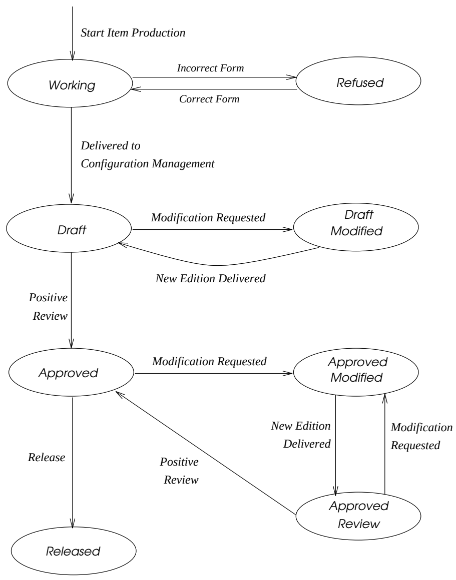


Figura 8.7: L'automa per la gestione dei documenti soggetti a verifica e validazione.

- **Draft Modified.** Il documento entra in questo stato a seguito dell'emissione di RIMO su di esso. Al completamento delle modifiche, dopo la rioconsegna riprenderà lo stato di 'Draft'.
- **ApprovedModified.** Questo stato viene utilizzato ogniqualvolta vengono emesse delle Richieste di Modifica su un documento che ha già avuto almeno una volta lo stato di 'Approved'.
- **ApprovedReview.** Viene utilizzato per indicare che un documento precedentemente in stato di 'Approved' è attualmente sotto revisione a causa di RIMO emesse su di esso e già implementate.

Automa per il Rapporto di Lancio dei Test (TLR)

Questo automa, mostrato in Fig. 8.8, non differisce di molto da quello relativo ai documenti soggetti a verifica e convalida: l'unica differenza consiste nel fatto che un TLR consegnato alla SGMR e approvato non può ricevere delle RIMO: poiché essi sono collegati agli elementi di configurazione, una RIMO sarebbe originata solo da una modifica all'elemento associato al TLR. In tali

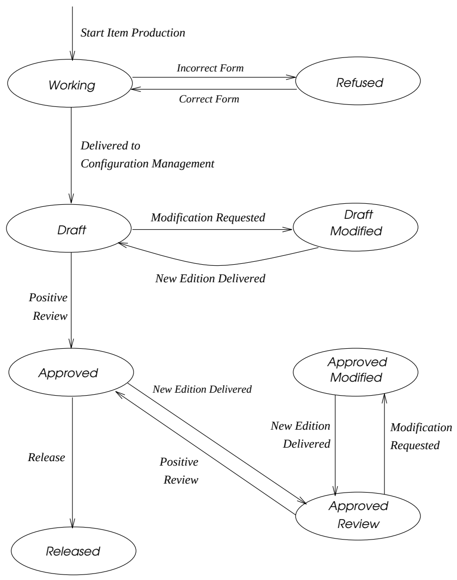


Figura 8.8: L'automa per la gestione dei documenti TLR.

casi, la procedura seguita è quella di rieseguire i test, ottenendo una nuova versione da sottoporre a revisione.

Automa per il codice sorgente (SCU)

Questo è l'automa più complesso tra quelli relativi agli elementi di configurazione; come si può vedere dalla Fig. 8.9, rispetto all'automa che descrive il TLR, viene aggiunto un solo stato, **Tested**, che viene utilizzato solo dopo che il codice sorgente è stato non solo revisionato ma anche testato ed inoltre la revisione sul suo TLR è stata positiva. In caso contrario, vengono emesse RIMO sul codice, provocando una transizione di stato in 'Approved Modified'.

8.2 Aspetti di gestione delle risorse umane

Una modellazione sufficientemente accurata del processo software utilizzato in Italtel BUTD non poteva prescindere dagli aspetti di gestione delle risorse umane, i quali tuttavia non erano presenti nella sezione di Manuale di Qualità a noi fornita. Attraverso colloqui con l'esperto del processo è stato possibile

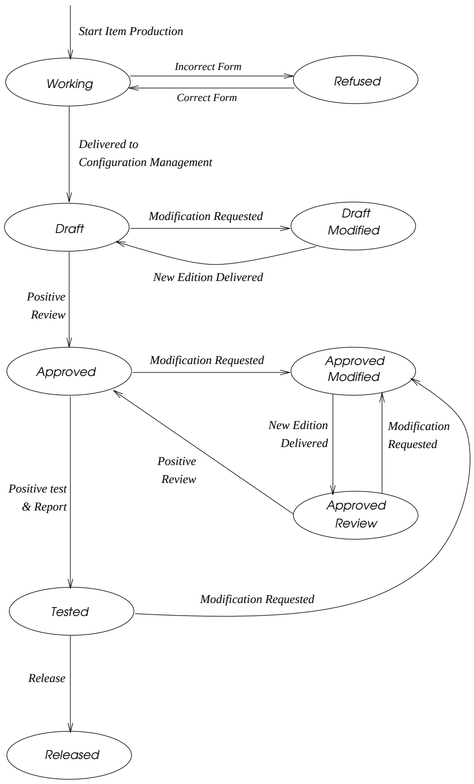


Figura 8.9: L'automa per la gestione dei documenti SCU.

stabilire alcuni punti importanti ai fini della modellazione, sia in termini di conoscenza del processo che di assunzioni da imporre:

- Ogni attore del processo software possiede una lista di compiti da eseguire. Inoltre, ha la possibilità di eseguire "concorrentemente" i vari compiti, nella maniera che gli è più congeniale, purché i compiti medesimi vengano portati a termine nelle scadenze prefissate.

- Un attore del processo può ricoprire, in tempi diversi, ruoli diversi. Ad esempio, uno sviluppatore può essere anche membro della qualificazione per un altra parte di progetto, o fare parte di un Team di Review.

- Un compito (o *task*) è definito mediante tre variabili:

  - **Durata.** È la durata stimata del compito. Viene definita in genere da chi assegna il compito, oppure dall'attore che deve eseguirlo.
  - **Scadenza.** È la data entro la quale il compito deve essere terminato. Viene fissata da chi assegna il legame tra compito e persona che deve eseguirlo.
  - **Priorità.** Non è necessaria per ogni compito: risulta indispensabile solo per risolvere conflitti di allocazione tra tasks. La priorità del task, se necessaria per risolvere conflitti di questo tipo, viene stabilita dall'esecutore del task in accordo con il capo progetto.

- Per quanto riguarda la gestione delle risorse umane in relazione con la porzione di processo presa in esame, è il CCB che determina gli assegnamenti dei task.

- È possibile, in prima approssimazione, identificare due metodi di allocazione delle risorse umane:

  - Una gestione centralizzata, che tiene conto di tutti i vincoli facendo un'allocazione globale, da utilizzarsi all'inizio del progetto.
  - Una gestione distribuita, che demanda la gestione dei task e di eventuali conflitti agli esecutori, da utilizzarsi durante la fase centrale del progetto.

- È possibile definire un piccolo algoritmo su cui basarsi per definire la gestione dell'allocazione delle risorse umane, da utilizzarsi nella gestione delle anomalie:

    1. L'esecutore valuta la durata del task in relazione a quelli presenti sulla sua "agenda", verificando che l'introduzione del nuovo task non provochi conflitti tra le scadenze.
    2. Se non vi sono conflitti l'assegnamento diventa effettivo,
    3. altrimenti
        - stabilisce delle priorità con il suo diretto superiore;
        - tenta una ripianificazione locale assumendo di ritardare i task a priorità minore.
            * Se la ripianificazione ha avuto successo, l'assegnamento diventa effettivo, ma devono essere emesse richieste di ripianificazione per i task ritardati.
            * Se la ripianificazione non ha avuto successo, emette una richiesta di ripianificazione globale

Si noti che questo nucleo di informazioni sulla gestione delle risorse umane verrà esteso al fine di formalizzare questi aspetti, i quali si trovano nel Manuale di Qualità in maniera dispersa, incompleta e frammentaria.

Parte IV

Descrizione e risultati della modellazione

Capitolo 9

Descrizione del modello

Questo capitolo descrive i concetti, i meccanismi e le peculiarità del modello SLANG relativo al processo software adottato in Italtel BUTD. Esso dunque non costituisce il modello completo, il quale è invece presentato nella Appendice A.

9.1 Struttura del modello e scelte progettuali

Dal momento che la porzione di processo considerata pur non coprendo l'intero processo risulta essere comunque di dimensioni e complessità considerevoli, il modello è strutturato in modo da sfruttare i costrutti per la progettazione *in-the-large* [BFG93b] offerti dal linguaggio, in modo da impiegare il più possibile i concetti di modularizzazione, information hiding e incapsulazione.

Seguendo questi principi, le attività sono state decomposte in sotto-attività, iterando tale procedimento fino a raggiungere un livello di complessità tale da poter essere descritto da una rete SLANG che non necessiti di impiegare ulteriormente il costrutto di invocazione di attività. Utilizzando l'albero statico di una attività, definito nella Sez. 3.2, è possibile fornire una visione d'insieme della gerarchia di sotto-attività definita per quella in esame, in maniera analoga alla gerarchia dei moduli di un linguaggio di programmazione come Ada o Modula-2.

Poiché la parte di processo considerata è costituita di fatto da quattro diverse sezioni del Manuale di Qualità [1] identificabili come attività concorrenti e cooperanti, è risulta abbastanza intuitivo pensare queste quattro parti come altrettante attività, una volta definiti gli ingressi e le uscite previsti per tali attività, le risorse utilizzate e le interazioni permesse:

- **ManageSoftwareAnomalies.** Modella le attività di gestione delle anomalie software.
- **ManageConfigurationArchive.** Definisce il controllo configurazione relativo a tutto il processo software, quindi non soltanto relativo alla gestione delle anomalie.
- **ManageSoftwareReview.** Corrisponde alla gestione della verifica e validazione del software. È implementata soltanto parzialmente.
- **ManageHumanResourcesAllocation.** Di fatto costituisce una prima formalizzazione della gestione delle risorse umane, assente nel Manuale di Qualità.

Va puntualizzato però che, sebbene nella modellazione finale questo è l'approccio seguito, l'identificazione delle attività da modellare come tali ha seguito uno sviluppo incrementale durante la stesura del modello. Ad esempio, nella prima stesura la gestione della configurazione non era rappresentata da un'attività di alto livello come nel modello attuale, ma all'interno delle attività contenute in ManageSoftwareAnomalies venivano invocati direttamente gli eventi necessari di competenza della Sottofunzione Gestione Modifiche e Rilasci. In tal modo, la modellazione mancava di uniformità e chiarezza e possedeva inoltre una notevole quantità di ridondanze: l'operazione di controllo formale del documento, ad esempio, veniva invocata un gran numero di volte nel modello, replicando ogni volta la relativa sottorete. Il costrutto di attività in questo caso ha consentito, attraverso la modularizzazione e l'incapsulazione, di evitare replicazioni e rendere più omogenea la specifica; perdipiù, la semantica associata al costrutto di attività è coerente con la semantica così associata all'attività di gestione della configurazione,

[1] In realtà le sezioni sono soltanto tre, in quanto per la gestione dell'allocazione delle risorse umane non vi è nessuna precisa istruzione operativa o procedura. Queste verranno probabilmente inserite in seguito al tentativo di formalizzazione prodotto da questo lavoro.

che viene vista come un fornitore di servizi (emessi attraverso gli eventi di uscita) in risposta alle richieste (immesse attraverso gli eventi di ingresso), in maniera congruente a quanto viene fatto attualmente in Italtel BUTD. Strutturando la specifica in questa maniera, inoltre, si sono sfruttati i vantaggi notevoli dell'information hiding, cioè della possibilità di mantenere l'interfaccia dell'attività distinta dalla sua implementazione sottostante: ogniqualvolta è stato necessario cambiare la maniera in cui erano state definite le operazioni del controllo configurazione è bastato cambiare l'implementazione della sottoattività relativa, senza la necessità di modificare puntualmente ogni chiamata in ManageSoftwareAnomalies.

Analoghe osservazioni valgono per ManageSoftwareReview, la quale era precedentemente una delle sotto-attività della gestione delle anomalie.

Si noti, a questo punto, la mancanza della definizione di un'attività radice all'interno del modello, o comunque di un'attività di livello superiore a quelle create per la definizione delle parti di processo prese in esame: la ragione di ciò è la mancanza di informazione dettagliata sul processo globale tale da permettere di costruire una struttura sufficientemente coerente e significativa che abbracciasse le attività descritte, mancanza causata peraltro dalla decisione di limitare la parte di processo da prendere in esame.

D'altro canto, l'assenza di tale attività non invalida il modello sviluppato, né limita il suo futuro utilizzo in quanto, come già segnalato, l'interfaccia di ciascuna attività è correttamente definita e può essere facilmente collegata con una futura specifica che tenga conto anche di tali aspetti. Guardando le cose da un diverso punto di vista, anche in questo caso si sono utilizzati proficuamente i costrutti di modularità ed information hiding, costruendo di fatto dei moduli correttamente definiti in termini di interfacce, da collegare in futuro con il "programma di processo" principale.

Le Figg. 9.1, 9.2, 9.3 mostrano gli alberi statici per le attività di gestione delle anomalie, gestione del controllo configurazione e gestione della revisione rispettivamente: la gestione dell'allocazione delle risorse umane è invece costituita da un'attività descritta da un'implementazione che non contiene nessuna invocazione di attività. Si noti, a questo proposito, che in un'ipotetico albero statico contenente le quattro attività componenti il modello, tali attività si troverebbero tutte al medesimo livello, come mostrato in Fig. 9.4.

Anche la descrizione dei tipi dei dati utilizzati nel processo ha utilizzato pesantemente i costrutti offerti dal linguaggio: in particolare, è stata

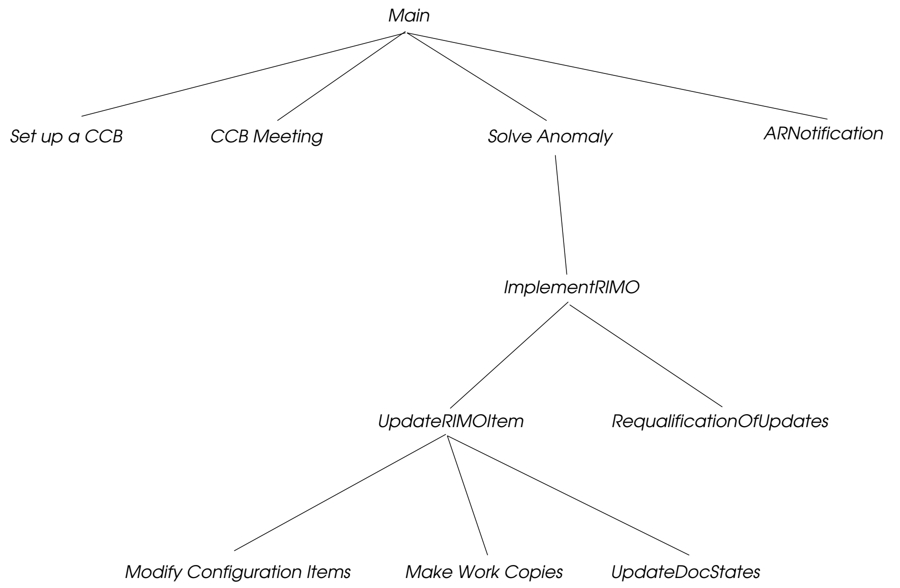


Figura 9.1: L'albero statico dell'attività ManageSoftwareAnomalies.

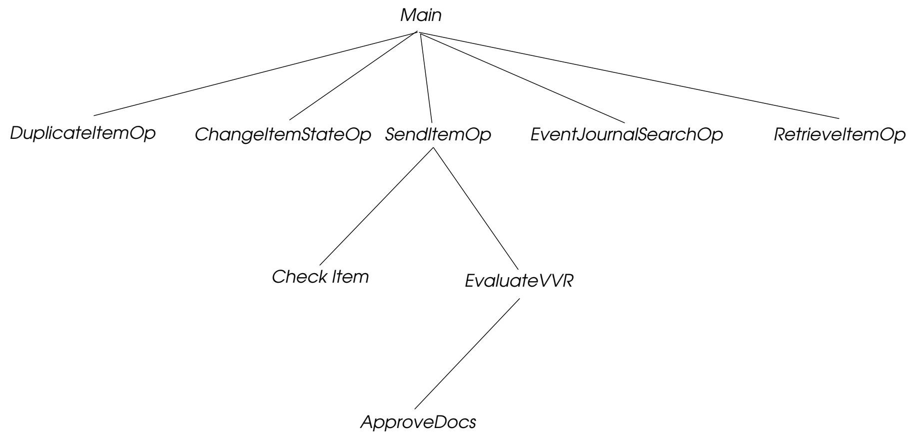


Figura 9.2: L'albero statico dell'attività ManageConfigurationArchive.

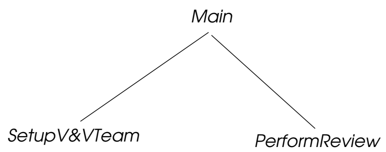


Figura 9.3: L'albero statico dell'attività ManageSoftwareReview.

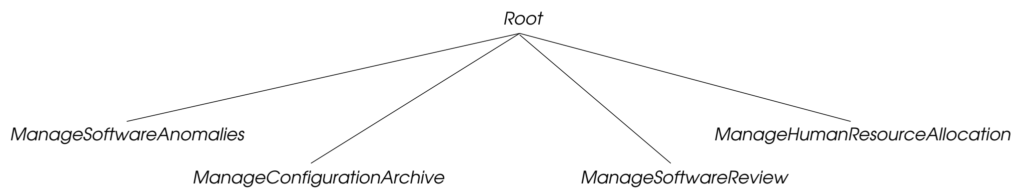


Figura 9.4: Un'ipotetico albero statico che mostra le quattro attività del modello al medesimo livello.

costruita una gerarchia basata sulla nozione di ereditarietà per descrivere i documenti, il codice ed i *form* [2] utilizzati all'interno del processo, nonché una seconda gerarchia per descrivere i tools impiegati nello sviluppo del software e nella gestione del processo.

Il procedimento usato per costruire la prima gerarchia è stato il seguente:

- si sono raccolte le descrizioni in nostro possesso relative ai documenti necessari alla modellazione, stilando per ciascuno di essi in una lista degli attributi (ed eventualmente metodi) necessari a descriverlo;
- si sono raggruppati gli attributi comuni a tutti i tipi in un unico insieme, che è stato utilizzato per definire la radice della gerarchia.;
- si è cercato di raggruppare ulteriormente gli attributi rimanenti mediante criteri di similitudine o relazioni logiche tra essi, ottenendo altri insiemi di attributi;
- da questi ultimi insiemi si sono tratti i sottotipi della radice: in altre parole, l'unione degli attributi di questi tipi con l'unione dei tipi della radice fornisce l'insieme di attributi caratteristici del documento originario;
- il procedimento è stato iterato fino ad esaurire gli attributi disponibili, ottenendo a questo punto [3] la gerarchia mostrata in Fig. 9.5.

[2] Usiamo qui il termine *form* al posto del più corretto *modulo* per evitare confusioni tra i moduli utilizzati per il controllo all'interno del processo ed i moduli del codice sorgente.

[3] In realtà, come è facile intuire, non si è ottenuta subito la versione finale della gerarchia, che è stata modificata alcune volte per meglio adattarla all'insieme dei documenti che sono stati via via ritenuti rilevanti ai fini della modellazione.

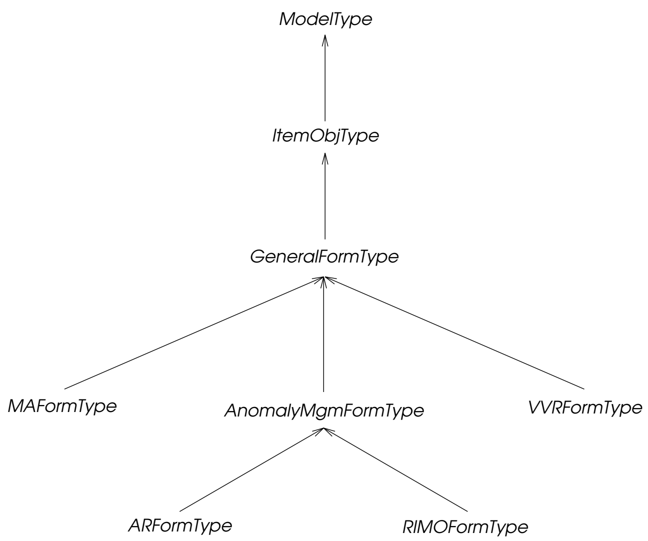


Figura 9.5: La gerarchia di tipi basata sulla relazione di ereditarietà che descrive gli elementi di configurazione necessari alla gestione delle anomalie.

Da tale figura si vede che tutto l'albero dei tipi così definiti eredita dal tipo predefinito `ModelType`, coerentemente a quanto descritto nel Cap. 3; il documento più generale è modellato dal tipo `ItemObjType` il quale contiene, per costruzione, l'insieme minimo di caratteristiche necessarie a modellare un elemento in controllo configurazione.

Nella versione attuale del modello questo tipo di dato descrive praticamente tutti gli elementi in controllo configurazione, esclusi i *form*. Questa approssimazione non riduce di molto la validità del modello, in quanto le azioni da intraprendere sugli elementi di configurazione durante la gestione delle anomalie o il controllo configurazione non vengono mai descritte ad un livello di dettaglio tale da richiedere una modellazione più particolareggiata. Il processo impone però che sia possibile identificare la natura dell'elemento, cioè se si tratti ad esempio di un SCU oppure di un DD: questo compito è assolto dal campo `Kind` il quale, durante l'esecuzione del processo, contiene sempre un identificatore associato alla tipologia di documento rappresentata dal token.

Oltre a questa informazione, i token di tipo `ItemObjType` (o di un suo sottotipo) contengono anche informazioni circa lo stato dell'elemento (come definito dall'automa associato), il suo identificatore all'interno del controllo configurazione, le informazioni relative al testing (se necessario), l'oggetto fisico vero e proprio ed i metodi per effettuare il controllo di correttezza formale e costruire le *mailing list* per lo specifico elemento.

Su `ItemObjType` è innestata la gerarchia che definisce i vari tipi di *form* utilizzati nella parte di processo considerato. Tale gerarchia segue i criteri di costruzione descritti precedentemente, pertanto la radice `GeneralFormType` contiene tutte le caratteristiche ritenute basilari per un form, quali ad esempio la data della consegna, l'istituzione che ne gestisce l'emissione e la descrizione testuale associata (ad esempio la descrizione dell'anomalia nel caso di un'AR o la descrizione della modifica ad un elemento nel caso di una RIMO).

Da tale tipo derivano nell'ordine:

- `MAFormType`: è il cosiddetto Modulo di Accompagnamento, utilizzato per la consegna di un nuovo elemento alla Sottofunzione Gestione Modifiche e Rilasci.

- `AnomalyMgmType`: costituisce la radice per i form peculiari della gestione delle anomalie, vale a dire il Rapporto di Anomalia e la Richiesta di Modifica. È definito qui un campo `Instance`, contenente la copia attiva dell'attività che sta portando avanti la soluzione dell'anomalia o della modifica associata al token ed il cui significato all'interno del modello sarà chiarito più avanti.

- `VVRFormType`: definisce le caratteristiche del Rapporto di Verifica e Validazione, quali l'elemento revisionato, l'esito della revisione, le RIMO da considerarsi risolte e quelle da considerarsi irrisolte, la composizione del Team di Revisione all'atto della riunione.

Infine, sono definiti:

- `ARFormType`: definisce il Rapporto di Anomalia, fornendo tra l'altro riferimenti ai TLR e TLVVR associati al Rapporto, agli elementi di configurazione coinvolti nella soluzione ed a livelli di gravità dell'anomalia opportunamente definiti mediante tipi enumerativi.

- `RIMOFormType`: definisce la Richiesta di Modifica e contiene, tra le altre informazioni, i riferimenti agli AR interessati dalla modifica, la scadenza per la realizzazione della medesima, l'esecutore ed i requisiti di riqualificazione.

La gerarchia di tipi descritti finora non è l'unica utilizzata all'interno del modello, sebbene sia l'unica coinvolta nella modellazione dei dati del processo: in particolare sono definiti dei tipi per gestire il corretto interfacciamento di ManageConfigurationArchive verso le altre attività, i quali verranno descritti in 9.2.1 ed un certo numero di tipi ausiliari, dei quali si può trovare una descrizione nell'Appendice A.

9.2 Tecniche e meccanismi utilizzati

Questa sezione illustra le principali meccanismi di SLANG e tecniche di modellazione con tale linguaggio impiegate nello sviluppo del modello. Per ogni tecnica è mostrato un esempio all'interno del modello. Al contrario, la descrizione particolareggiata del comportamento di ogni singola attività del modello si trova nell'Appendice A.

9.2.1 Conflitti in accesso alle attività

Nel modello SLANG alcune delle attività possono essere descritte come fornitrici di servizi ad altre attività: è il caso ad esempio dell'attività che realizza la gestione della configurazione, ManageConfigurationArchive, la quale possiede un unico posto in ingresso contenente i token che descrivono la richiesta di un servizio e un unico posto in uscita all'interno del quale viene depositato il token generato da tale servizio.

La semantica associata all'invocazione di attività genera tuttavia dei problemi di ambiguità quando si tratta di reperire il token in uscita corrispondente ad un determinato token di richiesta del servizio.

Con riferimento alla Fig. 9.6, si supponga che le attività A e B possano comunicare attraverso i posti I1 e I2: in particolare A può richiedere determinati servizi a B tramite tali posti. Poiché le due attività si trovano al medesimo livello all'interno dell'albero statico delle attività, I1 ed I2 sono locali all'attività Father, ma condivisi al livello in cui si trovano A e B. Pertanto, una volta che A ha emesso una richiesta sottoforma di token in I1, e B ha assolto la richiesta emettendo un token in I2, A non ha più modo di distinguere il token emesso in seguito alla propria richiesta dai token eventualmente emessi in seguito alle richieste emesse da altre attività o da altre istanze aventi lo stessa definizione di A.

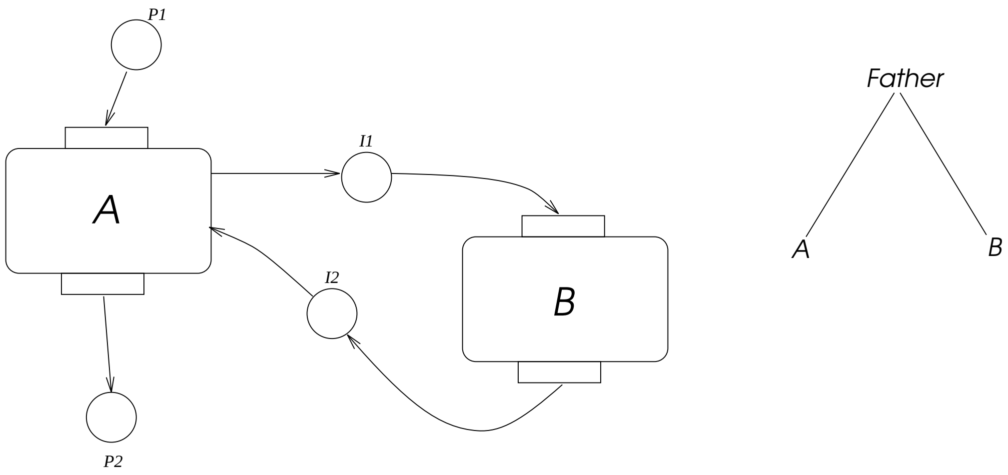


Figura 9.6: Conflitti in accesso tra attività poste allo stesso livello nell'albero statico delle attività.

Alternativamente, si osservi la Fig. 9.7: in questo caso B è un attività effettivamente invocata da A, poiché si trova ad un livello inferiore dell'albero statico, dunque i posti I1 ed I2 questa volta sono dedicati solo a ricevere i token in ingresso ed in uscita a B. Ciononostante, dalla struttura della rete si nota che può essere attivato più di un flusso [4] di esecuzione, attraverso la "retroazione" in T, ciascuno dei quali può generare più di una invocazione di B e di un diverso output in I2. A questo punto si comprende che se è importante associare ad ogni token immesso in I1 il corrispondente emesso in I2, A non ha nessun mezzo per ricostruire tale associazione, a meno di non fare determinate assunzioni sul contenuto dei token.

Il problema può infatti essere risolto mediante l'introduzione di informazione ausiliaria da associarsi ai token in ingresso ed uscita all'attività che fornisce i servizi. Un esempio della politica che si può adottare per risolvere il conflitto è mostrato in Fig. 9.8, con riferimento alla gestione della configurazione:

- L'azione associata alla transizione $t_{calling}$ genera un *identificatore di richiesta* unico, che viene associato al token in *I1* e contemporaneamente inserito nel posto *ReqID*.

[4] Il termine tecnico corretto è *thread*.

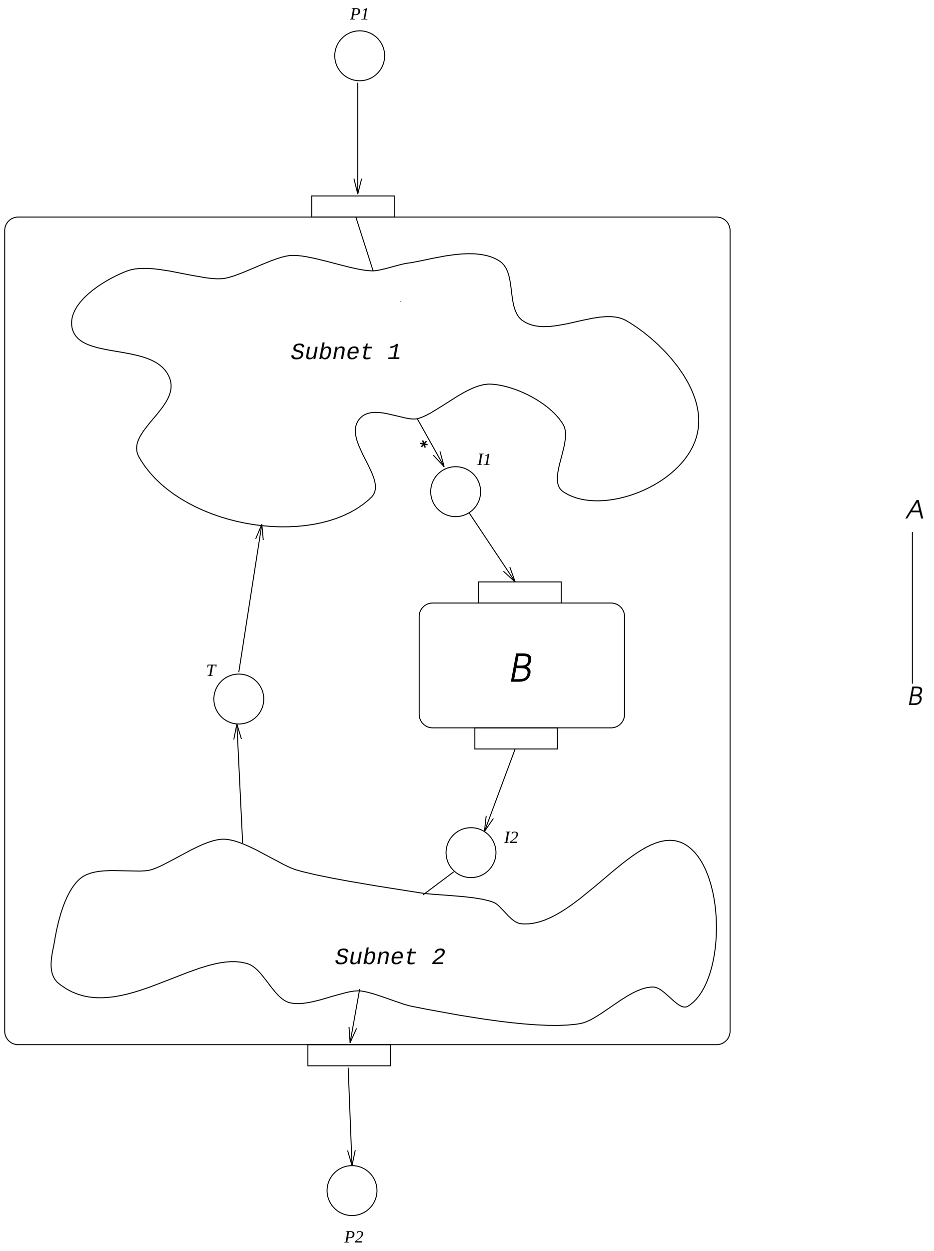


Figura 9.7: Conflitti in accesso tra attività a differenti livelli, dovuti a thread multipli nell'esecuzione della rete.

- L'attività B esegue le operazioni richieste tenendo traccia dell'identificatore durante l'esecuzione della sua implementazione.
- Al termine dell'invocazione di B, viene emesso un token in *I2*, il quale contiene a sua volta un campo con l'*identificatore di richiesta*.
- La selezione del token corretto avviene mediante il confronto degli identificatori dei token contenuti in *ReqID* e *I2*: se l'identificatore è costruito in maniera da essere unico, esiste soltanto una tupla che soddisfa la guardia della transizione $t_{checking}$.

Questo meccanismo viene utilizzato per la gestione del controllo configurazione. In altri casi può non essere necessario inserire un identificatore, ad esempio perché la natura dei dati in ingresso ed uscita dall'attività fornitrice di servizi è tale da garantire già tale informazione.

9.2.2 Il controllo configurazione

L'attività di gestione della configurazione del prodotto software è modellata come un attività che offre dei servizi (le operazioni coinvolte nel controllo configurazione) ad altre attività che necessitano di accedere all'Archivio di Configurazione.

La richiesta del servizio desiderato avviene mediante token complessi che specificano l'operazione da eseguire (cioè il servizio da fornire) ed eventuali parametri necessari all'esecuzione; analogamente, il token emesso in uscita presenta dei campi che contengono gli oggetti prodotti dai servizi richiesti insieme a valori convenzionali che permettono di determinare se l'attività è stata eseguita correttamente.

Tali tipi sono organizzati in una gerarchia, il cui unico scopo è mettere a fattor comune le informazioni necessarie sia all'interno del tipo che specifica la richiesta di servizi sia all'interno del tipo usato per descrivere i risultati.

Una visione sintetica del meccanismo utilizzato per comunicare con l'attività di gestione della configurazione è dato dalle Tabelle 9.1 e 9.2, le quali mostrano rispettivamente i campi da specificare per realizzare una richiesta al Controllo Configurazione e i campi riempiti con i valori opportuni a seguito dell'esecuzione di un servizio.

Tali tabelle propongono informazioni differenziate a seconda del tipo di servizio richiesto alla gestione della configurazione, selezionato tra i seguenti:

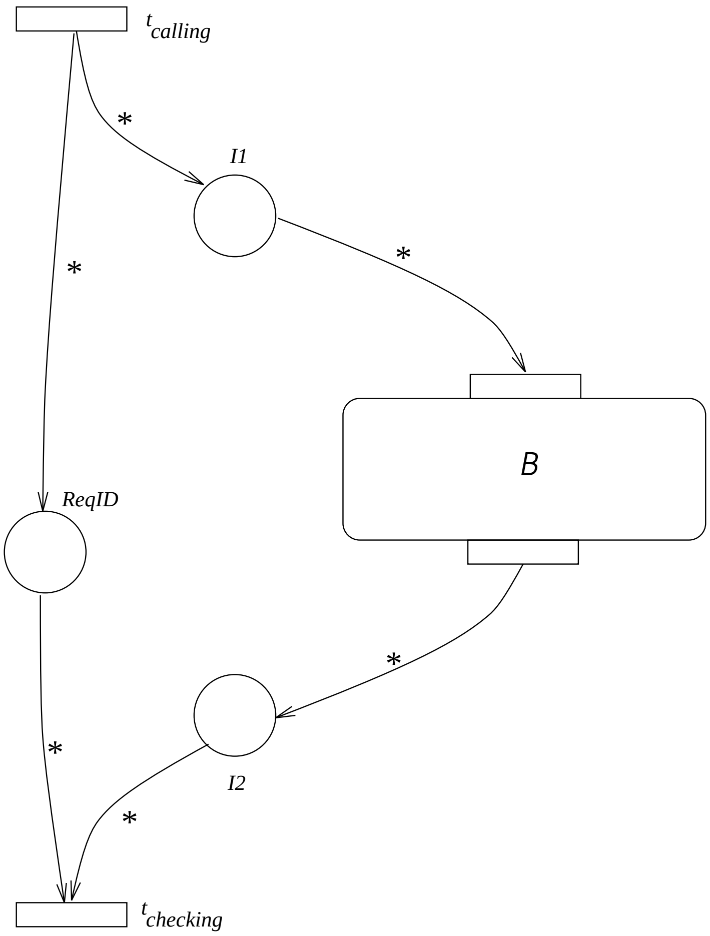


Figura 9.8: Una soluzione ai conflitti in accesso.

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

Tabella 9.1: Come riempire i campi di un token `CARequests` per ottenere servizi dal controllo configurazione.

- `Duplicate.` Consiste nella duplicazione di un elemento dall'Archivio di Configurazione ad una destinazione precisata all'interno dell'Archivio di Sviluppo.

- `Retrieve.` Invoca le procedure necessarie a recuperare un elemento di configurazione dall'Archivio.

- `Send.` Sottomette un elemento alla gestione della configurazione. L'attività relativa realizza prima un controllo formale: se questo mostra che l'elemento rispetta gli standard di consegna viene registrato all'interno dell'Archivio di Configurazione attivando, se necessario, ulteriori procedure quali ad esempio la convocazione di un Team di Verifica e Validazione.

- `ChangeState.` Effettua una transizione di stato sull'elemento considerato. Questo comporta una modifica dello stato associato all'elemento nonché la registrazione della transizione all'interno del Giornale degli Eventi.

- `EJSearch.` Dato un predicato, effettua una *query* sul Giornale degli Eventi per raccogliere tutte le informazioni che soddisfano il predicato. Un esempio di query: "gli identificatori di tutti gli elementi consegnati prima del 21 luglio e rifiutati perché formalmente non corretti".

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

Tabella 9.2: Come il controllo configurazione riempie i campi di un token `CAResultsType` in risposta ad una richiesta.

Si noti che coerentemente con quanto espresso nella Sez. 9.2.1 a proposito dei conflitti in accesso, l'attività ManageConfigurationArchive estrae l'identificatore della richiesta all'inizio della computazione e lo reimmette nel token di uscita al termine di essa.

Va fatto presente che attraverso questi meccanismi di gestione del controllo configurazione ed assumendo di avere effettivamente a disposizione i tool utilizzati nel modello (ipotesi peraltro verificata) non è più necessaria la Sottofunzione Gestione Modifiche e Rilasci, la quale è *completamente* automatizzata.

9.2.3 Cancellazione e sospensione di attività

Come è stato evidenziato all'interno del Cap. 8, il nostro lavoro ha contribuito ad estendere gli automi utilizzati in Italtel BUTD per la descrizione della gestione delle anomalie e delle RIMO, in maniera tale da poter gestire non solo l'esecuzione delle procedure connesse, ma anche la loro cancellazione, evento tra l'altro abbastanza comune all'interno della pratica industriale.

A questo punto è nato il problema di rappresentare all'interno del modello SLANG la gestione della sospensione e della cancellazione di attività già in corso di esecuzione, cioè di cui esistono copie attive all'interno del sistema.

La soluzione adottata si basa sul meccanismo descritto in [BFG93a] e riportato nella Sez. 3.4 per la sospensione delle copie attive al fine di fornire supporto al metaprocesso. Si noti quindi che l'utilizzo che si è fatto in questo lavoro dei meccanismi di sospensione di SLANG non è quello per cui tali meccanismi erano stati originariamente pensati: il meccanismo di sospensione, infatti, è stato concepito per permettere di rimuovere una copia attiva dal normale flusso di esecuzione, modificarla all'esterno del modello e reimmetterla (secondo un qualche tipo di politica) all'interno del flusso di esecuzione.

L'uso che ne è stato fatto all'interno della specifica oggetto di questo lavoro, invece, è strettamente collegato alla definizione del *processo* e non del *metaprocesso*: in altre parole, tali meccanismi sono strettamente necessari a rappresentare delle azioni, la sospensione e la cancellazione di attività attualmente in esecuzione, previste come necessarie ed intimamente collegate all'esecuzione del processo.

Le Fig. 9.9 e 9.10 mostrano l'applicazione dei meccanismi all'interno del modello.

Con riferimento alla Fig. 9.9, si può notare come l'attività che può essere sospesa o cancellata è in questo caso SolveAnomaly: in essa sono evidenziati i posti predefiniti (in realtà presenti per ogni attività del modello) che permettono la gestione delle operazioni di sospensione.

L'interruzione dell'attività viene eseguita a fronte di determinati cambiamenti di stato sull'anomalia decisi dall'attività CCBMeeting, la quale modella la riunione del CCB; l'esecuzione dell'interruzione avviene come descritto nella Sez. 3.4, inserendo l'identificatore della copia attiva relativa all'attività da sospendere nel posto `SuspendRequest` dell'attività.

A questo punto nasce il problema di associare ad ogni istanza dell'attività SolveAnomalies la rispettiva anomalia, cioè la rispettiva istanza del tipo `ARFormType`. Ciò è ottenuto attraverso il campo `Instance` inserito all'interno della radice della gerarchia dei moduli per la gestione delle anomalie, `AnomalyMgmType` il quale contiene la copia attiva associata al Rapporto di Anomalia modellato dall'istanza di `ARFormType` oppure associata alla Richiesta di Modifica modellata dall'istanza di `RIMOFormType`.

La necessità di sospendere o meno un'anomalia o una singola richiesta di modifica è determinata dal CCB durante una delle riunioni convocate per esaminare i Rapporti di Anomalia generati all'interno del processo. In particolare, con riferimento alla Fig. 9.9, si nota che gli eventi che interagiscono con i posti di SolveAnomaly deputati alla modifica dell'esecuzione di una copia attiva sono i seguenti:

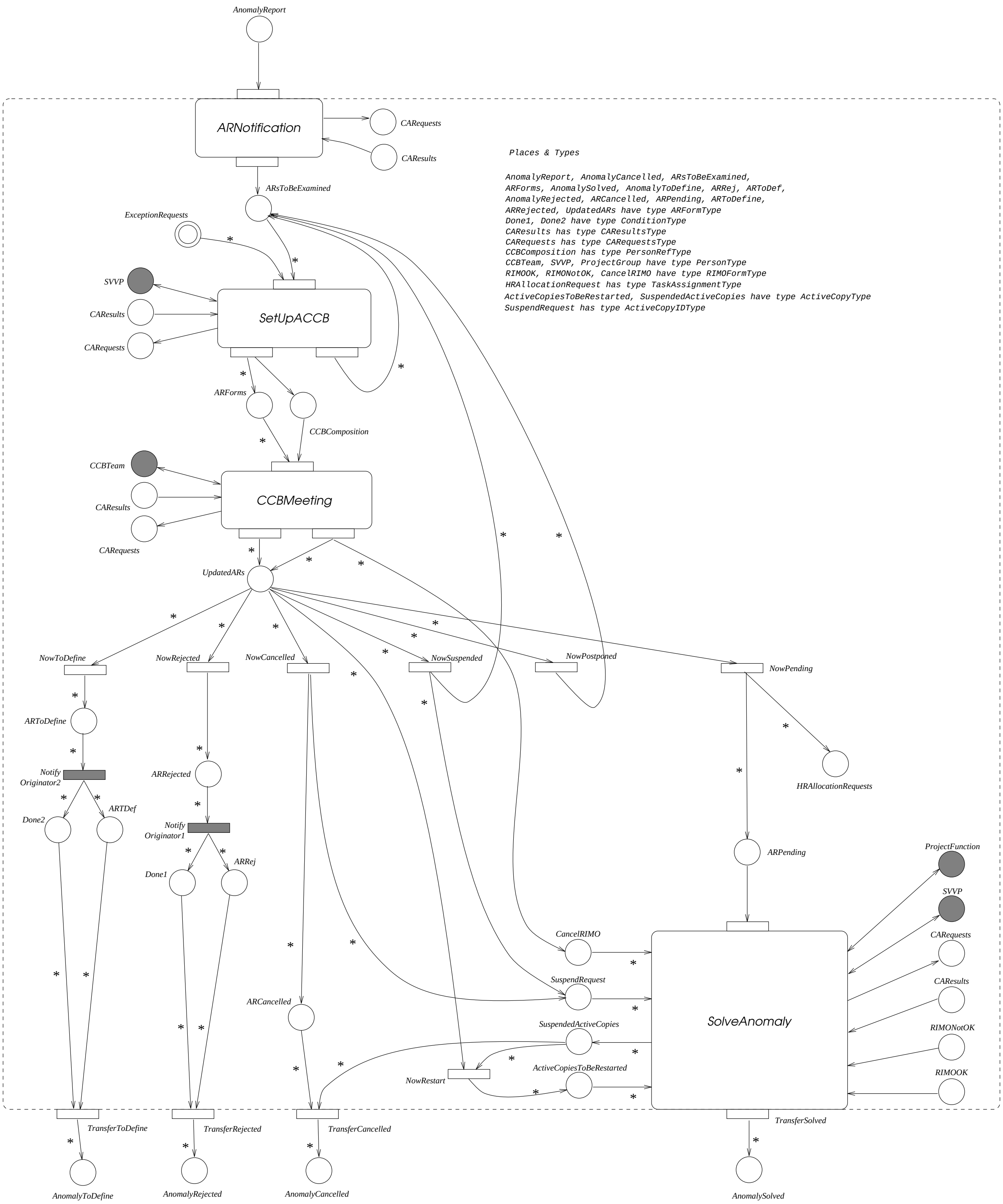


Figura 9.9: Un esempio di applicazione dei meccanismi di sospensione e cancellazione: la modifica dello stato di SolveAnomalies nell'implementazione dell'attività ManageSoftwareAnomalies.

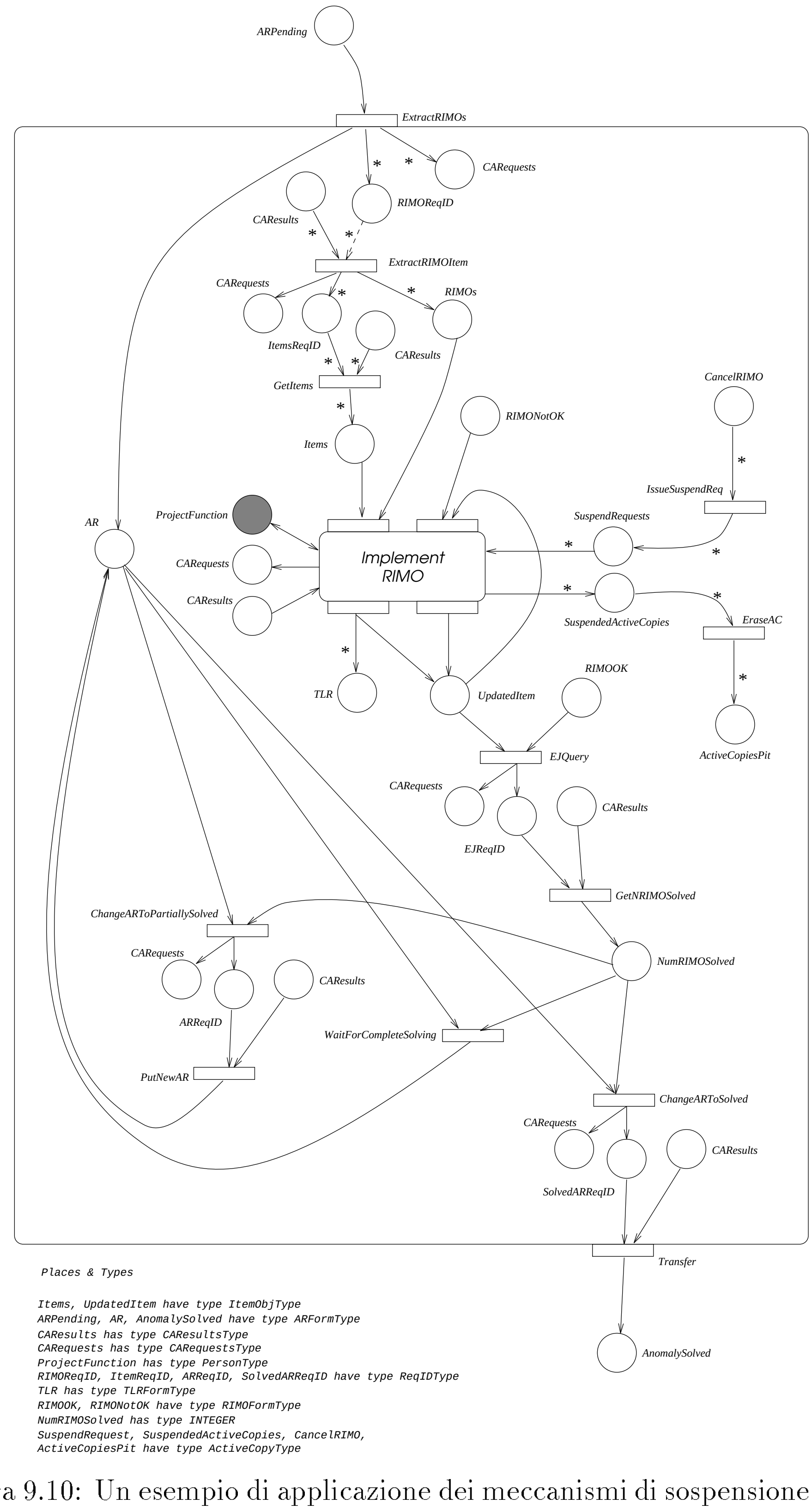


Figura 9.10: Un esempio di applicazione dei meccanismi di sospensione e cancellazione: la modifica dello stato di ImplementRIMO nell'implementazione di SolveAnomalies.

- `NowSuspended`. Il CCB ha deciso che le anomalie in uscita a questo evento debbono assumere il nuovo stato di "Suspended" e pertanto il loro stato deve essere "congelato" in attesa di nuovi sviluppi. Ciò viene ottenuto inserendo nel posto `SuspendRequest` di SolveAnomaly l'identificatore della copia attiva associata all'anomalia e presente nel campo `Instance` di quest'ultima.

- `NowCancelled`. Il CCB ha deciso che l'anomalia deve essere cancellata dal processo correntemente in esecuzione, pertanto la copia attiva corrispondente deve essere rimossa dal sistema. Anche in questo caso viene inserito in `SuspendRequest` l'identificatore della copia attiva, ma viene svolta un'ulteriore operazione: allorchè il sistema rende disponibile la copia attiva in `SuspendedActiveCopies`, originariamente per permetterne la manipolazione, tale copia attiva viene selezionata dall'evento `TransferCancelled`, il quale di fatto consuma il token corrispondente rimuovendolo da `SuspendedActiveCopies` e quindi impedendone la riattivazione in un secondo tempo.

- `NowRestarted`. Viene abilitato quando il CCB ritiene che sia possibile riattivare l'esecuzione di un'attività precedentemente sospesa: in tal caso la copia attiva presente in `SuspendedActiveCopies` viene prelevata e trasferita in `ActiveCopiesToBeRestarted`, consentendo all'interprete SLANG di riconsiderarla tra le attività in esecuzione.

Da questo esempio si nota come la cancellazione di un'attività, operazione non esplicitamente prevista tra i meccanismi SLANG, sia di fatto realizzabile con il "trucco" descritto precedentemente: prima l'attività viene sospesa e poi la sua copia attiva viene rimossa dal posto `SuspendedActiveCopies` ed in qualche maniera resa inaccessibile al resto della rete.

Tale tecnica viene sfruttata, in una sua variante, all'interno dell'attività SolveAnomaly, per agire sullo stato di ImplementRIMO in modo da eseguire la cancellazione di Richieste di Modifica da parte del CCB.

Come evidenziato in Fig. 9.9, la selezione delle RIMO da cancellare avviene depositandone gli oggetti corrispondenti in `CancelRIMO`, posto intermedio di SolveAnomaly. Come si vede dalla Fig. 9.10, tale posto è sfruttato nell'implementazione dell'attività per estrarre le copie attive associate alle RIMO e depositarle nel posto `SuspendRequest` di ImplementRIMO.

Una volta che l'interprete ha provveduto a depositare in `SuspendedActive-Copies` le corrispondenti copie attive sospese, queste vengono estratte dal posto in questione e depositate in un posto che svolge la funzione di "pozzo", nel quale vengono raccolte tutte le copie attive da rendere inutilizzabili al fine di realizzare la cancellazione. Si noti che tali copie attive non sono più utilizzabili e verranno fisicamente rimosse dal sistema allorché l'attività SolveAnomalies che le contiene avrà terminato la sua esecuzione.

9.2.4 Le risorse umane

La descrizione delle risorse umane e della loro gestione costituisce uno degli aspetti del processo modellati direttamente in SLANG senza l'ausilio del Manuale di Qualità, in quanto quest'ultimo non descrive affatto questo tipo di aspetti. Il nostro modello in questo caso si è dunque dovuto basare soltanto sulle informazioni dedotte dai colloqui con l'esperto del processo e già riportate nella Sez. 8.2.

Le risorse umane sono rappresentate all'interno del modello in maniera omogenea agli altri dati del processo, cioè fluiscono all'interno delle attività sottoforma di token di tipo `PersonType`, la cui definizione è la seguente:

```
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];
```

Come si può vedere, una persona è rappresentata attraverso le comuni informazioni di tipo amministrativo, necessarie ad identificare univocamente e reperire la risorsa umana in questione, oltre ad una coda di compiti che devono essere da questa eseguiti e che viene utilizzata per la gestione dell'allocazione delle risorse secondo i criteri descritti nella Sez. 8.2 e modellati nell'attività ManageHumanResourcesAllocation.

Il meccanismo utilizzato per la rappresentazione delle risorse umane si avvale delle seguenti considerazioni:

- Le risorse umane, come già rimarcato, sono descritte da token di tipo `PersonType`, i quali contengono le informazioni di base per la loro identificazione ed allocazione.

- I posti del modello destinati a contenere insiemi di risorse umane logicamente correlati, quali possono essere ad esempio dei dipartimenti oppure una delle Funzioni descritte nel Cap. 4, sono distinti da un colore bruno: in particolare, questo tipo di posto è stato chiamato *human place*.

- Gli human place sono condivisi all'interno della specifica, cioè possono essere pensati come definiti locali all'attività radice nella gerarchia in Fig. 9.4.

- Quando una transizione avente in ingresso uno human place scatta, uno o più token vengono prelevati: questo evento modella di fatto l'allocazione *fisica* della persona al task. In particolare, se nel posto in ingresso non vi sono token, la transizione non può scattare: in altre parole, nessuna risorsa umana è attualmente disponibile per l'esecuzione del task.

- La riallocazione delle risorse umane dopo l'esecuzione del task è compito di chi modella la rete: in altre parole, non è fornito nessun meccanismo predefinito per realizzare ciò.

Una ulteriore possibilità di estensione di questo meccanismo consiste nell'associare un campo addizionale al tipo `PersonType` tale da contenere la percentuale di occupazione della risorsa: in questo modo l'allocazione di una persona ad un task potrebbe avvenire fintantoché la percentuale di risorsa libera non scende a zero. Questo "frazionamento" della risorsa permetterebbe una rappresentazione maggiormente aderente alla pratica industriale, dove una persona non viene allocata *in toto* ad un compito, bensì può dedicarsi in maniera concorrente allo svolgimento di più compiti assumendo magari più di un ruolo.

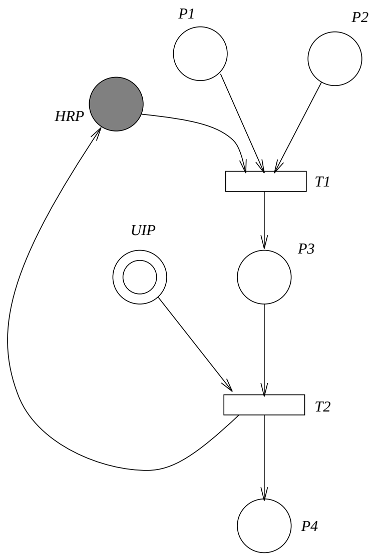


Figura 9.11: Il meccanismo usato per modellare l'interazione con parti di processo svolte esternamente al modello.

9.2.5 Le attività esterne al modello

Il modello presentato in questo lavoro non pretende di descrivere esaustivamente ogni aspetto della produzione del software in Italtel BUTD, almeno non in questo prima versione: si assume dunque che alcuni dei compiti previsti per lo svolgimento di un'attività possano essere eseguiti al di fuori del modello, il quale controllerà dunque le precondizioni e le postcondizioni per l'esecuzione del compito ma non i dettagli dell'esecuzione medesima.

Tra l'altro, in genere gli aspetti non modellati all'interno della nostra specifica coincidono con aspetti non affrontati dal Manuale di Qualità.

È questo il caso ad esempio degli aspetti legati all'implementazione di modifiche ad elementi di configurazione: nessuna istruzione operativa definisce in modo preciso un insieme di operazioni da eseguire per effettuare le modifiche, ma sono definiti in maniera precisa gli oggetti che devono essere presenti affinché tale attività possa iniziare.

Nelle varie attività del modello si è utilizzato sempre uno stesso meccanismo per modellare l'interazione tra questo tipo di compiti svolti esternamente al modello e il modello medesimo, il cui schema generale è riportato in Fig. 9.11.

In tale figura, i token in P1 e P2 rappresentano le condizioni necessarie per l'inizio del compito da svolgersi esternamente, mentre i token in HRP rappresentano la (o le) risorse umane che possono essere selezionate per l'esecuzione. L'inizio del compito è segnalato dallo scatto dell'evento T1: tale transizione trasferisce i token in HRP (di tipo PersonType) in P3, dove rimangono fino allo scatto dell'evento T2. Tale evento, in questo modo, può scattare soltanto dopo T1 e dopo che nello *user interface place* UIP siano comparse le informazioni generate all'esterno del processo da parte degli esecutori del compito assegnato.

La situazione in cui T1 è già scattato (per cui vi sono dei token in P3) e T2 non può ancora scattare (perché non vi sono tokens in UIP) è rappresentativa del fatto che il processo è in attesa dei risultati provenienti dall'esecuzione del compito svolto esternamente. Inoltre, poiché P3 contiene un token rappresentante una risorsa umana, il quale è stato prelevato dall'insieme delle risorse umane utilizzabili per l'esecuzione del task, tale situazione rappresenta anche il fatto che una ben precisa risorsa umana è allocata all'esecuzione del task.

Lo scatto della transizione T2 trasferisce i risultati dallo user interface place ad un posto semplice, per successivi utilizzi, e rialloca le risorse umane utilizzate per il compito, prelevandole da P3 e reinserendole nel posto HRP, permettendo dunque l'abilitazione di altre transizioni nel modello che abbiano tale posto in ingresso.

9.2.6 Uso del polimorfismo e late binding

Come è stato puntualizzato nel Cap. 3 SLANG fornisce pieno supporto alle caratteristiche orientate agli oggetti impiegabili nella costruzione di un modello. In particolare, l'interprete SLANG, una volta ultimato all'interno del progetto SPADE, effettuerà un *type-checking* durante l'esecuzione delle reti per verificare che effettivamente i token in ingresso ad un posto siano accettabili: se `TPlace` è il tipo di un posto P e `TToken` è il tipo di un token T che viene depositato in P, il token può essere effettivamente depositato *se e soltanto se* `TToken` *coincide con* `TPlace` *oppure ne è un sottotipo.*

Si noti che questa regola coincide di fatto con quella comunemente utilizzata per l'assegnazione di variabili complesse in molti linguaggi di programmazione. Tale analogia lascia intuire come un controllo di questo tipo, per esempio effettuato in simulazione mediante animazione della rete, sia un valido aiuto per la verifica della correttezza del modello, analogamente a quanto avviene per la fase di type-checking eseguita dai compilatori di linguaggi di programmazione tipizzati.

La possibilità di depositare un token all'interno di un posto il cui tipo può essere un supertipo di quello del token permette di sfruttare i meccanismi di polimorfismo e late binding per specializzare le operazioni da effettuare sul token da parte degli eventi che lo utilizzano a seconda del tipo di oggetto che esso rappresenta.

Tale tecnica è stata utilizzata all'interno del modello per due scopi: la verifica della correttezza formale di un elemento consegnato al controllo configurazione e la selezione delle *mailing list* da utilizzarsi ogniqualvolta un elemento in controllo configurazione subisce una transizione di stato.

Il problema, in questi casi, consiste nell'eseguire parti del processo che in parte sono costituite da operazioni ripetitive e di routine ed in parte vanno specializzate a seconda dell'elemento su cui sono applicate. Ad esempio, il controllo formale effettuato su un documento si limita a controllare la corretta compilazione dei campi presenti sul Modulo di Accompagnamento, ma il numero ed il significato di tali campi varia se l'elemento è ad esempio un SCU oppure un AR. Analogamente, la costruzione di una mailing list è un operazione banale, tuttavia ogni tipologia di documento possiede i propri criteri per la sua redazione. La soluzione è automaticamente fornita dal late binding e dal polimorfismo.

Ad ogni documento sono associati due metodi, `FillDistribution` e `CheckFormalCorrectness`, definiti nella radice della gerarchia dei documenti, `ItemObjType`. I metodi definiti in tale tipo definiscono i criteri di base per l'effettuazione delle operazioni descritte precedentemente, in altre parole costituiscono l'insieme di operazioni di base indipendenti dalla tipologia del documento. Ogni metodo che appare nei discendenti, invece, (ad esempio `ARFormType`) può ridefinire il metodo base per adattarlo alla tipologia di documento che rappresenta (ad esempio un Rapporto di Anomalia). L'evento che attiva il metodo utilizzerà un unico nome del metodo per effettuare l'operazione: tuttavia, grazie ai meccanismi di late binding e polimorfismo soltanto il metodo effettivamente collegato all'oggetto in ingresso verrà selezionato.

La Fig. 9.12 riporta un esempio di applicazione di questo meccanismo, dove la transizione `PerformCheck` è così descritta:

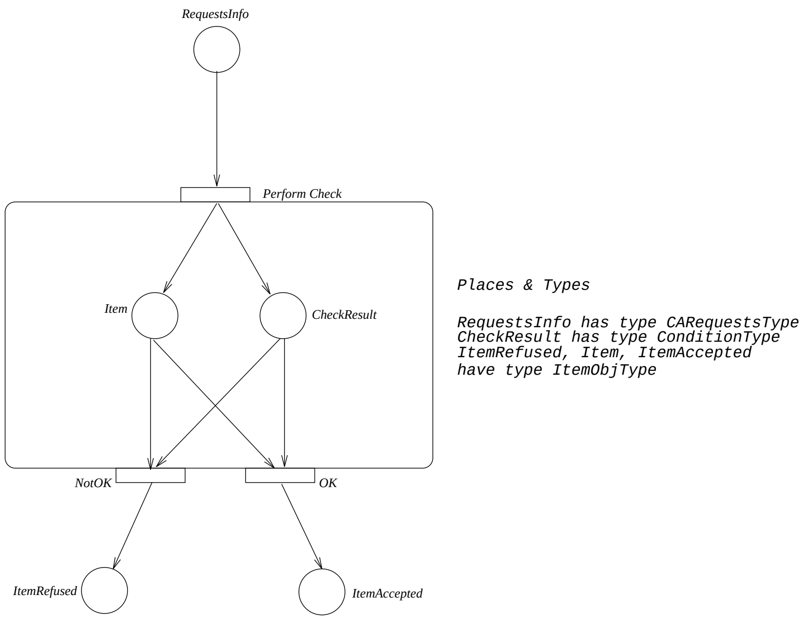


Figura 9.12: Un esempio di applicazione di polimorfismo e late binding: il controllo formale di un elemento di configurazione.

```
EVENT PerformCheck(RI; I,CR)
GUARD
RI IN RequestInfo
SUCH THAT
RI.OpReq=CheckItem;
ACTION
CR:=RI.ItemObj.CheckFormalCorrectness;
```

da cui si vede che `PerformCheck` invoca sempre e comunque `CheckFormalCorrectness` per determinare se l'elemento è o meno corretto. `RI.ItemObj` è definito avere tipo `GeneralFormType`, tipo radice della gerarchia dei moduli: ogni tipo discendente da esso avrà una sua versione specializzata di `CheckFormalCorrectness`.

9.2.7 Controllo delle scadenze temporali

La parte di processo presa in considerazione all'interno di questo lavoro non ha comportato un pesante uso delle notazioni di SLANG relative alla specifica degli aspetti temporali; ciò è dovuto al fatto che praticamente l'unico aspetto di questo genere apparso durante la raccolta delle informazioni relative al processo è costituito dalle scadenze associate all'esecuzione di una Richiesta di Modifica.

Poichè tale compito è eseguito attraverso un insieme di eventi, la natura di tale vincolo farebbe intuire un possibile utilizzo del costrutto di *timeout*, nato proprio per inserire vincoli temporali su sequenze complesse di eventi. Al contrario, la natura statica della definizione originale di tale vincolo ne ha impedito l'uso: il significato di tale affermazione sarà più chiaro alla luce di quanto verrà discusso nel Cap. 10.

Ci si è dovuti pertanto accontentare di imporre un vincolo temporale tradizionale associato alla consegna dell'elemento di configurazione modificato in accordo con la RIMO: come mostrato in Fig. 9.13, le transizioni di uscita del meccanismo di attesa che gestisce l'attività esterna di modifica sono due, una delle quali (`NowTimedOut`), una delle quali contiene un vincolo temporale così descritto:

```
EVENT NowTimedOut(U,RIMO; UOut,CARq)
LOCAL c: CARequestsType;
```

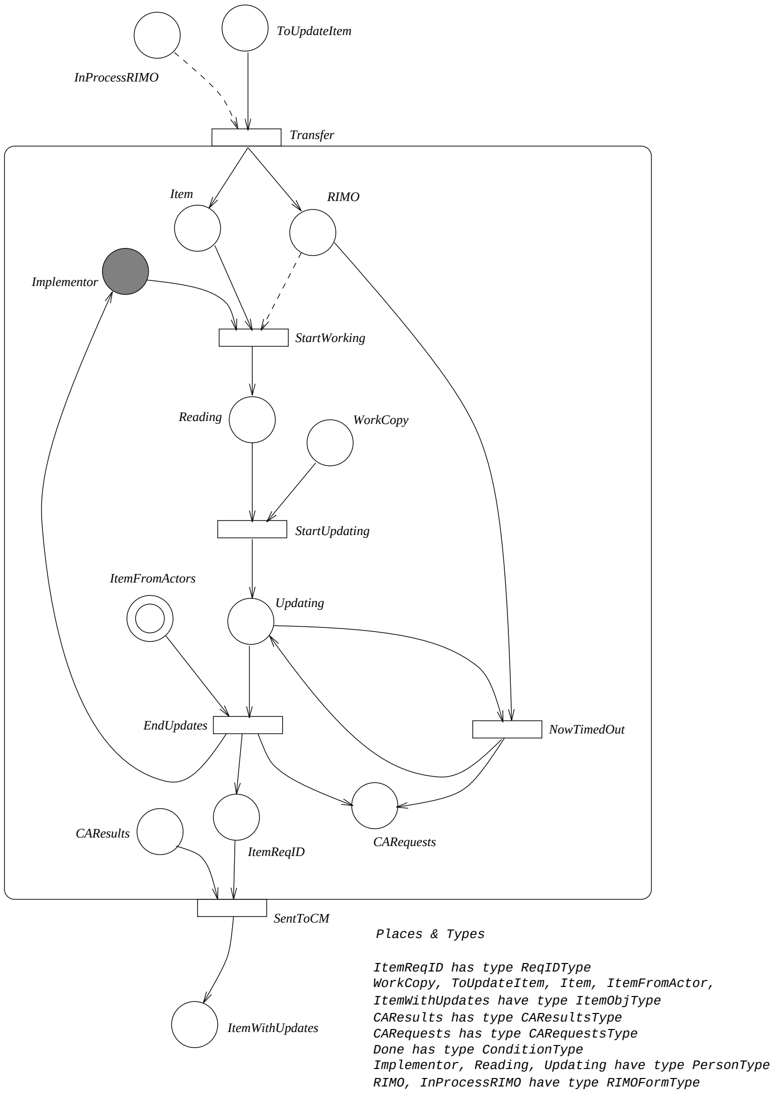


Figura 9.13: L'attività di modifica di un elemento di configurazione, contenente un vincolo temporale.

```
GUARD
U IN Updating,
RIMO IN RIMO
ASAP AFTER RIMO.Deadline;
ACTION
c.ItemRef:=RIMO.ID;
c.OpReq:=ChangeState;
c.Status:=TimedOut;
CARq:=CARq UNION c;
UOut:=U;
```

Quindi in pratica vengono modificate tutte le RIMO che sono scadute, mutando il loro stato in “TimedOut”.

Si noti che il vincolo `ASAP` legge il valore della scadenza direttamente dal token di tipo `RIMOFormType` in ingresso alla transizione, in maniera dinamica.

9.2.8 Invocazione dei tool

Come noto dal Cap. 3, l'invocazione dei tool all'interno del processo è gestita dal costrutto *black transition*, il quale contiene all'interno dell'azione legata alla transizione l'invocazione della routine eseguibile (non-SLANG) che contiene di fatto il tool vero e proprio.

Tuttavia, non vi è un'indicazione precisa su come reperire i risultati dell'invocazione al termine dell'esecuzione: la soluzione adottata nel modello è consistita nell'assumere che il tool sia definito come un oggetto all'interno del sistema, quindi con propri attributi e metodi, sebbene esso non possa essere utilizzato come token all'interno di una rete SLANG.

In altre parole, la descrizione dei tool in termini di gerarchia di oggetti ha semplicemente un valore operativo, per facilitare la definizione delle modalità di reperimento dei valori finali dell'esecuzione del tool e non è necessariamente la rappresentazione che verrà adottata dalla versione finale di SPADE.

Alla luce di queste considerazioni, un tool del modello è definito come un oggetto di un tipo appartenente alla gerarchia descritta in Fig. 9.14.

In particolare, la radice di tale gerarchia è costituita dal tipo `GeneralToolType`, i cui campi sono:

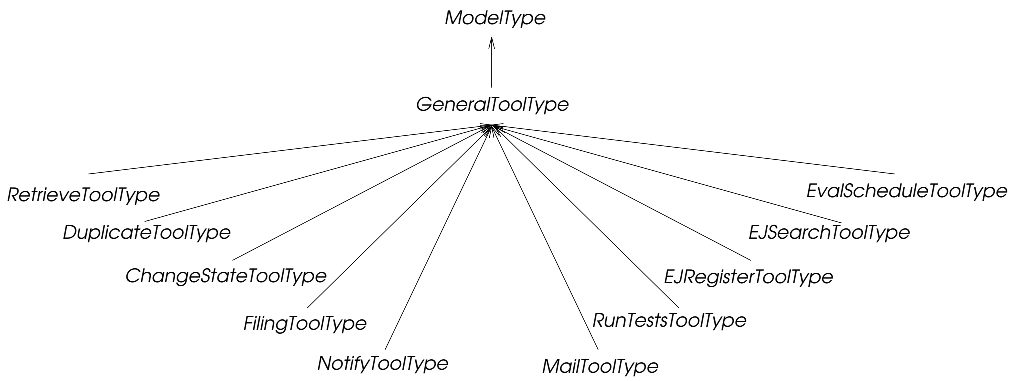


Figura 9.14: La gerarchia di oggetti utilizzata per descrivere il comportamento dei tool.

- `Invoke`. È il metodo che assumiamo essere utilizzato per realizzare l'invocazione del tool.

- `ToolResult`. Qui si assume che vengano depositati i risultati prodotti dal tool al termine della sua esecuzione. Ad esempio, nel caso di invocazione di un compilatore qui verrebbe posto il codice oggetto.

- `ToolOK`. Qui si assume che venga depositato l'esito dell'operazione. Nell'esempio della compilazione, qui verrebbero inseriti eventuali codici d'errore.

La descrizione dei risultati ottenuti dai tool e dai parametri necessari per l'invocazione di questi è a questo punto semplificata: ogni tool viene definito come discendente di `GeneralToolType`, di cui ridefinisce la lista di parametri del metodo `Invoke` (specificando quindi gli oggetti necessari in ingresso al tool per l'invocazione) e i due attributi `ToolResult` e `ToolOK`, uno dei quali può essere anche definito come assente.

Ad esempio, il tool impiegato per la notifica di informazioni è definito nella seguente maniera:

```
NotifyToolType SUBTYPE OF GeneralToolType
ToolResult=NIL;
PROCEDURE Invoke(notifyTo: SET[PersonRefType]; msg: Text;
  items: SET[ItemObjType]);
```

dove appare chiaro che il metodo `Invoke` necessita di un insieme di persone a cui effettuare la notifica, di un messaggio di notifica e di un elemento di configurazione che ha generato la necessità della notifica; inoltre, l'unico output del tool è l'esito dell'operazione (definito nel supertipo `GeneralToolType`), dal momento che `ToolResult` viene considerato non definito.

L'invocazione del tool e la successiva raccolta dei risultati avverrà mediante una sequenza di istruzioni (da specificarsi nell'azione) analoga alla seguente:

```
NotifyTool.Invoke(T1.Autore,'Messaggio',T1);
T2:=NotifyTool.ToolOK;
```

dove `T1` e `T2` sono variabili usate per riferirsi ai token e `NotifyTool` è il tool di tipo `NotifyToolType`.

Capitolo 10

Problemi ed esperienze nella modellazione con SLANG

Questo capitolo raccoglie i risultati ottenuti dall'esperienza di modellazione che possono servire sia per aumentare l'espressività e l'efficacia del linguaggio sia per guidare eventuali lavori di modellazione futuri. In tal senso, quando si parla di "risultati" l'enfasi è sugli aspetti legati a SLANG piuttosto che su quelli legati al Manuale di Qualità, i quali peraltro sono già stati discussi nel Cap. 7.

10.1 Problemi del linguaggio

Come evidenziato nell'introduzione a questo lavoro, uno degli obiettivi fissati per il lavoro di modellazione consiste nella segnalazione di insufficienze del linguaggio nella modellazione di situazioni reali di processo. Tali inadeguatezze sono state effettivamente scoperte e la presente sezione ne costituisce un resoconto.

10.1.1 Descrizione dei vincoli temporali

Come descritto nella Sez. 3.3 SLANG offre due tipologie di costrutto per la specifica di vincoli temporali imposti agli eventi di un'attività, e cioè i *timestamps* associati ai token ed il *timeout*.

L'utilizzo dei timestamps è *locale*, in quanto vengono usati per imporre vincoli sullo scatto di singole transizioni e pertanto sono in grado di vincolare soltanto l'esecuzione di singoli eventi.

Al contrario, il costrutto di timeout abbraccia un intera parte della specifica di un'attività, delimitata da due eventi: in tal modo esso può essere utilizzato per vincolare l'esecuzione di un insieme complesso di eventi, offrendo quindi la possibilità di modellare vincoli di tipo *globale*.

Abbiamo inoltre identificato due categorie di vincoli che possono sorgere all'interno del processo software:

- *Vincoli statici.* Essi si basano su conoscenze *a-priori* circa l'esecuzione del processo, magari già codificate all'interno del Manuale di Qualità. Un esempio potrebbe essere: "Ogni attività di sviluppo software deve produrre un rapporto entro 5 giorni dal suo inizio". Si noti che nella parte di Manuale di Qualità esaminata per questo lavoro di modellazione non è stato incontrato neanche un caso appartenente a questo tipo di vincolo.

- *Vincoli dinamici.* Vengono generati all'interno del processo, tipicamente come conseguenza della presenza di altre fasi evolventi in parallelo e le loro caratteristiche sono determinate dalla immagine del processo. Un esempio di tale tipologia di vincolo temporale è costituito dalle caratteristiche delle Richieste di Modifica; come visto nel Cap. 5, le RIMO emesse dal CCB devono essere realizzate in un tempo massimo indicato nel modulo di RIMO. In questo caso è evidente che il valore della scadenza non può essere inserito all'atto della stesura del modello del processo, in quanto esso viene deciso dal CCB durante la riunione e varia, anche sensibilmente, a seconda dello stato di avanzamento del processo, della fase in cui è stata originata l'anomalia, della disponibilità di risorse e da tutta una serie di altri fattori impredicibili in fase di progetto del modello.

Il costrutto di timeout, così come definito in [BFGG92], ci permette di definire soltanto vincoli di tipo statico: la ragione di ciò va ricercata nel fatto che nel definire un timeout la sua scadenza viene specificata mediante un numero intero e risulta perciò predefinita all'atto della descrizione del modello, senza alcuna possibilità di parametrizzazione di tale valore o di collegarlo dinamicamente ad altre parti del processo.

	SLANG Constructs	
Vincoli	*Timestamps (locali)*	*Timeout (globali)*
Statici	x	x
Dinamici	x	

Tabella 10.1: Construtti e vincoli temporali.

L'uso dei meccanismi legati ai timestamps, invece, consente di descrivere sia vincoli statici che dinamici in quanto le informazioni legate agli aspetti temporali possono essere fissate all'interno della specifica oppure determinate attraverso il contenuto dei token in ingresso all'evento su cui è imposto il vincolo.

Le considerazioni svolte finora sono riassunte nella Tab. 10.1, in cui viene mostrato come nella versione attuale di SLANG si sia in grado di modellare vincoli statici in maniera locale e globale, mentre si è in grado di specificare vincoli dinamici solo in maniera locale.

Il tipo di vincolo che non si riesce a modellare, cioè quello dinamico e globale, ha rilevanza tutt'altro che secondaria all'interno del processo di sviluppo del software in generale e del processo Italtel BUTD in particolare. Ad esempio, come già accennato nel Cap. 9, la modellazione della scadenza della RIMO si sarebbe potuta effettuare in maniera più precisa considerando anche le attività di riqualificazione impiegando un costrutto di timeout anziché un vincolo locale quale quello impiegato nella definizione dell'attività ModifyConfigurationItems.

Va osservato che una descrizione del processo che non tenga conto degli aspetti temporali perde molte delle sue attrattive, dal punto di vista della sua reale applicazione nel controllo del processo, in quanto il soddisfacimento delle scadenze è una delle sfide più impegnative per il management, a causa degli elevati costi in genere associati a ritardi nella consegna.

La soluzione proposta in questo lavoro per estendere il timeout in modo da riuscire a specificare anche vincoli dinamici semplicemente aggiunge un posto al costrutto, collegato alla barra che delimita la porzione di rete sotto controllo del timeout.

Tale soluzione è illustrata in Fig. 10.1 mediante un "esempio giocattolo" (tratto da [BF93]), il quale descrive un'attività di codifica soggetta ad una scadenza temporale fissata nel modello, quindi inaccessibile dinamicamente.

La Fig. 10.2 mostra un'attività parallela a quella di codifica che necessita di intervenire sulla scadenza fissata per essa: nella fattispecie potrebbe essere un'attività di monitoraggio del processo (per esempio da parte del management) che, in seguito alla rilevazione di problemi hardware o di scarsità di risorse decide di allungare la scadenza. Tale attività non avrebbe modo di intervenire su un timeout tradizionale, ma può intervenire sulla nuova definizione dinamica del costrutto, inserendo direttamente la nuova scadenza nel posto TimeoutExtent associato al timeout.

In questo caso la modifica della scadenza avviene esternamente all'attività controllata dal timeout, ma nulla vieta di adottare tecniche più sofisticate in grado di modificare dall'interno dell'attività la scadenza associata: la Fig. 10.3 mostra un esempio. Quest'ultima caratteristica permette la definizione di vincoli temporali *distribuiti* anziché *puntuali*, cioè di vincoli che possono essere parzialmente rilasciati o modificati a seconda dell'andamento del processo, anzichè costituire delle scadenze monolitiche e inalterabili.

Questa è una pratica abbastanza frequente nella gestione del processo, dove il costo (in termini monetari diretti oppure in termini di riorganizzazione del processo) associato al ritardo in una consegna non è fisso ma varia a seconda dell'entitaà e delle cause del ritardo. È quindi utile avere un meccanismo, quale quello descritto, in grado di descrivere la riorganizzazione della scadenza e le azioni associate.

Una volta illustrati in maniera informale i possibili vantaggi derivanti dall'introduzione di un'estensione dinamica del timeout, alcuni aspetti rimangono ancora indefiniti: in particolare, la definizione del posto TimeoutExtent è a livello della definizione di attività (ve n' è uno solo per tutte le copie attive, analogamente al posto predefinito SuspendRequest) oppure a livello di istanza (ve n'è uno per ogni copia attiva)?

Per omogeneità con gli altri posti predefiniti, cioè SuspendedActiveCopies, SuspendRequest, ActiveCopiesToBeRestarted, il posto TimeoutExtent dovrebbe essere inserito a livello della definizione dell'attività: in tal caso allora bisognerebbe definire un meccanismo simile a quello adottato all'interno della gestione della configurazione, per associare ogni scadenza con la corretta copia attiva. Tale problema si presenta anche nel caso di *thread* multipli di esecuzione della parte di rete contenente il timeout: in tal caso, se la semantica da assegnare a questa situazione impone che ogni thread possegga

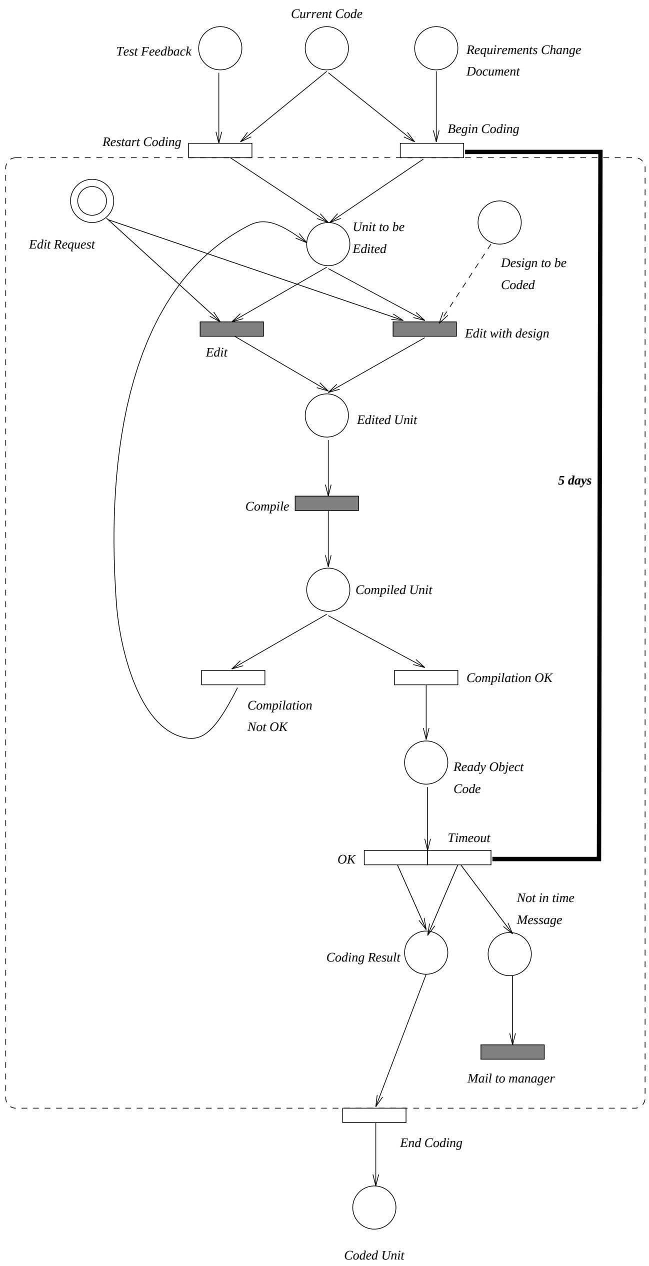


Figura 10.1: Un esempio di attività software soggetta a scadenza: la codifica.

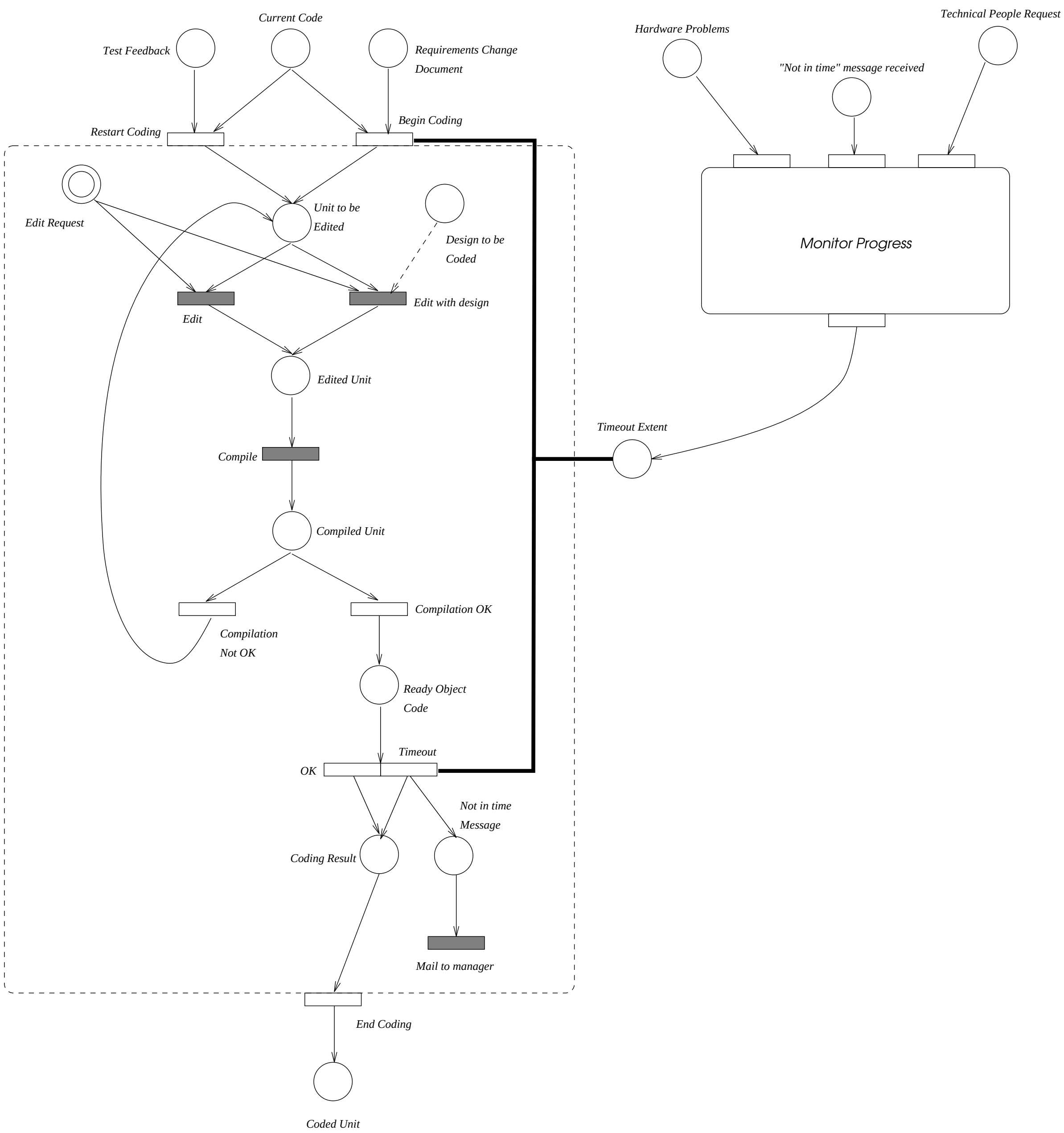


Figura 10.2: Un esempio di uso della definizione dinamica del timeout.

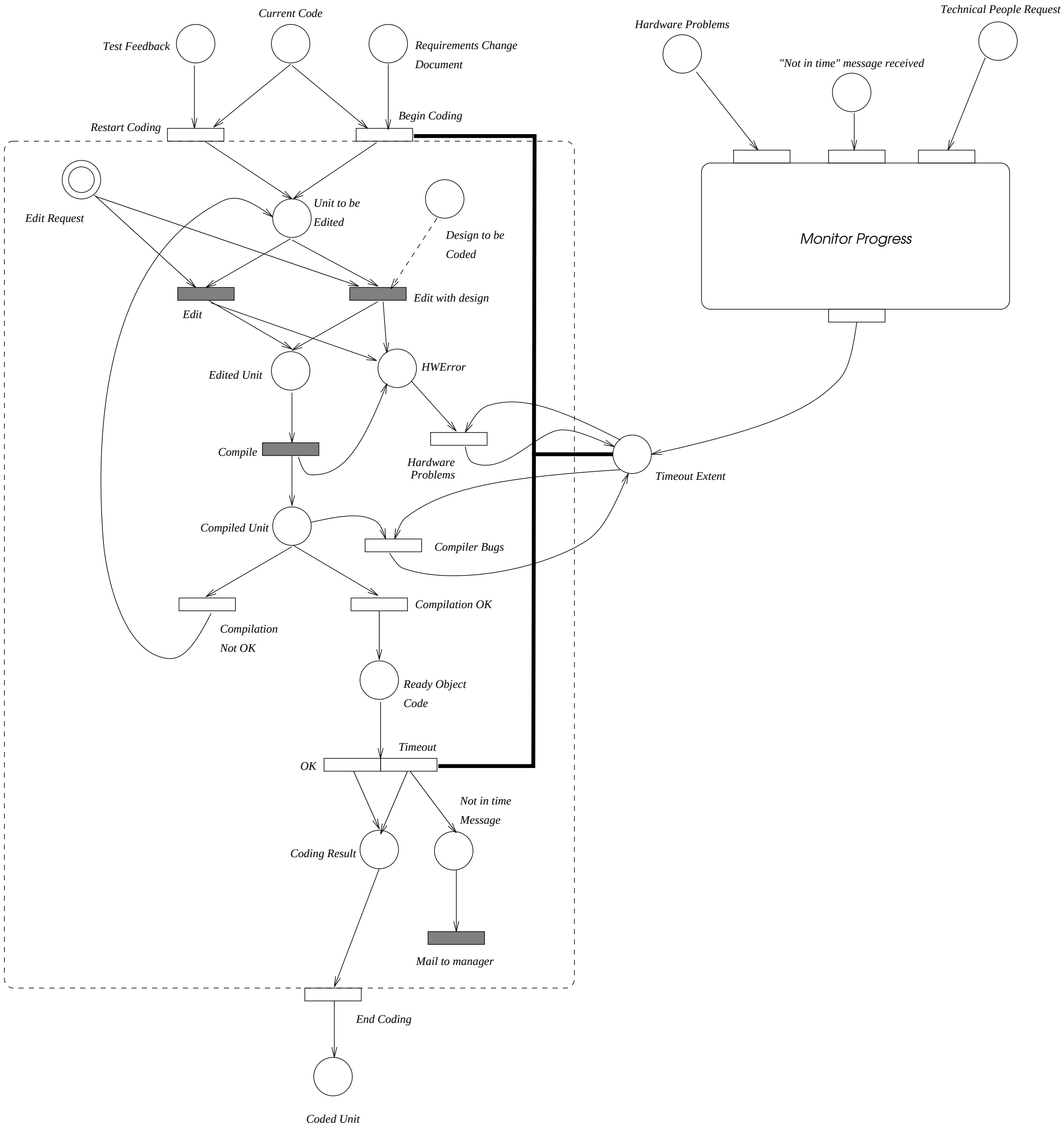


Figura 10.3: Un uso sofisticato del timeout dinamico.

una propria scadenza di timeout, è necessario associare un identificatore ad ogni flusso di esecuzione.

Per completezza, va puntualizzato il fatto che probabilmente la nuova versione del linguaggio terrà conto dei problemi mostrati da SLANG nella gestione degli aspetti temporali e conterrà un ripensamento globale di tutti i costrutti coinvolti.

10.1.2 Modularizzazione di reti

Come abbiamo avuto modo di ricordare in più di una occasione, un modello SLANG può essere organizzato in un insieme di blocchi modulari costituiti dalle attività, le quali comunicano le une con le altre attraverso interfaccie definite dal modellatore del processo.

Ciononostante, come verrà puntualizzato nella Sez. 10.2, per chi ha una certa consuetudine con i linguaggi di programmazione il costrutto di attività può essere talvolta fuorviante, in quanto si deve tenere presente che il "codice" contenuto all'interno dell'implementazione dell'attività viene replicato all'interno del sistema tante volte quante sono le copie attive in quel momento presenti e, perdipiù, viene generata una nuova copia attiva ogniqualvolta una tupla abilitante è presente nei posti d'ingresso dell'attività. In tal modo si riesce a rappresentare la situazione, tipica della consuetudine industriale, in cui attività simili per modalità di esecuzione sono eseguite concorrentemente su dati di ingresso diversi, pertanto *distribuendo* il carico di lavoro (le tuple di ingresso) su più esecutori (le attività).

Tuttavia, l'esperienza maturata in un processo software reale ha evidenziato che tale distribuzione di compiti all'interno del processo risulta talvolta indesiderata, in quanto in questi casi si desidera *centralizzare* la gestione di un insieme di oggetti del processo. Un caso tipico sono ad esempio le attività di monitoraggio e supervisione.

Un esempio all'interno del processo è costituito dall'attività di gestione delle anomalie; una prima modellazione aveva descritto questa parte di processo mediante un attività SLANG (in Fig. 10.4), in cui l'unico posto in ingresso doveva raccogliere i Rapporti di Anomalia generati all'interno del processo. L'arco che collegava tale posto con l'evento d'inizio dell'attività era di peso unitario, pertanto veniva creata una copia attiva *per ogni Rapporto di Anomalia.* Mediante questo approccio, anche le attività invocate da Ma-

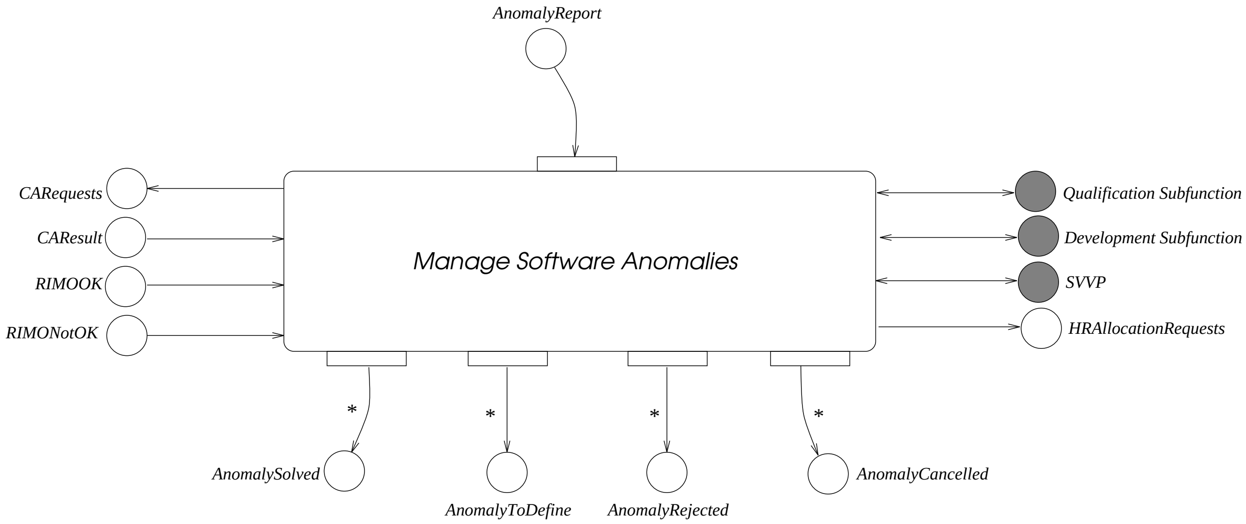


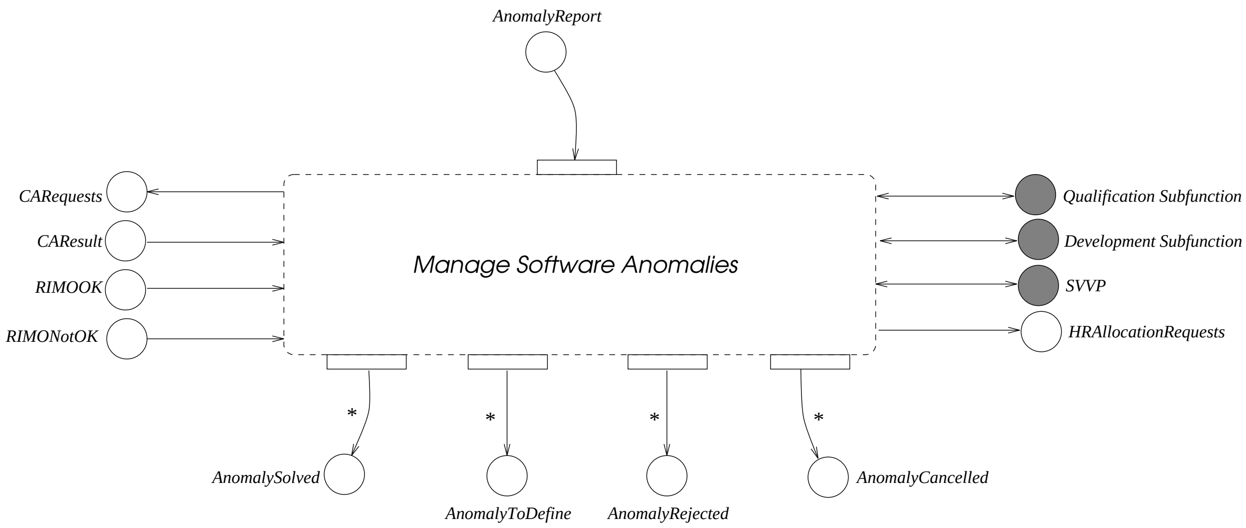
Figura 10.4: La prima versione dell'interfaccia di ManageSoftwareAnomalies.

nageSoftwareAnomalies lavoravano su un unico AR, quindi anche le attività che gestiscono la preparazione e la realizzazione del CCB, cioè SetupCCB e CCBMeeting.

In realtà, questa costituisce un'approssimazione troppo pesante rispetto a quanto effettivamente realizzato all'interno del processo: in quest'ultimo, infatti, la SVVP raccoglie *tutti* gli AR generati all'interno del processo e convoca il CCB solo quando tali AR sono in numero sufficiente o la loro gravità è molto elevata. Inoltre, il CCB non esamina mai una anomalia alla volta ma, per ottimizzare i tempi, durante una convocazione del CCB vengono esaminate più anomalie.

A questo punto, se si vuole modellare la gestione delle anomalie come un'attività SLANG che riceve in ingresso degli AR ed emette in uscita gli AR con gli stati terminali definiti nella Sez. 8, appare chiaro che il costrutto di attività non è utilizzabile per una tale situazione, in quanto AR che giungono in tempi diversi a tale attività generano copie attive diverse dell'attività, senza la possibilità di centralizzarne la gestione.

Per risolvere il problema sarebbe necessario descrivere il controllo da applicare all'attività di gestione delle anomalie non mediante una rete che costituisca l'implementazione di un'attività SLANG, ma mediante una rete SLANG *tout court*, svincolata dal costrutto di attività.




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

Figura 10.5: L'interfaccia del modulo ManageSoftwareAnomalies.

In questo modo però si verrebbero a perdere i vantaggi offerti dalla separazione tra interfaccia ed implementazione, tipici del costrutto di attività.

La soluzione adottata all'interno del modello si colloca a metà tra l'introduzione di un nuovo costrutto e la creazione di una *macro*.

Abbiamo infatti definito un nuovo oggetto SLANG, il **modulo**, il quale è graficamente indicato in maniera identica ad un'attività, con la sola differenza che il rettangolo che la identifica è in questo caso tratteggiato, come mostrato in Fig. 10.5. Le differenze tra il modulo e l'attività sono così riassumibili:

- Quando un'attività è eseguita, a fronte dello scatto di una tupla di ingresso, quest'ultima viene gestita da un'istanza dell'attività dedicata unicamente a tale tupla. In altre parole, se l'evento di ingresso scatta n volte, vi saranno n copie attive che gestiscono le tuple corrispondenti.

- Quando un modulo viene eseguito, a fronte dello scatto di una tupla di ingresso, non viene generata alcuna istanza, ma la tupla viene gestita direttamente dall'implementazione del modulo. Ciò significa che se l'evento di ingresso scatta n volte ci sarà solo una istanza del modulo che gestisce le tuple: al limite vi potranno essere più *thread* di esecuzione.

In pratica, la "invocazione" di un modulo corrisponde alla macroespansione della sua implementazione all'interno della rete che descrive il chiamante.

Questo tipo di soluzione è stato impiegato per l'attività di gestione delle anomalie, ManageSoftwareAnomalies, e per l'attività di gestione della revisione, ManageSoftwareReview.

10.1.3 Meccanismi di gestione dello stato di attività cancellate

Si è visto nel Cap. 9 come attraverso SLANG si siano potuti descrivere anche le sospensioni e addirittura le cancellazioni di compiti complessi già in corso di esecuzione.

Tuttavia, questo tipo di meccanismo garantisce una corretta gestione di tali operazioni soltanto per quanto riguarda gli aspetti strettamente legati al linguaggio, garantendo che le attività che rappresentano tali compiti siano correttamente gestite all'interno del modello, ma un importante aspetto viene lasciato fuori dal modello attualmente presentato: infatti, non viene affatto considerato il problema di come gestire i "semilavorati" nel frattempo prodotti dalle attività che sono poi successivamente cancellate.

Ad esempio, cosa fare se si decide di cancellare un'attività che ha già prodotto dei componenti software consegnati in controllo configurazione? La cancellazione deve rimuovere dall'Archivio di Configurazione tutti i prodotti intermedi e ripristinare lo stato del processo antecedente all'inizio dell'attività, oppure tali semilavorati vanno comunque archiviati perché potrebbero in ogni caso tornare utili in fasi successive dello sviluppo? E ancora, supponendo di optare per la cancellazione dall'Archivio, che fare se tali elementi erano utili anche alla soluzione di altre RIMO?

Nel momento in cui si afferma la possibilità di cancellare tutti i risultati intermedi prodotti da un'attività non si può fare a meno di pensare ad un'analogia con quanto viene fatto nelle basi di dati per la gestione delle transazioni: quando una transazione non va a buon fine, i risultati intermedi

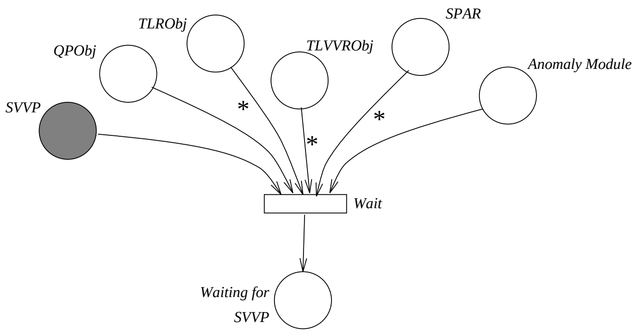


Figura 10.6: Un esempio relativo al problema delle alternative multiple.

vengono cancellati dalla base di dati (attraverso l'operazione di *roll-back*), per assicurare la proprietà di atomicità [1].

Appare evidente che è necessario poter specificare, in maniera formale, la semantica associata alla cancellazione di *una* singola attività, in termini di interazioni con gli archivi, in modo da garantire una corretta e consistente gestione degli oggetti in essi memorizzati.

Proseguendo nell'analogia con i modelli transazionali, ciò di cui si sente la necessità è simile ai cosiddetti *framework* formali nati in letteratura per permettere la specifica delle politiche transazionali più adatte a modellare transazioni non convenzionali, mantenendo l'astrazione dai dettagli implementativi.

La letteratura scientifica offre almeno due esempi di una certa rilevanza al proposito: il *framework* ACTA [CR91] ed il modello ConTract [WR91]. La possibilità di adattare i risultati di questo tipo di ricerca al linguaggio SLANG è resa ancora più appetibile dal fatto che un framework analogo è in corso di sviluppo presso il Politecnico di Milano all'interno di una tesi per il Dottorato di Ricerca. Questo modello, TANGO [BCF93] è il suo nome, è basato come SLANG sulle reti ER, pertanto sarebbe auspicabile una convergenza tra gli studi al fine di permettere l'intergrazione dei due formalismi.

10.1.4 Alternative multiple

[1] Almeno nel modello tradizionale, in cui le transazioni godono delle cosiddette proprietà "acide" (dall'inglese ACID, Atomicity, Consistency, Isolation, Durability).

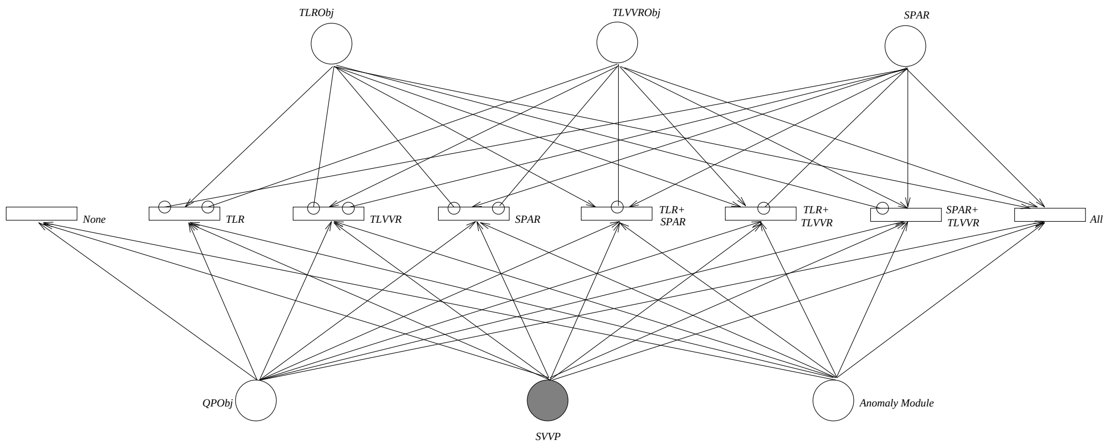


Figura 10.7: La soluzione tradizionale al problema.

Una precedente versione del modello, in particolare nell'attività SetupCCB, conteneva una porzione di rete (un meccanismo di attesa) strutturata come in Fig. 10.6: assumendo la semantica tradizionale delle reti di Petri, la transizione Wait non può scattare fino a che tutti i posti in ingresso non contengono token che soddisfino la guardia. Nella situazione descritta dall'evento, ciò significherebbe che la SVVP non può' cominciare la sua attività finché non sono pronti almeno un membro di tale Funzione, una copia del Piano di Qualità, un TLR, un TLVVR, un riferimento ad un anomalia sospesa o in soluzione (SPAR) ed un Rapporto di Anomalia. Questa non è però la semantica che si vorrebbe dare a tale evento: infatti, per esempio non sempre è necessario avere TLR e TLVVR, ed almeno in un caso (nell'esame del primo Rapporto di Anomalia) l'evento deve scattare in assenza di riferimenti ad altri AR sospesi o pendenti.

Di conseguenza, l'evento che si sta cercando di descrivere necessita di un numero variabile di posti d'ingresso per condizionare lo scatto: questo fatto può essere espresso mediante la definizione attuale di SLANG, ma determina il tracciamento di un gran numero di transizioni ed archi, come mostrato in Fig. 10.7.

In tale figura l'evento originale è stato decomposto in otto eventi, uno per ogni possibile combinazione dei posti di ingresso: così facendo, stiamo usando più eventi "logici" per rappresentare un solo evento "fisico", con conseguente perdita di espressività.

Un primo approccio alla soluzione di questo problema è stata la creazione di una macro che, data in ingresso la configurazione di posti ritenuti necessari allo scatto e quella dei posti "facoltativi" costruisse automaticamente la rete corretta, analogamente a quanto fatto in figura. Questo approccio però non è stato ritenuto soddisfacente in termini di leggibilità e perdipiù funzionava bene soltanto per un sottoinsieme del problema.

Un'altra soluzione possibile consiste nell'inserire la possibilità, all'interno della guardia, di controllare la presenza di token all'interno dei posti in ingresso alla transizione mediante un predicato, permettendo lo scatto della transizione se l'assenza di token si verifica per i posti il cui contenuto è ritenuto non essenziale allo scatto della transizione.

Tuttavia, una soluzione di questo tipo va valutata con attenzione, in quanto modifica alla base la semantica di scatto tipica delle reti di Petri.

10.1.5 Identità dei token

Si è visto come per descrivere alcune attività, quali la gestione della configurazione, sia stato necessario creare un meccanismo sufficientemente elaborato per associare e controllare l'appartenenza di due token ad una stessa elaborazione generata dall'invocazione di un'attività; nel fare ciò si è assunto che questo tipo di lavoro sia a carico del modellatore del processo, ciononostante sarebbe auspicabile che, in future versioni del sistema definitivo, ciò venga fornito dall'ambiente ad esempio sottoforma di *macro* predefinite, alfine di alleggerire il modellatore del processo da questi aspetti "di basso livello" del linguaggio, permettendogli di concentrarsi maggiormente sugli aspetti di processo.

Il gruppo che si occupa dell'implementazione dell'ambiente SPADE [BS93] si sta già occupando anche di questi aspetti.

10.2 Esperienze di modellazione

L'esperienza di modellazione con SLANG, a prescindere dai problemi presentati nella Sez. 10.1 ha mostrato come il linguaggio sia sufficientemente facile ed intuitivo da usare, ammesso che la persona che si accinge ad apprenderne i meccanismi abbia già una sufficiente preparazione riguardo alle tecniche di progettazione orientate agli oggetti e alle reti di Petri. Del resto va puntualizzato che sebbene questo lavoro sia stato svolto da persone dell'ambiente accademico in collaborazione con persone dell'industria, in generale la figura più adatta al ruolo di modellatore di un processo software è un manager o comunque un ruolo dell'azienda esperto delle problematiche di gestione del processo e quindi avvezzo ai concetti chiave dell'Ingegneria del Software, di cui le tecniche menzionate precedentemente fanno ormai parte a tutti gli effetti.

In particolare, sono risultati molto utili i costrutti di progettazione *in-the-large* del modello, quali il costrutto di attività e la possibilità di utilizzare tipi *object-oriented*, in quanto il processo da modellare, sebbene copra soltanto una parte dell'intero processo adottato in Italtel BUTD, è comunque di dimensioni tali che l'assenza di validi costrutti di modularizzazione, incapsulazione ed information hiding avrebbe senz'altro comportato una minore qualità ed un maggiore dispendio di energia nello sviluppo del modello.

Tra l'altro, un grande vantaggio di tali costrutti è costituito dalla loro stretta somiglianza con analoghi costrutti dei linguaggi di programmazione, in generale ben noti a chiunque sia impegnato in un'attività di gestione del processo software.

Sfruttando la presenza di tali costrutti, si è potuto seguire un approccio di tipo *top-down* alla specifica, seguendo grosso modo la seguente sequenza di operazioni:

1. Costruzione dell'interfaccia per un'attività di alto livello.

2. Specifica della sua implementazione, determinando le interfacce di attività eventualmente chiamate da tale implementazione.

3. Selezione delle attività da specificare ulteriormente. Il processo viene iterato fino a che tutte le attività ritenute rilevanti sono specificate al livello di dettaglio ritenuto significativo.

Il significato dell'affermazione contenuta nel terzo punto sta a significare che è possibile lasciare delle attività indefinite a livello di implementazione, fornendo unicamente l'interfaccia: quest'ultima è infatti necessaria all'interfacciamento corretto dell'attivit"a all'interno del resto della specifica. Ad esempio, come già ricordato altrove, l'attività ManageSoftwareReview è definita soltanto parzialmente, in quanto non si è ritenuto utile, in questa fase, una sua modellazione esaustiva: ciononostante, la sua funzione e il suo significato all'interno del modello sono resi chiari dalla corretta definizione della sua interfaccia.

Anche nella specifica dell'implementazione di un'attività, si sono seguiti dei criteri di progetto, privilegiando nelle sequenze operative alcuni aspetti rispetto ad altri; in generale i passi di progetto sono stati i seguenti:

1. *Specifica ad alto livello del controllo all'interno della rete.* Viene definita la rete e le invocazioni di attività al primo livello. Vengono fissati i nomi per i posti, le transizioni e le attività, fornendo così un primo modo altamente informale ed ambiguo per assegnare un qualche significato alla rete.

2. *Definizione dei tipi.* Vengono definiti i tipi di dato che fluiranno all'interno della rete. Eventualmente si possono individuare gerarchie e definire già in questa fase dei metodi, anche se risulta più probabile che questi ultimi vengano definiti in fasi successive della modellazione.

3. *Specifica a basso livello del controllo.* Vengono definite le implementazioni delle attività invocate dalla rete definita al punto 1. In verità si riparte dal punto 1, passando al punto 4 soltanto quando tutte le attività sono state definite.

4. *Definizione formale delle transizioni.* Vengono definite le guardie e le azioni per tutte le reti definite fino a questo momento.

Si noti che questa sequenza di passi non va assunta né come obbligatoria, né come strettamente sequenziale: ad esempio, la definizione dei tipi o delle guardie spesso porta alla scoperta di errori nella definizione del controllo che necessitano una rimodifica delle reti.

Sempre riguardo ai criteri di progettazione della specifica, altre considerazioni vanno tenute presente: in primo luogo, si deve fare attenzione che sebbene il concetto di attività in SLANG sia assai simile al concetto di modulo tipico di un linguaggio di programmazione, e ne condivida le caratteristiche di modularità, incapsulazione ed information hiding, un'attività *non* è soltanto un modulo, ma ha una sua ben precisa semantica dinamica, rappresentata dal costrutto di invocazione di attività. In altre parole, quando si

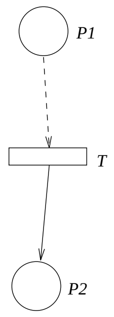


Figura 10.8: Un esempio di uso "pericoloso" degli archi *read-only*.

progetta un'attività si deve fare attenzione al fatto che durante l'esecuzione vi potranno essere più istanze di tale attività (le copie attive) contemporaneamente in esecuzione, a condividere i posti di uscita ed i posti in ingresso: ciò può ad esempio provocare conflitti, o forzare la distribuzione in varie copie attive di compiti che si volevano invece centralizzati, come illustrato nella Sez. 10.1.

Un'altra "trappola semantica" è costituita dall'uso degli archi *read-only*: si osservi ad esempio la Fig. 10.8.

La semantica degli archi read-only, come esposta nel Cap. 3, afferma che il posto d'ingresso, collegato da tali archi ad una transizione, non viene privato di alcuno dei suoi token in seguito allo scatto della transizione. Nella figura, però, pur essendo rispettata la semantica degli archi read-only, la transizione risulta essere sempre abilitata allo scatto, il che è normalmente una situazione altamente indesiderata. Una modellazione corretta potrebbe essere quella mostrata in Fig. 10.9, dove abbinato al posto con l'arco a sola lettura vi è un posto collegato con un arco tradizionale, il quale permette di disabilitare la transizione dopo il primo scatto.

Oltre alle osservazioni viste finora, nella descrizione del controllo bisogna bilanciare opportunamente le informazioni da inserire nella rete e quelle da inserire all'interno delle guardie: la prima risulta più naturale per il lettore, in quanto sfrutta le caratteristiche grafiche del formalismo, viceversa può risultare maggiormente gravosa in termini di stesura della specifica e più problematica quando è necessario operare delle modifiche mentre la seconda, sebbene inizialmente più ostica risulta essere più concisa e maggiormente flessibile ai cambiamenti.

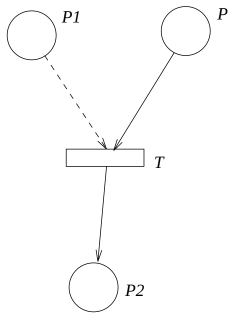


Figura 10.9: Un possibile uso degli archi *read-only* che elimina il problema degli scatti multipli.

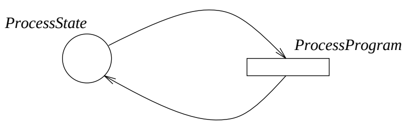


Figura 10.10: Una descrizione teoreticamente accettabile ma praticamente inutile di processo.

Analoghe considerazioni valgono per le scelte su cosa inserire nelle reti e cosa in metodi associati ai dati.

In ambedue i casi è opportuno puntualizzare che esistono due estremi teorici: uno è rappresentato da un processo completamente specificato in termini di reti, in cui non è necessario avere altra informazione all'interno delle guardie e delle azioni che la presenza di token ed il loro trasferimento; l'altro è caratterizzato dalla situazione di Fig. 10.10 in cui la descrizione del processo in termini di reti è elementare e si può supporre di avere come dato l'intero processo ed operare su di esso mediante un unico metodo invocato nella transizione, il quale contenga tutte le informazioni necessarie all'esecuzione del processo.

La scelta di come collocarsi all'interno di questo spettro di possibilità è compito del modellatore del processo, il quale dovrà fare valutazioni basate sulle considerazioni esposte precedentemente.

Parte V

Conclusioni

Capitolo 11

Conclusioni e sviluppi futuri

Questo lavoro di tesi ha presentato la formalizzazione di una parte significativa del processo software attualmente impiegato nella divisione Business Unit Defense Telecommunications Software dell'Italtel mediante il linguaggio per la modellazione del processo software SLANG, componente essenziale dell'ambiente di supporto al processo software SPADE, attualmente in corso di sviluppo presso CEFRIEL e Politecnico di Milano.

Il processo di formalizzazione si è incentrato principalmente sul Manuale del Sistema di Qualità Software, impiegato comunemente nella pratica aziendale come unica definizione ufficiale del processo, integrandone vari aspetti attraverso frequenti colloqui con l'esperto del processo, Ing. Maurizio Loi.

Il modello SLANG del processo è stato descritto, fornendo sia il documento originale e completo che costituisce il risultato finale e primario della modellazione sia una visione a più alto livello focalizzata sull'utilizzo dei meccanismi di SLANG all'interno del modello e sullo sviluppo di tecniche *ad hoc* per la risoluzione dei problemi di modellazione. Inoltre, sono stati raccolte indicazioni su come potrebbero essere organizzati futuri lavori di modellazione analoghi a questo, sulla base dell'esperienza maturata in questo lavoro, e sono stati identificati una serie di aspetti a cui prestare attenzione quando ci si accosti per la prima volta a questo tipo di attività.

Sono stati discussi i problemi relativi al linguaggio, evidenziando come SLANG necessiti ancora di qualche raffinamento per poter essere usato in maniera veramente efficace per un lavoro di reale modellazione, ma mostrando anche che nella sostanza i concetti ed i meccanismi di fondo del linguaggio colgono aspetti essenziali e fondamentali delle problematiche di formalizzazione del processo produttivo del software, consentendone comunque, anche a questo stadio di definizione, un impiego più che proficuo, come mostrato dai risultati di questo lavoro di tesi.

Sono stati discussi anche i problemi riscontrati nel Manuale di Qualità, evidenziando sia gli aspetti non corretti di tale descrizione, sia indicazioni per il miglioramento di essa, cercando di far convergere il più possibile quest'ultima, attraverso l'uso dello strumento formale, verso la realtà del processo. A questo proposito, risultati particolarmente significativi sono stati la ridefinizione degli automi che descrivono i Rapporti di Anomalia e le Richieste di Modifica, insieme con la formalizzazione, benché parziale, degli aspetti di allocazione delle risorse umane. È importante puntualizzare che i nostri suggerimenti in tal senso sono stati accolti e verranno ampliati ed inglobati in future versioni del Manuale di Qualità.

La sperimentazione ha dunque mostrato come il lavoro di modellazione di un processo software apporti dei benefici per l'organizzazione che vanno aldilà del risultato finale, cioé del modello: in questo caso, ad esempio, le modifiche illustrate sono state identificate quando ancora il modello era allo stato embrionale. Tuttavia, anche il modello in sé può suggerire ampi margini di miglioramento: a titolo di esempio, attraverso un supporto al processo quale quello che potrà essere fornito dall'ambiente SPADE, mediante l'esecuzione del nostro modello si potrebbe essere in grado di automatizzare completamente i compiti di una delle Funzioni presenti in Italtel BUTD, e cioè della Sottofunzione Gestione Modifiche e Rilasci, consentendo quindi un'ottimizzazione dell'allocazione delle risorse all'interno del processo.

Gli sviluppi futuri di questo lavoro comprenderanno:

- A brevissimo termine, la verifica di parti significative del modello mediante il prototipo dell'ambiente SPADE, in modo da testare le funzionalità di quest'ultimo e la consistenza della specifica.

- L'estensione del modello mediante la formalizzazione di altre attività significative all'interno del processo adottato in Italtel BUTD, in modo da creare una serie di attività che andranno poi integrate in un'unica specifica.

- La realizzazione di ulteriori esperienze simili a quella descritta, in altre realtà industriali e non, per ottenere una più ampia capacità di valutazione delle potenzialità del formalismo e la possibilità di definire una metodologia.

- Il raffinamento dei costrutti del linguaggio in maniera coerente alle indicazioni provenienti da questo lavoro, in modo da renderlo effettivamente e proficuamente utilizzabile nella modellazione del processo reale.

Parte VI

Appendici

Appendice A

Il modello SLANG completo

A.1 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.

A.2 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);

% 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;
% 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.
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.
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);

% 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;
% 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
% 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;

% 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
% (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 management. A fac-simile is shown in Table A.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.

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:

Tabella A.1: A fac-simile of Italtel Anomaly Report form, the main form used within Software Anomaly Management.

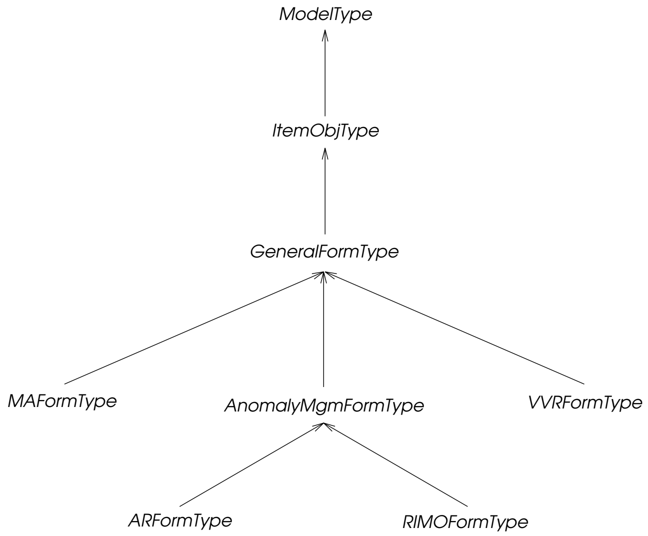


Figura A.1: The type hierarchy used to describe configuration items and forms.

All these considerations have been concretized in the object type hierarchy shown in Fig. A.1.

`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.

- `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.

- `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 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.

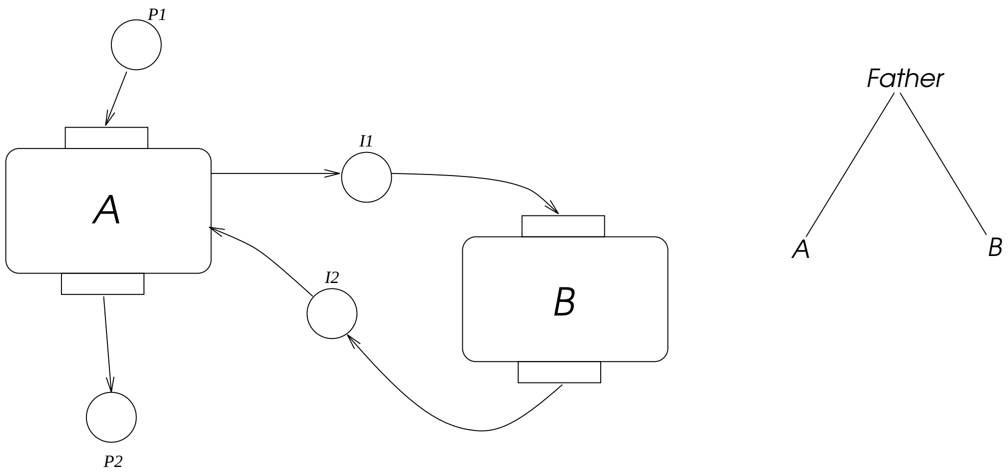


Figura A.2: Access conflict between activities at the same level in the static tree.

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 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 A.2 and A.3 show typical situations allowed by such rules but causing ambiguities.

In Fig. A.2 the activity B performs some kind of service that is used by

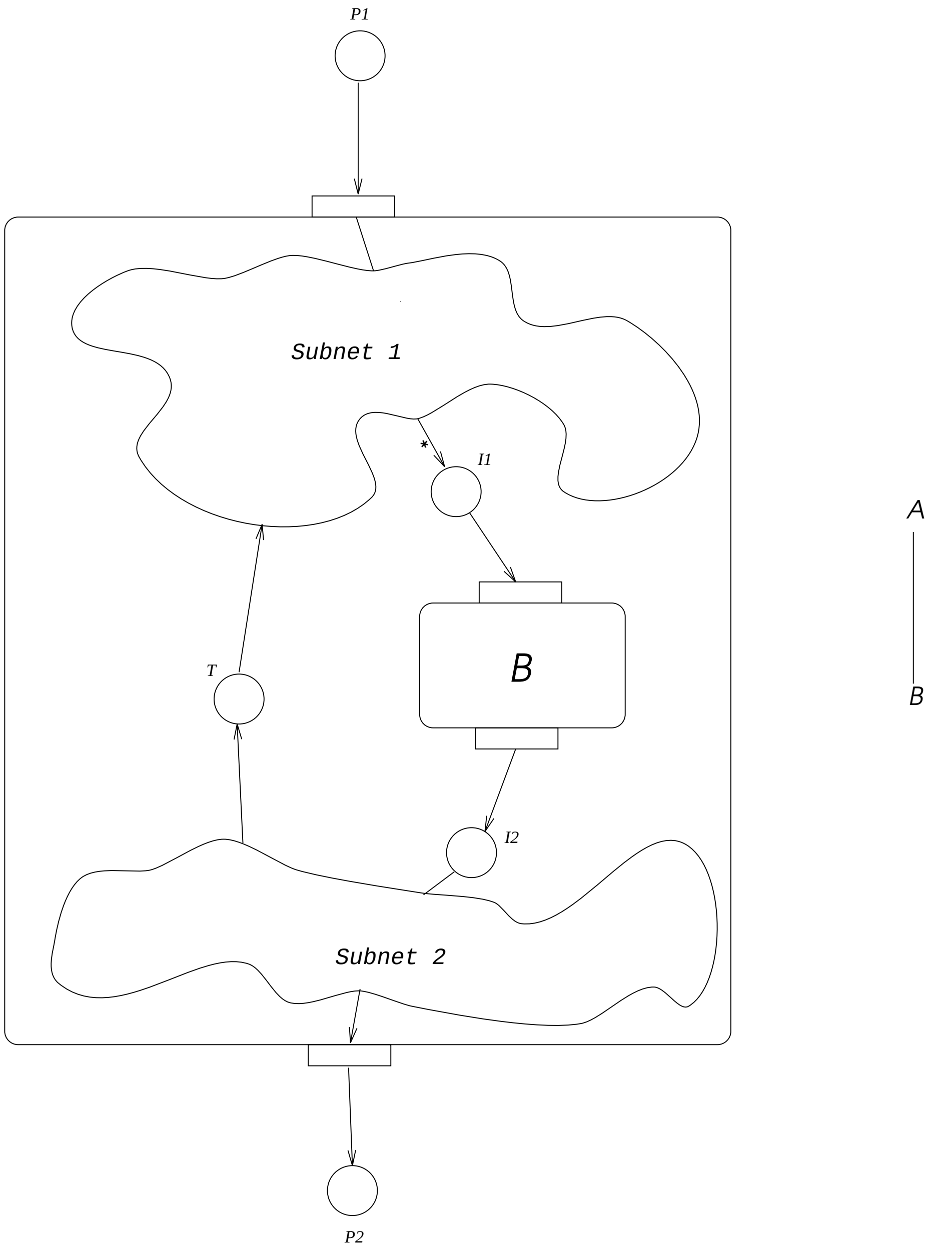


Figura A.3: Access conflicts between activities at different levels, due to multiple threads in net execution.

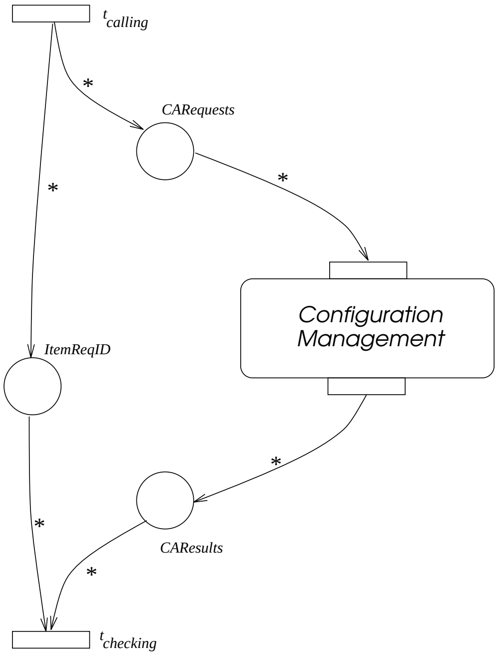


Figura A.4: An example to explain access policy to Configuration Management used within our specification.

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. A.3 the ambiguity is more subtle: this time the activity B is a 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. A.4

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 CAResults 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. A.4, 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 A.2 explains how fields must be used to request the desired operations:

By converse, Table A.3 shows how output fields are filled by Configuration Management activity.

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

Tabella A.2: How to fill CARequests fields to issue Configuration Management requests.

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:

- `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.

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

Tabella A.3: How Configuration Management fills CAResults fields after the execution of requested operations.

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.

A.3 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. A.5. The root of the hierarchy contains the following built-in tool description on which rely every user-defined description:

```
GeneralToolType SUBTYPE OF ModelType
```

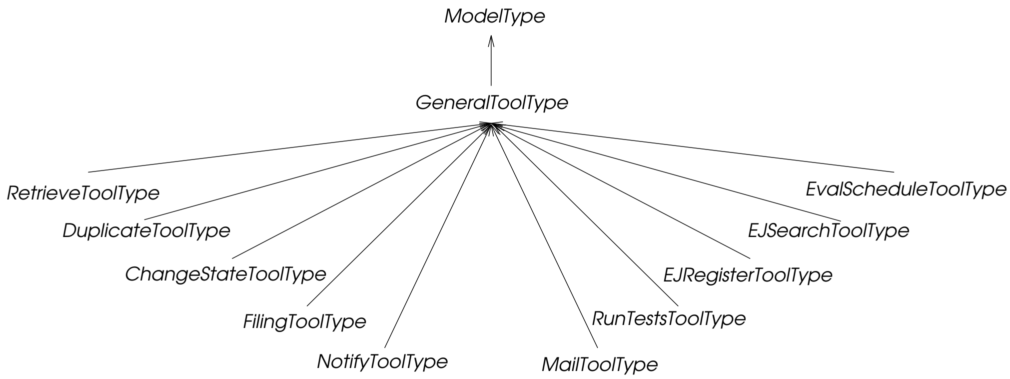


Figura A.5: Object hierarchy for tools.

```
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:

- 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.
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.
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.

A.4 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.

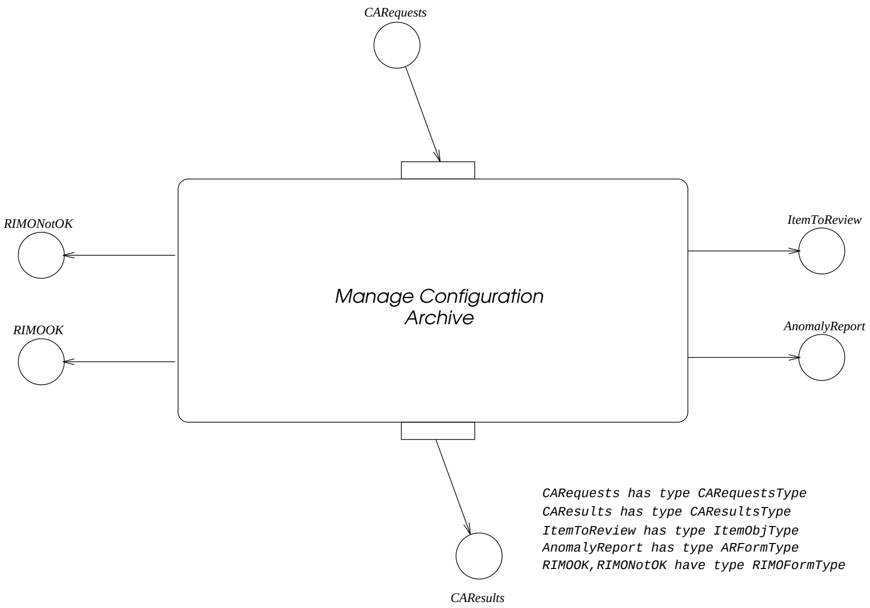


Figura A.6: High-level view of the main activity for Software Configuration Management.

- 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. A.6.

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.

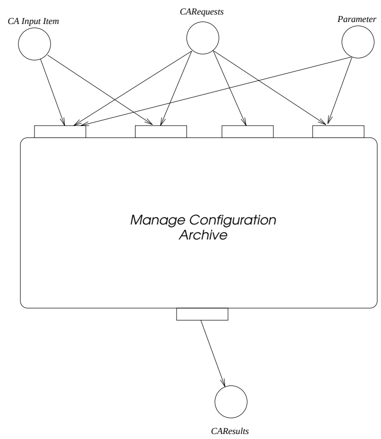


Figura A.7: An alternative solution for the high-level description of Software Configuration Management.

This is a design choice, i.e. an alternative idea is the one showed in Fig. A.7: 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 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. A.8.

The higher-level activity description of Software Configuration Management can be exploded to show inner implementation as done in Fig. A.9: this

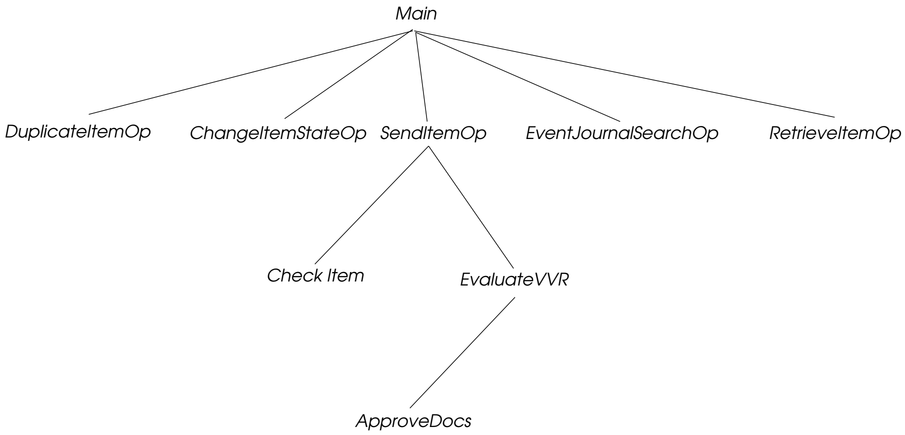
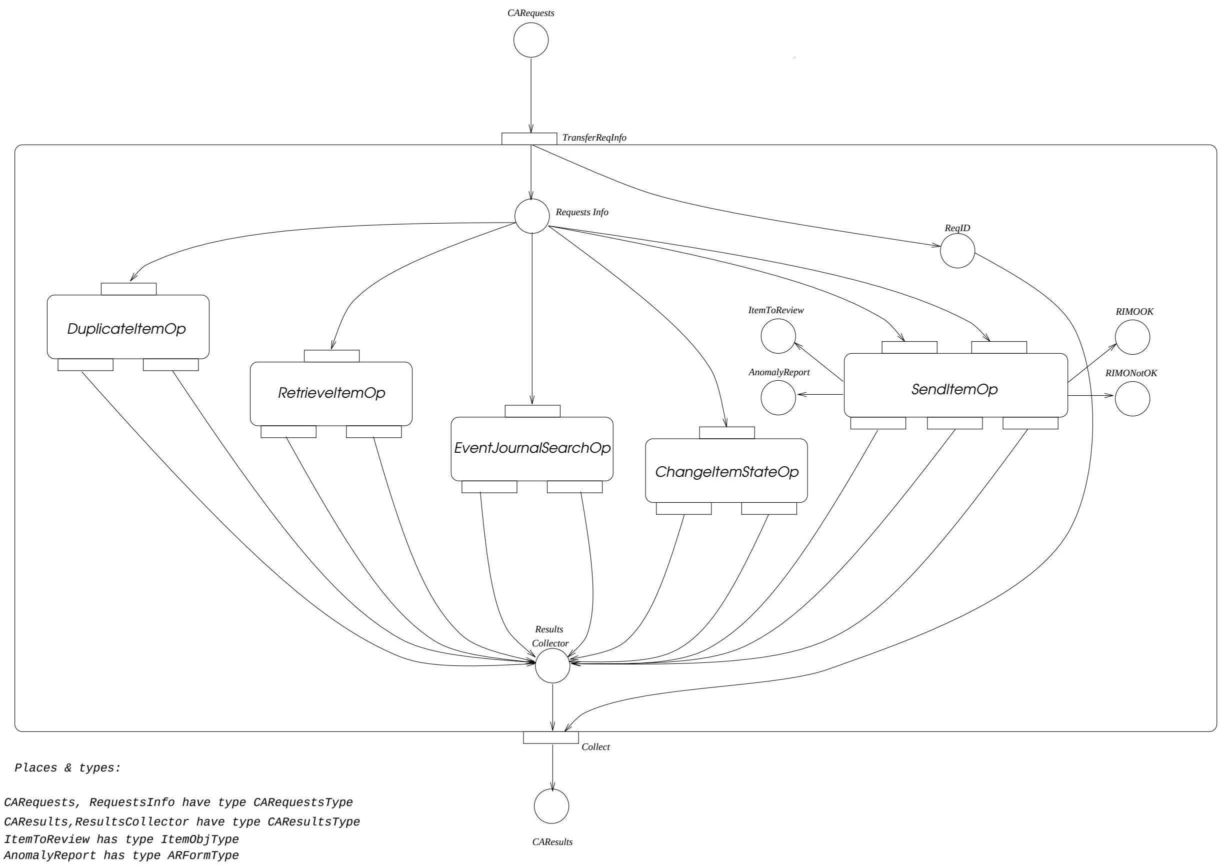


Figura A.8: The activity static tree relative to software configuration management.

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.

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 A.2, 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:



Places & types:

CARequests, RequestsInfo have type CARequestsType
CAResults,ResultsCollector have type CAResultsType
ItemToReview has type ItemObjType
AnomalyReport has type ARFormType
ReqID has type ReqIDType
RIMOOK,RIMONotOK have type RIMOFormType

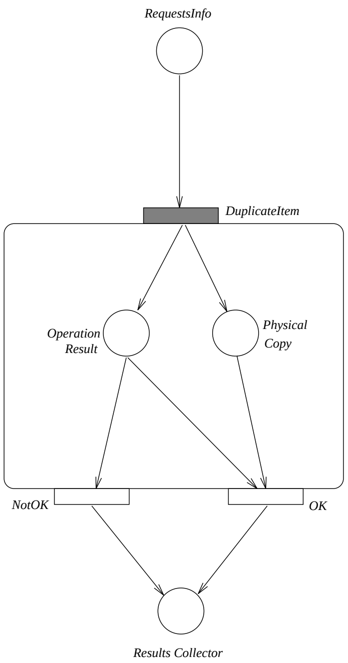
Figura A.9: **ManageConfigurationArchive Activity.** The implementation of Software Configuration Management higher-level activity.

```
% Transfers a token request into the activity body
% and keeps track of request identifier.
EVENT TransferReqInfo(CARq; RI,Rid)
GUARD
CARq IN CARequest;
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. A.10, 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
```




```
Places & Types

RequestsInfo has type CARequestsType
OperationResult has type ConditionType
PhysicalCopy has type ItemObjType
ResultsCollector has type CAResultsType
```

Figura A.10: **DuplicateItemOp Activity.** The implementation.

```
SUCH THAT
RI.OpReq=Duplicate;
ACTION
% Tool invocation.
CopyTool.Invoke(RI.ItemRef,RI.Destination);
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. A.11 is identical to the previous, except for the black transition.

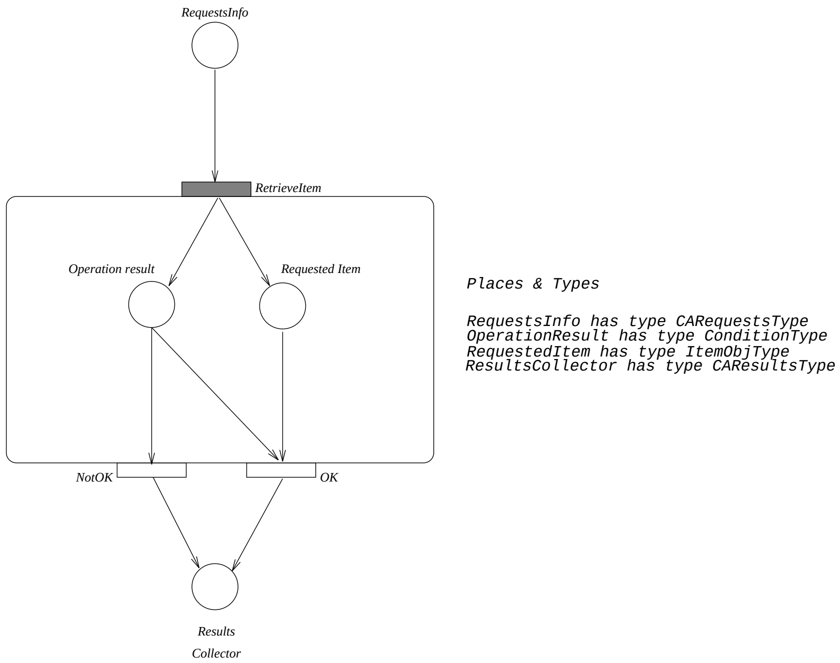


Figura A.11: **RetrieveItemOp Activity.** The implementation.

It has the same behavior, formally described by the following specification:

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

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. A.12: 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
```

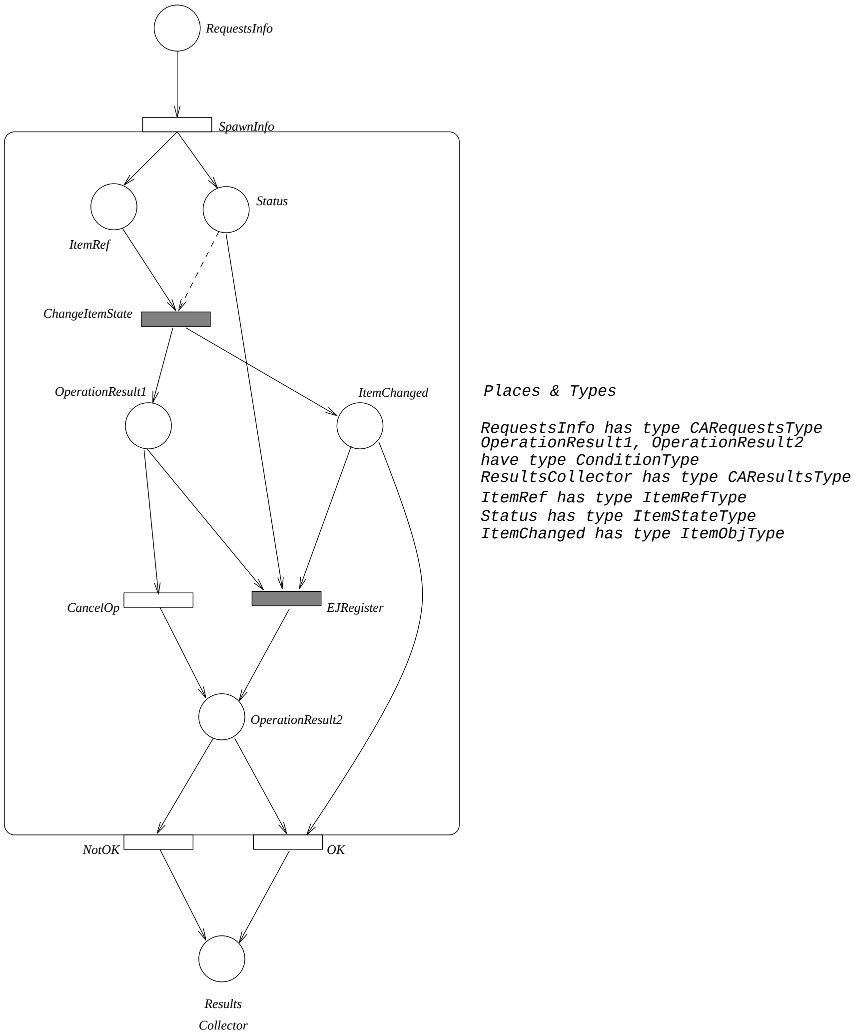


Figura A.12: **ChangeItemStateOp Activity.** The implementation.

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

SendItem Activity, whose net is described in Fig. A.13 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.
  - 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'.
EVENT NoVVNeeded(IA; I)
GUARD
IA IN ItemAccepted
SUCH THAT
IA.Kind=DocNotVV;
ACTION
I:=IA;
I.Status:=Approved;
```

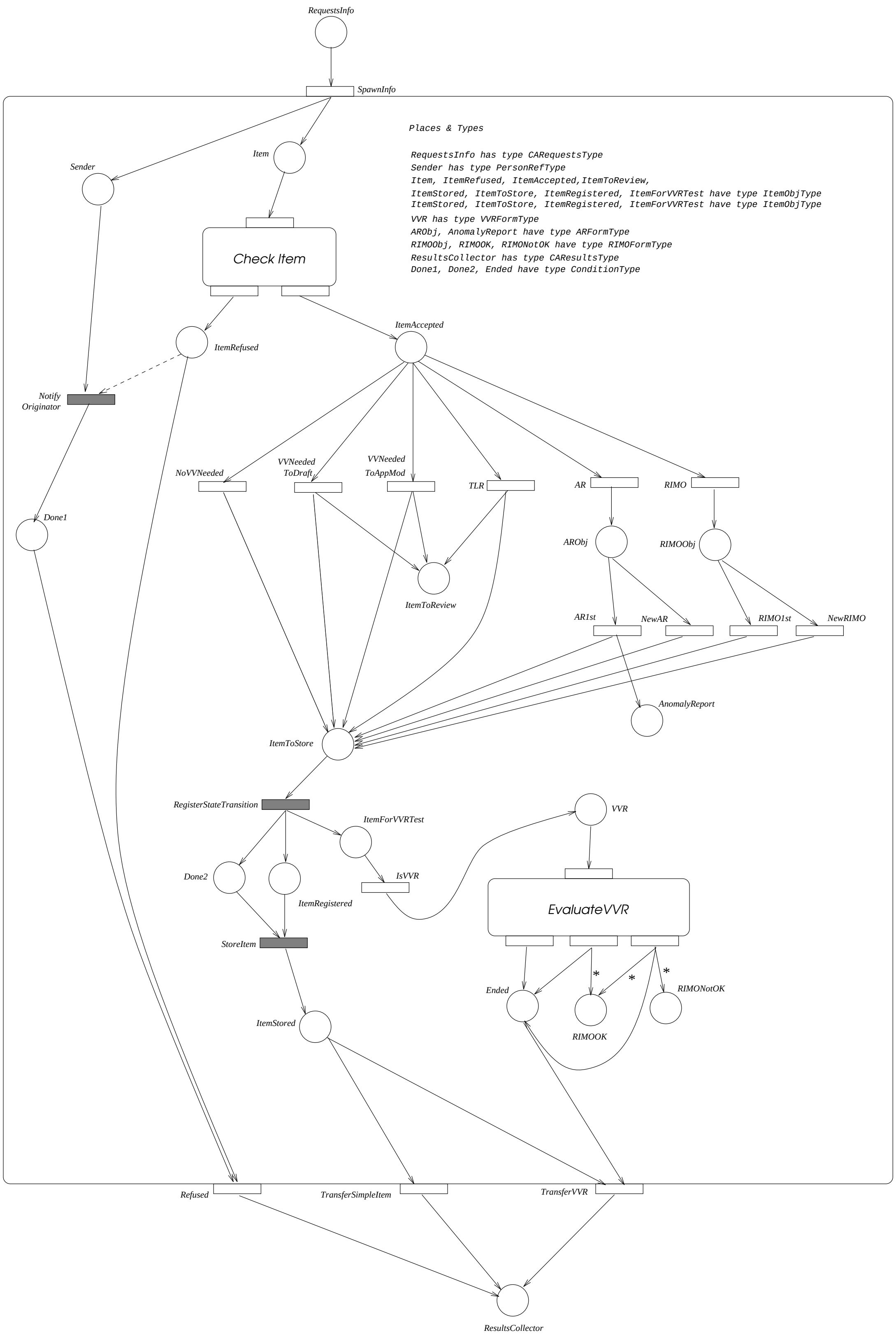


Figura A.13: **SendItemOp Activity.** The implementation.

```
% 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;

% 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;
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
```