGSOFT/SIGPLAN Symphosium on Software Development Environments*, pages 1–13. ACM, 1988.

[WR91] Helmut Wächter and Andreas Reuter. The ConTract Model. In A.K. Elmagarmid, editor, *Transaction Models for Advanced Database Applications*. Morgan Kaufmann Publishers, 1991.