```
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. 4.24. This relatively simple activity is quite interesting

```
% 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
I:=AR;
I.Status:=Originated;
ARep:=I;

% The AR is an old AR with a state change.
% The desired state becomes effective.
EVENT NewAR(AR; I)
GUARD
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
```

```
% The item is a TLR, so an immediate review
% is needed, and invoked.
EVENT TLR(IA; I,ITR)
GUARD
IA IN ItemAccepted
SUCH THAT
IA.Kind=TLR;
ACTION
I:=IA;
I.Status:=ApprovedReview;
ITR:=I;

% The item sent is an Anomaly Report.
EVENT AR(IA; AR)
GUARD
IA IN ItemAccepted
SUCH THAT
IA.Kind=AR;
ACTION
AR:=I;

% The item sent is a RIMO (Modification Request form).
EVENT RIMO(IA; RIMO)
GUARD
IA IN ItemAccepted
SUCH THAT
IA.Kind=RIMO;
ACTION
RIMO:=I;

% The AR is sent for the first time: it takes
% the 'Originated' state.
EVENT AR1st(AR; I,ARep)
GUARD
AR IN ARObj
SUCH THAT
AR.Status=None;
```

```
EVENT NoVVNeeded(IA; I)
GUARD
IA IN ItemAccepted
SUCH THAT
IA.Kind=DocNotVV;
ACTION
I:=IA;
I.Status:=Approved;

% The item is subject to review and must take
% the new state 'Draft'. Review is invoked.
EVENT VVNeededToDraft(IA; I,ITR)
GUARD
IA IN ItemAccepted
SUCH THAT
(IA.Kind=DocVV) AND
((IA.Status=Working) OR (IA.Status=DraftModified));
ACTION
I:=IA;
I.Status:=Draft;
ITR:=I;

% The item is subject to review and must take
% the new state 'ApprovedModified', because it was
% previously in either 'Approved' or 'Tested'.
% Review is invoked.
EVENT VVNeededToAppMod(IA; I,ITR)
GUARD
IA IN ItemAccepted
SUCH THAT
(IA.Kind=DocVV) AND
((IA.Status=Approved) OR (IA.Status=Tested));
ACTION
I:=IA;
I.Status:=ApprovedModified;
ITR:=I;
```

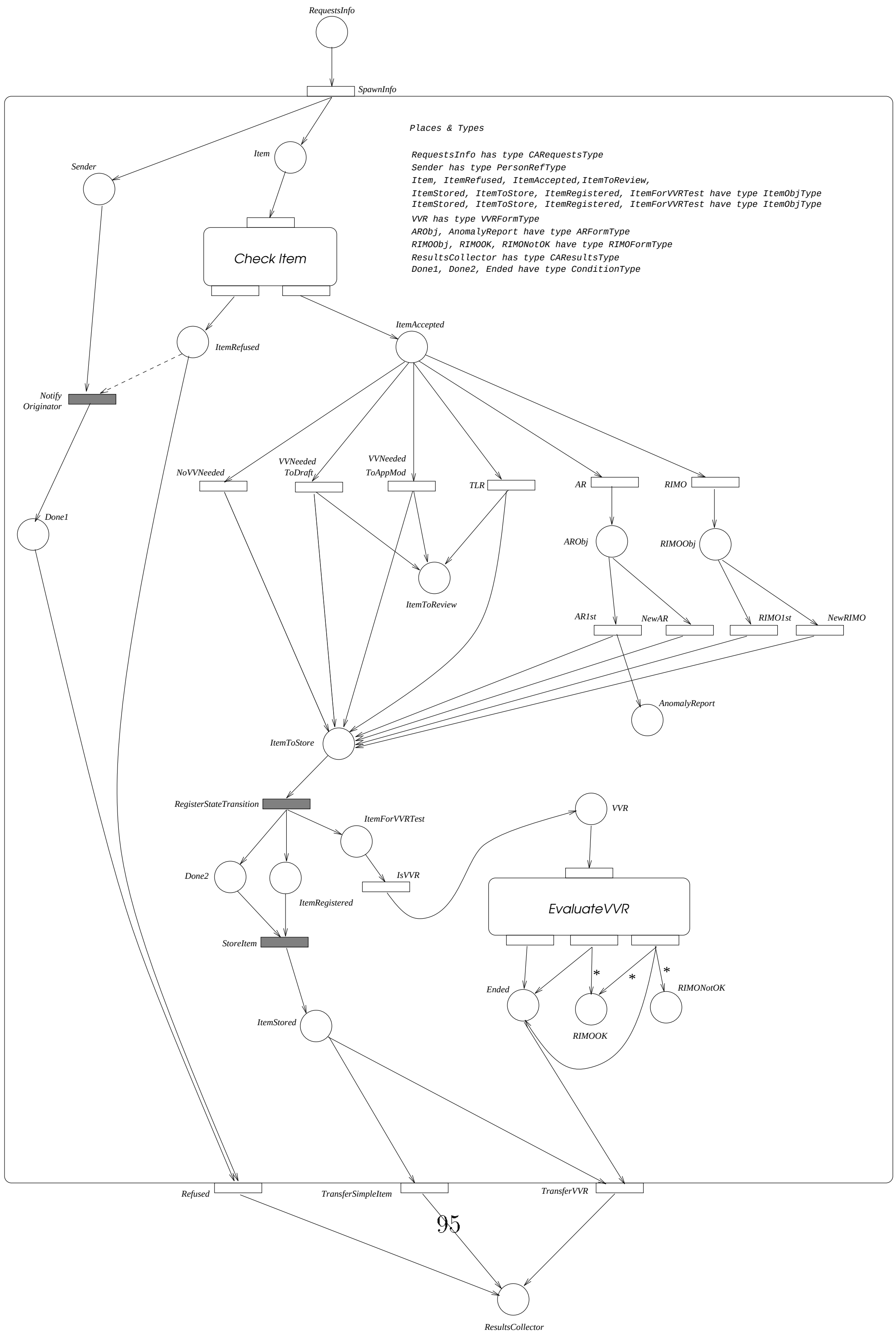


Figure 4.23: **SendItemOp Activity.** The implementation.

  - Subsequent sendings of the form. The AR is resent by CCB at the end of the meeting, so the AR form carries the indication for the new state to be assigned to the related anomaly. This update can become effective only after the form is formally checked, i.e. just at this point of the net.

- `RIMO`. The same distinction between first and further sendings made for ARs is made also for RIMOs. The only difference is that no action is explicitly initiated.

The item so "filtered" is then stored into Configuration Archive, after its state transition is registered into Event Journal. Special actions must be performed, possibly in parallel, if the item is a Verification & Validation Report: the evaluation of such an item, to determine eventual transitions for related items, is performed by lower-level activity EvaluateVVR. Formal specification of activity's transitions follows:

```
% Fills inner places with relevant information
% extracted from the request token.
EVENT SpawnInfo(RI; Se,I)
GUARD
RI IN RequestsInfo
ACTION
Se:=RI.Sender;
I:=RI.ItemObj;

% Notifies that the item has been rejected.
EVENT NotifyOriginator(Se,IR; D1)
GUARD
IR IN ItemRefused,
Se IN Sender;
ACTION
% Tool invocation.
NotifyTool.Invoke(Se,IR,'Incorrect Form');
D1:=NotifyTool.ToolOK;

% If the document is not subject to review,
% the new status is always 'Approved'.
```

```
RC.OpOK:=FALSE;
RC.Accepted:=NIL;
```

SendItem Activity, whose net is described in Fig. 4.23 is a central activity in Configuration Management, and one of the most important within the whole specification. The first operation performed when the item is received by Configuration Management is the formal check over the form, which can be either the Accompanying Module associated with a proper software item, like a source code or a design document, or the item itself, as in the case of an Anomaly Report. If the check evidences some error in the form, the item is sent to the originator for corrections and re-sending. Instead, if the item is formally correct, a set of transition works as a filter firing the appropriate event depending on the kind of the sent item:

- `NoVVNeeded`. The item does not require review and it is stored with 'Approved' state.

- `VVNeededToDraft`. This event is fired when the item sent is currently in either the 'Working' or 'DraftModified' state. The item requires review, which is invoked by filling a copy of the item sent into the ReviewManagement's input place ItemToReview and the new state becomes Draft. Note that state update is simply an object's field update. The update will become persistent in the subsequent invocation of the filing tool.

- `VVNeededToAppMod`. In this case the item has either the 'Approved' or 'Tested' state. It takes the new 'ApprovedModified' state and review is invoked with the mechanism described previously.

- `TLR`. When the item is a Test Launch Report a review is soon invoked, because it means that a new software item has been created or updated and qualified. The TLR takes the new 'ApprovedReview' state.

- `AR`. If the item is an AR two cases are evidenced:

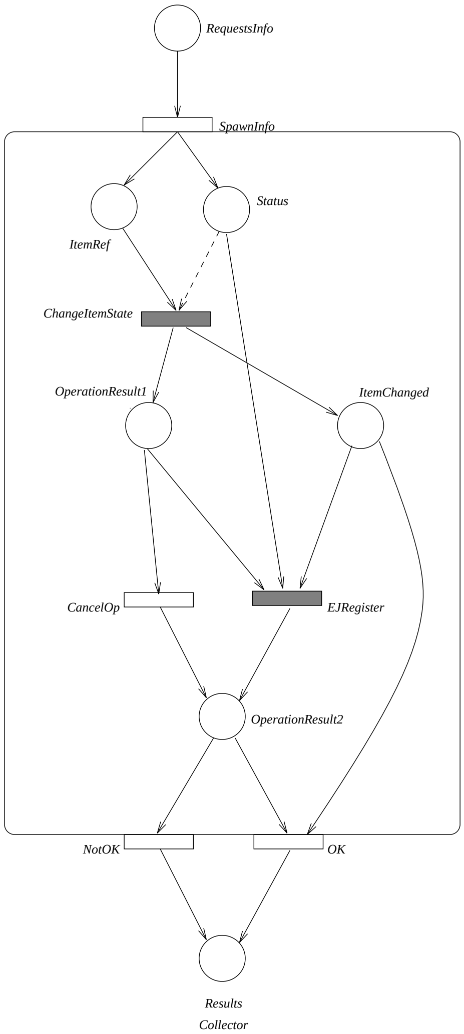
  - First sending of the form. In this case no state is specified on the form, and the AR enters the first state of the associated finite state machine, i.e. 'Originated'. Moreover, the anomaly solving process is initiated by filling a copy of Anomaly Report into ManageSoftwareAnomalies's input place AnomalyReport.

```
S IN State
SUCH THAT
OR1=OK;
ACTION
% Tool invocation.
EJSearchTool.Invoke(IC,S);
OR2:=EJSearchTool.ToolOK;

% Cancels the current operation.
EVENT CancelOp(OR1; OR2)
GUARD
OR1 IN OperationResult1
SUCH THAT
OR1=NotOK;
ACTION
OR2:=OR1;

EVENT OK(OR2,IC; RC)
GUARD
OR2 IN OperationResult2,
SUCH THAT
OR2=OK;
ACTION
% Builds the output token.
RC.EJResults:=IC;
RC.OpOK:=TRUE;
RC.Accepted:=NIL;

EVENT NotOK(OR2; RC)
GUARD
OR2 IN OperationResult2,
SUCH THAT
OR2=NotOK;
ACTION
% Builds the output token.
RC.ItemObj:=NIL;
RC.ItemRef:=NIL;
```




```
Places & Types

RequestsInfo has type CARequestsType
OperationResult1, OperationResult2
have type ConditionType
ResultsCollector has type CAResultsType
ItemRef has type ItemRefType
Status has type ItemStateType
ItemChanged has type ItemObjType
```

Figure 4.22: **ChangeItemStateOp Activity.** The implementation.

ChangeItemStateOp Activity is invoked to change the status associated to a configuration item already stored in configuration archive. The update of the state is composed of two distinct steps, as shown in Fig. 4.22: first the item status is changed by invoking the appropriate tool then, if this first operation succeeded, the state transition is registered into the Event Journal.

```
% Prepares input parameters for subsequent tool invocations.
EVENT SpawnInfo(RI; IR,S)
GUARD
% Fires the transition only if the requested
% operation is a state transition.
RI IN RequestsInfo
SUCH THAT
RI.OpReq=ChangeState;
ACTION
IR:=RI.ItemRef;
S:=RI.Status;

% Changes a configuration item state.
EVENT ChangeItemState(IR,S; OR1,IC)
GUARD
IR IN ItemRef,
S IN State;
ACTION
% Tool invocation.
ChangeStateTool.Invoke(IR,S);
OR1:=ChangeStateTool.ToolOK;
IC:=ChangeStateTool.ToolResult;

% Registers the event in the Event Journal.
EVENT EJRegister(OR1,IC,S; OR2)
GUARD
% Fires the transition only if the previous
% operation succeeded.
OR1 IN OperationResult1,
IC IN ItemChanged,
```

```
% Fires the transition only if the requested
% operation is a retrieving.
RIn IN RequestsInfo
SUCH THAT
RIn.OpReq:=Retrieve;
ACTION
% Tool invocation.
RetrieveTool.Invoke(RIn.ItemRef);
OR:=RetrieveTool.ToolOK;
RIt:=RetrieveTool.ToolResult;

% Successful operation.
EVENT OK(OR,RI; RC)
GUARD
OR IN OperationResult,
RI IN RequestedItem
SUCH THAT
OR=OK;
ACTION
% Builds the output token.
RC.ItemObj:=RI;
RC.ItemRef:=RI.ID;
RC.OpOK:=TRUE;
RC.Accepted:=NIL;

% Unsuccessful operation.
EVENT NotOK(OR,RI; 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;
```

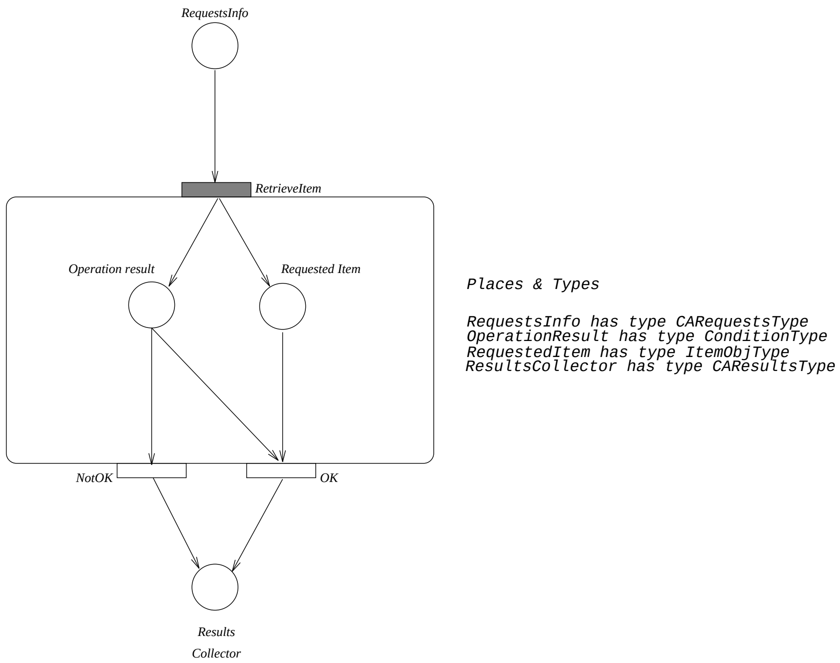


Figure 4.21: **RetrieveItemOp Activity.** The implementation.

```
OR:=CopyTool.ToolOK;
PC:=CopyTool.ToolResult;

% Successful operation.
EVENT OK(OR,PC; RC)
GUARD
OR IN OperationResult,
PC IN PhysicalCopy
SUCH THAT
OR=OK;
ACTION
% Builds the output token.
RC.ItemObj:=PC;
RC.ItemRef:=PC.ID;
RC.OpOK:=TRUE;
RC.Accepted:=NIL;

% Unsuccessful operation.
EVENT NotOK(OR,PC; 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;
```

RetrieveItemOp Activity is very similar to DuplicateItemOp: its net, shown in Fig. 4.21 is identical to the previous, except for the black transition. It has the same behavior, formally described by the following specification:

```
% Retrieves an item object from Configuration Archive.
EVENT RetrieveItem(RIn; OR,RIt)
GUARD
```

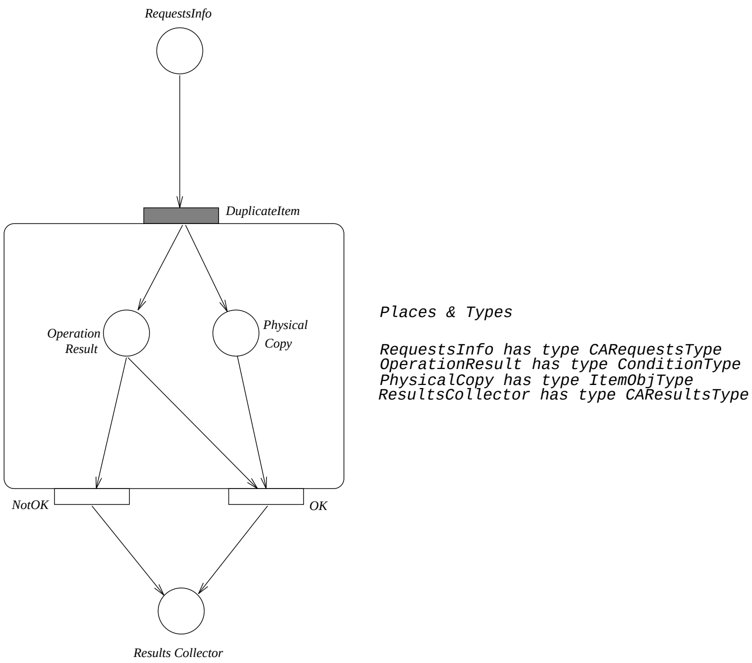


Figure 4.20: **DuplicateItemOp Activity.** The implementation.

```
ACTION
RI:=CARq;
Rid:=CARq.ReqID;

% Transfers the output token to the output place
% keeping track of request identifier.
EVENT Collect(Rid,RC; CARs)
LOCAL c: CAResultsType;
GUARD
Rid IN ReqID,
RC IN ResultsCollector;
ACTION
c:=RC;
c.ReqID:=Rid;
CARs:=CARs UNION c;
```

DuplicateItemOp Activity, expanded in Fig. 4.20, performs the duplication of a configuration item from Configuration Archive to a given location, when the token contained in RequestsInfo satisfy the condition `OpReq=Duplicate`. The black transition invokes the appropriate tool, which can operate successfully or not: place PhysicalCopy will contain the object output by tool invocation, while OperationResult will contain a value telling if the operation has been successful or not. Transitions OK and NotOK interpret this results and build the appropriate output token to be put in ResultCollector. Formally:

```
% Invokes the tool to perform the copy.
EVENT DuplicateItem(RI; OR,PC)
GUARD
% Fires the transition only if the requested
% operation is a duplication.
RI IN RequestsInfo
SUCH THAT
RI.OpReq=Duplicate;
ACTION
% Tool invocation.
CopyTool.Invoke(RI.ItemRef,RI.Destination);
```

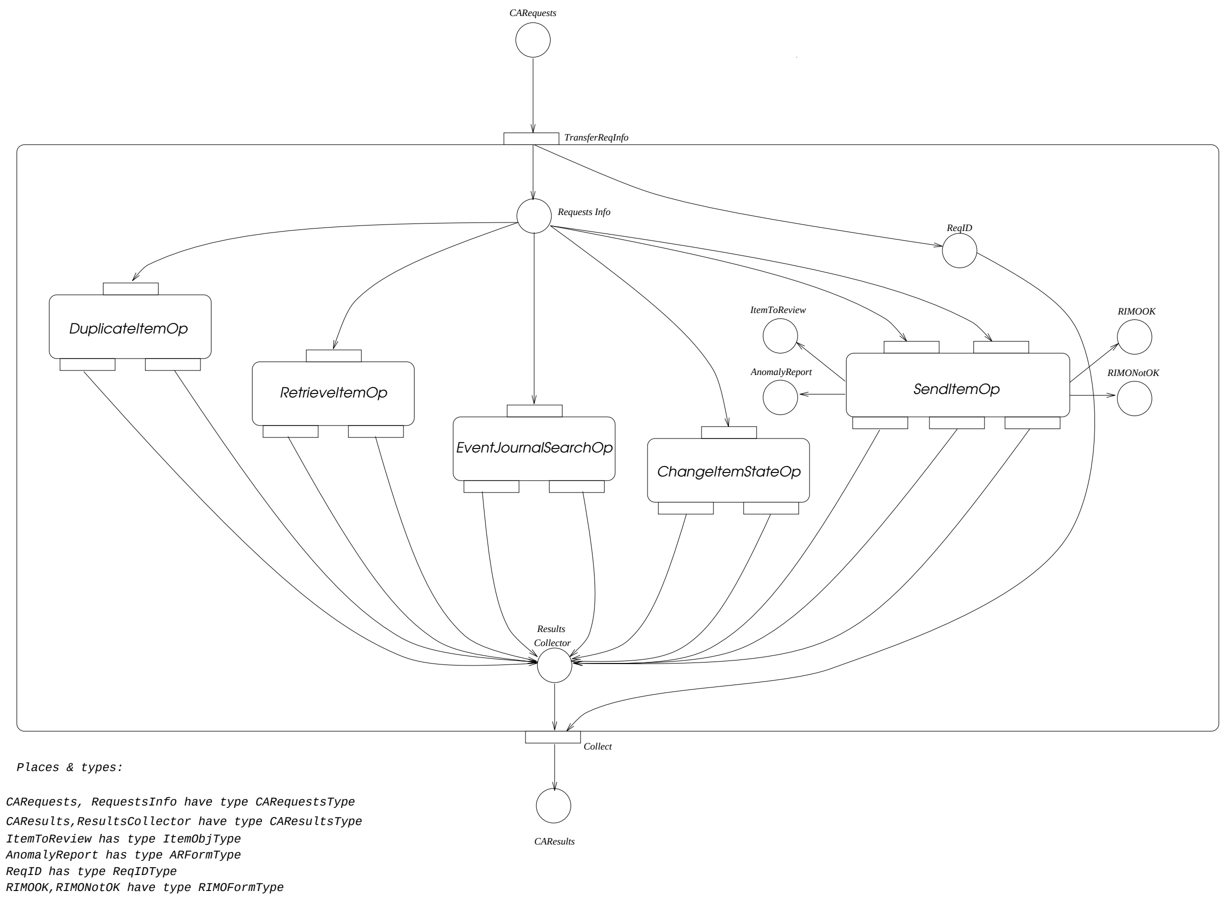


Figure 4.19: **ManageConfigurationArchive Activity.** The implementation of Software Configuration Management higher-level activity.

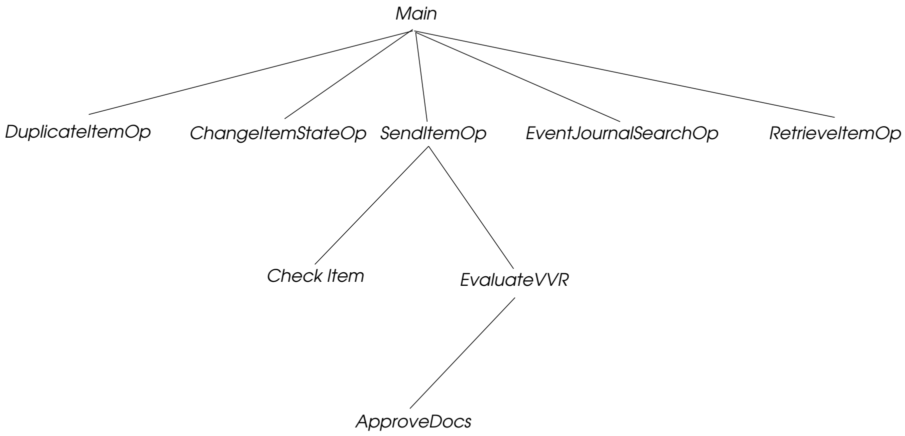


Figure 4.18: The activity static tree relative to software configuration management.

ManageConfigurationArchive activity hides under its activity box five activities, performing the real work: only one of these is enacted depending on the information contained in the token within RequestsInfo, transferred from CARequests. After the computation performed by one of the activities is ended, the output token, properly constructed following the rules described in Section 4.6, is placed in ResultsCollector and then transferred in the output place (shared) CAResults. The activity keeps track of request identifier using the place ReqID, filled at the entering of activity and used to construct the output token. Remember that since there is one SLANG istance of this activity for every CARequests token (weight of input arc is 1) the request identifier is unique, i.e. ReqID will contain only a token. Note also that the intermediate places are all managed within the SendItemOp activity: as we will see later, this is a key activity in our specification. The following are transitions specification:

```
% Transfers a token request into the activity body
% and keeps track of request identifier.
EVENT TransferReqInfo(CARq; RI,Rid)
GUARD
CARq IN CARequest;
```

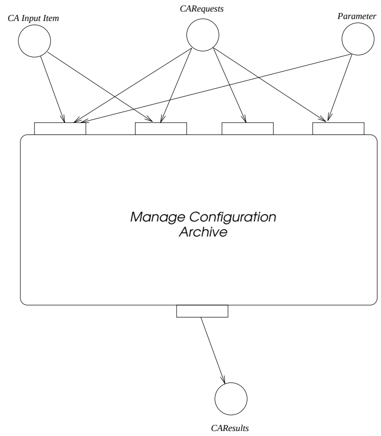


Figure 4.17: An alternative solution for the high-level description of Software Configuration Management.

this case is higher than in previous case.
As one can see, the techniques and method used to decide how to better structure the SLANG modelization are quite similar to the ones commonly used in modular programming languages.

ManageConfigurationArchive interacts with the rest of the net also through intermediate places: ItemToReview is used to start the review process on an item sent to Configuration Management, AnomalyReport starts the anomaly solving process codified in the ManageSoftwareAnomalies activity, while RIMOOK and RIMONotOK are used to keep track of what RIMOs are currently in a solved state, after VVR is sent to Configuration Management.

Description of SLANG activities

The static tree of ManageConfigurationArchive is shown in Fig. 4.18.

The higher-level activity description of Software Configuration Management can be exploded to show inner implementation as done in Fig. 4.19: this is a major advantage of SLANG specifications in terms of visual understanding of activity behavior.

We will first describe net behavior in an informal way, then showing and commenting the formal behavior in term of transitions' guards and actions. We will follow this steps during all nets descriptions, for every activity used during specification.

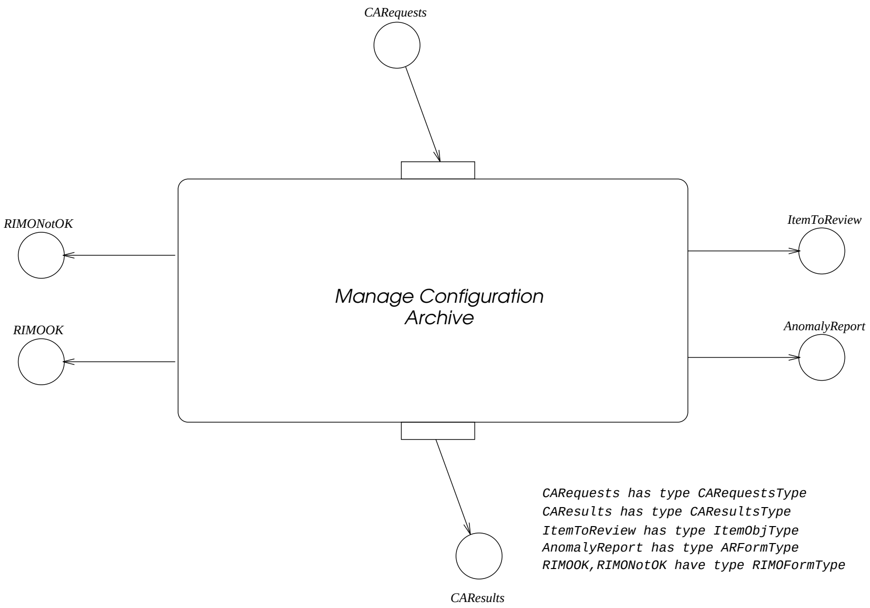


Figure 4.16: High-level view of the main activity for Software Configuration Management.

Interaction with Configuration Management is performed through the only input place CARequests (whose type is `CARequestsType`) and the only output place CAResults (of type `CAResultsType`). The rationale behind this decision is quite intuitive: all data relevant to configuration management are described by the complex data structure of CARequests and elaborated by the activity body. When computation ends, the activity outputs operation results in a complex token which is output in the output place.

Consequently, information about operations to perform, objects involved in the operations and other auxiliary information is not explicited by the net at an high level view, but are hidden into the activity body and types used to interface the activity with the rest of the specification.

This is a design choice, i.e. an alternative idea is the one showed in Fig. 4.17: here input data (operations to perform and data to modify) and output data (returned object and operation conditions) are clearly shown by the net. It is our opinion that such a solution does not allow a proper information hiding when faced with changes, i.e. if for example an operation is added or a new data is required, the amount of modifications needed in

```
EJSearchToolType SUBTYPE OF GeneralToolType
ToolResult=SET[ModelType];
PROCEDURE Invoke(predicate: PredStringType);

% Tool to evaluate schedules: given a task and a list
% of tasks, determines if the new task can be fitted with
% the others.
EvalScheduleToolType SUBTYPE OF GeneralToolType
ToolResult=ConditionType;
PROCEDURE Invoke(task: TaskType; taskQueue: LIST[TaskType]);
```

Tool descriptions are built from the root type `GeneralToolType` by redefinition of both the method `Invoke` and associated fields. For example, `RetrieveTool` is constructed specifying that the object result has type `ItemObjType` and its method signature contains a parameter specifying the reference to the item to extract.

4.8 Software Configuration Management

The first draft of anomaly management modeled interaction with Configuration Management in a distributed fashion, i.e. SGMR human resources where employed directly into anomaly management activities when needed. In other words, there was no Configuration Management activity.

This solution is not satisfactory due to the following reasons:

- The nets describing SGMR operations were simple but not trivial. Since SGMR interaction is a major issue in software anomaly management, this led to replication.

- Configuration Management is an activity, as Anomaly Management is. Moreover is described as an activity in the Quality Manual, too. Thus is more natural to model it as a SLANG activity rather than distribute it throughout the nets.

Following these considerations, we built a specification constructed as shown in the high level view of Fig. 4.16.

```
DuplicateToolType SUBTYPE OF GeneralToolType
ToolResult: ItemObjType;
PROCEDURE Invoke(item: ItemRefType; destination: LocationType);

% Tool to change the state of a Configuration item.
ChangeStateToolType SUBTYPE OF GeneralToolType
ToolResult: ItemObjType;
PROCEDURE Invoke(item: ItemRefType; state: ItemStateType);

% Tool used to store an item under configuration control.
FilingToolType SUBTYPE OF GeneralToolType
ToolResult: ItemRefType;
PROCEDURE Invoke(item: ItemObjType);

% Tool used to notify messages (and possibly objects)
% to persons or groups.
NotifyToolType SUBTYPE OF GeneralToolType
ToolResult=NIL;
PROCEDURE Invoke(notifyTo: SET[PersonRefType]; msg: Text;
items: SET[ItemObjType]);

% Mailing tool.
MailTool SUBTYPE OF GeneralToolType
ToolResult=NIL;
PROCEDURE Invoke(item: ItemObjType);

% Tool used to run given tests over given configuration items.
RunTestsToolType SUBTYPE OF GeneralToolType
ToolResult: TLRFormType;
PROCEDURE Invoke(tests: SET[TestType]; items: ItemObjType);

% Tool to register an event into the Event Journal.
EJRegisterToolType SUBTYPE OF GeneralToolType
ToolResult=NIL;
PROCEDURE Invoke(item: ItemRefType; state: ItemStateType);

% Tool to perform searches on Event Journal, given a predicate.
```

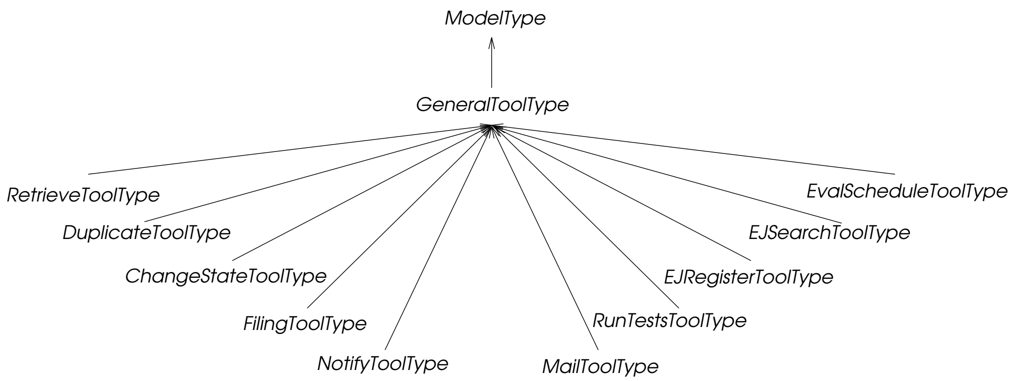


Figure 4.15: Object hierarchy for tools.

- Proper outputs are stored within the `ToolResult` field. If more than an output is provided, more than a field must be given by the modeler.
- Outcomes (error conditions and so on) are stored within the *ToolOK* field.

The observation of these constraints is up to the modeler, or the implementor of the `Invoke` method.

Following these considerations we developed this tool description:

```
% Root of tools' hierarchy.
GeneralToolType SUBTYPE OF ModelType
ToolOK:   ConditionType;
ToolResult:  ModelType;
PROCEDURE Invoke();

% Tool to retrieve item objects from Configuration Archive.
RetrieveToolType SUBTYPE OF GeneralToolType
ToolResult: ItemObjType;
PROCEDURE Invoke(item: ItemRefType);

% Tool to duplicate Configuration Archive objects to
% a given destination.
```

Field	*Duplicate*	*Retrieve*	*Send*	*ChangeState*	*EJSearch*
OpOK	x	x	x	x	x
Accepted			x		
ItemRef	x	x	x	x	
ItemObj	x	x	x	x	
EJResult					x
ReqID	x	x	x	x	x

Table 4.3: How Configuration Management fills CAResults fields after the execution of requested operations.

4.7 Tools description

Tool management is a key issue within process modeling: SLANG offers the black transition special construct to deal with it, but currently there is no agreement on how to model tool objects and their interaction in terms of output results management. Our proposal tries to describe tools in an object-oriented way, like any other object flowing around a SLANG net.

This has surely the immediate benefit to describe all process objects in an homogeneous way, but is highly constrained to implementation issues. This means that this is the way we *now* describe tools and not necessarily the way tools will be described in the final version of SPADE.

In our description we describe tools through the type hierarchy shown in Fig. 4.15. The root of the hierarchy contains the following built-in tool description on which rely every user-defined description:

```
GeneralToolType SUBTYPE OF ModelType
ToolOK: ConditionType;
ToolResult: ModelType;
PROCEDURE Invoke();
```

The `Invoke` method contains the real invocation of the tool. Here we do not describe the code contained within this method, but it must observe the following constraints:

Field	*Duplicate*	*Retrieve*	*Send*	*ChangeState*	*EJSearch*
OpReq	x	x	x	x	x
ItemRef	x	x		x	
ItemObj			x		
SenderRef			x		
DestinationRef	x				
Status				x	
Pred					x
ReqID	x	x	x	x	x

Table 4.2: How to fill CARequests fields to issue Configuration Management requests.

- `TaskType`. The task descriptor, carrying the basic information needed to determine if the task can be scheduled successfully and under which constraints. Moreover, it contains a text description of the task to be executed.

- `TaskAssignmentType`. It is used to interact with the activity that manages the allocation of human resources: it is a simple data structure collecting the actor selected for the task and the task description.

- `PersonType, PersonRefType`. The first of these two types describes the relevant information associated with a software process actor, while the second one, `PersonRefType`, provides a shortcut to quickly reference a person without carrying a complex data structure. By the way, this is what is actually done in current real process at Italtel BUTD.

Together with these types directly involved and used in specification of Italtel BUTD process, we needed some "auxiliary" types to model details of the specification, as dates, identifiers, conditions and so on. They are listed at the end of the previous type declaration list: some of these types are not defined following a precise definition, as it was for types describing forms used during anomaly management: they are defined on an intuitive basis, and so they could be changed in a future version of our specification.

sults when the body execution is ended.

- The guard of transition $t_{checking}$ selects only the tokens satisfying the following condition:

```
ANY ID IN ItemReqID,
ANY CARs IN CAResults
SUCH THAT
ID=CARs.ReqID;
```

  Informally, it selects the tokens in CAResults that have a request identifier equal to the one stored in ItemReqID, associated to the current request.

Now, the meaning of remaining types is more clear: `CAAccessType` collects the basic information needed to interact with Configuration Management, i.e. the request identifier and either a reference to an item (ItemRef) or an object item (ItemObj); `CARequestsType` provides a data structure to specify what operation is requested and parameters needed; by converse, `CAResultsType` provides a structure to be filled with results and outcomes of the Configuration Management access.

Referring to Fig. 4.14, the action of $t_{calling}$ fills input tokens fields with appropriate values, while the action (and possibly the guard) of $t_{checking}$ read the output tokens field values and perform appropriate actions.

Note that not all the input token fields have to be filled by the caller, and not all the output token fields have to be filled by Configuration Management. Non-filled fields get the default value of *NIL*. Table 4.2 explains how fields must be used to request the desired operations:

By converse, Table 4.3 shows how output fields are filled by Configuration Management activity.

In addition to anomaly management and configuration management, we tried to develop a specification of human resource allocation problems. We considered only allocation problems generated during anomaly management, which differ from normal task assignment, because the software process actor that must execute the task is always determined directly by Change Control Board. The details of the allocation mechanism adopted will be described later in this document. In order to cope with problems raised by this problem, a set of types has been delivered:

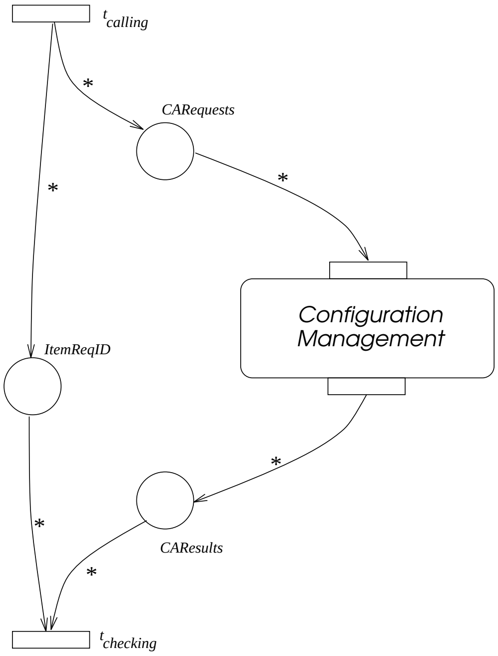


Figure 4.14: An example to explain access policy to Configuration Management used within our specification.

child for activity A, so B's interface places are local within the father A, and there would be no ambiguity. By converse, an ambiguity may exists in the case of multiple threads within the activity A. In the figure, Subnet1 prepares tokens in I1 for the callee B, which performs its elaborations and fills properly I2, enabling the Subnet2. If some kind of "feedback" exists between Subnet1 and Subnet2, like the place T, creating some kind of multiple thread, there is no mean to distinguish among threads.

To bypass the problem, we used the following policy, illustrated in Fig. 4.14 with reference to Configuration Management:

- The action of transition $t_{calling}$ generates an unique request identifier (by means of the `GenReqID` method contained in `CARequestsType`) and output it both into the ItemReqID place (of type `ReqIDType`) and the `ReqID` field of CARequests.

- Software Configuration Management reads the input token and keeps track of the request identifier, filling with it the `ReqID` field of CARe-

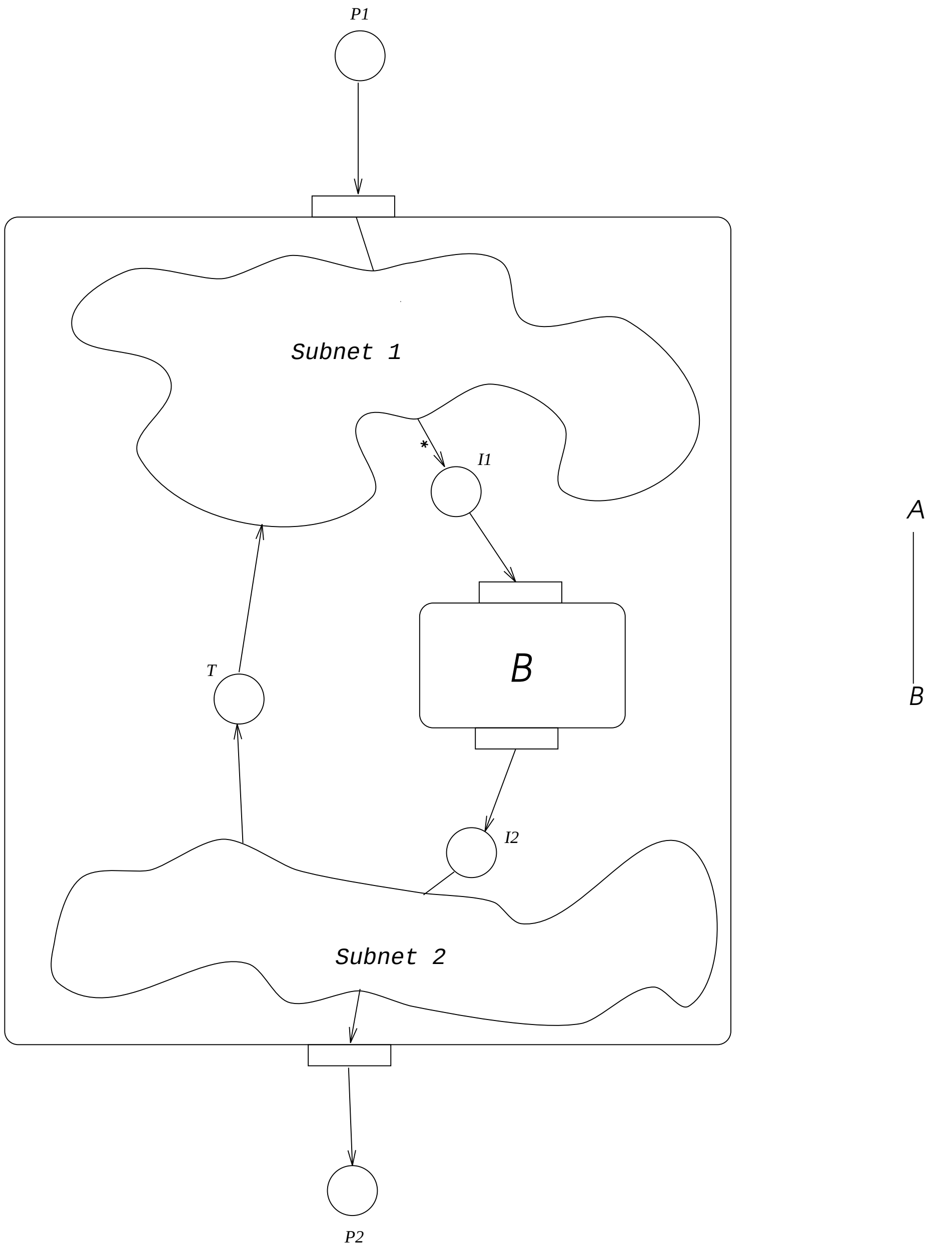


Figure 4.13: Access conflicts between activities at different levels, due to multiple threads in net execution.

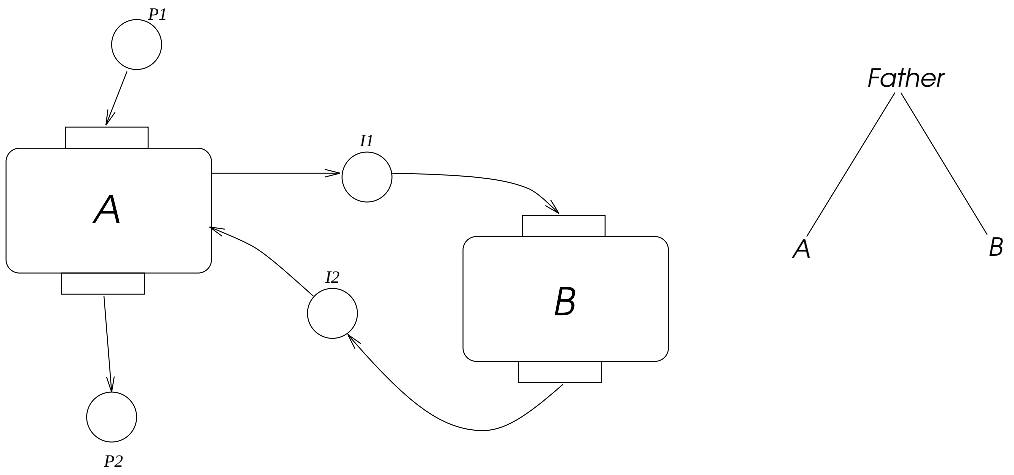


Figure 4.12: Access conflict between activities at the same level in the static tree.

such mechanisms: in our specification we used also a set of types, namely the ones associated to the interface places of ManageConfigurationArchive, to establish a sort of access protocol to this activity.

We identified the set of operations that can be requested to configuration management and described with the enumerative type `CAOpType`.

`ReqIDType` is the type of what we called the *request identifier*. The need for such an identifier is raised by the semantic of the activity construct. In fact, every time an activity is entered, a new instance of the body is generated, that shares interface places with the rest of the specification, according to SLANG scope rules.

Figures 4.12 and 4.13 show typical situations allowed by such rules but causing ambiguities.

In Fig. 4.12 the activity B performs some kind of service that is used by activity A. Since the two activities *are at the same level in the static tree*, the intermediate places of A, alias the interface places of B, alias I1 and I2, are really shared by the two activities. Let us suppose that A issues more than a request to B: for every request a new B active copy is spawned, performing the elaboration on its input tuple and filling the output place I2 with the output. But I2 is shared, in other words, A has no means to distinguish the output generated by a call from the one generated from another call.

In Fig. 4.13 the ambiguity is more subtle: this time the activity B is a

specifying the active copy involved into special places bound to every activity definition. Example of such mechanisms will be encountered when examining the implementation of specification's activities.

In addition to this "service" type, `AnomalyMgmFormType` includes a field explaining the problem for which the form has been issued, i.e. anomaly for ARs and item changes for RIMOs. This description is completely textual, no formality is expected. A little formality level is provided by enumerating types specifying state, kind, class and origination phase for the anomaly:

- `AnomKindType` specifies the allowed kind of anomaly. The kind of an anomaly is an informal mean to specify the "simptoms" associated with the anomaly.

- `AnomClassType` specifies the level of importance of the anomaly: it is an intuitive and informal way to specify planning and constraints that will be described more precisely within RIMO forms.

- `AnomOrgPhaseType` specifies the developing phase during which the anomaly was originated.

An analogous approach is followed for RIMOs:

- `UpdateKindType` specifies the kind of change needed is specified by the enumerative type, that follows the same terminology used in software engineering.

- `RIMOReviewType` specifies the information needed to perform qualification on the updated item. This is needed since a different documentation and testing amount is required depending on the value of this field.

The types associated to AR and RIMO do not deserve further comments: the only thing worthy to be mentioned is the presence of a Requisite field in `RIMOFormType` containing the re-qualification requirements to be submitted to Qualification Subfunction after update completion, and it typically consists of the information about what tests must be re-executed.

The set of types needed to effectively model mechanisms of configuration archive are not limited to the ones necessary to describe objects involved in

- `Distrib`. This field contains the *mailing list* associated to each type of configuration item. The mailing list is a list of references to groups and persons interested in changes (state transition and/or updates) of the considered item, and is used to notify them. The `FillDistribution` method provides a way to dynamically change the mailing list, depending on what transition occurred.

In addition to the `FillDistribution` method, another method, `CheckFormalCorrectness`, is provided to formally check the item sent to configuration management.

These two methods are redefined using the inheritance mechanism, allowing (through polymorphism and late binding) the specification of a single action performing different operations depending on the type of the object the operations are applied to.

`GeneralFormType` is the root for the form hierarchy: here all the fields that are common to every form are collected. `GeneralFormType` has three children:

- `MAFormType`: the accompanying module.

- `VVRFormType`: the verification & validation module.

- `AnomalyMgmFormType`: the root type for the types of forms directly involved in anomaly management, i.e. `ARFormType` for Anomaly Report form and `RIMOFormType` for Modification Request form.

The Accompanying Form (MA) is currently used in Italtel BUTD whenever an item is sent to SGMR and/or retrieved from this. `MAFormType` description does not deserve further comments.

The Verification & Validation Report is used after review activities to keep track of the results and drive further operations depending on review's results. It collects fields to describe both review results and relationships with related items.

`AnomalyMgmFormType` is the root type for anomaly-specific forms.
Among its fields, `Instance` deserves major interest: it contains the active copy of the activity actually driving the anomaly or RIMO solution. Such information is used when it is necessary to suspend or even cancel the solution process for the related object: in fact, these operations are performed by

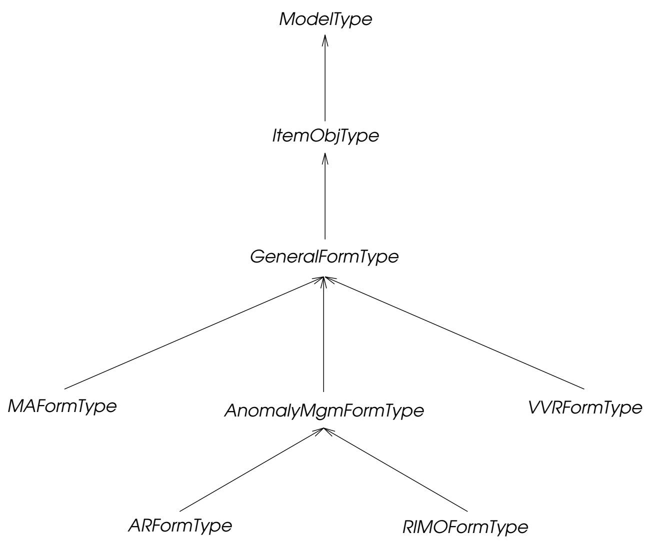


Figure 4.11: The type hierarchy used to describe configuration items and forms.

- `Kind`. This field is used to perform different actions depending on the particular configuration item sent to configuration management. For example, a TLR always starts an associated review task, while a RIMO starts the task of implementing updates.

- `Status`. As described in other documents, the transformations available for a configuration item are ruled by an associated finite state machine, which is different for every kind of configuration item, as defined in the previous field. Hence, the Status field contains the current status for the associated configuration item: note that the consistency of such a state with the item's FSM is automatically ensured through type checking, using the types shown at the beginning of the above list, i.e. ARStatusType, RIMOStatusType, DocNotVVStatusType, DocVVStatusType, TLRStatusType, and SCUStatusType. These type are used to redefine the type of status field into descendants, so that only an allowed state may be assigned to this field.

- `ToBeTested`. This `BOOLEAN` takes the `TRUE` value when the item must be tested, e.g. an SCU.

RAPPORTO DI ANOMALIA (AR)	
IDENTIFICATIVO AR:	
STATO DELL'ANOMALIA:	
EVIDENZIATA Data:	NOTIFICATA Data:
RIFIUTATA Data:	PENDENTE Data:
SOSPESA Data:	RISOLTA Data:
TIPO DI MODULO: RAPPORTO DI ANOMALIA PROPOSTA DI MODIFICA	
RIFERIMENTO TLR/TLVVR:	Data:
RELEASE INTERESSATA	
SOFTWARE:	HARDWARE:
TIPO DI ANOMALIA:	CLASSI DI ANOMALIA:
ANOMALIA DI DOCUMENTAZIONE	CORREZIONE INDISPENSABILE
FUNZIONE MANCANTE	CORREZIONE IMPORTANTE
FUNZIONE NON ATTIVATA	CORREZIONE CONSIGLIATA
FUNZIONE ERRATA	
INTERRUZIONE PROGRAMMA	
ANOMALIA DI NORMATIVA	
TIPO DI MODIFICA	FASE DI RILEVAZIONE:
CORRETTIVO	REVISIONE
EVOLUTIVO	DEBUGGING
PERFETTIVO	QUALIFICAZIONE
ADATTIVO	INTEGRAZIONE(SW/HW)
DI COMPLETAMENTO	ESERCIZIO
	ALTRA
DESCRIZIONE DELLA ANOMALIA:	
SOLUZIONE PROPOSTA:	
RIFERIMENTI/ALLEGATI:	
DISTRIBUZIONE:	
Trattato da:	Ente emittente:

Table 4.1: A fac-simile of Italtel Anomaly Report form, the main form used within Software Anomaly Management.

agement. A fac-simile is shown in Table 4.1.

- Verification & Validation Report (VVR). Issued after review on configuration items.

- Modification Request (RIMO). Issued to cope with changes to configuration items.

- Accompanying Module (MA). Used when an software process actor submits an item to Configuration Management to specify item composition, configuration requirements and other "administrative" information.

In addition to the forms described before, we used a type to describe the most general software configuration item, quite general to describe the basic features of whatever configuration item, but quite simple to provide a simple and low-level details free representation of the objects. Using the object-oriented features of SLANG, we solved the problem of representing items in the following way:

- we listed all the fields for each module;

- we grouped fields under classes of elements related by logical relationship;

- we associated to each logical class a SLANG type, making use of *inherit* relationship.

All these considerations have been concretized in the object type hierarchy shown in Fig. 4.11.

`ItemObjType` is the hierarchy root: every configuration item not further specified is assumed being of this type. Relevant fields are:

- `Object`. It contains the physical software object stored under configuration control and free of configuration information, for example the physical source code.

- `ID`. The identifier of configuration item. This number is constructed and managed according to the rules defined in Italtel BUTD standards.

```
% (and not in alternative) with item's identifier.
ItemNumType SUBTYPE OF ModelType}
Number: INTEGER;

% Used to model insitutions involved
% in software process.
InstitutionType SUBTYPE OF ModelType
Name: STRING;
Address: AddressType;
Responsible: PersonRefType;
DepID: SET[DepRefType];

% Models a location in Development Archive.
LocationType SUBTYPE OF ModelType
Workstation: STRING;
Path: STRING;

% It is the equivalent of a human signature.
SignatureType = STRING;

% Item release.
ReleaseID = STRING;

% Models a year date.
DateType SUBTYPE OF ModelType
Day: 0..31;
Month: 0..12;
Year: 1900..MaxInt;
```

As described in other documents, configuration management items consist of both "traditional" software documents, as design documentation and source code, and forms, used mainly as a support to anomaly management. We did not model all the different kinds of configuration items used in Italtel BUTD sotware production: instead, we identified a subset of basic configuration objects to be modeled with dedicated types:

- Anomaly Report (AR). The main module in software anomaly man-

```
% Task assignment representation.
TaskAssignmentType SUBTYPE OF ModelType
Actor: PersonRefType;
Task: TaskType;

% Type to represent a human resource.
PersonType SUBTYPE OF ModelType
Name: STRING;
FirstName: STRING;
Birth: DateType;
CompanyLocationCode: STRING;
PersonalCode: PersonRefType;
ProfessionalLevel: PLType;
RoleCode: STRING;
AccountingCode: STRING;
DepartmentCode: STRING;
EMail: STRING;
Phone: STRING;
Fax: STRING;
TaskQueue: LIST[TaskType];

% Reference to a human resource. (To be defined).
PersonRefType = STRING;

% This type is used to notify the outcome
% of an operation or, more generally, a
% positive or negative condition.
ConditionType = (OK, NotOK);

% Item identifier. Used to reference
% configuration items.
ItemRefType SUBTYPE OF ModelType}
Version: INTEGER;
Edition: INTEGER;
Identifier: STRING;

% Item number. This is used in conjunction
```

```
% form (like AR or RIMO).
ItemObj: SET[GeneralFormType];
% Identifier of the person issuing the request.
Sender: PersonRefType;
% Destination of the item.
Destination: LocationType;
% Desired state of the item.
State: ItemStateType;
% Predicate used for Event Journal searches.
Pred: PredStringType;
% Method to generate unique request identifiers.
PROCEDURE GenReqID: ReqIDType;

% Objects of this type are output by
% Configuration Management.
CAResultType SUBTYPE OF CAAccessType
% Item objects of interest for operation requested.
% Note a difference between this field and the corresponding
% one in the previous declaration: here ItemObj identifies only
% the object, while previously it carries also the associated
% Accompanying Form.
ItemObj: SET[ItemObjType];
% Event Journal search results.
EJResults: SET[ModelType]
% True if the operation succeeded.
OpOK:    BOOLEAN;
% True if the item has been accepted.
Accepted:   BOOLEAN;

% Task representation.
TaskType SUBTYPE OF ModelType
Duration: INTEGER;
Priority: [1..10];
Deadline: DateType;
Description: Text;
TaskID: TaskIDType;
```

```
% Associated ARs.
AR: SET[ItemRefType];
% Reference to the actor that has to
% effect the update.
Actor: PersonType;
% Planning information. On the form is a text:
% we will use here only a date specifying the deadline.
Deadline: DateType;
% Re-qualification requisites.
Requisite:  Text;
% CCB composition.
CCB: SET[PersonRefType];
% Method used to check the module's correctness.
PROCEDURE CheckFormalCorrectness: ConditionType;

% Operations allowed during Configuration Management.
CAOpType=(Duplicate,Retrieve,Send,ChangeState,EJSearch);

% This is the type of identifiers associated to
% Configuration Management requests.
% The choice of INTEGER is only demonstrative.
ReqIDType = INTEGER;

% A supertype collecting the basic features of
% Configuration Management interaction.
CAAccessType SUBTYPE OF ModelType
%Request identifier
ReqID: ReqIDType;
% Item identifier.
ItemRef: SET[ItemRefType];

% Objects of this type must be used to issue
% requests to Configuration Management.
CARequestType SUBTYPE OF CAAccessType
% Operation requested.
OpReq: CAOpType;
% Item object, with its Accompanying Module or
```

```
% Phase of anomaly raising;
AnomOrgPhaseType =
(Review, Debugging, Qualification, Integration, Work, Other);

% Defined kinds of updates.
UpdateKindType =
(Corrective, Evolutive, Perfective, Adaptive, ToComplete);

% Anomaly Report Form.
ARFormType SUBTYPE OF AnomalyMgmFormType;
% Related TLRs & TLVVRs identifiers.
TLRRef: ItemRefType;
TLVVRRef: ItemRefType;
% Related RIMOs.
RIMO:  SET[ItemRefType];
% HW/SW release.
Release: ReleaseID;
AnomKind: AnomKindType;
AnomClass: ClassType;
AnomOrgPhase: AnomOrgPhaseType;
UpdateKind: UpdateKindType;
% Suggested solution.
Solution: Text;
% Method used to check the module's correctness.
PROCEDURE CheckFormalCorrectness: ConditionType;

% Review information.
RIMOReviewType =
(CompleteBasicAspects, PartialBasicAspects, FormalAspects);

% RIMO Form.
RIMOFormType SUBTYPE OF AnomalyMgmFormType;
% Identifier for configuration item to update.
Item: ItemRefType;
% Department identifier.
DepID: DepRefType;
```

```
RevNotes: Text;
% Review Team;
RevTeam: SET[PersonRefType];
% Drafter of the VVR.
Drafter: PersonRefType;
% Control signature.
Signature: SignatureType;
% Method used to check the module's correctness.
PROCEDURE CheckFormalCorrectness: ConditionType;

% Supertype for anomaly management forms,
% i.e. ARs & RIMOs.
AnomalyMgmFormType SUBTYPE OF GeneralFormType;
% Textual description of the problem
% (anomaly for AR, change for RIMO).
Description: Text;
% Reference to actor handling the form.
Tratt: PersonRefType;
% Identifiers for related documentation.
Rif: SET[ItemRefType];
% Accompanying documentation.
Alleg: SET[ItemRefType];
% Active copy instance associated to the anomaly
% or the RIMO. It is used whenever is necessary to
% suspend or restart the activity that actually
% solve the anomaly or the RIMO.
Instance: ActiveCopyType;
% Method used to check the module's correctness.
PROCEDURE CheckFormalCorrectness: ConditionType;

% Defined kinds of anomaly.
AnomKindType =
(Documentation, MissingFunction, NonActiveFunction,
 IncorrectFunction, ProgramInterruption, NormativeAnomaly);

% Defined classes of anomaly.
AnomClassType = (Mandatory, Important, Suggested);
```

```
ItemDate:   DateType;
% Title.
ItemTitle:  STRING;
% Physical parts of the item (pathnames).
ItemParts:  LocationType;
% Configuration requisites.
ConfReq:    Text;
% Reference to the actor that handles the item.
Tratt: PersonRefType;
% Reference to the actor responsible for the item.
Actor:    PersonRefType;
% Method used to check the module's correctness.
PROCEDURE CheckFormalCorrectness: ConditionType;

% Verification & Validation Form.
VVRFormType SUBTYPE OF GeneralFormType;
% Reviewed item physical object.
RevItem: ItemRefType;
% OK if the review was positive.
RevOutcome: ConditionType;
% Department identifier.
DepID: DepRefType;
% Review date.
RevDate: DateType;
% References to related documentation.
Rif: SET[ItemRefType];
% Documentation accompanying the item.
Alleg: SET[ItemRefType];
% RIMOs successfully solved.
RIMOOK: SET[ItemRefType];
% RIMOs partially solved.
RIMONotOK: SET[ItemRefType];
% Non-examined RIMOs.
NotExamRIMO: SET[ItemRefType];
% Associated Anomaly Report.
AR: ItemRefType;
% Review notes.
```

```
% Current item status, as defined by the
% associated FSM. It will be redefined in descendants
% with the correct type (for example ARStatusType for
% an AR) to give an automatic consistency check on FSM
% states using type checking.
Status: ItemStatusType;
% TRUE if the item is subject to qualification.
ToBeTested: BOOLEAN;
% Identifiers of executed RIMOs.
ExeRIMO:    SET[ItemRefType];
% Identifiers of non-executed RIMOs.
NotExeRIMO: SET[ItemRefType];
% Mailing list.
Distrib: SET[PersonType];
% Method used to create the mailing list.
PROCEDURE FillDistribution;
% Method used to check the module's correctness.
% Here serves only as a placeholder: it will be
% redefined in descendants.
PROCEDURE CheckFormalCorrectness: ConditionType;

% Definition of the most general form.
GeneralFormType SUBTYPE OF ItemObjType;
% Redaction Date;
Date:     DateType;
% Emitting institution.
Institution: InstitutionType;
% Information form-dependent.
CustomInfo: Text;
% Method used to check the module's correctness.
PROCEDURE CheckFormalCorrectness: ConditionType;

% Accompanying Form.
MAFormType SUBTYPE OF GeneralFormType;
% Item number.
ObjNum:      ItemNumType;
% Receiving date.
```

```
% Allowed states for a document not subject to
% verification & validation.
DocNotVVStatusType =
(None,Working, Refused, Approved, Released);

% Allowed states for a document subject to
% verification & validation.
DocVVStatusType = DocNotVVStatusType UNION
(Draft, DraftModified, ApprovedModified, ApprovedReview);

% Allowed states for Test Launch Reports.
TLRStatusType = DocVVStatusType;

% Allowed states for a Source Code Unit.
SCUStatusType = DocVVStatusType UNION (Tested);

% Allowed states for a generic configuration item.
ItemStatusType =
ARStatusType UNION RIMOStatusType UNION TLRStatusType
UNION SCUStatusType UNION DocVVStatusType UNION
DocNotVVStatusType;

% Allowed kinds of items.
ItemKindType = (DocVV,DocNotVV,VVR,TLR,SCU,AR,RIMO);

% Definition of the most general document object.
ItemObjType SUBTYPE OF ModelType;
% Physical object.
Object: SWObjectType;
% Item identifier.
ID: ItemRefType;
% Used to determine if current owner of
% the document can change it.
ReadOnly: BOOLEAN;
% Used to determine the kind of item object
% the Configuration Management is dealing with.
Kind: ItemKindType;
```

4.6 Type declarations

In this section we group all type declarations used in our specification. Some of these are of particular interest because they play a decisive role in specification structure.

It must be pointed out that since SLANG is not currently supported by a real process-centered environment, at the time of writing, we made some assumptions on the types to be considered as built-in in the SLANG modeling language; we consider as built-in types the following:

- `BOOLEAN`
- `STRING`
- `TEXT`
- `INTEGER`

Note the use of upper case letters to distinguish them from user-defined types. Obviously, in addition to listed types, also `ModelType` must be considered a built-in type, but this feature is already well-defined and included into current language definition.

Our user-defined type definitions are the following:

```
% Allowed states for anomalies. Within this and
% subsequent definitions, the 'None' status must
% not be considered a real status used during
% configuration management, but a 'dummy' state
% used when the item is not delivered to CM yet.
ARStatusType =
(None,Originated, Refused, Notified, ToDefine, Suspended,
 Pending,PartiallySolved ,Solved, Rejected, Cancelled);

% Allowed states for a modification request.
RIMOStatusType =
(None,Issued, Refused, InProcess, TimedOut, Terminated,
 PartiallySolved, Solved, Cancelled);
```

4.5 Organization of specification

The specification is related to two major activities of Italtel BUTD software process: anomaly management and configuration management, which are deeply related.

Every SLANG specification is composed of two parts:

- a *graphical* part, describing the control through the use of Petri nets;
- a *textual* part, describing conditions on events (transition guards), event execution (transition actions), data (type declarations) and operations on them (methods associated to types).

Moreover, a non-trivial SLANG specification is defined using the activity construct, which is intrinsically a top-down construct, in the sense that it allows the definition of the control specification by modularization and functional decomposition. Thus, the *activity static tree* built using the *invoke* relationship is both the description of SLANG allowed control flow among activities and a functional description of the modularization determined by modeler choices.

The specification we developed involves the development of other two activities, that are left partially specified, i.e. review management and human resource allocation management. Consequently, the specification presented here is composed of four activities *at the same level*: in other words, there is no "main process" activity defined.

Following these considerations, we have structured our specification in a top-down fashion, starting from high-level activities and developing more detailed decompositions, until we reached a granularity level allowing a description of the activity in terms of reasonably complex SLANG nets. The activity hierarchy is shown by means of the activity static tree.

Type declarations are provided apart from the rest of the specification. For every activity the linking between places and type declarations is described, together with the textual description of transition, in terms of guards and actions.

Part II

The SLANG specification of Italtel BUTD software process

3. If not:
   - priorities are set;
   - a local replanning is performed, assuming that tasks with lower priority are delayed;
     * If replanning is successful, the assignment becomes real, but replan requests must be issued in order to fit the new deadlines for delayed tasks.
     * If replanning is not successful, a global replan request is issued.

After interviews with process expert, we have pinpointed some key aspects:

- Every process actor has a list containing the tasks to be (possibly concurrently) executed.
- A process actor can get different roles at different times. For example, a developer must be also a reviewer, possibly in a parallel project.
- A task is defined by the following parameters:
  - **Duration.** It the estimated duration of the task. It is generally defined by the actor that defines the task or by the actor that must execute it.
  - **Deadline.** It is the deadline associated to the task termination. It is set up by the actor making the task assignment.
  - **Priority.** It is not always necessary: it is mandatory when there are conflicts in allocating the tasks. In this case, priority is set by the actor, in agreement with its superior.
- With respect to human resources management related to anomaly management, it is the CCB that determines task assignments.
- It is possible to define, as a first approximation, two methods for human resources allocation:
  - A centralized management, performing a global allocation, taking in account all constraints.
  - A distributed management, delegating the task management (including conflicts) to executors, to be used in central phases of the project.
- It is possible to define a little algorithm to describe human resource allocation, only for anomaly management:
  1. The actor evaluates task duration, related to the ones already owned in its agenda, verifying the absence of conflicts.
  2. If there are not conflicts the assignment becomes real.

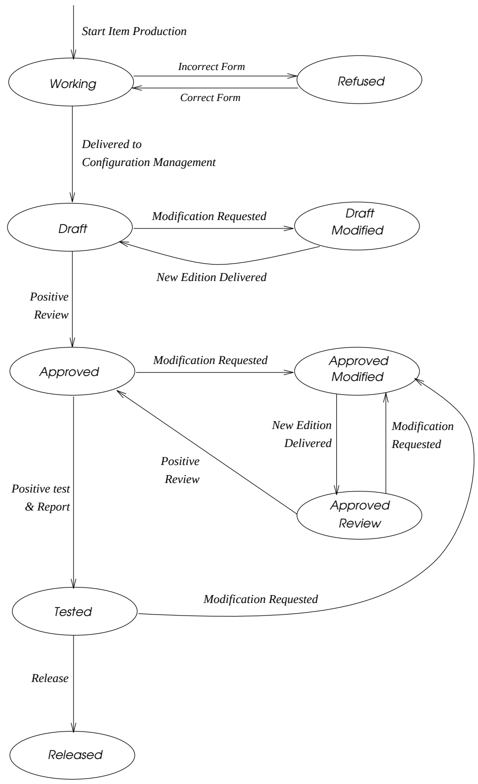


Figure 4.10: SCU automata.

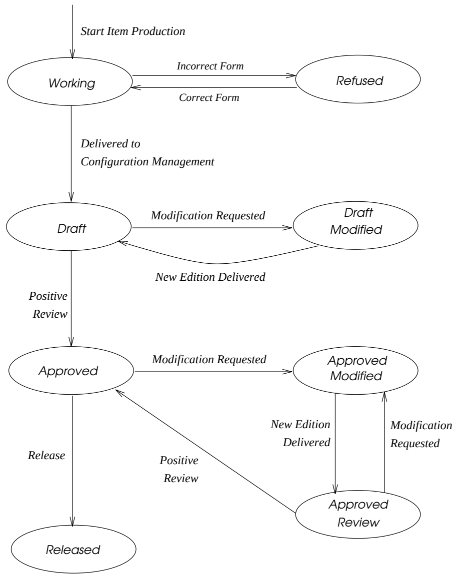


Figure 4.8: FSM for documents subject to verification and validation.

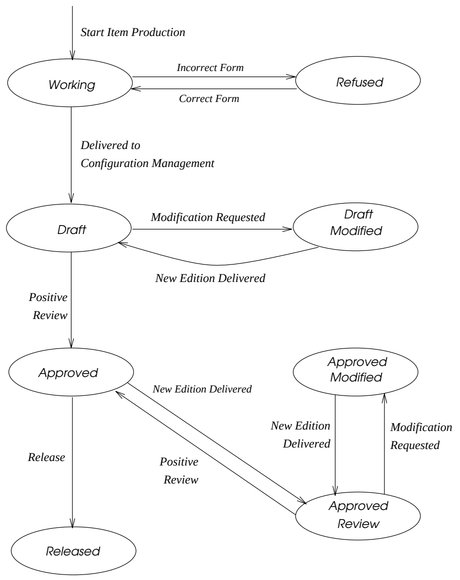


Figure 4.9: TLR automata.

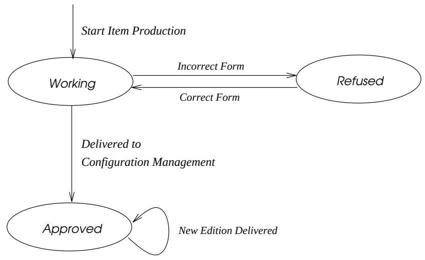


Figure 4.7: FSM for documents not subject to verification and validation.

- **Draft.** When the document takes this state, the drafter has determined that it is in a consistent state and can be sent to SGMR. Now it must be reviewed.

- **Draft Modified.** The item enters this state after RIMO issuing about it. After updates completion, the item takes the 'Draft' state again.

- **ApprovedModified.** Used whenever RIMO are issued on an item that already took the 'Approved' state.

- **ApprovedReview.** Used to signal that a document previously 'Approved' is under review due to RIMO on it.

The *Test Launch Report* automata, in Fig. 4.9, isn't very different from the previous one: the only difference is that a TLR sent to SGMR and approved cannot get RIMO because, since TLR are associated to items, a RIMO would be issued for the item, causing test re-execution.

The FSM for **Source Code Units** is the most complex among the presented automata, as shown in Fig. 4.10. With respect to the TLR automata only a state is added, **Tested**, which is used only after that the source code is not only reviewed but also tested and the review on its TLR has been positive. On the contrary, RIMO are issued on the code, changing the item state in 'Approved Modified'.

4.4.2 Human resources management issues

An accurate modeling of a software process must take in account human resources management: despite of it, these issues were not included in Quality Manual sections we own.

- **Cancelled.** CCB decided, after requests from technical people and examination of AR associated, to cancel the updates associated to the RIMO, whatever is the current state.

- **PartiallySolved.** Review evidenced that only some among expected updates are really implemented: consequently, the RIMO is re-sent to implementor.

- **Solved.** Review evidenced that the RIMO has been completely and effectively implemented. Consequently RIMO management can be considered ended.

By the way, note that in both the automata, arcs that determine inter-related consequencies are named with omologous numbers.

FSM for configuration items

Initially we had no information about this kind of automs, as for RIMO: later, we have been provided with a high detailed operative instruction describing automata for configuration items, which are specialized in several types. Here we describe the new version for these automata.

Finite state machine for **documents not subject to verification and validation** is shown in Fig. 4.7. States have the following meaning:

- **Working.** The document gets this state by default during its construction. This is not a real state since the item is not under configuration management, consequently there is no control on it.

- **Refused.** The document is refused by SGMR since it is incorrect: it is resent to drafter.

- **Approved.** Since there is no review needed, after SGMR check the item can suddenly be approved.

- **Released.** Some of this kind of documents can be requested by committent: in this case, they take this state.

The FSM for **documents subject to verification and validation** is shownn in Fig. 4.8 and is an extension of the previous one. In addition, the following states are present:

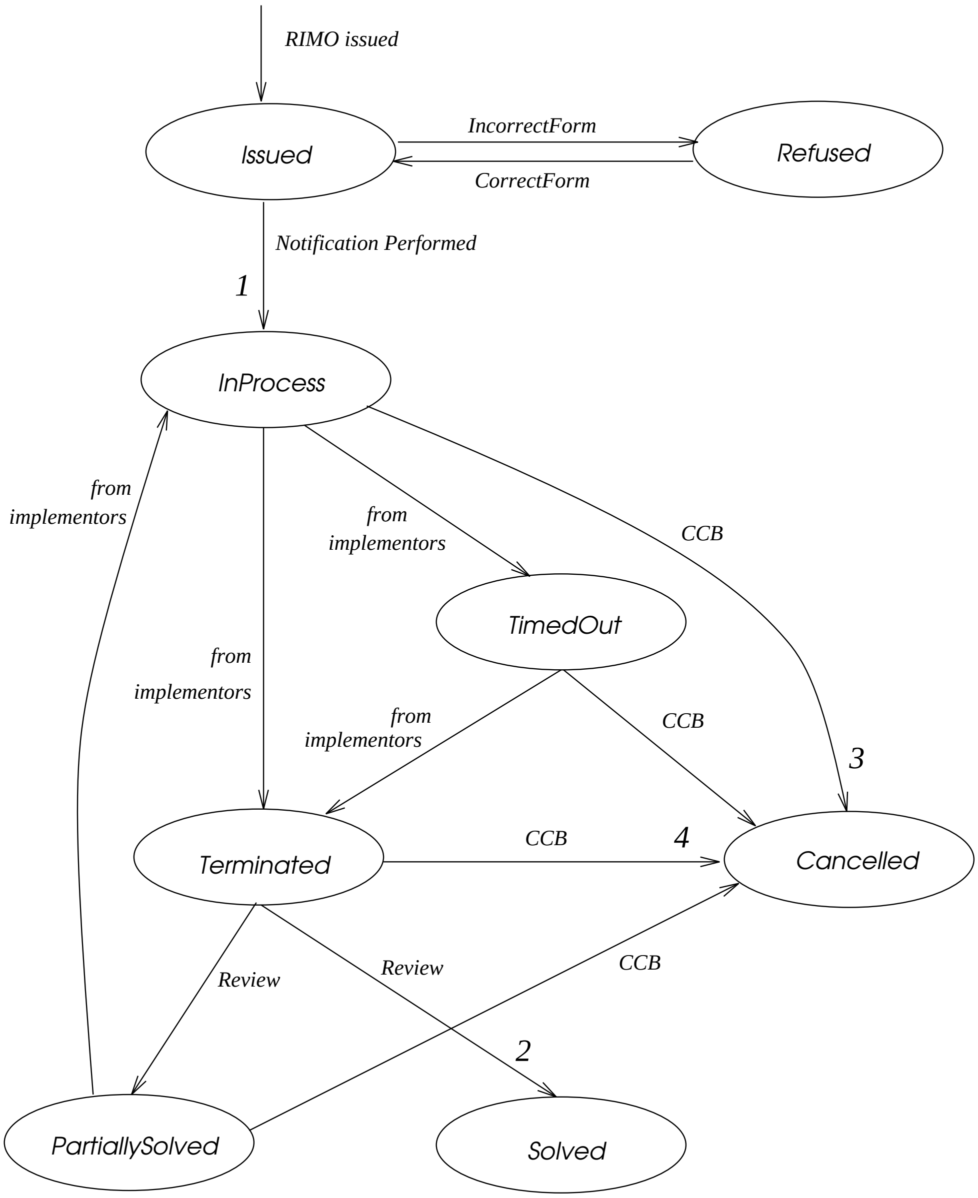


Figure 4.6: The RIMO anomaly life-cycle.

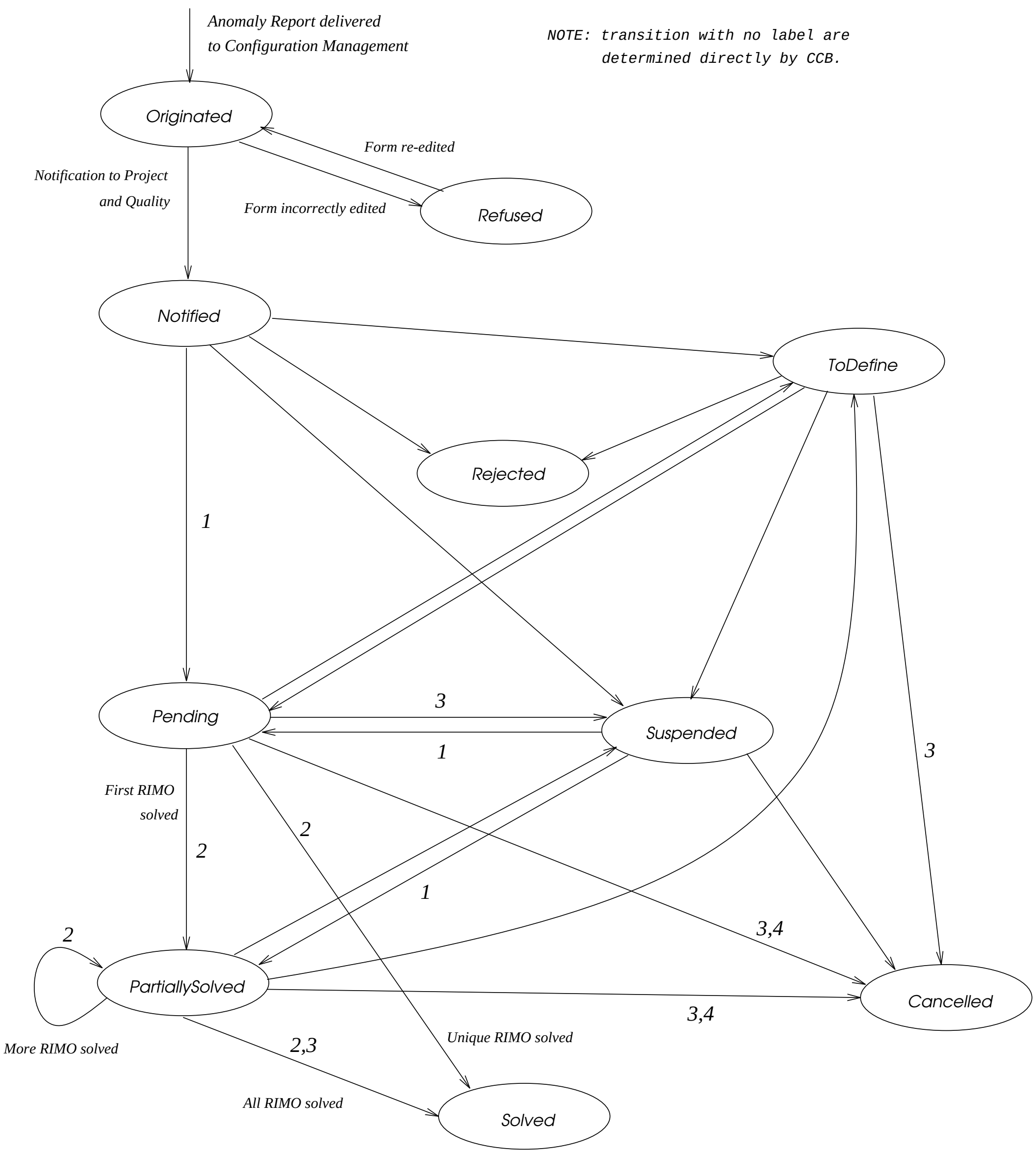


Figure 4.5: The new version of anomaly life-cycle.

Our suggestion has been accepted by Italtel and after some updates a new version of anomaly life-cycle has been issued, shown in Fig. 4.5.

This new automata has the following new states:

- **ToDefine.** It is needed more information to determine if the anomaly must be accepted.
- **PartiallySolved.** The anomaly gets this state when at least a RIMO associated to it has been terminated and positively reviewed.
- **Cancelled.** The anomaly must be cancelled.
- **Refused.** The Anomaly Report is incorrect: the form must be re-sent. This state has a meaning during metrics collection.

FSM for Modification Request (RIMO)

As said before, a RIMO FSM was already in use, despite it was not included in Quality Manual: we found this ineffective, due to underspecification and lack of cancellation issues, so we developed a more faithful automata, shown in Fig. 4.6:

This life-cycle is characterized by six states:

- **Issued.** The RIMO has been generated by CCB after examining the anomaly.
- **Refused.** The RIMO sent to SGMR has incorrect aspects: it is notified to originator for corrections.
- **InProcess.** The RIMO has been notified to the process actor the must implement it.
- **TimedOut.** The RIMO under development has missed the associated deadline.
- **Terminated.** The RIMO implementor decides that the RIMO is ready to be sent to Configuration Management. Now the RIMO must be reviewed.

4.4 Issues not included in current version of Quality Manual

In this section we present the process features that we identified as lacking in current Quality Manual description and tried to specify "from scratch" in our model.

4.4.1 New FSMs for configuration items and forms control

As describe before, in Quality Manual are described two Finite State Machines (FSM): one is used to specify control to be applied to AR and the other the control to be applied to configuration items.

After talking with the expert, we knew that other FSM were used in really enacted process, even if not included in Quality Manual, including FSM for RIMO and a set of FSM specialized depending on the type of configuration item.

Our work consisted in extending AR and RIMO FSMs, making them more faithful with respect to the real process, and extend and uniform the configuration FSM.

Here is the result of our work.

The FSM for Anomaly Report

Looking at the original automata, shown in Fig. 4.1, you can see that there is no mean to specify the cancellation of an anomaly whose state is "Pending"; instead, this would be a major feature in controlling AR, since the cancellation of an anomaly is a very frequent event in process management.

This observation is supported also by the ISPW-6 example [K^+90], which requires the modeling of a piece of software process involving a change in a module and allowing the cancellation of change request. Moreover, it is intuitive to think that a decision about an anomaly solving can be cancelled for many reasons: unexpected costs (in terms of time and resources), unexpected theoretical and/or implementation problems, third-party ceased collaboration, and so on. So this is a critical issue within software process.

The first aspect has been defined together the expert and has been described within our SLANG specification. See for example the implementation of ManageConfigurationArchive, to model the answer to incorrect modules.

The second aspected listed above is quite startling if we think that temporal issues are one among the factors that make a process description mandatory and one of the most important source of money loss for a software factory.

The last one is a very important aspect in software process: we have even built a whole activity, ExtraCCBMeeting, to model it. By the way, this activity is not specified yet, since it must be modeled from scratch; this is one of the goals of the second part of our project.

4.3.2 Ambiguities aspects

Ambiguities are scattered through the Manual and mixed with incompleteness and inconsistency aspects. For example, whenever the Quality Manual talks about the Project Function, the phrase "Development and/or Qualification" is repeated, with no indication on how correcty interpret this statement. Moreover, it is not clear what is done when an anomaly is raised for design phase, or for every phase not involving directly nor test or source code. The solution to this second aspect is simply the bypassing of testing phase, performing only verification & validation and review over changed items. This is not modeled correctly within our SLANG specification. A review of our work will keep track of this, to include it in a later specification version.

4.3.3 Inconsistency aspects

Inconsistencies have been (and will be) the most boring thing we have to face with during the software process elicitation work. As stated before, this can be understood if we think that the documents we own have been redacted by different persons at different times with different aims, nevertheless this can be a serious obstacle to Quality Manual deep understanding. As an example, the most boring inconsistency was that diagrams shown different things with respect to the associated textual descriptions, this probably due to temporal offsets in redaction times.

3. A formal notation generally impose a methodology or some constraints to its user, restricting the range of things that a modeler can leave unspecified. An example is the activity construct whose interface, in terms of inputs and outputs specification, constrains the user to specify these issues.

4. Graphic notations, like SLANG, help understanding the specification: a better understanding probably will lead to better specifications.

Italtel has widely supported us during our modeling work, helping us to solve problems by means of direct interaction with experts. This work has been useful for both the parts, since Italtel has sometimes modified its process description according to our suggestions.

The interaction has been in terms of question lists sent to the expert of quality software process: the answers came either by answer lists or verbally.

What follows is a report of the results of the interaction with Italtel. We will examine the problems raised with respect to the classification made above, with the exception of 1), that will be expanded in Section 4.4.

4.3.1 Incompleteness aspects

Some aspects are left unspecified within Quality Manual description, and among these some are substantial:

- No recycle is specified when an action performed by some process actor is wrong: examples are an incorrect module sent to SGMR, or a modified item that fails re-qualification.

- Sometimes informations are “hidden”, like temporization issues that are never described explicitly in the portion of Quality Manual we have.

- A RIMO becomes Solved when all the expected updates has been performed and their effectiveness has been proved. No explicit reference to test method or everything else is done.

- No information was provided about state transition for RIMOs and new anomaly life-cycle.

- If VVR shows a negative outcome, the software anomaly management procedure must be started.
- Updates the Event Journal.

4.3 Problems with Italtel Quality Manual

Quality Manual has been our main reference to understand Italtel software process; despite of it, we found that in some part the Manual is:

1. **Ambiguous.** The Quality Manual is written for the most part in natural language, which is inherently ambiguous. A little level of formalization is introduced by means of finite state machines and semiformal diagrams, but they are not widely used within specification.

2. **Inconsistent.** The Quality Manual description is very large in extension, and quite complex, consequently is very difficult to maintain a full-consistent version of the Manual, even after the annual review of the Manual, performed by Quality Function.

3. **Incomplete.** Some aspects of software process, sometimes important ones, are left undescribed, or described with a detail level not sufficient to provide a full understanding of the problem.

4. **Improvable.** Some parts of the description can be reformulated not only to better reorganize the description, but also to change basic aspects, making more flexible the software process structure.

Note that the use of SLANG as process modeling language instead of natural language would help to solve problem, since:

1. A formal notation, avoids ambiguity problems, if defined in a unique way. SLANG syntax and semantics are formally defined [Lip93].

2. A formal notation does not avoid inconsistency problems by itself, nevertheless it allows the construction of systems for automatic checking of information: an example is the type checking mechanism found in SLANG.

- Starting from reference documntation for the specific phase, the Development Subfunction or Qualification Subfunction build the expected document. This document has to be sent to SGMR, together with the Accompanying Module for first sending.
- SGMR, when receiving the document, performs the following activities:
  - Accept or reject the module performing consistency checks with Configuration Plan and standards.
  - Assigns the 'Draft' state to the document.
  - Registers the document in Configuration Archive.
  - Notifies SVVP of the presence of a new document and consequent review need, while sending a copy of the document to be verified and validated.
  - Updates the Event Journal.
- SVVP, when notified, performs the following activities:
  - Summons the Review Team, composed similarly to the one for anomaly management.
  - Ensures, as expected by Quality Plan, that review support documentation is available.
- Review Team performs the review. As outcome of this task, the person representing SVVP issues the Verification and Validation Report for the considered item and sends it to SGMR, possibly together with Anomaly Reports raised in the review.
- SGMR, when receiving the VVR, performs the following activities:
  - Accept or reject the module performing consistency checks with Configuration Plan and standards.
  - Stores the VVR under configuration control with the 'Approved' state and possibly the ARs in 'Originated' state.
  - If VVR shows a positive outcome, the document takes the 'Approved' state and all people interested is notified of positive review, using the specific mailing list for the document.

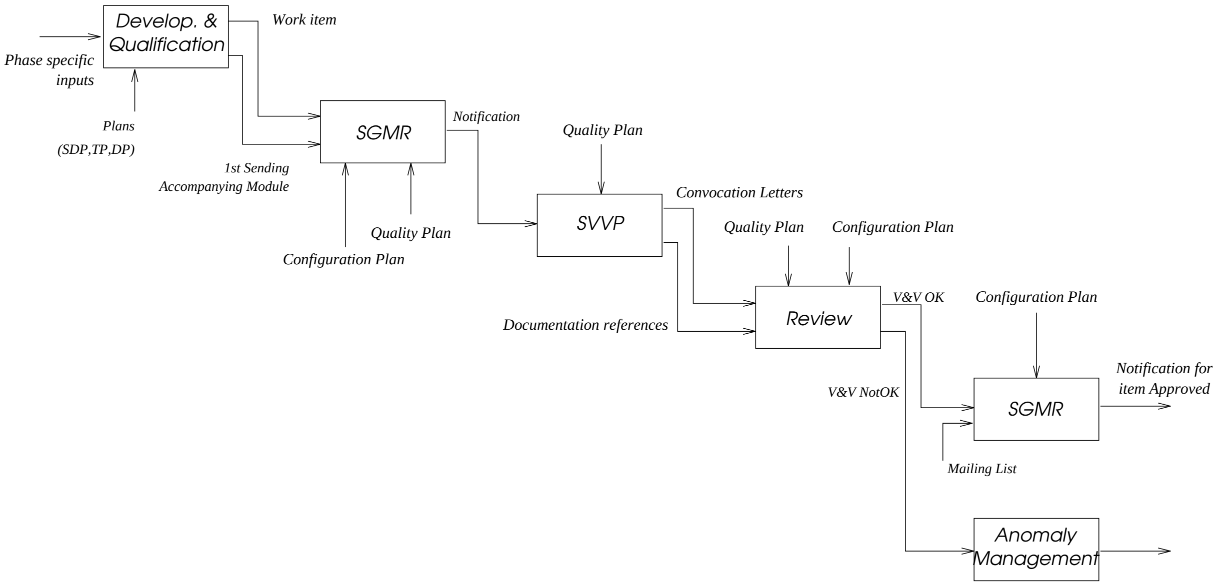


Figure 4.4: The information flow relative to documents and code configuration management.

- the modalities for final release;
- the structure and update modalities for the Event Journal, on which all relevant events (sending, updates, anomaly reports and so on) involving Configuration Management must be recorded.

Detailed description of steps to be performed

The operative instruction describing Configuration Management (`A7060-W23-A880-*-35`) distinguishes between documents and code management and test management: since we did not approach the test management, except for the part directly involved in anomaly management, we will not describe it here. By converse, we modeled all the documents and code management, consequently we describe this one only.

The figure 4.4 shows the information flow needed to manage documents and code kept under configuration control. Operating details are reported below:

    * anomaly raised on an "Approved" element and/or during test phases or use: a new edition of the item is generated.
  - Notifies the groups interested in update implementation.
  - If RIMO(s) are solved, SGMR analyzes the associated Anomaly(ies) state: if all the RIMOs associated to an anomaly are solved, the anomaly is solved too.
  - If the verification & validation activity ended with a negative result, SGMR updates the updated configuration item edition and notifies the Development and/or Qualification Subfunctions authors of the modification. The anomaly and associated RIMOs remains in the current state.
  - Updates the Event Journal.

4.2 Software Configuration Management

Definition. The goal of Software Configuration Management is the identification and the management of items composing the software product.

In particular, the term *identification* means the attribution of a unique identifier to documents and source code composing the software product, while the term *management* means the application to documents and source code of operative instructions allowing to know, at every time, the set of "official" items composing the software product, consequently ensuring that Functions involved in product development work always with consistent and updated items.

In order to achieve this goal, it is necessary a definition for:

- the modalities for sending an item to SGMR;
- the modalities for the attibution of identifiers to documents and code;
- the information flow related to review activities on configuration items, within software product development;
- the information flow related to software product qualification activities;
- the information flow related to software anomalies management;

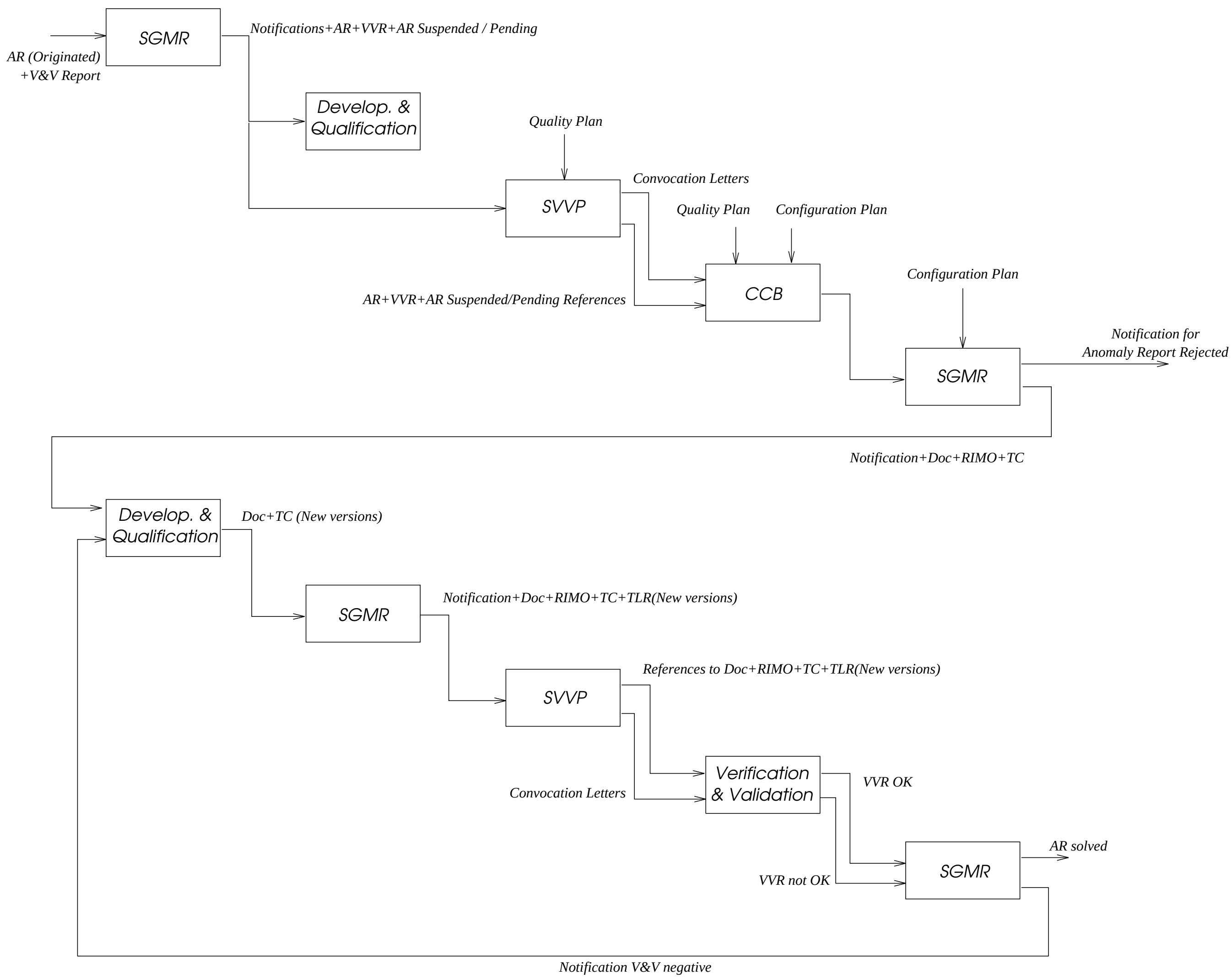


Figure 4.3: A diagram showing information flow in Italtel Software Anomaly Management.

- Accept or reject the module performing consistency checks with Configuration Plan and standards.
- Updates the edition of modified configuration items and possibly of TLRs.
- Notifies the SVVP that a review activity is needed for the configuration elements. With the notification, a TLR copy is sent, if available.
- Updates the Event Journal.

- Verification & Validation Subfunction summons the Verification & Validation team. V&V Team composition refers to the one used for the review of the previous edition of the items examined, including the authors of the associated RIMOs.

- Verification & Validation Team performs the review activity analyzing available documentation and particularly the RIMO module, the updated version of the considered configuration item and the TLRs, if any. On this basis, the V&V Team determines if the RIMO can be considered satisfied. A RIMO is *solved* when all expected modification are implemented and their effectiveness is tested.
  As outcome of the V&V Team meeting, a Verification & Validation Report is issued and sent to SGMR, relative to the examined configuration item.

- SGMR, when receiving the Verification & Validation Report, performs the following operations:
  - Accept or reject the module performing consistency checks with Configuration Plan and standards.
  - Stores the Verification & Validation Report under Configuration Control with "Approved" state.
  - If verification & validation activity ended with a positive result, SGMR updates the state or edition of modified element. When the result is positive the following cases may happen:
    * anomaly raised during a configuration item in the "Draft" state: the item change its state to "Approved".

updated and associated RIMO references are indicated. This updated version of AR is sent to SGMR.

- The SGMR performs the following operations when the AR is received:
  - Accept or reject the module performing consistency checks with Configuration Plan and standards.
  - Registers into Configuration Archive the AR with the state indicated by the updated AR, issued by CCB.
  - If the anomaly has been accepted with immediate solution proposal, registers into Configuration Archive the associated RIMO modules.
  - Notifies Development Subfunction and/or Qualification Subfunction about RIMOs.
  - Updates the Event Journal.
- If the anomaly is accepted and RIMOs are issued, Development and/or Qualification Subfunctions, when receiving the corresponding notification must:
  - Implement modifications to documents and/or code according to what specified within RIMOs. To allow this, a copy from Configuration Archive to Development Archive is performed. Updates are effectuated on this copy.
  - Send to SGMR the updated configuration items.
  - Test the updates efficacy. To do this, if the updates involves the software [2] the Qualification Subfunction performs the tests defined during CCB on the software already sent to SGMR. The TLR so generated is sent to SGMR. If the update is relative to test code, the updated tests must be re-executed by Qualification Subfunction, and sent to SGMR.
- SGMR, when configuration items and (possibly) TLR are received, performs the following operations:

[2] This is the term used in Quality Manual. The correct meaning is "source code".

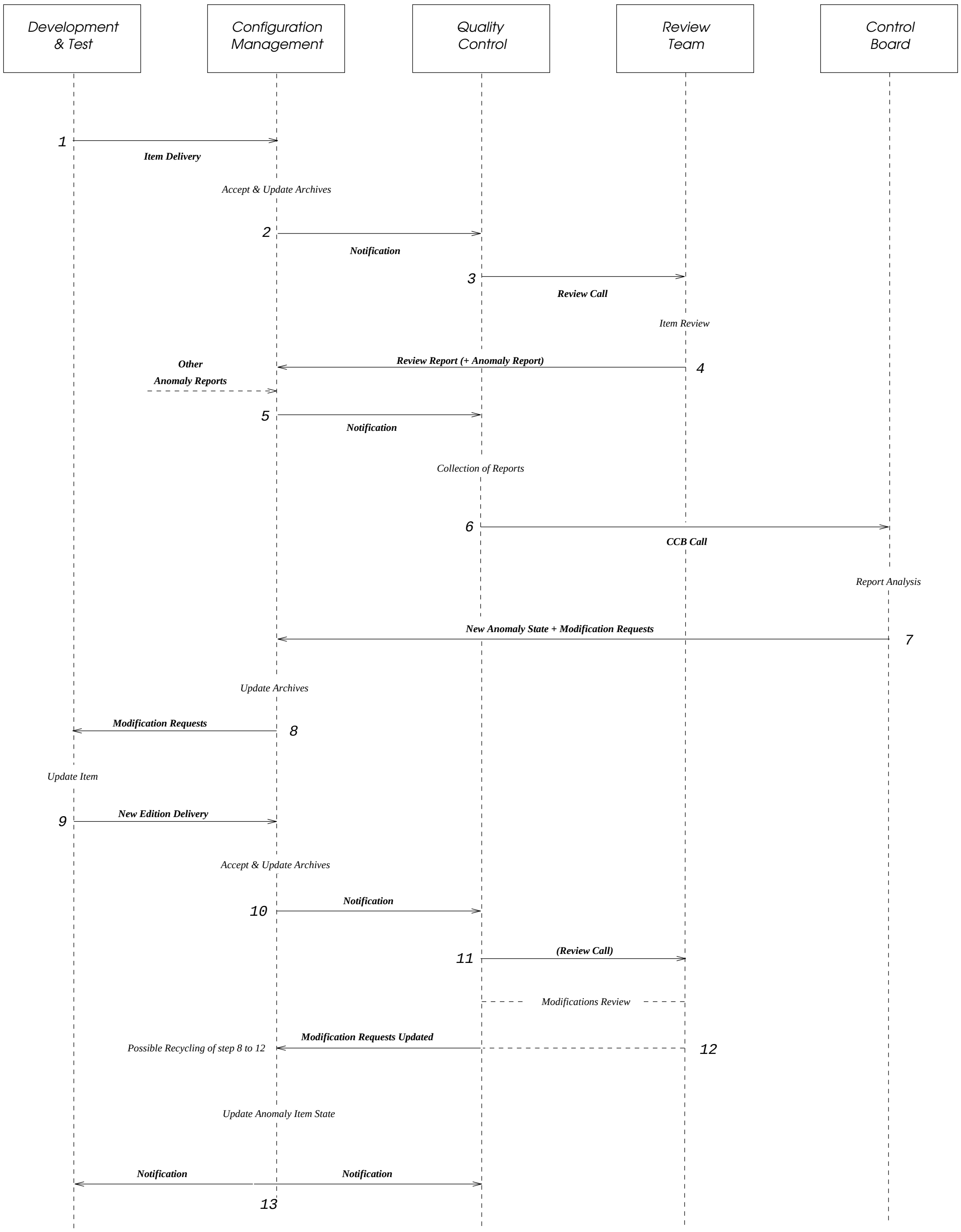


Figure 4.2: A diagram showing control flow in Italtel Software Anomaly Management.

- SVVP convenes a meeting for CCB. To determine CCB composition SVVP refers to both hardware and software Quality Plans. CCB composition depends on the current life-cycle phase and project features. A CCB is frequently composed of:

  - system analyists, both hardware and software, more frequently for SSR and SR phases;
  - the Committee, if expected by contract or when the anomaly has contractual effects;
  - responsibles for Development and Qualification Subfunctions involved;
  - persons belonging to the organization that will perform hardware/software integration;
  - users of items involved in anomalies.

  CCB convocation is performed through convocation letters.

- CCB examines the anomaly/ies. The meeting can end with one of the following outcomes:

  - The anomaly is accepted and some RIMOs are issued, carrying the modification suggested to solve the anomaly. The anomaly becomes "pending". CCB compiles a RIMO module for every configuration item to update: modules so compiled are sent to SGMR. RIMO associated with a "pending" anomaly must be immediately solved, within a predefined deadline.
  - The anomaly is accepted but no solution is defined; the anomaly becomes "suspended". The solution for this kind of anomaly is deferred to a subsequent version/release of software product, or it can be deferred to a further CCB meeting, when are identified items needed for the solution but not immediately available.
  - The anomaly is not recognized as a real anomaly, so it is "rejected".

  However, the CCB for every anomaly updates the Anomaly Report, registering the meeting's outcomes. In particular the anomaly state is

At every CCB convocation, all notified, pending and suspended anomalies and associated RIMOs are examined.

When Qualification Report is written, references to all non-resolved or rejected anomalies are included.

The Operating Instruction for Software Product Anomalies and Updates Management describes with more details what is done during software anomaly management. Now we will describe the steps composing this activity, with the help of two diagrams showing information flow (Fig. 4.3) and control flow (Fig. 4.2).

Detailed description of steps to be performed

- SGMR, when receiving the Anomaly Module performs the following operations:
  - Accept or reject the module performing consistency checks with Configuration Plan and standards.
  - Registers into Configuration Archive the AR with "Originated" state.
  - Notifies to:
    - Development Subfunction.
    - Qualification Subfunction (when the anomaly is raised during Qualification Phase or later).
    - Project Verification & Validation Subfunction (SVVP).

    the presence of an "Originated" anomaly. After this notifying, SGMR change the anomaly's state to "Notified".
  - Provides the reference to the possible Test Launch Report(TLR) or Verification & Validation Report (TLRVVR) associated to AR and the references to every pending or suspended ARs to allow an integrated evaluation.
  - updates Event Journal.
- Development and/or Qualification Subfunction, when notified, start the necessary action for technical evaluation of the anomaly. This will be discuss within CCB.

particular life-cycle, described in Fig. 4.1 a full-described iter from this point to solution is provided [1].

States and transitions have the following meaning:

- The anomaly is signaled when the AR is given to SGMR. At this time the anomaly becomes **Originated**.
- SGMR send a copy of the Anomaly Report to Development and Qualification Subfunctions and to Project Verification & Validation Subfunction (SVVP). In such a way the anomaly becomes **Notified**. The three groups above examine anomaly within CCB.
- During this analysis the anomaly state can change to:
  - **Rejected**, when the described behavior or problem is not considered anomalous. This is a terminal state, i.e. no further actions are needed.
  - **Pending**, when the anomaly is really recognized and needs, to be solved, an update to either a document or a source code or a test. A pending anomaly needs mandatorily a solution, identified within CCB. By CCB are issued one or more Modification Requests (RIMO) addressed to Development or Qualification Subfunctions.
  - **Suspended**, when the anomaly is really recognized but its solution is delayed. However, a suspended anomaly must be solved: it becomes "Pending" when conditions hold for solving or becomes "Rejected" if conditions holds to decide that way.
- A "Pending" anomaly becomes **Solved** when all RIMO associated with it are solved.

The state of every existing anomalies is registered into the **Events Journal (Giornale degli Eventi)**, which is a log permitting the retrieving of temporal information about configuration item creations and state transitions.

[1] Here we describe the original anomaly life-cycle reported by Quality Manual. This is slightly different from the one currently used by Italtel and updated following the suggestion raised by our modeling work. These suggestion and the new anomaly life-cycle will be described in Chapter 4.3

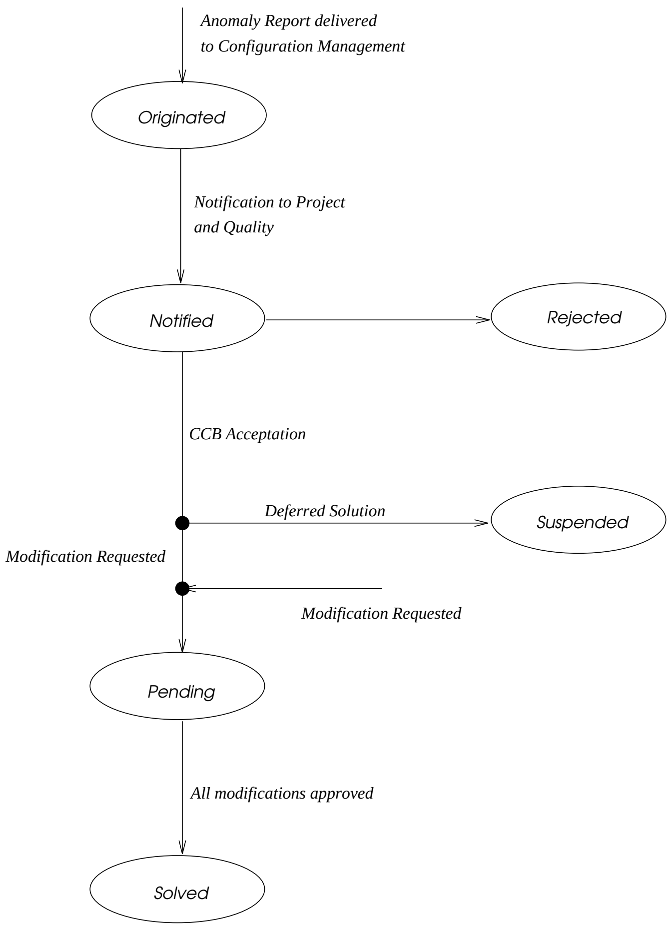


Figure 4.1: The finite state machine describing Anomaly Report allowed state transitions (old version).

- **Change Control Board (CCB).** Change Control Board (Commissione di Valutazione) is composed of Development, Qualification and Verification & Validation Subfunction components. Its goal is to examine Anomaly Reports and Modification Proposals (PROMO) issuing Modification Requests (RIMO).

- **Development Archive.** This archive contains items composing the software product on which people are working.

- **Configuration Archive.** This archive contains items composing the software product and managed under configuration control.

- **Release archive.** This archive contains items composing a released software product.

Anomalies can be raised by any originator, including users, both during production phases and while in operation, but frequently during verification & validation phases, particularly:

- As negative outcome after a review.
  In this case, verification & validation team issues the VVR and one Anomaly Report for every raised anomaly.

- As a particular case of verification & validation activity, during testing.
  In this case the Project Qualification Subfunction issues TLRs and an Anomaly Report for every raised anomaly.

Every anomaly describes an incorrect behavior or an inadequacy with respect to the software product, so it is not necessary linked in an unique way to a single configuration item. The following relationship holds:

- an anomaly can be solved by many updates;

- an update can solve many anomalies;

- an update can be generated without anomalies.

Whatever is the originator, to signal an anomaly, an Anomaly Report module must be compiled and sent to SGMR. Since anomalies own their

Chapter 4

The software process actually modeled

In this chapter we describe the contents Quality Manual sections detailing the process parts on which we focused the modeling work.

This are the management of software anomalies and configuration control, two key issues in process management.

4.1 Software Anomalies Management

Among the documentation we have been provided with, the following procedures are of particular interest when dealing with this problem:

- Software Product Anomalies and Updates (`A7060-W22-A820-*-35`).
- Software Product Anomalies Management (`A7060-W22-A821-*-35`).
- Operating Instruction for Software Product Anomalies and Updates Management (`A7060-W23-A881-*-35`).

These procedures contain definitions to provide the necessary background and understanding of the problem. Among these definitions, the following are basic:

- **Anomaly**. An anomaly is a software product irregularity with respect to what expected. Generally it is raised by verification & validation or audit activities.

- **Verification**: this is the review activity ensuring that the final outcome of a phase is coherent with requirements defined at the phase starting.

- **Validation**: this the review activity ensuring that the final outcome of a phase is coherent with fixed starting requirements.

The review is a task for a different role with respect to the one who produced the on-review item.

3.7 Relations with other standards

Since the process we are studying is developed for the needs of a military department, there is no wonder it is highly related with standards coming from American Department of Defense or other military organizations. In particular, Italtel BUTD Software Quality Manual reports some tables showing direct correlations between the description in the Manual and the following military standards:

- *MIL-STD-2167a* [DOD88], standard developed by American Department of Defense.

- *AQAP-13* [AQA87], a NATO full-comprehensive software process standard.

- *ISO-9001-3* [ISO91]. International Standard Organization Quality Management and Quality Assurance Standards.

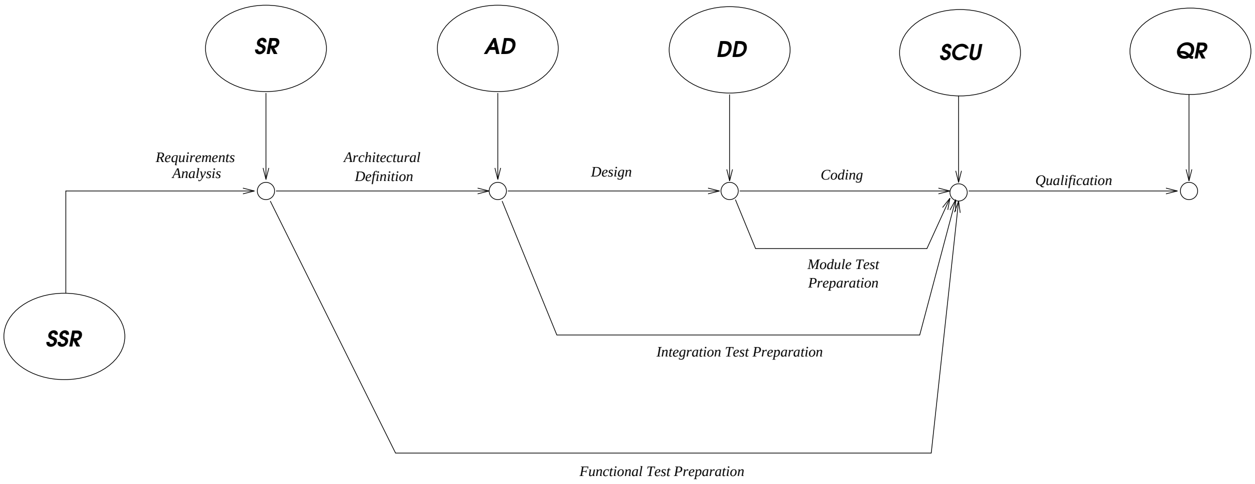


Figure 3.3: Italtel's BUTD life-cycle description

Qualification (phase 6): tests defined within the previous phase are executed and Qualification Report is built. The qualification activity is performed in a bottom-up fashion, starting with the testing of modules, which are followed by integration testing and finally functional testing. At the end is built the Qualification Report. For each step the appropriate TLR is issued. During test execution anomalies can be raised: in this case an Anomaly Report is compiled.

3.6.4 Review activity

Review activity is performed concurrently to the development life-cycle, at each step of it. The following are naming conventions used to specify this parallel activity:

- **Review**: is the formalized activity whose task is the issuing of a verification & validation document relative to software product element.
- **Review Team**: this is a board composed of development, qualification and quality functions' persons, whose goal is to assess the reviewing items and output the relative Verification & Validation Reports and possibly Anomaly Reports.

- verification & validation.

3.6.1 Development activity

It is divided in four phases:

- *Requirements analysis (phase 1)*: starting from SSR one or more SR are determined, describing detailed functional requirements for any software component.
- *Architecture definition (phase 2)*: starting from SR, high-level physical architecture of software component is defined, showing software subcomponents and their interfaces.
- *Detailed design (phase 3)*: starting from AD document, one or more DD are constructed; they may be further decomposed in subcomponents, reaching modules.
- *Coding (phase 4)*: for every DD the corresponding module set is built, with every module physically structured in compilation units.

3.6.2 Test Preparation activity

Test activity is performed through repeatable tests and a methodology allowing to determine the test coverage. Test strategy is replanned for every single project, in order to define the best strategy to suit functional and structural features of software product. Test are prepared and driven by persons belonging to Software Qualification Function but different from the ones who produced the on-test software component. Test preparation is constituted by the following phase:

Test Preparation (phase 5): this phase comprehends Functional, Integration and Module test preparation.

3.6.3 Qualification activity

Qualification activity is composed of the following phase:

- rules to adopt during product element identification and configuration management;
- anomaly management and related corrective actions;
- management and storing of physical media;
- third parties' software control.

Other expected planning documents are :

Software Development Plan (SDP). This document identifies the issuing of a project document on a temporal basis, human resources, tools and computational resources, and other activities needed to coherently develop the program. Writing and issuing responsibilities for this document are of Project Function.

Configuration Plan (CP). This document identifies, within general configuration management criteria, the list of element constituting the final software product in terms of all documents and modules to be developed. The document is issued as "partial" and then enriched during product development. Writing and issuing responsibilities for this document are of Project Function.

Test Plan (TP). This document describes test strategy. Test plan defines approach, methods, tools suitable for every expected test and qualification activities. Writing and issuing responsibilities for this document are of Software Qualification Function.

3.6 Product life-cycle

Figure 3.3 shows the whole life-cycle adopted by Italtel. As shown by the figure activity flows are divided in:

- software development;
- test preparation;
- qualification;

3.4.2 Qualification documentation subject to verification & validation

Qualification Report. This document contains all the references to verification & validation outcomes and documents related, allowing the assessment of product quality and subsequent delivery.

Also Quality Plan belongs to this document set, as described later in Section 3.5.

Moreover, Quality Function issues all Verification & Validation Reports (VVR) output by Review Team at various levels in the process, with the sole exclusion of Quality Plan VVRs issued by Project Function.

3.5 Planning

This section is of particular interest since contains the information about documents needed to deal with temporal issues in process management.

During the initial phase of the contract a **Quality Plan (QP)** is redacted, containing:

- responsibilities for the various activities allocated within organizational structure;
- verification & validation activity according to Software Development Plan;
- software product identification;
- list of document to issue, with responsibilities references;
- standards, techniques, conventions to use during development and qualification;
- verification & validation activities list, allocated on a element basis, describing methods and tools to use and possibly constraints for approving;
- test strategy guidelines;

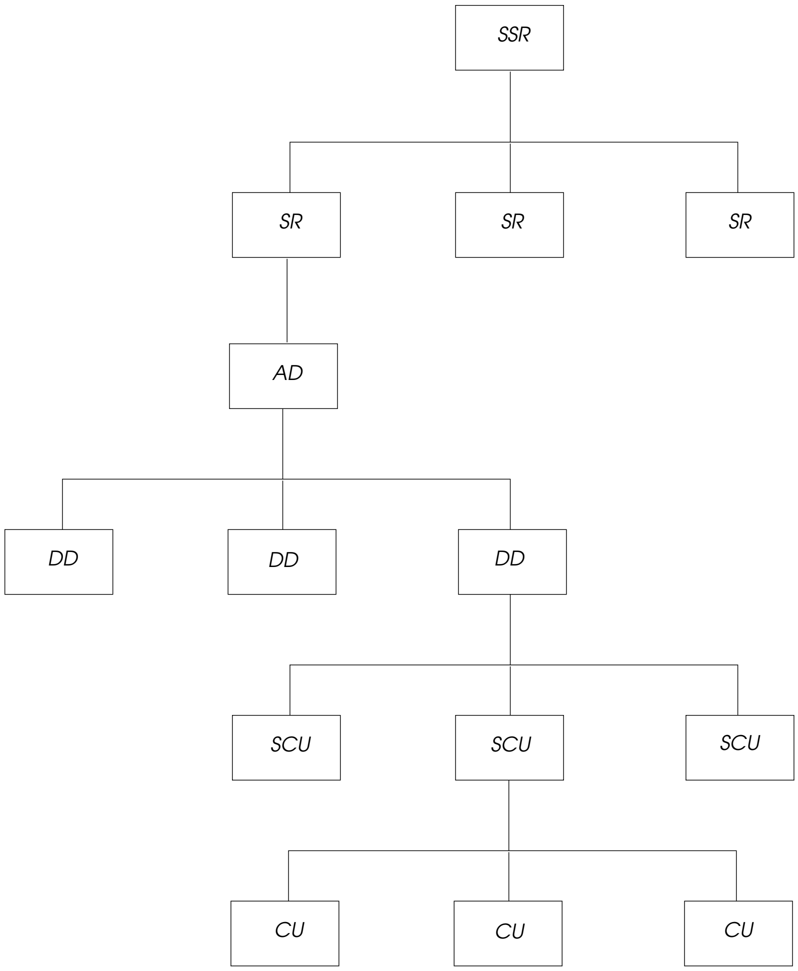


Figure 3.2: Italtel's BUTD software product structure.

offered by the programming language.

User Documentation. This documentation is composed of one or more manuals describing:

- product installation in working environment;
- use of software product, with respect to final user;
- product maintenance.

Check List (CKL). This document allows the Software Qualification Subfunction to correctly project tests and allows the Verification & Validation Function to check if the amount of testing defined is adequate to the goal. Checklists are differentiated by *typology* (Functional, Integration, Module) and by *level* (SR, AD, DD). Checklists describe what tests must be performed, with respect to functionalities, interfaces, data, special execution sequences.

Test Specification (TS). This document describes the goal and ways of execution of a test, with explicit reference to relative checklists and collocation of the test in the general test strategy (preconditions generated by other tests and so on).

Test Code (TC). This document contains, according to the particular situation, test data definition, test program (source code) and/or non-automatical test procedures. Test Code is subject to a standard; it is not directly subject to verification but it is included into the corresponding TLR review.

Test Launch Report (TLR). This document contains test execution results, containing informations about outcomes, describing execution conditions, identifying executed tests and raised anomalies.

Software Development Plans, Test Plans, Configuration Plans fit in this area, but will be described in Section 3.5. Fig. 3.2 shows the structure of development documentation. Depending on program complexity, other documents may be issued even if not subject to verification & validation.

reliability and maintainability.

Software Requirements (SR). The document contains reference to SSR document and describes functional requirements of one among the software parts identified in SSR as software component. In particular, are treated here the following:

- *user issues*: functionalities, use and execution modalities, input and output protocols, error messages, commands, help and so on;
- *project issues*: throughput, turnaround time, resource allocation and so on;
- *qualification issues*: it indicates specific needs in term of quality attributes and functional tests implementation.

Architectural Design (AD). The document contains the high-level description of the physical structure of a software component described in the specific SSR (for every SR there is one and only one AD). It reports indications about identified subcomponents, detailed descriptions of subcomponents input-output informations, and detailed description of transformation performed. In other words, AD is the document described by the "functional specification" of all subcomponents identified at the first level of decomposition.

Detailed Design (DD). Detailed design inherits both philosophy and structure of AD. The aim of this document is to allow further decomposition in smaller elements, better manageable in terms of complexity and dimension. Detailed design constitute a reference for the test implementation phases. Dealing with small products it is allowed to glue AD and DD documents in only one document. The so originated document is called AA-DD (Architectural and Detailed Design).

Source Code Unit (SCU). This document contains the source code relative to a software component identified as the smallest executable and testable one, written using the selected programming language. It is made of Compilation Units (CU), code portions compilable in a separate way. Each of this Compilation Units can be further articulated according to features

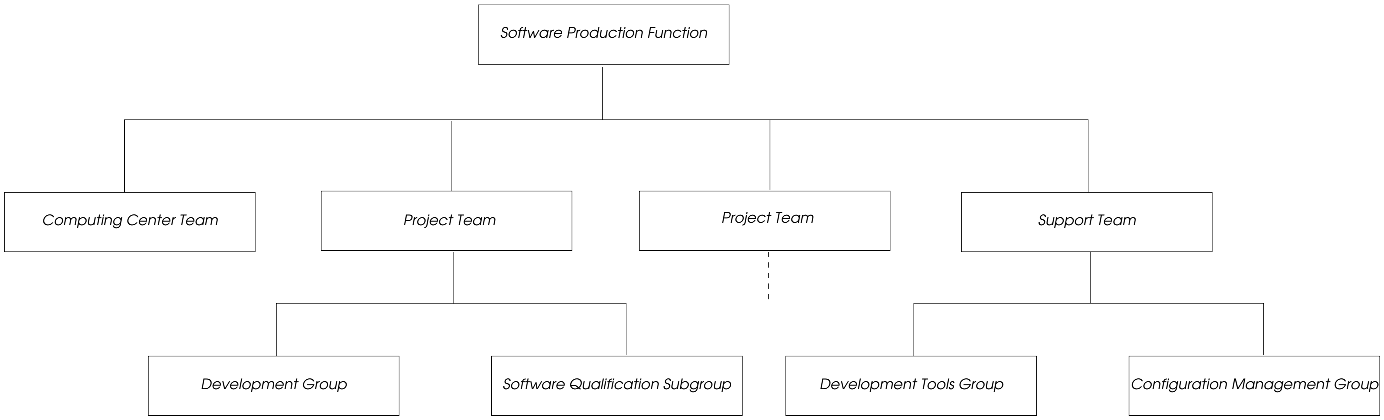


Figure 3.1: An organizational view of Italtel's BUTD software process roles

Quality Function acts with its own Verification & Validation and Audit Subfunction.

A diagram explaining relations among roles is shown in Fig. 3.1.

3.4 Software documentation

We describe here the documentation that must be produced during the software process enaction. The understanding of terms, definitions and acronyms, though boring, is mandatory to understand the SLANG specification.

3.4.1 Development documentation subject to verification & validation

Software System Requirements (SSR). This document is the result of the identification of product parts to develop as software product. It shows the first level of physical architecture of software system, describing components, their correspondence with functional areas and relationships. Moreover, it describes the working context where the product will be used and high-level functions this will perform acting on surrounding environment, including hardware interfaces and interdependencies, and requirements of

- **Development Subfunction**, responsible of document and code implementation and of presence in Verification & Validation Team and Change Control Board (CCB).

- **Software Qualification Subfunction**, responsible of definition, implementation and execution of qualification activities according the formalities reported in operative instructions defined by Quality Function and of presence to Verification & Validation Team and Change Control Board (CCB).

Quality Function. Quality Function is responsible for validation and conformity check of software product and integration validation on the whole system, and particularly:

- Issues Product Quality Plan.

- Performs verification & validation activities to assess the correct process operability in predefined checkpoints, according to predefined methods and procedures.

- Performs audits to check quality system adequacy and function operability with respect to applicable standards and procedures.

- Issues the final document (Qualification Report) about quality activities.

Computing Management Function. Computing Management Function is responsible for hardware and support devices employed in production.

Support Function. Support Function is composed of the following subfunctions:

- **Tool and Development Environment Subfunction**, responsible of development tools and methods management, qualification tools management and standardization of selection and use of tools and methods.

- **Updates and Releases Subfunction (SGMR)**, responsible of product management, configuration control and update control.

Thus we own a whole path from high-level description to the low-level one for the Software Anomaly Management activity and some spots at variable levels of detail on other (not all) aspects of Italtel process.

In the sequel we describe the basic issues of Italtel Quality Manual, with special attention deserved to concepts, definitions and acronyms whose understanding is mandatory to read the following chapters.

3.3 Roles description and management

Roles and tasks within organizational structure are so defined:

Software Production Function. Software Production Function is responsible for system/project features in terms of:

- Architectural choices, allocating services to functional blocks and defining interfaces among them, identifying computational structures and consequently the partitioning between hardware and software functionality;
- Verification of integration phase and system tests, to perform after execution and verification of hw/hw and sw/sw integration phases. With *integration* we mean the functionality verification activity of software installed on target hardware involving on all computing systems composing it.

Software Project Function. Software Project Function is responsible for software design and production, starting from requirements specification and articulating the following phases:

- functional specification;
- architectural definition;
- software product implementation;
- definition, implementation and execution of qualification tests.

Depending on extension and complexity of the project may be composed of:

- Software Quality (`A7060-W22-A805-*-35`)[1] The higher-level description of Italtel Software Quality System.
- Software Product Anomalies and Updates (`A7060-W22-A820-*-35`). General description of concepts involved in software anomaly management.
- Software Product Anomalies Management (`A7060-W22-A821-*-35`). A more detailed description of the process followed during anomaly solving.
- Quality System Audit (`A7060-W22-A829-*-35`). High-level description of audit task.
- Operating Instruction for Software Product Configuration Management (`A7060-W23-A880-*-35`). A detailed description on how Configuration Management is performed.
- Operating Instruction for Software Product Anomalies and Updates Management (`A7060-W23-A881-*-35`). A detailed description on how Anomaly Management is performed.
- Operating Instruction for Software Product Review Activity (`A7060-W23-A891-*-35`). A detailed description on how Review Activity is performed.
- Test Launch Report Document Standard (`A7060-W23-A844-*-35`). This was not originally provided but later requested because needed for modeling.
- PAPICS tool: manual for anomaly, configuration and updates management. (`A7060-W23-A883-*-35`).
  Contains the finite state machines associated with the different kinds of configuration items.

[1]This number is the documentation identifier, used for organizational storing and retrieving. All cross-references provided within Quality Manual to reference sections related with the one currently read are given by means of this numbers.

this is achieved through the Quality Manual at various levels of prescription, starting from general high-level criteria and going down to operative procedures, whose nature is inherently low-level since they details how to perform given tasks. Here follows an extended list of Quality Manual contents:

- software life cycle
- technical activities needed for project and product assessment, and for the definition of corrective actions.
- responsibilities for Function activities
- software product identification with respect to its specification
- expected documentation list for develop and quality
- methods and tools used for configuration management
- methods and criteria for Verification and Validation and phases of utilization
- device management
- suppliers qualification and verification
- report and audit of Quality Function documenting and assessing production process and software product quality and Software Quality Control System efficiency.

3.2 Organization of provided documentation

Quality Manual is organized hierarchically: “root” chapters describe the software process in abstract terms by means of definitions and concepts, while “leaf” chapters contain the operative instructions and standards, needed to really enact the process in a developing context.

The Quality Manual extension is very large, so we do not have a complete copy; below are listed BUTD procedures provided by Italtel:

Chapter 3

An introduction to Italtel BUTD Software Quality Manual

The informations contained in this and in following chapters about Italtel BUTD Software Quality Manual can be used only in cases provided for by law and within the activity of modeling the Italtel BUTD software process at CEFRIEL.

Italtel Quality Manual is both a reference for our work, when we need to deeply understand the software process we are modeling, and the object of specification when we take it as a correct natural-language specification of the process. For this reason it constitutes a basic piece of the whole modelization effort.

3.1 Application domains and goals

Definition. BUTD Software Quality Manual defines *management criteria*, *project standards*, *production rules* and *operative procedures* adopted to satisfy the requirements of a Software Quality Control System in order to ensure the *quality* of Software Product and/or the Service stated in the contract.

Thus, the basic goal is to ensure the service requested and its quality;

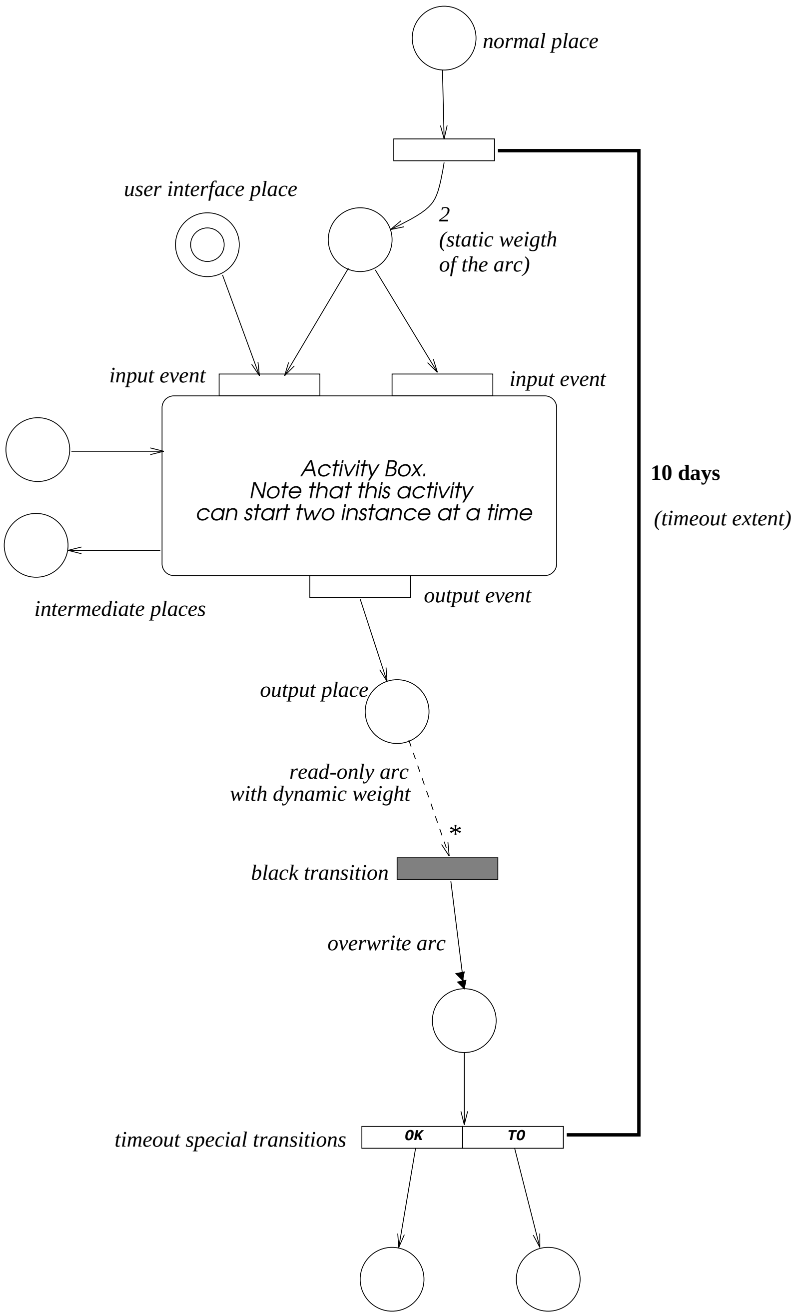


Figure 2.1: A dummy example to show SLANG syntax

to the activity.
Places internal to the implementation part, i.e. not belonging to the implementation part, are called *local places*.

Activities can invoke other activities from implementation part but *no recursive call is allowed.*

The way the activity construct is made up brings to mind the well-known high-level programming language concept of unit: activities can be regarded as the global procedure definition, having input and output places as procedure's formal parameters, intermediate places as global variables accessed during procedure execution and local places as variables local to the procedure body. Moreover, in the same manner we employ the *use* relationship to define a hierarchical (top-down) decomposition among units, we can use the *invocation* relationship for the same aim, defining a tree of activities, thus specifying logical precedence between them. Note that we can define only trees and not graphs, because of the absence of recursive calls between activities.

Advanced features of SLANG involve mechanisms to suspend and restart activities [BFG93a] for supporting the language reflective issues [BF93], needed to deal with changes involving the intimate structure of the process.

We used these features for modeling the suspension and cancellation of anomaly solving activities (so to model *process* features instead of *metaprocess* ones, nevertheless we will not describe here the mechanisms offered by SLANG, in order to lighten the language description: the complete description is provided in papers cited above.

that are executed concurrently, according to their logical precedence and, at the same time, scheduled to meet some global deadlines. Thus, the activity concept [BFG93b] is a central one in software process understanding, as pointed out by [DNR91] and [CFFS92]: SLANG offers the activity construct to support such a concept. An activity is defined as made by an *interface part* and an *implementation part*. The interface part is composed of:

- interface places;
- starting and ending events (transitions);
- weighted arcs connecting input-output places to starting-ending events;
- weighted arcs connecting intermediate places to activity box.

Interface places provide an activity the capability of interacting with the rest of the net, with three distinct set of places:

- *Input places*: are input places to transitions representing starting events for the activity.
- *Output places*: are output places to transitions representing ending events for the activity.
- *Intermediate places*: are places that may belong both to the preset or postset of any transition internal to the activity.

When an activity starts, it removes the enabling tuple from input places of the starting event. Note that input places of a starting event are a subset of input places of the whole activity. Thus, when an activity starts, it may remove tokens only from *some* of its input places. The activity can remove or insert tokens from/in an intermediate place at *any time* during its activation. When the activity finishes, the produced tokens are inserted in the output places of the ending event that caused the ending of the activity. Note once more that the output places of the ending event are a subset of output places of the activity.

The implementation part of the activity is a SLANG net describing how tokens are transformed between the starting and the ending events. This net accesses input and output places through starting and ending events, whereas it accesses intermediate places through any other transition internal

Notation	*Semantics*	*Time interval*
ASAP	strong	$[0, \infty]$
ASAP AFTER α	strong	$[\alpha, \infty]$
MAY BEFORE α	weak	$[0, \alpha]$
AFTER α *AND MAY BEFORE* β	weak	$[\alpha, \beta]$

Table 2.1: SLANG time notation.

means also that the black transition itself may "fire" many times with different input tuples, without waiting for each routine execution to complete. Black transition are graphically represented with a brown transition box.

Human Interaction. It is not possible to model the whole process, because this is not completely automatizable, but actually executed by humans. Human interaction is modeled via *user interface places*, that offer the dual way to black transitions to model SLANG interaction with non-SLANG world. User interface places are special places that may change their contents as a consequence of human intervention. User interface places are graphically represented with a double circle.

Timing issues. Every transition is associated with a time constraint [BFGG92] defining the time interval during which an enabled transition may or must fire, as shown in Table 2.1. As in ER nets, every token carries a *timestamp*, which contains the time at which the token has been created. Moreover, SLANG offers a *timeout* construct that allows the process specifier to define the maximum time interval between the firing of two transitions, namely the start-timeout and end-timeout transitions. The end-timeout transition actually specifies two actions, one representing successful completion within the timeout, and the other signaling that the timeout has not been met. A complete definition of the timeout construct is provided in [BFGG92]. As we will point out in Chapter 5, we have not been able to use the timeout construct within our model, due to the semantic lack currently associated to it.

Activities. A software process may be viewed as a set of related activities

transition preset) and is used to decide whether an input token tuple enables the transition (such an input tuple is called *enabling tuple*). The dynamic behavior of a transition is described by the *firing rule*. The firing rule states that when a transition fires, tokens satisfying the guard are removed from input places and the transition's action is executed. As a result of executing the action, an output tuple is inserted in the output places of the fired transition. Thus, *process evolution* is described by the firing of transitions and precedence between events and parallelism are described by net topology. Syntax for guards and actions is described in Appendix A.

Arcs. Weighted arcs are used to connect places to transitions and vice versa. The weight is used to specify the number of tokens fetched or filled in a place during as a consequence of a transition firing. It can be a statically defined number (the default is 1) or it may be dynamically computed, indicating the weight by a "*". In the latter case, the arc weight is known only at run-time and determined by the guard or the action of the connected transition. This is useful to model events requiring, for example, *all tokens* that verify a certain property.

Besides "normal" arcs, SLANG provides two other special kinds of arcs:

- *Read-only arcs* may connect a place to a transition, avoiding token transferring but enabling transition firing.

- *Overwrite arcs* may connect a transition to a place: when a token transferring has to occur, the associated semantic says that first the destination place must be emptied of all its tokens, then the tokens produced by the firing are inserted in the output place, thus producing an overall overwrite effect.

Graphical notation distinguishing arcs is shown in Fig. 2.1.

Tool Invocation. A software development process involves the activation of large variety of software tools. Tool invocation is modeled in SLANG by using the *black transition* construct. A black transition is a special transition associated with non-SLANG executable routines that can be performed *asynchronously*, i.e. with non-negligible and non-predictable execution time and allowing the firing of other transition while running its action. This

Chapter 2

An introduction to SLANG

The process modeling language we used to approach the case study modeling problem is SLANG, the basic component of the process-centered environment SPADE. This language further extends a Petri Nets extensions well known in literature, namely ER nets [GMMP91], in a way that allows a formal translation from SLANG semantics to ER nets semantics. Thus, SLANG really inherits all ER nets properties and formalization and in particular SLANG basic objects are ER nets objects, i.e. tokens, places, transitions and so on, though slightly modified. Here we list SLANG basic objects, then describing high-level constructs enriching SLANG expressive power.

Tokens. Tokens represent the objects involved in the process (documents, files and so on) that flow through the net. Tokens are typed in a object-oriented fashion, with attributes defining the object features and operations that can access it in a unique way. Syntax is described in Appendix A. Multiple inheritance is allowed, permitting the creation of a type hierarchy based on the "is-a" relationship. Every token must be of a type inheriting from the built-in type *ModelType*.

Places. A place is a token repository that contains only tokens *of its predeclared type*. So the *process state* is described by the net marking.

Transitions. Transitions represent *events* whose occurrence takes a negligible amount of time. The occurrence of an event corresponds to the firing of a transition. Each transition is associated with a *guard* and an *action*. The transition guard is a predicate on input tokens (tokens belonging to the

Part I

Backgrounds: the language and the process

2. *An assessment of SLANG as a language for process modeling.* This possibly involves the raising of proposals about modification of existing construct or about introduction of new constructs to better suit the needs of process modeling.

3. *Experimentation in process model elicitation.*

Quality Manual is written for the mostly part in natural language, thus intrinsically ambiguous: the modeling work, conducted by a person watching the problem from a different point of view, offers the possibility of testing the consistency of the whole document. Consequently, an expected *side-effect* of our work is the discovery of a set of ambiguities and inconsistencies in Italtel BUTD Software Quality Manual description.

This document reports the state of the work done during the Fifth Edition of the Master in Information Technology at CEFRIEL, and it is organized as follows: Part I describes both SLANG features (in Chapter 2) and Italtel Quality Manual features, thus giving the reader the background needed to understand the Part II, were the real work is explained. The discussion of Quality Manual is presented in a general fashion in Chapter 3, while more detailed informations about the portion of process modeled, together with a report of problems encountered are given in Chapter 4.

Part III collects the list of problems and "impression of use" for the language, together with the conclusions and future work. In Appendix A it is explained the "operative" syntax used for SLANG modeling.

- *Reusing.* Processes do not need to be defined from scratch every time.

The need for a formal and possibly executable description of software process has generated a lot of proposal [ABGM92] starting from different approaches, including Statecharts [Kel89], knowledge-based systems [BK91], [MS90], [HJP+90] and Petri nets [GJ92], [BFG93b].

Most of the available scientific literature about these formal approaches illustrates their basic issues, possibly using toy examples but almost never showing a tougher application of the proposed approach to a real and actually enacted (or enactable) software process.

Moreover, there are also approaches to description of software metaprocess , i.e. to the process of modeling (eliciting knowledge about) software processes, but not even these show detailed experiences of real assessment in the field of their methods.

Starting from these consideration, we developed a complete specification of a real process using SLANG, the software process modeling language of the process-centered environment SPADE (Software Process Analysis, Design and Enactment) currently developed at CEFRIEL at Politecnico di Milano. The chosen real case involves the specification of the software process enacted at Italtel and particularly in the Business Unit Defense Telecommunications(BUTD): this process is partially described in the BUTD Quality Manual most by means of natural language, sometimes using finite state machines to specify control flow.

We have adopted an incremental approach to modeling the Italtel process. We have identified a critical activity namely the *software anomaly management*.

Then we addressed configuration management issues, that are deeply bounded with conventional software process issues and spread throughout the software process.

Finally, we considered also review management and human resource allocation management, though these acitivity have not been fully specified in our model.

The expected results of the modeling effort include:

1. *A complete and consistent SLANG specification of a basic part of Italtel Quality Manual.*

Current work on software process modeling may be classified in two broad areas:

- the first focuses on *managerial issues* and aims at improving current industrial practices by focusing on organizational issues;
- the second addresses *technical issues* involved in specifying and mechanically supporting software processes.

Our work may be fit in this second area, addressing specification issues from a formal point of view.

The basic idea of process modeling is not new, in the sense that some software factories already made their own *written* description of their process; what is new is the *formal* approach, since these description are written for the much part in natural language and consequently inherently ambiguous.

Some goals are expected to be reached by a formal process description:

- *Communication improvement.* Modern software production is a multi-person cooperative activity.
- *Estimating and planning.* These are activities typical in every industrial process, to monitor and possibly adapt the way the process is enacted.
- *Enaction.* If the notation has an operational semantics it can be executed, simulating and/or driving the real process.
- *Verifying.* Formal notation allow us to verify the specification in many ways, including verifying of formal properties, simulation and on the field verification during enaction.
- *Managing and re-planning.* Software production is a high evolutionary activity.
- *Measuring.* Measures are a fundamental step in improving process as stated in [Hum89].
- *Configuring.* The process manager must be capable of changing the tools and techniques used in the process possibly without changing the deep structure of the process itself.

Chapter 1

Introduction

Software has distinguishing features, that make it difficult to handle:

- it is *malleable*, i.e. the product (not the design) can be modified very easily;
- it is not material;
- the cost for reproduction of the product is negligible while the manufacturing cost may be very high.

Traditional industrial products are produced following a set of pre-established methods and procedures, integrated in a production process, whose goal is the production of quality products in a reliable, predictable and efficient manner. Since software products permeate the whole modern life extending its application to critical systems, like avionics or biomedical instruments, quality requirements are more and more strict producing the need for a shift from the traditional way of software production, sometimes called the art of software production, highly dependent on individual skills, to an industrial way, process-driven and with some level of automatization ensuring appropriate quality levels.

This shift is typical of every expanding technology, but if the stated software properties make hard to deal with software products, the variety of existing application contexts and development techniques for software make impossible to define a universal suitable process: in other words, we can say with F.P.Brooks [Bro87] that *there is no silver bullet* for software process.

A Extended BNF syntax of SLANG textual part **199**
A.1 Guards & Actions 199
A.2 Type Declarations 200

4.3 Problems with Italtel Quality Manual 42
4.3.1 Incompleteness aspects 43
4.3.2 Ambiguities aspects 44
4.3.3 Inconsistency aspects 44
4.4 Issues not included in current version of Quality Manual . . . 45
4.4.1 New FSMs for configuration items and forms control . 45
4.4.2 Human resources management issues 50

II The SLANG specification of Italtel BUTD software process 55

4.5 Organization of specification 56
4.6 Type declarations 57
4.7 Tools description 77
4.8 Software Configuration Management 80
4.9 Software Anomaly Management 115
4.10 Software Review Management 168
4.11 Human Resource Allocation Management 175

III Summing up 182

5 Problems and assessment of SLANG 183

5.1 SLANG as a kernel for higher-level specifications 183
5.1.1 The multiple alternative problem 184
5.1.2 The human resource representation problem 185
5.1.3 Net modularization 186
5.1.4 The token identity problem 187
5.2 Temporal semantics 187
5.3 A tougher semantic problem 193
5.4 Considerations and experiences 194

6 Future work and conclusions 196

7 Acknowledgments 198

Contents

1 Introduction **4**

I Backgrounds: the language and the process **8**

2 An introduction to SLANG **9**

3 An introduction to Italtel BUTD Software Quality Manual **15**

3.1 Application domains and goals 15
3.2 Organization of provided documentation 16
3.3 Roles description and management 18
3.4 Software documentation 20
3.4.1 Development documentation subject to verification & validation 20
3.4.2 Qualification documentation subject to verification & validation 24
3.5 Planning 24
3.6 Product life-cycle 25
3.6.1 Development activity 26
3.6.2 Test Preparation activity 26
3.6.3 Qualification activity 26
3.6.4 Review activity 27
3.7 Relations with other standards 28

4 The software process actually modeled **29**

4.1 Software Anomalies Management 29
4.2 Software Configuration Management 39

Abstract

The need for a formal description of the software production process generates a need for formal methods and languages to help the process expert in developing such description. Several languages have been proposed but there is very little experience about the application of these languages to real processes. We used SLANG, the SPADE environment modeling language, to model the real software process described into the Italtel Business Unit Defense Telecommunications Software Quality Manual. The ultimate goals of this effort were the formalization of the process, the assessment of the suitability of SLANG as process modeling language, and the maturation of experience in dealing with software processes of a real complexity. In addition, the modeling task gave as a side-effect a set of problems and improvements in Quality Manual description.
The full model is presented, together with problem discussion and SLANG assessment.

Modeling a *Real* Software Process with SLANG

Gian Pietro Picco
CEFRIEL
Via Emanueli 15, 20126 Milano, Italy
Tel. +39-2-66100083, Fax. +39-2-66100448
e-mail: se4@bach.cefriel.it

June 25, 1993