Université de Montréal

Implantation du protocole de signalisation ATM en utilisant une spécification SDL

par

Roxana-Irina Marcoci

Département d'informatique et de recherche opérationnelle

Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures

en vue de l'obtention du grade de

Maître ès sciences (M.Sc.)

en informatique

Août, 1997

Page d'identification du jury

Université de Montréal

Faculté des études supérieures

Ce mémoire intitulé:

Implantation du protocole de signalisation ATM en utilisant une spécification SDL

présenté par

Roxana-Irina Marcoci

a été évalué par un jury composé des personnes suivantes:

Rachida Dssouli..... présidente du jury

Gregor von Bochmann..... directeur de recherche

Rudolf Keller..... membre du jury

Mémoire accepté le 15 décembre 1997

Sommaire

La spécification formelle joue un rôle important dans le cycle de développement d'un système. Elle peut être utilisée pour la simulation du modèle du système permettant le dépistage d'éventuelles erreurs de spécification avant le processus d'implantation. Elle représente aussi un point de départ pour le développement semi-automatique de cas de test qui vérifient le comportement défini dans la spécification. Également, une spécification peut être traduite dans un langage de programmation, comme C, pour obtenir un prototype d'implantation.

L'objectif du présent projet est d'expérimenter l'utilisation d'une spécification formelle dans le développement du protocole de signalisation ATM à l'interface utilisateur-réseau (UNI) du côté de l'utilisateur. La référence de base est le document "ATM User-Network Interface Specification", version 3.1 de ATM Forum.

Puisque SDL représente un standard utilisé dans la description de nombreux protocoles de communication, il a été choisi en tant que langage de spécification dans le présent projet. Il existe au moins deux outils commerciaux associés à SDL, à savoir SDT et GEODE. Celui disponible à l'Université de Montréal lors de la réalisation de notre projet était ObjectGEODE, la version orientée objet de GEODE.

Dans une première étape nous avons réalisé une spécification SDL du protocole standard, sans aucun complément spécifique à l'implantation, mais assez complète pour réaliser la validation par simulation. La spécification résultante a été utilisée comme base pour la génération automatique de code C. Le code généré par l'outil a été adapté à des besoins spécifiques. Par ailleurs, du fait des limitations du langage SDL qui ne permet pas la déclaration de pointeurs et pour des raisons d'optimisation en termes de rapidité, nous avons été amenés à concevoir du code C pour la représentation de certaines sections du protocole (par exemple le codage et le décodage). Ce code C, nous l'avons ensuite intégré dans le code généré par l'outil afin d'obtenir un prototype d'implantation.

Table des matières

Liste des tableaux

Liste des figures

Liste des sigles et abréviations

AAL	ATM adaptation layer
ADT	Abstract data type
ANSI	American National Standard Institute
AP	Application process
ASCII	American standard code for information interchange
ATM	Asynchronous transfer mode
B-ICI	B-ISDN intercarrier interface
B-ISDN	Broadband integrated servics digital network
CBR	Constant bit rate
CCITT	Comité Consultatif International pour le Télégraphe et le Téléphone
CLP	Cell loss priority
CNET	Centre National d'Études de Télécommunications
CP	Common part
CPCS	Common part convergence sublayer
CS	Convergence sublayer
EBCDIC	Extended binary coded decimal interchange code
EFSM	Extended finite state machin
EFSM	Extended finite state machine
FDT	Formal description techniques
FSM	Finite state machin
GFC	Generic flow control
HEC	Header error control
ILMI	Interim local management interface
INI	Inter-Network Interface
ISO	International Standard Organization
ITU	International Telecommunication Union
MSG	Message sequence chart
NNI	Network Node Interface

OSI	Open System Interconnection
PDU	Protocol data unit
PID	Proccess identifier
PM	Physical Media
PTI	Payload type identifier
QoS	Quality of service
SAAL	Signalling ATM adaptation layer
SAP	Service acces point
SAR	Sagmentation and reassembly
SDL	Specification and description language
SSCF	Service-specific connexion oriented protocol
SSCOP	Service-specific convergence sublayer
TC	Transmission convergence
TCP	Transmission control protocol
UNI	User-Network interface
VBR	Variable bit rate
VC	Virtual channel
VCI	Virtual channel identifier
VP	Virtual path
VPC	Virtual path connection
VPCI	Virtual path connection identifier
VPI	Virtual path identifier

À mes parents

Préface

Cadre du travail

Le présent projet à été réalisé dans le cadre du projet de recherche EPAC qui regroupe les compagnies Eicon Technology Corporation, Positron Fiber Systems, Technologies Innovations AIKS Inc. et le Centre de Recherche Informatique de Montréal (CRIM).

L'objectif du projet EPAC est d'expérimenter l'utilisation des spécifications exécutables dans le développement du protocole de signalisation ATM.

De l'énoncé des besoins à l'exploitation, un système, en particulier un protocole de communication, passe par plusieurs étapes de développement.

Généralement, la description d'un système est réalisée en langage naturel, en incluant, éventuellement, d'autres formes de structuration de l'information: tableaux, diagrammes, listes. Bien que la spécification en langage naturel ait l'avantage d'être facilement compréhensible par les lecteurs, elle présente des inconvénients: elle contient des ambiguïtés, est sujette à des diverses interprétations et il est difficile de vérifier qu'elle est en mesure de couvrir toutes les situations qui peuvent surgir ou que le comportement du système est celui attendu.

Une spécification formelle décrit un système dans un langage possédant une syntaxe et une sémantique bien établies. Elle présente l'avantage majeur de pouvoir être traitée par la machine permettant ainsi l'automatisation, totale ou partielle, de certaines activités du développement d'un protocole, par exemple, la validation, l'implantation et le test.

Il est bien connu que la correction d'une erreur devient plus coûteuse à mesure qu'on avance dans le cycle de développement. L'utilisation de la spécification exécutable pour la simulation du comportement du système permet le dépistage et la correction d'éventuelles erreurs avant l'étape d'implantation.

La possibilité de générer une grande partie du code de l'implantation à partir de la spécification formelle d'un système représente un autre atout important dans l'utilisation des spécifications formelles. Néanmoins, le code généré doit être intégré dans le contexte logiciel réel et nécessite, des fois, des adaptations.

La série de tests utilisés dans la validation d'une spécification exécutable peut être construite, de façon automatique, selon différentes méthodes. Cependant, certains aspects, comme la variation des paramètres, peuvent ne pas être couverts par les cas de test générés et une intérvention manuelle est alors nécessaire. L'automatisation peut également intervenir dans l'analyse des résultats des tests.

L'objectif du présent projet est de réaliser une implantation du protocole de signalisation ATM en utilisant une spécification formelle, à savoir SDL. Une comparaison de cette implantation avec celle développée par EICON directement du standard du protocole, permet d'évaluer la pertinence de l'utilisation des méthodes formelles dans différents cas de développement de systèmes. Le développement des cas de tests applicables à une spécification exécutable du protocole de signalisation ATM constitue le sujet d'un autre projet de maîtrise[1].

Structure du présent mémoire

Le premier chapitre donne quelques notions de téléinformatique, nécessaires à la compréhension du concept d'ATM et des protocoles de communication.

Le chapitre 2 décrit le processus de développement d'un système, en mettant l'accent sur les particularités relatives au développement des protocoles de communication; différentes techniques de spécification de protocoles sont présentées. Nous abordons aussi l'aspect de l'automatisation de différentes étapes du développement des protocoles.

Le chapitre 3 est structuré en deux sections. La première section présente le concept d'ATM, tandis que la deuxième décrit le protocole de signalisation ATM.

---

[1] Hristov D., *Test development for ATM Signalling Protocol*, Master Thesis, McGill University, en préparation

Le chapitre 4 est destiné à la spécification SDL du protocole de signalisation. Il débute avec une brève présentation du langage SDL. Ensuite, aprés avoir décrit l'approche utilisée, nous expliquons la spécification SDL du protocole (présente dans l'annexe 1). Les aspects relatifs à la simulation sont exposés dans la dernière section du chapitre.

Le chapitre 5 est consacré à la construction du code C représentant le protocole de signalisation. Tout d'abord, nous expliquons les étapes qui conduisent à l'obtention du code C à partir de la spécification SDL. La stratégie de génération de code est également présentée. Une section du chapitre porte sur le code C que nous avons écrit séparément et que nous avons intégré au code C généré par l'outil ObjectGEODE. Des commentaires et une évaluation des performances complètent ce chapitre.

Nous terminons ce mémoire par une conclusion.

1. Éléments de téléinformatique

1.1 Communication entre ordinateurs

Bien que dans beaucoup de circonstances les ordinateurs soient utilisés pour travailler dans un mode autonome, il est souvent nécessaire qu'ils interopèrent et échangent des données.

Un **réseau** est un ensemble d'équipements informatiques interconnectés qui assure le transport des données.

Un **système distribué** est un ensemble d'ordinateurs, qui travaillent simultanément et qui collaborent à la réalisation d'une tâche. Contrairement au réseau, l'existence de plusieurs ordinateurs est invisible à l'utilisateur. Cette transparence est assurée par le système d'exploitation. On peut donc considérer qu'un système distribué est un cas particulier de réseau. Un réseau est un système distribué ou non selon la façon dont il est utilisé [20].

Les applications qui impliquent deux ou plusieurs ordinateurs exigent une fonction de communication convenable, dont la complexité dépend de la nature des applications, du nombre et des types des ordinateurs impliqués, ainsi que de la distance physique qui les sépare. Par exemple, s'il faut transférer un fichier de données d'un ordinateur à un autre de même type qui se trouvent dans la même pièce, la fonction de communication sera plus simple que si le fichier devait être transféré entre deux ordinateurs de type différent localisés dans des sites éloignées. Normalement, dans ce dernier cas, le réseau est impliqué (Figure 1).

Le type d'application détermine dans une large mesure l'organisation du système, ainsi que la nature du trafic qui doit être écoulé par le réseau de communication.

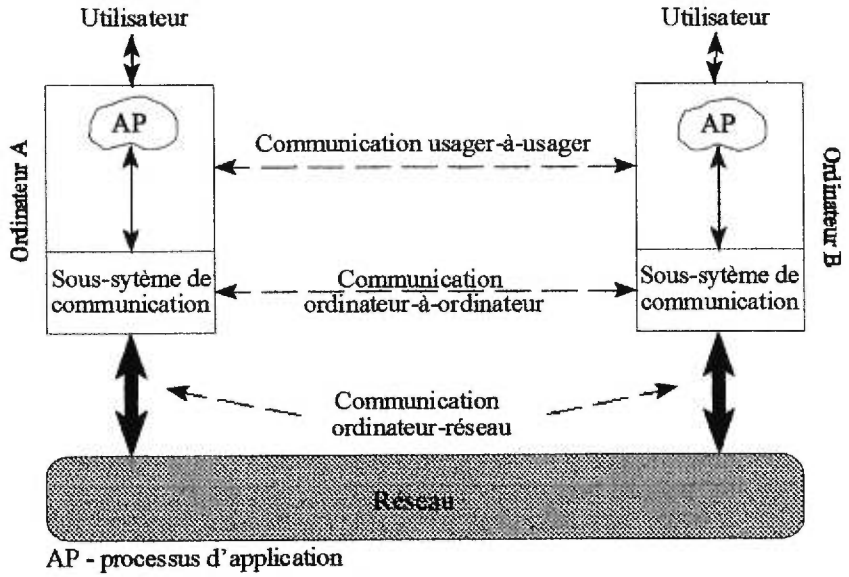


Figure 1: Communication entre deux ordinateurs

1.2 Modes de communication

Il existe deux grands modes de communication:

La **commutation de circuits** où on établit un chemin physique entre les deux utilisateurs avant le début de l'échange de données. Cette technique présente l'avantage de limiter l'utilisation des voies de communication aux seules périodes d'activité des utilisateurs. Elle est donc beaucoup plus adaptée au trafic téléphonique qu'au trafic téléinformatique, car dans beaucoup d'applications téléinformatiques les utilisateurs peuvent rester connectés au réseau, alors qu'ils n'échangent que des messages courts à des intervalles relativement longs.

La **commutation de paquets** où les données sont découpées en unités (généralement de l'ordre d'une centaine d'octets) appelées **paquets** qui sont acheminés sur le réseau et réassemblés à la destination. On distingue deux techniques de communication de paquets:

- la technique **datagramme**: chaque paquet possède l'adresse de son destinataire et est acheminé, de noeud en noeud, indépendamment des autres. Chaque paquet peut suivre un chemin différent sur le réseau et il n'est pas garanti que les messages arrivent à la destination dans l'ordre de leur l'émission. Les paquets doivent donc transporter une information de séquence, nécessaire au réassemblage.

- la technique **circuit virtuel** ou **voie logique**: on établit un chemin logique entre les utilisateurs avant l'échange de données. Tous les paquets de la communication (connexion) suivent le même chemin et donc la séquentialité est assurée.

1.3 Multiplexage

Le **multiplexage** consiste à acheminer les données provenant d'utilisateurs différents sur le même support physique. On distingue deux techniques de multiplexage:

- **multiplexage en fréquence** (ou spatial): chaque utilisateur utilise une bande de fréquence particulière sur le support.
- **multiplexage temporel**: le débit offert par le support est partagé dans le temps entre les utilisateurs. Ce partage peut se faire suivant deux modalités:
  - **multiplexage temporel synchrone**: des tranches de temps fixes sont affectées cycliquement aux différents utilisateurs;
  - **multiplexage temporel asynchrone**: les utilisateurs ont accès au canal quand ils en ont besoin. Une fois l'accès obtenu, un usager est le seul utilisateur du canal tout au long de la durée de la transmission.

1.4 Modèle de référence de l'ISO

Afin de résoudre les problèmes de compatibilité qui se posent pour l'interconnexion d'équipements conçus par des constructeurs différents, un effort de normalisation internationale a été entrepris par ISO (*International Standards Organization*). L'architecture développée, appelée **modèle de référence de base pour l'interconnexion de systèmes ouverts**, et plus connue sous le sigle **OSI** *(Open System Interconnection Basic Reference Model)* est une structure qui stratifie les fonctions de communication en sept couches (Figure 2).

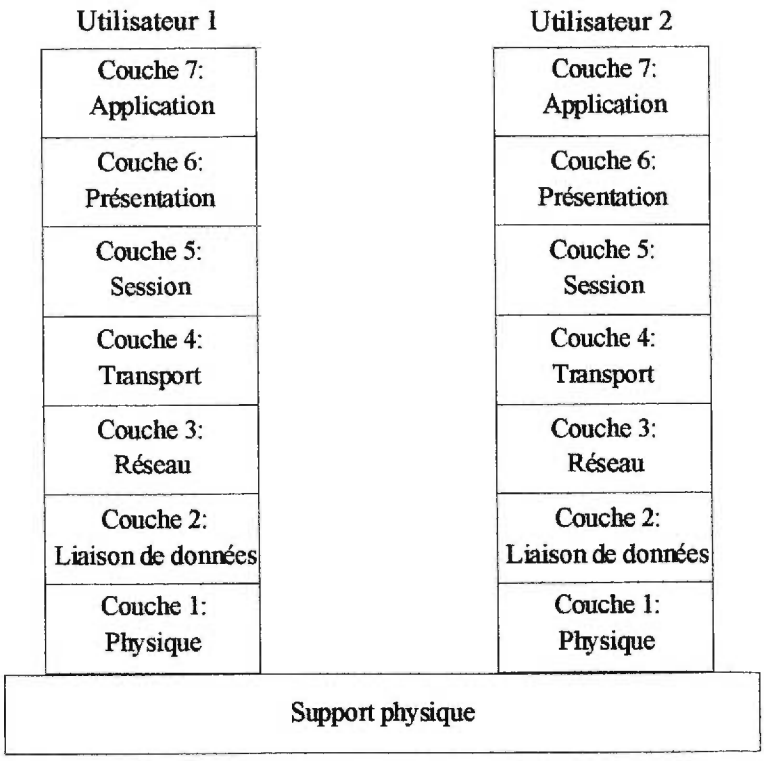


Figure 2: Modèle OSI

Les sept couches peuvent être divisées en deux infrastructures:

1. Les couches basses (1, 2 et 3) fournissent des services bout-en-bout, responsables du transfert de données, effectué généralement en série par bits.
2. Les couches supérieures prennent en charge les fonctions liées à la communication entre les applications.

Le **support physique** est l'élément matériel qui assure physiquement le transport des données. Il peut être le câble, la fibre optique, les ondes électromagnétiques (pour les communications sans fil).

La **couche physique** assure l'interface mécanique et électrique avec le support physique. Elle ne concerne que la transmission transparente des bits sur le support physique.

La **couche de liaison de données** assure l'acheminement de blocs de données (trames, LPDU, *Link Protocol Data Unit*) sur une liaison. Elle détecte et corrige des erreurs pouvant se produire pendant la transmission.

La **couche réseau** assure l'acheminement - le routage - des données de la source à la destination sous forme de paquets (NPDU, *Network Protocol Data Unit*) à travers le réseau, indépendamment du réseau sous-jacent ou de sa topologie. Les paquets peuvent traverser des noeuds intermédiaires. La couche réseau règle aussi les problèmes de congestion et d'interconnexion de réseaux.

La **couche transport** réalise le contrôle de bout en bout (les messages des émetteurs doivent parvenir à leurs destinataire), en cachant au couches supérieures le réseau sous-jacent. Elle est responsable de la fiabilité et du multiplexage du transfert des données. À ce niveau a lieu le réassemblage des paquets entrants (TPDU, *Transport Protocol Data Unit*) en messages.

La **couche session** fournit à deux entités paires de la couche application (à travers la couche présentation) les outils de gestion et de synchronisation du dialogue. Elle fournit des services de gestion et de contrôle du flux de données.

La **couche présentation** est dédiée à la présentation des données échangées (par exemple, en format ASCII ou EBCDIC).

La **couche application** est le dernier responsable de la gestion des communications entre les applications. Elle fournit à l'utilisateur les services qui lui permettent d'exploiter le système téléinformatique.

Toutes les couches ne sont pas toujours présentes, en particulier les couches hautes. Les fonctions correspondantes sont soit inutiles, soit prises en charge par les applications.

Les données émises par un utilisateur traversent toutes les couches présentes qui ajoutent des informations de service et peuvent découper ou regrouper les données. L'opération inverse a lieu à la réception.

1.5 Organisation des communications avec le modèle OSI

Chaque couche de niveau n utilise les services de la couche immédiatement inférieure, de niveau $n-1$, pour fournir des services à la couche immédiatement supérieure, de niveau $n+1$, qui est son utilisateur. La frontière entre deux couches adjacentes représente

leur **interface**. Les **entités** sont les éléments actifs d'une couche. Les interactions entre une entité qui fournit un service et son utilisateur s'appellent **primitives de service**. Elles sont invoquées à des **points d'accès à des services n** ou **n-SAP** *(n-Service Access-Point)*.Généralement, les primitives de service contiennent des paramètres représentant de l'information de contrôle ou des données utilisateur *(user data)*.

Les entités de même niveau peuvent échanger des informations en établissant une **connexion**. L'ensemble des règles qui régissent la communication entre deux entités de même niveau *n* s'appelle **protocole n**. Leurs interactions s'appellent **unités de données de protocole** ou **PDU** *(Protocol Data Unit)*.

Habituellement, les primitives de service sont regroupées selon la phase d'exécution d'un protocole. Par exemple on regroupe les primitives pour l'établissement de connexion sous le nom *Setup*. À l'intérieur d'un groupe, les primitives sont distinguées par des suffixes ayant différentes significations:

request - utilisé quand un couche demande un service à la couche inférieure;

indication - utilisé par une couche pour annoncer à son utilisateur toute activité relative au service qu'elle fournit;

response - utilisé par la couche utilisatrice de service pour l'acquittement de la réception d'une primitive de type *indication*;

confirmation - utilisé par la couche fournisseur de service pour confirmer que la phase est achevée.

2. Développement d'un système

2.1 Cycle de vie d'un système

On définit un **système** comme étant un ensemble d'éléments, ses composantes, en interaction, organisés en fonction d'une finalité.

Le **processus de développement** d'un système représente le chemin entre la reconnaissance des besoins du système (la définition du problème) et le produit final qui les satisfait (une solution).

Il existe plusieurs approches dans la description du processus de développement des systèmes, dont la plus connue est le cycle de vie. Celui-ci apparaît sous une variété de formes dans la littérature de génie logiciel. Le **cycle de vie** d'un système décrit le processus de production du système comme étant divisé en une séquence de **phases** (stages), elles aussi divisées, éventuellement, en sous-phases. Cette approche est apparue au début des années 50.

Le modèle de cycle de vie plus connu est le **modèle en cascade** (*waterfall*) dont la particularité tient à la présence des boucles de retour (*feedback loops*). Plus explicitement,

• chaque phase se termine par une vérification ou par une validation qui est destinée à éliminer le plus possible les anomalies et les défauts dans les composantes ainsi que sur l'ensemble des produits réalisés durant cette phase;

• chaque phase traitée doit éviter les retours sur les phases précédentes en les limitant à un retour sur la phase immédiatement antérieure.

Les phases principales du cycle de vie d'un système sont:

• l'analyse des besoins (*requirement analysis*) - identifie les nécessités du système;

• la spécification - dont l'objectif est de préciser de façon non-ambiguë ce que le système fait pour satisfaire les besoins;

• le *design* - décrit la façon dont le système réalise ses tâches pour répondre à la spécification. Cette phase comporte plusieurs niveaux de raffinement. Elle met l'accent sur la structure des données, l'architecture du système en termes de modules et leurs interfaces et les détails procéduraux.

• l'implantation - traduit le design dans une forme qui peut être utilisée par l'ordinateur.

• les tests - rend possible la vérification de l'implantation afin de démontrer qu'elle est conforme aux demandes initiales, à la spécification et au design.

• la maintenance - comprend plusieurs aspects parmi lesquels on mentionne:

  • la maintenance corrective - permet de corriger des erreurs qui n'ont pas été détectées antérieurement et qui peuvent surgir lors de l'exécution;

  • la maintenance adaptive - harmonise le système aux changements de son environnement (par exemple à une nouvelle architecture de la machine);

  • la maintenance de performance - répond à des nouvelles fonctionnalités et à des performances accrues.

2.2 Méthodes formelles

2.2.1. Définition

La spécification d'une composante de système définit les aspects essentiels de son comportement. Certains aspects peuvent être laissés comme choix d'implantation.

Généralement, la description initiale d'une composante de système est faite de façon informelle, en langage naturel et complété, éventuellement, avec d'autres formes de structuration de l'information: diagrammes, tableaux, listes. Ce type de spécification a l'avantage d'être facilement compréhensible par les lecteurs (humains). Il présente, par contre, plusieurs inconvénients. Une spécification en langage naturel contient des ambiguïtés, est sujette à des interprétations, parfois même contradictoires, et il est difficile de vérifier sa complétude (i.e. qu'elle est en mesure de couvrir toutes les situations qui peuvent surgir) et sa consistance (i.e. le comportement est celui attendu et identique si répété). En outre, une spécification informelle ne peut pas être traitée par des outils d'automatisation.

Les **méthodes formelles** cherchent à contourner ces problèmes, par l'intermédiaire des **langages de spécification formelle** qui fournissent un ensemble de constructions bien définies et des techniques mathématiques pour la description des propriétés du système et

de son comportement. De nombreux langages de spécification formelle ont été développés qui sont basés sur différentes approches: machines à états finis, grammaires formelles, réseau de Petri, calcul algébrique, langages de programmation, types abstraites de données, logique temporelle etc.

2.2.2 Rôle des spécifications formelles

L'utilisation de spécifications formelles rend possible l'automatisation totale ou partielle des certaines activités du cycle de vie d'un système. Il en bénéficie:

• la validation de la spécification qui représente le processus qui assure l'obtention d'une spécification qui correspond aux besoins formulés;

• l'implantation par génération de code source;

• la sélection de cas de tests dans le but de couvrir tous les aspects du comportement qu'on doit vérifier;

• l'analyse des résultats de tests.

2.3 Spécification de protocole

2.3.1 Aspects à spécifier

Dans le contexte de l'architecture OSI, les spécifications entre les systèmes communicants concernent (Figure 3):

• Le **service de communication** qui définit le service de communication fourni aux entités utilisateur de la couche supérieure (i.e. le comportement de la partie grisée);

• Le **protocole** qui définit le comportement d'une entité de protocole en vue d'assurer le service de communication désiré et la communication à travers différents systèmes.

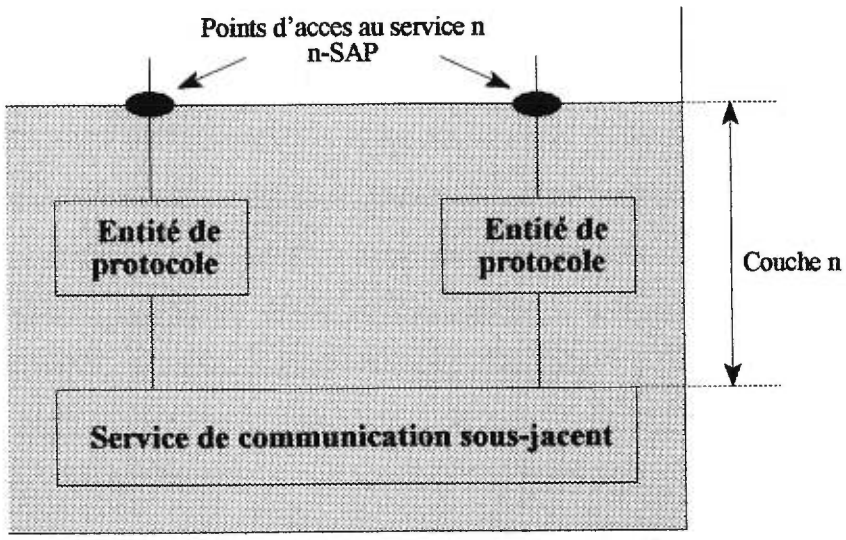


Figure 3: Service de communication

On peut considérer la spécification de protocole comme un raffinement de la spécification du service de communication.

La spécification de protocole représente une référence pour l'implantation car elle est à la base de celle-ci (son plus grand niveau d'abstraction), ainsi que de la sélection de cas de tests et de l'analyse des résultas de tests.

On distingue les aspects suivants de la spécification des services et des protocoles de communication [6]:

• **L'ordonnancement temporel des interactions** définit l'ordre chronologique dans lequel les interactions sont exécutées;

• Les **paramètres des interactions**. Généralement, les primitives de service et les PDUs contiennent des paramètres qui permettent aux entités de protocole de se passer des informations. Le domaine des valeurs des paramètres définit l'ensemble de valeurs valides de ceux-ci. Il comprend deux volets:

- les valeurs valides des paramètres d'une interaction sortant d'une entité, qui dépend généralement du contexte;
- l'interprétation des valeurs des paramètres d'une interaction entrante;

• Le **codage** définit le format sous lequel les PDUs et leurs paramètres seront véhiculés à travers le service sous-jacent;

• Les **propriétés de vivacité** spécifient des propriétés qualitatives voulant que certains événements ou actions aient lieu. Par exemple, une demande de connexion sera acceptée à un moment donné.

• Les **propriétés de temps réel** fournissent des mesures quantitatives. Généralement, elles concernent les délais de communication et la capacité de transport qui peut être atteinte (*attainable throughput*), ainsi que la fiabilité, le temps moyen entre les défaillances et les probabilités de refus des services demandés.

2.3.2. Techniques de spécification

La communication entre des systèmes comprend deux aspects:

• le flux de contrôle - concerne le comportement;

• les structures de données - concerne la représentation des données.

Il existe des techniques de spécification pour les deux, mais on ne va mentionner que celles qui seront utilisées lors du développement du protocole de signalisation. Elles appartiennent à la première catégorie.

2.3.2.1 Diagramme d'ordonnancement temporel (*time-sequence diagram*)

Un diagramme d'ordonnancement temporel permet la représentation des règles qui régissent l'ordre d'exécution des différentes primitives par entités de protocole paire (de même niveau). La figure 4 représente un exemple d'établissement d'une connexion par un diagramme de ordonnancement en temps.

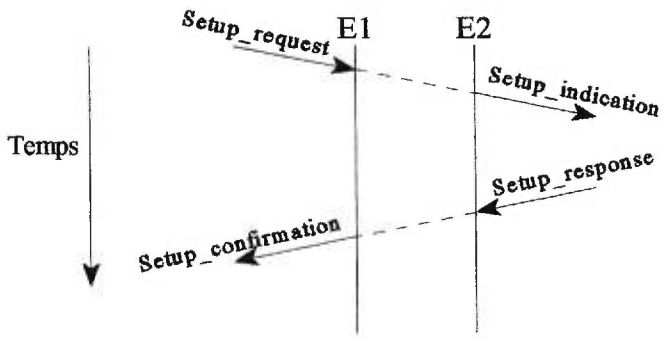


Figure 4: Exemple de diagramme de ordonnancement temporel

2.3.2.2 Diagramme de ordonnancement des messages (MSC - *Message sequence chart*)

Un diagramme de ordonnancement des messages décrit les interactions entre différentes composantes d'un système. Les abscisses représentent les composantes du système entre lesquelles des interactions sont échangées. Les interactions sont représentées par des flèches. La figure 5 représente un exemple de MSC, où quatre composantes d'un système échangent les interactions i1,...,i5.

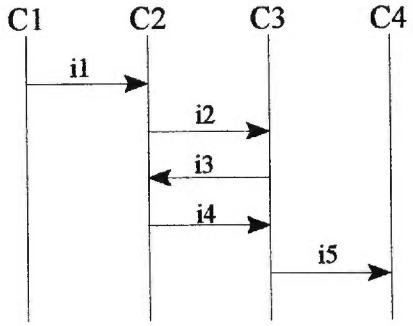


Figure 5: Exemple de MSC

2.3.2.3 Tableau d'états (*State table*)

Un tableau d'états décrit le comportement d'une composante du système par un ensemble d'états et par les transitions que celle-ci est censée faire en inter-agissant avec d'autres composantes du système. Le tableau d'états peut contenir des prédicats, identifiant les conditions qui causent une transition, et des actions, surgies comme résultat d'une transition. Ces prédicats peuvent être définis de façon informelle.

2.3.2.4 Automate à états finis (FSM - *Finite State Machine*)

L'automate à états finis ou machine d'états finis est le plus simple modèle de tableau d'états pour lequel le nombre d'états et celui de transitions sont finis. Il peut être représenté sous différentes formes, dont la plus répandue est celle d'un graphe orienté. Les noeuds désignent les états, tandis que les transitions sont représentées par les arcs du graphe. Un arc est identifié par les opérations d'entrée/sortie qui ont lieu au cours de la transition respective.

La figure 6 représente un FSM qui a deux états et trois transitions. Sur l'entrée *a* l'automate change d'état de *S1* à *S2* et produit la sortie *x*.

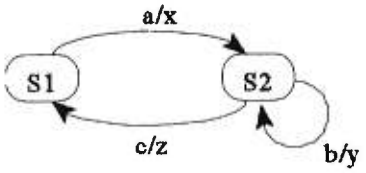


Figure 6: Exemple de FSM

Les automates à états finis étendus (EFSM, *Extended Finite State Machine*) sont des automates à états finis qui manipulent des variables. On peut alors associer aux transitions des prédicats, i.e. des conditions (fonctions booléennes). Une transition est exécutée seulement si la condition associée est satisfaite.

2.3.3. Techniques de description formelle pour OSI

Les méthodes formelles fournissent, outre une syntaxe bien définie, une sémantique formelle. Les méthodes formelles développées par CCITT et ISO, à savoir Estelle, LOTOS et SDL, s'appellent **techniques de description formelle** (FDT, *Formal Description Technics*) et sont utilisées principalement pour la description des protocoles OSI.

2.3.4. Outils

Des outils sont normalement associés à des langages spécifiques et fournissent une aide importante dans les différentes activités de développement d'un protocole de communication.

2.3.4.1 Création

La création des spécifications peut être facilitée par l'utilisation de différents outils, tels que des éditeurs de texte ou des éditeurs graphiques. Certains outils permettent la vérification en ligne de la syntaxe du langage.

2.3.4.2 Validation

Les outils de validation prennent comme entrée la spécification formelle. Ils peuvent effectuer deux types d'analyse:

• statique - similaire à la compilation des programmes, examine la logique et le code de point de vue algébrique. Cette opération détecte les erreurs de syntaxe, les variables non-déclarées, sans utiliser les valeurs des données d'entrée ou sortie.

• dynamique - détecte des erreurs qui peuvent surgir lors de l'exécution du système.

Les méthodes de validation dynamique peuvent être classifiées en [10]:

• preuve logique. Cette méthode est basée sur la démonstration d'assertions, ce qui est très difficile à appliquer aux protocoles de communication. Même avec les outils disponibles, cette tâche est laborieuse et coûteuse en temps de travail.

• exhaustive - explore toutes les situations qui peuvent surgir pendant l'exécution du système, ce qui offre une garantie totale quant au respect des propriétés vérifiées. Mais, cette méthode génère une explosion combinatoire du nombre de cas. Elle est utilisée sur des spécifications à un grand niveau d'abstraction et rien ne garantit que les propriétés soient préservées lors du raffinement ultérieur. Dans le cas d'une spécification formelle basée sur le modèle FSM, comme la nôtre, la validation exhaustive revient à explorer tous les états globaux du système. Un état global du système est donné par les états de chaque processus modélisé par un FSM et les interactions en transit. Le but de la méthode est de construire et explorer tous les états globaux qui peuvent être atteints à partir de l'état initial en vue de détecter les interblocages (*deadlocks*), les situations de bouclage (*livelocks*) et les réceptions non-spécifiées. Cette méthode est connue sous le nom d'**analyse d'accessibilité**. Les outils d'analyse d'accessibilité utilisent différentes techniques pour éviter l'explosion d'états: analyse du graphe d'accessibilité au fur et mesure qu'il est construit, utilisation d'un seul bit pour le stockage d'un état global (qui indique s'il a été visité ou non) etc.

• simulation - explore seulement certaines situations d'exécution. Les outils qui appliquent cette méthode proposent divers types d'exploration: aléatoire, guidée, basée sur des probabilités etc.

Normalement, les outils qui réalisent la validation produisent des rapports d'exécution.

Certains outils possèdent des facilités d'observation de la simulation et même d'intervention, par exemple via des fenêtres d'affichage des valeurs des variables ou de la trace de simulation.

2.3.4.3 Implantation

L'implantation représente une combinaison matériel-logiciel (*hardware-software*) qui permet l'exécution du protocole. Contrairement à la simulation, l'exécution d'une implantation est faite dans un environnement réel. Différents outils permettent de dériver automatiquement une partie de l'implantation à partir de la spécification formelle et, généralement, ils prennent en entrée la spécification formelle la plus raffinée et validée. L'automatisation consiste dans:

• la génération de code source d'un langage de programmation qui décrit:

- les structures de données,
- le comportement,
- la variation des valeurs des paramètres des interactions,
- le codage/décodage des paramètres en PDUs.

• la fourniture d'un support d'exécution: gestion de mémoire, communication entre les tâches (*task*) etc.

L'implantation générée de façon automatique doit être complétée et adaptée par un code source écrit manuellement.

2.3.4.4 Sélection des cas de tests

Le problème de la validation d'une implantation par des tests est de trouver la série de tests en mesure de couvrir tous les aspects du comportement du protocole, dans le but de s'assurer qu'il correspond bien au standard. Un cas de test est une séquence d'entrées qui

vise à faire la preuve d'une certaine capacité ou d'un aspect de comportement du protocole. L'ensemble des cas de test utilisé pour tester l'implantation d'un protocole s'appelle **suite de tests**. Une suite de tests doit être courte et offrir une bonne couverture.

Les **tests de conformité** sont utilisés pour vérifier si le comportement extérieur de l'implantation d'un protocole est équivalent à une spécification choisie comme référence. La spécification de référence peut être la spécification initiale (dans le cas "*black box*" où la structure interne du système n'est pas accessible) ou une spécification plus abstraite (dans le cas *"white box"* où la structure interne du système est connue). La plupart des techniques de génération de suites de tests de conformité, et par conséquent les outils qui les appliquent, sont basés sur le modèle FSM des spécifications (par exemple, les méthodes W, *Unique-Input-Output* etc.). Certains aspects, comme la variation des valeurs des paramètres des interactions, ne sont pas pris en considération.

Par ailleurs, les tests de conformité ne couvrent pas les aspects de performance et robustesse.

Les lacunes des cas de tests générés de façon automatique peuvent être complétés par d'autres méthodes.

Le diagnostic indique la partie responsable du comportement fautif.

2.3.4.5 Analyse des résultats des tests

L'exécution d'un cas de test sur une implantation produit une trace d'exécution représentée par les sortie observables. L'analyse des résultats des tests réside sur la comparaison entre la trace d'exécution et les sorties attendues (i.e. les sorties prévues par la spécification de référence). Elle produit un **verdict**. L'automatisation de l'analyse des résultats des tests est possible dans la mesure où la spécification de référence est décrite dans un langage formel. Généralement, les sorties attendues sont fournies dans le cas de test lui-même et, dans certains types de test, une sortie imprévue est facilement détectable. Par contre, dans d'autres types de test, par exemple dans à distance ou aléatoire, l'automatisation devient plus complexe.

Si le langage utilisé pour la définition des cas de tests est différent de celui de la spécification formelle de référence, des outils de traduction sont à envisager.

3. Réseau ATM

3.1 Introduction

Dans le monde des futures télécommunication, de nouveaux services tel que la vidéo-téléphonie, la vidéo-bibliothèque et le transfert de données à haute vitesse s'ajouteront aux services déjà existants, comme le transfert de la parole, la distribution de la TV et le transfert à basse vitesse. L'arrivée de ces nouveaux services crée de nouveaux besoins en matière de réseau de télécommunication. Il est nécessaire de mettre au point de nouvelles techniques de télécommunication, offrant un certain nombre d'avantages par rapport aux techniques existantes.

L'**ATM** (*Asynchronous Transfer Mode*) est un protocole de réseau haut débit qui transmet les données en paquets de longueur fixe, appelés cellules. Le concept de l'ATM est né au début des années 80, au CNET Lannion (*Centre National d'Études de Télécommunications*).

Les réseaux haut débit, en particulier l'ATM, permettent d'une part d'absorber davantage de connexions simultanées, et d'autre part l'échange de données numériques très variées: fichiers informatiques, voix, images, vidéo. Cette possibilité permet aux applications multimédias qui intègrent ces différents types de données d'accéder aux réseaux et de communiquer entre elles. Un autre type d'applications intéressées par les réseaux à haut débit est l'informatique répartie, c'est-à-dire des applications traitées par plusieurs ordinateurs reliés par des réseaux, qui peut nécessiter des échanges volumineux de données. De façon générale, toutes les applications manipulant de grandes quantités de données (la simulation par exemple) peuvent bénéficier des réseaux haut débit pour leurs communications.

L'ATM est destiné à interconnecter aussi bien des abonnés terminaux que des réseaux publics ou privés (Figure 7). L'architecture du réseau est maillée, c'est-à-dire qu'il est constitué de noeuds de commutation ou commutateurs, reliés entre eux par des liaisons point à point.

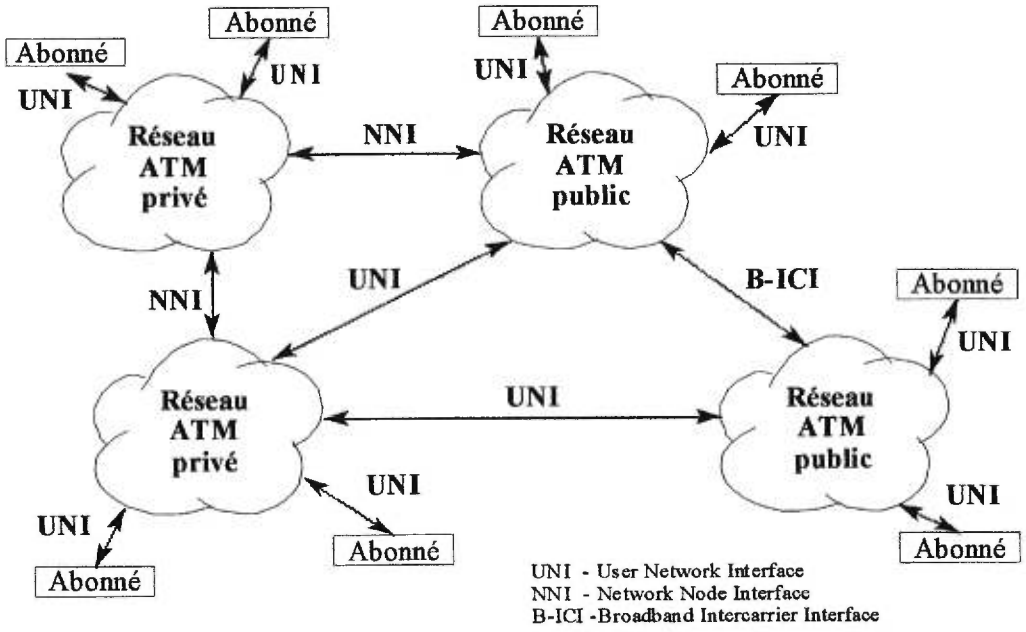


Figure 7: Différentes interfaces réseau ATM

La figure 7 décrit les divers composants de l'architecture d'un service ATM. On distingue trois types d'interfaces:

• utilisateur-réseau (UNI, *User Network Interface*) permet à un abonné de communiquer à travers une réseau ATM;

• réseau-noeud (NNI, *Network Node Interface*) permet l'intéropérabilité des noeuds ATM construits par différents manufacturiers;

• inter-réseaux (B-ICI, *Broadband Intercarrier Interface* dans l'appellation de ATM Forum ou INI, *Inter-Network Interface* en général) permet, outre l'intercommunication, la délimitation opérationnelle et administrative des réseaux ATM interconnectés.

3.2 Principes de base

3.2.1. Transfert de l'information

ATM utilise une variante de la communication de paquets, appelée commutation de cellules. La technique utilisée est celle des circuits virtuels (ou voies logiques). ATM est fondé sur le multiplexage temporel asynchrone.

Une **cellule** est un paquet de petite taille et de longueur fixe. Elle comprend un en-tête de cinq octets et une partie de données utilisateur de 48 octets, représentant la charge utile.

3.2.2. Ressources

Dans la technique de communication utilisée par ATM, avant tout transfert d'information, l'appelant source doit établir une connexion (chemin virtuel) avec l'appelé. Deux types de connexions sont réalisables:

• par canaux virtuels, VCC (*Virtual Channel Connexion*). Une connexion par canal virtuels est une capacité de transmission de bout en bout entre deux points d'accès (*VCC end points*). Elle est formée par la concaténation de **liaisons de canal virtuel** (*virtuel channel links*). Une exemple de liaison de canal virtuel est la capacité de transmission entre l'appelant et le commutateur ATM local.

• par conduits (ou faisceaux) virtuels, VPC (*Virtual Path Connexion*). Plusieurs connexions par canaux virtuels peuvent être regroupées pour donner une connexion par conduits virtuels. Elle est formée par la concaténation de **liaisons de conduit virtuel** (*virtuel path links*).

La relation entre les est montrée sur la figure 8.

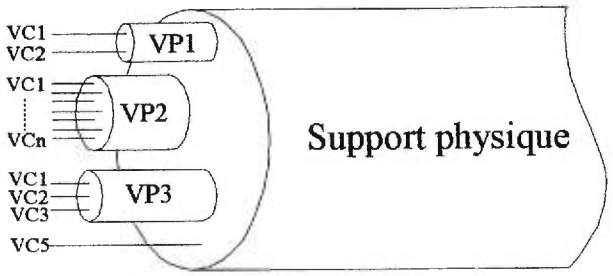


Figure 8: Relation VP/VC

Les cellules d'une même connexion par canaux virtuels ou par conduits virtuels sont repérées, respectivement, par l'indicateur de canal virtuel VCI (*Virtuel Channel Identifier*)

ou par l'indicateur de conduit virtuel VPI (*Virtuel Path Identifier*) de l'en-tête de la cellule. Ces identificateurs n'ont de signification que sur une seule liaison. Leurs valeurs sont traduites à chaque passage d'un équipement de commutation. Les commutateurs gèrent des tableaux de correspondance qui leur permettent de remplacer le VPI et/ou le VCI des cellules entrantes par les valeurs associées à la liaison suivante (figures 9 et 10). Un commutateur qui route les cellules par conduit virtuel, donc qui travaille uniquement sur le champ VPI, s'appelle **brasseur** ou ATM DCC (*Digital Cross Connect*).

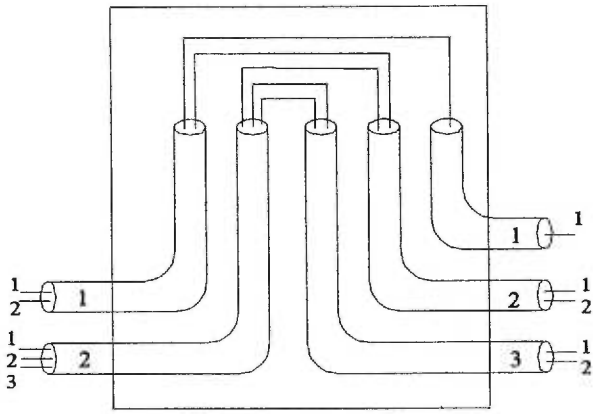


Figure 9: Commutateur ATM

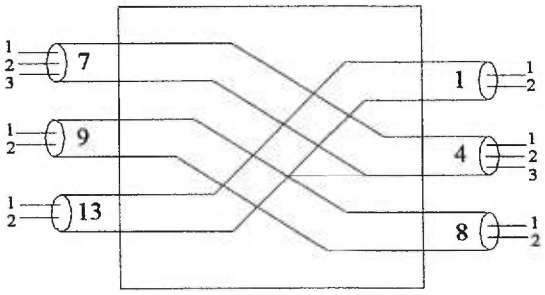


Figure 10: Brasseur ATM

Outre l'affectation d'un identificateur de canal virtuel VCI (*Virtual Channel Identifier*) et/ou d'un identificateur de conduit virtuel VPI (*Virtual Path Identifier*) l'établissement d'une connexion implique aussi l'affectation des ressources requises sur l'accès utilisateur et sur le réseau. Ces ressources s'expriment en termes de capacité (débit binaire) et de qualité de service. Elles peuvent être négociées entre l'utilisateur et le réseau pour les connexions commutées lors de l'établissement de l'appel, voire pendant l'appel.

3.2.3. Cellule ATM

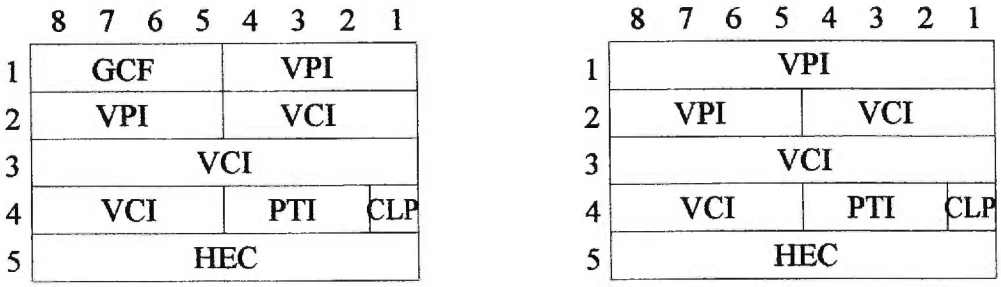
Une cellule ATM a une longueur fixe de 53 octets, dont:

- 5 octets d'en-tête contenant des informations de service,

• 48 octets de charge utile i.e. des informations de service des couches supérieures ou des données utilisateur.

L'en-tête sert au routage de la cellule. La charge utile est transférée de façon transparente sur le réseau autrement dit, elle ne fait l'objet d'aucun traitement sur le réseau, tel que le contrôle d'erreur. Son contenu est utilisé par les couches supérieures.

Le contenu de l'en-tête diffère légèrement suivant l'endroit où se trouve la cellule, à une extrémité de la connexion (UNI, *User Network Interface*) ou dans un noeud du réseau (NNI, *Network Node Interface*) (Figure 11).



Structure de l'en-tête ATM à l'UNI

Structure de l'en-tête ATM à l'NNI

Figure 11: En-tête de cellule ATM

Les champs de l'en-tête:

GFC (*Generic Flow Control*), présent seulement dans les cellules qui se trouvent à l'interface utilisateur-réseau, est prévu pour l'implantation ultérieure d'un contrôle de flux en cas d'accès multipoint (canal partagé entre plusieurs utilisateurs). Comme son contenu n'est pas encore défini, sa valeur est toujours 0.

VPI (*Virtual Path Identifier*) et VCI (*Virtual Channel Identifier*) sont les indicateurs de conduit (faisceau) virtuel et de canal virtuel respectivement. Les indicateurs VPI et VCI sont définis localement (pour chaque section) entre deux noeuds. Chaque noeud possède une table de routage qui contient, pour chaque indicateur d'entrée, l'indicateur de sortie correspondant. Les noeuds changent donc la valeur des indicateurs VPI et/ou VCI sur les en-têtes des cellules routées.

PTI (*Payload Type Identifier*) décrit le type de la charge utile transportée par la cellule ce qui détermine la manière dont la cellule est traitée. Si la charge utile représente des données utilisateur (premier bit est égal à 0), elle est transportée de façon transparente sur le réseau. Si la charge utile représente des informations de gestion et contrôle (premier bit est égal à 1), elle peut être traitée sur le réseau.

CLP (*Cell Loss Priority*) - préférence à l'écartement - décrit le mode de gestion de la perte de cellules lors de la congestion du réseau. Les cellules dont le CLP est égal à 1 ont une priorité plus faible et sont plus susceptibles d'être supprimées par le réseau, tandis que la valeur 0 de ce champ indique que les cellules transportent des informations fondamentales. De cette façon l'utilisateur a la possibilité de choisir deux niveaux de taux de perte de cellules sur un connexion en cas de congestion du réseau.

HEC (*Header Error Control*) - est un code de contrôle d'erreur. Il s'applique seulement à l'en-tête et permet la détection des erreurs survenues sur plusieurs bits et la correction des erreurs d'un bit. Son rôle est de fournir une protection contre les cellules étrangères à la connexion, mais livrées à cause d'une erreur de routage.

3.3 Architecture ATM

Le réseau B-ISDN de l'ATM utilise la même architecture stratifiée que celle utilisée dans l'interconnexion de systèmes ouverts (OSI). Toutefois, seules les couches inférieures sont examinées.

Le modèle de référence du protocole B-ISDN pour l'ATM, représenté sur la figure 12, utilise le concept de plans séparés permettant la séparation des fonctions de l'utilisateur, du contrôle et de la gestion. Pour chaque plan, une approche en couches comme dans l'OSI est utilisée.

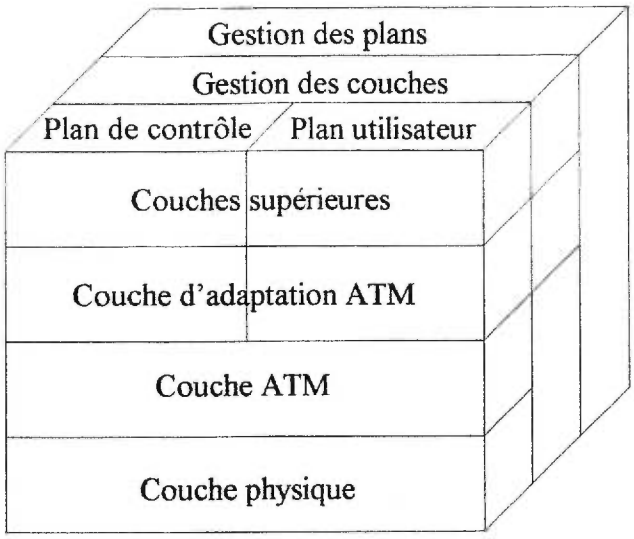


Figure 12: Modèle de référence du protocole B-ISDN pour l'ATM

Le **plan utilisateur** (*U-plane*) regroupe les fonctions de transfert des données, de détection et de correction d'erreurs. Il contient la couche physique, la couche ATM et plusieurs types de couches d'adaptation ATM requises pour différents utilisateurs de services (type de trafic).

Le **plan de contrôle** (*C-plane*) regroupe les fonctions de commande, contrôle et suivi des communications. Il permet en particulier l'établissement, la gestion et la libération des connexions i.e. la signalisation. Il partage les couches physique et ATM avec le plan utilisateur. Il inclut des procédures de la couche d'adaptation ATM et des protocoles de signalisation des couches supérieures.

Le **plan de gestion** (*M-plane*) fournit des fonctions de gestion, d'administration et d'exploitation du réseau. Il contient deux sections:

- **la gestion de couches** (*Layer Management*) - réalise des fonctions de gestion spécifiques aux couches;
- **la gestion de plans** (*Plane Management*) - réalise des fonctions de gestion et de coordination relatives au système complet.

3.3.1. Plan utilisateur

3.3.1.1 Couche physique

La couche physique ATM adapte, en émission, les cellules ATM au support physique (qui utilise des trames, blocs, ...). Elle est divisée en deux sous-couches, la sous-couche média (ou support) physique (PM, *Physical Media*) et la sous-couche de convergence de transmission (TC, *Transmission Convergence*).

La sous-couche média physique est très proche du support physique et se charge de la transmission et de la réception correctes des bits sur le support physique. En outre, elle doit garantir un rétablissement approprié de l'horloge binaire dans le récepteur.

La sous-couche convergence de transmission joue le rôle d'interface entre la couche ATM (qui utilise des cellules) et le système de transmission utilisé (qui utilise des trames ou des bits). Elle prend en charge les aspects suivants:

Adaptation au débit. Le débit de la couche ATM est en général différent (inférieur) au débit offert par le système de transmission. Selon la nature du système de transmission, l'adaptation du débit se fait par:

- insertion de cellules vides;
- insertion de caractères de bourrage;
- regroupement des cellules en blocs de cellules avec insertion éventuelle de cellules vides et insertion de caractères de bourrage entre les blocs.

Protection de l'en-tête de la cellule ATM. L'en-tête de la cellule ATM regroupe les informations de service utilisées par la couche ATM. La protection se fait par le champ HEC, qui est généré et employé par la sous-couche de convergence. Cette protection est en particulier nécessaire aux numéros de canal logique et de faisceau logique qui permettent le routage des cellules sur le réseau.

3.3.1.2 Couche ATM

La couche ATM fournit un transfert transparent des cellules entre deux entités communicantes des couches supérieures. Ce transfert a lieu à travers une connexion ATM

préétablie d'après un contrat de trafic (qui spécifie la classe de la qualité de service, les paramètres de trafic etc.).

La couche ATM est complètement indépendante du support physique utilisé, et donc de la couche physique. Elle exécute les fonctions essentielles suivantes:

• multiplexage et démultiplexage. En émission, des cellules provenant des différentes connexions sont dirigées en un flux de cellules unique sur une couche physique, ce qui constitue la fonction de multiplexage. En réception, le démultiplexage consiste à reconnaître chaque cellule comme appartenant à une connexion particulière;

• traduction de l'identificateur des cellules lors de la commutation d'une liaison physique en une autre. La traduction s'effectue sur le VCI et/ou VPI de l'en-tête de chaque cellule;

• fourniture à l'utilisateur d'une VCC ou d'une VPC d'une classe de qualité de service parmi celles offertes par le réseau;

• ajout à l'émission et extraction à la réception de l'en-tête des cellules;

• contrôle de flux par des mécanismes de limitation du débit d'entrée et de rejet de cellules;

• fonction de gestion par la discrimination des cellules - utilisateur ou de contrôle - selon l'identificateur PT;

• indication de la priorité de perte;

• calibrage du trafic (*traffic shaping*) qui est une méthode qui permet de modifier le trafic soumis pour le contraindre à rester dans des caractéristiques acceptables.

3.3.1.3 Couche d'adaptation ATM

La couche d'adaptation ATM ou AAL (*ATM Adaptation Layer*) adapte les protocoles des couches supérieures aux cellules ATM de taille fixe autrement dit, elle fait la projection (*mapping*) des unités de données des couches supérieures dans le champ de charge utile d'une ou plusieurs cellules ATM, ainsi que leur réassemblage.

La couche d'adaptation ATM se décompose en deux sous-couches:

• la **sous-couche de segmentation et réassemblage** (SAR, *Segmentation And Reassembly*) dont la fonction est le découpage en cellules ATM des unités de données à l'émission et le réassemblage des cellules en PDU, à la réception.

• la **sous-couche de convergence** (CS, *Convergence Sublayer*) qui est dépendante du service.

Les services offerts par la couche d'adaptation ATM sont classifiés en trois **classes** ou **types** selon le type de trafic à transmettre.

Adaptation aux services à débit binaire constant: AAL1

Les services à débit binaire constant (CBR, *Constant Bit Rate*) sont destinés aux applications qui nécessitent un transfert d'informations à débit constant, ainsi qu'une synchronisation entre la source et la destination. Exemple: le transfert de la vidéo ou de la voix.

La sous-couche SAR accepte de la sous-couche CS des blocs de longueur fixe (47 octets) auxquels elle ajoute son propre en-tête d'un octet contenant le numéro de séquence (reçu de la sous-couche CS) et un champ de protection de l'en-tête contre les erreurs. À la réception, le numéro de séquence permet de détecter une éventuelle perte de cellules ou l'insertion de cellules acheminées sur cette connexion suite a une erreur dans le(s) champ(s) VPI/VCI, tandis que le champ de protection contre les erreurs de l'en-tête permet la correction des erreurs sur un bit et la détection des erreurs sur plusieurs bits, fait annoncé à la sous-couche CS.

La sous-couche de convergence gère principalement le traitement des erreurs, récupération de l'horloge, élimine la gigue des cellules (la variation des délais de transfert des cellules dans le réseau, due aux tampons des noeuds).

Adaptation aux services à débit binaire variable: AAL2

La couche AAL2 offre un transfert d'informations à débit binaire variable (VBR, *Variable Bit Rate*). Par ailleurs, l'information d'horloge est transférée de la source vers la destination. Puisque la source génère un débit binaire variable, le niveau de remplissage des cellules varie d'une cellule à l'autre. Par conséquent, outre le numéro de séquence et la protection contre les erreurs, les informations de service ajoutées par la couche SAR (comme en-tête et queue) spécifient le type de l'information de la cellule (signal pour la synchronisation ou autre) et le nombre d'octets utiles d'une cellule.

La sous-couche CS exécute la récupération de l'horloge et le traitement des erreurs.

Adaptation aux services à débit binaire variable: AAL3/4

La CCITT recommande son utilisation pour le transfert de données sensibles à la perte, mais pas au retard. Exemple: le transfert de fichiers informatiques.

La sous-couche SAR reçoit de la couche CS des unités de données de longueur variable (entre 1 et 65535 octets) qu'elle transforme en cellules ATM dont la charge utile est de 44 octets. L'information de service (en-tête et queue) contient des champs permettant:

- la segmentation/réassemblage des unités de données: l'emplacement des données de la cellule dans l'unité de données (début, suite, fin ou unique), le numéro de séquence, la longueur de la charge utile;
- la détection d'erreurs dans la charge utile;
- le multiplexage/démultiplexage des cellules provenant d'unités de données différentes sur la même connexion ATM: indicateur de la connexion.

La sous-couche CS réalise le traitement des erreurs.

Deux procédures de fonctionnement sont offertes par la couche AAL de type 3/4:

- le mode assuré - les unités de données de la couche sont retransmises en cas de perte ou d'erreurs;
- le mode non assuré - les unités de données manquantes ou corrompues ne sont pas corrigées par retransmission.

Adaptation aux services à débit binaire variable: AAL5

La couche AAL5 est une version simplifiée de la couche AAL3/4. L'objectif de la couche AAL5 est de fournir un service avec une quantité d'information de service inférieure à celle utilisée par AAL3/4 et une détection d'erreurs identique. Il n'y a plus de multiplexage de plusieurs unités de données.

La sous-couche SAR accepte des unités de donnés de longueur variable, entiers multiples de 48 octets de la couche CPCS qui vont être segmentées. En fait, SAR de type 5 utilise la cellule ATM et n'a besoin que d'un bit d'information de service (le champ PT de la cellule ATM) pour signaler la fin de l'unité de données. (La première cellule d'une unité de données est celle qui suit une cellule étant la dernière d'une unité de données.)

Les informations de service de CS de type 5 sont ajoutées à la fin de l'unité de données. Par conséquent, elles vont se retrouver dans la charge utile de la dernière cellule ATM.

3.3.1.4 Couches supérieures

Les couches supérieures du plan utilisateur comprennent tous les protocoles spécifiques aux différents services nécessaires pour la communication de bout en bout. En principe, les protocoles des couches supérieures déjà existants peuvent être adaptés en fonction de l'application: dans certains cas ils sont simplifiés, car plusieurs fonctions sont prises en charge par les couches basses, tandis que dans d'autres cas ils doivent être complétés.

3.3.2. Plan de gestion

Les procédures de gestion du réseau local dans le plan M sont considérées comme étant encore à l'étude. Jusqu'à ce qu'elles soient disponibles le protocole ILMI (*Interim Local Management Interface*) (interface provisoire de gestion locale) fournit à tout utilisateur ATM des informations relatives à l'état et à la configuration du conduit (faisceau) virtuel (VP, *Virtual Path*) et du canal virtuel (VC, *Virtual Channel*) existant à l'interface utilisateur/réseau (UNI, *User Network Interface*).

3.3.3. Plan de contrôle

Les couches ATM et physique du plan de contrôle restent identiques à celles du plan utilisateur.

Par contre, le plan de contrôle utilise une couche d'adaptation ATM différente de la couche AAL utilisateur, appelée **SAAL** (*Signalling ATM Adaptation Layer*). Elle est

nécessaire afin d'ajuster les protocoles de signalisation aux services fournis par la couche ATM sous-jacente. La structure de la couche SAAL comprend (Figure 13):

• la partie commune (CP, *Common Part*) fournit un transfert non-assuré de l'information et un mécanisme de détection des unités de données de signalisation corrompues. Bien que les deux sous-couches AAL de type 3/4 et de type 5 peuvent être utilisées en tant que partie commune, les spécifications du Forum ATM et ITU-T vont dans le sens de AAL5.

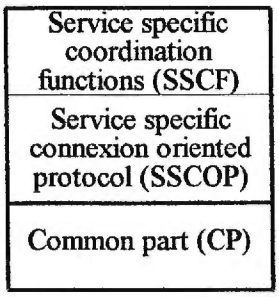


Figure 13: Structure de la couche SAAL

• la partie spécifique (SSP, *Service-specific part*) est divisée en deux sous-couches:

- le protocole en mode connecté spécifique au service (SSCOP, *Service Specific Connexion Oriented Protocole*), qui fournit les mécanismes pour le transfert fiable des unités de données entre deux utilisateurs SSCOP. Il récupère les unités perdues ou erronées;
- les fonctions de coordination spécifiques au service (SSCF, *Service Coordination Function*) assurent l'adaptation entre les exigences de la couche supérieure et celles de la couche inférieure.

3.4 Signalisation ATM à l'interface utilisateur-réseau

Avant tout transfert de données sur un réseau ATM, l'appelant doit établir une connexion avec l'appelé. La **signalisation ATM** spécifie les procédures d'établissement, de surveillance et de libération des connexions ATM. Le transfert des données et la signalisation sont traités dans des plans différents: plan utilisateur et plan de contrôle respectivement.

Un **utilisateur** ATM représente tout équipement qui utilise un réseau ATM, via une **interface utilisateur-réseau ATM, UNI (*User-Network Interface*).** (Figure 14)

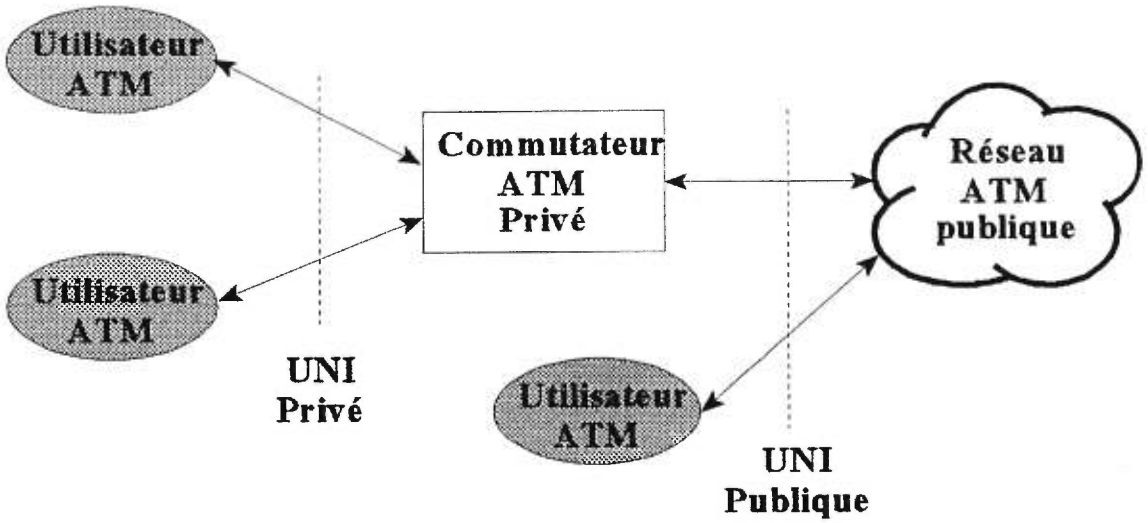


Figure 14: Implantation des interfaces utilisateur-réseau de type ATM

Par exemple, un utilisateur ATM peut être:

• un routeur qui incapsule les données dans des cellules ATM qui seront ensuite passées, à travers une interface ATM, vers un commutateur (*switch*) (privé ou dans un réseau public).

• un commutateur de réseau privé qui utilise le service d'un réseau public pour le transfert des cellules ATM vers un autre utilisateur ATM.

La **signalisation UNI** spécifie les procédures pour l'établissement, le maintien et la libération dynamiques des connexions ATM à l'interface utilisateur-réseau.

La spécification du protocole de signalisation exploité par l'utilisateur à l'interface UNI, que nous avons utilisée, est décrite dans ATM Forum UNI 3.1. Elle est basée sur un sous-ensemble de standards du protocole de signalisation à large bande présentement en développement par ITU et identifié par Q.2931 [12]. Ce sous-ensemble a été complété afin de supporter des capacités identifiées par ATM Forum comme étant importantes pour le déploiement ultérieur et l'intéropérabilité des équipements ATM. Dans ce document, les procédures pour l'établissement, le maintien et la libération des connexions sont définies en termes de messages véhiculés à l'interface utilisateur-réseau. Ces messages représentent les PDUs (*Protocol Data Unit*) échangées entre les entités de protocoles. On suppose que les

services entre les couches sont assurés par un ensemble de primitives semblables à celles décrites dans l'annexe A de la Recommandation Q.2931 [12].

L'architecture des interfaces du protocole de signalisation avec le programme d'application (AP) et avec la couche d'adaptation ATM (SAAL) est représentée sur la figure 15.

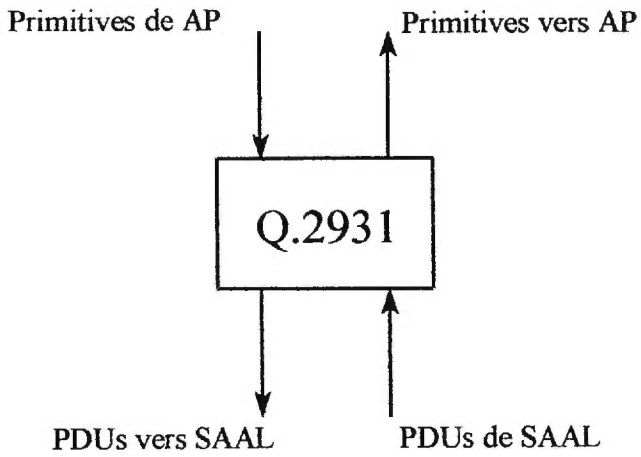


Figure 15: Architecture des interfaces du protocole de signalisation

On distingue deux configurations de connexions:

• point-à-point - qui est une collection de canaux virtuels qui connectent deux points terminaux;

• point-à-multipoint - qui est une collection de canaux virtuels qui relient les noeuds selon une topologie d'arbre.

3.4.1 Messages

3.4.1.1 Noms et significations

Messages pour les connexions point-à-point

Les messages pour les connexions point à point sont les suivants:

SETUP permet d'initier une demande de connexion. Il est émis par l'appelant vers le réseau et par le réseau vers l'appelé afin d'établir une connexion.

CALL PROCEEDING est émis par le réseau vers l'appelant ou par l'appelé vers le réseau et indique que la demande de connexion est en cours de progression.
CONNECT indique que la connexion est établie. Il est émis par l'appelé vers le réseau et par le réseau vers l'appelant.
CONNECT ACKNOWLEDGE est envoyé par le réseau vers l'appelé et par l'appelant vers le réseau pour confirmer l'appel.
RELEASE est envoyé au réseau par l'utilisateur qui demande la déconnexion ou par le réseau pour indiquer que la connexion est finie.
RELEASE COMPLETE est émis par l'utilisateur ou le réseau pour indiquer que les ressources (canal virtuel et la référence d'appel) ont été libérées.
STATUS ENQUIRY est émis par un des utilisateurs ou par le réseau afin d'obtenir des informations de l'entité paire.
STATUS est émis par l'utilisateur en réponse à un message STATUS ENQUIRY ou à tout moment pour signaler certaines erreurs.
RESTART est émis par l'utilisateur ou par le réseau pour demander le redémarrage d'une ou de toutes les connexions. Il se traduit par la libération de toutes les ressources associées à un canal virtuel spécifique ou à tous les canaux contrôlés par le canal de signalisation.
RESTART ACKNOWLEDGE confirme la réception du message RESTART et indique que le redémarrage demandé est achevé.

Messages pour les connexions point-à-multipoint

Une connexion point-à-multipoint distribue l'information de façon unidirectionnelle de l'appelant, la racine *(Root)*, vers plusieurs appelés, les feuilles (*Leaves*). La racine réalise le lien de toutes les entités de la connexion. Initialement, une connexion point-à-point est établie entre la racine et une entité appelée et, par la suite, d'autres entités entrent dans la connexion.

Les messages utilisés dans des connexions point-à-multipoint sont envoyés soit par l'utilisateur, soit par le réseau. À l'exception du dernier, tous les messages ont une signification de bout en bout.

ADD PARTY est envoyé par la racine et par le réseau pour demander l'ajout d'une entité à une connexion déjà établie.

ADD PARTY ACKNOWLWDGE confirme l'ajout.

ADD PARTY REJECT est envoyé par le réseau ou par l'appelé qui n'accepte pas l'appel.

DROP PARTY déconnecte une entité d'une connexion multipoint.

DROP PARTY ACKNOWLEDGE confirme la déconnexion de l'entité.

3.4.1.2 Format général

Le format utilisé par les messages du protocole de signalisation comporte les parties suivantes (Figure 16):

• le discriminateur de protocole - distingue les messages du présent protocole des messages des autres standards. Il a la valeur 9.

• la référence d'appel (*call reference*) - identifie l'appel auquel un message est adressé. Elle contient la valeur proprement-dite et un drapeau (*flag*) identifie le côté du canal virtuel de signalisation qui a attribué la valeur. Le côté qui initialise l'appel attribue la valeur 0 au drapeau.

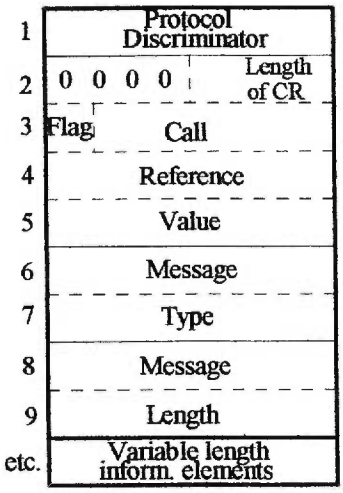


Figure 16: Format des messages

• le type du message - identifie la fonction du message et la façon dont un message non-reconnu doit être traité.

• la longueur du message - identifie la longueur du contenu du message.

• des éléments d'information - sont utilisés pour caractériser les connexions ATM et assurer l'intéropérabilité (Figure 17).

Les quatre premières parties (qu'on va appeler l'en-tête du message) sont communes à tous les messages et doivent être toujours présentes, tandis que les éléments d'information sont spécifiques à chaque message.

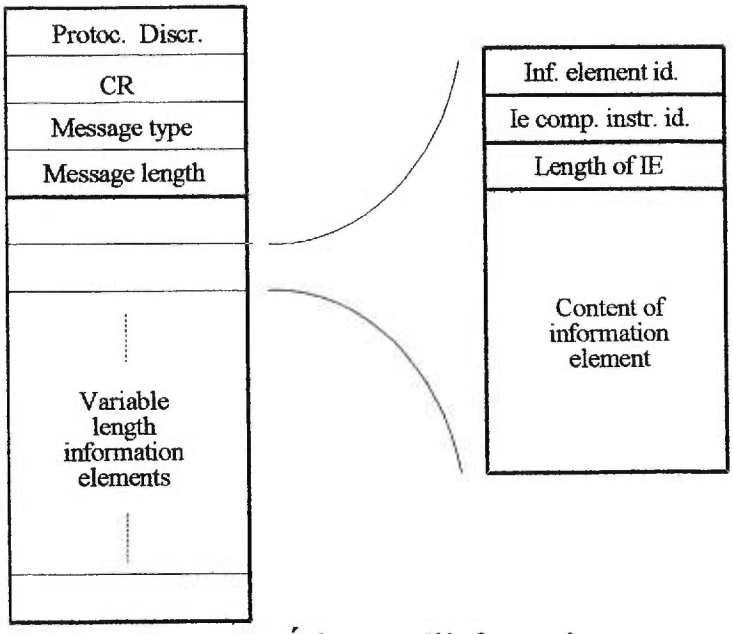


Figure 17: Éléments d'information

Les éléments d'information supportés par le protocole sont:

• Paramètres AAL - *AAL parameters* - permet de spécifier les valeurs des paramètres de bout en bout des AAL. Les valeurs des paramètres peuvent être définies pour chaque direction: aller (*forward*) i.e. de l'appelant vers l'appelé, ou retour (*backward*) i.e. de l'appelé vers l'appelant.

- AAL1 permet de décrire les propriétés du service AAL1: débit, méthode de régénération du signal d'horloge, taille des blocs, partially filled call methode.
- AAL3/4 permet de spécifier la taille des données utilisateur dans chaque sens, l'espace de valeurs pour l'identificateur de message et le type de fonction de convergence utilisé en haut de l'AAL3/4.
- AAL5 permet de décrire la taille des unités de données utilisateur dans chaque sens et le type de sous-système de convergence utilisé au-dessus de l'AAL5.
- AAL utilisateur est utilisée pour définir des AAL expérimentales.

• Descripteur du trafic - *ATM traffic descriptor* - définit les caractéristiques (débit maximal obligatoire des cellules, débit crête et moyen, variation du rythme d'arrivée des cellules etc.) de la connexion ATM pour chaque direction, séparément.

• *Broadband bearer capability* définit certaines caractéristiques du service au réseau. Par exemple, dans le cas de la transmission de la voix, la suppression de l'écho peut être demandée.

• *Broadband high layer information* permet de vérifier la cohérence des entités paire de couches hautes d'extrémités (i.e. TCP parle avec TCP, etc.).

• *Broadband low layer information* (max 3 occurrences) permet de vérifier la cohérence des entités paires de couches basses d'extrémités. Selon les protocoles des couches basses définis dans cet élément d'information, on peut spécifier d'autres caractéristiques comme la taille de la fenêtre, des paquets etc.

• État de l'appel - *call state* - décrit l'état courant d'un appel ou l'état global de l'interface.

• Adresse de l'appelé - *called party number* - permet de fournir l'adresse de l'appelé.

• Sous-adresse de l'appelé - *called party subaddress* - permet d'identifier une sous-adresse de l'appelé.

• Adresse de l'appelant - *calling party number* - permet de fournir l'adresse de l'appelant.

• Sous-adresse de l'appelant - *calling party subaddress* - permet d'identifier une sous-adresse associée à l'origine d'un appel.

• Cause (max. 2 occurrences) - *cause* - décrit la raison qui a conduit à la génération du message qui la contient (généralement un message d'erreur ou de libération de connexion), fournit un diagnostic dans le cas des erreurs procédurales et indique l'endroit qui a signalé l'erreur.

• Identificateur de la connexion - *connexion identifier* - identifie les ressources ATM à l'interface locale. Il contient les identificateurs de canal virtuel et de conduit virtuel.

• Paramètres de qualité de service - *QoS parameter* - permet de spécifier la qualité de service demandée et supportée dans chaque sens d'une connexion.

• *Broadband repeat indicator* fournit la sémantique à donner aux éléments d'information répétés.

• *Broadband sending complete* indique que le numéro d'appelé est complet. Il est utilisé pour garantir la compatibilité avec certains réseaux publics.

• Sélection du réseau de transit - *Transit network selection* - permet de choisir un réseau intermédiaire de transit.

• *Endpoint reference* identifie un membre d'une connexion multipoint.

• *Endpoint state* indique l'état d'un membre d'une connexion multipoint.

• Indicateur de démarrage - *Restart indicator* - spécifie la classe de service à réinitialiser.

Dans un message, un élément d'information est obligatoire ou optionnel. La présence d'un élément d'information dans un message est dictée par:

• le type de message. Par exemple, dans un message de type SETUP l'élément d'information qui spécifie la qualité de service doit être toujours présent.

• le type de la connexion (point-à-point ou point-à-multipoint). Par exemple, s'il s'agit d'une connexion point-à-multipoint l'élément d'information *Endpoint reference* doit être présent dans tous les messages.

• direction du message (usager → réseau ou réseau → usager). Par exemple, l'identificateur de la connexion (VCI/VPCI) est obligatoire dans un message de type SETUP qui arrive du réseau et interdit dans un message SETUP envoyé au réseau.

• l'ordre des messages. Par exemple, la présence de l'élément d'information *Cause* est obligatoire seulement dans le premier message de libération de connexion.

Les éléments d'information peuvent apparaître dans n'importe quel ordre dans le message. Toutefois, les occurrences des éléments d'information dont la répétition est permise doivent se suivre une après l'autre, précédées, éventuellement, par l'élément d'information *Broadband repeat indicator* qui indique la répétition.

3.4.1.3 Format des éléments d'information

Le format d'un élément d'information est le suivant (Figure 18):

• l'identificateur de l'élément d'information;

• l'indicateur de compatibilité;

• la longueur du contenu de l'élément d'information;

• le contenu - un ensemble de sous-éléments (champs).

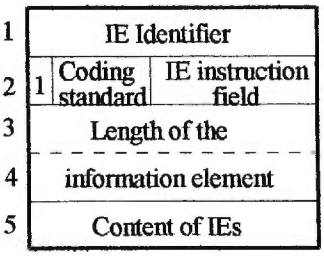


Figure 18: Format des éléments d'information

Les trois premières parties forment l'en-tête de l'élément d'information.

La présence d'un sous-élément d'information dans un élément d'information peut être obligatoire ou optionnelle.

Les circonstances qui rendent la présence d'un sous-élément d'information obligatoire sont:

• le type de l'élément d'information. Par exemple, l'élément d'information *Cause* contient toujours le sous-élément *Value*.

• la valeur d'un autre sous-élément d'information. Par exemple, si le sous-élément d'information *CBR Rate* de l'élément d'information *ATM Adaptation Layer Parameters* indique "nx64 kbit/s" ou "nx8kbps" alors le sous-élément d'information *Multiplier* doit être présent.

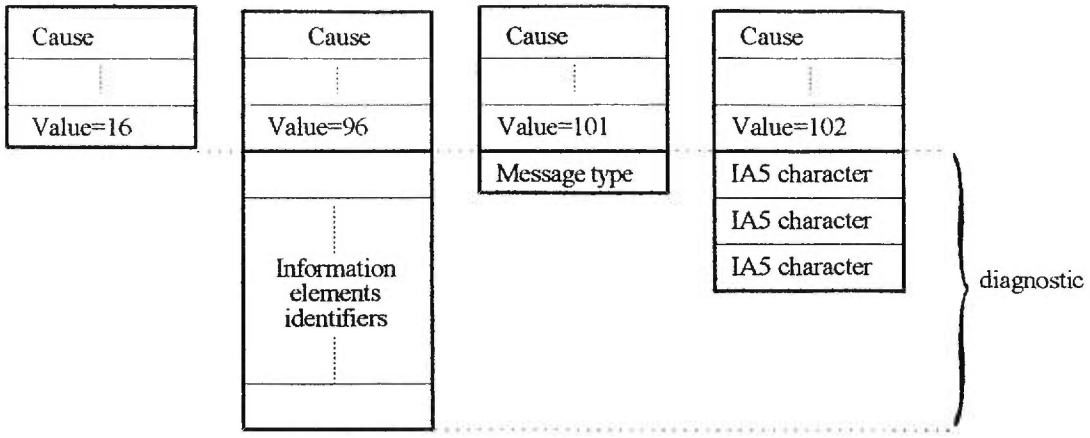
• la présence d'un autre sous-élément d'information. Par exemple, dans l'élément d'information *ATM traffic descriptor* le sous-élément d'information *Forward Sustainable Cell Rate (CLP=0)* est présent si et seulement si *Forward Maximum Burst Size (CLP=0)* est présent.

Un sous-élément d'information optionnel peut être présent:

• indépendamment de tout autre sous-élément d'information. Par exemple, dans l'élément d'information *ATM adaptation layer parameters* on peut spécifier la méthode de correction des erreurs ou non.

• en fonction de la valeur d'un autre sous-élément d'information. Par exemple, pour la valeur de la cause = 16 le diagnostic n'existe pas, tandis que d'autres valeurs (96, 101, 102) de la *Cause* le sous-élément *diagnostic* peut être présent sous des formats correspondants (Figure 19).

• en fonction de la présence d'un autre sous-élément d'information. Par exemple, dans l'élément d'information *Braodband low layer information* on peut indiquer la largeur de la fenêtre seulement si la valeur par défaut a été spécifiée.



Cause values meanings:
16 - normal call clearing
96 - mandatory information element missing
101 - message not compatible with the call state
102 - recovery on timer expiry

Figure 19: Différents types de sous-éléments d'information

3.4.2 Primitives

Puisque l'interface entre le protocole de signalisation et la couche supérieure n'est pas spécifiée, nous avons supposé l'existence d'un ensemble de primitives de service, tout comme dans l'Annexe A de la référence [12].

On a associé des primitives à des messages comme suit:

• un message d'établissement ou de libération de connexion entrant est rapportée à la couche supérieure par une primitive;

• un message d'établissement ou de libération de connexion est envoyé suite par une primitive.

• la vérification de l'état de l'entité paire est initialisée par la primitive de service *InitiateStatusEnquiry*.

On obtient les correspondances montrées dans le tableau i.

Messages	Primitives qui provoque l'envoi du message	Primitives qui rapporte la réception du message
Établissement de connexion		
SETUP	Setup_req	Setup_ind
CALL PROCCEDING	Proceeding_req	Proceeding_ind
CONNECT	Setup_resp	Setup_conf
CONNECT ACK	-	Setup_complete_ind
Libération de connexion		
RELEASE	Release_req	Release_ind
RELSEAE COMPLETE	Release_resp	Release_conf
Autres		
STATUS ENQUIRY	InitiateStatusEnquiry	-

Tableau i: Correspondance entre les messages et les primitives du protocole de signalisation

Les primitives, tout comme les messages, s'adressent à une connexion. L'initiation d'une demande de connexion, réalisée par la primitive *Setup_req*, implique l'affectation, par le protocole de signalisation, d'une référence d'appel unique qui identifie la nouvelle connexion. Cette référence d'appel est portée à la connaissance de la couche supérieure en tant que paramètre de la primitive, *Send_cr*, et sera véhiculée par toutes les primitives et tous les messages s'adressant à cette connexion.

On considère que les paramètres des primitives sont les éléments d'information du message correspondant.

3.4.3. Comportement du protocole

Dans la présente thèse on s'intéresse seulement au côté utilisateur de UNI, dans les connexions point-à-point, sans inclure la procédure de redémarrage (RESTART).

3.4.3.1 Spécification informelle

L'établissement d'une connexion

L'appelant initie un appel en transférant (sur la connexion virtuelle de signalisation) un message SETUP à travers l'interface. Il démarre le timer T303. À ce moment-là il est considéré être dans l'état U1 (*Call Initiate*). Le message SETUP contient des informations nécessaires au réseau pour traiter l'appel. S'il n'y a pas de réponse de la part du réseau avant l'expiration du timer T303, le message SETUP est retransmis et le timer T303 redémarré. Après la deuxième expiration du timer l'appelant libère l'appel.

À la réception du message SETUP le réseau détermine si l'accès au service est autorisé et disponible. Si c'est le cas, le réseau peut envoyer à l'utilisateur un message CALL PROCEEDING pour indiquer que l'appel a été traité (transmis vers l'appelé). Au moment de la réception du message CALL PROCEEDING, l'utilisateur appelant arrête le timer T303, démarre le timer T310 et entre dans l'état U3 (*Outgoing Call Proceeding*).

Si l'appelant a reçu un message CALL PROCEEDING, mais celui-ci n'est pas suivi par un message CONNECT ou RELEASE avant l'expiration du timer T310, il initie les procédures de libération de l'appel. À la réception d'un message CONNECT, l'appelant arrête le timer T303 ou T310, envoie au réseau un message CONNECT ACKNOWLEDGE. Il passe à l'état U10 (*Active*) et, a ce moment-là, la connexion est établie.

À l'interface UNI de destination, le réseau indique à l'appelé l'arrivé d'un appel en lui envoyant un message SETUP à travers l'interface. Le message contient toutes les informations nécessaires à l'appelé pour traiter l'appel. Sur la réception du message SETUP, l'appelé passe à l'état U6 (*Call Present*). Si l'appelé est en mesure de fournir les ressources indiquées (la classe de qualité de service, le descripteur de trafic) et veut accepter l'appel, il répond avec un message CONNECT, éventuellement précédé par un message CALL PROCEEDING[1]. Après l'envoi du message CALL PROCEEDING l'état de l'appelé sera U9 (*Incoming Call Proceeding*). Après avoir envoyé le message CONNECT, l'appelé enclenche le timer T313 et entre dans l'état U8 (*Connect Request*). À l'arrivée de la confirmation CONNECT ACKNOWLEDGE du réseau, l'appelé arrête le timer T313. Il

---

[1] Le message CALL PROCEEDING peut être envoyé par un utilisateur qui ne peut pas répondre à un message SETUP avec CONNECT ou RELEASE COMPLETE avant l'expiration du timer T303.

passe à l'état actif, U10 (*Active*) et la connexion est établie à l'interface UNI de destination. La connexion bout-en-bout sera réalisée seulement à l'arrivé du message CONNECT à l'interface UNI qui a initié l'appel.

Rejet d'un appel

Si le réseau détermine qu'il n'est pas en mesure de fournir les ressources indiquées dans le message de demande de connexion (SETUP) ou l'information reçue de l'utilisateur appelant est invalide, il rejette l'appel en indiquant la cause. L'appelant revient à l'état U0 (*Null*).

Si, par contre, le réseau a fait avancer le message SETUP, mais c'est l'utilisateur appelé qui n'est pas en mesure de prendre l'appel, celui-ci doit entamer la déconnexion en transférant un message RELEASE COMPLETE à travers l'interface et revient à l'état U0 (*Null*). À l'expiration du timer T310, l'appelant passe aussi à l'état U0 (*Null*) après avoir initié la libération de la connexion.

Libération de connexion

La libération d'une connexion dans des conditions normales est initiée par n'importe quel utilisateurs (appelant ou appelé) par l'envoi vers le réseau d'un message RELEASE. Le timer T308 est démarré et l'entité passe à l'état U11 (*Release Request*). À la réception d'un message RELEASE, le réseau fait avancer le message vers l'entité éloignée, libère le canal virtuel et répond à l'utilisateur avec un message RELEASE COMPLETE. La réception du message RELEASE COMPLETE provoque l'arrêt du timer T308, la libération du canal virtuelle et de la référence d'appel et le retour à l'état U0 (*Null*). Si le timer T308 expire pour la première fois, l'utilisateur retransmet au réseau le message RELEASE. La deuxième expiration du timer détermine une déconnexion unilatérale de l'utilisateur et l'équipement doit réaliser une récupération, telle qu'exécuter les procédures de redémarrage (RESTART). À la réception d'un message RELEASE l'utilisateur passe à l'état U12 (*Release Indication*), déconnecte le canal virtuel, envoie au réseau la confirmation RELEASE COMPLETE et passe à l'état U0 (*Null*).

Traitement des erreurs

Tout message entrant est objet d'une vérification. Le résultat de la vérification détermine les actions ultérieures de l'entité de protocole.

Les procédures de traitement des erreurs s'appliquent, successivement, à l'en-tête du message et aux éléments d'information.

Une entité de protocole:

• ignore tout message dont:

- le discriminateur de protocole est différent de Q.2931;
- la longueur n'est pas suffisante pour inclure au moins l'en-tête (i.e. < 9);
- le deuxième octet (contenant la valeur de la longueur de la référence d'appel) indique une valeur différente de 3;

• libère la connexion à la réception de tout message, exceptés SETUP, RELEASE_COMPLETE, STATUS ou STATUS_ENQUIRY, dont la référence d'appel n'est pas reconnue comme active;

• ignore un message de type RELEASE_COMPLETE dont la référence d'appel n'est pas active;

• ignore un message de type SETUP dont la référence d'appel est erronée (*flag*=1) ou active;

• envoie à l'entité paire un message STATUS si le message reçu (à l'exception de STATUS) contient une référence d'appel globale (i.e. dont la valeur est égale à 0);

• envoie un message STATUS (contenant la cause et l'état du processus) à la réception d'un message dont le type est inconnu ou d'un message inattendu, exceptées RELEASE et RELEASE_COMPLETE; il existe deux exceptions: 1) la réception d'un message RELEASE en réponse à un message SETUP provoque l'envoi d'un message RELEASE COMPLETE; 2) la réception d'un message RELEASE COMPLETE inattendu provoque la libération des ressources et l'entrée dans l'état U0 (*Null state*).

Les procédures de traitement des erreurs concernant les éléments d'information ont les effets suivants:

• l'absence d'au moins un élément d'information obligatoire dans tout autre message que SETUP, RELEASE ou RELEASE COMPLETE provoque l'envoi d'un message STATUS dont la cause a la valeur 96;

• l'absence d'au moins un élément d'information obligatoire d'un message SETUP provoque la libération de la connexion par l'envoi d'un message RELEASE_COMPLETE ayant la cause 96;

• l'absence de l'élément d'information *Cause* d'un premier message de libération de la connexion (i.e. RELEASE ou RELEASE_COMPLETE) provoque les mêmes actions que s'il était présent et avait la valeur 31 (*normal, unspecified*), autrement dit, la déconnexion se poursuit. S'il s'agit d'un message RELEASE, le message RELEASE COMPLETE envoyé à l'interface locale aura la cause 96;

• l'existence d'une erreur dans le contenu d'un ou de plusieurs éléments d'information obligatoires dans tout autre message que SETUP, RELEASE ou RELEASE COMPLETE provoque l'envoi d'un message STATUS dont la cause a la valeur 100;

• l'existence d'une erreur dans le contenu d'un ou de plusieurs élément s d'information obligatoires d'un message SETUP provoque la libération de la connexion par l'envoi d'un message RELEASE_COMPLETE dont la cause a la valeur 100;

• l'existence d'une erreur dans le contenu de l'élément d'information *Cause* d'un premier message de libération de la connexion (i.e. RELEASE ou RELEASE_COMPLETE) provoque les même actions que s'il était correct et avait la valeur 31 (*normal, unspecified*), autrement dit, la déconnexion se poursuit. S'il s'agit d'un message RELEASE, le message RELEASE COMPLETE envoyé à l'interface locale aura la cause 100;

• la présence d'un ou plusieurs éléments d'information inconnus dans un message autre que RELEASE ou RELEASE COMPLETE est ignorée. Un message STATUS peut être envoyé. La valeur de la cause sera 99 *(information element non-existent or not implemented)* et le diagnostic, si présent, contiendra les identificateurs des éléments d'information non-reconnus.

• la présence d'un ou plusieurs éléments d'information inconnus dans un message RELEASE implique l'affectation de la valeur 99 à la cause du message de réponse, RELEASE COMPLETE;

• la présence d'un ou plusieurs élément(s) d'information inconnu(s) dans un message RELEASE COMPLETE ne modifie pas le comportement de l'entité de protocole;

• l'existence d'une erreur dans le contenu d'un ou de plusieurs élément(s) d'information optionnel(s) d'un message est équivalente à l'absence de ceux-ci. Optionnellement, un message STATUS contenant la cause 100 (*invalid information element contents*) et,

éventuellement, les identificateurs des éléments fautifs dans le diagnostic est envoyé à l'entité paire;

• les éléments d'information reconnus par le protocole et présents dans un message dont la définition ne les inclut pas sont traités comme des éléments d'information inconnus.

3.4.3.2 Spécification par des diagrammes d'ordonnancement temporel

Les diagrammes d'ordonnancement temporel permettent de représenter l'ordre d'exécution des interactions par les entités de protocole. Les figures 20, 21, 22 et 23 représentent des exemples de diagrammes d'ordonnancement temporel des différentes phases d'une connexion ATM aux deux interfaces utilisateur-réseau (initiatrice et destination).

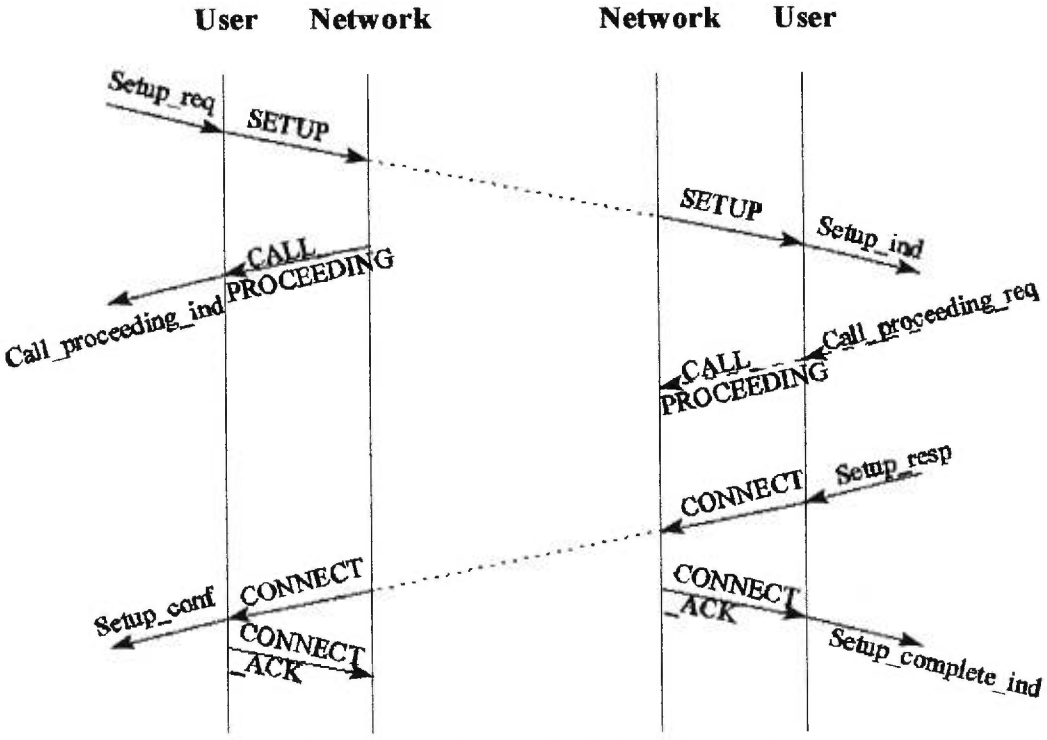


Figure 20: Exemple d'établissement de connexion

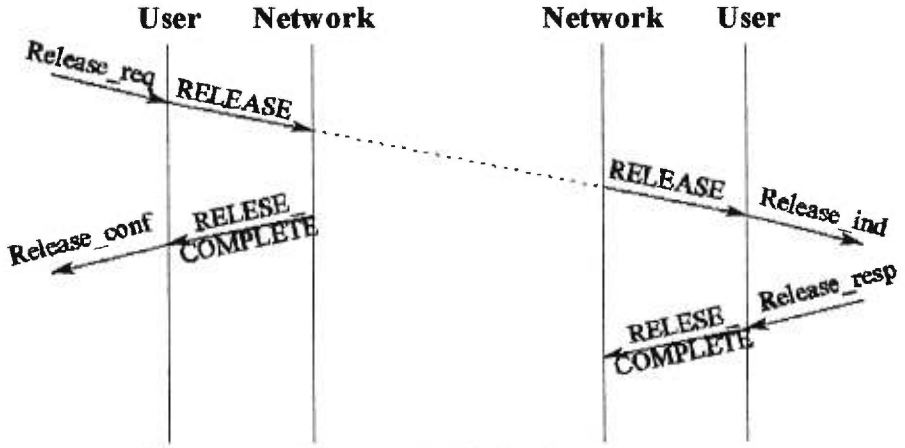


Figure 21: Exemple de libération de connexion

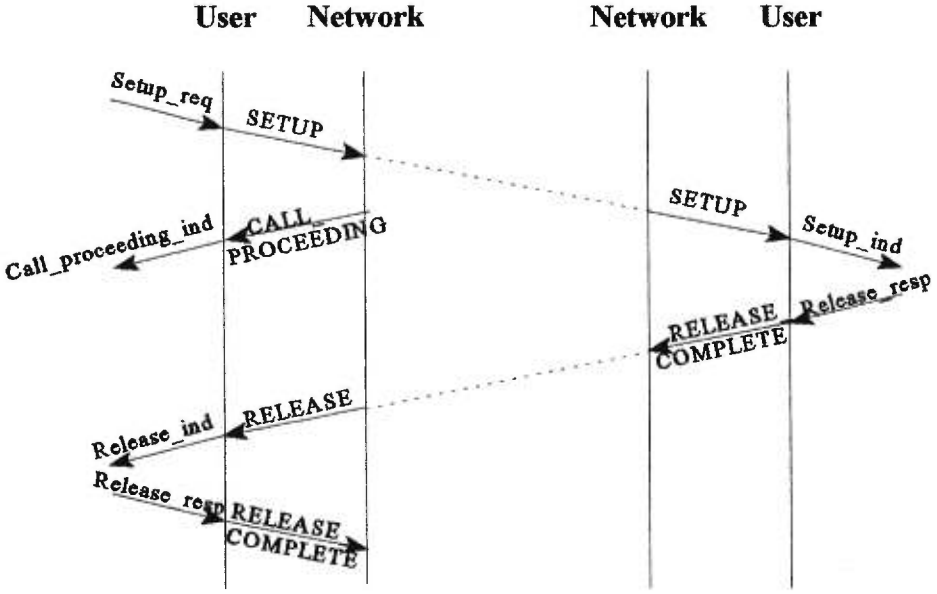


Figure 22: Rejet d'une connexion fait par l'utilisateur appelé

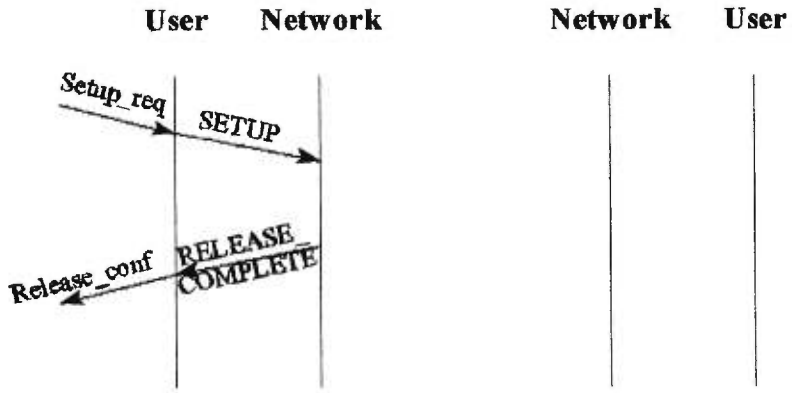


Figure 23: Rejet d'une connexion fait par le réseau

3.4.3.3 Spécification par un automate à états finis

Le comportement de base d'une entité du protocole, du côté de l'utilisateur, peut être représenté par un automate à états finis (FSM, *Finite State Machine*) déterministe (Figure 24), qui est défini par:

- un ensemble d'états:

  U0 - Null State

  U1 - Call Initiate

  U3 - Outgoing call proceeding

  U6 - Call present

  U8 - Connect request

  U9 - Incomming call proceeding

  U10 - Active

  U11 - Release request

  U12 - Release indication;

- un ensemble d'entrées:

  - les primitives: Setup_req, Setup_resp, Proceeding_req, Release_req, Release_resp, InitiateStatusEnquiry;

- les messages: SETUP, CALL_PROCEEDING, CONNECT, CONNECT_ACK, RELEASE, RELEASE_COMPLETE, STATUS, STATUS_ENQUIRY;
- les timers: T303, T308, T310, T313, T322;

• un ensemble de sorties:

- les primitives: Setup_ind, Setup_conf, Setup_complete_ind, Proceeding_ind, Release_ind, Release_conf, Send_cr;
- les messages: SETUP, CALL_PROCEEDING, CONNECT, CONNECT_ACK, RELEASE, RELEASE_COMPLETE, STATUS, STATUS_ENQUIRY.

• une fonction de transition (les flèches) qui, à partir d'un état, sur une entrée, produit une sortie et change d'état.

Par exemple, on dit que dans l'état U0, à l'entrée de la primitive *Setup_req*, l'automate produit comme sortie le message SETUP et passe à l'état U1.

La partie gauche du diagramme - U0, U1, U3 (éventuellement), U10 - est parcourue quand le protocole demande un appel (une connexion), tandis que la partie droite - U0, U6, U9 (éventuellement), U8, U10 - est parcourue quand l'entité de protocole reçoit un appel. Dans l'état U10 la connexion est complètement établie et les deux entités connectées peuvent échanger des données. Selon l'entité qui demande la déconnexion, l'automate passe à l'état U11 ou U12, pour passer ensuite, sur la réception de la réponse appropriée, à l'état initial U0.

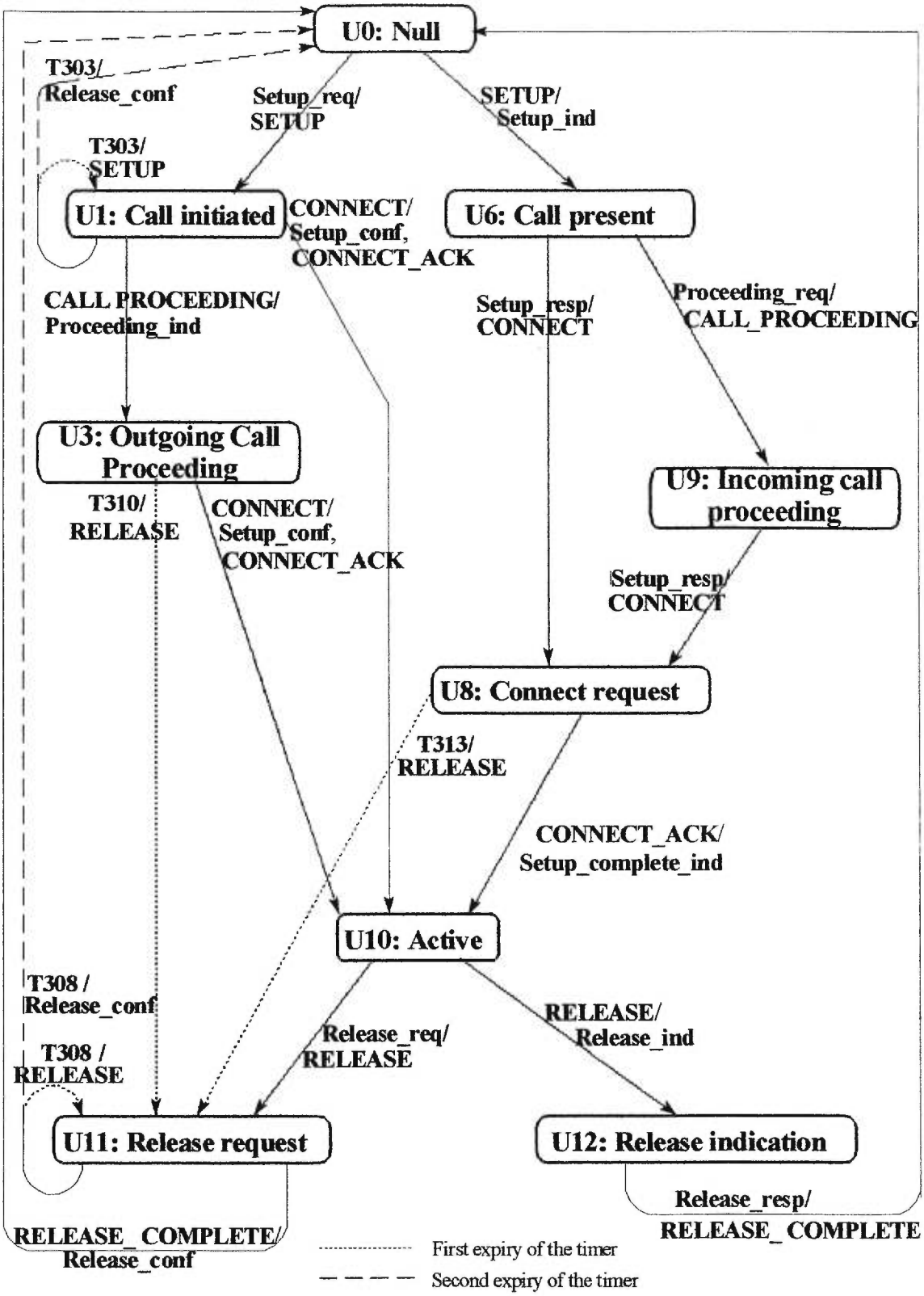


Figure 24: Comportement de base d'une entité de protocole

3.4.3.4 Spécification par un tableau d'état

Un tableau d'états décrit le comportement du protocole de la même façon qu'un automate d'états finis: par un ensemble d'états et de transitions. Il offre, en plus, l'avantage de pouvoir exprimer plus d'aspects du comportement comme par exemple, les conditions qui détermine une transition, des actions à faire sur une certaine entrée (par exemple la libération d'une référence d'appel ou l'affectation d'une valeur à un paramètre) etc.

Le tableau d'états que nous avons construit (Annexe 1) regroupe, pour chaque état du protocole (la colonne **State**), les informations suivantes:

Input - contient les entrées qui arrivent dans l'état donné. Chaque message contient une série de paramètres. Un paramètre peut être obligatoire (M) ou optionnel (O), selon le type du message et l'état dans lequel celle-ci est reçue;

Type - désigne le type de chaque entrée. Une entrée qui se présente dans un état dans lequel il n'est pas supposé qu'elle arrive est appelée entrée inattendue et a le type U (*unexpected*);

Condition - représente le résultat de la vérification faite sur le message reçu et, éventuellement, sur les capacités de l'utilisateur;

Output - contient la ou les sorties déterminées par l'entrée dans la condition correspondante.

Action - décrit la ou les actions entamées;

Next State - est l'état dans lequel l'automate passe sur la réception de l'entrée et dans la condition correspondante;

Ref. - contient la ou les sections dans la norme [2] qui décrit le comportement en question.

4. Spécification du protocole de signalisation en SDL

4.1 Introduction au langage SDL

4.1.1. Historique

SDL - *Specification and Description Language* - est un langage formel développé et normalisé par le CCITT (*Comité Consultatif International pour le Télégraphe et le Téléphone*). Sa première version date de 1976, suivie des versions 1980, 1984, 1988, 1992. Bien que SDL soit développé pour la spécification et la description du comportement des systèmes de télécommunication, il peut être utilisé pour tout système réactif. Il supporte la modularisation du système et couvre différents niveaux d'abstraction.

Il existe plusieurs outils, plus ou moins automatiques, pour traduire SDL en un langage de programmation, dont les plus connus sont GEODE et SDT.

4.1.2. Spécification d'un système

SDL décrit le comportement d'un système sous la forme de stimulus/réaction, étant admis que les stimuli, aussi bien que les réactions, sont des entités discrètes et contiennent de l'information. La spécification d'un système peut être vue comme étant la séquence de réactions à une séquence de stimuli. Le modèle de spécification d'un système est fondé sur la notion de machine d'états finis étendue, i.e. une machine à états finis qui peut manipuler des variables.

SDL utilise des concepts structurels qui facilitent la spécification des grands systèmes et/ou complexes. Il est ainsi possible de subdiviser la spécification d'un système en unités qui peuvent être traitées de manière indépendante.

4.1.3 Formes de représentation

SDL permet deux formes de représentation: textuelle et graphique. La forme textuelle, SDL/PR, fournit une syntaxe textuelle. La forme graphique, SDL/GR, fournit des symboles graphiques pour les constructions SDL. Les deux formes sont équivalentes du point de vue sémantique.

L'importance des outils réside sur l'édition des spécifications SDL en forme graphique et sur la vérification syntaxique et sémantique.

4.1.4 Principes de base

Les notions SDL présentées dans ce chapitre se résument à celles nécessaires à la compréhension de la spécification formelle du protocole de signalisation. Pour plus de détails se référer à la réf. [21].

4.1.4.1 Structure d'un système

Une spécification SDL définit le comportement d'un système qui communique avec son environnement au moyen de messages discrets. Elle couvre différents niveaux d'abstraction.

L'environnement d'un système désigne tout ce qui ne fait pas partie du système. On suppose qu'il interagit avec le système selon la description de ce dernier.

La description du **système** constitue le plus haut niveau d'abstraction: une machine qui communique avec son environnement. La communication se fait par un ou plusieurs **canaux**, uni- ou bidirectionels. Figure 25.

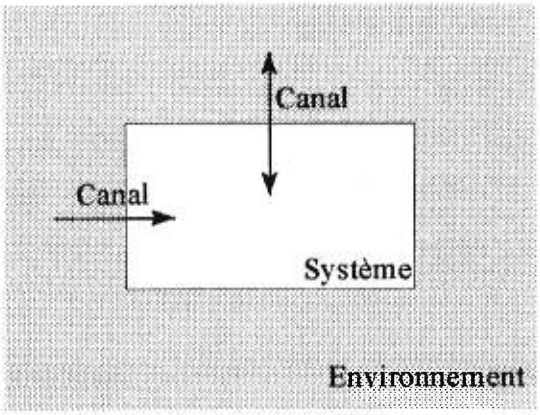


Figure 25: Communication entre un système et son environnement

Un système contient un ou plusieurs blocs. Le **bloc** est un concept de structuration: un objet indépendant de point de vue développement, description, compréhension etc. Il communique avec un autre bloc ou avec l'extérieur par des **canaux** (Figure 26).

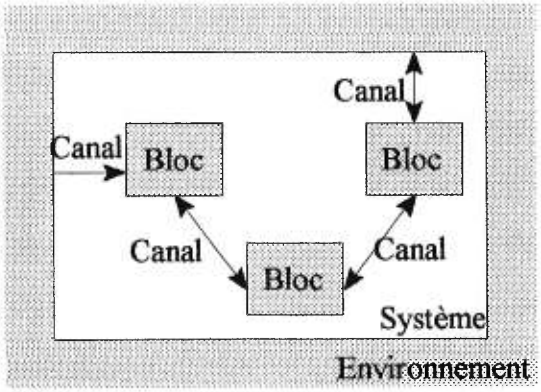


Figure 26: Communication de blocs

Un bloc peut, lui aussi, contenir un ou plusieurs (sous)bloc(s). Il en résulte une structure d'arbre, dont la racine est le système (Figure 27). Un bloc-feuille contient un ou plusieurs processus. Un **processus** est un automate à états finis étendu (EFSM). La communication entre les processus se fait par des **routes** (Figure 28).

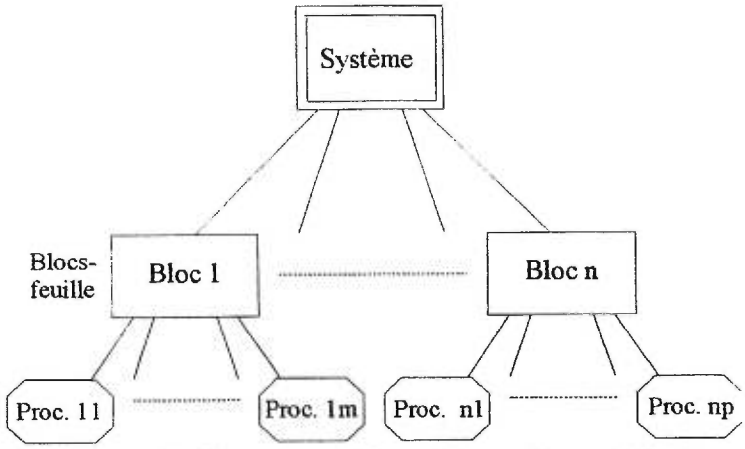


Figure 27: Structure hiérarchique d'un système

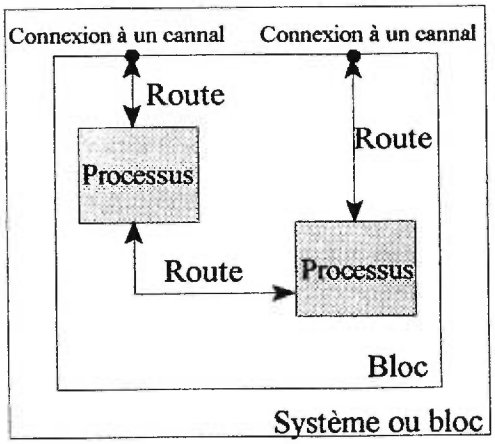


Figure 28: Communication de processus

Un processus peut avoir zéro ou plusieurs **instances**. Le comportement des instances est identique, mais les instances peuvent être dans des états différents, autrement dit, les instances diffèrent de point de vue dynamique, mais pas statique. Le nombre d'instances d'un processus peut varier dans le cycle de vie d'un système. Une instance de processus peut être créée ou détruite.

Le comportement d'un système est la résultante du comportement des processus qu'il contient.

4.1.4.2 Communication

Chaque instance de processus possède un identificateur unique, attribué par le système d'exécution SDL. La communication entre les instances de processus ou avec l'environnement est réalisée de façon asynchrone, par la transmission des messages, appelés **signaux**. Un signal peut transporter des valeurs de données. Chaque instance de processus a une file unique où les signaux qui lui sont adressés par d'autres processus ou par l'environnement sont stockés en ordre d'arrivée. Le premier signal de la file d'une instance de processus est **consommé** par celle-ci, i.e. enlevé de la file (Figure 29). Si le signal fait partie des signaux (entrées) acceptées par l'instance de processus dans l'état courant, alors la transition correspondante est intialisée. Dans le cas contraire, le signal est ignoré (*discard*).

On suppose que l'environnement se comporte "à la façon de SDL", i.e. il est capable d'envoyer et de recevoir des signaux.

Les contraintes de temps apparaissant dans un système peuvent être modélisées par le **mécanisme de timer** qui permet la mise dans la file d'une instance de processus d'un signal spécial, **timer**. Ce signal indique l'expiration d'une certaine durée de temps, écoulée depuis le moment où le timer a été activé.

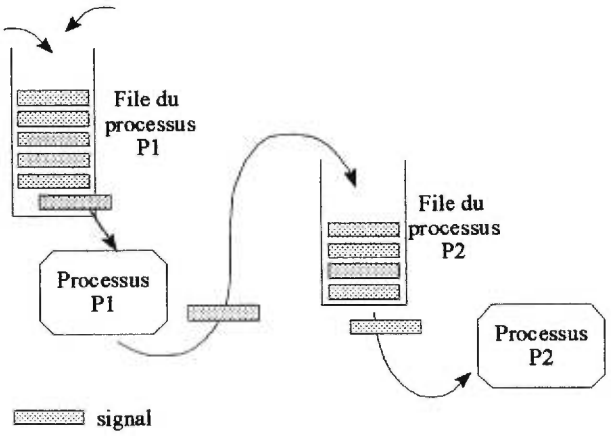


Figure 29: Communication par files

4.1.4.3 Types de données

SDL utilise l'approche "type de données abstraites" (ADT, *Abstract Data Type*), qui définit les types de données en termes de propriétés, donc indépendamment de l'implantation. La définition d'un type de données abstraite comprend trois niveaux:

• un ensemble de valeurs;

• un ensemble d'opérateurs appliqués à ces valeurs;

• un ensemble de règles algébriques qui définissent les opérateurs.

SDL fournit un ensemble de types prédéfinis. D'autres types peuvent être dérivés des types existants ou entièrement définis.

Les **types prédéfinis simples** sont:

• entier (*Integer*);

• booléen (*Boolean*);

• réel (*Real*);

• naturel (*Natural*)

• caractère (*Character*)

• temps (*Time*) - représente un instant du temps absolu;

• durée (*Duration*) - représente la différence entre deux valeurs de type temps;

• identificateur de processus (*Pid*) - utilisé dans la gestion des instances de processus.

Un **type de données utilisateur** (*user-defined*) est défini entre les mots clé *Newtype* et *Endnewtype*. Les valeurs ou constantes du type sont définies (s'il y en a) par des littéraux (*Literals*). Les opérateurs (s'il y en a) sont introduits par le mot clé *Operators* et les équations qui les définissent par *Axioms*. Le mot clé *Ordering* permet d'accéder à des opérateurs implicites. Utilisé dans un type énuméré, il indique que les littéraux sont écrits en ordre et donne l'accès aux opérateurs: <, <=, >, >=.

Une **structure** est un type dont l'ensemble de valeurs est composé à partir des valeurs des champs (le produit cartésien). En tant que nouveau type, elle est définie entre les mots clé *Newtype* et *Endnewtype* et comprend:

• l'identificateur du type structure (i.e. le nom);

• le mot clé *Struct*;

• les identificateurs et les types des champs;

Un **générateur de type** permet de définir un squelette de texte paramétré qui sera utilisé pour la définition de nouveaux types semblables. Par exemple, les files (queues) possèdent les mêmes propriétés indépendamment du type des éléments qu'elles contiennent. Un générateur de files permet de décrire le comportement général d'une file. Le type des éléments constitue le paramètre dont la valeur sera transmise à l'appel du générateur. SDL fournit trois générateurs prédéfinis: de tableaux (*Array*), d'ensembles (*Powerset*), et de chaînes (*String*).

Un **syntype** (*Syntype*) introduit un sous-ensemble de valeurs d'un type.

Un **synonyme** (*Synonym*) donne un nom à une expression.

Les **variables**, comme dans les langages de programmations, manipulent des données. Une variable est déclarée par le mot clé *dcl*, est associée à un type et on lui attribue une valeur qui peut être modifiée par une nouvelle affectation.

Les **opérateurs impératifs** retournent des valeurs d'état de certains éléments du système sous-jacent. On mentionne les suivants:

• **now** - est une expression de type *Time* et permet d'accéder à l'horloge du système pour déterminer le temps absolu du système;

• les quatre expressions de type *Pid* associées à chaque instance de processus:

self - donne l'identificateur de l'instance de processus;

parent - donne l'identificateur de l'instance du processus créateur;

offspring - donne l'identificateur de l'instance de processus la plus récente créée par l'instance de processus;

sender - donne l'instance de processus en provenance de laquelle le dernier signal entrant a été utilisé.

• **active** - est une expression booléenne qui indique si le timer auquel elle s'applique est actif (i.e. a été enclenché sans avoir expiré) ou non.

4.1.5. Constructions SDL

Dans la présente section nous présenteront, de façon simplifiée, la forme graphique et les mots clé de la forme textuelle des constructions SDL utilisées dans la spécification formelle du protocole de signalisation.

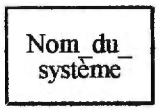
4.1.5.1 Structure du système

Système

La définition d'un système (située entre les mots clé *System* et *Endsystem*) est la représentation en SDL d'une spécification ou de la description d'un système.

Les définitions de tous les signaux, canaux, types de données, types de synonyme utilisés dans l'interface avec l'environnement et entre les blocs du système sont contenus dans la définition du système.

Bloc



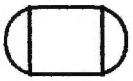
La définition d'un bloc (située entre les mots clé *Block* et *Endblock*) est la représentation en SDL d'une spécification ou de la description d'un bloc.

Processus

Un processus SDL (défini entre les mots clé *Process* et *Endprocess*) est un automate à états finis étendu. Sa définition spécifie un ensemble d'instances de processus. Le nombre d'instances d'un processus est indiqué en donnant les limites inférieure, *n*, et supérieure, *m*. L'absence des limites désigne un processus mono-instancié.

Procédure

Une procédure représente un moyen de donner un nom à un ensemble de constructions et de le représenter par une référence unique. Elle s'avère utile pour éviter les répétitions des parties de processus ou pour des calculs répétés. Sa description est donc très semblable à celle d'un processus. Les constructions caractéristiques d'une procédure sont:



Début (*Start*) - représente le point d'entrée dans une procédure.

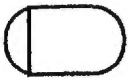
Retour (*Return*) - représente la sortie de la procédure.

Une procédure peut avoir des paramètres formels (*fpar*) d'entré/sortie (*in/out*) ou seulement d'entrée (*in*) et des variables locales (*dcl*).
L'appel d'une procédure provoque la création d'une instance de procédure où les paramètres formels prennent les valeurs données par les paramètres actuels (effectifs) de l'appel et les variables locales sont créées. Au retour de la procédure les paramètres de sortie auront les valeurs respectives obtenues au corps de la procédure et les variables locales cesseront d'exister.

Une définition référencée (*Referenced*) est une définition qui a été déplacée de son contexte pour avoir une meilleure vue d'ensemble à l'intérieur d'un système. Par exemple, on réfère les blocs à l'intérieur du système, les processus à l'intérieur des blocs et les procédures afin de mieux observer la structure de système.

Macro

Le concept de macro en SDL est similaire à celui des langages de programmation. Une macro définit un ensemble d'unités lexicales (SDL/PR) ou de symboles graphiques (SDL/GR) qui peuvent apparaître à un ou plusieurs endroits de la spécification. Un tel endroit est indiqué pas un appel de macro. Une macro peut contenir des paramètres (*fpar*).

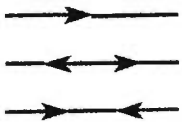


Inlet - représente le point d'entrée dans une macrodéfinition.

Outlet - représente le point de sortie.

4.1.5.2 Communication

Canal



Le canal (*Channel*) est un moyen de transport pour les signaux entre deux blocs ou entre un blocs et l'environnement. Au point de destination d'un canal les signaux se présentent dans le même ordre que celui au point d'origine. Les signaux arrivant simultanément sont ordonnés arbitrairement.

Route

La route (*Signalroute*) est un moyen de transport pour les signaux dont au moins un des points extrêmes est un processus.

Les canaux et les routes ont des noms et portent des signaux (*Signal*). Une instance de signal est un flot d'information entre des processus ou entre un processus et l'environnement. Une liste de signaux (*Signallist* en SDL/PR et *[ ]* EN SDL/GR) est une façon abrégée d'énumérer les identificateurs de signaux.

Connexion

Une connexion (*Connect*) l'élément d'information les canaux avec d'autres canaux ou avec des routes d'un autre niveau de structuration

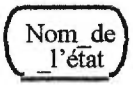
4.1.5.3 Comportement

Les constructions SDL d'un processus sont:

Début

Début ou départ (*Start*) - désigne le point d'entrée au processus.

État



Un état du processus représente une condition dans laquelle un processus peut traiter une instance de signal en exécutant une transition. Une transition change le processus d'un état (*State*) à un autre (*Nextstate*). Un astérisque '*' à la place du nom de l'état désigne tous les états (utilisé dans *State*). Toutefois l'exclusion de certains états peut être signifiée en mettant leurs noms entre parenthèses, à la suite de l'astérisque. Un trait '-' à la place du nom de l'état désigne le même état (utilisé pour *Nextstate*).

Entrée

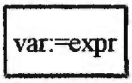
Une entrée (*Input*) indique la consommation du signal désigné. Le traitement du signal d'entrée rend l'information véhiculée par le signal disponible pour le processus.

Sortie

Sortie (*Output*) - indique l'envoi du signal désigné. Dans le but d'éviter les ambiguïtés, une sortie peut être dirigée, autrement dit envoyée explicitement sur une route ou à une instance de processus, en utilisant les mots clé *via* et *to*, respectivement. La route est désignée par son nom, tandis que l'identité d'une instance de processus peut être identifiée en utilisant une des expressions pré-définies: *offsping*, *self*, *parent* ou *sender*.

Les signaux, d'entrée ou de sortie, peuvent contenir des paramètres en tant que transporteurs de valeurs d'un processus à un autre.

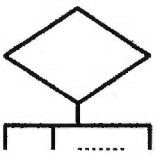
Tâche



Une tâche peut contenir des assignations ou un texte informel. Le rectangle désigne soit l'affectation des variables (*Task*), soit l'activation/désactivation des timers. Un timer *T* est activé par la construction SET(*now+durée*, *T*), où

now donne le moment courant, *durée* représente la période d'expiration du timer (en unités de temps) et *T* est le nom du timer. Il est désactivé par RESET(*T*).

Décision



Une décision (entre les mots clé *Decision* et *Enddecision*) permet de contrôler le flux d'exécution. La valeur de l'intérieur de la décision est comparée aux valeurs associées aux branches de réponse. Une seule réponse doit être correcte et pour s'en assurer on peut utiliser le mot clé ELSE comme variante de réponse.

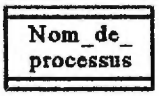
Étiquette

Étiquette représente le point d'entrée d'un transfert de contrôle.

Branchement

Un branchement (*join*) transfère le contrôle vers l'étiquette désignée.

Création

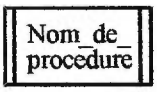


Création (*Create*) provoque la création dynamique d'une instance du processus désigné réalisée pendant une transition du processus créateur. L'identificateur de l'instance la plus récemment créée est donné par l'expression prédéfinie *offspring*.

Arrêt

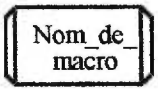
Arrêt (*Stop*) provoque la destruction de l'instance qui le contient.

Appel de procédure



Appel de procédure (*Call*) - transfère l'interprétation vers la définition de procédure indiquée.

Appel de macro

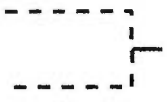


Appel de macrodéfinition (*Macro*)

Alternative

L'alternative (entre les mots clé *Alternative* et *Endalternative*) a la même signification que la décision, mais son action peut être vue plutôt comme une option de compilation.

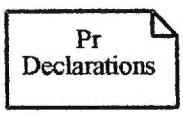
Commentaire



Un commentaire (*Comment*) est un texte (commentaire) associé à des symboles.

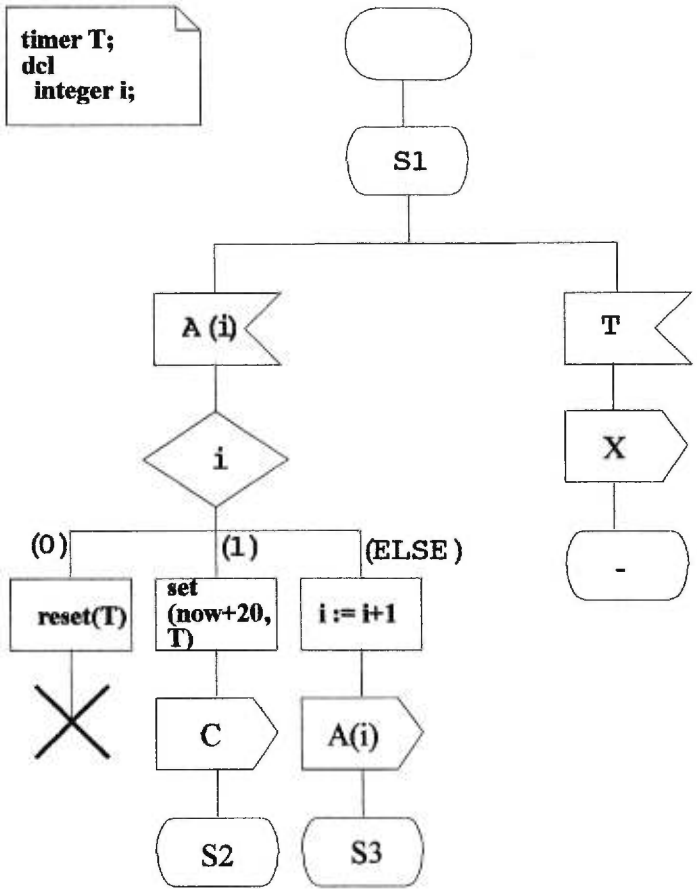
4.1.5.4 Déclarations

Symbole de texte



La déclaration des types de données, des variables (*dcl*) et des timers (*timer*) se fait en forme textuelle, dans une zone spécialement identifiée *Pr Declarations*, au plus haut niveau qui les utilise.

4.1.5.5 Exemple



L'état initial du processus est S1. Dans cet état, l'expiration du timer T provoque l'envoi du signal X sans changement d'état. Lors de la réception du signal A, selon la valeur du paramètre i du signal, le processus:

- désactive le timer T et détruit l'instance courante;
- active le timer T, envoie le message C et passe à l'état S2;

ou

- incrémente la variable i, envoie le signal paramétré A et passe à l'état S3.

4.2 Approche du problème

4.2.1. Suppositions et restrictions

Dans le développement du protocole de signalisation à l'interface utilisateur-réseau, du côté de l'utilisateur que nous avons fait, on a introduit les hypothèses et les restrictions suivantes:

• une connexion SAAL en mode assuré entre l'utilisateur et le réseau est déjà établie;

• le service fourni à la couche supérieure (non-spécifié dans le document de référence) est assuré par un ensemble de primitives de service comme suit:

- chaque message d'établissement et de libération de connexion est envoyé suite à la réception d'une primitive;
- chaque message d'établissement et de libération de connexion reçu est rapporté à la couche supérieure par une primitive;

• la vérification de l'état de l'entité paire est initiée par une primitive de service;

• on considère seulement les connexions point-à-point;

• les procédures de redémarrage (restart) ne sont pas incluses;

4.2.2. Architecture

Le fait que le protocole puisse supporter plusieurs connexions dont le comportement est le même suggère la pluri-instanciation (de 0 à n) du processus qui modélise ce comportement. Une entité de protocole crée une instance de processus chaque fois qu'elle accepte une demande de connexion ou qu'elle en initie une. Une instance de processus reçoit, à sa création, un identificateur unique, de type *Pid*, qui lui est attribué par le système d'exécution SDL. Par ailleurs, tous les messages contiennent une référence d'appel qui identifie, de façon unique, la connexion à laquelle ils appartiennent. Une entité de protocole qui reçoit un message SETUP doit associer l'identificateur de l'instance de processus créée à cette occasion à la référence d'appel du message pour pouvoir acheminer les autres

messages et les primitives visant la même connexion. Une entité qui demande une connexion doit générer la référence d'appel qu'elle va mettre dans le message SETUP envoyé au réseau. Pour les mêmes raisons de routage, elle aussi doit sauvegarder la correspondance entre la référence d'appel et l'identificateur d'instance de processus. La forme d'organisation des paires (référence d'appel, identificateur d'instance de processus) doit permettre une recherche rapide selon la référence d'appel. Nous avons opté pour une structure d'arbre binaire.

Nous avons regroupé les aspects de gestion dans un processus propre, appelé coordinateur. Ils concernent:

• la construction de la référence d'appel par la génération de la valeur et l'affectation de la valeur 0 au drapeau;

• la sauvegarde de chaque nouvelle paire (référence d'appel, identificateur d'instance de processus) dans un arbre binaire;

• la recherche d'un élément de l'arbre selon la référence d'appel. L'identificateur d'instance associé à la référence d'appel est utilisé pour le routage des messages et des primitives vers l'instance de processus appropriée.

Les messages sont définis comme une chaîne de bits structurée en octets. Il est nécessaire de pouvoir identifier et interpréter chacune des composantes, autrement dit de décoder le message. L'opération inverse, le codage, transforme une structure logique en une chaîne de bits. Pour des raisons d'optimisation, en termes de rapidité, nous avons choisi de réaliser le décodage et le codage en langage C et de les intégrer lors de l'implantation. Par conséquent, ces opérations ne seront pas testées lors de la simulation. Bien que le codage et le décodage puissent être faits au fur et à mesure que les composantes d'un message sont utilisées, nous les avons regroupés dans un processus propre. Ainsi chaque message est entièrement décodé avant d'initier les actions correspondantes. Cette modularité nous a permis l'exécution de cas de test avant l'implantation, par l'exclusion de ce processus.

L'architecture du protocole est montrée sur la figure 30:

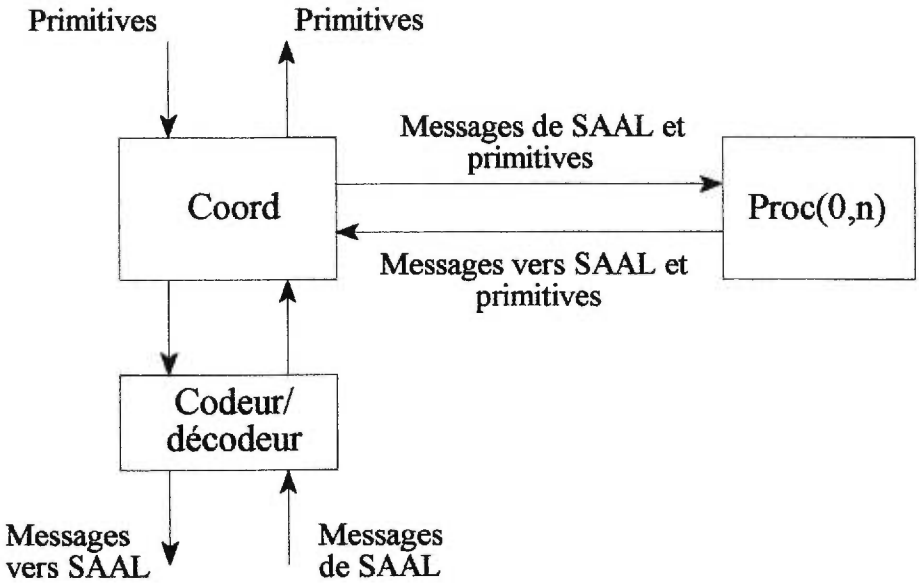


Figure 30: Architecture de protocole de signalisation ATM

Le format des messages véhiculés entre le Codeur/Décodeur et le SAAL est une chaîne de bits organisés en octets. Une chaîne entrante est transformée en une structure logique. Cette structure, ainsi que les identificateurs des éléments d'information dont le contenu est détecté étant invalide (par exemple, la valeur de la longueur diffère de la longueur effective de l'élément d'information) et les valeurs non-reconnues des éléments d'information seront envoyées au coordinateur. En direction contraire, le Codeur/Décodeur reçoit du coordinateur une structure logique, qu'il transforme en chaîne d'octets afin d'être transmise à l'entité paire.

La structure d'un message véhiculé entre le Codeur/Décodeur (Figure 31) et le coordinateur comprend:

- le discriminateur du protocole,
- la référence d'appel,
- le type du message,
- la longueur décodée,

• tous les éléments d'information acceptés par le présent protocole, chacun avec le nombre maximum d'occurrences.

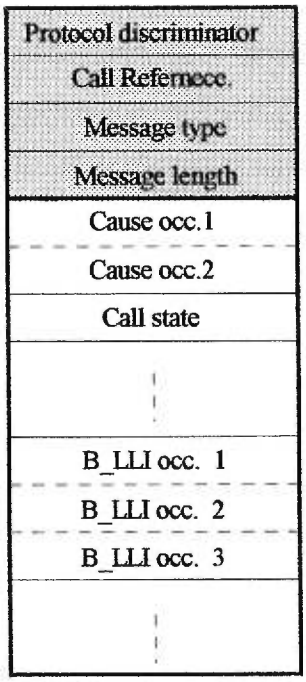


Figure 31: Structure des messages

Pour accomplir son rôle de gestion et d'aiguillage, le coordinateur n'a besoin d'inspecter que l'en-tête (la partie grisée sur la figure 31) des messages qui lui sont envoyés et initie des actions selon le contenu de cette partie. Puisqu'une connexion est caractérisée par les éléments d'information contenus dans les messages, une instance de processus est intéressée seulement par ceux-ci. Par conséquent, le coordinateur enlève du message la partie qu'il a utilisée. Selon le type du message et le résultat des vérifications de ses éléments d'information, une instance de processus initie les actions appropriées.

La structure d'une primitive comprend les éléments d'information définis dans le message auquel elle correspond (voir tableau i, page 39). Par exemple, *Setup_resp* (comme *Setup_conf* d'ailleurs) correspond au message CONNECT. Les éléments d'information permis à l'intérieur de ce message sont *AAL parameters*, *Broadband low layer information* et *Connexion identifier*. Par conséquent, la structure de la primitive *Setup_resp* contiendra ces trois champs.

Les primitives sont transportées par le coordinateur sans aucun traitement, à l'exception de la demande de connexion *Setup_req* pour laquelle on vérifie si le protocole peut accepter une nouvelle connexion, i.e. le nombre maximum de connexions n'est pas atteint. Si tel est le cas, le coordinateur génère une référence d'appel unique, crée une instance de processus et lui envoie la primitive *Connection_req* qui contient, outre les paramètres reçus dans la primitive *Setup_req*, la référence d'appel générée.

4.3 Spécification SDL

La spécification du protocole de signalisation en langage SDL dans les deux formes, graphique et textuelle, est représentée dans les annexes 2 et 3 respectivement.

4.3.1. Description du système

La description du **système** - *ATM_UNI_SP* - constitue le plus haut niveau d'abstraction: une machine qui communique avec l'environnement, ainsi que la description des types de données - *PRDeclaration* - qui seront visibles dans tout le système et par l'environnement.

4.3.1.1 Interconnexion

Le système contient un seul **bloc** - *Q2931* - qui communique avec l'environnement par des canaux: *to_AP* et *from_AP* pour l'échange de primitives avec la couche supérieure (le programme d'application) et *to_SAAL* et *from_SAAL* pour l'échange de messages et primitives avec la couche inférieure (SAAL).

Les signaux échangés entre le système et le programme d'application portent les noms des primitives.

Les signaux qui arrivent du programme d'application - *Setup_req*, *Proceeding_ind*, *Setup_resp*, *Release_req*, *Release_resp* - forment la liste de signaux *primitives_from_AP*.

Les signaux envoyés au programme d'application - *Setup_ind*, *Proceeding_req*, *Setup_conf*, *Setup_complete_ind*, *Release_ind*, *Release_conf*, *Send_cr* - forment la liste de signaux *primitives_to_AP*.

À l'exception de *Setup_req* et *Send_cr*, tout signal représentant une primitive possède deux paramètres:

• la référence d'appel;

• une structure dont les champs sont les éléments d'information du message auquel la primitive correspond (voir tableau i, page 39).

Par exemple, *Release_ind*, correspond au message RELEASE arrivé du réseau. Le seul élément d'information permis à l'intérieur de ce message est *Cause*. Par conséquent, le signal *Release_ind* aura comme paramètre, outre la référence d'appel, une structure dont le seul champ est la cause.

Une demande de connexion, *Setup_req*, ne contient pas de référence d'appel. C'est au protocole d'en générer une et elle sera annoncée à la couche supérieure en tant que paramètre du signal *Send_cr*.

Puisque la vérification de l'état de l'entité paire est faite, généralement, suite à une défaillance dans les couches inférieures (si, par exemple, une interruption au niveau de la couche liaison de données est signalée), nous avons considéré que la primitive *InitiateStatusEnquiry* arrive par le canal *from_SAAL*. Son seul paramètre est la référence d'appel.

Le signal *AAL_DATA_ind* arrivé de SAAL possède deux paramètres:

• la chaîne de bits représentant le message reçu;

• la longueur de la chaîne, en octets.

Le signal *AAL_DATA_req* transporte en tant que paramètre la chaîne de bits représentant le code du message à envoyer.

4.3.1.2 Description des types de données

Pour chaque élément d'information nous avons défini une structure dont les champs sont les composantes définies dans la spécification. Dans le cas des éléments d'informations dont la répétition à l'intérieur d'un message n'est pas permise, cette structure contient un

champ supplémentaire, *presence*, de type booléen, qui indique la présence ou l'absence de l'élément d'information respectif dans le message. Les éléments d'informations dont la répétition est permise sont des structures de deux champs:

• un entier naturel, *nb_occ*, indiquant le nombre d'occurrences de l'élément d'information dans le message;

• un tableau de structures (correspondant à l'élément d'information respectif) qui représente les occurrences proprement-dites.

Par exemple, nous avons défini les structures:

```
NEWTYPE connection_id_type STRUCT
  presence boolean;
  VPAS int_2;
  pref_excl int_3;
  VPCI natural;
  VCI natural;
ENDNEWTYPE;
```

```
NEWTYPE B_LLI_ie STRUCT
  coding_standard int_2;
  info_layer_1 optional_value_type;
  info_layer_2 layer_2_type;
  info_layer_3 layer_3_type;
ENDNEWTYPE;
```

On observe que l'élément d'information *Connection Identifier* contient un champ supplémentaire, *presence*. Par contre la structure l'élément d'information *Broadband low layer information* ne le contient pas, mais pour modéliser sa possible présence multiple à l'intérieur d'un message on définit la structure:

```
NEWTYPE B_LLI_type STRUCT
  nb_occ max_occ_B_LLI;
  occ B_BLLI_table;
ENDNEWTYPE;
```

ou:

nb_occ indique le nombre d'occurrences de l'élément d'information B_LLI dans le message; *B_LLI_table* un tableau de type *B_LLI_ie* qui contient les occurrences de l'élément d'information:

```
NEWTYPE B_LLI_table
 array(max_occ_B_LLI, B_LLI_ie);
ENDNEWTYPE;
```

Le type *max_occ_B_LLI* est défini comme ensemble des entiers positifs 0, 1, 2, 3 comme suit:

```
SYNTYPE max_occ_B_LLI = NATURAL
  CONSTANTS 0:3
ENDSYNTYPE;
```

Les sous-éléments d'un élément d'information, eux aussi, peuvent être présents de façon obligatoire ou optionnelle. Dans le premier cas un sous-élément d'un élément d'information est soit un type simple, soit une structure, selon la spécification. Par exemple, *coding_standard*, qui doit être toujours présent est un type simple: un entier positif sur 2 bits. Dans le cas d'une présence optionnelle, un sous-élément est défini comme une structure qui contient, outre l'information définie dans la spécification, un champ supplémentaire indiquant sa présence dans l'élément d'information. Par exemple, dans l'élément d'information *B_LLI*, l'information pour la couche 1, en fait un entier positif, est optionnelle. Par conséquent, le champ qui la représente, *info_layer_1*, est défini comme une structure de type *optional_value*, qui contient le champ indiquant la présence, *presence*, et la valeur proprement-dite, *value*.

Les structures sont définies, de façon récursive, jusqu'aux types simples, en ajoutant, si besoin, le champ supplémentaire de présence. Exemple:

```
NEWTYPE layer_2_type STRUCT
   presence boolean;
   value int_5;
   codings codings_type;
   window_size optional_value_type;
   user_spec optional_value_type;
ENDNEWTYPE;
```

```
NEWTYPE codings_type STRUCT
   presence boolean;
   mode int_2;
   q933 int_2;
ENDNEWTYPE;

NEWTYPE optional_value_type STRUCT
 presence boolean;
 value natural;
ENDNEWTYPE;
```

Les structures représentant des éléments d'information apparaissent comme champs dans différentes autres structures. En effet, au niveau du système, nous avons construit, pour chaque message défini dans le protocole, une structure dont les champs sont ses éléments d'information. Une telle structure constitue le type du deuxième paramètre des primitives correspondant au message (voir tableau 1). Par exemple, le message CONNECT permet la présence des éléments d'information *AAL parameters*, *Broadband low layer information* et *Connexion identifier*. La structure associée est:

```
NEWTYPE connect_struct STRUCT
   AAL_param AAL_param_type;
   B_LLI B_LLI_type;
   connection_id connection_id_type;
ENDNEWTYPE;
```

Elle est le type du deuxième paramètre des primitives *Setup_resp* et *Setup_conf*.

Une série de déclarations de type *syntype* regroupe les entiers dont la représentation binaire a la même longueur. Par exemple, le *syntype* appelé *int_3* représente les entiers dont la représentation binaire se fait sur 3 bits, i.e. les entiers de 0 à 7.

Nous avons déclaré un type *synonym* entre chaque type de message et sa valeur. Par exemple, un message SETUP est indiqué par la valeur 5 assigné au champ *Message Type*. La déclaration correspondante est:

```
SYNONYM SETUP Natural = 5;
```

La référence d'appel est déclarée comme structure, *call_reference_type*, dont les champs sont:

- le drapeau (*flag*), un entier sur un bit (de type *flag_type*);
- la valeur, un entier sur 23 bits (de type *int_23*).

Un message arrivant du SAAL est représenté par un tableau d'entiers en format hexadécimal, *hexa_array_type*. On a limité l'index du tableau à la longueur maximale du message (328 octets).

```
SYNTYPE max_len_mess = NATURAL
  CONSTANTS 0:328
ENDSYNTYPE;

NEWTYPE hexa_array_type
  Array(max_len_mess, hexadecimal);
ENDNEWTYPE;
```

De façon similaire on a déclaré un tableau d'entiers naturels, *natural_array_type*.

Le type *occ_array_type* est une structure dont les champs sont:

- un tableau d'entiers naturels, *occ*;
- le nombre d'éléments contenus dans le tableau, *nb_occ*.

4.3.2. Description du bloc Q.2931

4.3.2.1 Interconnexion des composantes

Le bloc contient, trois processus:

- *Coder_Decoder* - le codeur/décodeur qui décode tout message arrivé du réseau (via SAAL) et code tout message à envoyer vers le réseau;
- *Coord* - le coordinateur qui gère les connexions et achemine les entrées vers la connexion appropriée;
- *Proc* - dont une instance modélise une connexion.

Tous les signaux représentant des primitives de service qui entrent dans le bloc, sont dirigés vers le coordinateur afin d'être acheminés vers la connexion appropriée. Le signal entrant *SAAL_ind* est dirigé vers le codeur/décodeur pour que le message qu'il transporte soit décodé.

Le signal *mess_rec* transporte, du codeur/décodeur vers le coordinateur, le message décodé et les identificateurs des éléments d'information mal reçus.

En direction contraire, le signal *mess_send* est envoyé avec le message à coder.

Toutes les primitives véhiculées entre le coordinateur et une instance de processus possède deux paramètres:

• la référence d'appel qui identifie la connexion;

• une structure dont les champs sont les éléments d'information admis dans le message correspondant.

(La primitive *Setup_req* a été transformée en *Connection_req* par l'ajout du paramètre de type référence d'appel généré par le coordinateur.)

Les signaux représentant des messages véhiculés entre le coordinateur et une instance de processus transportent les paramètres suivants:

• la référence d'appel qui identifie la connexion;

• la structure contenant tous les éléments d'information;

• les identificateurs des éléments d'information erronés détectés au décodage.

4.3.2.2 Déclaration des types de données

On déclare au niveau bloc les types de données utilisées à ce niveau ou à un niveau inférieur (sous-blocs et processus du même bloc).

Le type *message_content_type* correspond au format du message (figure 31) véhiculé entre le codeur/décodeur et le coordinateur. Il est une structure dont les champs sont:

• le discriminateur de protocole, *pr_disc*;

• la référence d'appel, *CR*, de type *call_reference_type*;

• le type du message, *message_type*, un entier positif sur 8 bits;

• une structure, *ie_type*, dont les champs sont tous les éléments d'information acceptés par le protocole de signalisation.

Le type mode ensembliste (*powerset*) *ie_values_set* dont les éléments sont des entiers entre 8 et 120 (les identificateurs des éléments d'information) permet la construction des ensembles d'éléments d'information et supporte des opérations spécifiques:

```
SYNTYPE ie_values_type = NATURAL
  CONSTANTS 8:120
ENDSYNTYPE;

NEWTYPE ie_values_set
  POWERSET (ie_values_type)
ENDNEWTYPE;
```

Le type *ie_array_type* est un tableau dont les éléments sont de type booléen. Le type de l'index, *ie_id*, représente une énumération des éléments d'information acceptés par le protocole de signalisation:

```
NEWTYPE ie_id
  LITERALS ie8, ie20, ie88, ie89, ie90, ie92, ie93, ie94, ie95, ie98, ie99, ie108, ie109, ie112, ie113, ie120
  OPERATORS ORDERING;
ENDNEWTYPE;
```

(Le mot clé ORDERING permet d'accéder à des opérateurs implicites, tel que le successeur d'un élément de l'énumération (*succ*).) La valeur *true* d'un élément du tableau indique que l'élément d'information représenté par l'index a une certaine propriété. Par exemple, si *ic_ie* est une variable de type *ie_array_type*, l'affectation *ic_ie[ie8]:=true* faite dans le contexte du décodage signifie que l'élément d'information *Cause* (dont l'identificateur est 8) possède une erreur de contenu.

La durée de chaque timer est déclarée en tant que synonyme. Par exemple, la durée du timer T303 étant de 4 secondes on a:

```
SYNONYM d_T303 Natural = 4;
```

Bien qu'une macro puisse être declarée à tout niveau d'une spécification et être visible dans tout le système, pour une meilleure lisibilité on l'a faite au niveau de l'appel. Ainsi, la déclaration de la macro *init_ies* a été faite au niveau bloc pour suggérer qu'elle est

appelée par au moins deux processus du bloc (les macros appelées par un seul processus sont déclarées au niveau du processus respectif).

La macro *init_ies* initialise tous les éléments d'information d'une variable *ie* de type *ie_type* comme étant absents. Autrement dit, tous les champs de présence deviennent *false* ou *0*, selon le type.

4.3.3. Description du processus Coder_Decoder

Le codeur/décodeur reçoit de SAAL le signal *SAAL_ind* dont les paramètres sont le tableau d'entiers en format hexadécimal *ht* et le nombre d'éléments du tableau, qui représente la longueur du message, *ht_length*. Si la longueur n'est pas suffisante pour inclure la partie commune à tous les messages (i.e. *ht_length* < 9) ou si le deuxième octet n'a pas la valeur 3, le signal est ignoré. En cas contraire, le processus appelle la procédure de décodage, *decode*. La procédure identifie dans le tableau les composantes du message et attribue leurs valeurs aux champs correspondants. Elle retourne:

- le message décodé, *mess* de type *message_content_type*, dont seuls les éléments d'information correctement reçus sont remplis par la procédure;
- l'ensemble des identificateurs des éléments d'information dont le contenu est invalide, *ic_ie* de type *ie_values_set*;
- les identificateurs des éléments d'information qui ne sont pas reconnus, *u_ie* de type *occ_array_type*.

Les valeurs retournées par la procédure sont envoyées au coordinateur en tant que paramètres du signal *mess_rec*.

Dans la direction contraire, le codeur/décodeur reçoit du coordinateur le signal *mess_send* dont le paramètre est le message à envoyer *mess*. Le message, une structure de type *message_content_type*, est transformé en tableau d'entiers hexadécimaux par la procédure de codage, *code*.

Dans notre spécification SDL, les procédures *decode* et *code* sont vides. Lors de l'implantation, elles seront remplacées par des fonctions écrites en langage C.

4.3.4. Description du processus Coord

Réception des messages

Le coordinateur vérifie tout message *in_message* qui lui arrive du codeur/décodeur comme un des paramètres du signal *mess_rec*. Il ignore les messages dont le discriminateur de protocole n'indique pas le protocole Q.2931 et ceux dont tous les bits de la valeur de la référence d'appel sont égaux à 1 (*dummy_cr*) ou à 0 (référence globale).

Si le message est accepté, après avoir changé la valeur du drapeau, le coordinateur cherche, selon la référence d'appel, *cr*, l'instance de processus à laquelle le message est adressé. L'opération de recherche est désignée par la procédure *find_proc*. Cette procédure, vide dans notre spécification SDL, sera remplacée lors de l'implantation, par une fonction C. Une recherche fructueuse attribue la valeur *true* à la variable *active_call* et l'identificateur de l'instance de processus visée à la variable *Proc_id*. Dans le cas où la référence d'appel serait inactive, la variable *active_call* aura la valeur *false*.

Un message *in_message* de type SETUP dont la référence d'appel est déjà utilisée, i.e. *active_call* a la valeur *true*, constitue une erreur et le message est ignoré. Un message SETUP est également ignoré si, à la réception, le drapeau de sa référence d'appel n'était pas égal à 0. S'il n'y a pas d'erreur et si le cas maximum de connexions supportées n'est pas atteint, i.e. *more_connections* est *true*, le coordinateur crée une nouvelle instance de processus, sauvegarde la correspondance entre la référence d'appel reçue et l'identificateur de l'instance créée (*offspring*) et envoie un signal SETUP à l'instance qui vient d'être créée. (L'opération de sauvegarde est indiquée par la procédure *put* et son contenu, écrit en langage C, sera inséré lors de l'implantation.) Si le protocole ne supporte plus de connexions, le message *out_message* de rejet de la connexion sera construit et envoyé vers l'entité paire. Il sera de type RELEASE_COMPLETE et contiendra la cause du rejet.

Tout message *in_message* reconnu, autre que SETUP, dont la référence d'appel est inactive provoque la libération de la connexion à l'interface locale, c'est à dire l'envoi d'un message, *out_message*, ayant le type RELEASE_COMPLETE et la valeur de la cause égale à 81 (*invalid call reference value*). Si la référence d'appel est active, le coordinateur envoie

à l'instance de processus appropriée (dont l'identificateur *Proc_id* a été trouvé par la fonction *find_proc*) un signal correspondant à la valeur trouvée dans le champ *in_message!message_type*. Ses paramètres sont:

- la référence d'appel *in_message!CR*;
- la structure contenant les éléments d'information du message reçu, *in_message!ie*;
- les identificateurs des éléments d'information dont les contenu a été trouvé erroné lors du décodage, *ic_ie*;
- les identificateurs des éléments d'information non-reconnus lors du décodage, *u_ie*;

Un message dont le type ne fait pas partie de ceux acceptés par le protocole provoque l'envoi vers l'instance de processus d'un signal appelé UNRECOGNIZED dont les paramètres sont la référence d'appel et la valeur trouvée dans le champ *message_type*.

Tout signal arrivant d'une instance de processus a quatre paramètres, dont seulement deux sont utilisés: la référence d'appel, *CR*, et la structure *ies* qui contient tous les éléments d'information. Le coordinateur construit, via la macro *send_out_mess*, le message *out_message* qui sera envoyé au codage en tant que paramètre du signal *mess_send*. Un message RELEASE_COMPLETE entraîne aussi la libèration de la référence d'appel associée à la connexion qu'il achève (faite par la fonction C *release_cr*).

Réception des primitives

Les primitives reçues de la couche supérieure sont dirigées vers l'instance de processus dont l'identificateur *Proc_id* est trouvé par la fonction *find_proc*. La primitive de demande de connexion *Setup_req* est traitée seulement si le nombre maximum de connexions accéptées n'est atteint. Dans ce cas, le coordinateur génère une valeur unique de référence d'appel qu'il annonce à la couche supérieure par une primitive *Send_cr*, crée une instance de processus, stocke la correspondance entre la référence d'appel et l'identificateur de la nouvelle instance et envoie la primitive *Connection_req* vers l'instance de processus dernièrement créée. Les valeurs de la référence d'appel sont générées cycliquement de 1 à *dummy_cr-1* (i.e. 2^{24}-2) en utilisant la fonction *modulo*.

Les primitives reçues d'une instance de processus sont dirigées vers la couche supérieure sans aucun traitement. La primitive *Release_conf* qui confirme la cessation d'une connexion provoque la libération de la référence d'appel associée à celle-ci.

Macros

La macro *send_out_mess* construit le message de sortie *out_message* et l'envoie au codage. La construction du message se fait par:

• l'affectation des valeurs de l'en-tête du message: discriminateur du protocole, référence d'appel et type de message, ce dernier reçu en tant que paramètre de la macro;

• l'affectation de la valeur *ies* (reçue de l'instance de processus en tant que paramètre du signal) à la structure représentant les éléments d'information du message sortant, *out_message!ie*.

La macro *build_err_mess* construit le message qui sera envoyé en cas d'erreur. Elle comprend les actions suivantes:

• affectation des valeurs de l'en-tête du message: discriminateur du protocole, référence d'appel et type de message, ce dernier reçu en tant que paramètre;

• construction de la partie contenant les éléments d'information:

- appel de la macro *init_ies* pour obtenir une structure vide et
- affectation de la valeur de la *Cause* selon le type d'erreur (passé en tant que paramètre *cv*).

4.3.5. Description du processus Proc

Une instance de processus modélise une connexion. Elle est créée par le processus *Coord* pour toute nouvelle connexion acceptée et cesse d'exister une fois la connexion libérée.

Une instance de processus n'est intéressée que par les éléments d'information d'un message. Ceux-ci lui parviennent du coordinateur, dans une structure, *ie*, en tant que paramètre de signal. La structure *ie* est assujettie à une vérification qui porte sur:

• la présence de tous les éléments d'information obligatoires;

• la validité du contenu de chaque élément d'information obligatoire;

• la validité du contenu de chaque élément d'information optionnel présent;

• l'absence de tout autre élément d'information.

Dans le cas des messages SETUP, CALL_PROCEEDING, CONNECT et STATUS la vérification est faite par les procédures *setup_verif*, *call_pr_verif*, *connect_verif*, *status_verif*, respectivement. Une procédure de vérification de message retourne:

- un code d'erreur, *code_error*, qui prend une des valeurs:
  - MIEM (*mandatory information element missing*) - un ou plusieurs éléments d'information obligatoires sont absents;
  - MIECE (*mandatory information element content error*) - un ou plusieurs éléments d'informations obligatoires contiennent des erreurs;
  - UIE (*unrecognized information element*) - un ou plusieurs éléments d'informations ne sont pas reconnus par le protocole ou sont reconnus, mais ils ne devraient pas être présents dans le message;
  - NMIECE (*non mandatory information element content error*) - un ou plusieurs éléments d'informations optionnels contiennent des erreurs;
  - OK - tout est correct.
- un diagnostic, *diag*, de type *occ_array_type*, qui contient le nombre et les identificateurs des éléments d'information fautifs;
- une structure (*s*, *cp*, *c* ou *st*, selon le cas) dont les champs sont les éléments d'information définis dans le type de message reçu.

À la sortie de la procédure, en cas d'erreur, le diagnostic *diag* sera inclus dans l'élément d'information *Cause* du message annonçant l'erreur. La structure retournée par la procédure représente un paramètre de primitive.

Puisque les autres messages, i.e. CONNECT_ACK, STATUS_ENQUIRY, RELEASE et RELEASE_COMPLETE, admettent au plus un élément d'information, leur vérification sera faite par le processus (et non par une procédure propre).

Selon le type d'erreur trouvée sur les éléments d'information d'un message, des mesures sont prises.

La réception d'un signal représentant une primitive provoque l'envoi du message correspondant.

L'envoi de tout message implique l'appel préalable de la macro *init_ies* pour la construction d'une structure vide, *ies* de type *ie_type*, dont les champs seront en suite

remplis en fonction du message à envoyer. Par exemple, la détection d'une erreur dans le contenu d'un élément d'information optionnel provoque l'envoi d'un message STATUS. Une structure *ies* vide de type *ie_type* est créée par la macro *init_ies*. Ensuite, on affecte des valeurs aux seuls éléments d'information permis dans le message STATUS i.e.:

• *Cause* dont la première occurrence contient le code d'erreur détecté et le diagnostic

et

• *Call_state* qui contient le code de l'état courant du processus.

Procédures

Il existe une procédure pour la vérification du contenu de chaque élément d'information. Le nom d'une telle procédure est composé du nom de l'élément d'information à vérifier suivi du mot *_verif*. Une procédure examine l'appartenance des valeurs de chaque champ à un domaine de valeurs donné dans la spécification et s'assure que la combinaison des sous-éléments présents est correcte. Le paramètre d'entrée est l'élément d'information et le paramètre de sortie est une variable booléenne, *error_free*, qui indique l'existence éventuelle d'une erreur.

La procédure *check_absence* détecte l'absence dans la structure *ie* des éléments d'information indiqués dans le tableau *a* de type *ie_array_type*. La procédure parcourt le tableau *a* et si la valeur d'un élément est *true*, elle vérifie que l'élément d'information correspondant à son index est présent dans la structure *ie*. Le nombre des éléments d'information absents et leurs identificateurs sont retournés dans le paramètre de sortie *diag*.

Selon le même principe, la procédure *check_presence* détecte la présence dans la structure *ie* des éléments d'information indiqués dans le tableau *a* de type *ie_array_type*. Les identificateurs des éléments d'information présents, ainsi que leur nombre, sont retournés dans le paramètre de sortie *diag*.

La procédure *verif* s'assure de l'absence de tout élément d'information. Elle attribue la valeur *true* à tous les éléments du tableau *arr* et appelle la procédure *check_absence* pour vérifier leur absence. Le paramètre de sortie *diag* retourne les identificateurs des éléments d'information présents et leur nombre.

Les procédures *setup_verif, call_pr_verif, connect_verif, status_verif* vérifient les messages SETUP, CALL_PROCEEDING, CONNECT et STATUS respectivement. Elles prennent comme paramètres d'entrée:

• la structure à vérifier *ie*;

• l'ensemble *ic_ie* d'identificateurs des éléments d'information dont le contenu a été trouvé invalide lors du décodage;

• la structure *u_ie,* qui contient le tableau des identificateurs des éléments d'information non-reconnus et leur nombre.

La procédure *connect_verif* possède un paramètre d'entrée supplémentaire, *ci*, pour indiquer le type de présence (obligatoire ou optionnelle) de l'élément d'information *Connexion identifier* dans le message CONNECT. Le type de présence dépend de l'état dans lequel ce message est reçu.

Les paramètres de sortie sont:

• le code d'erreur, *error_code*;

• la structure *diag*, contenant les identificateurs des éléments d'information fautifs et leur nombre;

• une structure, *s*, *cp*, *c*, ou *st*, selon la procédure, dont les champs sont les éléments d'information définis dans le message; elle n'est utilisée que si le message n'a pas d'erreurs ou si l'erreur porte sur les éléments d'information optionnels. Dans ce cas-là, seules les valeurs des éléments d'information corrects sont assignées aux champs de la structure.

Une procédure de vérification de message peut être divisée en cinq étapes et le passage d'une étape à la suivante se fait seulement si des erreurs ne sont pas détectées. Ces étapes sont:

• Vérification de la présence de tous les éléments d'information obligatoires. Dans le cas du message SETUP, elle est faite par la procédure *setup_man*, dans les autres cas directement. À l'intérieur de la procédure *setup_man*, les éléments d'information qui doivent être présents dans le message SETUP sont indiqués dans le tableau *arr* et une éventuelle absence est détectée par la procédure *ckeck_absence*. Si un ou plusieurs sont absents, la procédure de vérification de message retourne le code d'erreur MIEM.

• Vérification du contenu des éléments d'information obligatoires. Si au moins un élément d'information obligatoire appartient à l'ensemble *ic_ie*, ce qui signifie qu'une erreur de contenu à été détectée lors de son décodage, ou si une des procédures qui vérifient le contenu des éléments d'information obligatoires du message retourne la valeur *false* pour la variable *error_free*, la procédure de vérification de message retourne l'erreur MIECE dans le paramètre *error_code.*

• Traitement des éléments d'information qui ne sont pas acceptés par le protocole. Les identificateurs non-reconnus et leur nombre sont détectés lors du décodage et transmis dans le paramètre *u_ie*. Si un message contient des éléments d'information non-reconnus, *u_ie* deviendra le diagnostic *diag*, tandis que le code d'erreur sera UIE.

• Vérification des éléments d'information optionnels. Si au moins un élément d'information optionnel appartient à l'ensemble *ic_ie*, ce qui signifie qu'une erreur de contenu à été détectée lors de son décodage, ou si une des procédures qui vérifient le contenu des éléments d'information optionnels du message retourne la valeur *false* pour la variable *error_free*, la procédure de vérification de message retourne le code d'erreur NMIECE.

• Vérification de l'absence de tout autre élément d'information. Les identificateurs des éléments d'information qui ne sont pas admis dans le message sont spécifiés dans le tableau *uie_array*. La procédure *check_presence* détecte une éventuelle présence. Si c'est le cas, la procédure de vérification de message retourne le code d'erreur UIE. Sinon, le paramètre de sortie *error_code* aura la valeur OK.

La procédure *check_comp* vérifie la compatibilité de deux états. Elle prend comme paramètres d'entrée les codes des états, *cs* et *rs*, et retourne le verdict dans le paramètre de sortie *incomp*, de type booléen.

La macro *reset_all_timers* désactive tous les timers actifs au moment de l'appel.

4.3.6. Spécification des choix d'implantation

Les choix d'implantation définissent les aspects de la spécification qui:

• sont laissés à la latitude du développer, par exemple les états considérés compatibles;

• ne sont pas spécifiés du tout, par exemple la façon d'attribuer, gérer ou libérer une référence d'appel;

• dépendent des conditions réelles (ressources), par exemple le nombre maximum de connexions supportées.

La spécification de ces options n'est pas nécessaire dans la phase de vérification par simulation. Leur effet peut être reproduit de différentes façons, par exemple, en utilisant des variables booléennes dans les décisions qui les impliquent. La spécification des options devient essentielle dans la réalisation de la phase suivante, l'implantation.

4.3.6.1 Compatibilité des états

Pour demander l'état de l'entité paire, une entité envoie un message STATUS_ENQUIRY (par exemple suite à une interruption au niveau de la couche liaison de données). Sur la réception de la réponse, c'est à dire du message STATUS, la comparaison des états des deux entités doit être faite. À l'exception des situations impliquant les états U0, U11 et U12 qui sont définies dans la spécification, il revient au développeur de décider quels sont les états compatibles. Nous avons décidé la compatibilité des états suivants: U1 et U6, U1 et U9, U3 et U6, U3 et U9, U10 et U10. Leur compatibilité est vérifiée par la procédure *check_comp*.

L'action qui doit être initiée dans le cas des états incompatibles est aussi un choix d'implantation entre la libération de la connexion et toute autre action pour la récupération de l'erreur. Nous avons choisi la première alternative.

4.3.6.2 Envoi d'un message signalant une erreur non-fatale

La détection d'une erreur qui n'implique pas les éléments d'information obligatoires d'un message, représente une erreur non-fatale, c'est à dire, le comportement du protocole n'est pas perturbé dans le sens où il ignore les éléments d'information erronés. Toutefois, l'entité réceptrice d'un message contenant une erreur non-fatale, peut annoncer à l'entité paire le type d'erreur (NMIECE ou UIE) et les éléments fautifs par un message STATUS. Dans la présente implantation nous avons choisi d'envoyer un tel message.

4.3.6.3 Gestion des appels

Le protocole gère plusieurs connexions à la fois, mais la façon dont une entité de protocole génère, stocke ou libère les identificateurs des connexions est laissée comme choix d'implantation.

Génération d'une référence d'appel

Une entité qui demande une connexion doit lui affecter une référence d'appel. La valeur de la référence d'appel doit être comprise entre 1 et *dummy_cr-1* et doit être unique à l'interface locale. Notre spécification génère une nouvelle valeur de référence d'appel en ajoutant 1 à la dernière valeur générée, *last_cr*. La fonction *mod dummy_cr* garantit que la nouvelle valeur reste dans les limites imposées. On s'assure de l'unicité de la valeur en utilisant la fonction *find_proc*. Si la fonction retourne la valeur *true* pour la variable *active_call*, ce qui signifie que la valeur est déjà utilisée, une autre valeur sera générée.

Mode d'organisation des identificateurs de connexion

Puisque la gestion des connexions réside principalement sur la recherche de l'identificateur d'une instance de processus selon la référence d'appel, nous avons choisi d'organiser ces paires dans une structure d'arbre binaire, ce qui permet une recherche rapide. Un élément de l'arbre binaire est une structure dont les champs sont la référence d'appel d'une connexion et l'identificateur de l'instance de processus qui la modélise. La valeur 1 du drapeau d'une référence d'appel signifie que la connexion a été initiée de l'autre côté. La clé de recherche est représentée par la référence d'appel. Pour des valeurs égales la distinction est faite par le drapeau. SDL ne permet pas la déclaration des pointeurs nécessaires aux opérations appliquées aux arbres et, par conséquent, les fonctions de gestion ont été écrites en langage C et elles ont été intégrées, lors de l'implantation, au code C généré par l'outil. Il s'agit des fonctions suivantes:

- *find_proc* - la recherche dans un arbre binaire d'un élément dont on connaît la référence d'appel;
- *put* - l'ajout d'un élément dans un arbre binaire;
- *release_cr* - enlève de l'arbre l'élément dont la référence d'appel est passée comme paramètre.

4.3.6.4 Ressources

D'autres contraintes de l'implantation sont reliées aux ressources limitées de l'environnement d'implantation:

- il y a un nombre maximum de connexions supportées (techniquement ou administrativement);
- les indicateurs des VPI/VCI ne peuvent pas être utilisés par plusieurs connexions en même temps;
- l'implantation supporte seulement certaines classes de qualité de service;
- la bande passante est limitée; des restrictions pourraient être globales ou spécifiques pour certains classes de service ou dépendant d'autres critères.

Les caractéristiques des ressources qui ne varie pas en temps peuvent être implantées comme constantes. Les décisions reviennent à comparer tout simplement la valeur des variables de contrôle ou des paramètres avec ces constantes. Par exemple, si on fixe le nombre maximum de connexions supportées par le protocole à 1024, la décision d'acceptation d'une nouvelle connexion revient à tester si cette limite est déjà atteinte.

Les caractéristiques dynamiques des ressources, par exemple la largeur de la bande passante disponible, peuvent être connues suite à des vérifications locales.

Dans notre spécification la présence de ces contraintes est indiquée par des décisions qui contiennent un texte informel.

4.4 Simulation

4.4.1. L'outil de simulation

La validation du protocole de signalisation a été réalisée en utilisant le simulateur de l'outil ObjectGÉODE.

Le simulateur permet l'exécution, de façon interactive ou automatique, d'un modèle SDL. L'utilisateur peut, à tout moment, consulter ou modifier l'état du modèle via des commandes. Une commande qui change l'état du modèle (affectation d'une variable,

exécution d'une transition etc.) représente un **pas** de simulation. Une séquence de pas constitue un **scénario**. Les informations affichées lors de l'exécution de la simulation forment la **trace** de simulation. Elle peut être configurée par l'utilisateur selon ses besoins.

Le simulateur met à la disposition de l'utilisateur les fonctions suivantes:

- traces conditionnelles;
- conditions d'arrêt;
- guidage automatique via des filtres;
- exécution arrière;
- mémorisation du scénario courant et réexécution;

Pour chaque processus et procédure, le simulateur produit des tableaux de couverture qui répertorient le nombre d'exécutions de chaque transition et le nombre de passages dans chaque état. Il en déduit le taux de couverture des transitions et des états d'un processus ou d'une procédure.

Le protocole de signalisation est un système ouvert, i.e. il communique avec l'environnement par des canaux qu'on appelle externes. L'environnement envoie et reçoit des signaux.

Les signaux envoyés par l'environnement peuvent être simulés de deux façons;

- par la commande *output*, qui est utilisée pour l'envoi d'un signal; le signal envoyé par cette commande est mis dans la file du processus visé;
- par la commande *feed*, qui est utilisée pour l'injection de signal; chaque fois qu'un processus est dans un état dans lequel il peut recevoir un signal de l'environnement, la transition qui correspond au signal injecté apparaît dans la liste des transitions tirables (*fireable*). L'exécution de cette transition consomme le signal sans qu'il soit passé par la queue. L'injection de signal est utilisée surtout dans la simulation automatique pour la modélisation du comportement de l'environnement qui peut envoyer, à tout moment, un signal. L'ensemble de signaux pouvant être envoyés est défini par une suite de commande *feed*.

Le simulateur permet trois types de simulation: guidée (ou encore interactive ou pas à pas), aléatoire et exhaustive.

4.4.1.1 Simulation guidée

La simulation guidée permet à l'utilisateur de choisir une des transitions tirables à un moment donné (qui sont affichées dans une zone spécialement conçue de la fenêtre du simulateur). L'utilisateur a ainsi la possibilité de mener l'analyse à l'endroit de son choix.

L'exécution d'une transition détermine les opérations suivantes:

- la mémorisation de l'état initial;
- l'exécution de la transition sélectionnée;
- la mise-à-jour des tables de couverture;
- la mise-à-jour du temps SDL et du numéro de pas;
- l'affichage des événements observables et les valeurs de leurs paramètres;
- l'éventuel filtrage des transitions (dicté par l'utilisateur);
- l'affichage des traces;
- l'affichage des conditions d'arrêt;
- l'affichage des prochaines transitions possibles.

À tout moment, l'utilisateur peut consulter ou modifier l'état du modèle.

Le simulateur garde toujours la liste des pas de simulation qui ont mené de l'état initial à l'état courant, i.e. le scénario courant. Cela permet à l'utilisateur de naviguer au long du scénario: retourner à un état déjà accédé et refaire le même chemin (pas à pas ou plusieurs pas à la fois) ou un autre. Un scénario peut être sauvegardé pour une éventuelle réutilisation.

4.4.1.2 Simulation aléatoire

Dans la simulation aléatoire, le simulateur exécute une séquence de transitions de façon aléatoire. La simulation aléatoire s'arrête:

- à la demande de l'utilisateur;
- quand le système atteint un blocage;
- lorsqu'une condition d'arrêt (antérieurement définie par l'utilisateur) est vraie;
- dans le cas d'une erreur.

Les pas suivis sont enregistrés dans le scénario courant et les tables de couverture sont mises-à-jour. La sélection de la transition dépend du générateur de nombres aléatoires, dont

le germe (*seed*) peut être contrôlé par l'utilisateur. Le même germe exécute le même scénario. De cette façon un scénario produit aléatoirement peut être refait.

4.4.1.3 Simulation exhaustive

Dans la simulation exhaustive, le simulateur construit un graphe d'états pour explorer toutes les possibilités (les états globaux du système) en fonction des entrées.

Contrairement à la simulation aléatoire, le simulateur exécute, à tour de rôle, toutes les transitions possibles dans un état donné. On obtient deux sortes de résultats:

- statistiques - affichés dans la fenêtre du simulateur;
- des scénarios - sauvegardés dans des fichiers dans le répertoire courant - qui conduisent à des blocages, à une condition d'arrêt, à des erreurs, à des feuilles des composantes fortement connexes (dans le mode d'exploration en profondeur); ces scénarios peuvent être par la suite exécutés et analysés séparément.

Il est possible de choisir:

- le nombre maximum des scénarios qui seront construits pour chaque catégorie mentionnée;
- la profondeur et/ou la largeur de l'arbre d'exploration;
- le nombre maximum d'états à explorer;
- le mode d'exploration;
- les conditions pour tronquer l'exploration;
- des optimisations (de stockage des états globaux et des états des processus).

L'affichage des tableaux de couverture - des états et des transitions - donne plus d'informations.

Généralement, l'explosion combinatoire des états globaux résultant de la simulation exhaustive empêche de les traiter en totalité.

4.4.2. Simulation du protocole de signalisation

Dans une première étape nous avons validé le protocole dans le cas d'une seule connexion et en ignorant les paramètres des signaux. Dans ces conditions, le modèle SDL

comprend un seul processus, *Proc*, mono-instancié, et les procédures de vérification des paramètres sont vides. Les décisions impliquant le résultat des vérifications contiennent le texte informel '*error_code ?*'. Pour chaque décision informelle, le simulateur construit un nouvel état et y arrête la simulation pour permettre à l'utilisateur de choisir la branche de réponse à suivre. Il est possible d'automatiser le choix d'une réponse en utilisant des filtres. Par exemple, si l'on veut tester le comportement du modèle supposant que tous les messages sont correctement reçus, autrement dit que les procédures de vérification des paramètres retournent toujours le code OK, on peut utiliser les filtres:

filter(decision_error_code('MIEM'))

filter(decision_error_code('MIECE'))

filter(decision_error_code('UIE'))

filter(decision_error_code('NMIECE'))

Cela enlève de la liste des transitions possibles celles correspondant à ces réponses. Il reste seulement la transition correspondant à la réponse OK.

Dans une deuxième étape, nous avons ajouté les paramètres des signaux et les procédures pour leur vérification. Cela rend la simulation beaucoup plus complexe, car d'une part les procédures de vérification contenant essentiellement des décisions, le nombre de possibilités de recherche augmente considérablement et, d'autre part, l'affectation des champs des structures représentant les messages et les primitives, qui doit se faire manuellement, est un processus laborieux. En plus, la variation des paramètres détermine des nouveaux cas d'analyse.

La dernière étape de simulation prend en considération la capacité du protocole de supporter plusieurs connexions. Alors nous avons inclus le coordinateur dans la spécification SDL et nous avons concentré l'analyse sur celui-ci. Le coordinateur contient trois fonctions dont le contenu ne sera inséré que lors de l'implantation:

• *put* qui sauvegarde, pour chaque nouvelle connexion, la correspondance entre la référence d'appel et l'identificateur d'instance de processus;

• *find_proc* qui détermine si la référence d'appel présente dans le signal entrant est active ou non, et, dans le cas affirmatif, retourne l'identificateur de l'instance;

• *release_cr* qui annule la correspondance entre la référence d'appel et l'identificateur d'instance de processus.

Par conséquent, il a fallu simuler leur effet, plus précisement le routage des signaux, par des commandes. Les décisions qui testent si la référence d'appel est active ou non contiennent le texte informel, *'active_call ?'*, ce qui provoque l'arrêt de la simulation pour que l'utilisateur indique la branche de réponse à suivre. Dans le cas où l'on veut simuler une référence d'appel active, avant de transmettre le choix de réponse, on doit indiquer l'identificateur de l'instance de processus vers laquelle la sortie sera acheminée, i.e. la valeur de la variable *Proc_id* de type *Pid* utilisée dans le OUTPUT. Il est possible de le faire sachant que chaque instance de processus est désignée par un entier qui représente le numéro de création. Par exemple, la commande *let coord!Proc_id = proc(3)!self* affecte à la variable *Pid* du processus *Coord* l'identificateur de la troisième instance du processus *proc* créée.

5. Construction du code exécutable du protocole de signalisation

La spécification en SDL du protocole de signalisation a été réalisée à l'aide de l'outil ObjectGEODE. Outre l'édition de la spécification formelle en forme graphique, l'outil fournit un environnement pour la vérification et la validation de la spécification (qui peuvent être faites à différents niveaux de raffinement) et permet également, l'automatisation de la génération de code exécutable. Le code C généré par l'outil est indépendant de l'environnement où l'application va être implantée. L'environnement est intégré dans le système à l'aide des librairies d'exécution (*run-time libraries*). Il existe des librairies d'exécution pour différents systèmes d'exploitation, dont UNIX. Toutes ces bibliothèques sont fournies en code source et peuvent être adaptées à des besoins ou à des contraintes spécifiques. L'utilisateur peut également ajouter son propre code C.

Le présent chapitre décrit les étapes suivies pour l'obtention du code exécutable du protocole de signalisation à partir de la spécification SDL. Il inclut une section sur le code C que nous avons écrit séparément et qui a été intégré au code C généré par l'outil.

5.1 Configuration d'une application

L'architecture physique d'une application est formée par les objets suivants:

- noeud: unité qui correspond à un processeur dans un environnement multiprocesseur;
- tâche (*task*): unité de parallélisme du système d'exploitation.

L'architecure SDL est projetée (*mapped*) dans l'architecture physique dans le but d'optimiser le code généré. Un changement dans la distribution des fonctions de l'application requière une modification de la projection (*mapping*) et non de l'architecture SDL.

Une tâche peut être associée (*mapped*) à un des objets SDL suivants:

- un processus (*Task/Process mapping* ou TP);
- une instance de processus (*Task/Instance mapping* ou TI);

• un bloc (*Task/Block mapping* ou TB).

Les communications entre les noeuds sont gérées via des sockets TCP/IP.

Les tâches d'un noeud sont gérées par le système d'exploitation tandis que l'intérieur des tâches est géré par ObjectGEODE. La communication entre les tâches d'un noeud est réalisée par un mécanisme de boîte aux lettres (*mailbox*). Ce mécanisme est implanté en utilisant les ressources de l'exécutif (*Real-Time Executive*). À chaque tâche est associé une boîte aux lettres. Puisque chaque instance de processus reçoit les signaux dans une queue (port d'entrée SDL) la correspondance (*mapping*) entre le port d'entrée SDL et la boîte aux lettres dépend de la stratégie sélectionnée (TI, TP ou TB) et des mécanismes offerts par l'exécutif. Une boîte aux lettres représente tous les ports d'entrée des instances d'une tâche. Une tâche reçoit tous les signaux qui lui sont transmis via la boîte aux lettres. Elle exécute l'instance de processus associée à chaque signal.

La correspondance entre les objects d'exécution (noeud, tâche) et les objects SDL (système, bloc, processus, instance de processus) est décrit dans un fichier de configuration (de type **.cfg**). On peut, par exemple, spécifier qu'un système SDL s'exécute sur un certain noeud d'un système distribué et qu'un bloc représente une tâche de ce noeud. Toujours par l'intermédiaire du fichier de configuration, il est possible d'intégrer une tâche externe (écrite en code C) représentant un bloc, un processus ou l'environnement. Le fichier de configuration est un fichier ASCII avec une syntaxe prédéfinie.

5.2 Préparation des entrées de la génération

Les entrées nécessaires à la génération de code C sont:

• le(s) fichier(s) contenant la forme textuelle de la description SDL. Notre description est contenue dans un seul fichier, **atm_sign.pr**;

• optionnellement, le fichier de configuration **.cfg**. S'il n'est pas fourni, un fichier de type .**cfg** vide sera généré. Puisque nous avons décidé d'exécuter notre implantation et puisque notre système est un système ouvert, il a fallu modéliser le comportement de l'environnement par un programme C, appelé tâche externe. La rélation entre la spécification SDL et cette tâche externe est declarée dans le fichier de configuration **atm_sign.cfg**.

5.3 Génération du code source

5.3.1 Préparation de l'environnement

Pendant l'étape de génération du code source sont construits les fichiers source, d'inclusion et makefiles. L'exécutif sous lequel l'application sera exécutée - dans notre cas **Unix** - est passé en tant que paramètre de la commande de génération **geodeb_rte**.

L'environnement de génération d'une application est configuré via des variables d'environnement. Les valeurs par défault de ces variables se trouvent dans le fichier **geodeb_unix_env** et peuvent être modifiées. Il est possible de spécifier:

• l'emplacement des fichiers de l'application. Par exemple, on peut indiquer le répertoire contenant les fichiers d'entrée (la description SDL, fichier de configuration) via la variable d'environnement GEODE_S_DES ou des répertoires différents pour les fichiers source (code C) et les fichiers d'inclusion générés via les variables GEOCRO_C_SRC et GEOCRO_G_INC respectivement.

• certains paramètres de la génération:

- la stratégie de définition. Par exemple, on peut indiquer le type de correspondance (*mapping*) en affectant à la variable d'environnement GEODEMAPPING une des valeurs TI pour *Task/Instance*, TP pour *Task/Process*, TB pour *Task/Block*.
- gestion de fichier: la longueur des noms des fichiers générés, la façon de générer les fonctions C (en utilisant le formalisme ANSI ou K & R), l'endroit de génération des signaux ou des procédures (dans un fichier à part ou non) etc.
- implantation SDL: le préfix des champs d'une structure C correspondant à une structure SDL, la façon de traduire les sorties (OUTPUT) SDL (par des macros ou, sélectivement, par appel de fonction).

Puisque le nombre de connexions supportées par le protocole de signalisation peut dépasser le nombre maximum de processus (tâches) admis par un noeud, la correspondance TI (tâche/instance de processus) n'est pas indiqué. Nous avons alors choisi la correspondance TP (tâche/processus).

Nous avons également augmenté, dans le fichier **geode_unix_env**, la valeur de deux variables d'environnement comme suit:

GEODE_LINE_SIZE=160 qui indique la longueur maximale d'une ligne dans les fichiers générés;

GEODE_NAME_LIMIT=15 qui indique la longueur maximale des noms des fichiers générés.

5.3.2. Fichiers générés

La génération du code C d'une application comporte la génération de quatre catégories de fichiers:

• contenant du code C qui dépend de la configuration: **c_values.h** et **c_declar.h**; ces fichiers décrivent l'application et sont utilisés par le fichier **g2_node.c** (de l'outil) lors de l'initialisation du noeud (création des ressources comme les boîtes aux lettres et les tâches, initialisation des instances initiales etc.).

• contenant du code C dépendant de la description SDL: un fichier d'inclusion **.h** et, éventuellement, un autre de type **.c** pour chaque objet structurel (système, bloc, processus).

Les fichiers générés pour notre application sont:

b_q2931.h
i_atm_un.c
p_coder_decoder.c
p_coder_decoder.h
p_coord.c
p_coord.h
p_proc.c
p_proc.h
r_coder_decoder.h
r_coord.h
r_proc.h
s_atm_un.h

• makefile: **atm_sign.mk**;

• d'information:

- un fichier de type **.log** contenant les informations, les avertissements et les erreurs détectés au cours de la génération, ainsi que la liste des types de données abstraites et leurs nombres d'opérateurs;
- un fichier **.ref** contenant les noms des fichiers de type **.pr** qui forment la description SDL;
- un fichier de type **.lt** contenant la liste de tous les fichiers modifiés depuis la dernière génération et deux autres **.lc** et **.lh** contenant les noms des fichiers **.c** et **.h** modifiés depuis la dernière génération;
- un fichier **.lp** contenant les paramètres utilisés dans la génération du code;

5.4 Compilation et édition de liens

Pendant l'étape de compilation et d'édition de liens a lieu:

• la configuration de l'environnement de travail;

• l'intégration des fichiers C écrits par l'utilisateur (non générés par l'outil);

• la compilation et l'édition de liens de l'application.

5.4.1. Préparation de l'environnement

L'environnement de développement d'une application est configuré via des variables d'environnement. Les valeurs par défault de ces variables se trouvent dans le fichier **gce_unix** et peuvent être modifiées. Il est possible de préciser:

• l'emplacement des fichiers de l'application. Les fichiers qui forment une application sont localisés dans différents répertoires. À l'aide des variables d'environnement on peut spécifier les répertoires contenant: les librairies de l'outil, les fichiers fournis par l'utilisateur, les fichiers générés etc.

• la configuration de la compilation: les suffixes des fichiers compilés et exécutables, des paramètres de la compilation et de l'édition des liens (*linking*), différents utilitaires (d'affichage ou de mise à jour);
• la configuration spécifique: le nom du compilateur, son type (ANSI ou non), les librairies utilisées etc.

5.4.2. Code C externe

Une application générée peut être complétée ou adaptée à des besoins spécifiques par du code C écrit par le développeur. Ces modifications visent différents aspects:
• opérateurs de type de données abstraites;
• prédéfinition des types, des constantes ou primitives C;
• redéfinition de primitives SDL;
• tâches externes implantant l'environnement SDL;
• tâches externes implantant des processus ou des blocs SDL.

Le code C externe peut être intégré dans une application générée à l'aide d'un fichier makefile utilisateur, de type **.umk**. Ce fichier sera automatiquement concaténé avec le fichier makefile généré par l'outil, **atm_sign.mk**.

5.4.3. Étapes

La commande **gc_unix** réalise la compilation et l'édition de liens comme suit:
• invoque le fichier **gce_unix** qui configure l'environnement de travail;
• crée le fichier makefile **atm_sign.tmk** par la concaténation du fichier makefile généré par l'application, **.mk**, avec celui construit par l'utilisateur, **.umk** (s'il existe);
• exécute le fichier **atm_sign.tmk** via l'utilitaire **make**;
• libère les ressources (effacement des fichiers temporaires).

5.5 Stratégies de génération

Nous décrivons dans cette section les stratégies de génération de code C réalisées par l'outil GEODE.

5.5.1. Types manipulés par une application

5.5.1.1 Synonyme

La déclaration d'un synonyme SDL est générée en code C comme la définition d'une macro. Le nom de la macro est composé par le préfix **syn_** suivi du nom du synonyme. Exemple: Les synonymes SDL déclarés au niveau de système sont générés dans le fichier C **n_atm_un.h** comme suit:

SDL:	**C:**
SYNONYM SETUP Natural = 5;	#define syn_setup 5
SYNONYM CALL_PROCEEDING Natural = 2;	#define syn_call_proceeding 2

5.5.1.2 Types de données de base

Les types SDL simples, i.e. *natural*, *integer*, *boolean*, *character*, *duration*, *time*, etc., reçoivent, dans le code généré, le préfix **SDL_** et sont traduits en C par des macros qui sont définies dans les fichiers d'inclusion. Néanmoins, l'utilisateur a la possibilité d'introduire sa propre équivalence entre les déclarations SDL et C.

Par exemple, le type SDL *natural* apparaît comme SDL_NATURAL dans le code C généré. Cette macro, située dans le fichier **g2_com.h**, est définie *unsigned int*.

5.5.1.3 Types de données utilisateur

Les types SDL définis par l'utilisateur reçoivent le préfix **GU_** et sont transformés, toujours par l'intermédiaire des macros, dans des types C appropriés.

Syntype. Un type SDL *syntype* devient en code C le même type que le type d'origine. Exemple:

SDL:

```
SYNTYPE flag_type = Natural
  CONSTANTS 0 : 1
ENDSYNTYPE;
```

C:

```
#define GU_FLAG_TYPE SDL_NATURAL
```

Structure. Une structure SDL est transformée dans une structure C. L'identificateur de la structure prend le préfix **GU_**. Les identificateurs des champs d'une structure C prennent le préfix **fd_**.

Exemple:

SDL:

```
NEWTYPE call_reference_type STRUCT
  flag flag_type;
  value int_23;
ENDNEWTYPE;
```

C:

```
typedef struct {
  GU_FLAG_TYPE fd_flag;
  GU_INT_23 fd_value;
} GU_CALL_REFERENECE_TYPE;
```

Tableau. La génération de code impose une restriction dans la définition d'un tableau SDL: que son index soit borné. Deux macros C sont définies pour la borne inférieure et supérieure et servent à calculer la dimension du tableau C généré.

Exemple:

SDL:

```
SYNTYPE max_occ_cause = Natural
  CONSTANTS 0: 2
ENDSYNTYPE;

NEWTYPE cause_table
  Array( max_occ_cause, cause_IE)
ENDNEWTYPE;
```

C:

```
#define inf_CAUSE_TABLE (0)
#define sup_CAUSE_TABLE (2)

typedef GU_CAUSE_IE GU_CAUSE_TABLE
[sup_CAUSE_TABLE - inf_CAUSE_TABLE +
1];
```

Mode ensembliste (POWERSET). Le type POWERSET est implanté comme un tableau (ARRAY) de booléens. La valeur booléenne d'un élément du tableau indique si son index appartient ou non à l'ensemble.

SDL:

```
SYNTYPE octet = Natural
  CONSTANTS 0 : 255
ENDSYNTYPE octet;

NEWTYPE octet_set
  POWERSET(octet)
ENDNEWTYPE;
```

C:

```
#define inf_OCTET_SET (0)
#define sup_OCTET_SET (255)

typedef SDL_BOOLEAN GU_OCTET_SET
[sup_OCTET_SET - inf_OCTET_SET + 1] ;
```

Autres types. Les types de données SDL définis par le mot clé NEWTYPE autres que de nature structure, tableau ou *powerset* apparaissent dans le code C comme étant de type SDL_NEWTYPE. Les éventuels littéraux d'un tel type sont numérotés à partir de 1.

SDL:

```
NEWTYPE error_code_type
     LITERALS OK, MIEM, MIECE, UIE,
NMIECE;
ENDNEWTYPE;
```

C:

```
typedef SDL_NEWTYPE
GU_ERROR_CODE_TYPE;

#define gu_ERROR_CODE_TYPE_ok 0
#define gu_ERROR_CODE_TYPE_miem 1
#define gu_ERROR_CODE_TYPE_miece 2
#define gu_ERROR_CODE_TYPE_uie 3
#define gu_ERROR_CODE_TYPE_nmiece 4
```

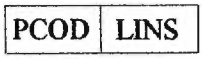
5.5.1.4 Types C supplémentaires

Un ensemble de types C nécessaires à la gestion interne est défini pour chaque type SDL. Le nom d'un tel type a le préfix **GX_** et le suffixe **_P** s'il s'agit d'un pointeur ou **_T** autrement.

5.5.1.5 PID

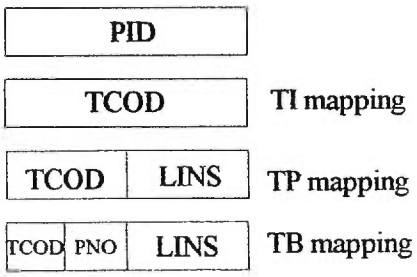
Le type SDL PID est généré comme l'appel de la macro SDL_PID, celle-ci définie par un autre appel de macro, GX_PID_T, dont la définition dépend du compilateur et de l'exécutif. Dans notre cas le PID est de type *unsigned long*.

La structure de PID regroupe le code du processus (PCOD) et le numéro de l'instance locale (LINS).



PID structure

Une autre façon de décrire le PID est d'utiliser le code de la tâche. Évidemment, dans ce cas la représentation du PID dépend du type demapping. Dans la correspondance TI, le Pid coïncide avec le code de la tâche. Pour les autres types de correspondance l'identification d'une instance de processus nécessite l'identification de la tâche (TCOD) et, éventuellement, du processus (PNO) à l'intérieur de la tâche.



5.5.1.6 Signaux

L'identificateur d'un signal est formé par le préfix **sig_** et le nom SDL du signal. Les signaux qui interviennent dans une application sont numérotés à partir du niveau du système, en suivant le parcours en profondeur de la structure hiérarchique SDL. Les deux premiers signaux sont toujours les signaux implicites START et EMPTYQ.

Exemple: La structure SDL du protocole de signalisation comprend le système, un bloc et deux processus. La numérotation des signaux commence par les deux signaux implicites,

suivis par les signaux déclarés au niveau système et ceux au niveau bloc. Le code C généré dans le fichier **s_atm_un.h** est:

```
#define sig_START    1
#define sig_EMPTYQ  2
#define sig_setup_req 3
#define sig_proceeding_req 4
...
#define sig_SAAL_ind 16
#define sig_SAAL_req 17
```

La numérotation continue au niveau de bloc, dans le fichier **b_q2931.h**:

```
#define sig_setup 18
#define sig_call_proceeding 19
...
#define sig_mess_req 28
#define mess_send 29
```

Chaque signal est généré comme une structure qui inclut en tant que champs, outre les éventuels paramètres, un en-tête qui dépend de l'exécutif et un autre qui dépend de la stratégie de génération. Les paramètres d'un signal paramétré sont numérotés. L'identificateur d'un paramètre est formé par:

- le préfix **sg**;
- son numéro;
- le caractère _;
- le type du paramètre.

Exemple: Le message SETUP est implanté:

```
typedef struct {
  GX_E_HD
  GX_G_HD
  GU_CALL_REFERENCE_TYPE sg1_CALL_REFERENCE_TYPE;
  GU_IE_TYPE sg2_IE_TYPE;
  GU_IE_VALUES_SET sg3_IE_VALUES_SET;
  GU_OCC_ARRAY_TYPE sg4_OCC_ARRAY_TYPE;
} SIGNAL_setup;
```

5.5.1.7 Timers (temporisateurs)

Les timers sont numérotés comme signaux après les signaux locaux de chaque processus. Deux macros par processus sont générées, GG_BASE_TIMER et GG_NB_TIMER, qui donnent respectivement, le numéro du premier timer et le nombre de timers.

Exemple: Les timers utilisés par le processus *Proc* sont générés dans le fichier **p_proc.h**:

```
#define sig_t303  30
#define sig_t308  31
...
```

Les deux macros générées de gestion sont:

```
#define GG_BASE_TIMER  30
#define GG_NB_TIMER    5
```

Puisque l'automate d'états traite un timer comme un signal, la structure générée pour un timer ressemble à celle d'un signal non-paramètré. Elle contient, outre les champs en-tête, un ensemble de champs de contrôle défini comme macro et dépendant du type du système d'exploitation.

Exemple:

```
typedef struct {
  GX_E_HD
  GX_G_HD
  GX_T_HD
  } GX_TIMER_T
```

5.5.2. Traduction des expressions

La façon de traduire une expression SDL en expression C dépend des types de données qu'elle véhicule.

5.5.2.1 Types simples

Booléen. Les valeurs FALSE et TRUE sont représentées par GX_FALSE (défini comme étant 0) et GX_TRUE respectivement.

Opération	SDL	Code C	Implantation
affectation:	TASK a:=FALSE;	a=GX_FALSE	a=0
comparaison:	a=b	G2M_COMP_BOOL (a,b)	a==b
	a=TRUE		(a)
	a=FALSE		!(a)
opérateurs:	not a	G2M_NOT(a)	!(a)
	a and b	G2M_AND(a,b)	a&&b
	a or b	G2M_OR(a,b)	a | | b
	...		

Autres: *integer, real, natural, character, duration, time*

L'affectation et la comparaison sont générées de façon directe:

Opération	SDL	C
affectation:	TASK a:=b;	a=b
comparaison:	a=b, a<b	a==b, a<b

5.5.2.2 Types complexes

Les opérations associées aux types complexes sont faites via des primitives.

Structure

Opération	SDL	Code C généré
Affectation	TASK a:=b;	G2M_COPY_REC (&a, &b, type) si b est une variable; GU_type_b (paramètres, &a) si b est un ADT opérateur; gg_sa_type (liste, &a) si b est une liste.
Comparaison	a=b	G2M_COMP_REC (&a, &b, type) si b est une variable; G2M_COMP_REC (&a, b, type) si b est un opérateur ou une liste.

Le résultat de la comparaison est une valeur booléenne.

Exemple:

L'affectation SDL

TASK out_message!cr:=cr;

qui attribue la variable *cr* de type GU_CALL_REFERENCE_TYPE au champ *out_message!cr* du même type est générée de la façon suivante:

G2M_COPY_REC(&(G2_VAR(out_message) .fd_cr),&(G2_VAR(cr)),GU_CALL_REFERENCE_TYPE);

(Les variables sont accédées via la primitive G2_VAR. Le champ *cr* de la variable *out_message* est désigné par le suffix *fd_cr.*)

Tableau

Opération	**SDL**	**Code C généré**
Affectation	TASK a:=b;	G2M_COPY_ARRAY (a, b, type) si b est une variable; GU_type_b (paramètres, a) si b est un ADT opérateur;
Comparaison	a=b	G2M_COMP_ARRAY (a, b, type).

Le résultat de la comparaison est une valeur booléenne.

Powerset

Le mode ensembliste (*powerset*) est géré comme un tableau d'éléments booléens, dont l'index correspond au type du powerset. Autrement dit, POWERSET(type) est équivalent au ARRAY(type, BOOLEAN).

Opération	**SDL**	**Code C généré**
Affectation	TASK a:=b;	G2M_COPY_SET (a, b, nb_items) si b est une variable; GU_type_b (paramètres, a) si b est un ADT opérateur;
Comparaison	a=b	G2M_COMP_SET (a, b, nb_items).

Opérateurs implicites:

Déscription	SDL	Code C généré
item ∈ pwset ?	item IN pwset	G2M_INSET(item, pwset, nb_items)
B={item}∪ A	B:=INCL(item,A)	G2M_INCLSET(item, A, nb_items, B)
...		

Le littéral SDL prédéfini *Empty* est représenté par la macro G2M_EMPSET.

Énumération

L'utilisation du mot clé ORDERING donne accès aux opérateurs prédéfinis:

SDL	Code C	Implantation
succ	G2M_SUCC(élément)	élément + 1
pred	G2M_PRED(élément)	élément - 1
ord	G2M_ORD(élément)	élément
min	G2M_MIN(e1, e2)	(e1<e2) ? e1 : e2
max	G2M_MAX(e1, e2)	(e1>e2) ? e1 : e2

5.5.3. Contexte de tâche

Le contexte d'une tâche est représenté par les variables qui doivent être visibles par tout le code correspondant à la tâche respective. Elles sont déclarées de type statique et peuvent être accédées par des macros et des primitives génériques.

5.5.4. Contrôle du flux

Une fonction est générée pour chaque processus et pour chaque procédure SDL. Elle gère:

- la récupération du signal à traiter;
- la commutation des contextes d'instance;
- l'automate d'états correspondant.

5.5.4.1 Récupération de signal et commutation de contexte

La récupération du premier signal de la queue d'une tâche est faite par la primitive G2S_READ_QUEUE(). Le contexte de l'instance à laquelle le signal a été envoyé devient le contexte courant par l'appel de la primitive G2I_SWITCH_CTX(). Le signal devient le signal courant et sera traité par l'instance de processus. Après le traitement du signal la zone de mémoire qu'il occupe sera libérée par la primitive G2S_RELEASE juste avant d'extraire le signal suivant de la file.

5.5.4.2 Transition

L'identificateur d'une transition est formé par le préfix **tra_** suivi du nom de l'état, '_' et du nom de l'entrée. Dans le cas où l'astérisque est employé, l'identificateur d'une transition est obtenu en concaténant:

- le préfix **tra_st_**;
- le nom du premier état exclu (s'il y en a);
- le nom de l'entrée.

Exemples:

SDL:

```
STATE U1;
  INPUT CALL_PROCEEDING(...);

 STATE * (U0,  U1, U11  , U12);
  INPUT RELEASE_COMPLETE(...);
```

C:

```
tra_U1_call_proceeding:

tra_st_U0_release_complete:
```

Chaque entrée SDL (INPUT) provoque une transition. Une transition est générée comme un appel de la primitive G2_TRANS. Celle-ci est implantée en C comme un saut à l'étiquette fournie comme premier paramètre de la primitive, où sont désignées les actions à prendre lors de la transition. À la fin de chaque transition la variable CURRENT_STATE est mise à jour par une des primitives SDL_NEXTSTATE(nouvel_état) ou SDL_NEXTSTATE_DASH, selon le cas. Le contrôle est passé à l'étiquette apparaissant

comme le deuxième paramètre de la primitive qui libère la zone de mémoire occupée par le signal avant de traiter une autre entrée.

Exemple: Dans l'état U0 l'entrée *Setup_req* provoque la transition qui sera à l'étiquette *tra_U0_setup_req*:

```
      SDL_INPUT(setup_req):
        G2_TRANS(tra_U0_setup_req,gc_release)
      SDL_INPUT(setup):
        G2_TRANS(tra_U0_setup,gc_release)
...
tra_U0_setup_req:
        ...
        SDL_NEXTSTATE(u1);
tra_U0_setup
        ...
```

5.5.4.3 Automate d'états

L'identificateur C d'un état est formé du préfix **sta_** suivi du nom SDL de l'état. Les états sont numérotés, pour chaque processus, à partir de 1. Le dernier état représente le pseudo-état START.

Exemple: Les états du processus *Proc*, sont générés dans le fichier **p_proc.h** comme suit:

```
#define sta_u0    1
#define sta_u1    2
#define sta_u3    3
#define sta_u6    4
#define sta_u8    5
#define sta_u9    6
#define sta_u10   7
#define sta_u11   8
#define sta_u12   9
#define sta_start  10
```

Le fonctionnement de l'automate d'états est généré comme deux constructions C "switch" imbriqués, une sur l'état courant, CURRENT_STATE, de l'instance de processus, l'autre sur le signal reçu, *cur_signo*.

Exemple:

```
switch((int)CURRENT_STATE) {
 SDL_STATE(start):
   switch(cur_signo) {
    SDL_START:
      G2_TRANS(tra_START,gc_release)
    default:
      break;
   }
 SDL_STATE(x):
   switch(cur_signo) {
    SDL_INPUT(mess_rec):
      G2_TRANS(tra_X_mess_rec,gc_release)
    SDL_INPUT(setup_req):
      G2_TRANS(tra_X_setup_req,gc_release)
    SDL_INPUT(proceeding_ind):
      G2_TRANS(tra_X_proceeding_ind,gc_release)
    ...
    default:
      break;
   }
   break;
/*
 * !!! No State !!!
 */
  default:
    goto gc_release;
 }
```

5.5.4.4 Décision

Les décisions d'un processus SDL sont numérotées (en système hexadécimal) à partir de 1. Dans le code C généré, le nom d'une décision est formé par le préfix SDL_DECISION_X auquel on concatène son numéro. La façon dont le corps d'une décision est générée dépend du type de la décision.

Une décision SDL booléenne est générée comme une construction C *if...else*.

Exemple:

SDL:

```
DECISION without_error;
(TRUE):
        ...
(FALSE):
        ...
ENDDECISION;
```

code C généré:

```
SDL_DECISION_X0001:
if (without_error) {
...
}
else {
...
}
```

Une décision dont l'expression est une variable SDL qui doit être comparées à plusieurs réponses implique la numérotation des branches de réponse. L'identificateur d'une branche représente l'étiquette sous laquelle sont regroupées les instructions de la branche. Une construction *if* est générée pour chaque chaque variante de réponse. Elle compare la variable avec la réponse de la branche envoie le contrôle à l'étiquette appropriée.

Exemple:

SDL:

```
DECISION error_code;
(MIEM):
        ...
(MIECE):
        ...
ELSE:
        ...
ENDDECISION;
```

code C généré:

```
SDL_DECISION_X0002:
if (error_code==MIEM) goto SDL_ANSWER_X0002_1;
if (error_code==MIECE) goto SDL_ANSWER_X0002_2;
...          /* pour ELSE */
SDL_ANSWER_X0002_1:
...
SDL_ANSWER_X0002_2:
...
ENDDECISION_X0002:;
```

5.5.4.5 Sortie

Le code C d'une sortie SDL (OUTPUT) est une primitive dont le nom est obtenu en ajoutant le préfix **SDL_OUTPUT_** au nom SDL du signal sortant. Les paramètres de la primitive sont l'identificateur de l'instance de processus (Pid) vers laquelle le signal est envoyé et les paramètres du signal SDL. Si le Pid n'est pas spécifié explicitement par le mot clé TO, il est trouvé par la primitive G2P_FINDPID(). La définition de la primitive générée pour la sortie se trouve dans le fichier d'inclusion **(.h)** de l'objet structural où le signal entrant est défini. Son implantation comprend les actions suivantes:

• l'allocation d'un tampon pour le signal sortant via la primitive G2S_CREATE;

• affectation des paramètres du signal via des primitives appropriées (qui dépendent du type des paramètres);

• l'envoi du signal à la boîte aux lettres de la tâche ou à l'instance de processus identifiée par le Pid via la primitive G2S_OUTPUT.

Exemple: L'envoi du signal *Releasee_ind* contenant les paramètres *cr* de type *call_reference_type* et *rc* de type *release_struct* par une instance du processus *Proc* vers le processus *Coord* est générée dans le fichier **p_proc.c** de la façon suivante:

```
SDL_OUTPUT_release_ind(G2P_FINDPID(coord),&(G2_VAR(cr) ),&(G2_VAR(rc) ));
```

Puisque le signal *Release_ind* est declaré au niveau système, la définition de cette macro se trouve dans le fichier **s_atm_uni.h**:

```
#define SDL_OUTPUT_release_ind(pid,param1,param2) \
  { G2S_LOC_SIGNAL \
        G2S_CREATE(sig_release_ind,sizeof(SIGNAL_release_ind),&sig_p);\
        G2M_COPY_REC(&(((SIGNAL_release_ind*)sig_p)->sg1_CALL_REFERENCE_TYPE),\
                                                    (param1),GU_CALL_REFERENCE_TYPE);\
        G2M_COPY_REC(&(((SIGNAL_release_ind *)sig_p)->sg2_RELEASE_STRUCT),\
                                                    (param2),GU_RELEASE_STRUCT);\
        G2S_OUTPUT(sig_release_ind,(pid),sig_p);\
    }
```

5.5.4.6 Entrée

Le code C généré pour une entrée SDL (INPUT) est une primitive dont le nom est obtenu en ajoutant **SDL_INPUT_** au nom SDL du signal entrant. Les paramètres de la primitive sont les paramètres du signal. La définition de la primitive générée pour l'entrée se trouve dans le fichier d'inclusion (**.h**) de l'objet structural où l'entrée est définie. Son implantation extrait les valeurs des paramètres reçus.

Exemple: La réception, par le processus *Coord*, du signal *Release_resp* contenant les paramètres *cr* de type *call_reference_type* et *r* de type *release_type* est générée dans le fichier **p_coord.c** à l'interieur de la transition *tra_X_release_resp* de la façon suivante:

```
SDL_INPUT_release_resp(G2_VAR(cr) ,G2_VAR(r) );
```

Puisque le signal *Release_resp* est declaré au niveau du système, la définition de cette macro se trouve dans le fichier d'inclusion du système, **s_atm_uni.h**, et consiste en deux affectations:

```
#define SDL_INPUT_release_resp(var1,var2) \
        G2M_COPY_REC(&var1,&G2S_PARAM(SIGNAL_release_resp, \
                                                    sg1_CALL_REFERENCE_TYPE), \
                                        GU_CALL_REFERENCE_TYPE);\
        G2M_COPY_REC(&var2,&G2S_PARAM(SIGNAL_release_resp, \
                                                    sg2_RELEASE_STRUCT), \
                                        GU_RELEASE_STRUCT)
```

L'identificateur du signal entrant current est INPUT_SIGNAL.

5.5.4.7 Procédure

Chaque procédure a son propre contexte représenté par ses paramètres formels et ses variables locales. Une procédure SDL peut être implantée comme une fonction C ou comme trois fonctions C, selon le type de correspondance choisi (TI, TP ou TB), le nombre de ses états (aucun ou au moins un) et le type du processus qui la contient (mono ou pluri-instancié). Le contexte de la procédure du premier type est représenté par des variables C locales à la fonction, tandis que le contexte d'une procédure implantée comme trois fonctions C est une structure C.

Il est possible de choisir, à l'aide de la variable d'environnement GEODE_FILE_PROCED, l'endroit où les procédures sont générées:

• dans le même fichier que le processus qui les appelle;

• chacune dans un fichier;

• dans le même fichier que le processus qui les appelle, à l'exception de celles marquées (par un commentaire spécial) qui seront générées dans des fichiers à part.

Le corps des fonctions C qui implantent des procédures SDL varie selon l'endroit où elles sont générées.

Dans notre application, toutes les procédures sont générées dans le même fichier que le processus qui les appelle et chacune correspond à une fonction C dont les variables locales sont des variables locales C.

Exemple: La procédure de vérification de l'élément d'information *QoS* apparaît dans le fichier qui l'appelle, i.e. **p_proc.c**, de la façon suivante:

```
SDL_PROCEDURE(qos_verif,(GU_QOS_TYPE * fpp_ie_p,SDL_BOOLEAN * fpp_error_free_p))
{
 GU_QOS_TYPE fpp_ie;
 GX_INT_T sav_state;

 G2K_PRD_INIT(proc);          /* cette primitive initialise les resources du processus */

 sav_state = CURRENT_STATE;

 G2M_COPY_REC(&fpp_ie,fpp_ie_p,GU_QOS_TYPE);

/* Transition 'START' */
/*====================*/
tra_START:

/* corps de la procédure */
...
 CURRENT_STATE = sav_state;
 SDL_RETURN();
}
```

5.5.4.8 Instances de processus

Contexte d'instance

Chaque instance de processus gère ses propres variables qui représentent son contexte. (Les variables locales et les paramètres formels font partie du contexte d'une instance.) Le contexte d'une instance est généré comme des appels de macros et de primitives génériques. G2I_BEG_CTX() et G2I_END_CTX() définissent le type du contexte (*static*, *structure* etc.), tandis que les déclarations des variables sont réalisées par des primitives dont le nom a le préfix G2I_DECL_. Dans le mapping TP, le contexte d'une instance est représenté par une structure C. Son pointeur est stocké dans une variable de contexte de la tâche et il est mis à jour pour chaque message qui arrive dans la boîte aux lettres.

Le contexte d'une instance comprend:

• les variables de gestion de l'instance i.e. les variables utilisées dans l'exécution de l'instance, par exemple *ctx_state* qui contient l'état courant de l'instance; ces variables ont le préfix **ctx_**. Elles sont déclarées globalement en utilisant la primitive G2I_DECL_CTRL() et sont accédées en utilisant la primitive G2I_CTX(*variable*).

• les variables SDL, comme les paramètres formels (FPAR) ou les variables déclarées au niveau de processus (DCL); elles sont déclarées individuellement en utilisant des primitives dont le préfix est G2I_DECL_, peuvent être accédées via les primitives G2_FPA(variable), G2_VAR(variable) et peuvent être identifiées par les préfix **fpa_** et **var_** respectivement.

Exemple: La déclaration du contexte d'instance du processus *Coord* est générée dans le fichier **p_coord.h** comme suit:

```
G2I_BEG_CTX()

G2I_DECL_CTRL()

/* déclaration des variables locales est faite par la primitive G2I_DECL_VAR(type, variable) */
G2I_DECL_VAR(GU_MESSAGE_CONTENT_TYPE,in_message)
G2I_DECL_VAR(GU_MESSAGE_CONTENT_TYPE,out_message)
G2I_DECL_VAR(GU_IE_TYPE,ies)
G2I_DECL_VAR(GU_SETUP_STRUCT,s)
```

```
G2I_DECL_VAR(GU_CALL_PR_STRUCT,cp)
G2I_DECL_VAR(GU_CONNECT_STRUCT,c)
G2I_DECL_VAR(GU_RELEASE_STRUCT,r)
G2I_DECL_VAR(GU_CALL_REFERENCE_TYPE,cr)
G2I_DECL_VAR(SDL_PID,proc_id)
G2I_DECL_VAR(SDL_BOOLEAN,more_connections)
G2I_DECL_VAR(SDL_BOOLEAN,active_call)
G2I_DECL_VAR(GU_IE_VALUES_SET,ic_ie)
G2I_DECL_VAR(GU_OCC_ARRAY_TYPE,u_ie)
G2I_DECL_VAR(SDL_NATURAL,last_cr)
G2I_DECL_VAR(GU_HEXA_ARRAY_TYPE,ht)
G2I_DECL_VAR(SDL_NATURAL,cv)

G2I_END_CTX()
```

Création et démarrage d'une instance

Une instance est créée soit par le noeud (les instances initiales), soit par l'instance d'un autre processus d'un même bloc par l'appel de la macro SDL_CREATE_nom_du_processus. Cette macro est définie par une autre, G2I_CREATE, dont l'implantation varie selon la valeur de la variable d'environnement GEODE_REMOTE_CREATE. Dans notre cas la macro G2I_CREATE est implanté comme un appel direct de la fonction de création de l'instance du processus respectif.

La fonction de création d'une instance de processus est générée dans chaque fichier de processus et:

- crée et initialise un nouveau contexte d'instance et met à jour le contexte de la tâche correspondante G2I_CRE_INST;
- crée et initialise les variables SDL;
- crée un PID pour la nouvelle instance et provoque la transition initiale de l'instance (START) par l'envoi d'un pseudo-signal START: G2I_CRE_START;
- retourne le PID de l'instance nouvellement créée.

Exemple. Le processus *Coord*, dont le code C est généré dans le fichier **p_coord.c**, crée les instances du processus *Proc* par l'appel de la macro SDL_CREATE_proc(). En fait cette macro appelle la fonction de création d'instances de processus *gg_cre_proc*. La fonction

gg_cre_proc se trouve dans le fichier **p_proc.c** contenant le code C du processus *Proc*. Le code C de la fonction est:

```
SDL_PID gg_cre_proc(SDL_PID parent)
{
 SDL_PID new_pid;

 G2I_CRE_INST(proc);

 G2M_INIT_REC(&(G2_INIT_VAR(u_ie)),GU_OCC_ARRAY_TYPE);
 G2M_INIT_SET(G2_INIT_VAR(ic_ie),GU_IE_VALUES_SET);
 G2M_INIT_REC(&(G2_INIT_VAR(diag)),GU_OCC_ARRAY_TYPE);
 G2M_INIT_REC(&(G2_INIT_VAR(ie)),GU_IE_TYPE);
 G2M_INIT_REC(&(G2_INIT_VAR(ies)),GU_IE_TYPE);
 G2M_INIT_REC(&(G2_INIT_VAR(s)),GU_SETUP_STRUCT);
 G2M_INIT_REC(&(G2_INIT_VAR(cp)),GU_CALL_PR_STRUCT);
 G2M_INIT_REC(&(G2_INIT_VAR(c)),GU_CONNECT_STRUCT);
 G2M_INIT_REC(&(G2_INIT_VAR(r)),GU_RELEASE_STRUCT);
 G2M_INIT_REC(&(G2_INIT_VAR(rc)),GU_RELEASE_STRUCT);
 G2M_INIT_REC(&(G2_INIT_VAR(st)),GU_STATUS_STRUCT);
 G2M_INIT_REC(&(G2_INIT_VAR(cr)),GU_CALL_REFERENCE_TYPE);
 G2I_CRE_START(proc);

 return(new_pid);
}
```

Arrêt d'une instance

La macro SDL_STOP_nom_de_processus() est générée pour l'arrêt d'une instance de processus. Elle est implantée comme un appel de la primitive, G2I_STOP() qui est implantée comme un appel de la fonction gg_stp_process(void) générée dans le fichier du processus.

Exemple: On trouve dans le fichier **p_proc.h** la définition de la macro:

```
#define SDL_STOP_proc() \
    G2I_STOP(proc)
```

La primitive G2I_STOP(proc) arrête l'instance de processus en appellant la fonction

```
static void gg_stp_proc(void)
{
 G2I_STP_INST(proc);
}
```

qui se trouve dans le fichier **p_proc.c.** La primitive de l'intérieur de la fonction libère le contexte de l'instance détruite et met à jour le contexte de la tâche correspondante.

5.6 Code C externe

Plusieurs raisons nous ont déterminé à écrire du code C séparément:

- la nécessité d'adapter le code généré à des besoins spécifiques;
- la limitation du langage SDL qui ne permet par la déclaration de pointeurs;
- la vitesse d'exécution.

Ce code C externe a dû être intégré au code C généré par l'outil.

5.6.1. Rédéfinition des types et des primitives

Les types de données abstraites permettent la définition des types de données en termes de propriétés. L'implantation d'un type de donnée est réalisée par les librairies d'exécution. Celles-ci contiennent des macros et des primitives (en code source) et peuvent être modifiées selon les besoins.

Nous avons modifié la définition du type hexadécimal et celle des appels de procédures qui réalisent la gestion des connexions.

Les messages échangés entre les entités apariées du protocole (PDUs) sont transportés comme des chaînes de bits regroupés en octets. Dans notre spécification SDL, une PDU est représentée par un tableau dont les éléments sont de type hexadécimal. Dans le code C généré, ce type est designé par la macro SDL_HEXADECIMAL dont la définition par défault est le type C *unsigned int*. Pour forcer la représentation d'un élément du tableau sur un octet on a redéfini la macro comme étant *unsigned char*. Cette opération est faite dans le fichier d'inclusion **hpostdef.h** (Annexe 3) de la manière suivante:

```
#undef SDL_HEXADECIMAL
#define SDL_HEXADECIMAL unsigned char
```

Les limitations du langage SDL qui ne permet pas la manipulation des pointeurs et l'optimisation en termes de rapidité nous ont amenés à représenter certaines aspects de la spécification par des fonctions C. C'est le cas des fonctions de sauvegarde, de recherche et de suppression des paires (référence d'appel, identificateur de processus), ainsi que celles de codage et décodage. Dans notre spécification SDL, ces opérations sont désignées par des appels des procédures non-paramétrées (*put*, *find_proc*, *release_cr*, *code* et *decode*) dont le contenu a été laissé vide. Il a fallu donc redéfinir les macros générées pour ces appels par les appels des fonctions C correspondantes, en ajoutant les paramètres nécessaires.
Par exemple, la redéfinition de la macros qui appellent les procédures *decode* et *put* est faite dans le fichier **hpostdef.h** de la façon suivante:

```
#undef SDL_CALL_decode
#define SDL_CALL_decode() \
  decoder(G2_VAR(ht), &(G2_VAR(mess) ), G2_VAR(ht_length), G2_VAR(ic_ie), &G2_VAR(u_ie))

#undef SDL_CALL_put
#define SDL_CALL_put() \
    put(G2_VAR(cr), SDL_OFFSPRING)
```

Les variables sont accédées via la primitive G2_VAR et l'identificateur de l'instance de processus la plus récente est désigné par la macro SDL_SELF.

5.6.2. Fonctions C

5.6.2.1 Gestion des identificateurs de connexions

La structure d'arbre binaire que nous avons choisie pour l'organisation des identificateurs de connexions (i.e la référence d'appel et l'identificateur de l'instance de processus correspondante) implique des opérations qui nécessitent la déclaration et la manipulation de pointeurs, ce qui n'est pas possible en SDL. Par conséquent, ces opérations ont été écrites comme fonctions C et ont été intégrées dans le code C généré par l'outil.

Nous avons défini un élément de l'arbre comme étant une structure dont les champs sont:

- la référence d'appel, de type GU_CALL_REFERENCE_TYPE, i.e. une structure dont les champs sont la valeur, *fd_value*, et le drapeau, *fd_flag*;
- l'identificateur de l'instance de processus correspondante, de type SDL_PID;
- deux pointeurs, *left* et *right*.

```
typedef struct elem
{
 GU_CALL_REFERENCE_TYPE cr;
 SDL_PID pr;
 struct elem * left, *rigth;
} tree;
```

Puisque l'opération de base appliquée sur l'arbre est la recherche d'un élément selon la référence d'appel, nous avons considéré celle-ci comme étant la clé. Deux références d'appel dont les valeurs sont égales (si elles sont attribuées à des extrémités différentes de la connexion) sont distinguées par le drapeau.

Les fonctions C qui réalisent les opérations effectuées sur l'arbre se trouvent dans le fichier d'inclusion **hpostdef.h.**

La fonction C *put* alloue de la mémoire pour un élément de l'arbre, attribue à ses champs les valeurs reçues comme paramètres et appelle la fonction *put_in* qui insère l'élément pointé par le pointeur *new* dans l'arbre dont la racine est pointée par *tr*.

La fonction *put_in* ajoute l'élément pointé par le pointeur *item* dans l'arbre dont la racine est pointée par *tr*. Si l'arbre est vide (pointe vers NULL) l'ajout revient à changer son pointage vers le nouvel élément. Si l'arbre n'est pas vide, l'élément est ajouté en tant que feuille après une descente vers la gauche si sa clé est plus petite ou égale à celle de l'élément courant *tempo* ou vers la droite en cas contraire.

La fonction *find* cherche dans l'arbre l'élément dont la référence d'appel est donnée comme paramètre, *to_find*. Si l'élément se trouve dans l'arbre, le pointeur *pr_id* va pointer vers son deuxième champ (i.e. l'identificateur de l'instance de processus) et *found* vers la valeur SDL_TRUE. Dans le cas contraire, le pointeur *found* pointe vers la valeur SDL_FALSE.

La fonction *release_cr* enlève de l'arbre l'élément dont la référence d'appel est donnée comme paramètre, *to_erase*. Elle recherche dans l'arbre l'élément *him* dont la référence d'appel est égale à *to_erase*. Si elle le trouve, l'élément est supprimé et l'arbre dont il est la racine réorganisé.

5.6.2.2 Codage/décodage

Afin de comparer les vitesses d'exécution, nous avons écrit la procédure de décodage d'un élément d'information (en occurrence BBC) de deux façons: en langage SDL et langage C. La vitesse d'exécution de la fonction écrite en C a été approximativement dix fois plus rapide que l'exécution de la fonction C générée par l'outil. Pour cette raison nous avons choisi d'écrire des fonctions C pour le codage et le décodage d'un message, plutôt que des procédures SDL.

Décodage

La fonction *decoder* réalise le décodage d'une PDU entrante. Les paramètres d'entrée sont:

- le tableau d'octets *c*, qui représente la PDU codée;
- la longueur (en octets) *c_length* de la PDU.

La fonction retourne:

• le pointeur *pmess* de la structure *mess* de type GU_MESSAGE_CONTENT_TYPE; la structure *mess* représente le message résultant du décodage;

• le pointeur *icie* d'un tableau d'éléments de type booléen. L'index du tableau est une énumération des éléments d'information acceptés par le protocole. Un élément du tableau a la valeur SDL_TRUE si la fonction détecte une erreur dans l'élément d'information désigné par son index. Par exemple, si la longueur de l'élément d'information *QoS* (dont l'identificateur est 92) est différente de 2, ce qui constitue une erreur, l'élément *icie[gu_IE_ID_ie92]* aura la valeur SDL_TRUE.

• le pointeur *uie* d'une structure de type GU_OCC_ARRAY_TYPE; le champ *fd_nb_occ* de la structure contiendra le nombre d'éléments d'information non-reconnus (dont les identificateurs ne sont pas définis dans le standard), tandis que le deuxième champ *fd_occ*, de type tableau, contiendra ces identificateurs.

La fonction ne décode que les PDUs dont la longueur est supérieure à 9, i.e. qui contiennent au moins l'en-tête du message. (Les autres PDUs seront de toute façon ignorées par le protocole.) En parcourant le tableau *c*, élément par élément, la fonction identifie chaque composante du message, extrait sa valeur et l'attribue au champ approprié.

Après le décodage de l'en-tête du message, une construction de type *switch* permet l'exécution de la séquence de décodage qui correspond à l'identificateur de l'élément d'information détecté. Les éléments d'information qui peuvent être répétés (*Cause* et *B_LLI*) bénéficient de fonctions de décodage propres (*decode_8* et *decode_95*), qui sont appelées pour chaque occurrence présente dans le message. Si l'identificateur d'un élément d'information ne fait pas partie du standard, il est ajouté dans le tableau des identificateurs non-reconnus *(*uie).fd_occ*.

Un élément d'information est considéré avec contenu invalide dans les cas suivants:

• sa longueur est plus petite que 4 (c'est-à-dire qu'il n'inclut pas l'en-tête au complet);

• la valeur du champ longueur ne correspond pas à la longueur réelle;

• la longueur du contenu d'un élément d'information n'est pas suffisante pour inclure au moins les champs obligatoires;

• s'il est le dernier élément d'information du message et la longueur du message n'est pas suffisante pour l'inclure au complet.

Les valeurs dont la représentation binaire est inférieure à un octet sont calculées en appliquant le masque appropriée et, éventuellement, une opération de décalage (*shift*). Par exemple, la valeur du drapeau de la référence d'appel est donnée par le bit 8 du troisième élément du tableau d'entrée *c*. Par conséquent on applique à *c[3]* le masque *1000 0000* pour annuler la valeur des bits 0-7. Le décalage de 7 positions à droite permet de retenir juste la valeur du $8^{ième}$ bit.

Les valeurs des champs dont la longueur dépasse un octet sont calculées par des multiplications de 2^8. Par exemple, la longueur d'un élément d'information s'étend sur deux octets, soitent-ils *c[i]* et *c[i+1]*. La valeur de la longueur sera $c[i]*2^8+c[i+1]$.

Codage

La fonction *code* transforme une structure représentant le message sortant (PDU) dans un tableau d'octets. Elle possède deux paramètres:

- un paramètre d'entrée, *m2code*, de type GU_MESSAGE_CONTENT_TYPE, qui représente le message à coder;
- un paramètre de sortie, *c*, un tableau d'octets, représentant la forme codée de la PDU.

Les neuf premiers éléments du tableau *c* contiennent l'en-tête du message. Les éléments suivants contiennent le code des éléments d'information présents dans la structure, chacun précédé par son identificateur. On rencontre plusieurs situations de codage:

- un champ doit être codé sur certains bits d'un octet. Dans ce cas-là, l'octet aura la valeur du champ décalée à gauche. Par exemple, la valeur du champ *fd_coding_standard* (0 ou 3), qui apparaît dans l'en-tête de chaque élément d'information, doit être représentée sur les bits 7 et 6 d'un octet. Par conséquent, la valeur de l'octet de code sera obtenue par le décalage de 5 positions vers la gauche de la valeur du champ.
- plusieurs champs doivent être codés sur un même octet. Dans ce cas-là, la valeur de l'octet sera la somme des valeurs des champs décalées à gauche selon leurs positions. Par exemple, dans l'élément d'information *Broadband Bearer Capability*, la valeur du champ *fd_traffic_type* doit être représentée sur les bits 5,4 et 3, tandis la valeur du champ *fd_timmig_req* sur les bits 2 et 1 d'un même octet. La valeur de l'octet sera la somme entre la valeur du premier champ décalé à gauche de 2 positions et la valeur du deuxième champ.

• la valeur d'un champ s'étend sur plusieurs octets. Dans ce cas-là, les valeurs des octets sont obtenues par des opérations de division entière et modulo. Par exemple, la valeur du champ *fd_vpci* de l'élément d'information *Connection identifier* doit être représentée sur deux octets. Le premier contiendra le résultat de la division de la valeur du champ par 2^8, tandis que le deuxième le reste de cette division.

• un ou plusieurs champs sont représentés par un groupe d'octets; La valeur 0 dans le bit 8 d'un octet du groupe signifie que le groupe continue sur l'octet suivant. Le dernier octet d'un groupe d'octets est désigné par la valeur 1 du bit 8. Par conséquent, lors du codage il faut ajouter 128 (i.e 2^7) à la valeur proprement dite du dernier octet d'un groupe. Par exemple, dans l'élément d'information *Broadband Bearer Capability*, les champs *fd_class* et *fd_class_options* doivent être représentés comme un groupe de deux octets. Si le deuxième champ, *fd_class_options*, est absent de l'élément d'information à coder, on ajoute 128 (i.e. 1 sur le bit 8) à la valeur du premier champ, *fd_class*, pour préciser qu'il s'agit du dernier octet du groupe. Si par contre *fd_class_options* est présent, c'est sa valeur qui sera augmentée de 128, car car c'est lui qui sera codé sur le dernier octet du groupe.

5.7 Commentaires

Dans le cas des système fermés (i.e. qui ne communiquent pas avec l'environnement) et dont le comportement peut être intégralement décrit en SDL, la consultation du code C généré n'est pas nécessaire, car l'outil fournit tout le cadre de développement de l'implantation. Comme habituellement les systèmes sont ouverts et le code C généré nécessite des ajouts, des modifications ou des adaptations, sa compréhension est requise.

Le code C généré par ObjectGEODE est constitué de macro et de primitives. Leurs définitions sont reparties dans plusieurs fichiers localisés dans différents répertoires et généralement, représentent, à leur tour, d'autres appels de macros et de primitives. Elles peuvent contenir des conditions impliquant d'autres macros. Par conséquent, le parcours du chemin entre la représentation C d'une construction SDL et son implantation constitue souvent une tâche ardue.

Les limitations des ressources de l'exécutif provoquent des erreurs fatales dans l'exécution de l'implantation. Bien qu'elle soit référencée par un code, la cause d'une telle erreur est difficile à identifier, d'une part à cause des définitions fortement imbriquées des macros du code généré et, d'une autre part, à cause de l'interposition des codes d'erreurs appartenant au système d'exploitation. Par exemple, l'exécution de notre implantation s'arrêtait en affichant le message:

[0x30001]ERROR 0x24 : (msgsnd in g2s_output) errno = 0x16

dont l'explication trouvée dans le fichier d'erreurs de l'outil,

ERROR number 0x24 :

invoked by bad msgsnd status in g2s_output function

cause: cannot send message to the receiver message queue

present in g2_lib.c file,

ne fournit pas une solution explicite.

Après des recherches assidues et des nombreux essais, nous avons découvert que l'erreur provenait du fait que la taille d'un message de notre spécification dépassait la taille maximum du message acceptée par le système d'exploitation. La solution a été l'augmentation des valeurs des variables de système fixant la taille maximum d'un message et celle de la file d'un processus Unix. (Cette opération est faite par l'administrateur du système et requiert le rédemarage (*reboot*) de la machine visée.) Toutefois, il arrive que lors de l'exécution la file devienne pleine et un nouveau message lui étant adressé provoque une erreur.

Un autre problème relié aux ressources du système d'exploitation rendait impossible la répétition de l'exécution de l'implantation. Il était dû au fait que, dans certains cas (par exemple, une fin anormale de l'exécution), les ressources allouées pendant l'exécution (sémaphores, memoire partagée, queues) restaient actives. Leur libération peut être faite par l'exécution du fichier de commandes **g_terminator** fourni par l'outil, mais, selon notre expérience, pour s'assurer de son bon fonctionnement, il faut tuer (si c'est le cas) tous les processus de l'exécution anterieure.

5.8 Performances

Le développement du protocole de signalisation a été réalisé sur une station Sparc-20 avec 128 MB de mémoire principale, sous le système d'exploitation SunOS, version 5.4. Nous avons utilisé l'outil ObjectGEODE.

La description SDL du protocole a été réalisée en forme graphique, en utilisant l'éditeur ObjectGEODE SDL Editor version 3.0. Le diagramme contient plusieurs branchements (envois du contrôle par des étiquettes) imposés par la mise en page plutôt que par les nécessités de la spécification. Le fichier **atm_sign.pr**, contanant la forme textuelle, est de 389402 octets. Il inclut des commentaires qualifiés i.e. des commentaires qui commencent par le caracrère "#" et qui contiennent des directives pour la représentation graphique (par exemple, les dimensions d'un symbole graphique). La compilation de la spécification SDL dure quelques minutes. La génération de code C, réalisé par SDL C Code Generator, version 2.3, dure, elle aussi, quelques minutes.

La taille (en octets) des fichiers C générés est:

atm_sign.cfg	12	i_atm_uni_sp.c	761
atm_sign.lc	52	p_coder_decoder.c	6957
atm_sign.lch	22	p_coder_decoder.h	1571
atm_sign.lh	99	p_coord.c	49423
atm_sign.log	2854	p_coord.h	1571
atm_sign.lp	49	p_proc.c	303888
atm_sign.lt	175	p_proc.h	12197
atm_sign.mk	4378	r_coder_decoder.h	712
atm_sign.ref	9365	r_coord.h	664
atm_sign.rmk	16134	r_proc.h	658
b_q2931.h	19721	s_atm_uni_sp.h	38390

Les fichiers contenant les variables d'environnement utilisées dans la génération du code C et dans la compilation et l'édition de liens, **geodeb_unix_env** et **gce_unix**, adaptés au besoins de notre implantation ont les tailles de 4658 et 3236 octets respectivement.

Les fichiers **hpredef.h** et **hpostdef.h** que nous avons créés dans le but d'adapter le code C généré par l'outil et et d'ajouter du code C externe ont la taille:

hpostdef.h 694

hpredef.h 56

Les fichiers contenant le code source des librairies et ceux de commandes utilisés dans la compilation et l'édition des liens se trouvent dans le sous-répertoire **/cross/geo_unix** du répertoire d'installation de ObjectGEODE. Ils sont regroupés dans deux sous-répertoires, **scr** dont la taille est de 174162 octets et **cux_sun4_5** dont la taille est de 20427 octets. L'installation des librairies d'exécution a été faite pour tous les types de mapping et pour les deux types d'exécution: avec et sans affichage des traces. Il en a résulté les sous-répertoires suivants:

lux_sun4_5_TI,	lux_sun4_5_DTI
lux_sun4_5_TP	lux_sun4_5_DTP
lux_sun4_5_TB	lux_sun4_5_DTB

Les outils nécessaires à l'exécution (comme le fichier **geoderun** qui lance l'exécution) se trouvent dans le sous-répertoire **tux_sun4_5** et occupent 5344 octets.

La taille (en octets) des fichiers exécutables résultant de la compilation de notre application est:

n_atm_uni_sp.rt	23000
t_SOCKET.rt	19980
t_TIMER.rt	20964
t_coder_decoder.rt	104568
t_coord.rt	172840
t_proc.rt	450592

Pour pouvoir tester notre implantation (qui représente un système ouvert), il faut simuler le comportement de l'environnement. L'environnement de notre système est représenté par le processus d'application et par la couche SAAL. Les deux envoient des messages à notre implantation et ils en reçoivent. Les messages échangés avec la couche SAAL représentent en fait les messages échangés avec le réseau. Nous avons simulé le comportement de l'environnement par un programme C, **ext_task.c**, et nous avons choisi comme exemple d'exécution l'établissement de dix connexions requises par des processus d'application. Dans la modélisation de la réponse du réseau nous avons supposé qu'il envoie les messages appropriés et sans erreurs. Le temps écoulé entre l'envoi de la première demande de connexion et l'établissement complet de la dernière connexion a été de 60000 microsecondes, c'est à dire 0.06 secondes. Il représente la durée de traitement des messages par notre implantation et, puisque le protocole se déroule de la même façon dans les deux cotés d'une connexion, il faut le multiplier par deux. Évidemment dans une telle expérimentation on ne peut pas prendre en considération le délai de transfert sur le réseau, ni le temps de traitement des messages par celui-ci. Généralement, dans un contexte réel, l'établissment d'une connexion est acheminée par plusieurs noeuds. Sa durée va inclure, outre le temps de traitement des messages à la source et à la destination, le temps de traitement dans les noeuds et le temps de propagation entre ceux-ci.

6. Conclusion

Le but du présent projet de maîtrise consistait à expérimenter l'utilisation d'une spécification formelle dans le développement d'un système, à savoir la réalisation d'une implantation du protocole de signalisation ATM, à l'interface utilisateur-réseau (UNI), du côté de l'utilisateur.

La spécification formelle du protocole de signalisation ATM UNI que nous avons développée a été réalisée en langage SDL. Elle a été exploité avec l'outil ObjectGEODE.

L'éditeur graphique de l'outil nous a servi à la création du modèle SDL. Les erreurs de syntaxe et de sémantiques SDL ont pu être détectées par compilation. La compilation de la spécification exempte de ce type d'erreurs, produit le fichier exécutable qui sera utilisé par le simulateur.

Le modèle SDL représentant le protocole de signalisation a été vérifié ("le système fonctionne?") et validé ("le système fait ce qu'il est supposé faire?") à différents niveaux de raffinement, par simulation guidée et aléatoire. Cela nous a permis de détecter et corriger, dans notre spécification SDL, une série d'erreurs de conception (par exemple, l'omission de réinitialiser un compteur). La simulation exhaustive, qui aurait offert une garantie totale quant au respect des propriétés vérifiées, n'a pas pu être appliquée à cause de l'explosion combinatoire du nombre des situations à explorer.

L'outil permet également la génération automatique de code C pour différents environnements d'exécution, à partir de la spécification formelle. Dans notre cas le code a été généré pour un environnement Unix et il a été adapté à des besoins spécifiques. Différents aspects du comportment du protocole n'ont pu être couverts par la spécification formelle à cause des limitations du langage (qui ne permet pas la manipulation des pointeurs). Nous les avons exprimés en langage C. Nous avons également écrit du code C pour certaines sections (codage/décodage) afin d'optimiser le temps d'exécution. Ce code externe, nous l'avons intégré dans le code généré par l'outil afin d'obtenir une implantation complète. Un problème que nous avons rencontré avec le code C généré était qu'il faisait appel à des macros fortement imbriquées, ce qui rendait la compréhension de l'implantation laborieuese.

L'implantation du protocole de signalisation que nous avons réalisée ne prend pas en considération les procédures de rédémarrage (RESTART), ni le cas des connexions point-multipoint. Ce sont des aspects à ajouter.

Puisqu'il existe différents outils associés au langage SDL (on peut mentionner, outre ObjectGEODE, SDT), il serait intéressant de comparer les codes qu'ils génèrent.

Un autre aspect intéressant à réaliser serait l'intégration du protocole dans un environnement d'exécution réel. Cela implique la conception des interfaces du protocole de signalisation avec un programme d'application (AP) concret et avec le protocole de signalisation de la couche d'adaptation ATM existante. Les interfaces vont servir à faire une correspondance entre les primitives accéptées par le protocole de signalisation ATM et les fonctions utilisées au niveau application ou les primitives de niveau SAAL. À ce moment-là, on pourrait envisager la réalisation des connexions dont la signalisation est réalisée par notre implantation. Un exemple d'interface API disponible sur le marché est Winsock [22]. Elle permet la communication entre deux applications dans un environnement Windows.

7. Bibliographie

[1] Angosto Paul, Caillaud Benoît, Delaure Michel, Guesrchel Rémi, Jouga Bernard, Le Foll, Dominique, Maret Jean-Jacques, Rocher Pierre, Vaucher Gilles, *L'inginérie des protocoles*, InterEditions, Paris, 1993

[2] ATM Forum, *ATM User-Netwok Interface Specification, version 3.1,* 1994

[3] Belina Ferec, Hogrefe Dieter, *The CCITT-Specification and Description Language SDL*, Computer Networks and ISDN Systems, vol. 16, no. 4, pp. 311-341, 1989

[4] Bochmann Gregor von, *Notes du cours IFT 6052 "Sujets avancés en téléinformaique*, Université de Montréal, Hiver 1996

[5] Bochmann G. von, Das A., Dssouli R., Dubuc M., Ghedamsi, Luo G., *Fault Models in Testing*, IFIP Transactions, Protocol Test Systems, IV (Proceedings of IFIP TC6 Fourth International Workshop on protocol Systems, 15-17 october, 1991), North-Holland, 1992

[6] Bochmann G. von, *Protocol Specification for OSI*, Computer Networks and ISDN Systems, vol. 18, no. 3, pp. 167-184, 1990

[7] Budgen David, *Software Design*, Addison-Wesley, 1994

[8] Clark Martin P., *ATM Networks Principles and Use*, Wiley and Teubner, 1996

[9] Færgemand O., Olsen A., *Introduction to SDL-92*, Computer Networks and ISDN Sytems vol. 26, pp. 1143-1167, 1994

[10] Gao Q., Groz R., Bochmann G. v., Dargham J., Htite E.H., *Validation of Distributed Algorithms and Protocols*, Rapport technique #845, Univérsité de Montréal, 1993

[11] Händel R., Huber M. N., Schröder S., *Comprendre ATM*, Addison-Wesley, France, 1995

[12] ITU-T: Draft Recommendation Q.2931, *B-ISDN User_network Interface Layer 3 Protocol*, Geneva, 1993

[13] Lorrains, *Réseaux téléinformatiques*, Hachette, Paris, 1979

[14] Macchi C., Guilbert J.-F., *Téléinformatique*, Bordass et C.N.E.T.-E.N.S.T., Paris 1983

[15] Nussbaumer Henri, *Tétéinformatique*, Presse Polytechnique romandes, Lausanne,1987

[16] Prycker M. de, *ATM Mode de transfert asynchrone*, Prentice Hall, London, 1994

[17] Rayner D., *OSI Conformance Testing*, Computer Network and ISDN Systems vol. 14, no. 1, pp. 79-98, 1987

[18] Rolin Pierre, *Réseau haut débit*, Hermes, Paris 1995

[19] Sarikaya, B., *Principles of protocol engineering and conformance testing*, Ellis Horwood series in computers and their applications, 1993

[20] Tanenbaum Andrew S., *Computer Networks*, Prentice Hall, London, 1990

[21] UIT-T: Recommandation UTI-T Z.100, *Langage de description et de spécification du CCITT* (1993)

[22] Vérilog SA, *ObjectGEODE - Reference Manual*, France, 1996

[23] Weiss Bernard, *ATM*, Hermès, Paris, 1995

[24] http://www.stardust.com/wsresource/winsock2/ws2sdk.html

Annexe 1

Le tableau d'états

Tableau d'états du protocole de signalisation UNI du côté utilisateur - version 3.1

State	Input	Type	Condition	Output	Action	Next State	Ref.
U0	Setup_req			SETUP	start T303	U1	5.5.1.1.
	SETUP protocol discriminator (M) call reference-CR (M) message type (M) message length (M) AAL parameters (O) ATM traffic descriptor (M) broadband bearer capability (M) broadband high layer inf. (O) broadband repeat indicator (O) broadband low layer inf. (O) called party number (M) called party subadress (O) calling party number (O) calling party subadress (O) connection identifier (M) Qos parameter (M) broadband sending complete (O) transit network selection (O) endpoint reference (O[1])		mandatory information element missing	RELEASE_COMPLETE (cause #96)	release CR	U0	5.5.6.7.1 5.5.4.2
			mandatory information element with invalid content	RELEASE_COMPLETE (cause #100)	release CR	U0	5.5.6.7.2 5.5.4.2
			indicated VCI not available within indicated VPCI	RELEASE_COMPLETE (cause #35)	release CR	U0	5.5.2.3 5.5.4.2
			unable to provide QoS class	RELEASE_COMPLETE (cause #49)	release CR	U0	5.5.2.4 5.5.4.2
			unable to provide indicated ATM traffic	RELEASE_COMPLETE (cause #47)	release CR	U0	5.5.2.4 5.5.4.2
			unrecognized information element	STATUS (cause #99, call state U6) Setup_ind		U6	5.5.6.8.1 5.5.6.8.3 5.5.2.1
			non-mandatory information element with invalid content	STATUS (cause #100, call state U6) Setup_ind		U6	5.5.6.8.2 5.5.2.1
			otherwise (o.k.)	Setup_ind		U6	5.5.2.1

[1] Not used for point-to-point connexion establishment. Must be included in SETUP message involved in point-to-point connexion establishment. Than, it will be included in all messages.

U0 - Null
U1 - Call Initiated
U3 - Outgoing Call Proceeding
U6 - Call Present
U8 - Connect Request
U9 - Incoming Call Proceeding
U10 - Active
U11 - Realease Request
U12 - Release Indication

State	Input	Type	Condition	Output	Action	Next State	Ref.
U1	CALL_PROCEEDING protocol discriminator (M) call reference (M) message type (M) message length (M) connector identifier (M[2]) endpoint reference (O)		mandatory information element missing	STATUS (cause #96, call state U1)		U1	5.5.6.7.1
			mandatory information element with invalid content	STATUS (cause #100, call state U1)		U1	5.5.6.7.2
			unrecognized information element	STATUS (cause #99, call state U3) Proceeding_ind	stop T303 start T310	U3	5.5.6.8.1 5.5.1.5
			non-mandatory information element with invalid content	STATUS (cause #100, call state U3) Proceeding_ind	stop T303 start T310	U3	5.5.6.8.2 5.5.1.5
			otherwise (o.k)	Proceeding_ind	stop T303 start T310	U3	5.5.1.5.
	Timeout T303		first expiry	SETUP	restart T303	U1	5.5.1.1.
			second expiry	Release_conf	release CR	U0	5.5.1.1
	RELEASE_COMPLETE protocol discriminator (M) call reference (M) message type (M) message length (M) cause (M[3])		cause missing or with invalid content	Release_conf (cause #31)	release all resources	U0	5.5.6.7.1 5.5.6.7.2
			otherwise (unrecognized information element or o.k.)	Release_conf	release CR, release VC	U0	5.5.1.2.1 5.5.1.3 5.5.1.4 5.5.1.5. 5.5.4.3 5.5.6.8.1 b
	RELEASE protocol discriminator (M) call reference (M) message type (M) message length (M) cause (M)	U		RELEASE_COMPLETE Release_conf	release VC, release CR, stop all timers	U0	5.5.2.1 5.5.2.3 5.5.6.4

[2] Mandatory if this message is the first message in response to SETUP (5.3.1.2).
[3] Mandatory since, received in this state, the message is the first clearing message (5.3.1.6).

State	Input	Type	Condition	Output	Action	Next State	Ref.
U3	CONNECT protocol discriminator (M) call reference (M) message type (M) message length (M) AAL parameters (O) broadband low layer inf. (O) connection identifier (O[4]) endpoint reference (O)		mandatory information element missing	STATUS (cause #96, call state U3)		U3	5.5.6.7.1
			mandatory information element with invalid content	STATUS (cause #100 call state U3)		U3	5.5.6.7.2
			unrecognized information element	STATUS (cause #99, call state U10) Setup_conf CONNECT_ACK	stop T303 or T310	U10	5.5.6.8.1
			non-mandatory information element with invalid content	STATUS (cause #100, call state U10) Setup_conf CONNECT_ACK	stop T303 or T310	U10	5.5.6.8.2
			otherwise (o.k.)	Setup_conf CONNECT_ACK	stop T303 or T310	U10	5.5.1.7.
	Timeout T310			RELEASE (cause #102)	start T308	U11	5.5.1.5.

[4] Mandatory if this message is the first message in the response to a SETUP message.

State	Input	Type	Condition	Output	Action	Next State	Ref.
U6	Setup_resp			CONNECT	start T313	U8	5.5.2.6.
	Proceeding_req			CALL_PROCEEDING		U9	5.5.2.5.1
	Release_resp			RELEASE_COMPLETE (cause #88, #17, #21 or #23)		U0	5.5.2.5.1 5.5.2.3 5.5.2.4.

State	Input	Type	Condition	Output	Action	Next State	Ref.
U8	CONNECT_ACK protocol dicriminator (M) call reference (M) message type (M) message length (M)		mandatory information element missing	STATUS (cause #96, call state U8)		U8	5.5.6.7.1
			mandatory information element with invalid content	STATUS (cause #100, call state U8)		U8	5.5.6.7.2
			unrecognized information element	STATUS (cause #99, call state U10) Setup_complete_ind	stop T313	U10	5.5.6.8.1
			otherwise (o.k.)	Setup_complete_ind	stop T313	U10	5.5.2.7.
	Timeout T313			RELEASE (cause #102)	start T308 disconnect VC	U11	5.5.2.7.

State	Input	Type	Condition	Output	Action	Next State	Ref.
U9	Setup_resp			CONNECT	start T313	U8	5.5.2.5.1

State	Input	Type	Condition	Output	Action	Next State	Ref.
U11	RELEASE_COMPLETE protocol discriminator (M) call reference (M) message type (M) message length (M) cause (O[5])		non-mandatory information element with invalid content	STATUS (cause #100, current state U0) Release_conf	stop T308, release CR, release VC	U0	5.5.4.3 5.5.6.8.2
			otherwise (unrecognized information element or o.k.)	Release_conf	stop T308, release CR, release VC	U0	5.5.4.3 5.5.6.8.1 b)
	RELEASE protocol discriminator (M) call reference (M) message type (M) message length (M) cause (M)		cause missing	Release_conf	stop T308, release CR, release VC	U0	5.5.6.7.1 5.5.4.5
			invalid content of cause	Release_conf	stop T308, release CR, release VC	U0	5.5.6.7.2 5.5.4.5
			unrecognized information element	Release_conf	stop T308, release CR, release VC	U0	5.5.6.8.1
			otherwise (o.k.)	Release_conf	stop T308, release CR, release VC	U0	5.5.4.5
	Timeout T308		first expiry	RELEASE (add cause #102)	restart T308	U11	5.5.4.3
			second expiry	Release_conf *Note 3*	release CR, release VC	U0	5.5.4.3

[5] Non-mandatory because in this state this message is not the first call clearing message

State	Input	Type	Condition	Output	Action	Next State	Ref.
	STATUS protocol discriminator (M) call reference (M) message type (M) message length (M) call state (M) cause (M) endpoint reference (O) endpoint state (O)		mandatory information element missing	STATUS (cause #96, call state U11)		U11	5.5.6.7.1
			mandatory information element with invalid content	STATUS (cause #100, call state U11)		U11	5.5.6.7.2
			unrecognized information element & call state # U0	STATUS (cause #99, call state U11)		U11	5.5.6.8.1 5.5.6.12. b
			unrecognized information element & call state = U0	STATUS (cause #99, call state U0) Release_conf	release all resources	U0	5.5.6.8.1 5.5.6.12
			non-mandatory information element with invalid content & call state # U0	STATUS (cause #100, call state U11)		U11	5.5.6.8.2 5.5.6.12.b
			non-mandatory information element with invalid content & call state = U0	STATUS (cause #100, call state U0) Release_conf	release all resources	U0	5.5.6.8.2 5.5.6.12.c
			call state # U0			U11	5.5.6.12 b
			call state = U0	Release_conf	release all resources	U0	5.5.6.12 c

U0 - Null
U1 - Call Initiated
U3 - Outgoing Call Proceeding
U6 - Call Present
U8 - Connect Request
U9 - Incoming Call Proceeding
U10 - Active
U11 - Realease Request
U12 - Release Indication

State	Input	Type	Condition	Output	Action	Next State	Ref.
U12	Release_resp			RELEASE_COMPLETE (cause #16)	release VC, release CR	U0	5.5.4.4
	RELEASE	U				U12	
	STATUS protocol discriminator (M) call reference (M) message type (M) message length (M) call state (M) cause (M) endpoint reference (O) endpoint state (O)		mandatory information element missing	STATUS (cause #96, call state U12)		U12	5.5.6.7.1
			mandatory information element error	STATUS (cause #100, call state U12)		U12	5.5.6.7.2
			unrecognized information element & call state # U0	STATUS (cause #99, call state U12)	release all resources	U12	5.5.6.8.1 5.5.6.12 b
			unrecognized information element & call state = U0	STATUS (cause #99, call state U0) Release_conf	release all resources	U0	5.5.6.8.1 5.5.6.12
			non-mandatory information element with invalid content & call state # U0	STATUS (cause #100, call state U12)		U12	5.5.6.8.2 5.5.6.12 b
			non-mandatory information element with invalid content & call state = U0	STATUS (cause #100, call state U0) Release_conf	release all resources	U0	5.5.6.8.2 5.5.6.12 c
			call state # U0			U12	5.5.6.12 b
			call state = U0	Release_conf	release all resources	U0	5.5.6.12 c

U0 - Null
U1 - Call Initiated
U3 - Outgoing Call Proceeding
U6 - Call Present
U8 - Connect Request
U9 - Incoming Call Proceeding
U10 - Active
U11 - Realease Request
U12 - Release Indication

State	Input	Type	Condition	Output	Action	Next State	Ref.
* except U0, U11 U12	Release_req			RELEASE (cause #16)	stop all timers start T308, disconnect VC	U11	5.5.4.3
* except U0, U1, U11, U12	RELEASE protocol discriminator (M) call reference (M) message type (M) message length (M) cause (M)		cause missing	(cause #96 in U12 RELEASE_COMPLETE) Release_ind (cause #31)	stop T322, release all resources	U12	5.5.6.7.1
			invalid content of cause	(cause #100 in U12 RELEASE_COMPLETE) Release_ind (cause #31)	stop T322, release all resources	U12	5.5.6.7.2
			unrecognized information element	(cause #99 in U12 RELEASE_COMPLETE) Release_ind (cause #31)	release all resources	U12	5.5.6.8.1 a
			otherwise (o.k.)	Release_ind	stop T322, release VC	U12	5.5.4.4.
* except U0, U1, U11, U12	RELEASE_COMPLETE protocol discriminator (M) call reference (M) message type (M) message length (M) cause (O[6])	U	mandatory cause missing or invalid cause	Release_conf (cause #31)	release all resources	U0	5.5.6.7.1 5.5.6.7.2
			otherwise (o.k. or unrecognized element)	Release_conf	release all resources	U0	5.5.6.8.1 b
* except U0	INITIATE_STATUS_ENQUIRY			STATUS_ENQUIRY	if T322 not active then start T322	-	5.5.6.11
	Timeout T322		1..n expiry [7]	STATUS_ENQUIRY	start T322	-	5.5.6.11
			n+1 expiry	RELEASE_COMPLETE (cause #41) Release_conf	stop T322 release all resources	U0	5.5.6.11

[6] Mandatory in the first call clearing message; including when the RELEASE_COMPELTE message is sent as a result of an error condition.

[7] The number of times the STATUS_ENQUIRY is retransmitted is an implementation dependent value.

State	Input	Type	Condition	Output	Action	Next State	Ref.
* except U0, U11, U12	STATUS *Note 1* protocol discriminator (M) call reference (M) message type (M) message length (M) call state (M) cause (M) endpoint reference (O) endpoint state (O)		mandatory information element missing	STATUS (current state, cause #96)		-	5.5.6.7.1
			mandatory information element error	STATUS (current state, cause #100)		-	5.5.6.7.2
			unrecognized information element & call state = U0	STATUS (cause #99, call state U0) Release_conf	release all resources	U0	5.5.6.8.1 5.5.6.12 a)
			unrecognized information element & incompatible state *Note 4*	STATUS (cause #99, call state U11) RELEASE (cause #101)	start T308	U11	5.5.6.8.1 5.5.6.12 a)
			unrecognized information element & (cause = 96, 97, 99, 100 or 101)	STATUS (cause #99, call state U11) RELEASE (received cause)	start T308	U11	5.5.6.8.1 5.5.6.12
			unrecognized information element (compatible & cause =30)	STATUS (current state, cause #99)	stop T322	-	5.5.6.8.1 5.5.6.12
			non-mandatory information element with invalid content & call state = U0	STATUS (cause #100, call state U0) Release_conf	release all resources	U0	5.5.6.8.2 5.5.6.12 a)
			non-mandatory information element with invalid content & incompatible state *Note 4*	STATUS (cause #100, call state U11) RELEASE (cause #101)	start T308	U11	5.5.6.8.2 5.5.6.12

U0 - Null
U1 - Call Initiated
U3 - Outgoing Call Proceeding
U6 - Call Present
U8 - Connect Request
U9 - Incoming Call Proceeding
U10 - Active
U11 - Realease Request
U12 - Release Indication

State	Input	Type	Condition	Output	Action	Next State	Ref.
			invalid content of non-mandatory information element & (cause = 96, 97, 99, 100 or 101)	STATUS (cause #100, call state U11) RELEASE (received cause)	start T308	U11	5.5.6.8.2 5.5.6.12
			non-mandatory information element with invalid content (compatible state & cause =30)	STATUS (current state, cause #100)	stop T322	-	5.5.6.8.2 5.5.6.12
			call state = U0	Release_conf	release all resources	U0	5.5.6.12 o
			incompatible state *Note 4*	RELEASE (cause #101)	start T308	U11	5.5.6.12
			compatible state & (cause = 96, 97, 99, 100 or 101) *Note 2*	RELEASE (received cause)	start T308	U11	5.5.6.12
			otherwise (compatible state & cause #30)		stop T322	-	5.5.6.11

U0 - Null
U1 - Call Initiated
U3 - Outgoing Call Proceeding
U6 - Call Present
U8 - Connect Request
U9 - Incoming Call Proceeding
U10 - Active
U11 - Realease Request
U12 - Release Indication

State	Input	Type	Condition	Output	Action	Next State	Ref.
* except U0	STATUS_ENQUIRY protocol discriminator (M) call reference (M) message type (M) message length (M) endpoint reference (O)		mandatory information element missing	STATUS (current state, cause #96)		-	5.5.6.7.1
			mandatory information element with invalid content	STATUS (current state, cause #100)		-	5.5.6.7.2
			unrecognized information element	STATUS (current state, cause #99) STATUS (current state, cause #30)		-	5.5.6.8.1 5.5.6.11
			non-mandatory information element with invalid content	STATUS (current state, cause #100) STATUS (current state, cause #30)		-	5.5.6.8.2 5.5.6.11
			otherwise (o.k.)	STATUS (current state, cause #30)		-	5.5.6.11
* except U0	any unexpected message except RELEASE or RELEASE_COMPLETE	U		STATUS (cause #97 or 101)		-	5.5.6.4

U0 - Null
U1 - Call Initiated
U3 - Outgoing Call Proceeding
U6 - Call Present
U8 - Connect Request
U9 - Incoming Call Proceeding
U10 - Active
U11 - Realease Request
U12 - Release Indication

Commentaires généraux:

1. On utilise les notations suivantes:
CR (call reference) - référence d'appel;
VC (virtual channel) - canal virtuel;
VCI (virtual channel identifier) - identificateur du canal virtuel;
VPCI (virtual path connection identifier) - identificateur de la voie de la connexion;
M (mandatory information element) - élément d'information obligatoire;
O (optional information element) - élément d'information optionnel;
* designe "tous les états";
U - le type d'un message inattendu.

2. Les termes utilisés:
- un *canal virtuel* est *connecté* quand le canal fait partie d'une connexion ATM;
- un *canal virtuel* est *déconnecté* quand le canal ne fait plus partie d'une connexion ATM, mais il n'est pas encore disponible pour une l'utilisation dans une nouvelle connexion ATM;
- un *canal virtuel* est *relâché* quand il ne fait plus partie d'une connexion ATM et est disponible pour une nouvelle connexion.
De façon similaire, une référence d'appel est relâchée si elle est disponible pour la réutilisation.

3. En accord avec les sections 5.5.6.8.1. et 5.5.6.8.2., l'envoi d'un message STATUS à la réception d'un message contenant un ou des élément(s) d'information inconnu(s) ou dont le contenu est invalide, est optionnelle. Si on choisit à ne pas envoyer un tel message, la partie ombragée du tableau sera ignorée.

4. On suppose que toutes les messages reçus:
- ont comme discriminateur de protocole le code de Q2931;
- contient le disciminateur de protocole, la reference d'appel, le type du message et la longueur (i.e. la longueur du message est plus grande ou égale à 9 octets);
- la référence d'appel a le premier octet égal à 00000011;
- aucun message n'utilise la reference d'appel globale.
Cette fonction de filtrage est réalisée par ce qu'on appelle un coordinateur.

Notes apparaissant dans le tableau:

Note 1
En accord avec la section 5.5.6.12 , à la réception d'un message STATUS rapportant un état incompatible, l'entité réceptrice doit:
a) soit fermer la connexion en envoyant les messages appropriés
b) soit prendre toute autre action qui essaie de récupérer l'erreur et qui est dépendante de l'implantation.
Dans ce document on a choisi a).

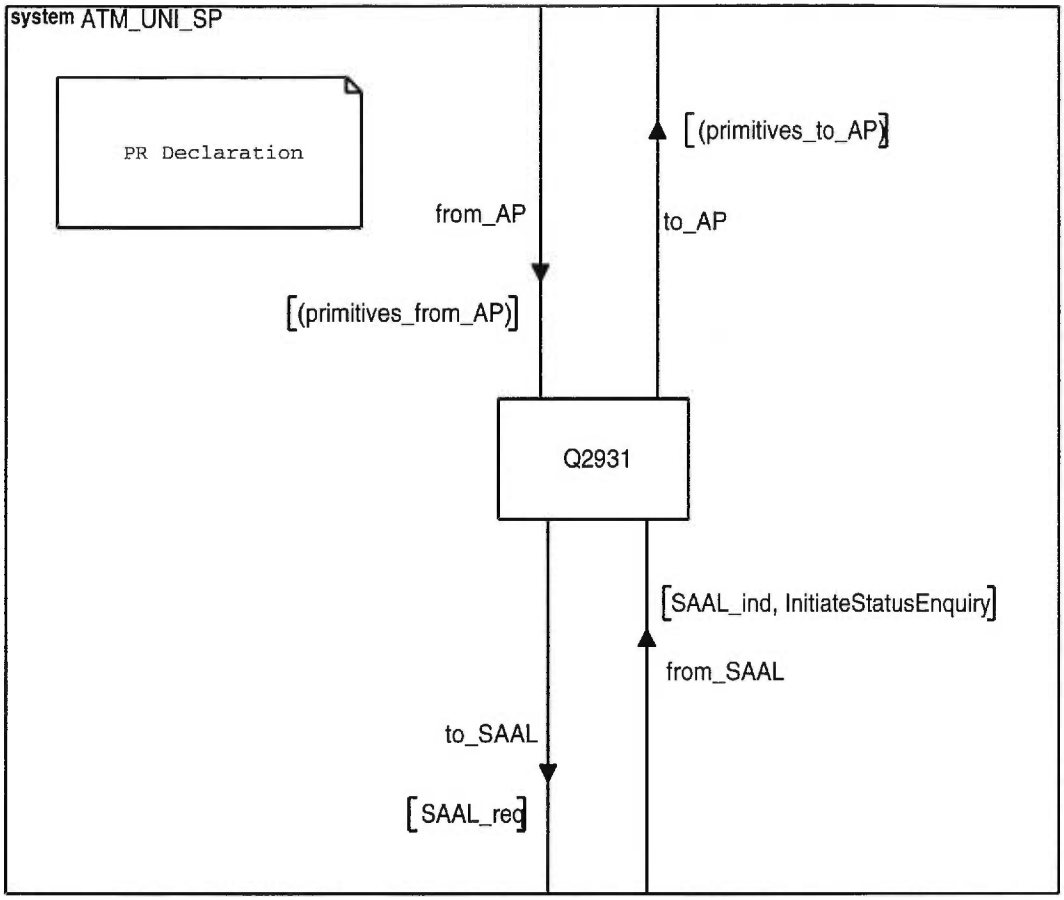
Note 2
Les actions à prendre sont une option de l'implantation. Si aucune autre procédure n'est pas définie, le récepteur doit fermer l'appel avec la procédure appropriée.

Note 3
L'équipement doit réaliser une récupération qui est dépendante de l'implantation.

Note 4
La décision des états qui sont compatibles est dépendante de l'implantation.

Annexe 2

Spécification SDL/GR du protocole de signalisation


system ATM_UNI_SP
PR Declaration
[(primitives_to_AP)]
from_AP
to_AP
[(primitives_from_AP)]
Q2931
[SAAL_ind, InitiateStatusEnquiry]
from_SAAL
to_SAAL
[SAAL_req]

```
SYNTYPE max_len_mess = NATURAL
  CONSTANTS 0:328
ENDSYNTYPE;

NEWTYPE hexa_array_type
  Array(max_len_mess, hexadecimal);
ENDNEWTYPE;

SYNTYPE max_len_arr = NATURAL
  CONSTANTS 0:25
ENDSYNTYPE;

NEWTYPE natural_array_type
  Array(max_len_arr, Natural);
ENDNEWTYPE;

NEWTYPE occ_array_type STRUCT
 nb_occ max_len_arr;
 occ natural_array_type;
ENDNEWTYPE;

SYNTYPE flag_type = Natural
  CONSTANTS 0 : 1
ENDSYNTYPE;
SYNTYPE int_2 = Natural
  CONSTANTS 0 : 3
ENDSYNTYPE;
SYNTYPE int_3 = Natural
  CONSTANTS 0 : 7
ENDSYNTYPE;
SYNTYPE int_4 = Natural
  CONSTANTS 0 : 15
ENDSYNTYPE;
SYNTYPE int_5 = Natural
  CONSTANTS 0 : 31
ENDSYNTYPE;
SYNTYPE int_6 = Natural
  CONSTANTS 0 : 63
ENDSYNTYPE;
SYNTYPE int_7 = Natural
  CONSTANTS 0 : 127
ENDSYNTYPE;
SYNTYPE octet = Natural
  CONSTANTS 0 : 255
ENDSYNTYPE octet;
SYNTYPE int_23 = Natural
  CONSTANTS 0 : 8388607
ENDSYNTYPE;

SYNONYM dummy_cr Natural = 0x7fffff;

NEWTYPE optional_value_type STRUCT
    presence Boolean;
    value Natural;
ENDNEWTYPE;


/*_____________________________________________________

          Protocol discriminator declaration
_____________________________________________________*/

SYNTYPE protocol_discriminator_type = Natural
  CONSTANTS 9 /*  Q2931 */
ENDSYNTYPE;
```

```
/*_____________________________________________________

            Call reference declaration
_____________________________________________________*/

NEWTYPE call_reference_type STRUCT
    flag flag_type;
    value int_23;
ENDNEWTYPE;

/*_____________________________________________________

        Messages' synonyms decaration
_____________________________________________________*/

SYNONYM SETUP Natural = 5;
SYNONYM CALL_PROCEEDING Natural = 2;
SYNONYM CONNEC Natural = 7;
SYNONYM CONNECT_ACK Natural = 15;
SYNONYM RELEASE Natural = 77;
SYNONYM RELEASE_COMPLETE Natural = 90;
SYNONYM STATUS Natural = 125;
SYNONYM STATUS_ENQUIRY Natural = 117;


/*_____________________________________________________

      Information elements delaration
_____________________________________________________*/

NEWTYPE MID_range_type STRUCT
    presence Boolean;
    lowest_value Natural; /* 2 octets */
    highest_value Natural; /* 2 octets */
ENDNEWTYPE;

NEWTYPE AAL_param_type STRUCT
    presence Boolean;
    coding_standard int_2;
/*SELECT*/
    AAL_type Natural;
/*(1):*/
    subtype optional_value_type;
    rate optional_value_type;
    multiplier optional_value_type;
    scfrm optional_value_type;
    error_correction optional_value_type;
    data_blocksize optional_value_type;
    partially_filled_cells optional_value_type;
/*(3,5):*/
    forward_max_size optional_value_type;
    backward_max_size optional_value_type;
    MID_range MID_range_type;
    SSCS_type optional_value_type;
/*(16):*/
    user_def_AAL_inf occ_array_type;
ENDNEWTYPE;


/*_____________________________________________________*/
NEWTYPE tagging_type STRUCT
    presence Boolean;
    backward flag_type;
    forward flag_type;
ENDNEWTYPE;
```

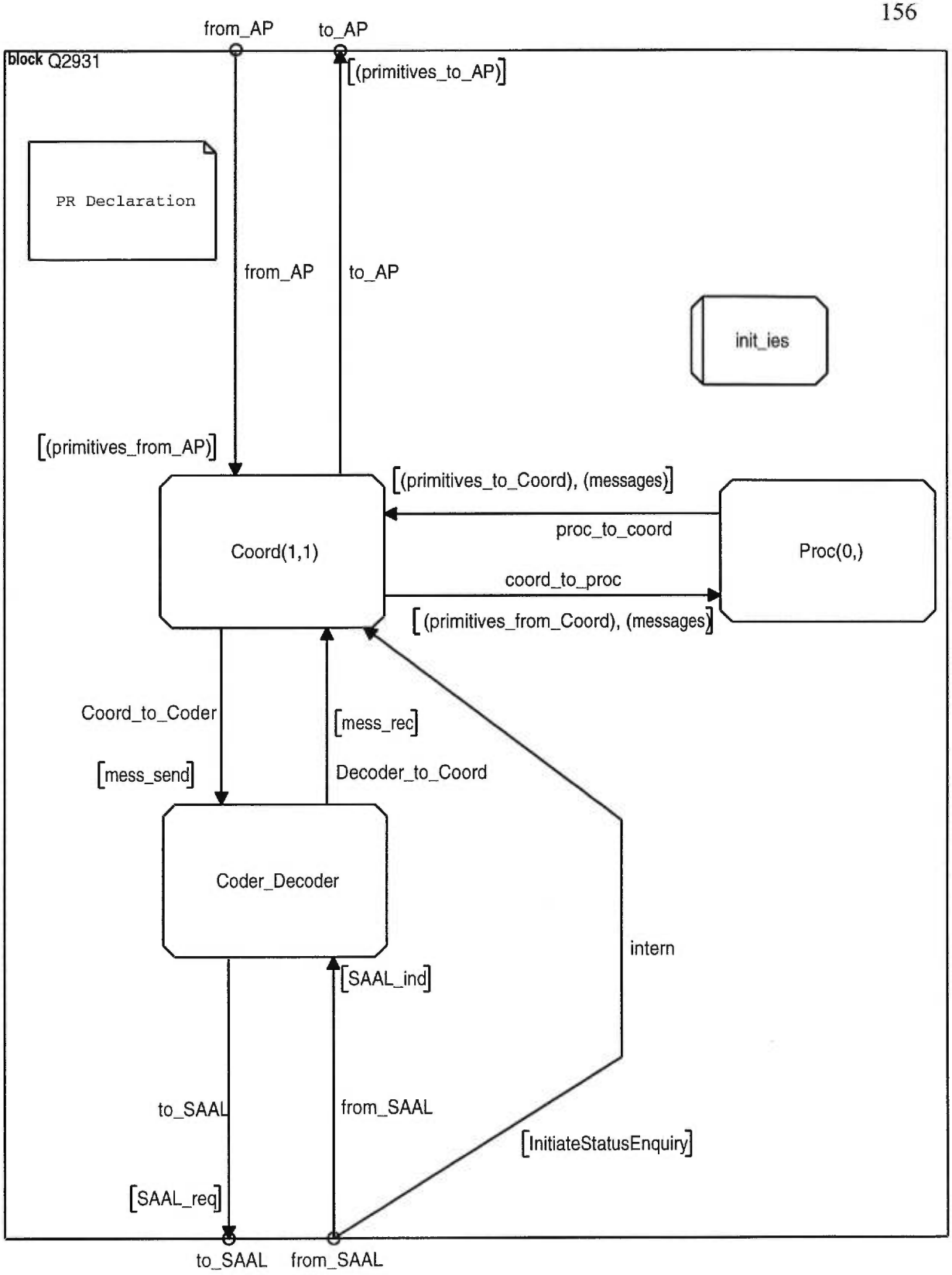
```
NEWTYPE ATM_traffic_desc_type STRUCT
    presence Boolean;
    coding_standard int_2;
    fpcr_0 optional_value_type;
    bpcr_0 optional_value_type;
    fpcr_0_1 optional_value_type;
    bpcr_0_1 optional_value_type;
    fscr_0 optional_value_type;
    bscr_0 optional_value_type;
    fscr_0_1 optional_value_type;
    bscr_0_1 optional_value_type;
    fmbs_0 optional_value_type;
    bmbs_0 optional_value_type;
    fmbs_0_1 optional_value_type;
    bmbs_0_1 optional_value_type;
    best_effort Boolean;
    tagging tagging_type;
ENDNEWTYPE;

/*_____________________________________________________________________*/
NEWTYPE class_options_type STRUCT
    presence Boolean;
    traffic_type int_3;
    timing_req int_2;
ENDNEWTYPE;

NEWTYPE B_BC_type STRUCT
    presence Boolean;
    coding_standard int_2;
    class natural;
    class_options class_options_type;
    clipping int_2;
    user_plane int_2;
ENDNEWTYPE;


/*_____________________________________________________________________*/
NEWTYPE B_HLI_type STRUCT
    presence Boolean;
    coding_standard int_2;
    hl_type int_7;
    hl_info occ_array_type;
ENDNEWTYPE;

/*_____________________________________________________________________*/
NEWTYPE codings_type STRUCT
    presence Boolean;
    mode int_2;
    q933 int_2;
ENDNEWTYPE;

NEWTYPE layer_2_type STRUCT
    presence Boolean;
/*SELECT*/
    value int_5;
/*(6,7,9,10,11,14,17):*/
    codings codings_type;
    window_size optional_value_type;
/* (16):*/
    user_spec optional_value_type;
ENDNEWTYPE;

NEWTYPE layer_3_type STRUCT
    presence Boolean;
/*SELECT*/
    value int_5;
```

```
/*(6,7,8):*/
    mode optional_value_type;
    default_packet_size optional_value_type;
    packet_window_size optional_value_type;
/* (16):*/
    user_spec optional_value_type;
/*(11):*/
    IPI optional_value_type;
    SNAP_id occ_array_type;
ENDNEWTYPE;

NEWTYPE B_LLI_IE STRUCT
    coding_standard int_2;
    info_layer_1 optional_value_type;
    info_layer_2 layer_2_type;
    info_layer_3 layer_3_type;
ENDNEWTYPE;

SYNTYPE max_occ_B_LLI = Natural
  CONSTANTS 0 : 3
ENDSYNTYPE;

NEWTYPE B_LLI_table
  Array( max_occ_B_LLI, B_LLI_IE);
ENDNEWTYPE;

NEWTYPE B_LLI_type STRUCT
    nb_occ max_occ_B_LLI;
    occ B_LLI_table;
ENDNEWTYPE;

/*_____________________________________________________________________*/
NEWTYPE called_nb_type STRUCT
    presence Boolean;
    coding_standard int_2;
    nb_type int_3;
    plan_id int_4;
    address occ_array_type;
ENDNEWTYPE;

/*_____________________________________________________________________*/
NEWTYPE subaddress_type /* la meme structure pour calling et called */
 STRUCT
    presence Boolean;
    coding_standard int_2;
    nb_type int_3;
    subaddress occ_array_type;
ENDNEWTYPE;


/*_____________________________________________________________________*/
NEWTYPE option_type STRUCT
    presence Boolean;
    presentation int_2;
    screening int_2;
ENDNEWTYPE;

NEWTYPE calling_nb_type STRUCT
    presence Boolean;
    coding_standard int_2;
    nb_type int_3;
    option option_type;
    plan_id int_4;
    address occ_array_type;
ENDNEWTYPE;
```

```
/*_____________________________________________________________*/
/*
NEWTYPE transit_option_type STRUCT
    presence Boolean;
    network transit_network_selection_type;
ENDNEWTYPE;
*/
NEWTYPE cause_IE STRUCT
    coding_standard int_2;
    location int_4;
/*SELECT*/
    value Natural;
/*(1,3,49):*/
    P_U flag_type;
    N_A flag_type;
    condition_2 int_2;
/*(21):*/
    reason int_5;
    condition_3 int_2;
    diag_info occ_array_type;
/*(22):*/
    called_nb called_nb_type;
    transit_option transit_network_selection_type;
/*(88):*/
    ie_ident octet;
/*(43,96,99,100):*/
    IE_id occ_array_type;
/*(37):*/
    subfield occ_array_type;
/*(82):*/
    VPCI Natural;
    VCI Natural;
/*(97,101):*/
    message_type octet;
/*(102):*/
    char1 Character;
    char2 Character;
    char3 Character;
/* ou rien */
ENDNEWTYPE;

SYNTYPE max_occ_cause = Natural
  CONSTANTS 0: 2
ENDSYNTYPE;

NEWTYPE cause_table
  Array( max_occ_cause, cause_IE)
ENDNEWTYPE;

NEWTYPE cause_type STRUCT
    nb_occ max_occ_cause;
    occ cause_table;
ENDNEWTYPE;


/*_____________________________________________________________*/
NEWTYPE connection_id_type STRUCT
    presence Boolean;
    coding_standard int_2;
    VPAS int_2;
    pref_excl int_3;
    VPCI Natural;
    VCI Natural;
ENDNEWTYPE;
```

```
/*_____________________________________________________________________*/
NEWTYPE Qos_type STRUCT
    presence Boolean;
    coding_standard int_2;
    class_forward octet;
    class_backward octet;
ENDNEWTYPE;


/*_____________________________________________________________________*/
NEWTYPE B_repeat_indic_type STRUCT
    presence Boolean;
    coding_standard int_2;
    value int_4;
ENDNEWTYPE;


/*_____________________________________________________________________*/
NEWTYPE B_sending_complete_type STRUCT
    presence Boolean;
    coding_standard int_2;
    indication int_7;
ENDNEWTYPE;


/*_____________________________________________________________________*/
NEWTYPE transit_network_selection_type STRUCT
    presence Boolean;
    coding_standard int_2;
    network_type int_3;
    plan int_4;
    id occ_array_type;
ENDNEWTYPE;


/*_____________________________________________________________________*/
NEWTYPE call_state_type STRUCT
    presence Boolean;
    coding_standard int_2;
    code int_6;
ENDNEWTYPE;


/*_____________________________________________________________________

       primitives parameters type declaration
_____________________________________________________________________*/

NEWTYPE setup_struct STRUCT
    AAL_param AAL_param_type;
    ATM_traffic_desc ATM_traffic_desc_type;
    B_BC B_BC_type;
    B_HLI B_HLI_type;
    B_LLI B_LLI_type;
    called_subaddress subaddress_type;
    called_nb called_nb_type;
    calling_subaddress subaddress_type;
    calling_nb calling_nb_type;
    connection_id connection_id_type;
    QoS Qos_type;
    B_sending_complete B_sending_complete_type;
    transit_network_selection transit_network_selection_type;
ENDNEWTYPE;
```

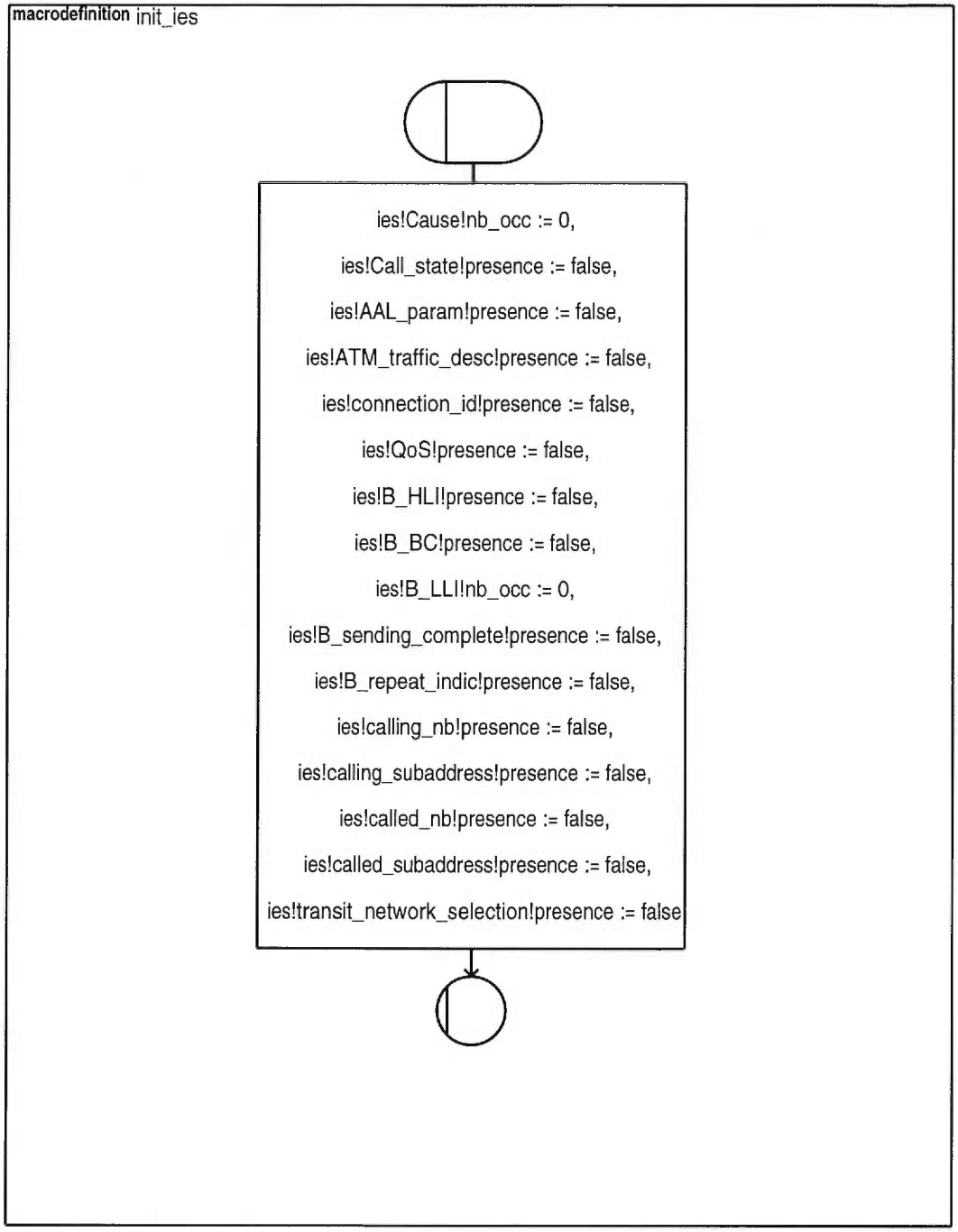
```
NEWTYPE call_pr_struct STRUCT
    connection_id connection_id_type;
ENDNEWTYPE;

NEWTYPE connect_struct STRUCT
    AAL_param AAL_param_type;
    B_LLI B_LLI_type;
    connection_id connection_id_type;
ENDNEWTYPE;

NEWTYPE release_struct STRUCT
    cause cause_type;
ENDNEWTYPE;

NEWTYPE status_struct STRUCT
    call_state call_state_type;
    cause cause_type;
ENDNEWTYPE;


/*_____________________________________________________________

            Implamenatation choice:
sending of a status message annoncing error(s) in the
optional information elements
_____________________________________________________________*/

SYNONYM send_st Boolean = true;


/*_____________________________________________________________

                  Signal declarations
_____________________________________________________________*/

SIGNAL
  SAAL_ind(hexa_array_type, natural),
  SAAL_req(hexa_array_type);
SIGNAL
  Send_cr(call_reference_type),
  Setup_ind(call_reference_type, setup_struct),
  Proceeding_ind(call_reference_type, call_pr_struct),
  Setup_conf(call_reference_type, connect_struct),
  Setup_complete_ind(call_reference_type),
  Release_ind(call_reference_type, release_struct),
  Release_conf(call_reference_type, release_struct);
SIGNAL
  Setup_req(setup_struct),
  Proceeding_req(call_reference_type, call_pr_struct),
  Setup_resp(call_reference_type, connect_struct),
  Release_req(call_reference_type),
  Release_resp(call_reference_type, release_struct),
  InitiateStatusEnquiry(call_reference_type);

SIGNALLIST primitives_from_AP =
  Setup_req, Proceeding_req, Setup_resp, Release_req, Release_resp;
SIGNALLIST primitives_to_AP =
  Setup_ind, Proceeding_ind, Setup_conf, Setup_complete_ind, Release_ind,
  Release_conf, Send_cr;
```

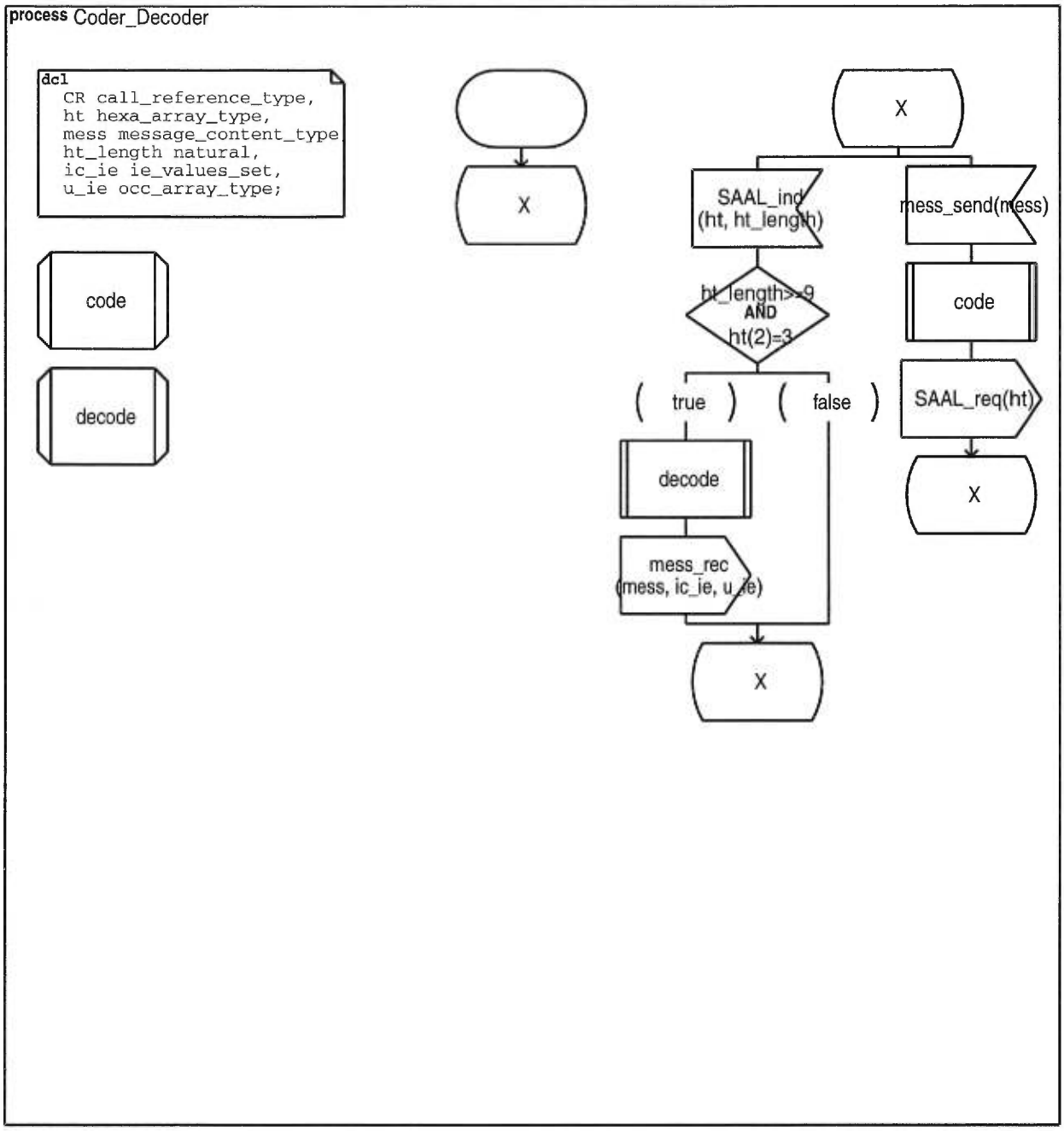


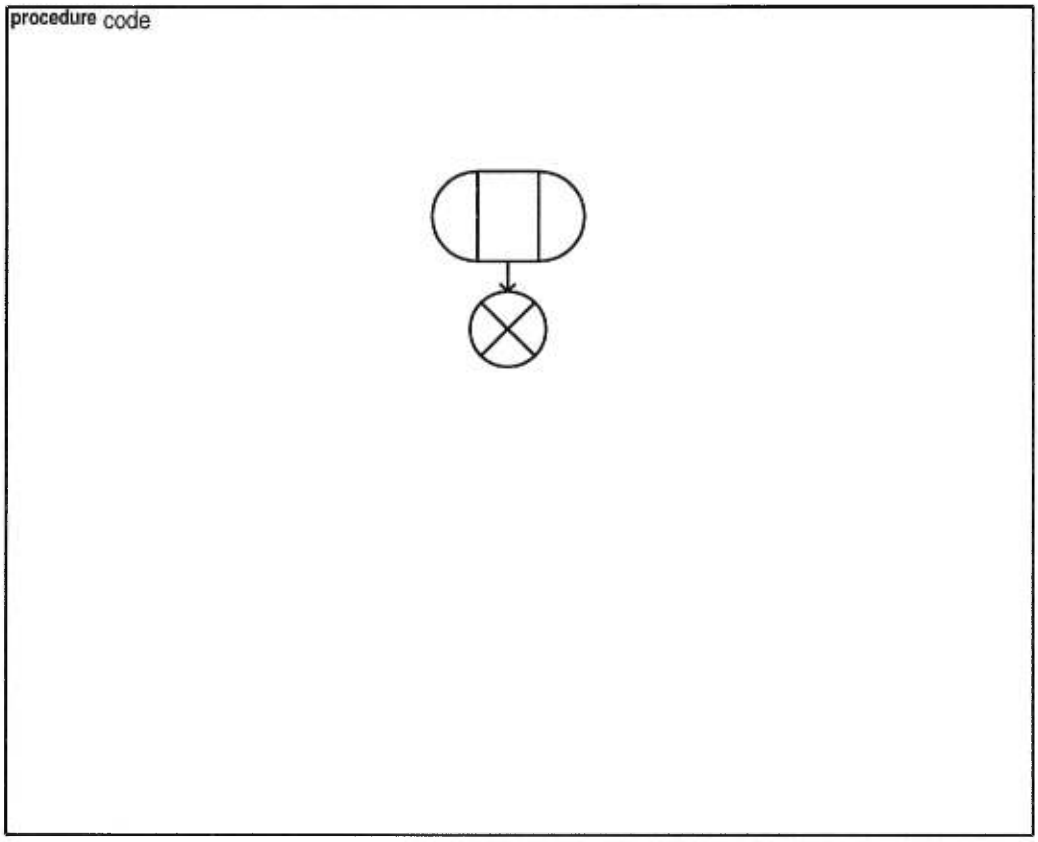
Author: Roxana Marcoci	File name: atm_sign.pr	Q2931, page: 1

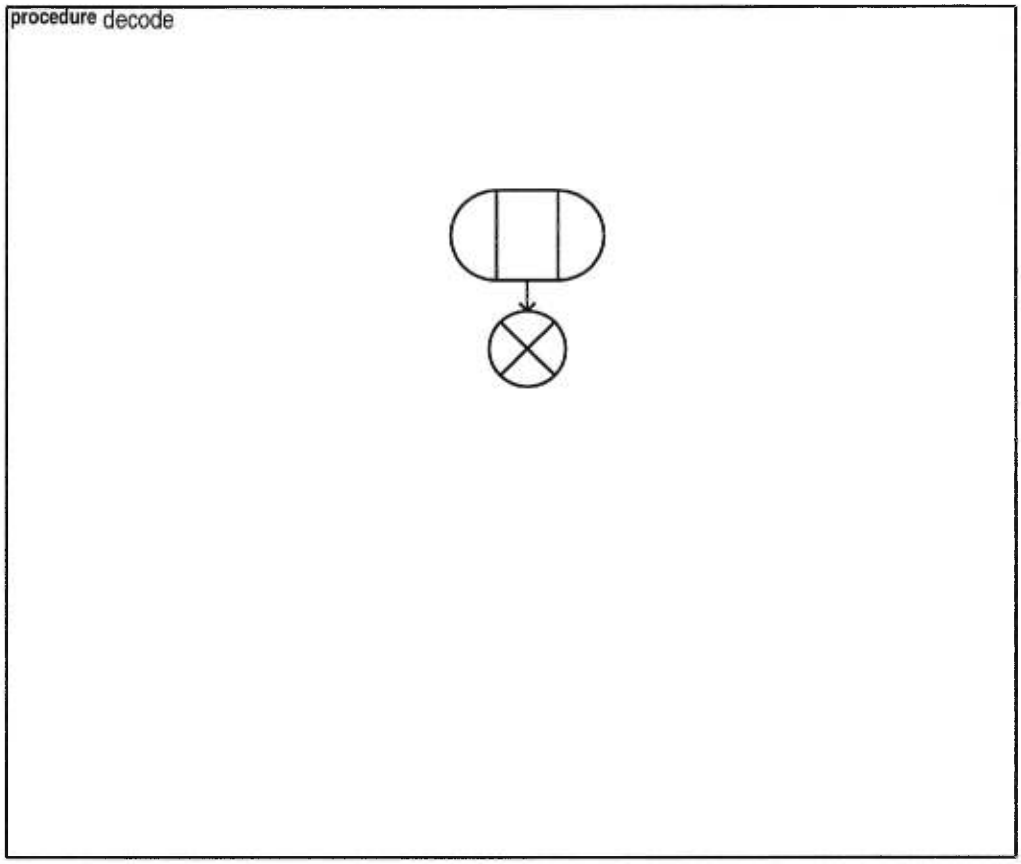
```
/*_____________________________________________________________

       Types de donnes des parametres des signaux
_____________________________________________________________*/

NEWTYPE ie_type STRUCT
    cause cause_type;
    call_state call_state_type;
    AAL_param AAL_param_type;
    ATM_traffic_desc ATM_traffic_desc_type;
    connection_id connection_id_type;
    QoS Qos_type;
    B_HLI B_HLI_type;
    B_BC B_BC_type;
    B_LLI B_LLI_type;
    B_sending_complete B_sending_complete_type;
    B_repeat_indic B_repeat_indic_type;
    calling_nb calling_nb_type;
    calling_subaddress subaddress_type;
    called_nb called_nb_type;
    called_subaddress subaddress_type;
    transit_network_selection transit_network_selection_type;
ENDNEWTYPE;

NEWTYPE message_content_type STRUCT
    pr_disc protocol_discriminator_type;
    CR call_reference_type; /* size 4 bytes */
    message_type octet; /* size 2 bytes */
    ie ie_type;
ENDNEWTYPE;

SYNTYPE ie_values_type = NATURAL
  CONSTANTS 8:120
ENDSYNTYPE;

NEWTYPE ie_values_set
  POWERSET (ie_values_type)
ENDNEWTYPE;

NEWTYPE ie_id
  LITERALS ie8,ie20,ie88,ie89,ie90,ie92,ie93,ie94,ie95,ie98,ie99,ie108,ie109,ie112,ie113,ie120
  OPERATORS ORDERING;
ENDNEWTYPE;

NEWTYPE ie_array_type
  ARRAY(ie_id,boolean);
ENDNEWTYPE;

/*_____________________________________________________________

            La duree des timers
_____________________________________________________________ */
SYNONYM d_T303 Natural = 4;
SYNONYM d_T310 Natural = 30;
SYNONYM d_T308 Natural = 30;
SYNONYM d_T313 Natural = 4;
SYNONYM d_T322 Natural = 4;

/*_____________________________________________________________

       Les siganux vehicules au niveau bloc
_____________________________________________________________ */
SIGNAL
  mess_rec(message_content_type,ie_values_set, occ_array_type),
  mess_send(message_content_type);
```

```
SIGNAL
  SETUP(call_reference_type, ie_type, ie_values_set, occ_array_type),
  CALL_PROCEEDING(call_reference_type, ie_type, ie_values_set, occ_array_type),
  CONNEC(call_reference_type, ie_type, ie_values_set, occ_array_type),
  CONNECT_ACK(call_reference_type, ie_type, ie_values_set, occ_array_type),
  RELEASE(call_reference_type, ie_type, ie_values_set, occ_array_type),
  RELEASE_COMPLETE(call_reference_type, ie_type, ie_values_set, occ_array_type),
  STATUS(call_reference_type, ie_type, ie_values_set, occ_array_type),
  STATUS_ENQUIRY(call_reference_type, ie_type, ie_values_set, occ_array_type),
  UNRECOGNIZED(call_reference_type, octet),
  Connection_req(call_reference_type, setup_struct);

SIGNALLIST messages =
  SETUP, CALL_PROCEEDING, CONNEC, CONNECT_ACK, RELEASE, RELEASE_COMPLETE,
  STATUS, STATUS_ENQUIRY, UNRECOGNIZED;
SIGNALLIST primitives_from_Coord =
  Connection_req, Proceeding_req, Setup_resp, Release_req, Release_resp, InitiateStatusEnquiry;
SIGNALLIST primitives_to_Coord =
  Setup_ind, Proceeding_ind, Setup_conf, Setup_complete_ind, Release_ind,
  Release_conf;
```

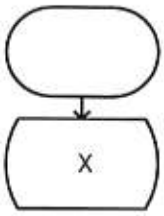

macrodefinition init_ies
ies!Cause!nb_occ := 0,
ies!Call_state!presence := false,
ies!AAL_param!presence := false,
ies!ATM_traffic_desc!presence := false,
ies!connection_id!presence := false,
ies!QoS!presence := false,
ies!B_HLI!presence := false,
ies!B_BC!presence := false,
ies!B_LLI!nb_occ := 0,
ies!B_sending_complete!presence := false,
ies!B_repeat_indic!presence := false,
ies!calling_nb!presence := false,
ies!calling_subaddress!presence := false,
ies!called_nb!presence := false,
ies!called_subaddress!presence := false,
ies!transit_network_selection!presence := false

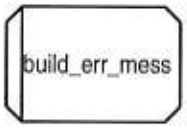

process Coder_Decoder
dcl
CR call_reference_type,
ht hexa_array_type,
mess message_content_type,
ht_length natural,
ic_ie ie_values_set,
u_ie occ_array_type;
code
decode
X
X
SAAL_ind
(ht, ht_length)
ht_length>=9
AND
ht(2)=3
true
false
decode
mess_rec
(mess, ic_ie, u_ie)
X
X
mess_send(mess)
code
SAAL_req(ht)
X

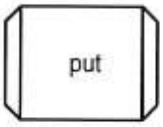

procedure code

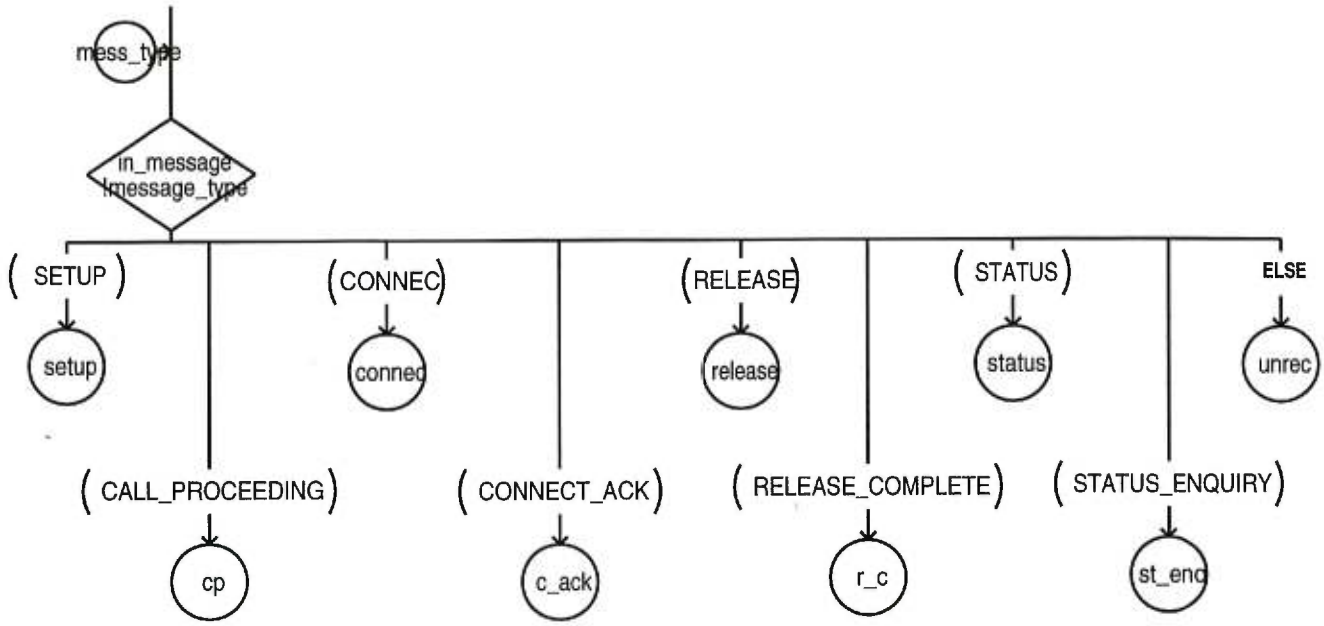

procedure decode

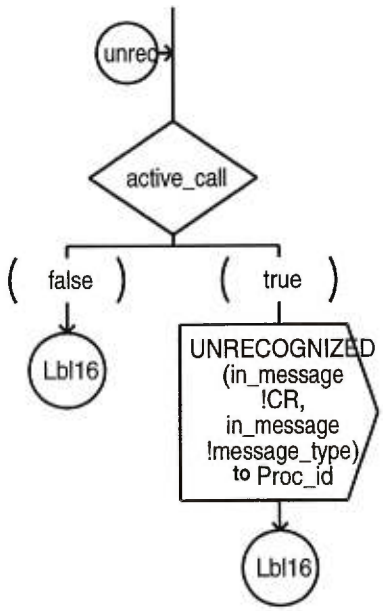
```
dcl
   in_message,
   out_message message_content_type;
dcl
   ies ie_type;
dcl
   s setup_struct,
   cp call_pr_struct,
   c connect_struct,
   r release_struct;
dcl
   CR call_reference_type;
dcl
   Proc_id PId;
dcl
   more_connections boolean := true,
   active_call Boolean;
dcl
   ic_ie ie_values_set,
   u_ie occ_array_type;

dcl last_cr natural:=0;

dcl cv natural;
```

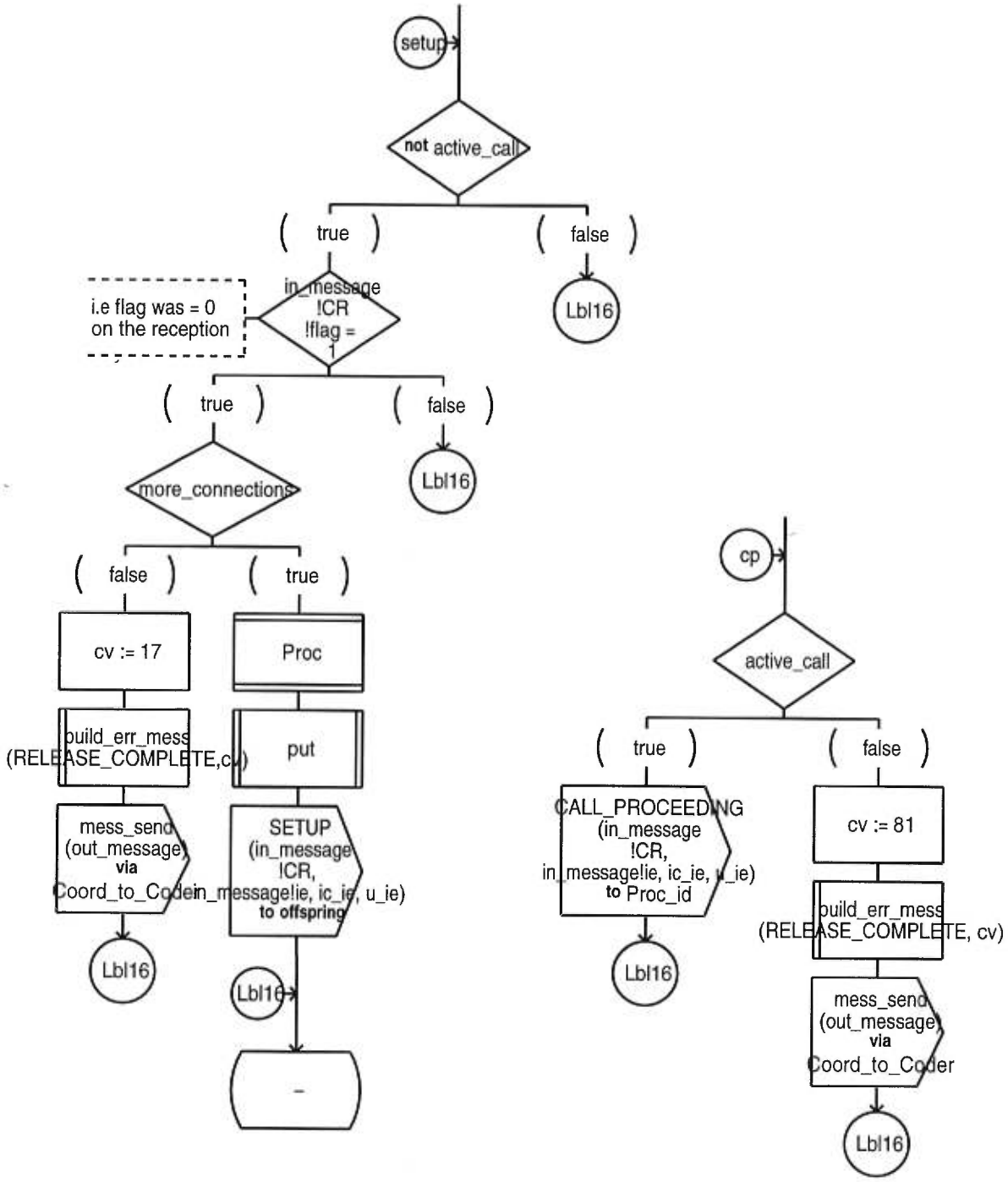


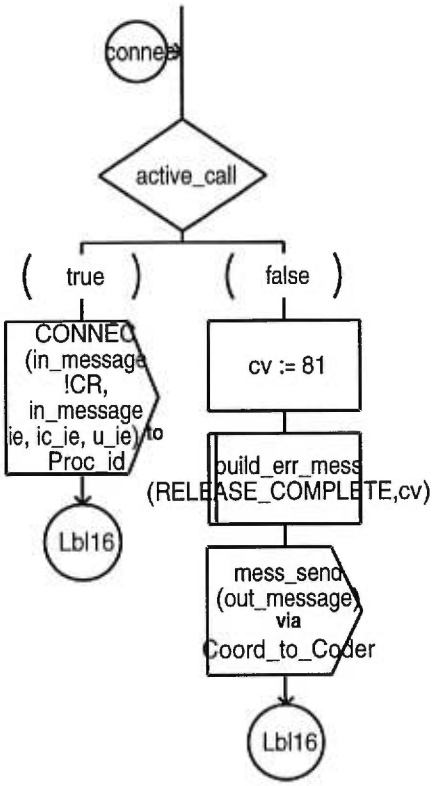


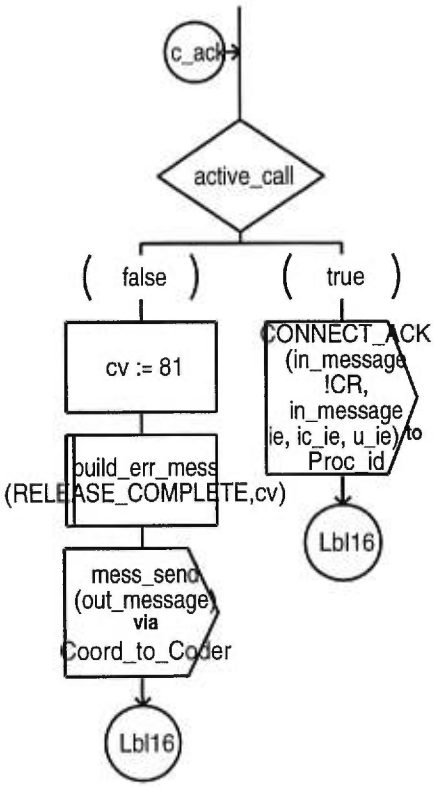


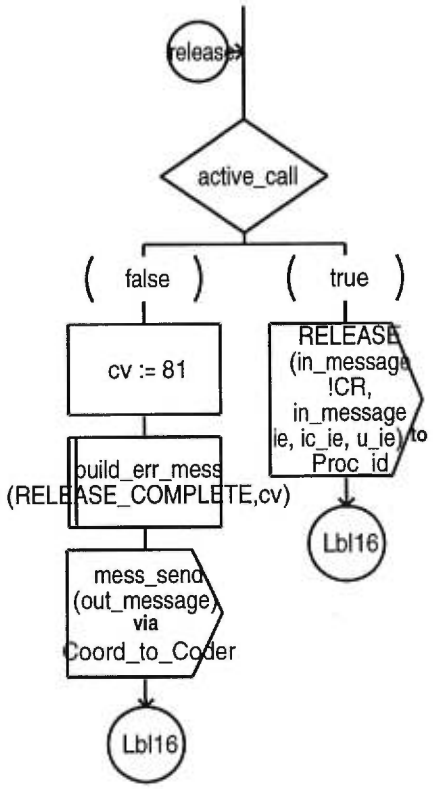

mess_type
in_message
!message_type
SETUP
setup
CONNEC
connec
RELEASE
release
STATUS
status
ELSE
unrec
CALL_PROCEEDING
cp
CONNECT_ACK
c_ack
RELEASE_COMPLETE
r_c
STATUS_ENQUIRY
st_enq

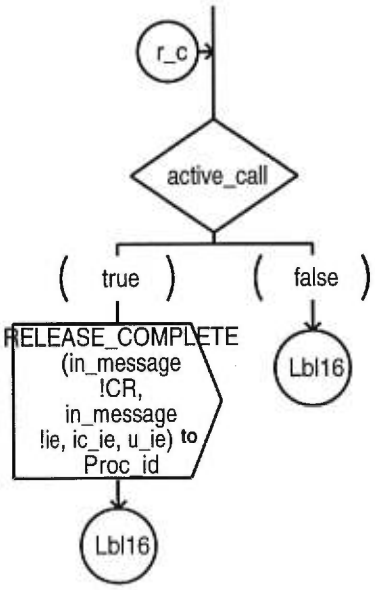

unrec
active_call
false
Lbl16
true
UNRECOGNIZED
(in_message
!CR,
in_message
!message_type)
to Proc_id
Lbl16

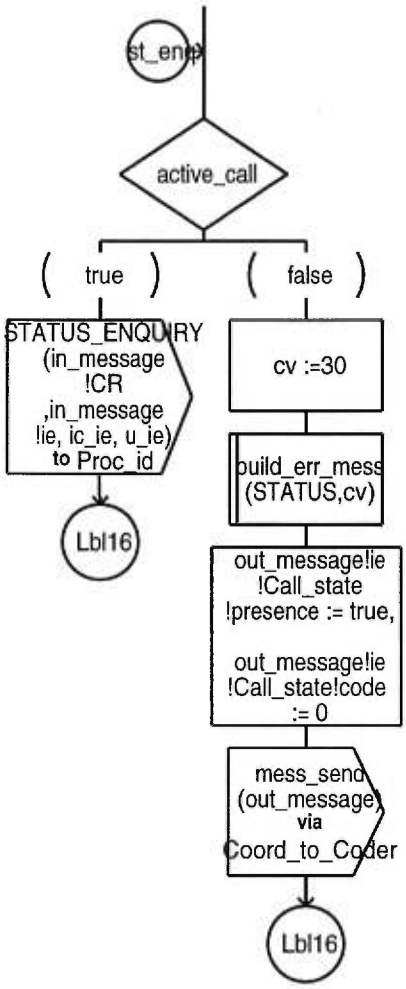

setup
not active_call
true
false
in_message !CR !flag = 1
i.e flag was = 0 on the reception
Lbl16
true
false
Lbl16
more_connections
false
true
cv := 17
Proc
build_err_mess (RELEASE_COMPLETE,cv)
put
mess_send (out_message) via Coord_to_Coder
SETUP (in_message !CR, in_message!ie, ic_ie, u_ie) to offspring
Lbl16
Lbl16
cp
active_call
true
false
CALL_PROCEEDING (in_message !CR, in_message!ie, ic_ie, u_ie) to Proc_id
cv := 81
build_err_mess (RELEASE_COMPLETE, cv)
Lbl16
mess_send (out_message) via Coord_to_Coder
Lbl16

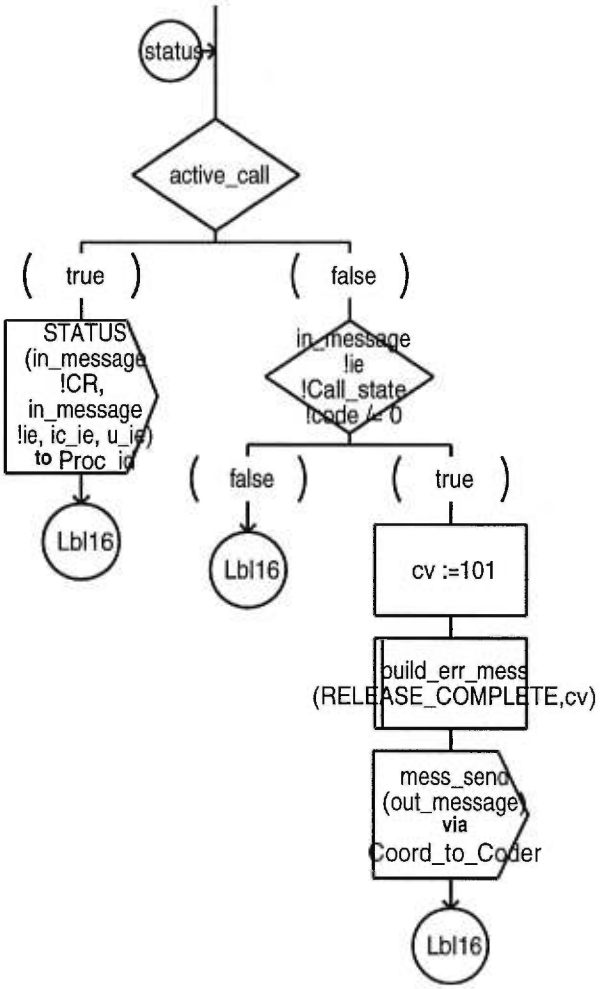

connect
active_call
true
false
CONNECT
(in_message
!CR,
in_message
ie, ic_ie, u_ie)
to
Proc_id
cv := 81
build_err_mess
(RELEASE_COMPLETE,cv)
mess_send
(out_message)
via
Coord_to_Coder
Lbl16
Lbl16

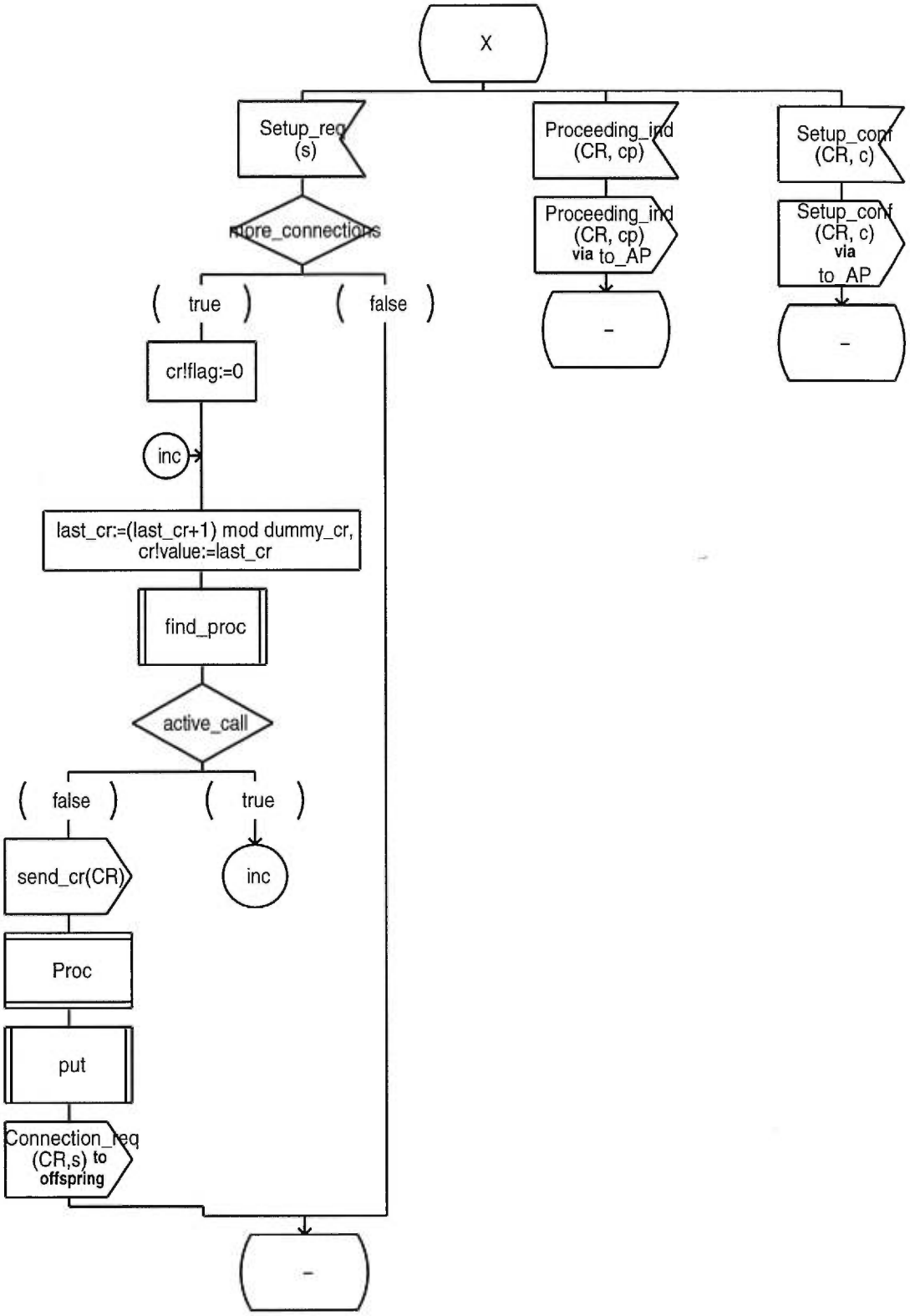

c_ack
active_call
false
true
cv := 81
build_err_mess
(RELEASE_COMPLETE,cv)
mess_send
(out_message)
via
Coord_to_Coder
CONNECT_ACK
(in_message
!CR,
in_message
ie, ic_ie, u_ie)
to
Proc_id
Lbl16
Lbl16

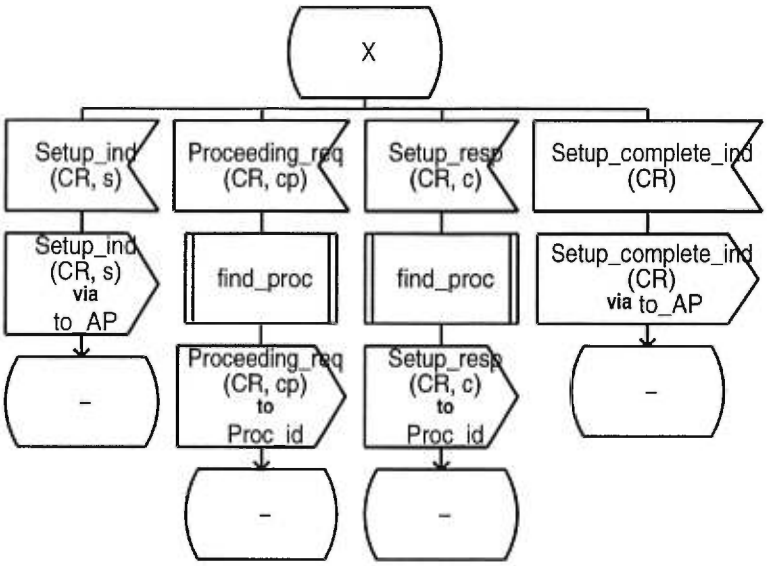

release
active_call
false
true
cv := 81
build_err_mess
(RELEASE_COMPLETE,cv)
mess_send
(out_message)
via
Coord_to_Coder
RELEASE
(in_message
!CR,
in_message
ie, ic_ie, u_ie)
to
Proc_id
Lbl16
Lbl16

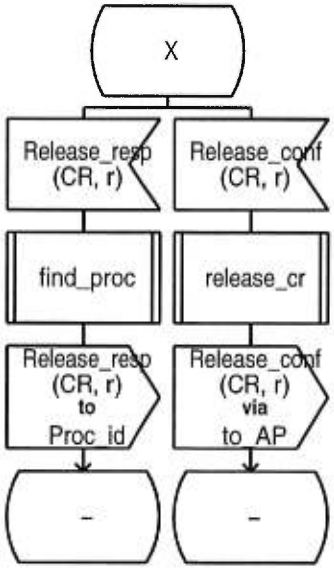

r_c
active_call
true
false
RELEASE_COMPLETE
(in_message
!CR,
in_message
!ie, ic_ie, u_ie)
to
Proc_id
Lbl16
Lbl16

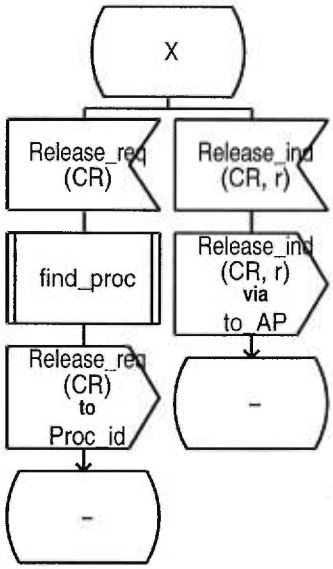

st_enter
active_call
true
false
STATUS_ENQUIRY
(in_message
!CR
,in_message
!ie, ic_ie, u_ie)
to Proc_id
Lbl16
cv :=30
build_err_mess
(STATUS,cv)
out_message!ie
!Call_state
!presence := true,
out_message!ie
!Call_state!code
:= 0
mess_send
(out_message)
via
Coord_to_Coder
Lbl16

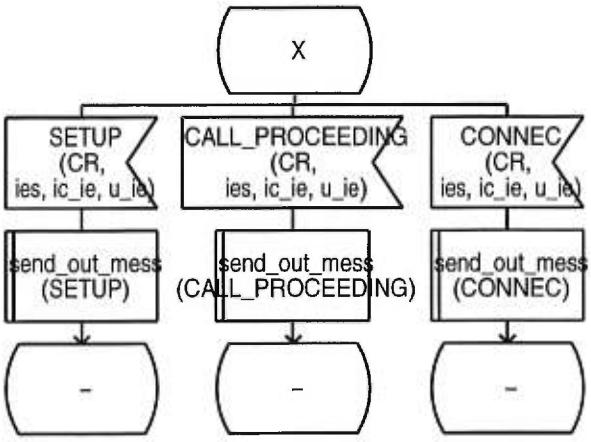

status
active_call
true
false
STATUS
(in_message
!CR,
in_message
!ie, ic_ie, u_ie)
to Proc_id
Lbl16
in_message
!ie
!Call_state
!code /= 0
false
true
Lbl16
cv :=101
build_err_mess
(RELEASE_COMPLETE,cv)
mess_send
(out_message)
via
Coord_to_Coder
Lbl16

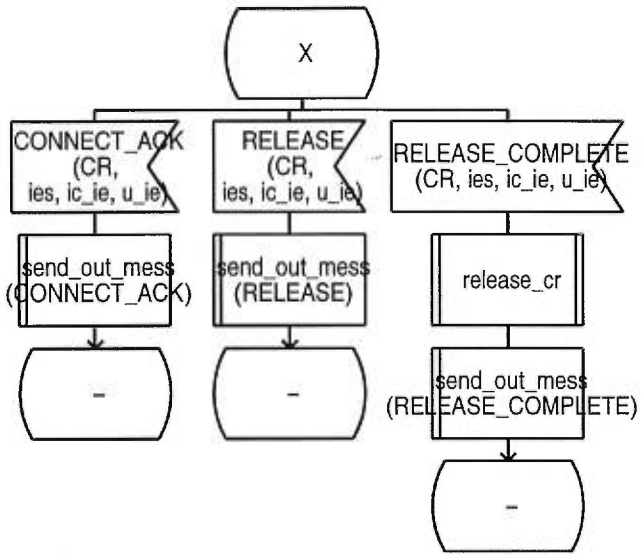

X
Setup_req
(s)
more_connections
true
false
cr!flag:=0
inc
last_cr:=(last_cr+1) mod dummy_cr,
cr!value:=last_cr
find_proc
active_call
false
true
send_cr(CR)
inc
Proc
put
Connection_req
(CR,s) to
offspring
-
Proceeding_ind
(CR, cp)
Proceeding_ind
(CR, cp)
via to_AP
-
Setup_conf
(CR, c)
Setup_conf
(CR, c)
via
to_AP
-

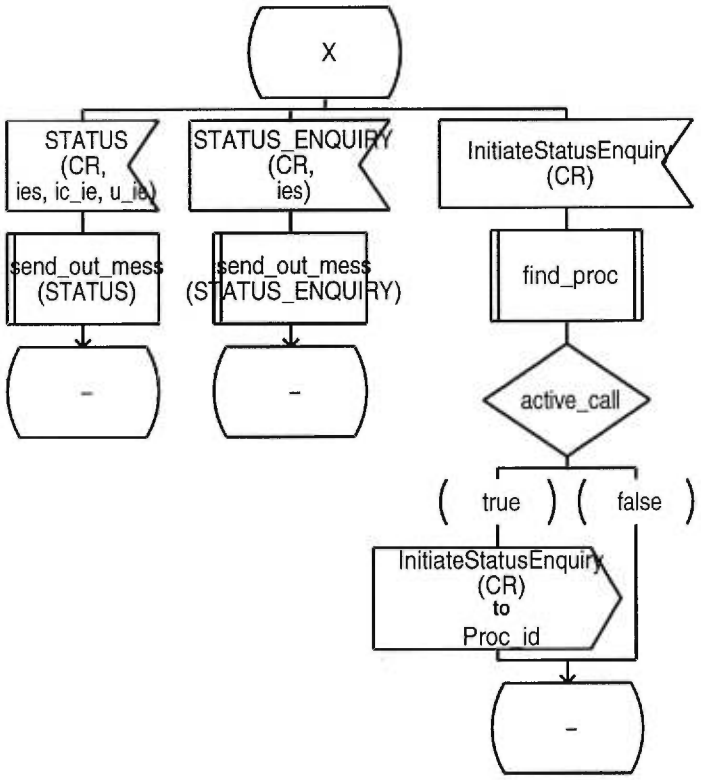

X
Setup_ind
(CR, s)
Proceeding_req
(CR, cp)
Setup_resp
(CR, c)
Setup_complete_ind
(CR)
Setup_ind
(CR, s)
via
to_AP
find_proc
find_proc
Setup_complete_ind
(CR)
via to_AP
Proceeding_req
(CR, cp)
to
Proc_id
Setup_resp
(CR, c)
to
Proc_id

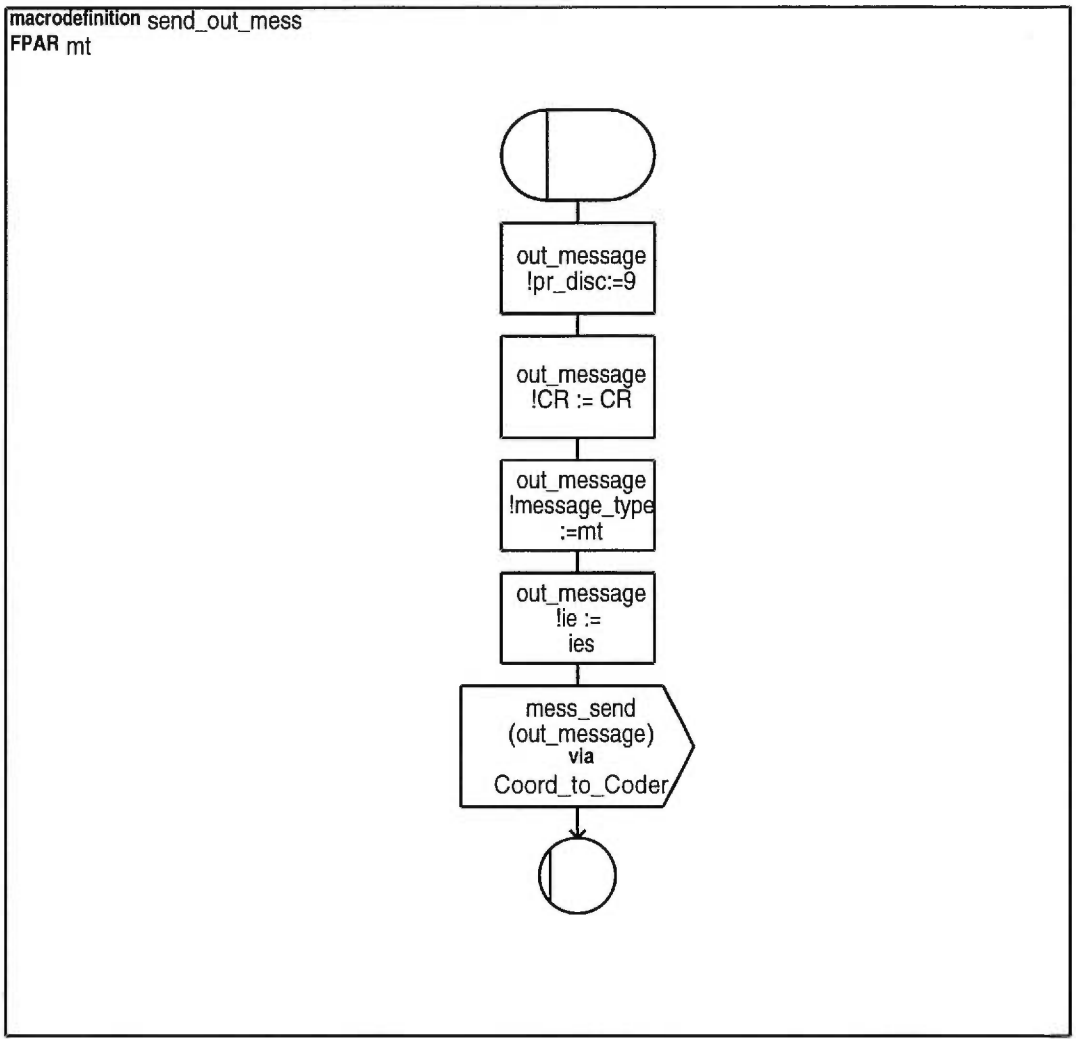

X
Release_resp
(CR, r)
Release_conf
(CR, r)
find_proc
release_cr
Release_resp
(CR, r)
to
Proc_id
Release_conf
(CR, r)
via
to_AP

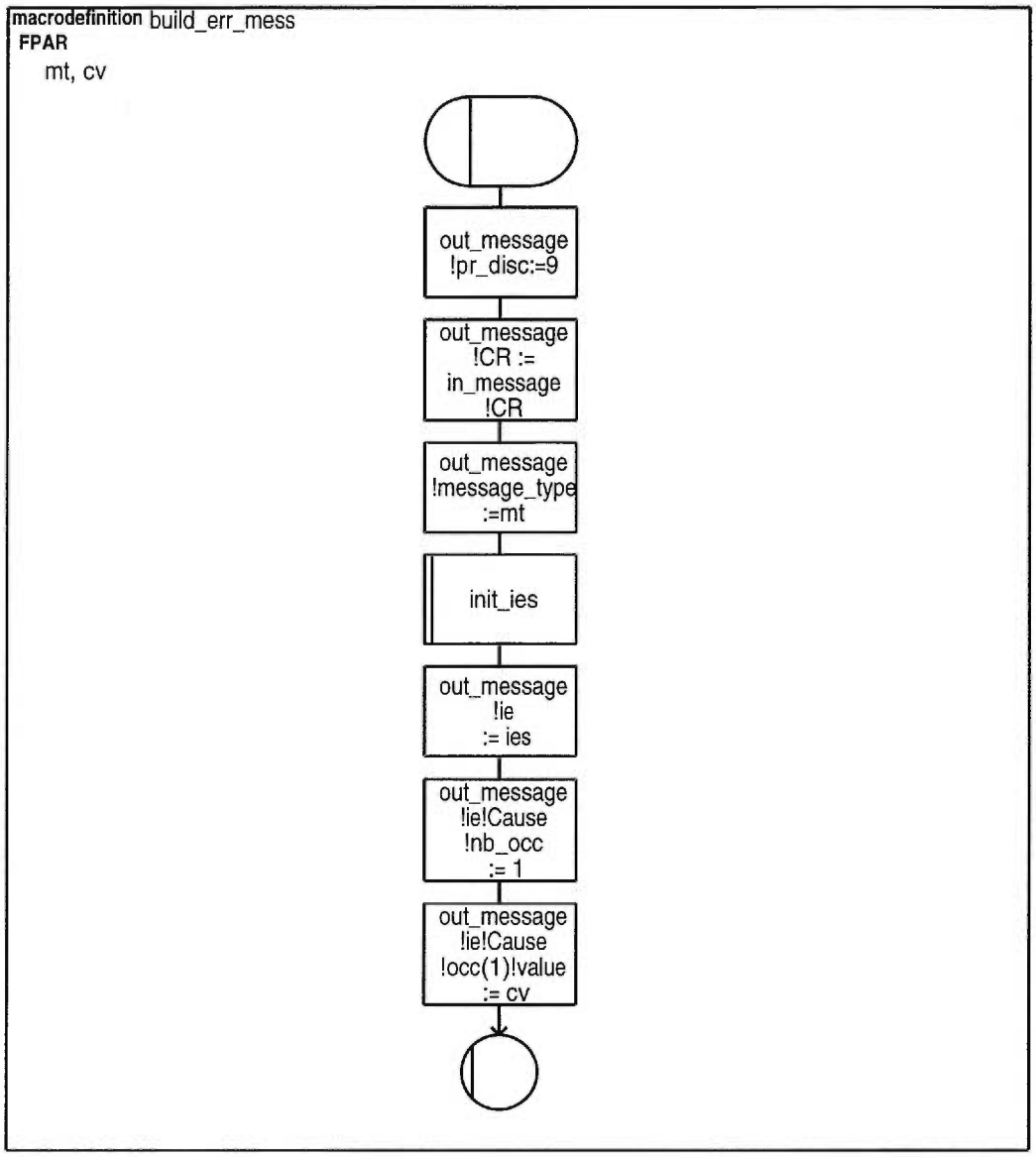

X
Release_req
(CR)
Release_ind
(CR, r)
find_proc
Release_ind
(CR, r)
via
to_AP
Release_req
(CR)
to
Proc_id

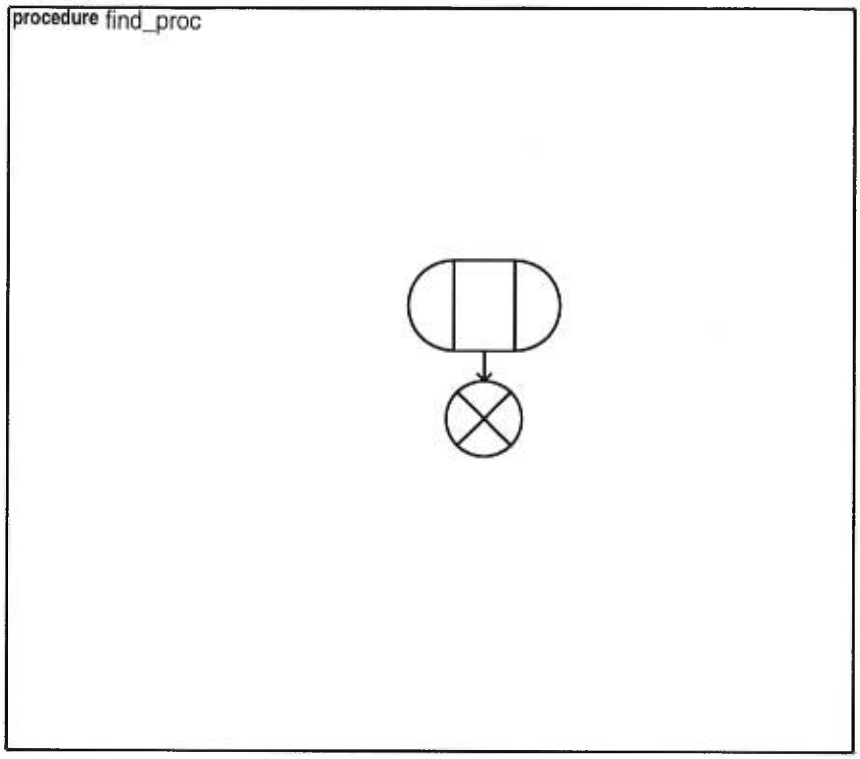

X
SETUP
(CR,
ies, ic_ie, u_ie)
CALL_PROCEEDING
(CR,
ies, ic_ie, u_ie)
CONNEC
(CR,
ies, ic_ie, u_ie)
send_out_mess
(SETUP)
send_out_mess
(CALL_PROCEEDING)
send_out_mess
(CONNEC)
-
-
-

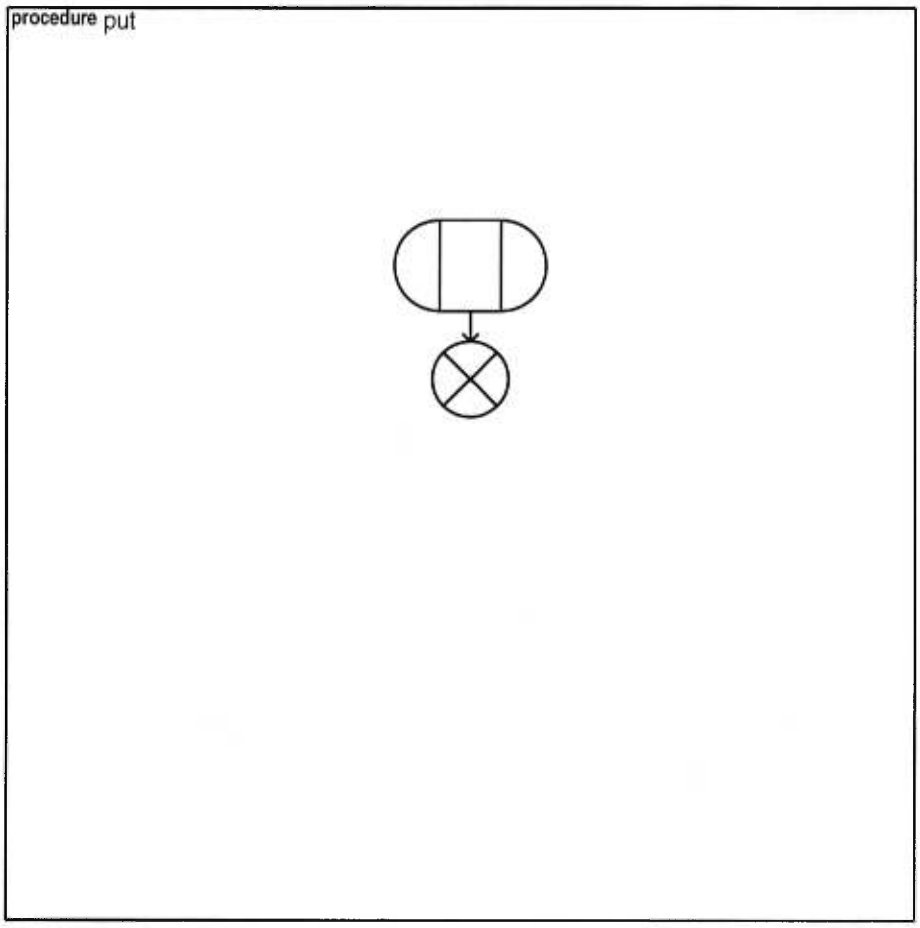

X
CONNECT_ACK
(CR,
ies, ic_ie, u_ie)
RELEASE
(CR,
ies, ic_ie, u_ie)
RELEASE_COMPLETE
(CR, ies, ic_ie, u_ie)
send_out_mess
(CONNECT_ACK)
send_out_mess
(RELEASE)
release_cr
send_out_mess
(RELEASE_COMPLETE)
-
-
-

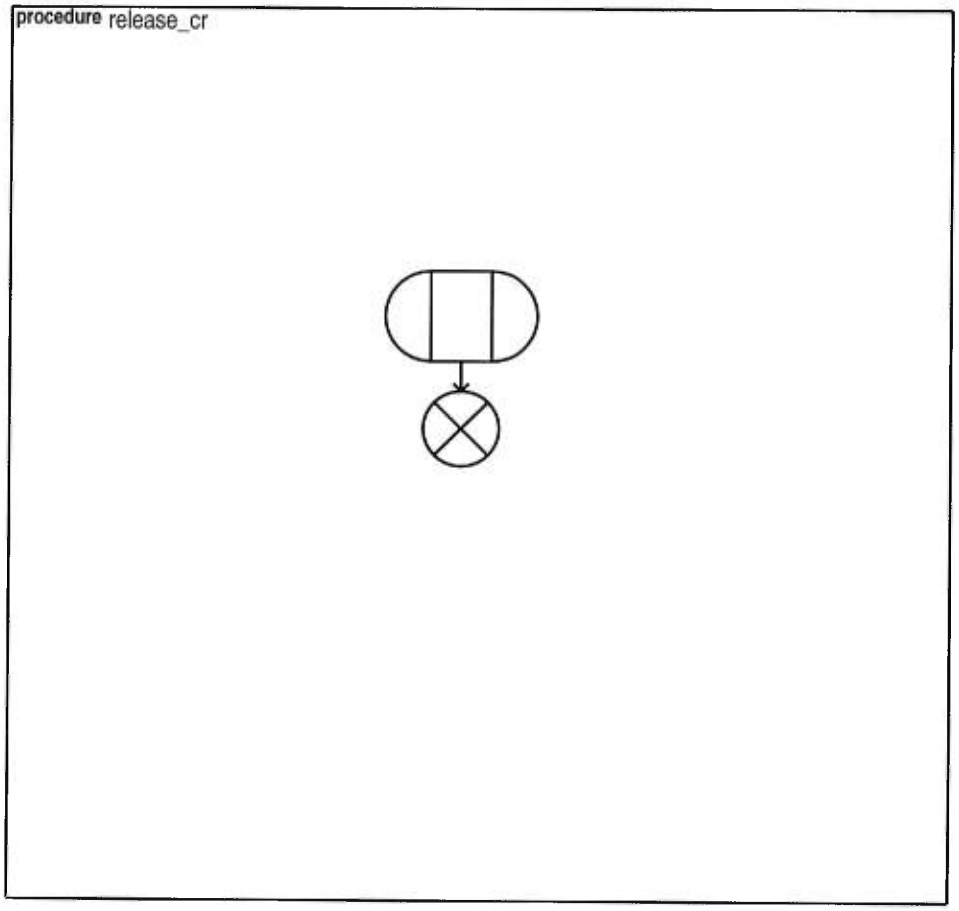

X
STATUS
(CR,
ies, ic_ie, u_ie)
STATUS_ENQUIRY
(CR,
ies)
InitiateStatusEnquiry
(CR)
send_out_mess
(STATUS)
send_out_mess
(STATUS_ENQUIRY)
find_proc
-
-
active_call
true
false
InitiateStatusEnquiry
(CR)
to
Proc_id
-


macrodefinition send_out_mess
FPAR mt
out_message
!pr_disc:=9
out_message
!CR := CR
out_message
!message_type
:=mt
out_message
!ie :=
ies
mess_send
(out_message)
via
Coord_to_Coder

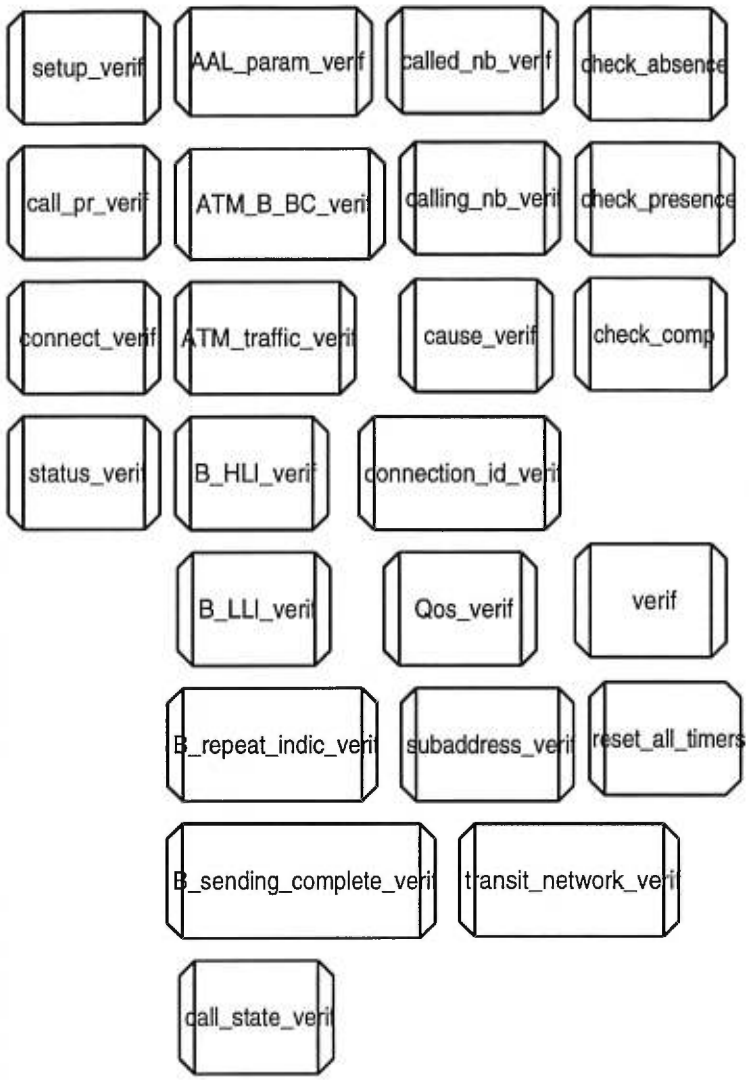

macrodefinition build_err_mess
FPAR
mt, cv
out_message !pr_disc:=9
out_message !CR := in_message !CR
out_message !message_type :=mt
init_ies
out_message !ie := ies
out_message !ie!Cause !nb_occ := 1
out_message !ie!Cause !occ(1)!value := cv

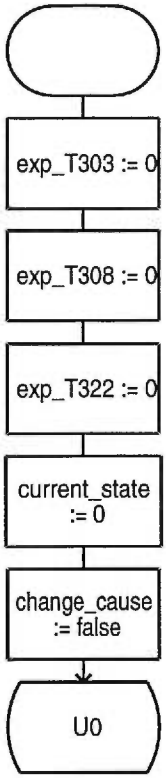

procedure find_proc

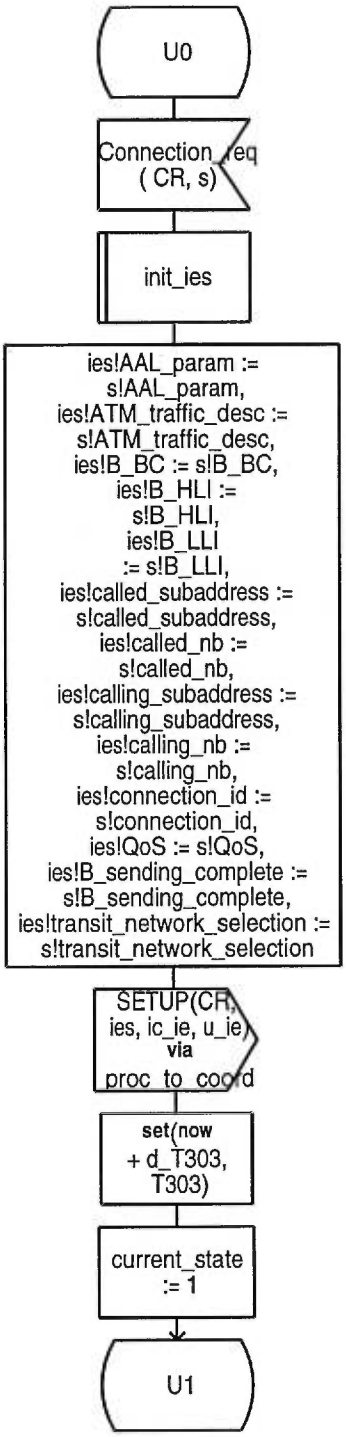

procedure put

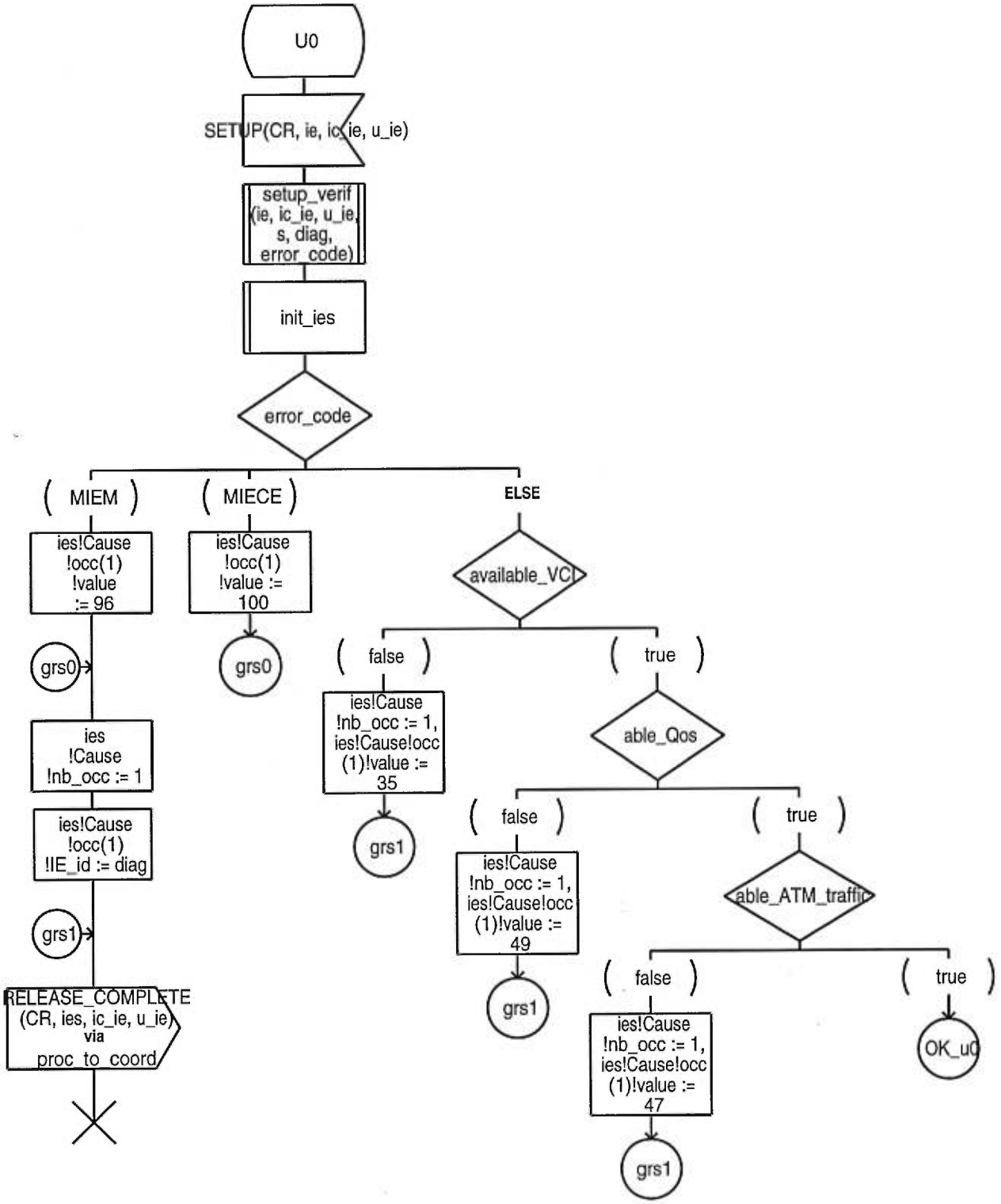

procedure release_cr

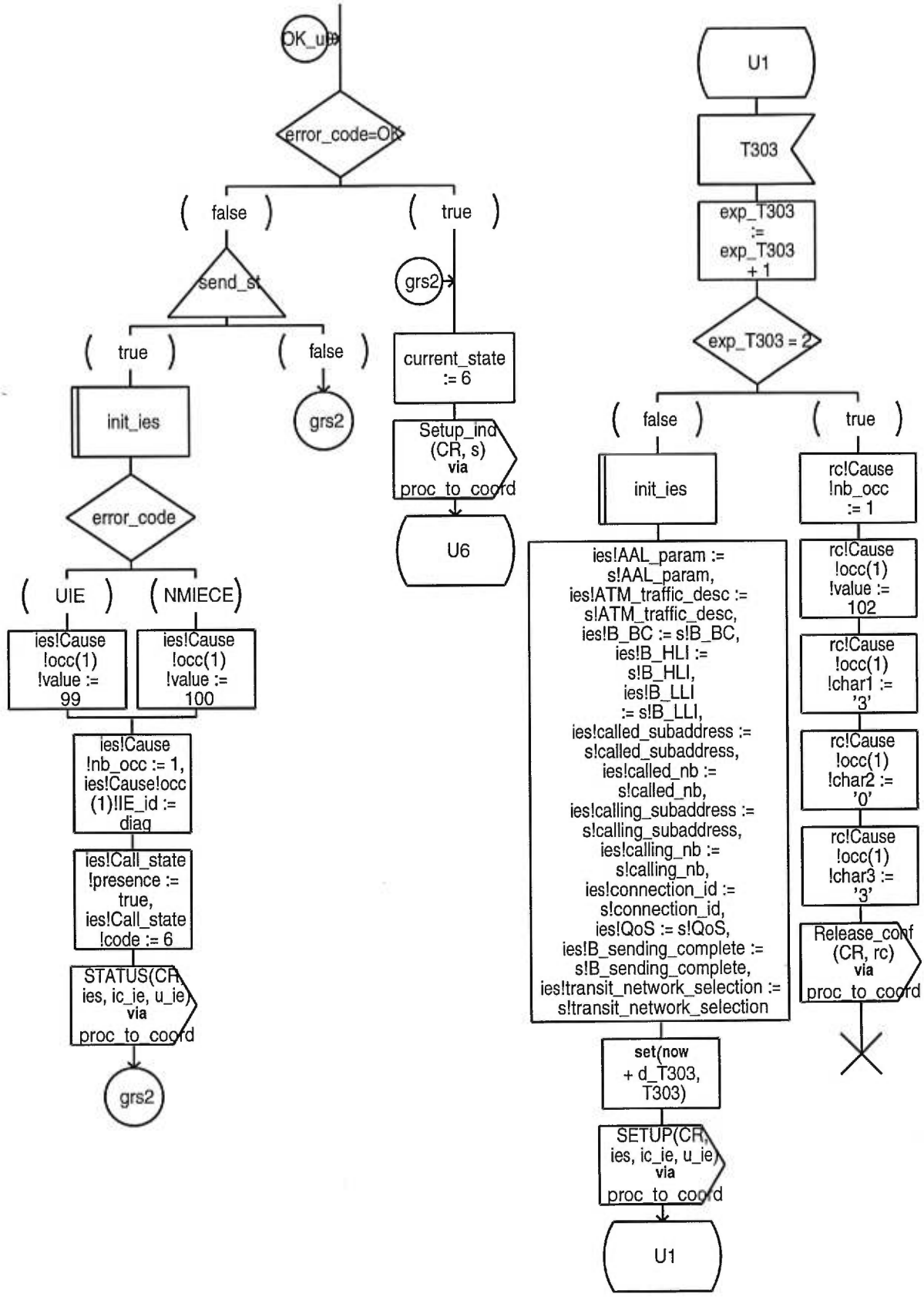
```
NEWTYPE error_code_type
  LITERALS OK, MIEM, MIECE, UIE, NMIECE;
ENDNEWTYPE;

NEWTYPE octet_set
  POWERSET(octet)
ENDNEWTYPE;

NEWTYPE int_5_set
  POWERSET(int_5)
ENDNEWTYPE;

NEWTYPE ci_type
 LITERALS ci_man, ci_opt;
ENDNEWTYPE;

/* nb. max. d'envois du
message STATUS_ENQUIRY */
SYNONYM nb_trans_T322 = 2;

timer
  T303,
  T308,
  T310,
  T313,
  T322;
dcl
  exp_T303 Natural,
  exp_T308 Natural,
  exp_T322 Natural;
dcl
  u_ie occ_array_type,
  ic_ie ie_values_set;
dcl
  current_state int_6;
dcl
  error_code error_code_type;
dcl
  diag occ_array_type;
dcl
  ie,
  ies ie_type;
dcl
  s setup_struct,
  cp call_pr_struct,
  c connect_struct,
  r,
  rc release_struct,
  st status_struct;
dcl
  CR call_reference_type;
dcl
  unrec_type octet;
dcl
  change_cause Boolean,
  new_value octet;
dcl
  incompatible_states,
  available_VCI,
  able_QoS,
  able_ATM_traffic Boolean := true;
dcl
  without_error Boolean;
```

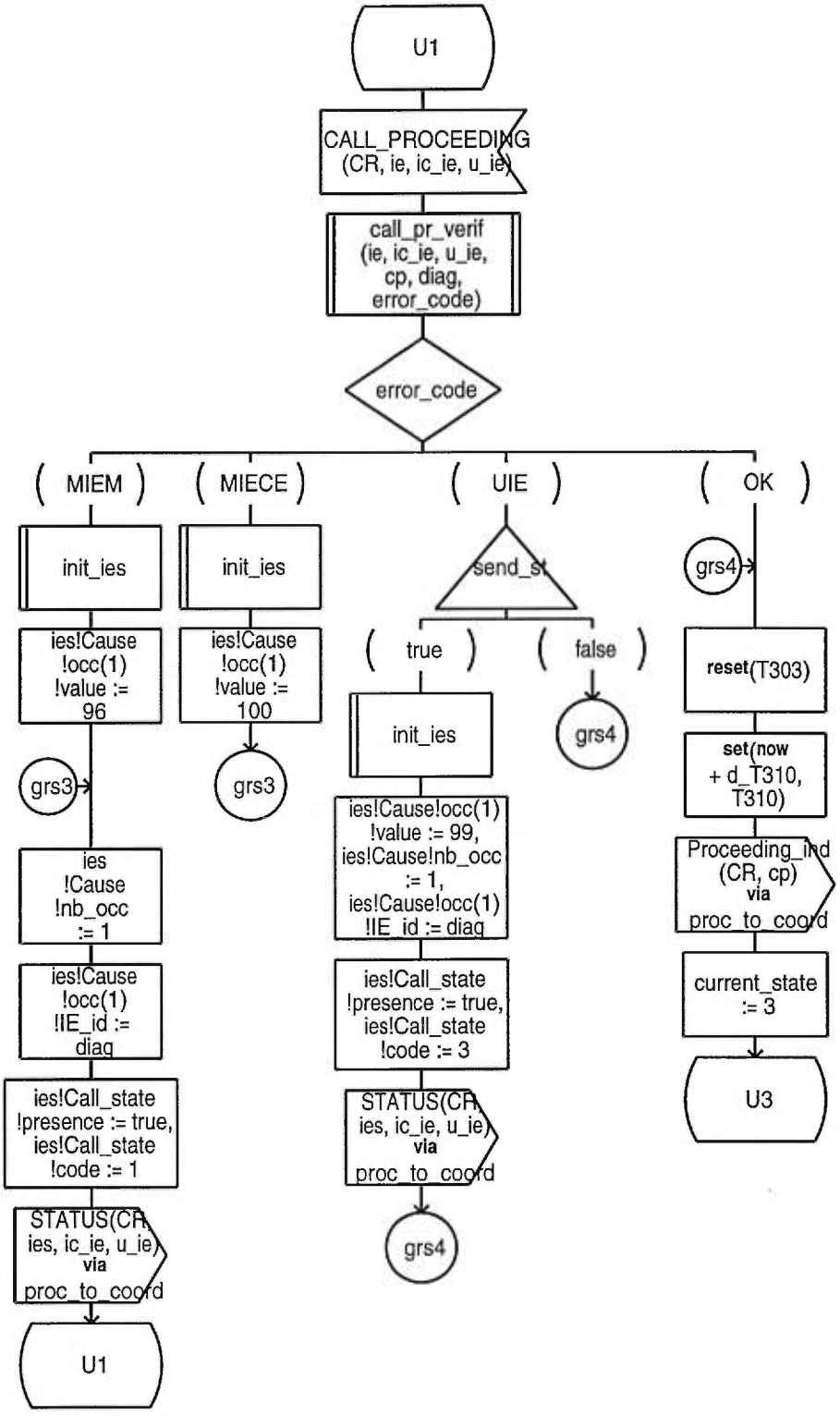


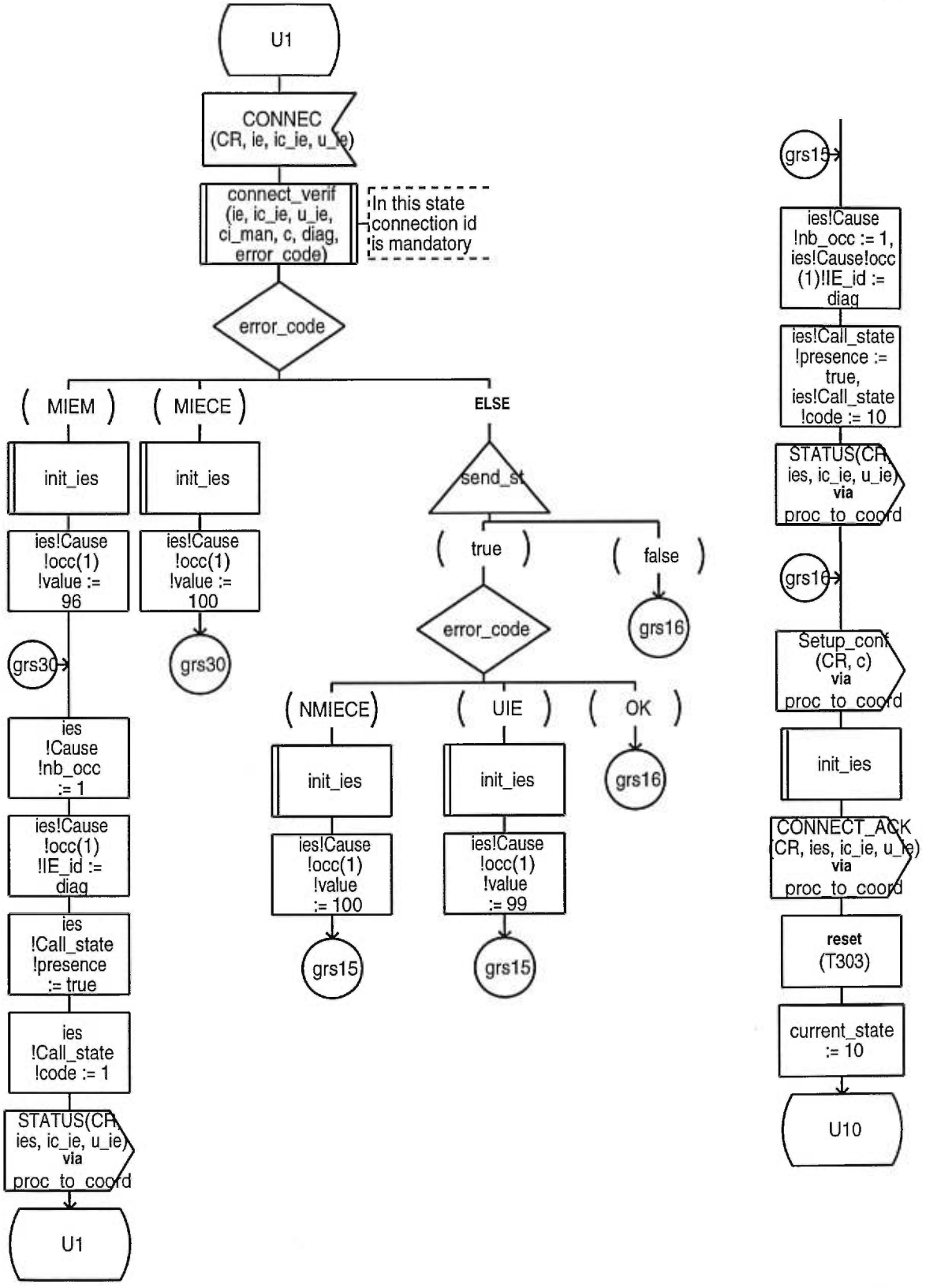

exp_T303 := 0
exp_T308 := 0
exp_T322 := 0
current_state := 0
change_cause := false
U0

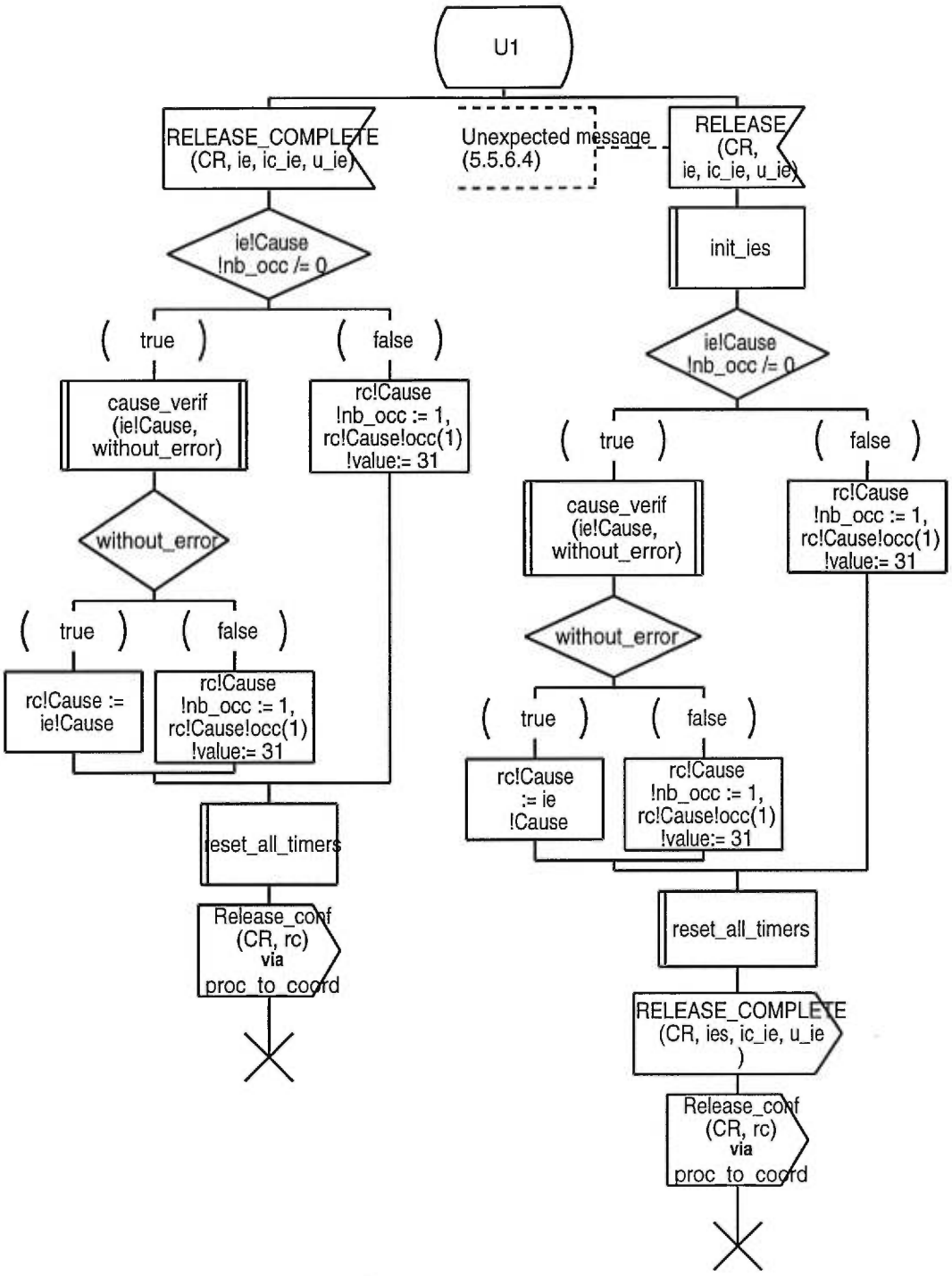

U0
Connection_req (CR, s)
init_ies
ies!AAL_param := s!AAL_param, ies!ATM_traffic_desc := s!ATM_traffic_desc, ies!B_BC := s!B_BC, ies!B_HLI := s!B_HLI, ies!B_LLI := s!B_LLI, ies!called_subaddress := s!called_subaddress, ies!called_nb := s!called_nb, ies!calling_subaddress := s!calling_subaddress, ies!calling_nb := s!calling_nb, ies!connection_id := s!connection_id, ies!QoS := s!QoS, ies!B_sending_complete := s!B_sending_complete, ies!transit_network_selection := s!transit_network_selection
SETUP(CR, ies, ic_ie, u_ie) via proc_to_coord
set(now + d_T303, T303)
current_state := 1
U1

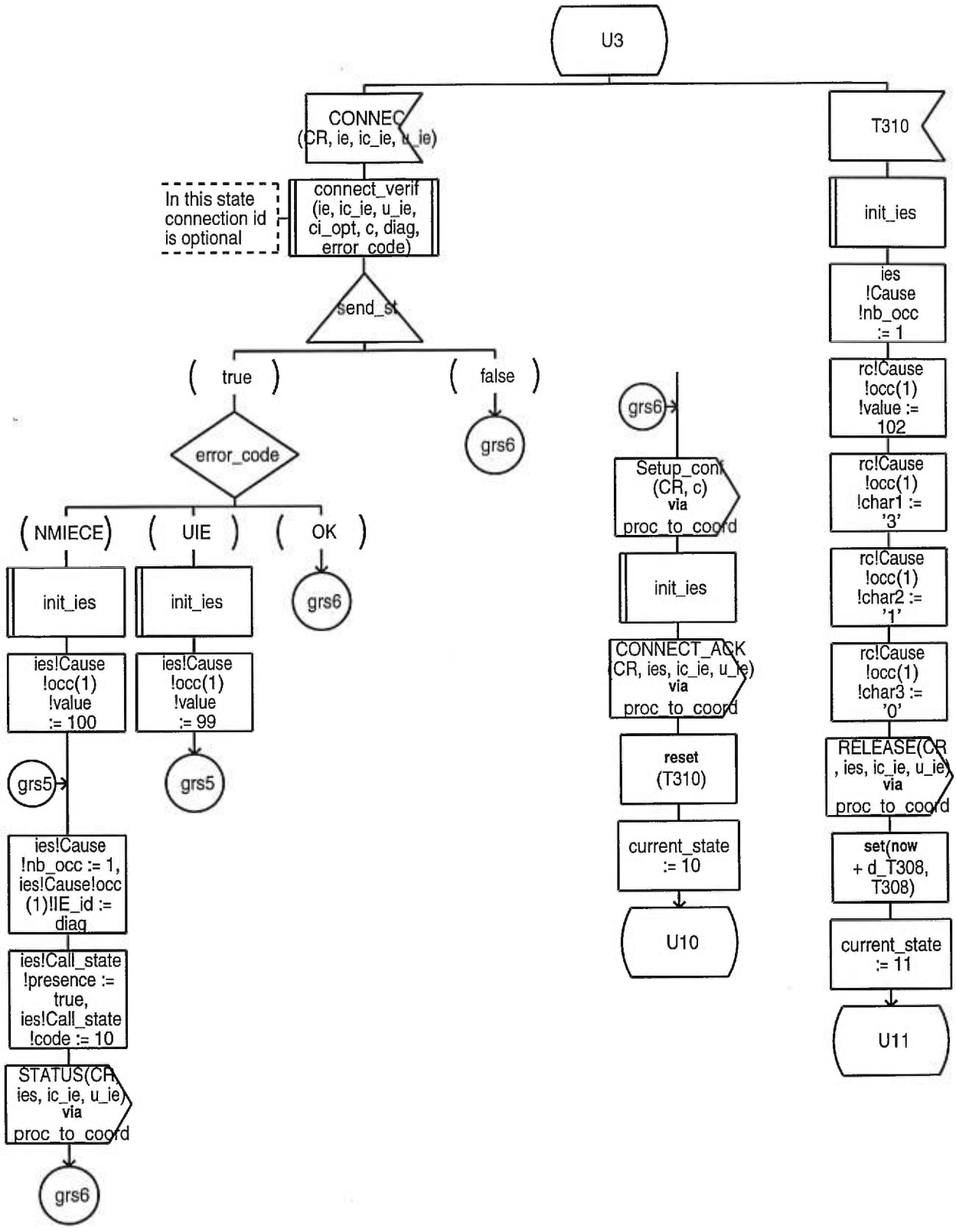

U0
SETUP(CR, ie, ic_ie, u_ie)
setup_verif (ie, ic_ie, u_ie, s, diag, error_code)
init_ies
error_code
MIEM
MIECE
ELSE
ies!Cause !occ(1) !value := 96
ies!Cause !occ(1) !value := 100
available_VCI
grs0
grs0
false
true
ies!Cause !nb_occ := 1, ies!Cause!occ (1)!value := 35
able_Qos
ies !Cause !nb_occ := 1
ies!Cause !occ(1) !IE_id := diag
grs1
grs1
false
true
ies!Cause !nb_occ := 1, ies!Cause!occ (1)!value := 49
able_ATM_traffic
RELEASE_COMPLETE (CR, ies, ic_ie, u_ie) via proc_to_coord
false
true
ies!Cause !nb_occ := 1, ies!Cause!occ (1)!value := 47
OK_u0
grs1

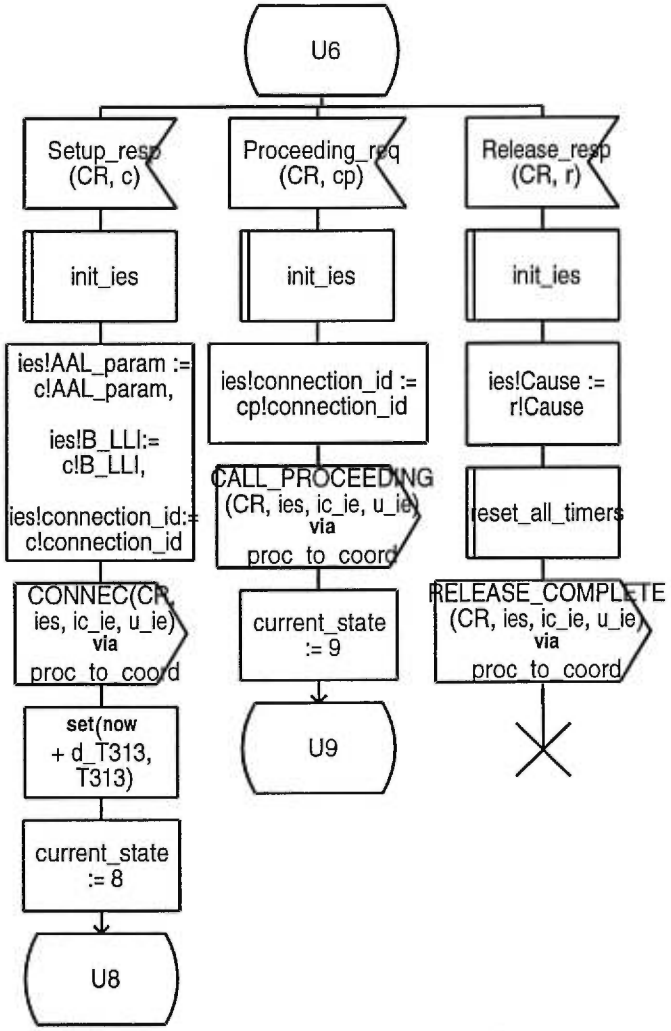

OK_u
error_code=OK
false
true
send_st
grs2
true
false
current_state
:= 6
init_ies
grs2
Setup_ind
(CR, s)
via
proc_to_coord
error_code
U6
UIE
NMIECE
ies!Cause
!occ(1)
!value :=
99
ies!Cause
!occ(1)
!value :=
100
ies!Cause
!nb_occ := 1,
ies!Cause!occ
(1)!IE_id :=
diag
ies!Call_state
!presence :=
true,
ies!Call_state
!code := 6
STATUS(CR,
ies, ic_ie, u_ie)
via
proc_to_coord
grs2
U1
T303
exp_T303
:=
exp_T303
+ 1
exp_T303 = 2
false
true
init_ies
rc!Cause
!nb_occ
:= 1
ies!AAL_param :=
s!AAL_param,
ies!ATM_traffic_desc :=
s!ATM_traffic_desc,
ies!B_BC := s!B_BC,
ies!B_HLI :=
s!B_HLI,
ies!B_LLI
:= s!B_LLI,
ies!called_subaddress :=
s!called_subaddress,
ies!called_nb :=
s!called_nb,
ies!calling_subaddress :=
s!calling_subaddress,
ies!calling_nb :=
s!calling_nb,
ies!connection_id :=
s!connection_id,
ies!QoS := s!QoS,
ies!B_sending_complete :=
s!B_sending_complete,
ies!transit_network_selection :=
s!transit_network_selection
rc!Cause
!occ(1)
!value :=
102
rc!Cause
!occ(1)
!char1 :=
'3'
rc!Cause
!occ(1)
!char2 :=
'0'
rc!Cause
!occ(1)
!char3 :=
'3'
Release_conf
(CR, rc)
via
proc_to_coord
set(now
+ d_T303,
T303)
SETUP(CR,
ies, ic_ie, u_ie)
via
proc_to_coord
U1

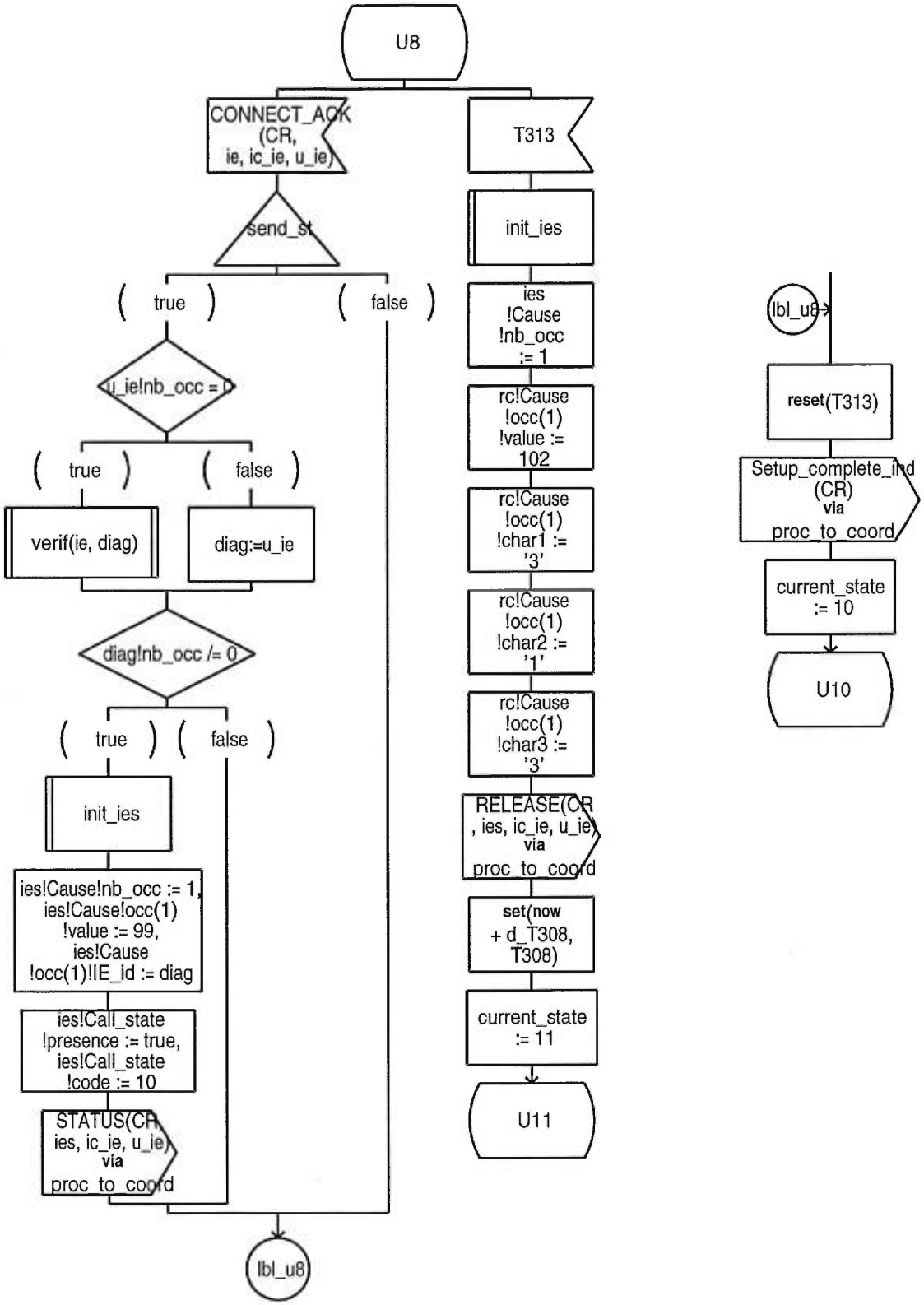

U1
CALL_PROCEEDING
(CR, ie, ic_ie, u_ie)
call_pr_verif
(ie, ic_ie, u_ie,
cp, diag,
error_code)
error_code
MIEM
MIECE
UIE
OK
init_ies
init_ies
send_st
grs4
ies!Cause
!occ(1)
!value :=
96
ies!Cause
!occ(1)
!value :=
100
true
false
reset(T303)
grs3
grs3
init_ies
grs4
set(now
+ d_T310,
T310)
ies
!Cause
!nb_occ
:= 1
ies!Cause!occ(1)
!value := 99,
ies!Cause!nb_occ
:= 1,
ies!Cause!occ(1)
!IE_id := diag
Proceeding_ind
(CR, cp)
via
proc_to_coord
ies!Cause
!occ(1)
!IE_id :=
diag
ies!Call_state
!presence := true,
ies!Call_state
!code := 3
current_state
:= 3
ies!Call_state
!presence := true,
ies!Call_state
!code := 1
STATUS(CR,
ies, ic_ie, u_ie)
via
proc_to_coord
U3
STATUS(CR,
ies, ic_ie, u_ie)
via
proc_to_coord
grs4
U1

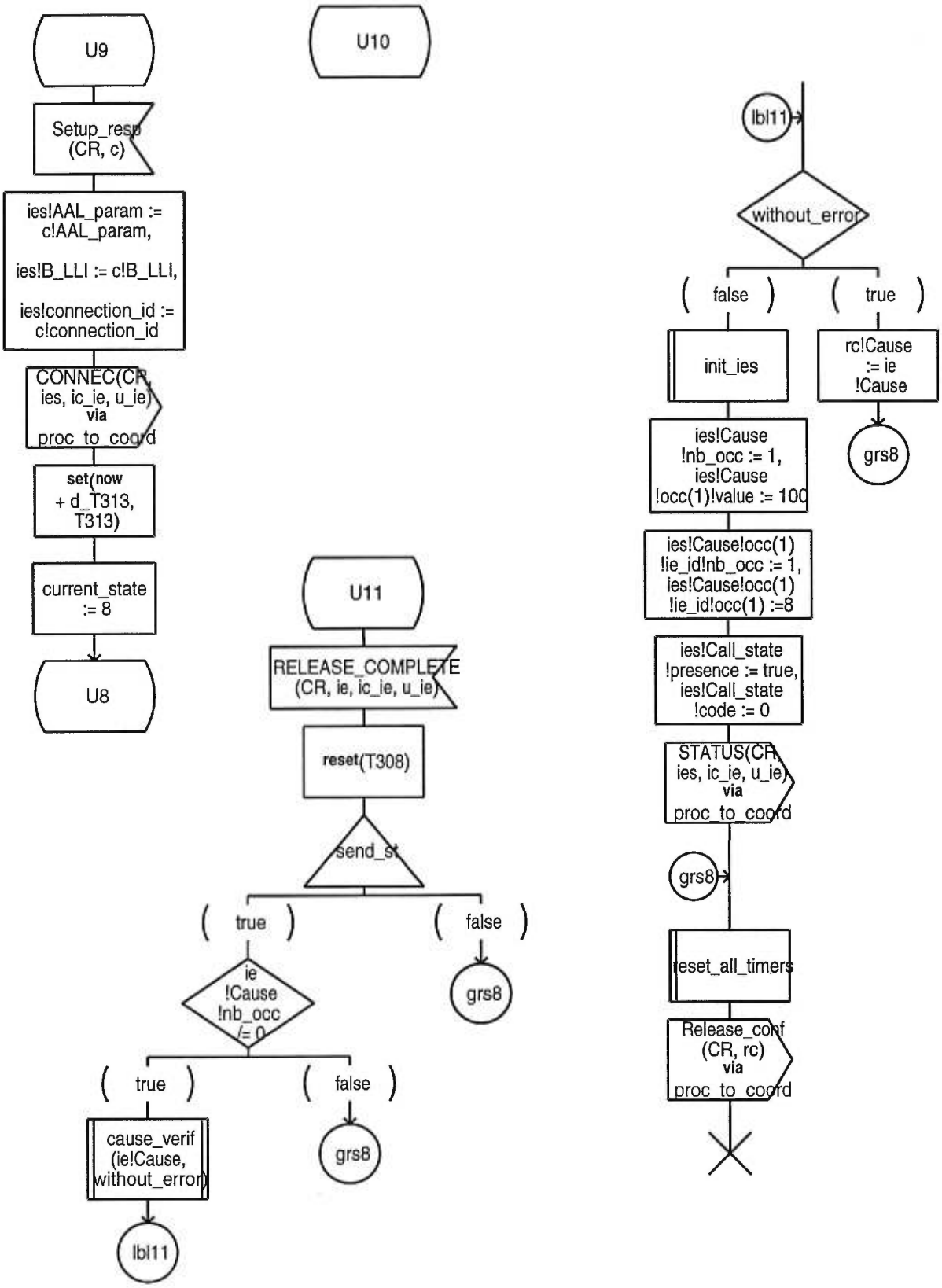

U1
CONNEC
(CR, ie, ic_ie, u_ie)
connect_verif
(ie, ic_ie, u_ie,
ci_man, c, diag,
error_code)
In this state
connection id
is mandatory
error_code
MIEM
MIECE
ELSE
init_ies
init_ies
ies!Cause
!occ(1)
!value :=
96
ies!Cause
!occ(1)
!value :=
100
grs30
grs30
ies
!Cause
!nb_occ
:= 1
ies!Cause
!occ(1)
!IE_id :=
diag
ies
!Call_state
!presence
:= true
ies
!Call_state
!code := 1
STATUS(CR,
ies, ic_ie, u_ie)
via
proc_to_coord
U1
send_st
true
false
error_code
grs16
NMIECE
UIE
OK
init_ies
init_ies
grs16
ies!Cause
!occ(1)
!value
:= 100
ies!Cause
!occ(1)
!value
:= 99
grs15
grs15
grs15
ies!Cause
!nb_occ := 1,
ies!Cause!occ
(1)!IE_id :=
diag
ies!Call_state
!presence :=
true,
ies!Call_state
!code := 10
STATUS(CR,
ies, ic_ie, u_ie)
via
proc_to_coord
grs16
Setup_conf
(CR, c)
via
proc_to_coord
init_ies
CONNECT_ACK
(CR, ies, ic_ie, u_ie)
via
proc_to_coord
reset
(T303)
current_state
:= 10
U10

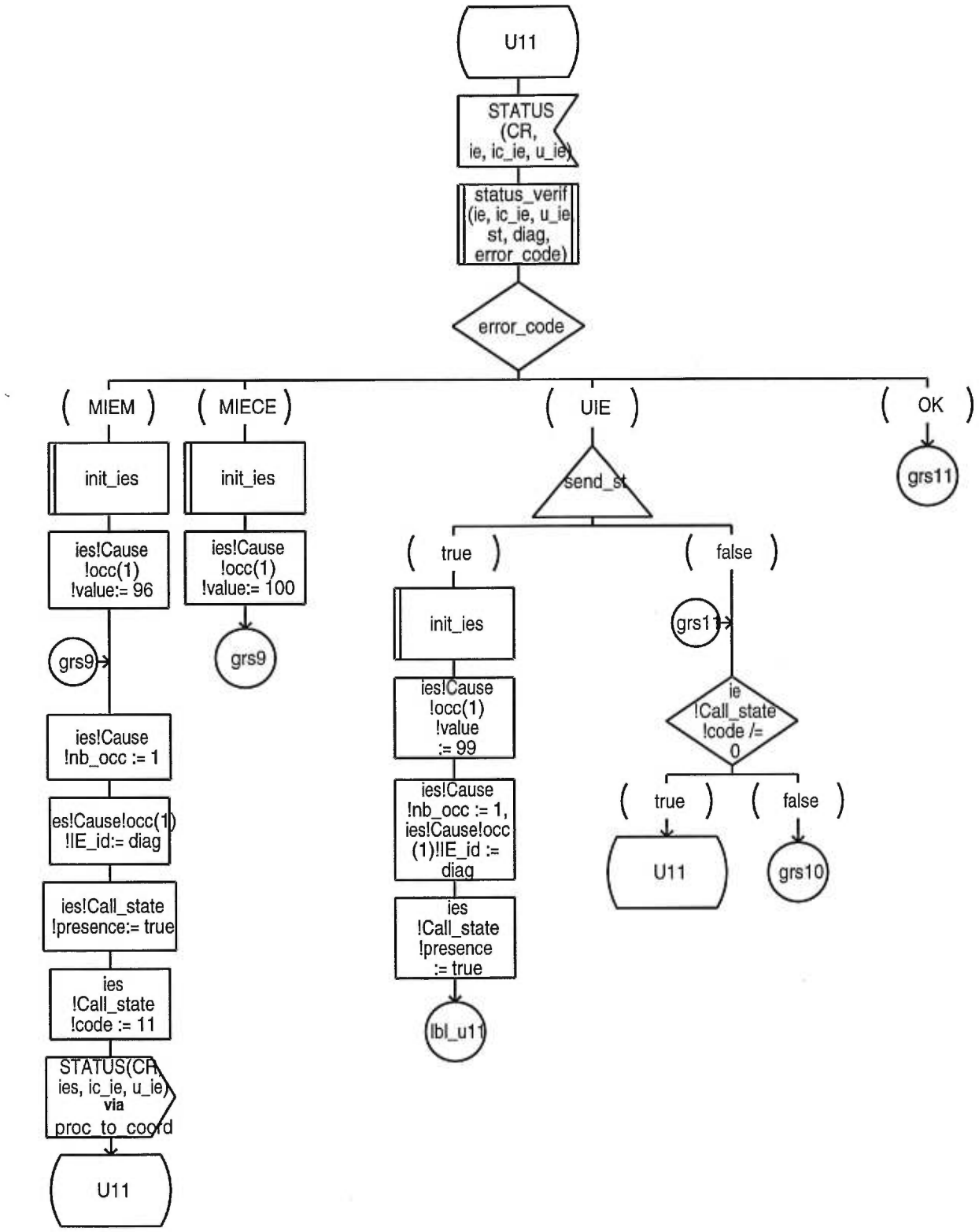

U1
RELEASE_COMPLETE (CR, ie, ic_ie, u_ie)
Unexpected message (5.5.6.4)
RELEASE (CR, ie, ic_ie, u_ie)
ie!Cause !nb_occ /= 0
true
false
cause_verif (ie!Cause, without_error)
rc!Cause !nb_occ := 1, rc!Cause!occ(1) !value:= 31
without_error
true
false
rc!Cause := ie!Cause
rc!Cause !nb_occ := 1, rc!Cause!occ(1) !value:= 31
reset_all_timers
Release_conf (CR, rc) via proc_to_coord
init_ies
ie!Cause !nb_occ /= 0
true
false
cause_verif (ie!Cause, without_error)
rc!Cause !nb_occ := 1, rc!Cause!occ(1) !value:= 31
without_error
true
false
rc!Cause := ie !Cause
rc!Cause !nb_occ := 1, rc!Cause!occ(1) !value:= 31
reset_all_timers
RELEASE_COMPLETE (CR, ies, ic_ie, u_ie)
Release_conf (CR, rc) via proc_to_coord

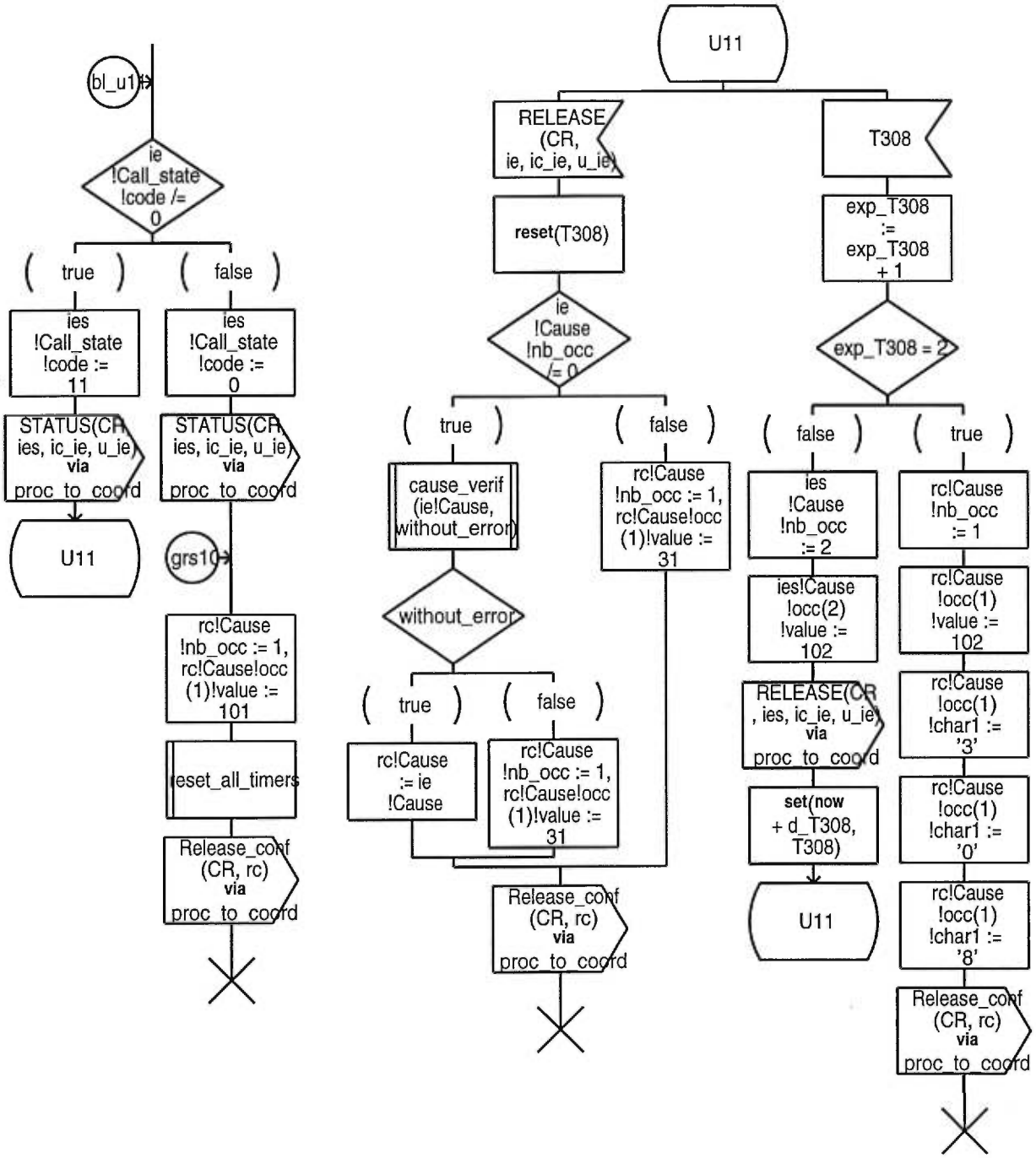

U3
CONNEC
(CR, ie, ic_ie, u_ie)
In this state connection id is optional
connect_verif (ie, ic_ie, u_ie, ci_opt, c, diag, error_code)
send_st
true
false
grs6
error_code
NMIECE
UIE
OK
init_ies
init_ies
grs6
ies!Cause !occ(1) !value := 100
ies!Cause !occ(1) !value := 99
grs5
grs5
ies!Cause !nb_occ := 1, ies!Cause!occ (1)!IE_id := diag
ies!Call_state !presence := true, ies!Call_state !code := 10
STATUS(CR, ies, ic_ie, u_ie) via proc_to_coord
grs6
grs6
Setup_conf (CR, c) via proc_to_coord
init_ies
CONNECT_ACK (CR, ies, ic_ie, u_ie) via proc_to_coord
reset (T310)
current_state := 10
U10
T310
init_ies
ies !Cause !nb_occ := 1
rc!Cause !occ(1) !value := 102
rc!Cause !occ(1) !char1 := '3'
rc!Cause !occ(1) !char2 := '1'
rc!Cause !occ(1) !char3 := '0'
RELEASE(CR, ies, ic_ie, u_ie) via proc_to_coord
set(now + d_T308, T308)
current_state := 11
U11

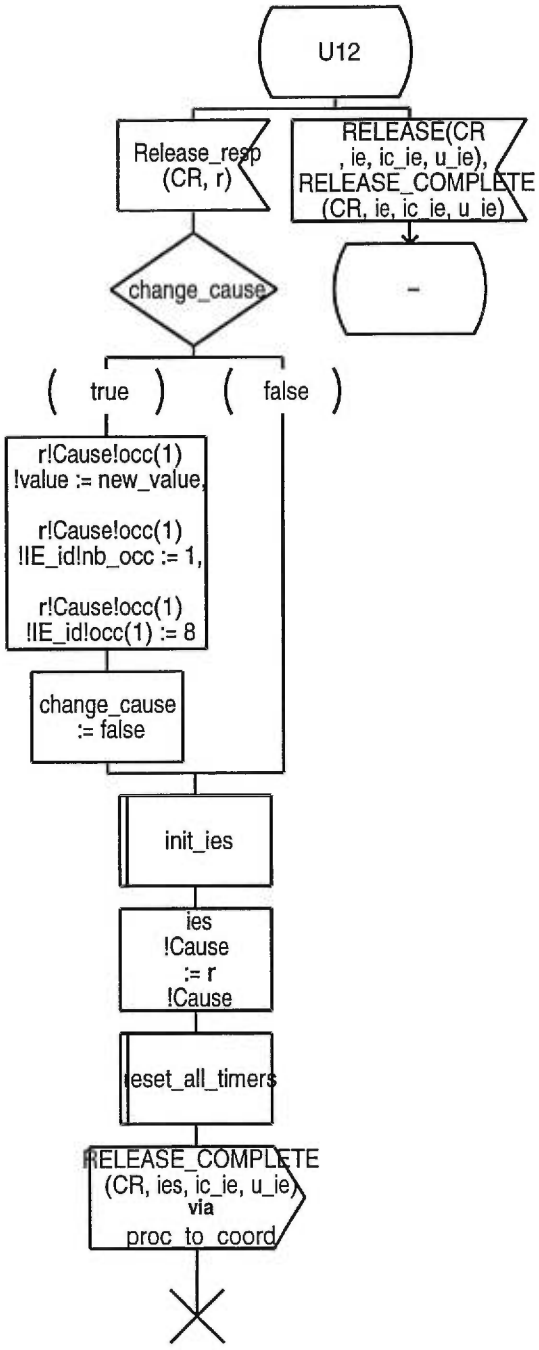

U6
Setup_resp
(CR, c)
Proceeding_req
(CR, cp)
Release_resp
(CR, r)
init_ies
init_ies
init_ies
ies!AAL_param :=
c!AAL_param,
ies!B_LLI:=
c!B_LLI,
ies!connection_id:=
c!connection_id
ies!connection_id :=
cp!connection_id
ies!Cause :=
r!Cause
CALL_PROCEEDING
(CR, ies, ic_ie, u_ie)
via
proc_to_coord
reset_all_timers
CONNEC(CR,
ies, ic_ie, u_ie)
via
proc_to_coord
current_state
:= 9
RELEASE_COMPLETE
(CR, ies, ic_ie, u_ie)
via
proc_to_coord
set(now
+ d_T313,
T313)
U9
current_state
:= 8
U8

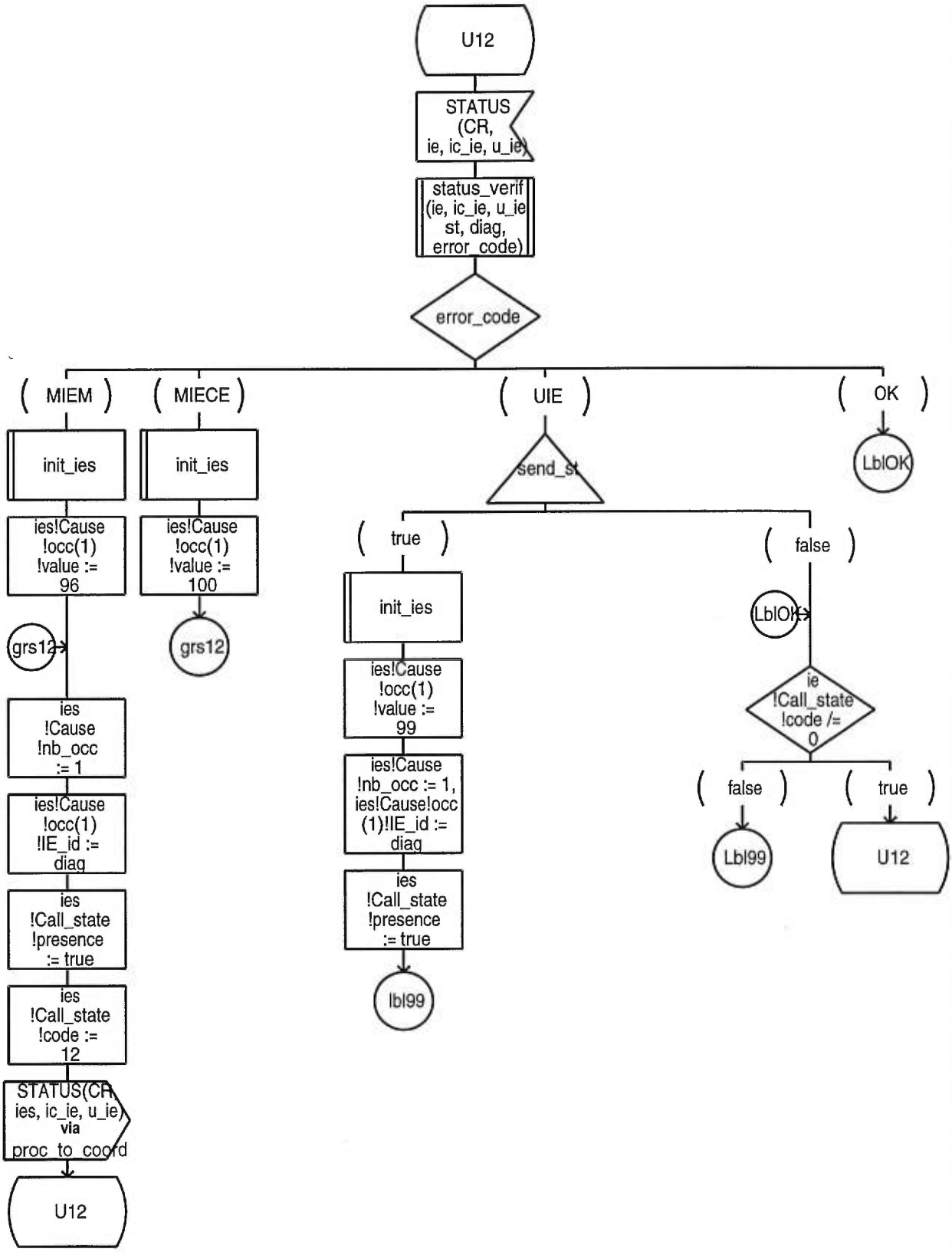

U8
CONNECT_ACK
(CR,
ie, ic_ie, u_ie)
send_st
true
false
u_ie!nb_occ = 0
true
false
verif(ie, diag)
diag:=u_ie
diag!nb_occ /= 0
true
false
init_ies
ies!Cause!nb_occ := 1,
ies!Cause!occ(1)
!value := 99,
ies!Cause
!occ(1)!IE_id := diag
ies!Call_state
!presence := true,
ies!Call_state
!code := 10
STATUS(CR,
ies, ic_ie, u_ie)
via
proc_to_coord
lbl_u8
T313
init_ies
ies
!Cause
!nb_occ
:= 1
rc!Cause
!occ(1)
!value :=
102
rc!Cause
!occ(1)
!char1 :=
'3'
rc!Cause
!occ(1)
!char2 :=
'1'
rc!Cause
!occ(1)
!char3 :=
'3'
RELEASE(CR
, ies, ic_ie, u_ie)
via
proc_to_coord
set(now
+ d_T308,
T308)
current_state
:= 11
U11
lbl_u8
reset(T313)
Setup_complete_ind
(CR)
via
proc_to_coord
current_state
:= 10
U10

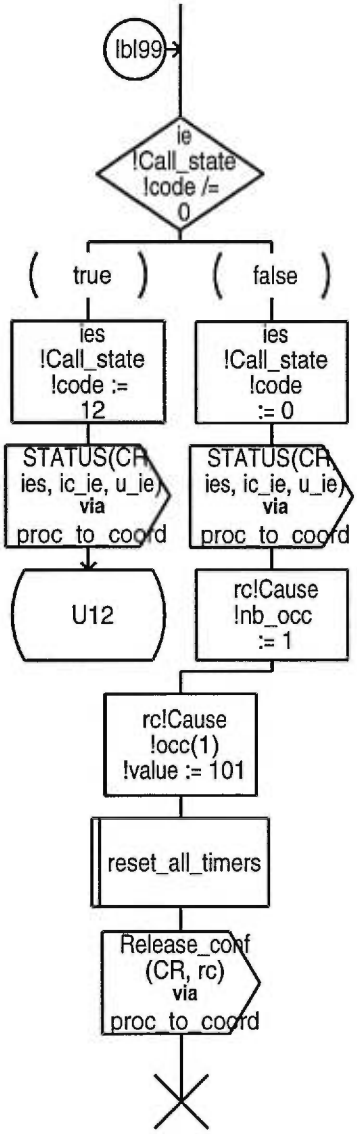

U9
Setup_resp
(CR, c)
ies!AAL_param :=
c!AAL_param,
ies!B_LLI := c!B_LLI,
ies!connection_id :=
c!connection_id
CONNEC(CR,
ies, ic_ie, u_ie)
via
proc_to_coord
set(now
+ d_T313,
T313)
current_state
:= 8
U8
U10
U11
RELEASE_COMPLETE
(CR, ie, ic_ie, u_ie)
reset(T308)
send_st
true
false
ie
!Cause
!nb_occ
/= 0
true
false
cause_verif
(ie!Cause,
without_error)
lbl11
grs8
grs8
lbl11
without_error
false
true
init_ies
rc!Cause
:= ie
!Cause
grs8
ies!Cause
!nb_occ := 1,
ies!Cause
!occ(1)!value := 100
ies!Cause!occ(1)
!ie_id!nb_occ := 1,
ies!Cause!occ(1)
!ie_id!occ(1) :=8
ies!Call_state
!presence := true,
ies!Call_state
!code := 0
STATUS(CR,
ies, ic_ie, u_ie)
via
proc_to_coord
grs8
reset_all_timers
Release_conf
(CR, rc)
via
proc_to_coord

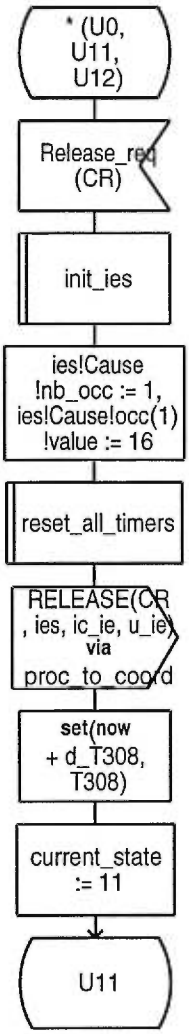

U11
STATUS (CR, ie, ic_ie, u_ie)
status_verif (ie, ic_ie, u_ie st, diag, error_code)
error_code
MIEM
MIECE
UIE
OK
init_ies
init_ies
send_st
grs11
ies!Cause !occ(1) !value:= 96
ies!Cause !occ(1) !value:= 100
true
false
grs9
grs9
init_ies
grs11
ies!Cause !nb_occ := 1
ies!Cause !occ(1) !value := 99
ie !Call_state !code /= 0
es!Cause!occ(1) !IE_id:= diag
ies!Cause !nb_occ := 1, ies!Cause!occ (1)!IE_id := diag
true
false
ies!Call_state !presence:= true
ies !Call_state !presence := true
U11
grs10
ies !Call_state !code := 11
lbl_u11
STATUS(CR, ies, ic_ie, u_ie) via proc_to_coord
U11

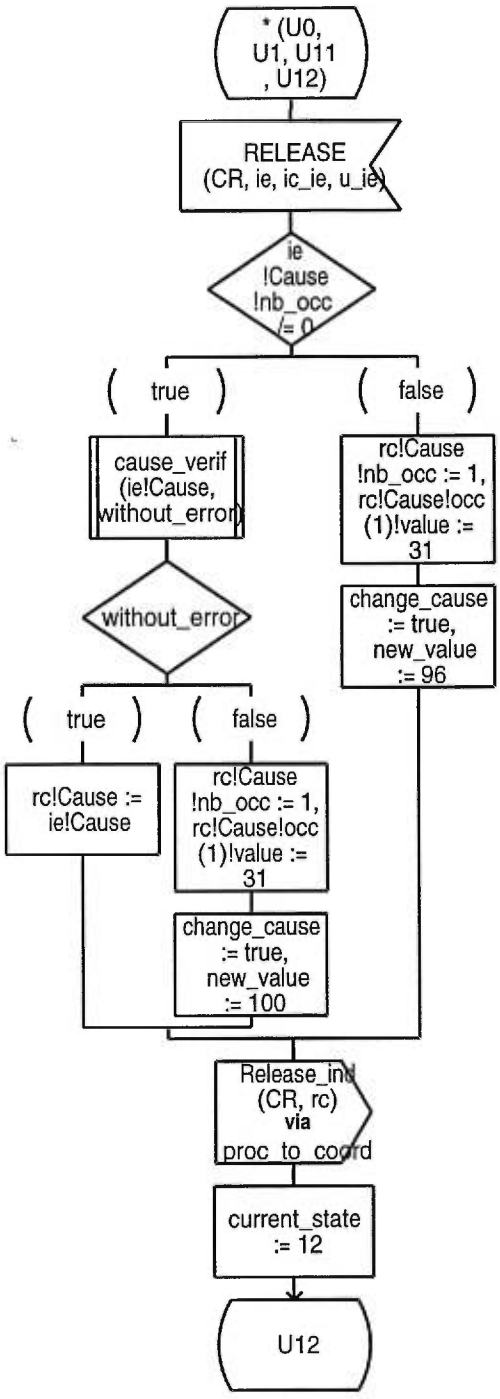

bl_u1
ie
!Call_state
!code /=
0
true
false
ies
!Call_state
!code :=
11
STATUS(CR,
ies, ic_ie, u_ie)
via
proc_to_coord
U11
ies
!Call_state
!code :=
0
STATUS(CR,
ies, ic_ie, u_ie)
via
proc_to_coord
grs10
rc!Cause
!nb_occ := 1,
rc!Cause!occ
(1)!value :=
101
reset_all_timers
Release_conf
(CR, rc)
via
proc_to_coord
U11
RELEASE
(CR,
ie, ic_ie, u_ie)
reset(T308)
ie
!Cause
!nb_occ
/= 0
true
false
cause_verif
(ie!Cause,
without_error)
without_error
true
false
rc!Cause
:= ie
!Cause
rc!Cause
!nb_occ := 1,
rc!Cause!occ
(1)!value :=
31
rc!Cause
!nb_occ := 1,
rc!Cause!occ
(1)!value :=
31
Release_conf
(CR, rc)
via
proc_to_coord
T308
exp_T308
:=
exp_T308
+ 1
exp_T308 = 2
false
true
ies
!Cause
!nb_occ
:= 2
ies!Cause
!occ(2)
!value :=
102
RELEASE(CR
, ies, ic_ie, u_ie)
via
proc_to_coord
set(now
+ d_T308,
T308)
U11
rc!Cause
!nb_occ
:= 1
rc!Cause
!occ(1)
!value :=
102
rc!Cause
!occ(1)
!char1 :=
'3'
rc!Cause
!occ(1)
!char1 :=
'0'
rc!Cause
!occ(1)
!char1 :=
'8'
Release_conf
(CR, rc)
via
proc_to_coord

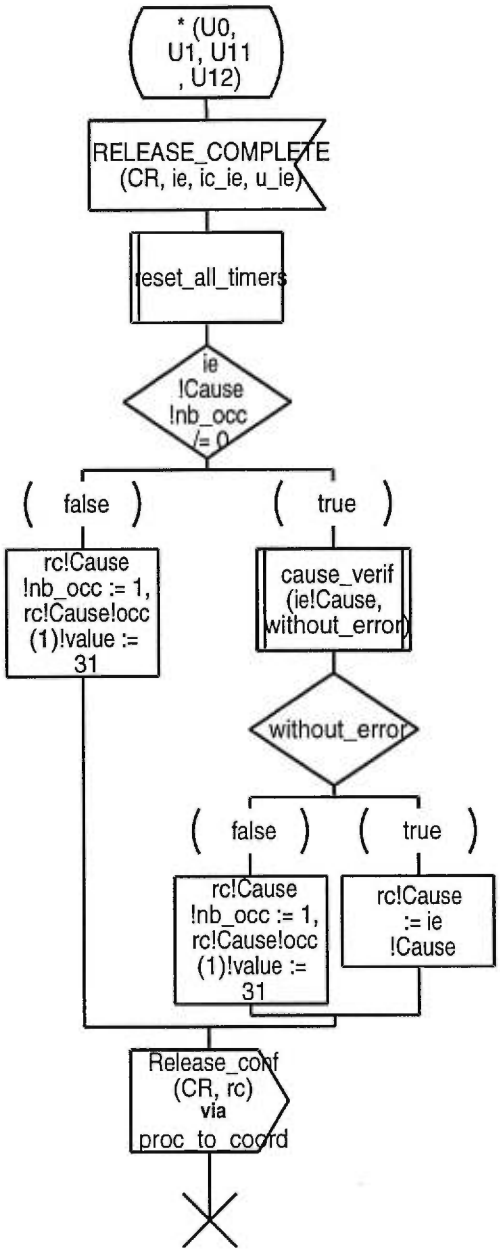

U12
Release_resp
(CR, r)
RELEASE(CR
, ie, ic_ie, u_ie),
RELEASE_COMPLETE
(CR, ie, ic_ie, u_ie)
change_cause
-
true
false
r!Cause!occ(1)
!value := new_value,
r!Cause!occ(1)
!IE_id!nb_occ := 1,
r!Cause!occ(1)
!IE_id!occ(1) := 8
change_cause
:= false
init_ies
ies
!Cause
:= r
!Cause
reset_all_timers
RELEASE_COMPLETE
(CR, ies, ic_ie, u_ie)
via
proc_to_coord

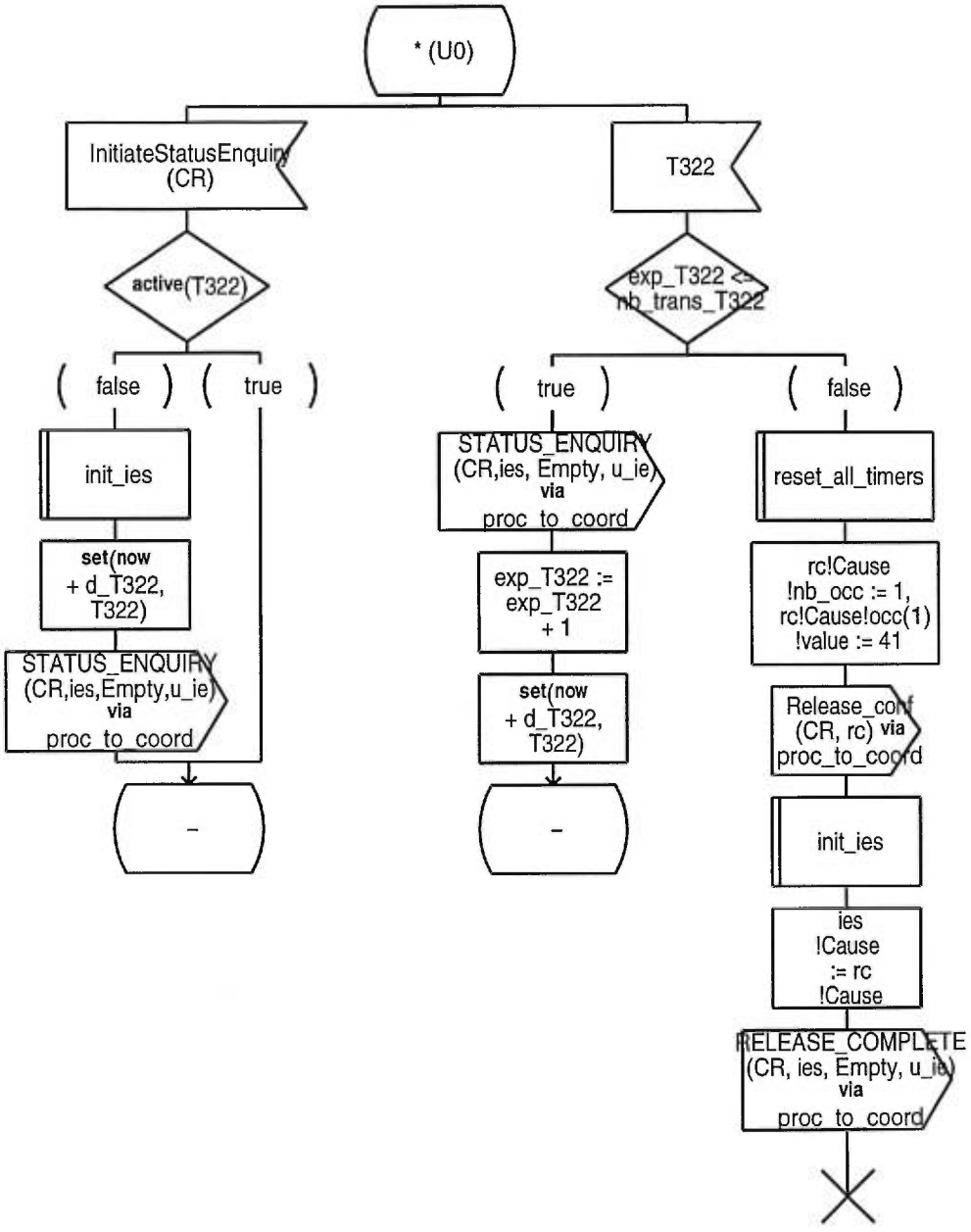

U12
STATUS (CR, ie, ic_ie, u_ie)
status_verif (ie, ic_ie, u_ie st, diag, error_code)
error_code
MIEM
MIECE
UIE
OK
init_ies
init_ies
send_st
LblOK
ies!Cause !occ(1) !value := 96
ies!Cause !occ(1) !value := 100
true
false
grs12
grs12
init_ies
LblOK
ies !Cause !nb_occ := 1
ies!Cause !occ(1) !IE_id := diag
ies !Call_state !presence := true
ies !Call_state !code := 12
STATUS(CR, ies, ic_ie, u_ie) via proc_to_coord
U12
ies!Cause !occ(1) !value := 99
ies!Cause !nb_occ := 1, ies!Cause!occ (1)!IE_id := diag
ies !Call_state !presence := true
lbl99
ie !Call_state !code /= 0
false
true
Lbl99
U12

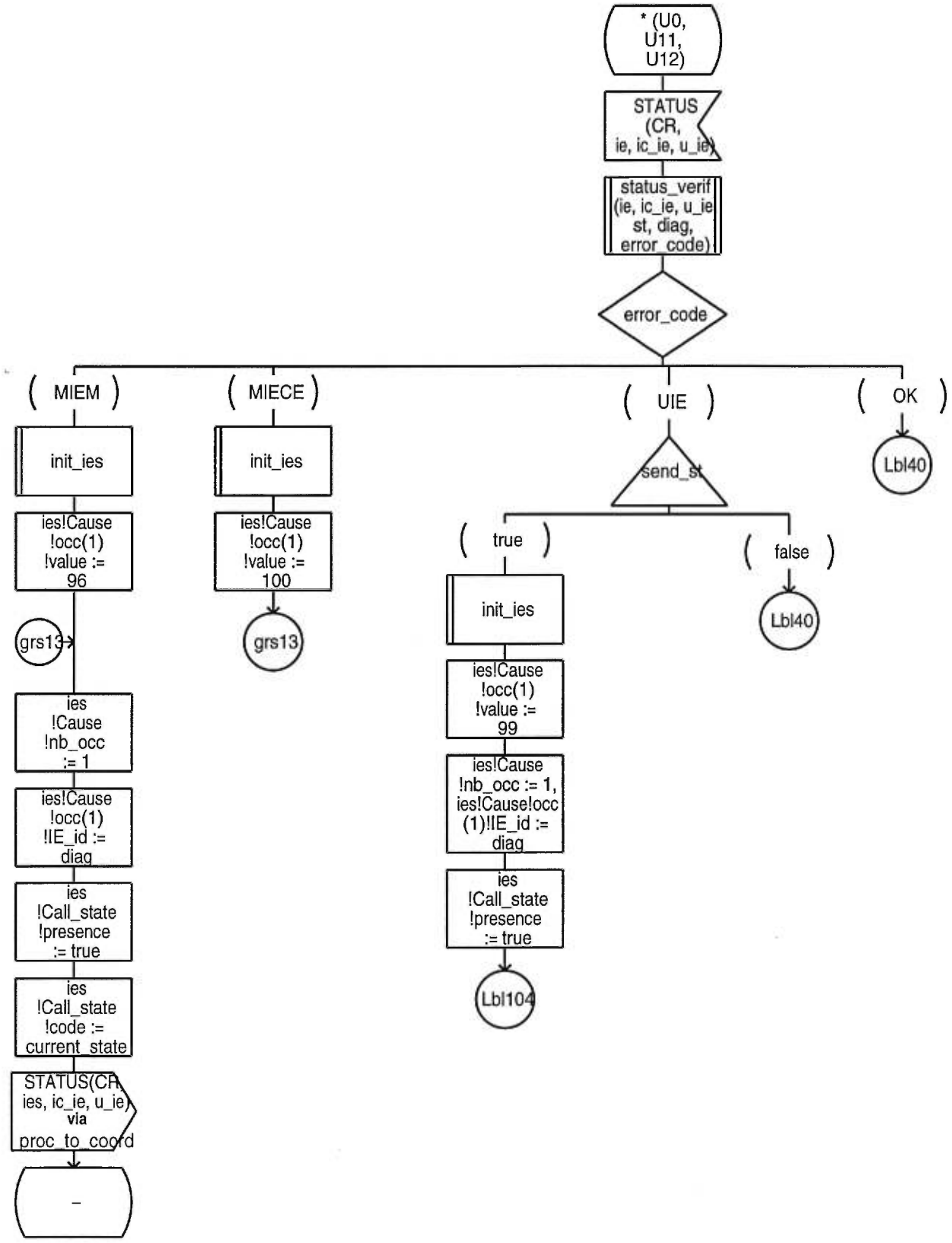

lbl99
ie
!Call_state
!code /=
0
true
false
ies
!Call_state
!code :=
12
ies
!Call_state
!code
:= 0
STATUS(CR
ies, ic_ie, u_ie)
via
proc_to_coord
STATUS(CR
ies, ic_ie, u_ie)
via
proc_to_coord
U12
rc!Cause
!nb_occ
:= 1
rc!Cause
!occ(1)
!value := 101
reset_all_timers
Release_conf
(CR, rc)
via
proc_to_coord

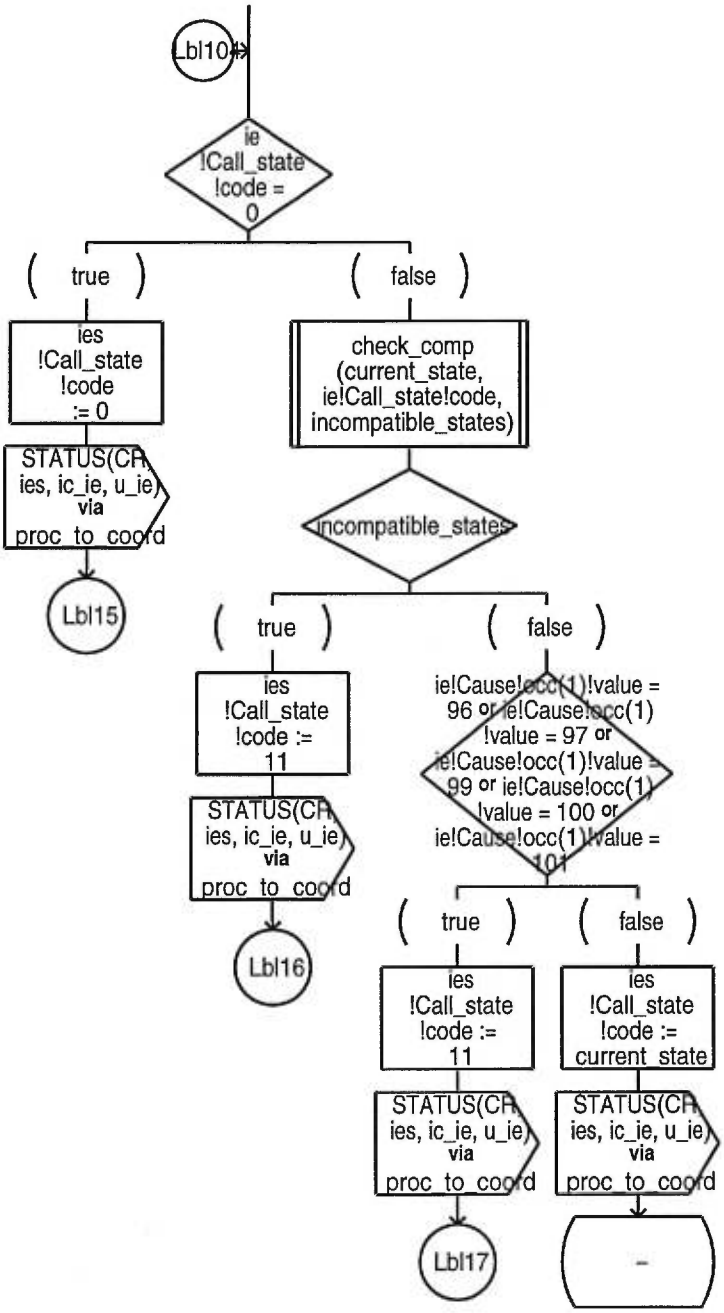

* (U0,
U11,
U12)
Release_req
(CR)
init_ies
ies!Cause
!nb_occ := 1,
ies!Cause!occ(1)
!value := 16
reset_all_timers
RELEASE(CR
, ies, ic_ie, u_ie)
via
proc_to_coord
set(now
+ d_T308,
T308)
current_state
:= 11
U11

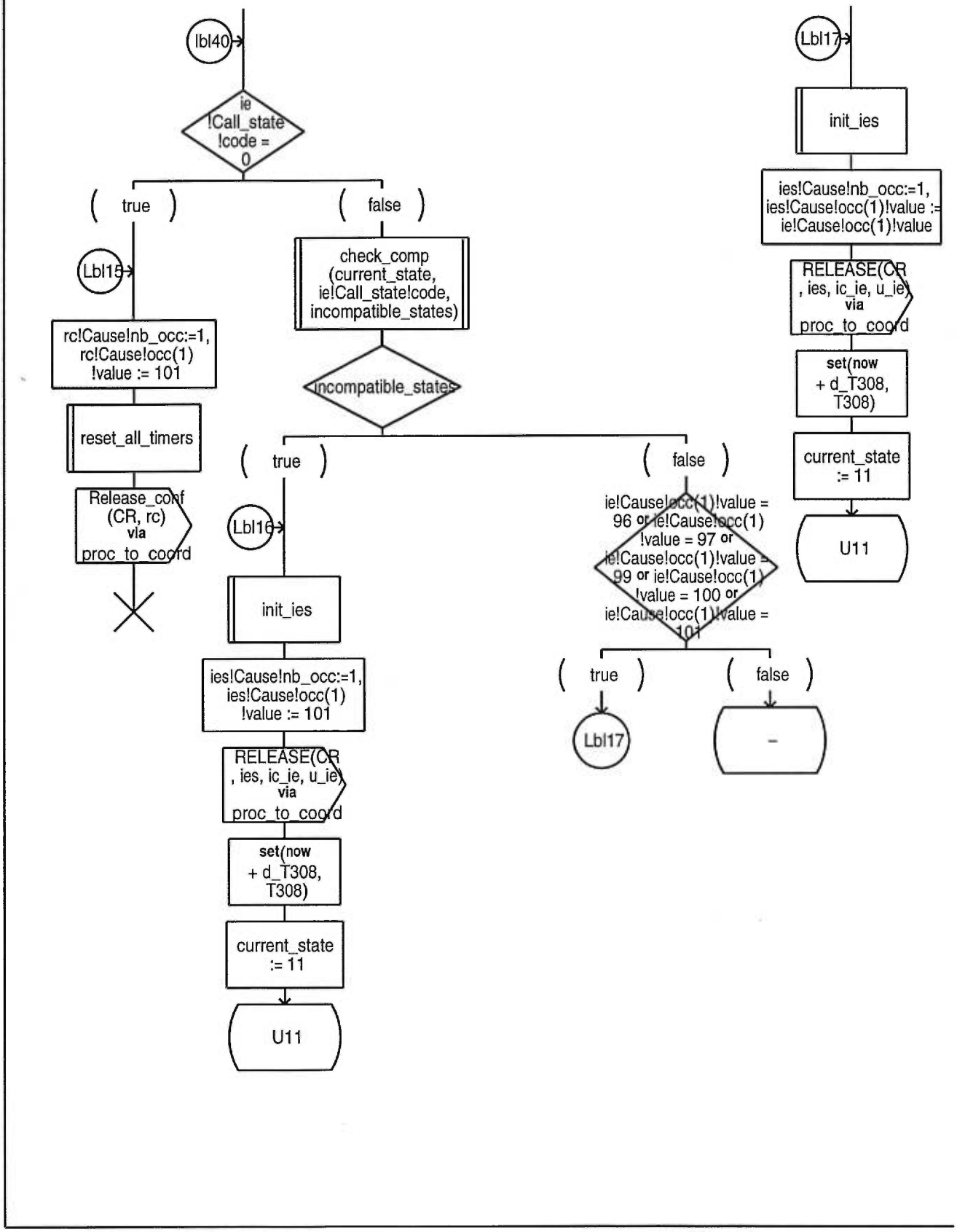

* (U0, U1, U11 , U12)
RELEASE (CR, ie, ic_ie, u_ie)
ie !Cause !nb_occ /= 0
true
false
cause_verif (ie!Cause, without_error)
rc!Cause !nb_occ := 1, rc!Cause!occ (1)!value := 31
without_error
change_cause := true, new_value := 96
true
false
rc!Cause := ie!Cause
rc!Cause !nb_occ := 1, rc!Cause!occ (1)!value := 31
change_cause := true, new_value := 100
Release_ind (CR, rc) via proc_to_coord
current_state := 12
U12

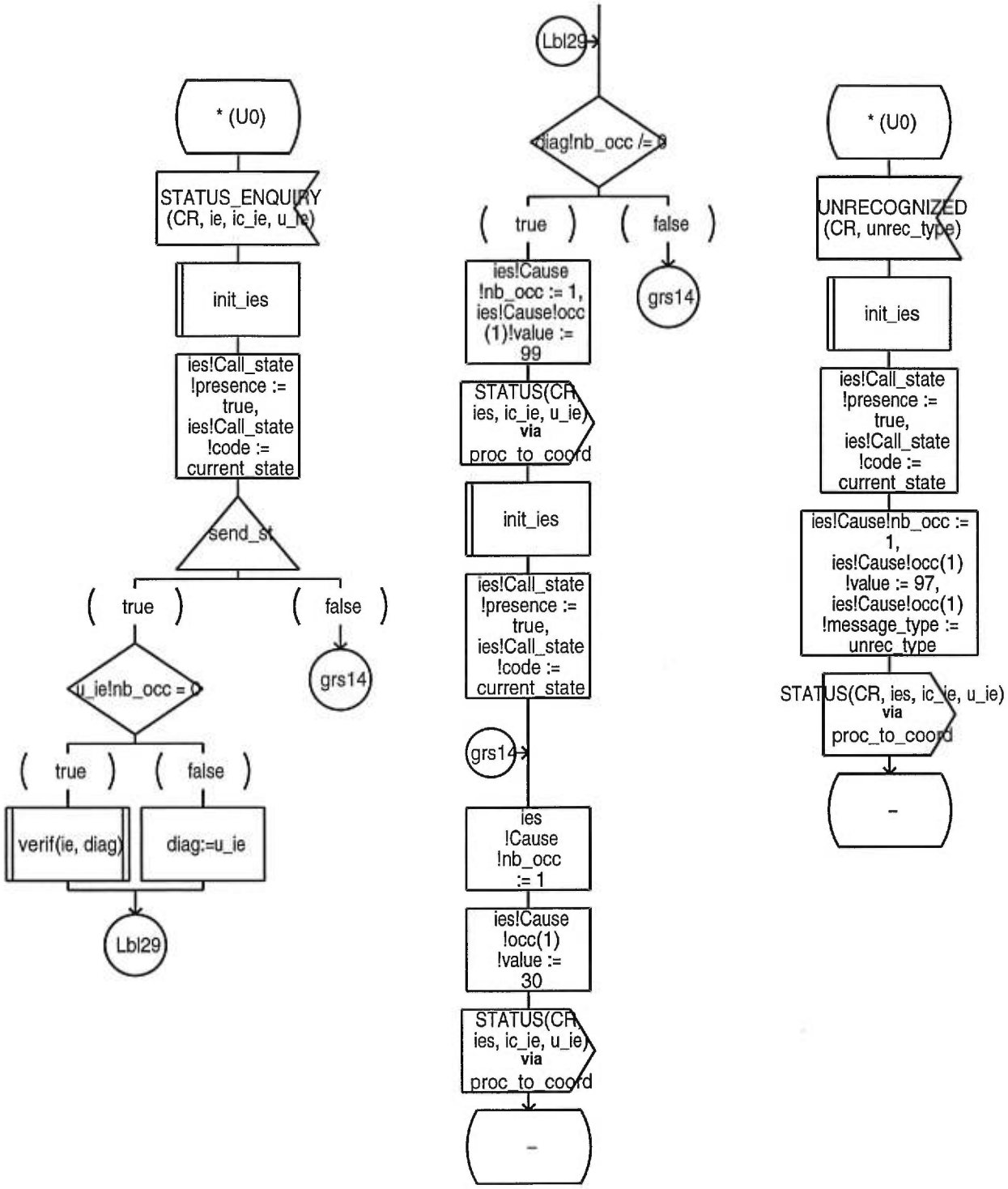

* (U0, U1, U11 , U12)
RELEASE_COMPLETE (CR, ie, ic_ie, u_ie)
reset_all_timers
ie !Cause !nb_occ /= 0
false
true
rc!Cause !nb_occ := 1, rc!Cause!occ (1)!value := 31
cause_verif (ie!Cause, without_error)
without_error
false
true
rc!Cause !nb_occ := 1, rc!Cause!occ (1)!value := 31
rc!Cause := ie !Cause
Release_conf (CR, rc) via proc_to_coord

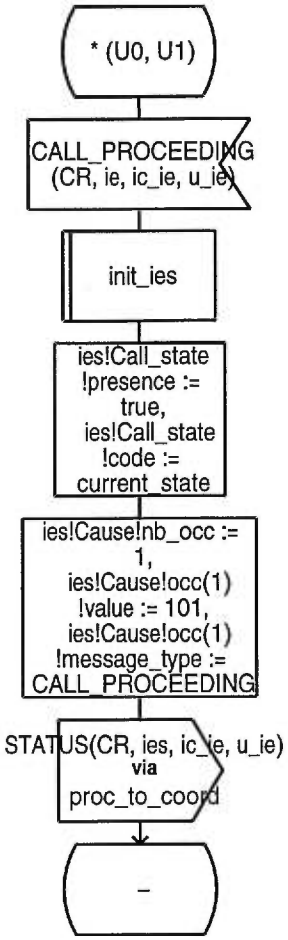

* (U0)
InitiateStatusEnquiry
(CR)
T322
active(T322)
exp_T322 <>
nb_trans_T322
false
true
true
false
init_ies
set(now
+ d_T322,
T322)
STATUS_ENQUIRY
(CR,ies,Empty,u_ie)
via
proc_to_coord
-
STATUS_ENQUIRY
(CR,ies, Empty, u_ie)
via
proc_to_coord
exp_T322 :=
exp_T322
+ 1
set(now
+ d_T322,
T322)
-
reset_all_timers
rc!Cause
!nb_occ := 1,
rc!Cause!occ(1)
!value := 41
Release_conf
(CR, rc) via
proc_to_coord
init_ies
ies
!Cause
:= rc
!Cause
RELEASE_COMPLETE
(CR, ies, Empty, u_ie)
via
proc_to_coord

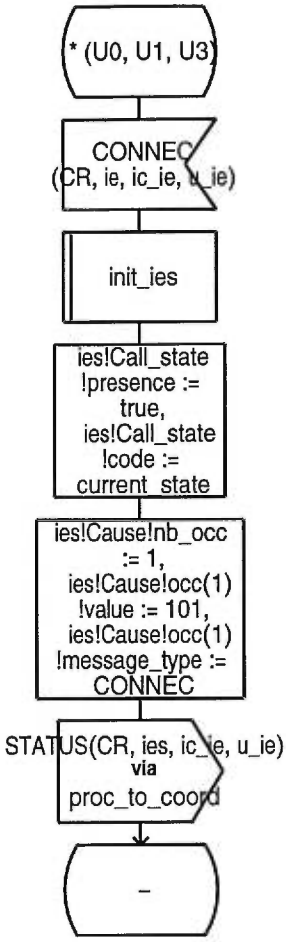

* (U0, U11, U12)
STATUS (CR, ie, ic_ie, u_ie)
status_verif (ie, ic_ie, u_ie st, diag, error_code)
error_code
MIEM
MIECE
UIE
OK
init_ies
ies!Cause !occ(1) !value := 96
grs13
ies !Cause !nb_occ := 1
ies!Cause !occ(1) !IE_id := diag
ies !Call_state !presence := true
ies !Call_state !code := current_state
STATUS(CR, ies, ic_ie, u_ie) via proc_to_coord
init_ies
ies!Cause !occ(1) !value := 100
grs13
send_st
true
false
init_ies
ies!Cause !occ(1) !value := 99
ies!Cause !nb_occ := 1, ies!Cause!occ (1)!IE_id := diag
ies !Call_state !presence := true
Lbl104
Lbl40
Lbl40

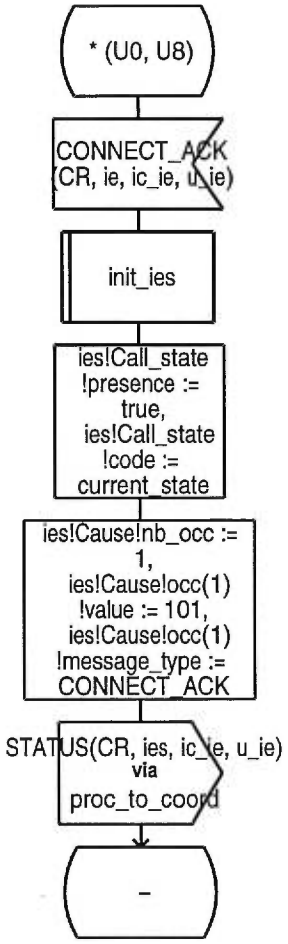

Lbl10
ie
!Call_state
!code =
0
true
false
ies
!Call_state
!code
:= 0
STATUS(CH
ies, ic_ie, u_ie)
via
proc_to_coord
Lbl15
check_comp
(current_state,
ie!Call_state!code,
incompatible_states)
incompatible_states
true
false
ies
!Call_state
!code :=
11
STATUS(CH
ies, ic_ie, u_ie)
via
proc_to_coord
Lbl16
ie!Cause!occ(1)!value =
96 or ie!Cause!occ(1)
!value = 97 or
ie!Cause!occ(1)!value =
99 or ie!Cause!occ(1)
!value = 100 or
ie!Cause!occ(1)!value =
101
true
false
ies
!Call_state
!code :=
11
ies
!Call_state
!code :=
current_state
STATUS(CH
ies, ic_ie, u_ie)
via
proc_to_coord
STATUS(CH
ies, ic_ie, u_ie)
via
proc_to_coord
Lbl17
-

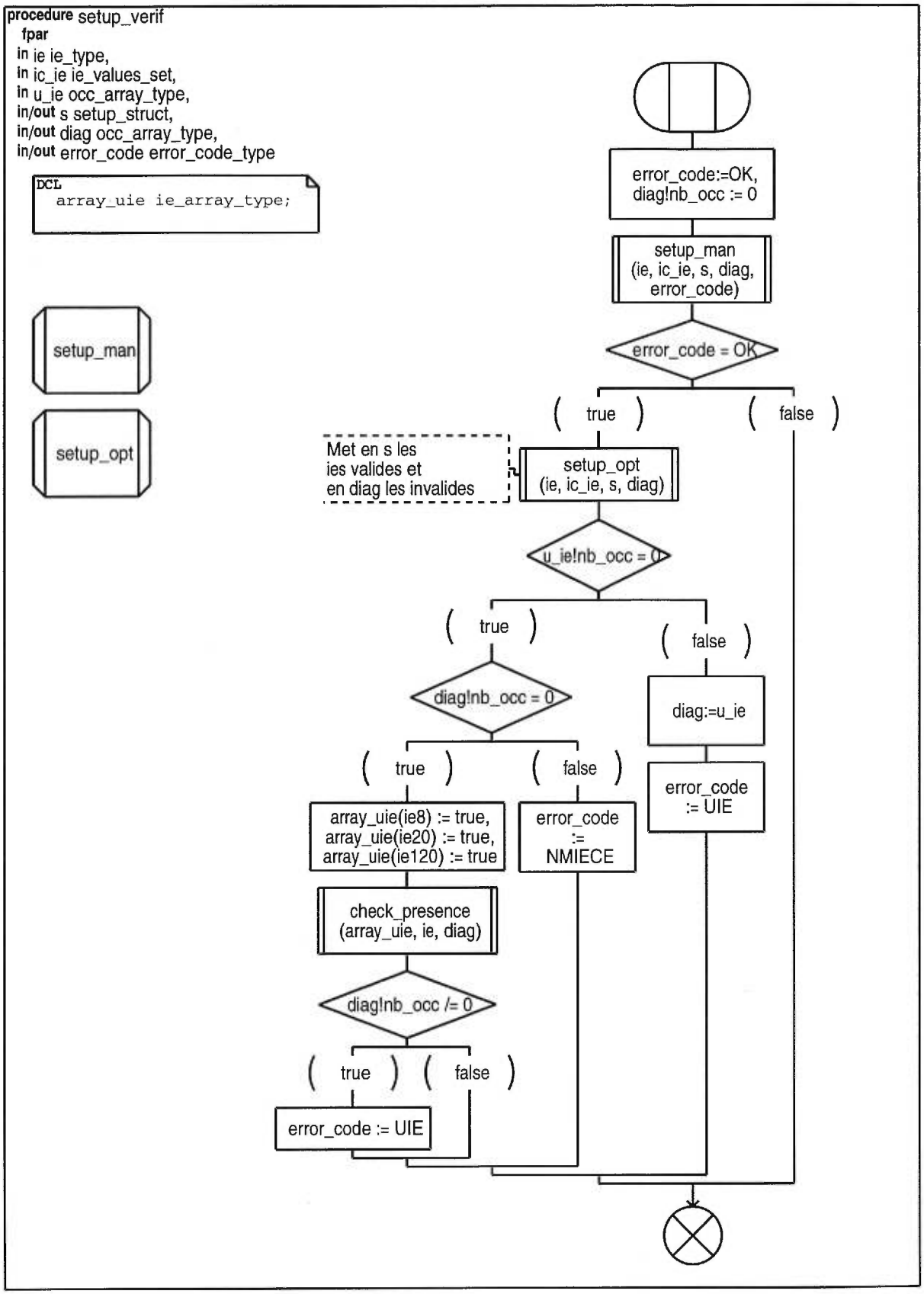

lbl40
ie !Call_state !code = 0
true
false
Lbl15
rc!Cause!nb_occ:=1, rc!Cause!occ(1) !value := 101
reset_all_timers
Release_conf (CR, rc) via proc_to_coord
check_comp (current_state, ie!Call_state!code, incompatible_states)
incompatible_states
true
false
Lbl16
init_ies
ies!Cause!nb_occ:=1, ies!Cause!occ(1) !value := 101
RELEASE(CR , ies, ic_ie, u_ie) via proc_to_coord
set(now + d_T308, T308)
current_state := 11
U11
ie!Cause!occ(1)!value = 96 or ie!Cause!occ(1) !value = 97 or ie!Cause!occ(1)!value = 99 or ie!Cause!occ(1) !value = 100 or ie!Cause!occ(1)!value = 101
true
false
Lbl17
-
Lbl17
init_ies
ies!Cause!nb_occ:=1, ies!Cause!occ(1)!value := ie!Cause!occ(1)!value
RELEASE(CR , ies, ic_ie, u_ie) via proc_to_coord
set(now + d_T308, T308)
current_state := 11
U11

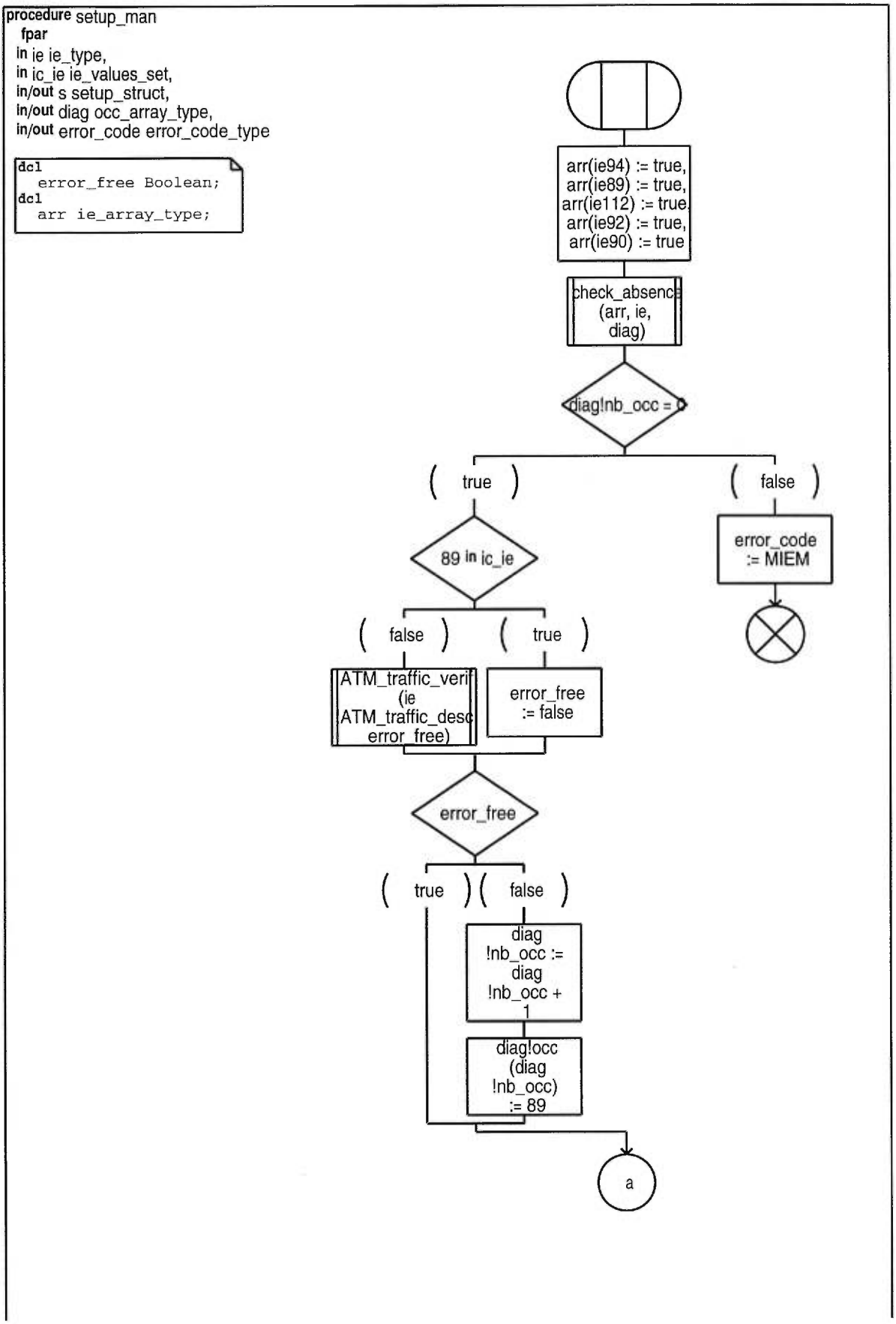

* (U0)
STATUS_ENQUIRY (CR, ie, ic_ie, u_ie)
init_ies
ies!Call_state !presence := true, ies!Call_state !code := current_state
send_st
true
false
u_ie!nb_occ = 0
grs14
true
false
verif(ie, diag)
diag:=u_ie
Lbl29
Lbl29
diag!nb_occ /= 0
true
false
ies!Cause !nb_occ := 1, ies!Cause!occ (1)!value := 99
grs14
STATUS(CR, ies, ic_ie, u_ie) via proc_to_coord
init_ies
ies!Call_state !presence := true, ies!Call_state !code := current_state
grs14
ies !Cause !nb_occ := 1
ies!Cause !occ(1) !value := 30
STATUS(CR, ies, ic_ie, u_ie) via proc_to_coord
-
* (U0)
UNRECOGNIZED (CR, unrec_type)
init_ies
ies!Call_state !presence := true, ies!Call_state !code := current_state
ies!Cause!nb_occ := 1, ies!Cause!occ(1) !value := 97, ies!Cause!occ(1) !message_type := unrec_type
STATUS(CR, ies, ic_ie, u_ie) via proc_to_coord
-

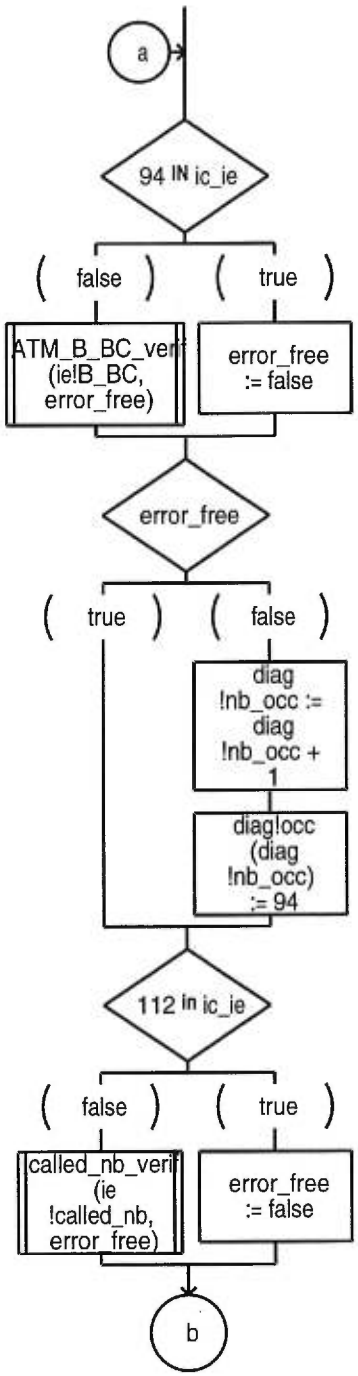

* (U0, U1)
CALL_PROCEEDING
(CR, ie, ic_ie, u_ie)
init_ies
ies!Call_state
!presence :=
true,
ies!Call_state
!code :=
current_state
ies!Cause!nb_occ :=
1,
ies!Cause!occ(1)
!value := 101,
ies!Cause!occ(1)
!message_type :=
CALL_PROCEEDING
STATUS(CR, ies, ic_ie, u_ie)
via
proc_to_coord
-

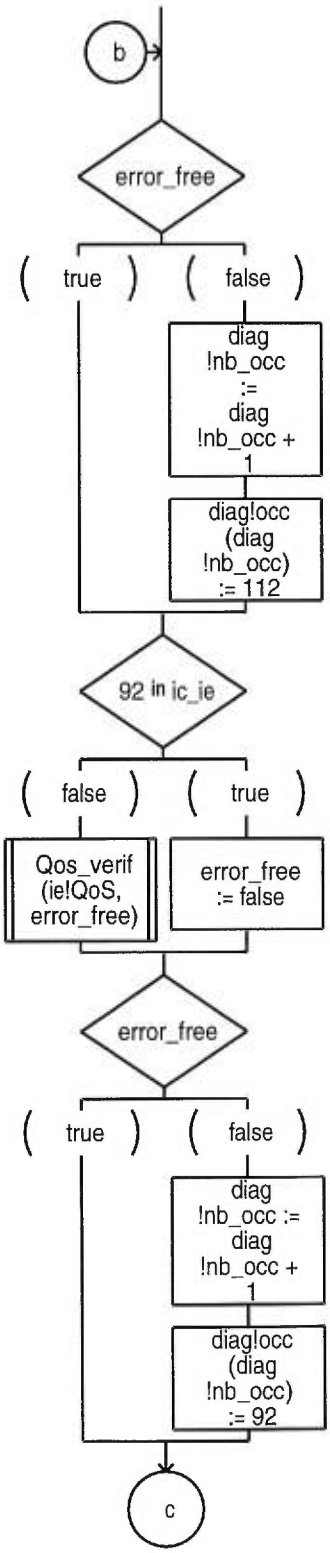

* (U0, U1, U3)
CONNEC
(CR, ie, ic_ie, u_ie)
init_ies
ies!Call_state
!presence :=
true,
ies!Call_state
!code :=
current_state
ies!Cause!nb_occ
:= 1,
ies!Cause!occ(1)
!value := 101,
ies!Cause!occ(1)
!message_type :=
CONNEC
STATUS(CR, ies, ic_ie, u_ie)
via
proc_to_coord
-

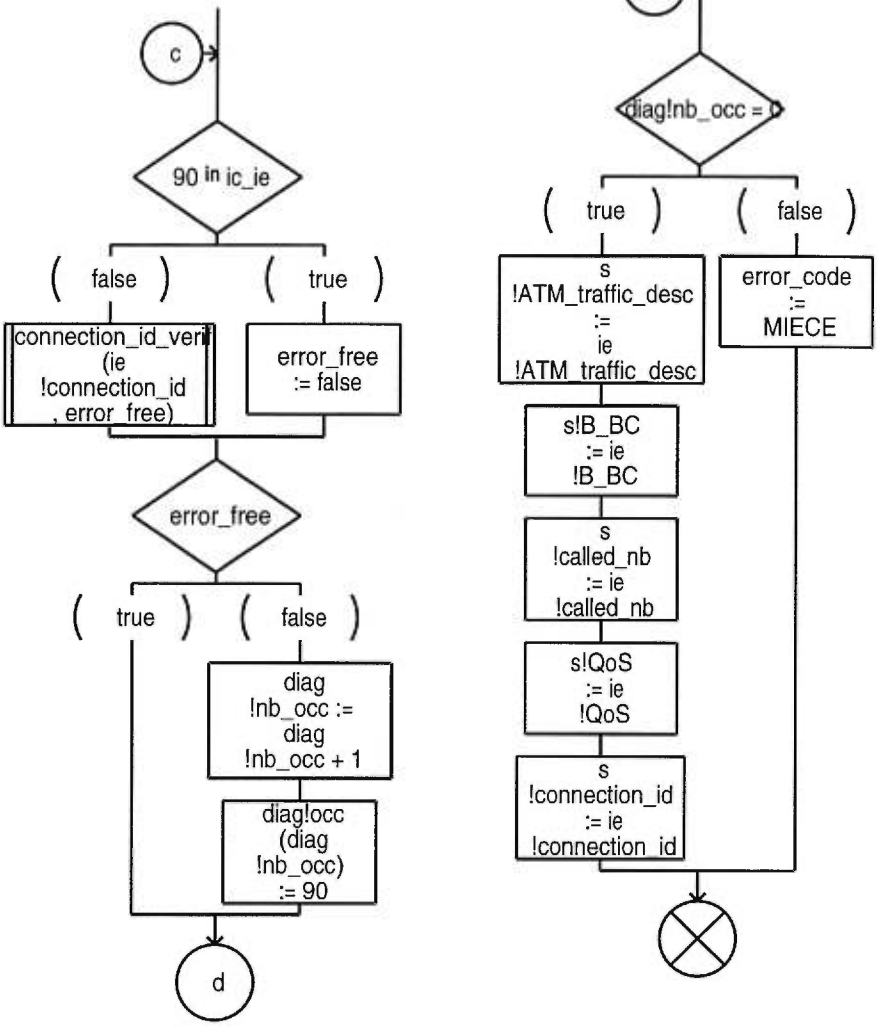

* (U0, U8)
CONNECT_ACK
(CR, ie, ic_ie, u_ie)
init_ies
ies!Call_state
!presence :=
true,
ies!Call_state
!code :=
current_state
ies!Cause!nb_occ :=
1,
ies!Cause!occ(1)
!value := 101,
ies!Cause!occ(1)
!message_type :=
CONNECT_ACK
STATUS(CR, ies, ic_ie, u_ie)
via
proc_to_coord
-


procedure setup_verif
fpar
in ie ie_type,
in ic_ie ie_values_set,
in u_ie occ_array_type,
in/out s setup_struct,
in/out diag occ_array_type,
in/out error_code error_code_type
DCL
array_uie ie_array_type;
setup_man
setup_opt
error_code:=OK,
diag!nb_occ := 0
setup_man
(ie, ic_ie, s, diag,
error_code)
error_code = OK
true
false
Met en s les
ies valides et
en diag les invalides
setup_opt
(ie, ic_ie, s, diag)
u_ie!nb_occ = 0
true
false
diag!nb_occ = 0
diag:=u_ie
true
false
error_code
:= UIE
array_uie(ie8) := true,
array_uie(ie20) := true,
array_uie(ie120) := true
error_code
:=
NMIECE
check_presence
(array_uie, ie, diag)
diag!nb_occ /= 0
true
false
error_code := UIE


procedure setup_man
fpar
in ie ie_type,
in ic_ie ie_values_set,
in/out s setup_struct,
in/out diag occ_array_type,
in/out error_code error_code_type
dcl
error_free Boolean;
dcl
arr ie_array_type;
arr(ie94) := true,
arr(ie89) := true,
arr(ie112) := true,
arr(ie92) := true,
arr(ie90) := true
check_absence
(arr, ie,
diag)
diag!nb_occ = 0
true
false
error_code
:= MIEM
89 in ic_ie
false
true
ATM_traffic_verif
(ie
ATM_traffic_desc
error_free)
error_free
:= false
error_free
true
false
diag
!nb_occ :=
diag
!nb_occ +
1
diag!occ
(diag
!nb_occ)
:= 89
a

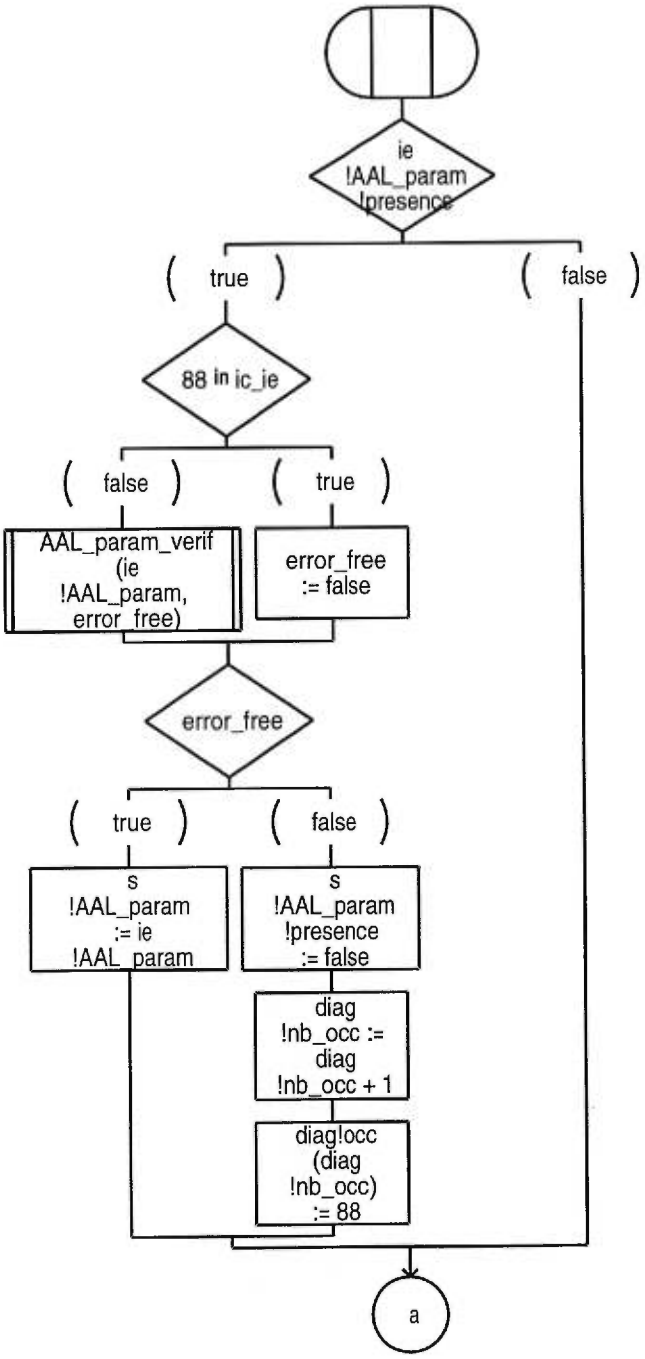

a
94 IN ic_ie
false
true
ATM_B_BC_verif (ie!B_BC, error_free)
error_free := false
error_free
true
false
diag !nb_occ := diag !nb_occ + 1
diag!occ (diag !nb_occ) := 94
112 in ic_ie
false
true
called_nb_verif (ie !called_nb, error_free)
error_free := false
b

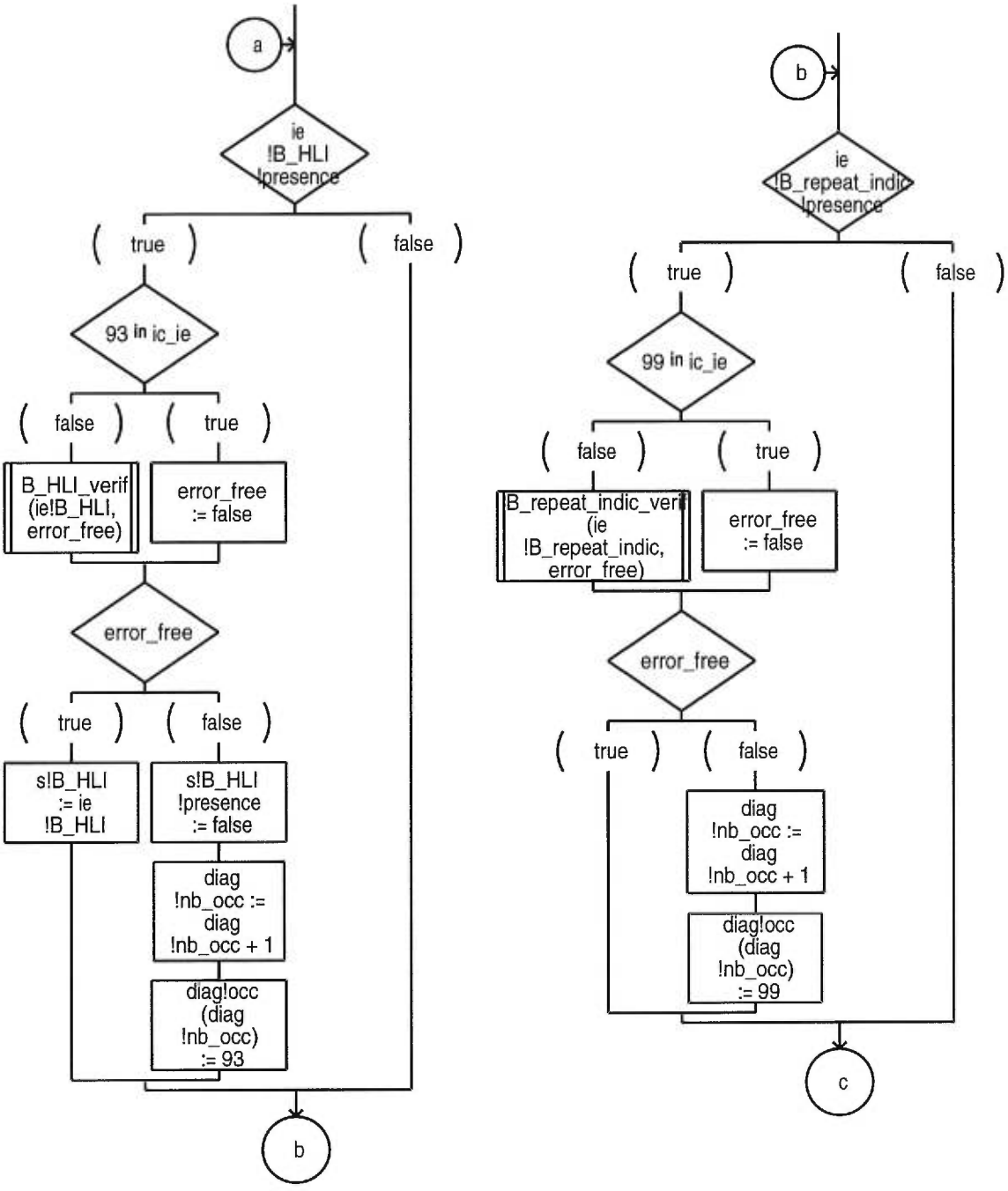

b
error_free
true
false
diag !nb_occ := diag !nb_occ + 1
diag!occ (diag !nb_occ) := 112
92 in ic_ie
false
true
Qos_verif (ie!QoS, error_free)
error_free := false
error_free
true
false
diag !nb_occ := diag !nb_occ + 1
diag!occ (diag !nb_occ) := 92
c

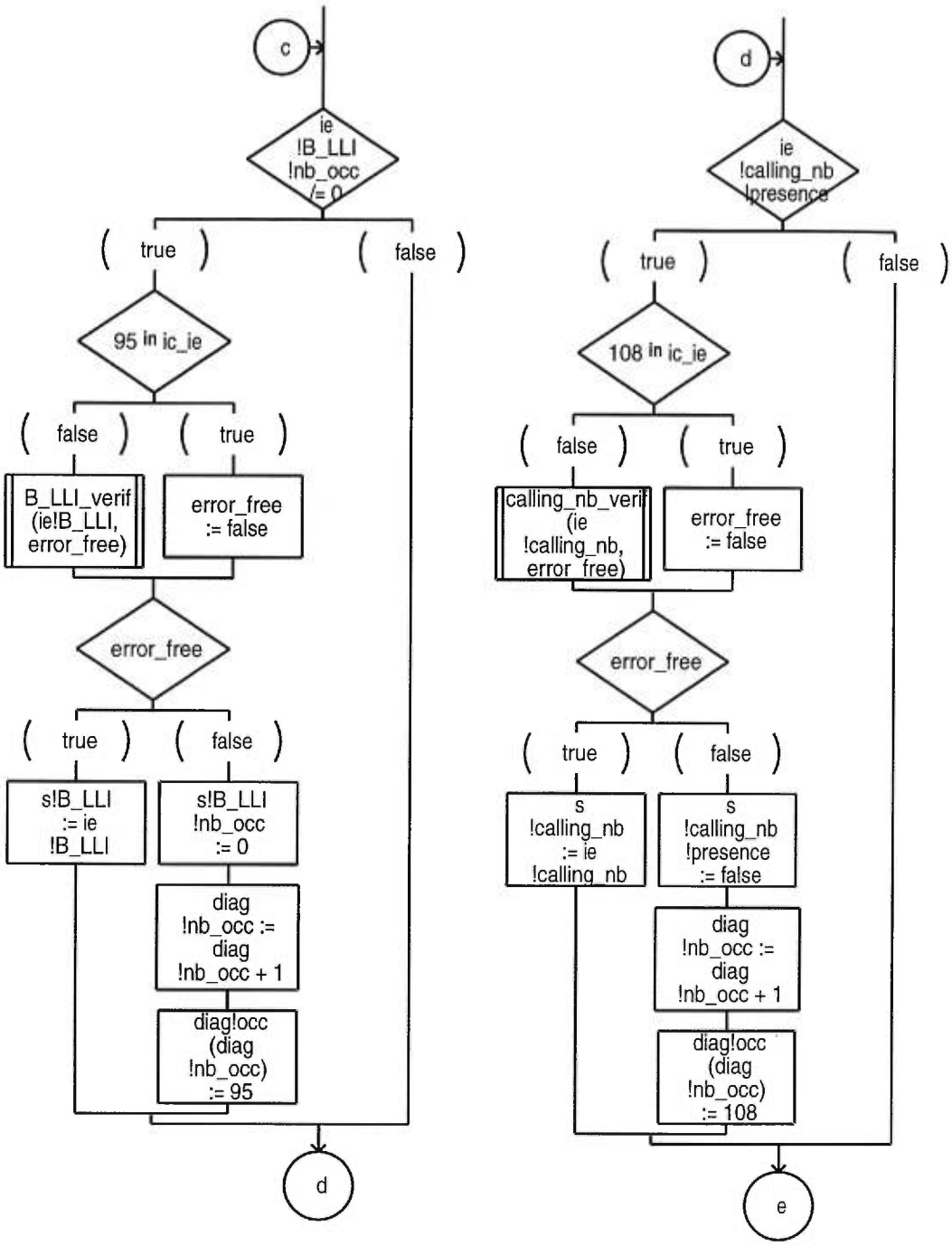

c
90 in ic_ie
false
true
connection_id_verif
(ie
!connection_id
, error_free)
error_free
:= false
error_free
true
false
diag
!nb_occ :=
diag
!nb_occ + 1
diag!occ
(diag
!nb_occ)
:= 90
d
d
diag!nb_occ = 0
true
false
s
!ATM_traffic_desc
:=
ie
!ATM_traffic_desc
error_code
:=
MIECE
s!B_BC
:= ie
!B_BC
s
!called_nb
:= ie
!called_nb
s!QoS
:= ie
!QoS
s
!connection_id
:= ie
!connection_id

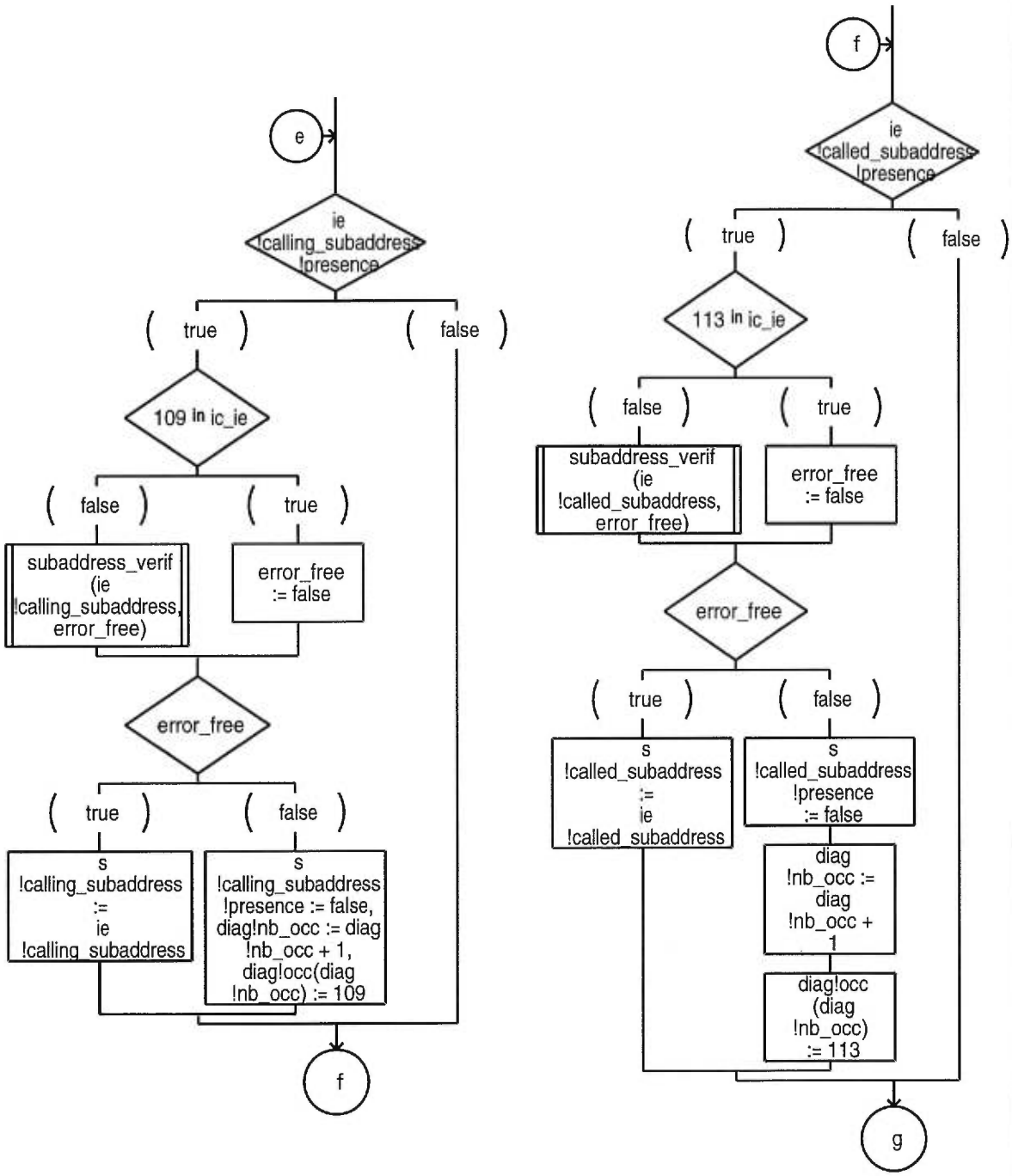
```
procedure setup_opt
  fpar
in ie ie_type,
in ic_ie ie_values_set,
in/out s setup_struct,
in/out diag occ_array_type
```

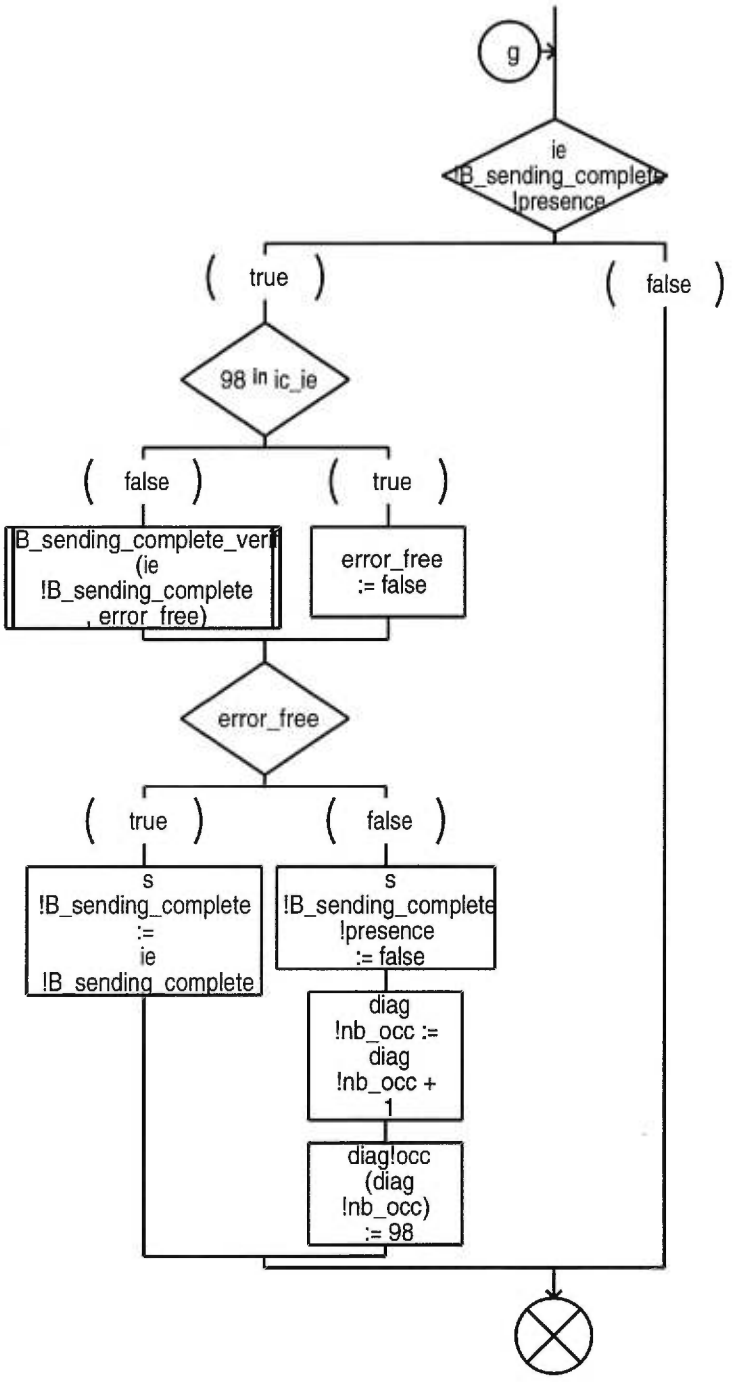
```
dcl
  error_free Boolean;
```




a
ie
!B_HLI
!presence
true
false
93 in ic_ie
false
true
B_HLI_verif
(ie!B_HLI,
error_free)
error_free
:= false
error_free
true
false
s!B_HLI
:= ie
!B_HLI
s!B_HLI
!presence
:= false
diag
!nb_occ :=
diag
!nb_occ + 1
diag!occ
(diag
!nb_occ)
:= 93
b
b
ie
!B_repeat_indic
!presence
true
false
99 in ic_ie
false
true
B_repeat_indic_verif
(ie
!B_repeat_indic,
error_free)
error_free
:= false
error_free
true
false
diag
!nb_occ :=
diag
!nb_occ + 1
diag!occ
(diag
!nb_occ)
:= 99
c

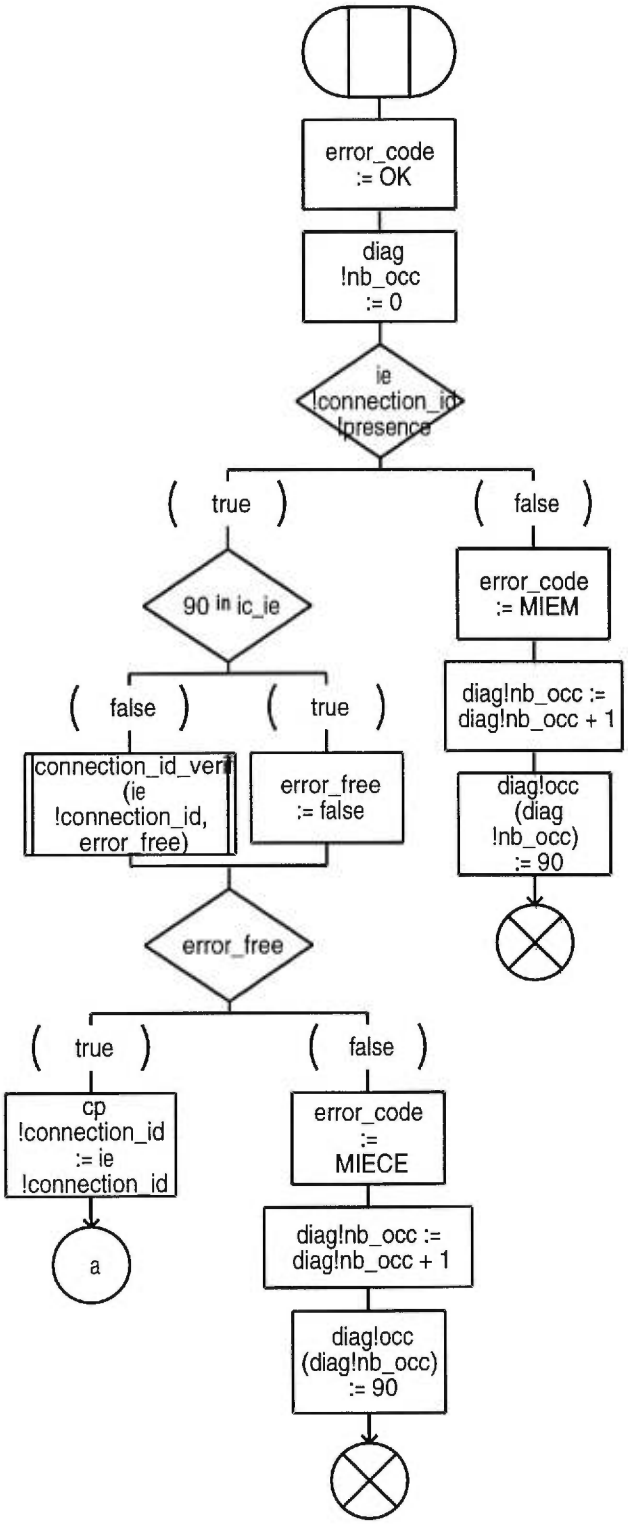

c
ie
!B_LLI
!nb_occ
/= 0
true
false
95 in ic_ie
false
true
B_LLI_verif
(ie!B_LLI,
error_free)
error_free
:= false
error_free
true
false
s!B_LLI
:= ie
!B_LLI
s!B_LLI
!nb_occ
:= 0
diag
!nb_occ :=
diag
!nb_occ + 1
diag!occ
(diag
!nb_occ)
:= 95
d
d
ie
!calling_nb
!presence
true
false
108 in ic_ie
false
true
calling_nb_verif
(ie
!calling_nb,
error_free)
error_free
:= false
error_free
true
false
s
!calling_nb
:= ie
!calling_nb
s
!calling_nb
!presence
:= false
diag
!nb_occ :=
diag
!nb_occ + 1
diag!occ
(diag
!nb_occ)
:= 108
e

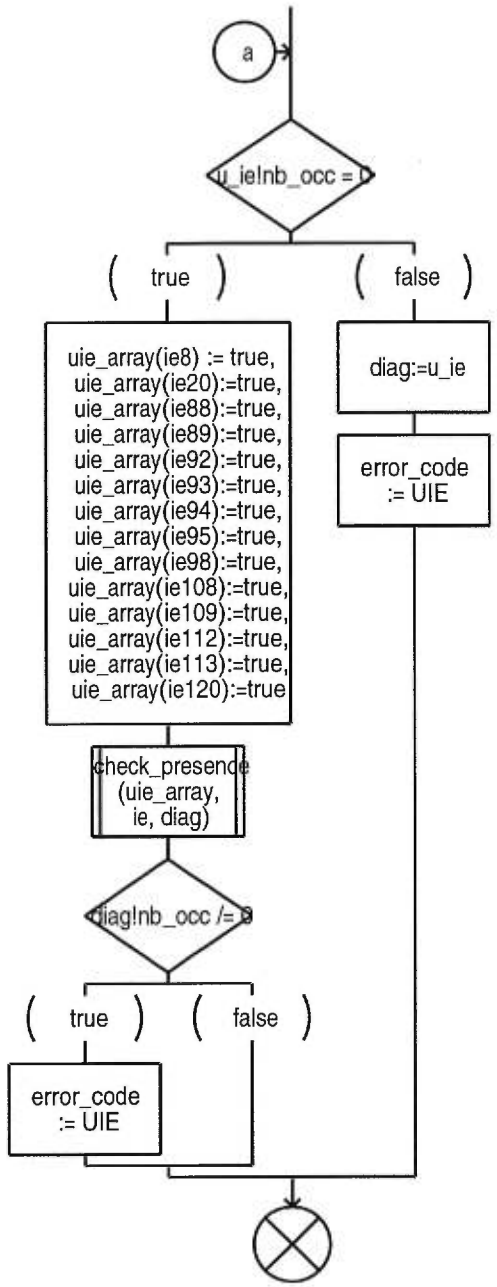

e
ie
!calling_subaddress
!presence
true
false
109 in ic_ie
false
true
subaddress_verif
(ie
!calling_subaddress,
error_free)
error_free
:= false
error_free
true
false
s
!calling_subaddress
:=
ie
!calling_subaddress
s
!calling_subaddress
!presence := false,
diag!nb_occ := diag
!nb_occ + 1,
diag!occ(diag
!nb_occ) := 109
f
f
ie
!called_subaddress
!presence
true
false
113 in ic_ie
false
true
subaddress_verif
(ie
!called_subaddress,
error_free)
error_free
:= false
error_free
true
false
s
!called_subaddress
:=
ie
!called_subaddress
s
!called_subaddress
!presence
:= false
diag
!nb_occ :=
diag
!nb_occ +
1
diag!occ
(diag
!nb_occ)
:= 113
g


g
ie
!B_sending_complete
!presence
true
false
98 in ic_ie
false
true
B_sending_complete_verif
(ie
!B_sending_complete
, error_free)
error_free
:= false
error_free
true
false
s
!B_sending_complete
:=
ie
!B_sending_complete
s
!B_sending_complete
!presence
:= false
diag
!nb_occ :=
diag
!nb_occ +
1
diag!occ
(diag
!nb_occ)
:= 98

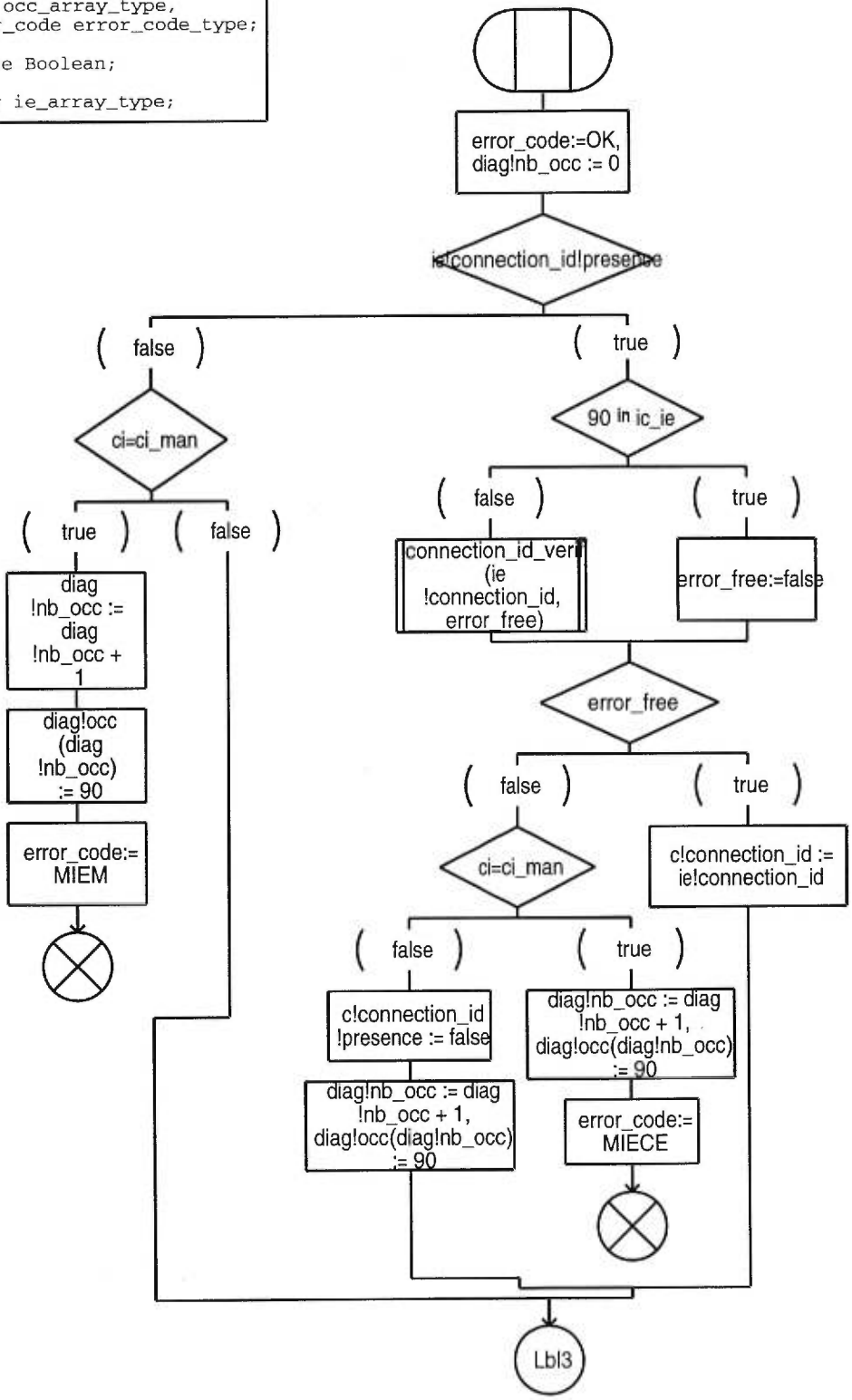
```
procedure call_pr_verif
  fpar
in ie ie_type,
in ic_ie ie_values_set,
in u_ie occ_array_type,
in/out cp call_pr_struct,
in/out diag occ_array_type,
in/out error_code error_code_type
```

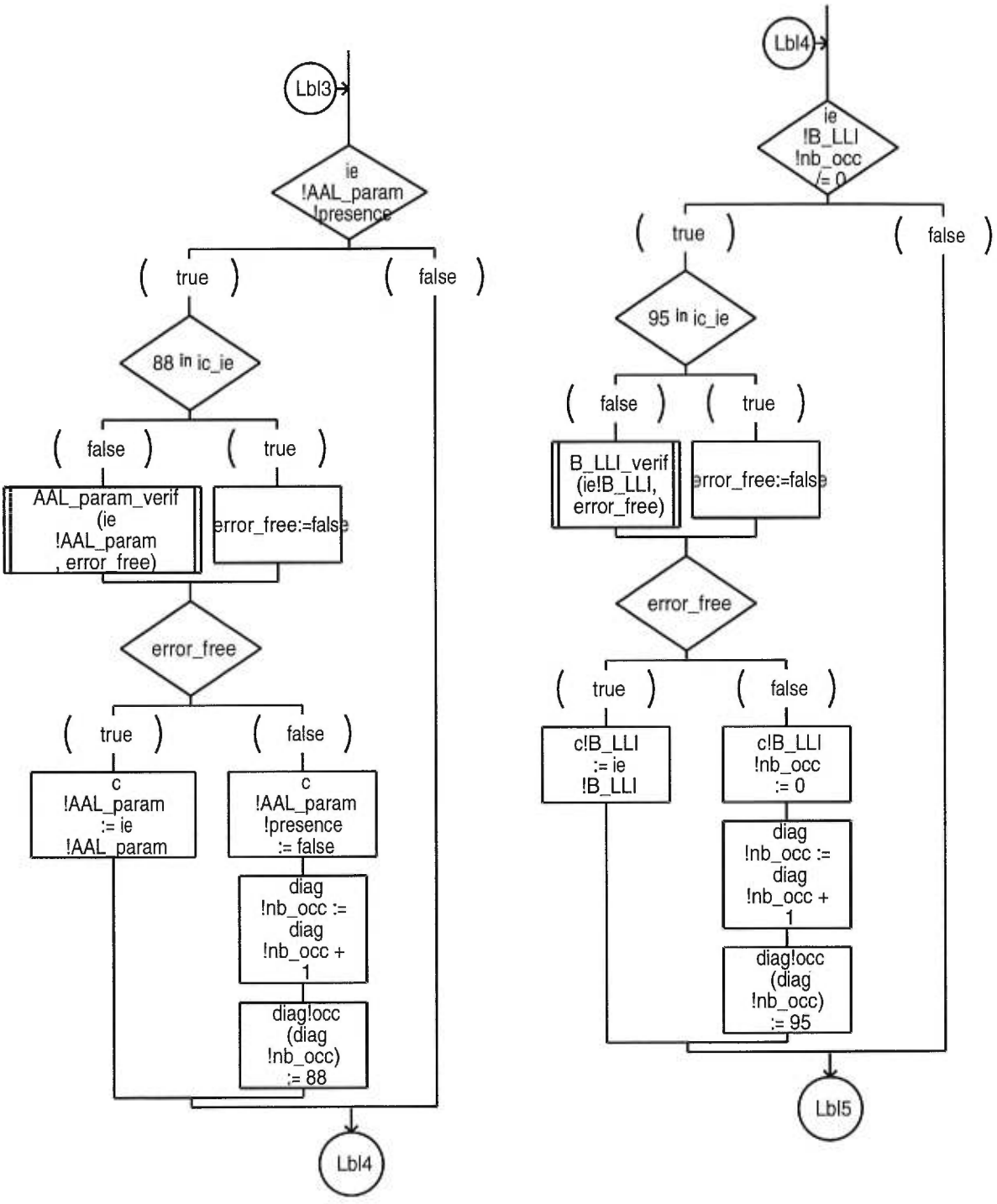
```
dcl
  error_free Boolean;
dcl
  uie_array ie_array_type;
```

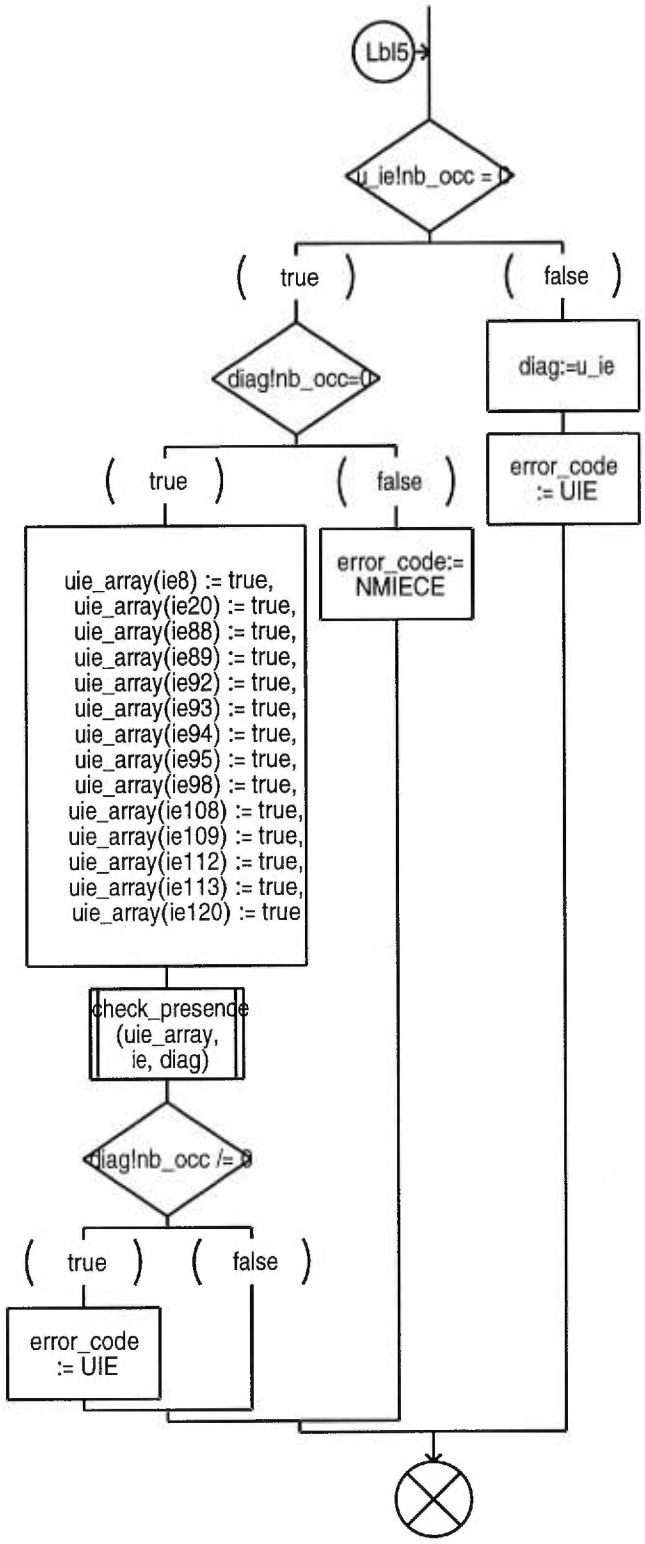



a
u_ie!nb_occ = 0
true
false
uie_array(ie8) := true,
uie_array(ie20):=true,
uie_array(ie88):=true,
uie_array(ie89):=true,
uie_array(ie92):=true,
uie_array(ie93):=true,
uie_array(ie94):=true,
uie_array(ie95):=true,
uie_array(ie98):=true,
uie_array(ie108):=true,
uie_array(ie109):=true,
uie_array(ie112):=true,
uie_array(ie113):=true,
uie_array(ie120):=true
diag:=u_ie
error_code
:= UIE
check_presence
(uie_array,
ie, diag)
diag!nb_occ /= 0
true
false
error_code
:= UIE

```
fpar
in ie ie_type,
in ic_ie ie_values_set,
in u_ie occ_array_type,
in ci ci_type,
in/out c connect_struct,
in/out diag occ_array_type,
in/out error_code error_code_type;
dcl
  error_free Boolean;
dcl
  uie_array ie_array_type;
```

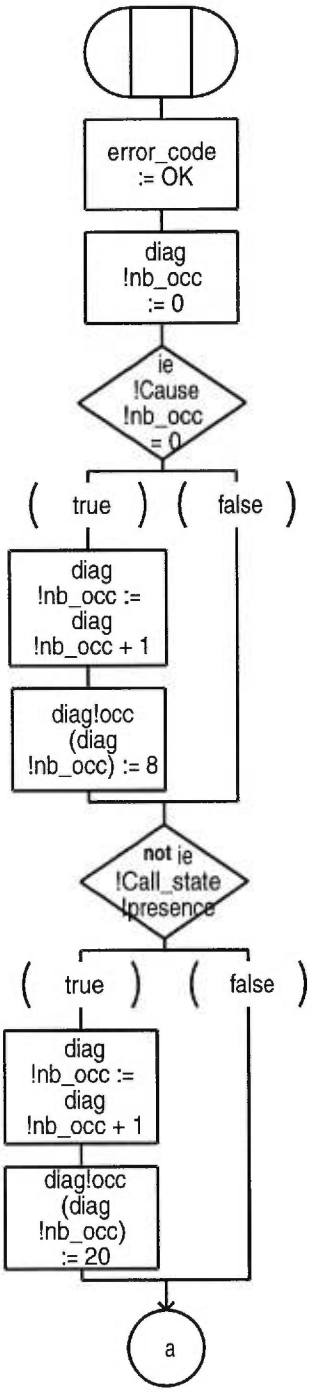


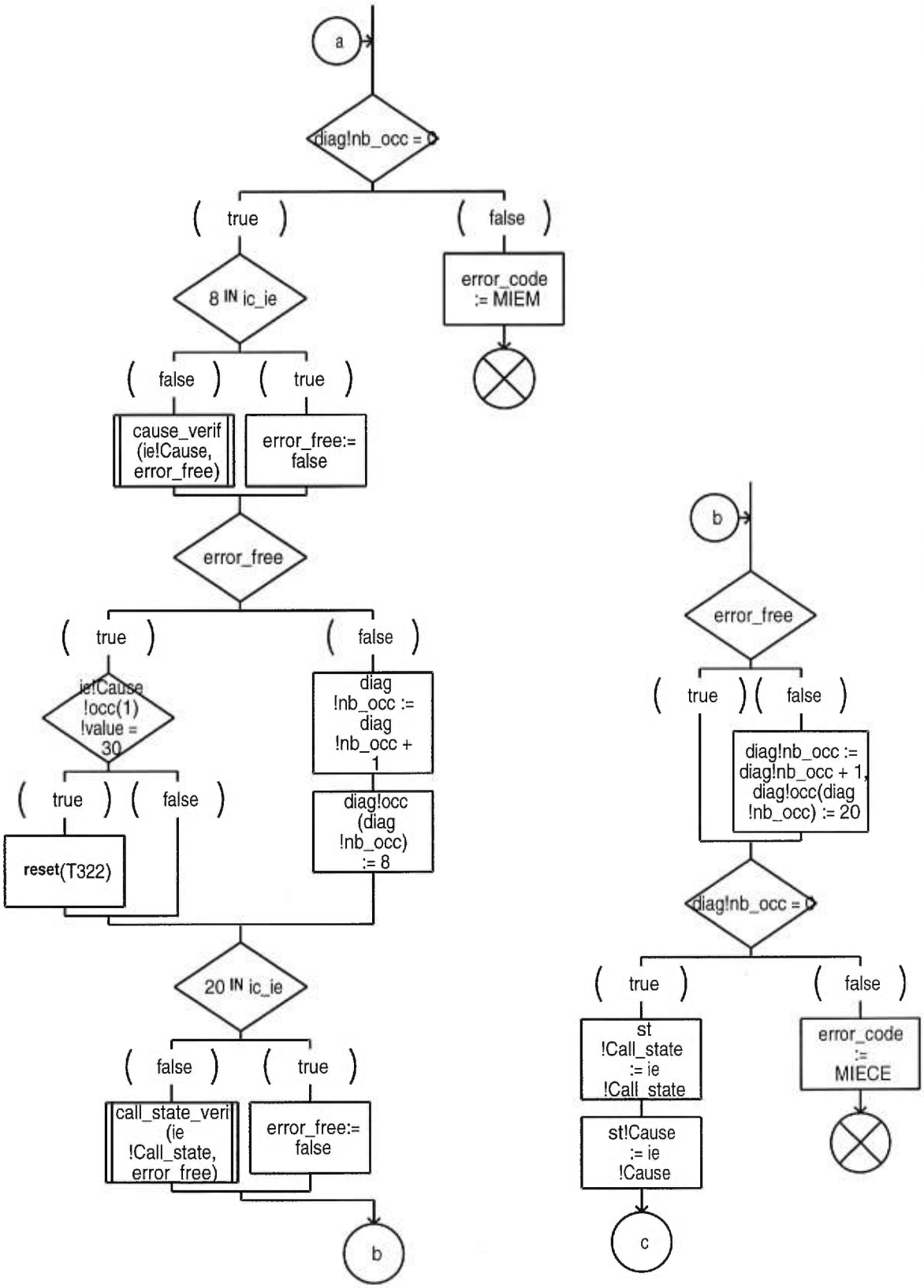

Lbl3
ie
!AAL_param
!presence
true
false
88 in ic_ie
false
true
AAL_param_verif
(ie
!AAL_param
, error_free)
error_free:=false
error_free
true
false
c
!AAL_param
:= ie
!AAL_param
c
!AAL_param
!presence
:= false
diag
!nb_occ :=
diag
!nb_occ +
1
diag!occ
(diag
!nb_occ)
:= 88
Lbl4
Lbl4
ie
!B_LLI
!nb_occ
/= 0
true
false
95 in ic_ie
false
true
B_LLI_verif
(ie!B_LLI,
error_free)
error_free:=false
error_free
true
false
c!B_LLI
:= ie
!B_LLI
c!B_LLI
!nb_occ
:= 0
diag
!nb_occ :=
diag
!nb_occ +
1
diag!occ
(diag
!nb_occ)
:= 95
Lbl5

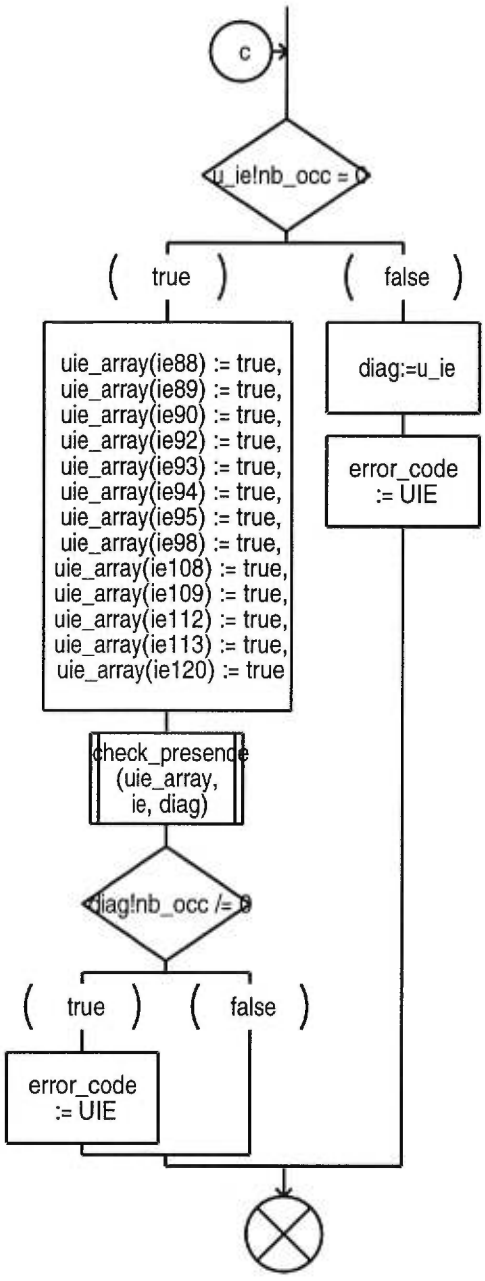

Lbl5
u_ie!nb_occ = 0
true
false
diag!nb_occ=0
diag:=u_ie
true
false
error_code
:= UIE
uie_array(ie8) := true,
uie_array(ie20) := true,
uie_array(ie88) := true,
uie_array(ie89) := true,
uie_array(ie92) := true,
uie_array(ie93) := true,
uie_array(ie94) := true,
uie_array(ie95) := true,
uie_array(ie98) := true,
uie_array(ie108) := true,
uie_array(ie109) := true,
uie_array(ie112) := true,
uie_array(ie113) := true,
uie_array(ie120) := true
error_code:=
NMIECE
check_presence
(uie_array,
ie, diag)
diag!nb_occ /= 0
true
false
error_code
:= UIE

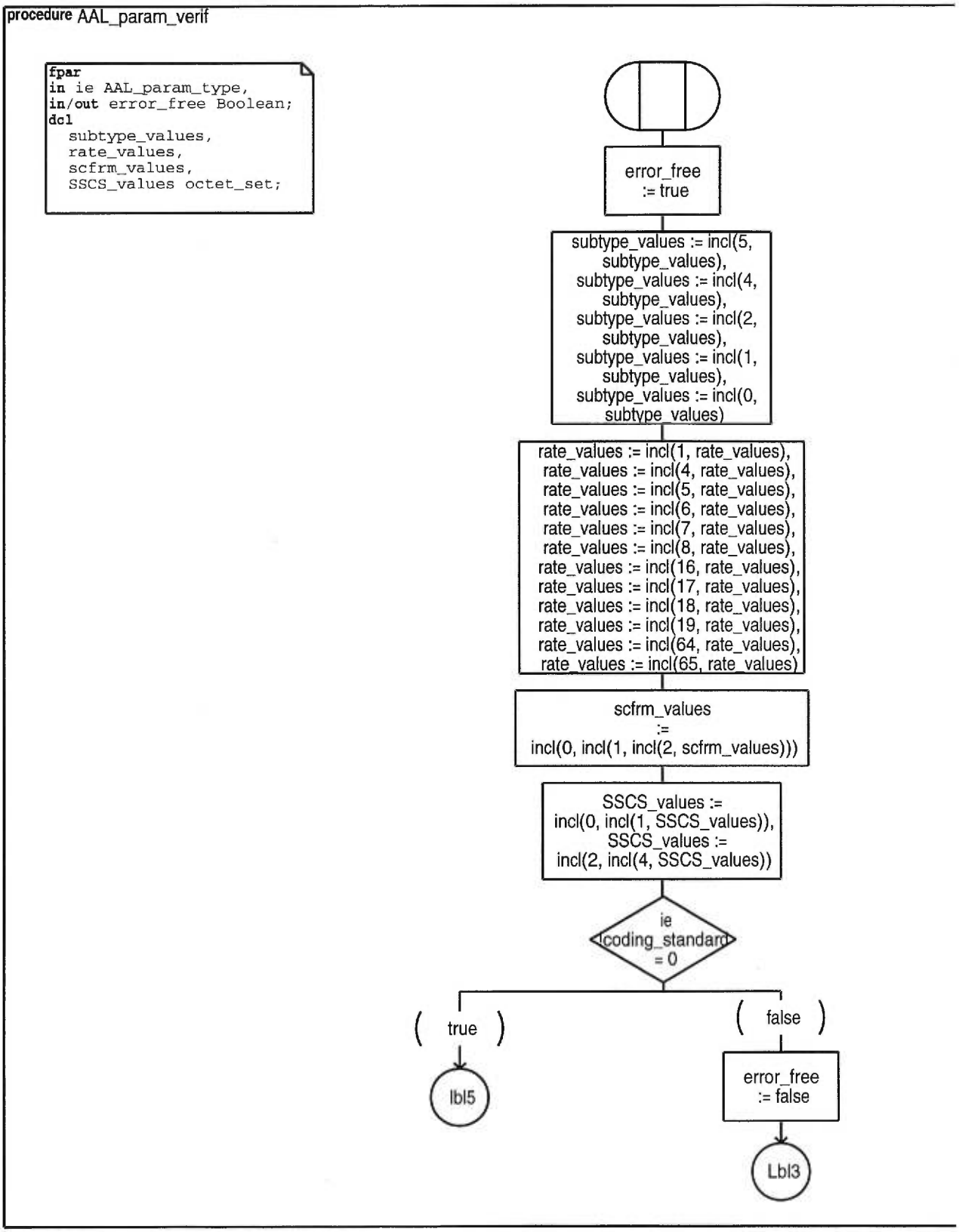
```
procedure status_verif
  FPAR
  in ie ie_type,
  in ic_ie ie_values_set,
  in u_ie occ_array_type,
  in/out st status_struct,
  in/out diag occ_array_type,
  in/out error_code error_code_type
```

```
dcl
  error_free Boolean;
dcl
  uie_array ie_array_type;
```

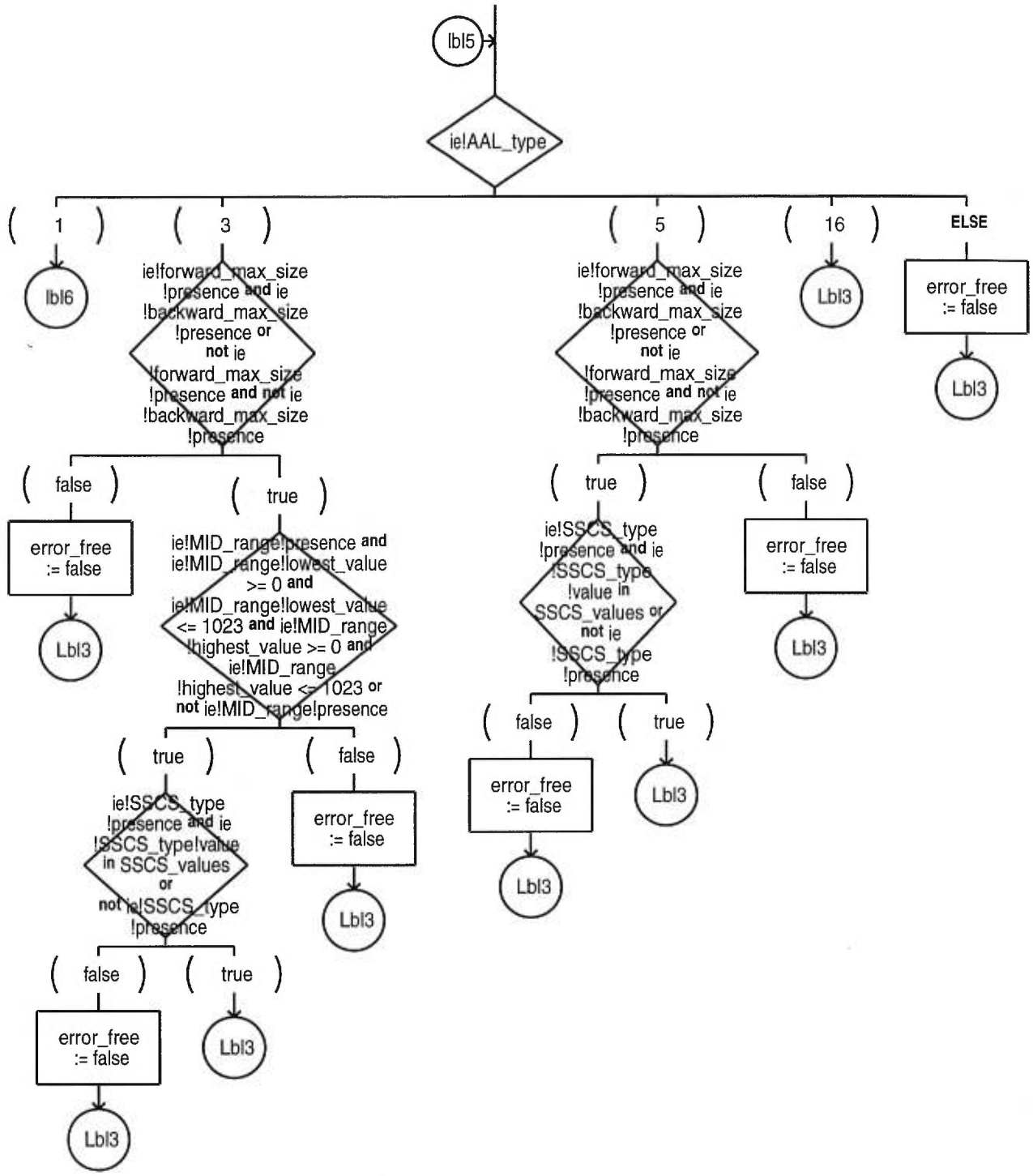



a
diag!nb_occ = 0
true
false
8 IN ic_ie
error_code := MIEM
false
true
cause_verif (ie!Cause, error_free)
error_free:= false
error_free
true
false
ie!Cause !occ(1) !value = 30
diag !nb_occ := diag !nb_occ + 1
true
false
reset(T322)
diag!occ (diag !nb_occ) := 8
20 IN ic_ie
false
true
call_state_veri (ie !Call_state, error_free)
error_free:= false
b
b
error_free
true
false
diag!nb_occ := diag!nb_occ + 1, diag!occ(diag !nb_occ) := 20
diag!nb_occ = 0
true
false
st !Call_state := ie !Call_state
error_code := MIECE
st!Cause := ie !Cause
c

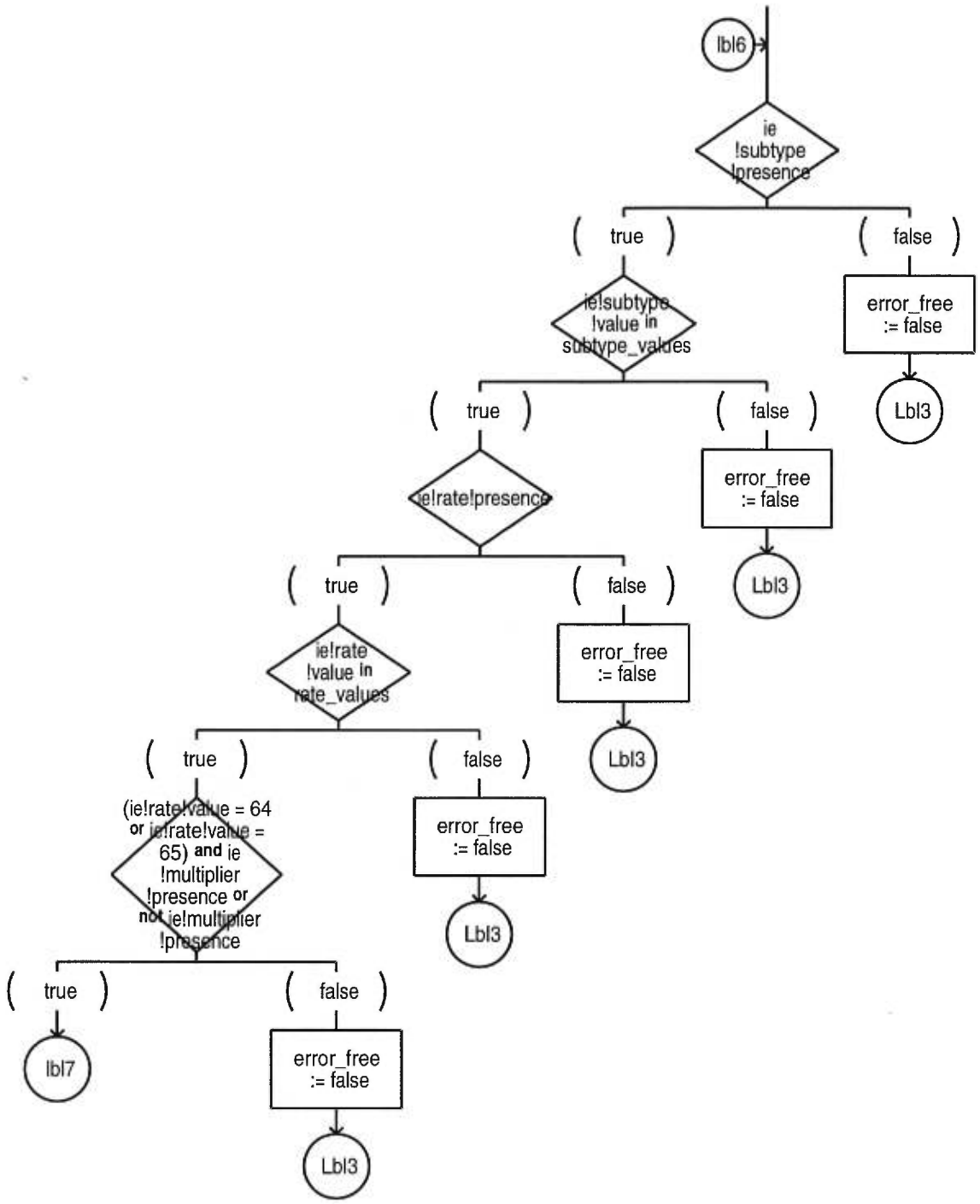

c
u_ie!nb_occ = 0
true
false
uie_array(ie88) := true,
uie_array(ie89) := true,
uie_array(ie90) := true,
uie_array(ie92) := true,
uie_array(ie93) := true,
uie_array(ie94) := true,
uie_array(ie95) := true,
uie_array(ie98) := true,
uie_array(ie108) := true,
uie_array(ie109) := true,
uie_array(ie112) := true,
uie_array(ie113) := true,
uie_array(ie120) := true
diag:=u_ie
error_code
:= UIE
check_presence
(uie_array,
ie, diag)
diag!nb_occ /= 0
true
false
error_code
:= UIE

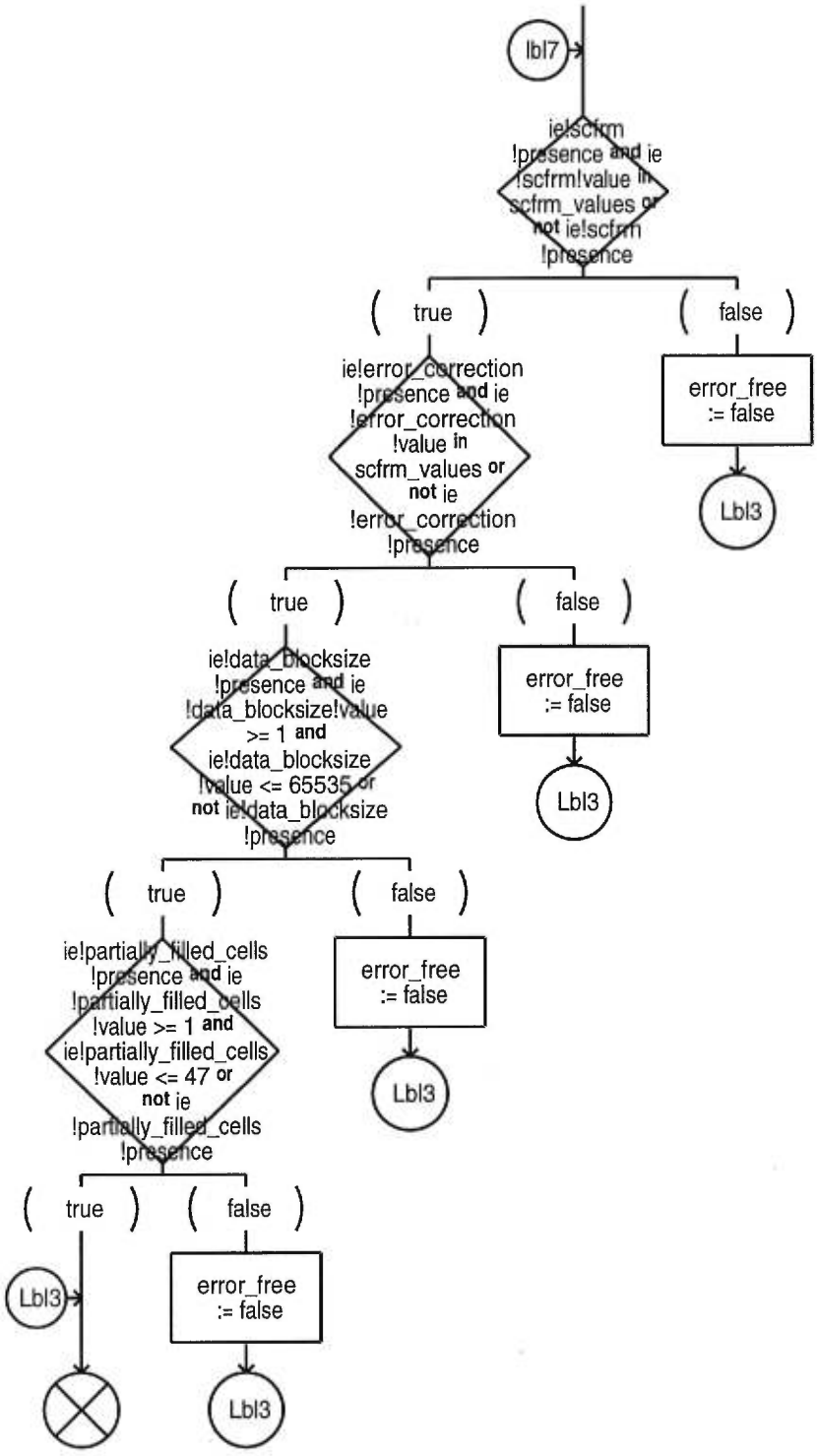


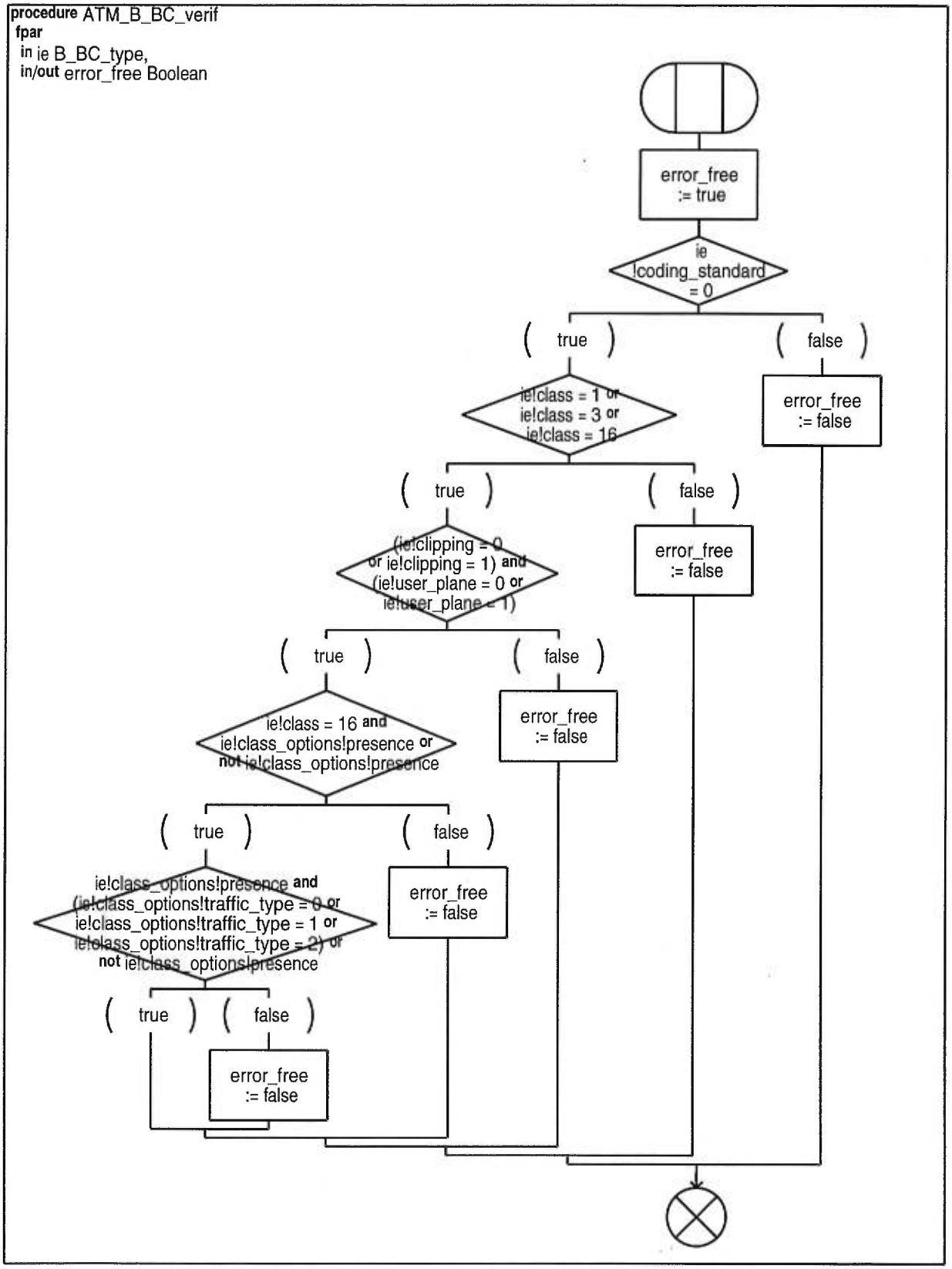
Author: Roxana Marcoci	File name: atm_sign.pr	AAL_param_verif, page: 1

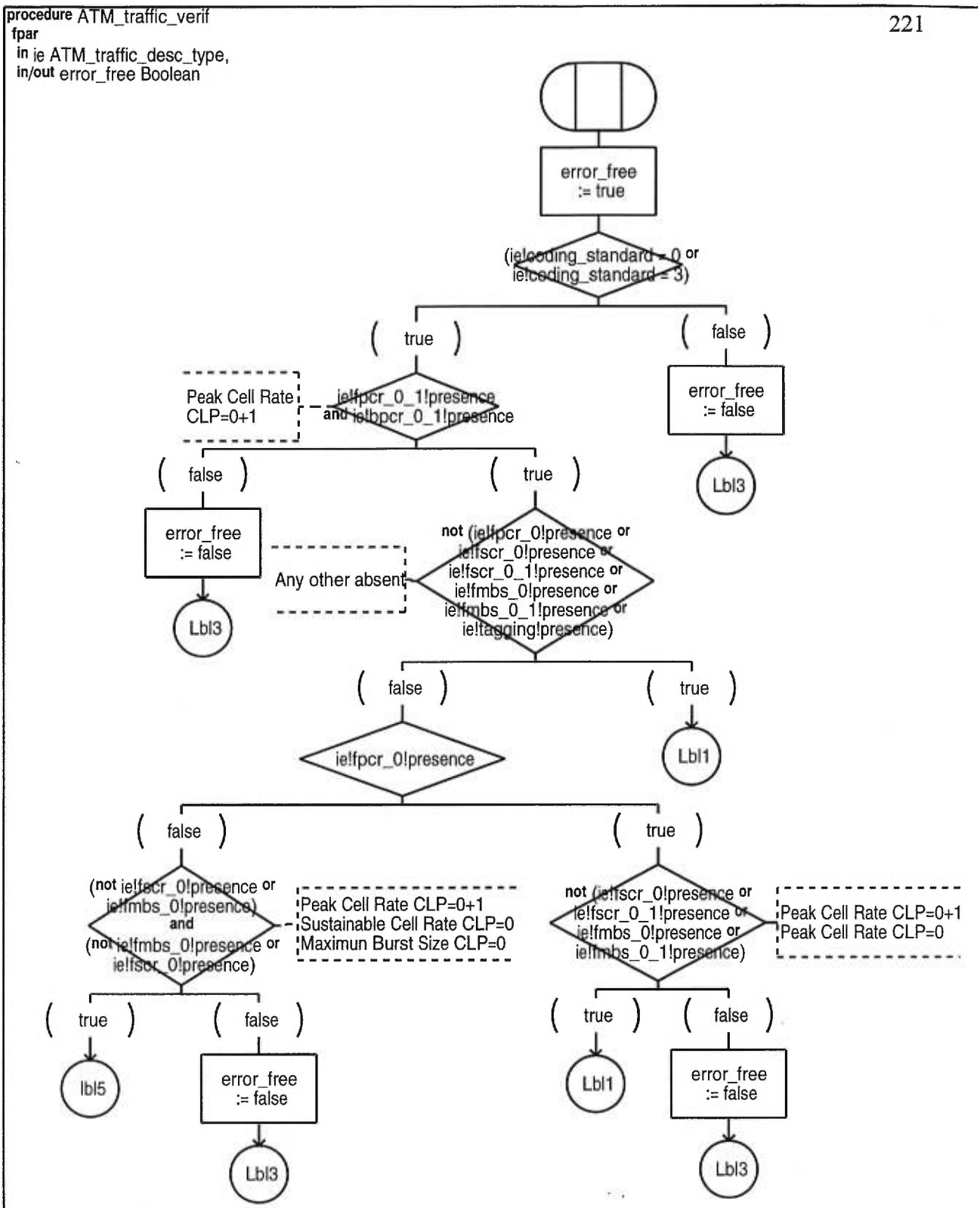


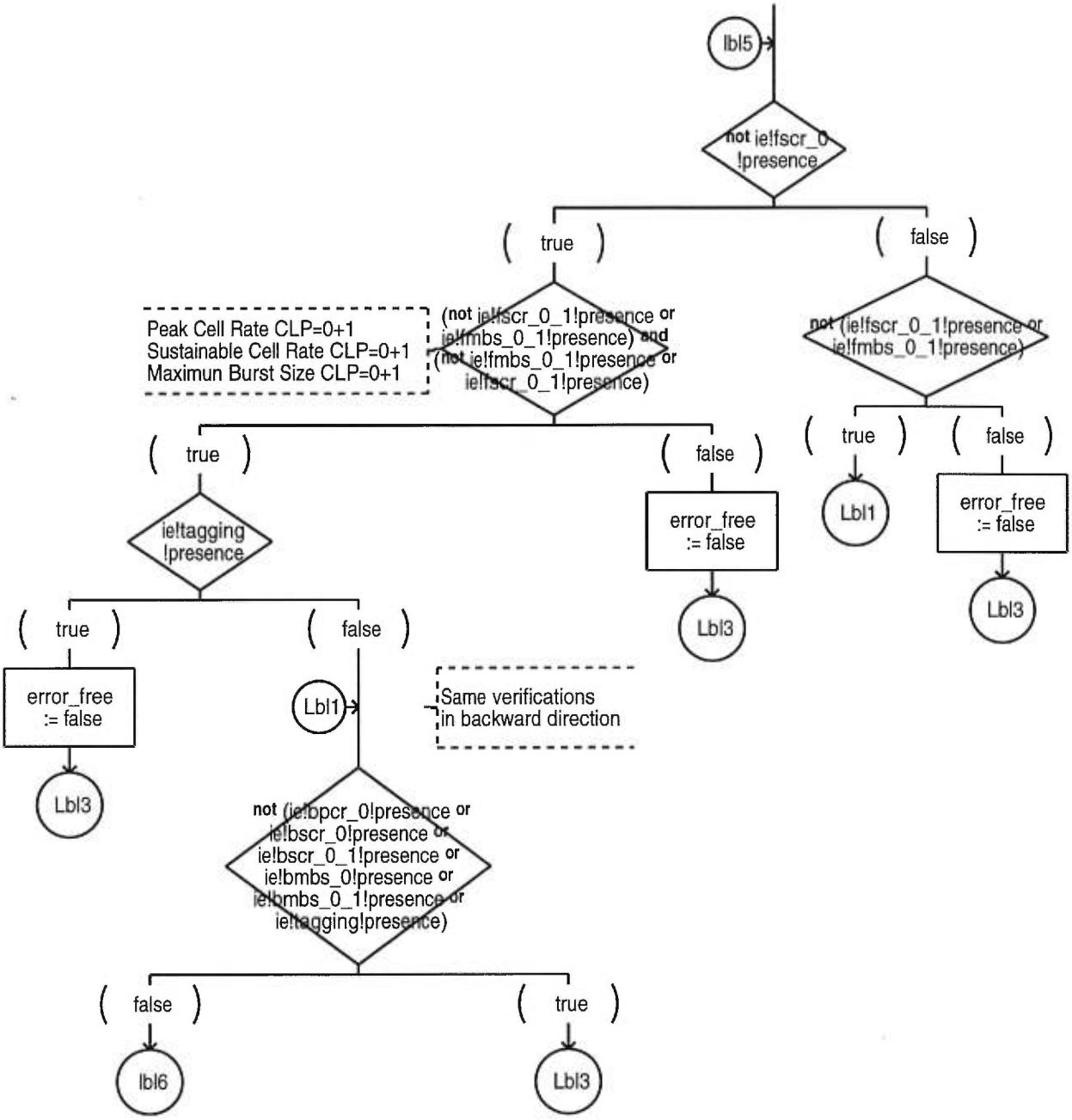
Author: Roxana Marcoci	File name: atm_sign.pr	AAL_param_verif, page: 2

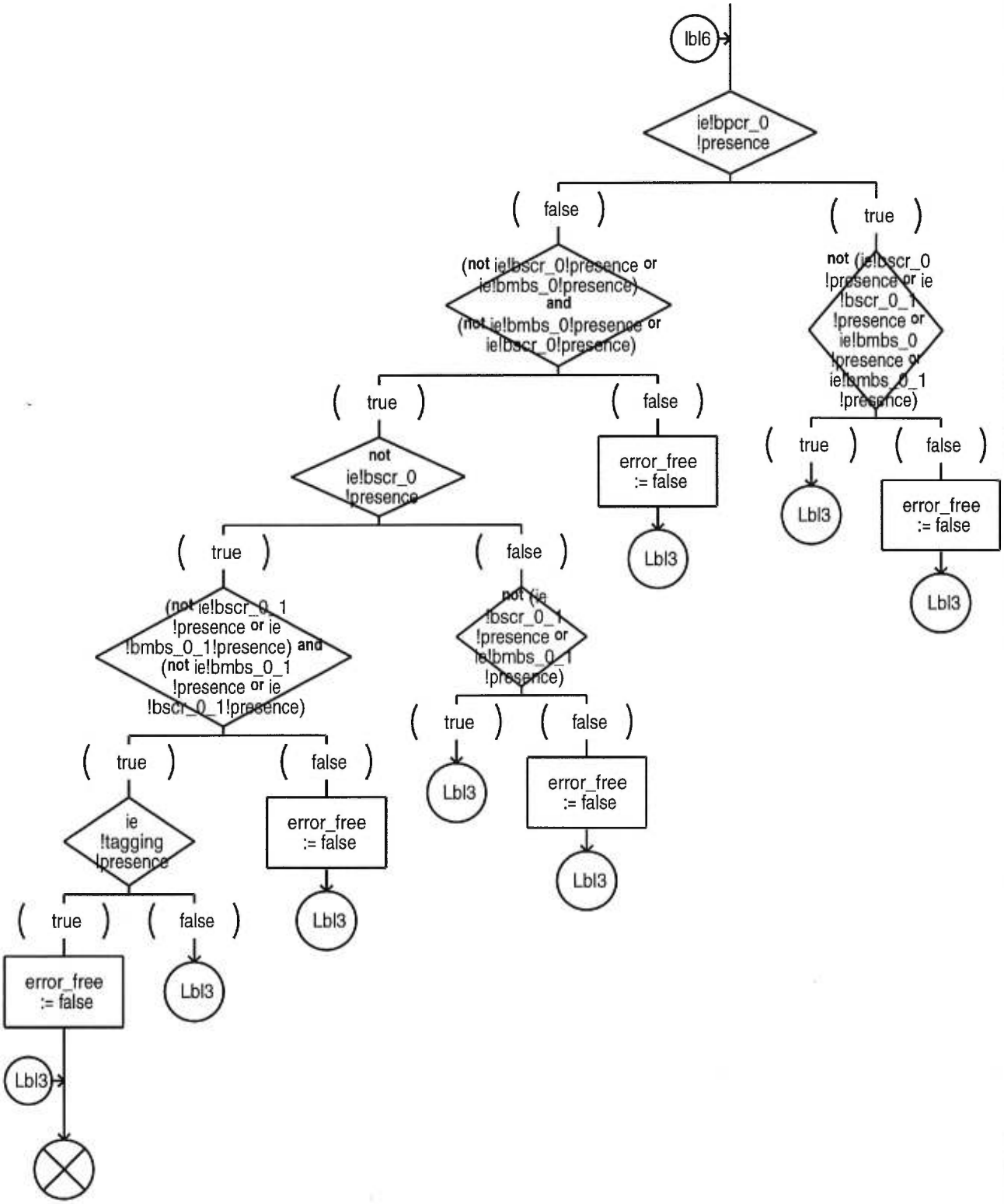

lbl6
ie !subtype !presence
true
false
error_free := false
Lbl3
ie!subtype !value in subtype_values
true
false
error_free := false
Lbl3
ie!rate!presence
true
false
error_free := false
Lbl3
ie!rate !value in rate_values
true
false
error_free := false
Lbl3
(ie!rate!value = 64 or ie!rate!value = 65) and ie !multiplier !presence or not ie!multiplier !presence
true
false
error_free := false
Lbl3
true
false
lbl7
error_free := false
Lbl3

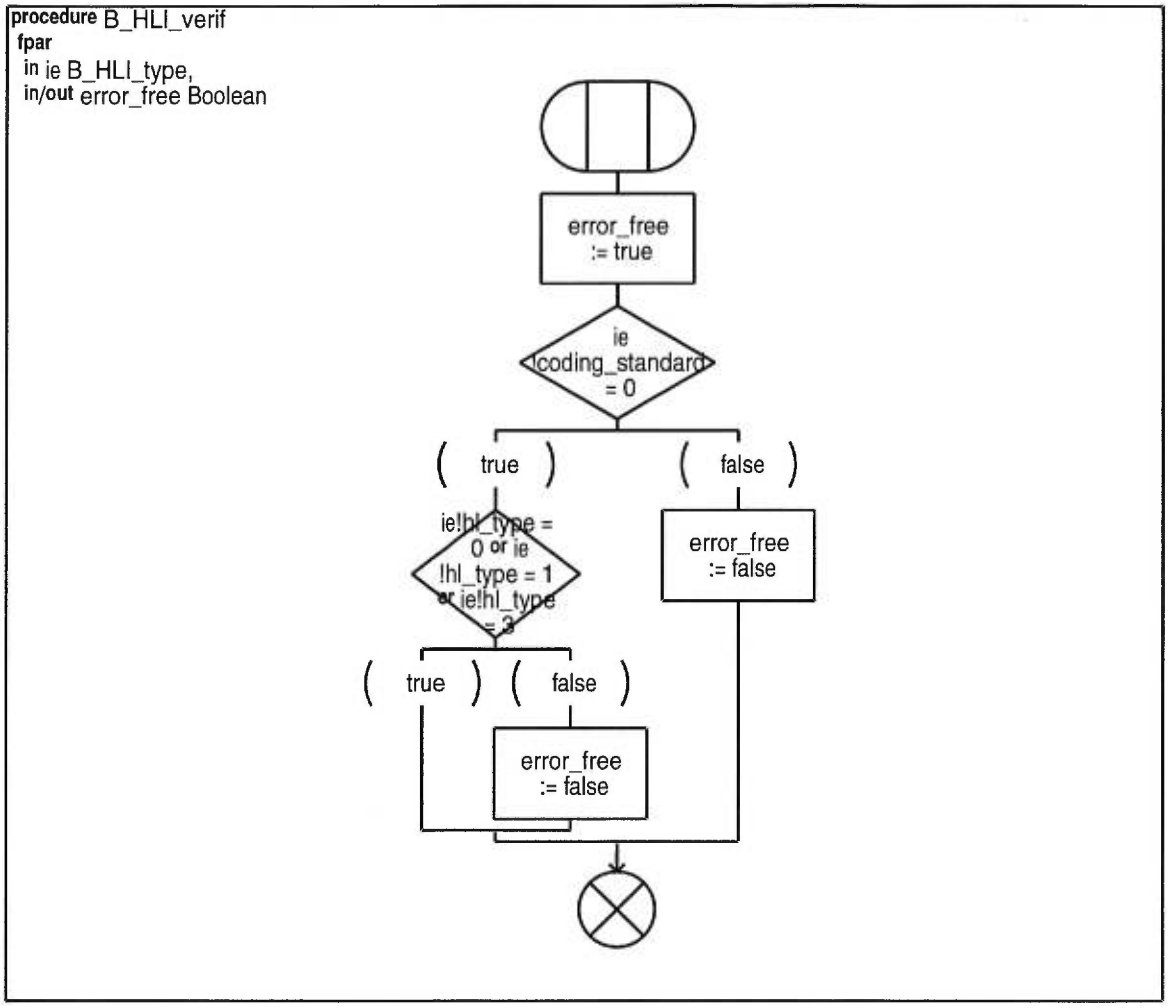

lbl7
ie!scfrm !presence and ie !scfrm!value in scfrm_values or not ie!scfrm !presence
true
false
error_free := false
Lbl3
ie!error_correction !presence and ie !error_correction !value in scfrm_values or not ie !error_correction !presence
true
false
error_free := false
Lbl3
ie!data_blocksize !presence and ie !data_blocksize!value >= 1 and ie!data_blocksize !value <= 65535 or not ie!data_blocksize !presence
true
false
error_free := false
Lbl3
ie!partially_filled_cells !presence and ie !partially_filled_cells !value >= 1 and ie!partially_filled_cells !value <= 47 or not ie !partially_filled_cells !presence
true
false
Lbl3
error_free := false
Lbl3

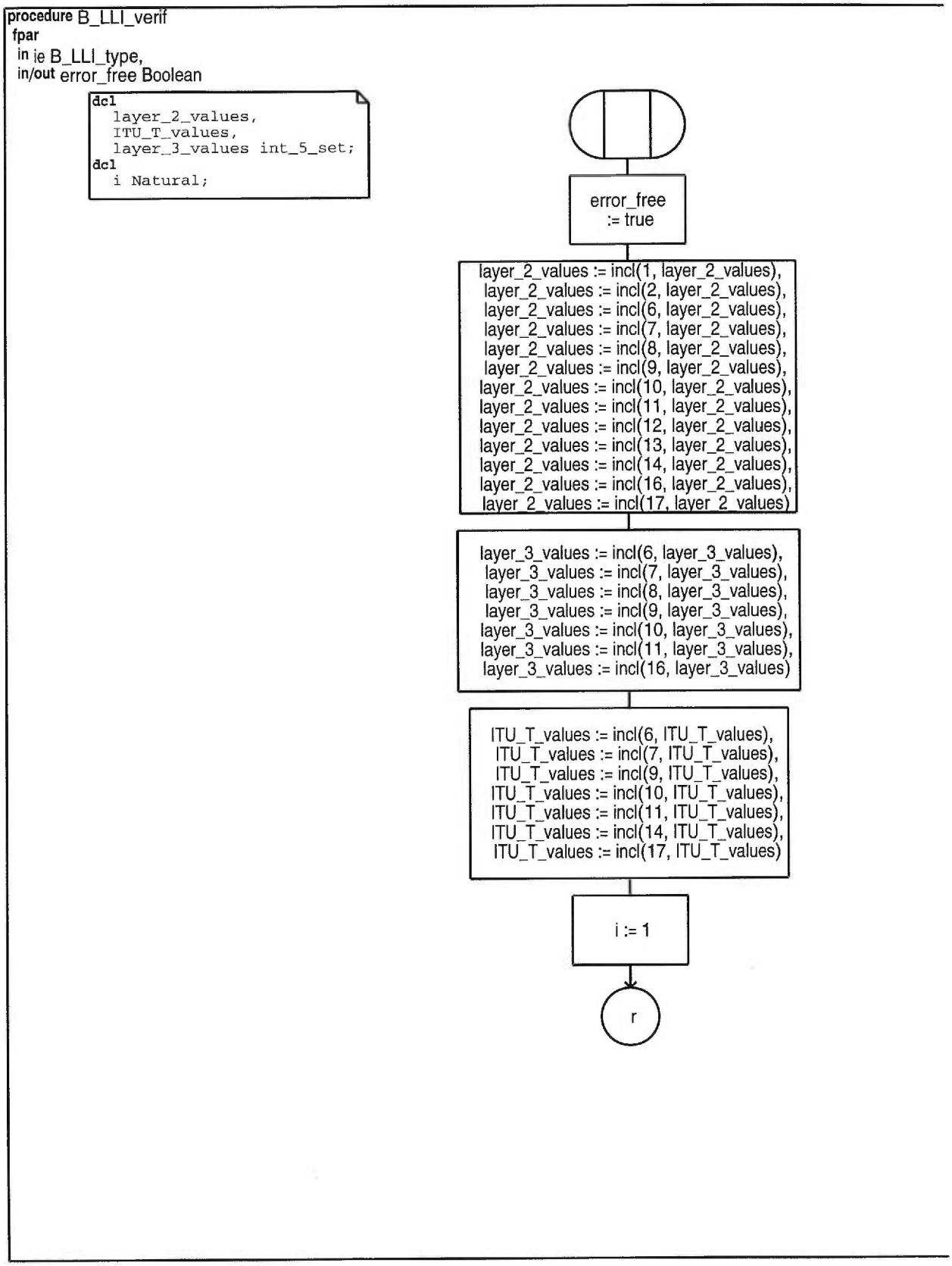

procedure ATM_B_BC_verif
fpar
in ie B_BC_type,
in/out error_free Boolean
error_free := true
ie !coding_standard = 0
true
false
error_free := false
ie!class = 1 or ie!class = 3 or ie!class = 16
true
false
error_free := false
(ie!clipping = 0 or ie!clipping = 1) and (ie!user_plane = 0 or ie!user_plane = 1)
true
false
error_free := false
ie!class = 16 and ie!class_options!presence or not ie!class_options!presence
true
false
error_free := false
ie!class_options!presence and (ie!class_options!traffic_type = 0 or ie!class_options!traffic_type = 1 or ie!class_options!traffic_type = 2) or not ie!class_options!presence
true
false
error_free := false

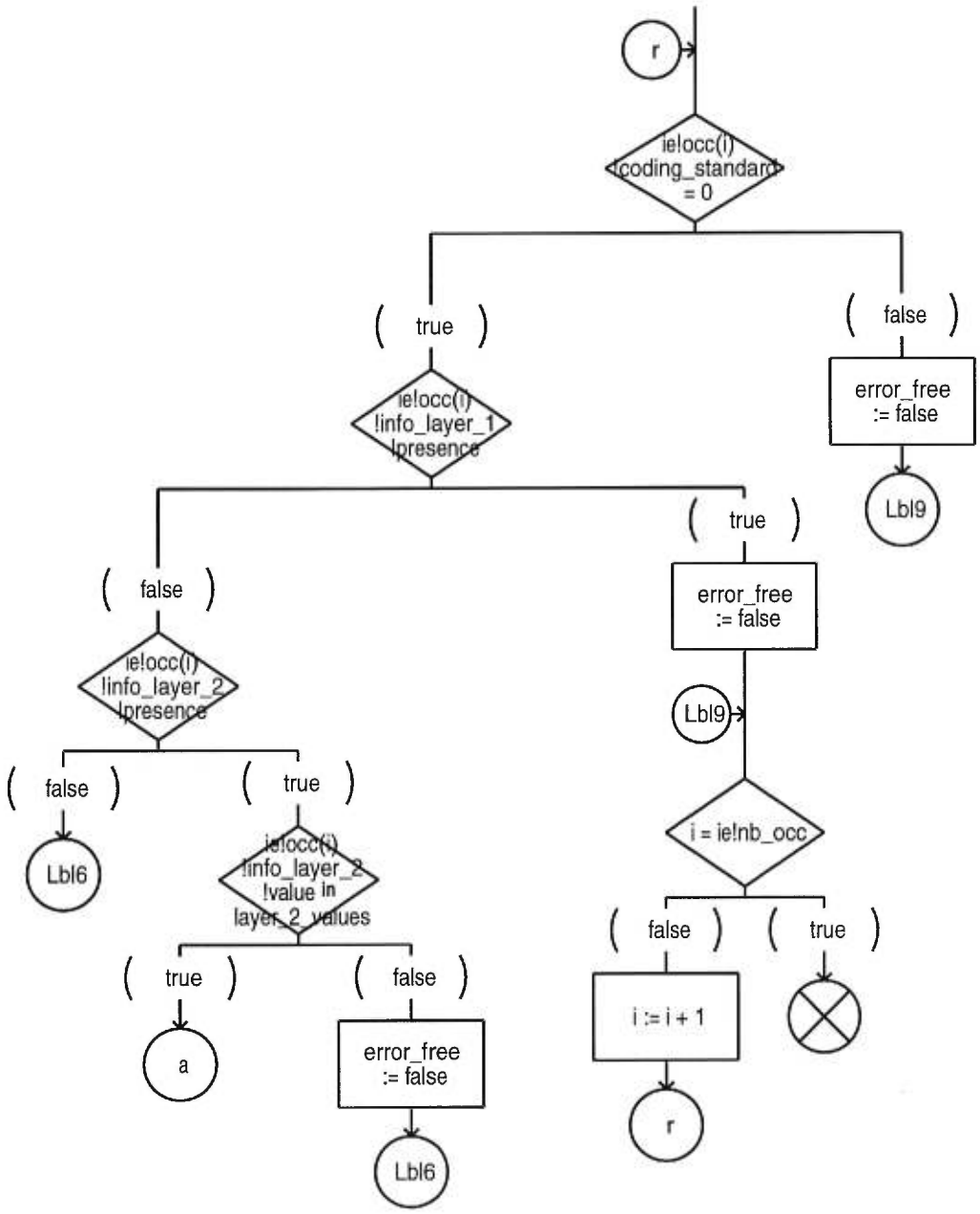

procedure ATM_traffic_verif
fpar
in ie ATM_traffic_desc_type,
in/out error_free Boolean
error_free := true
(ie!coding_standard = 0 or ie!coding_standard = 3)
true
false
error_free := false
Lbl3
Peak Cell Rate CLP=0+1
ie!fpcr_0_1!presence and ie!bpcr_0_1!presence
false
true
error_free := false
Lbl3
Any other absent
not (ie!fpcr_0!presence or ie!fscr_0!presence or ie!fscr_0_1!presence or ie!fmbs_0!presence or ie!fmbs_0_1!presence or ie!tagging!presence)
false
true
Lbl1
ie!fpcr_0!presence
false
true
(not ie!fscr_0!presence or ie!fmbs_0!presence) and (not ie!fmbs_0!presence or ie!fscr_0!presence)
Peak Cell Rate CLP=0+1
Sustainable Cell Rate CLP=0
Maximun Burst Size CLP=0
not (ie!fscr_0!presence or ie!fscr_0_1!presence or ie!fmbs_0!presence or ie!fmbs_0_1!presence)
Peak Cell Rate CLP=0+1
Peak Cell Rate CLP=0
true
false
true
false
lbl5
error_free := false
Lbl3
Lbl1
error_free := false
Lbl3

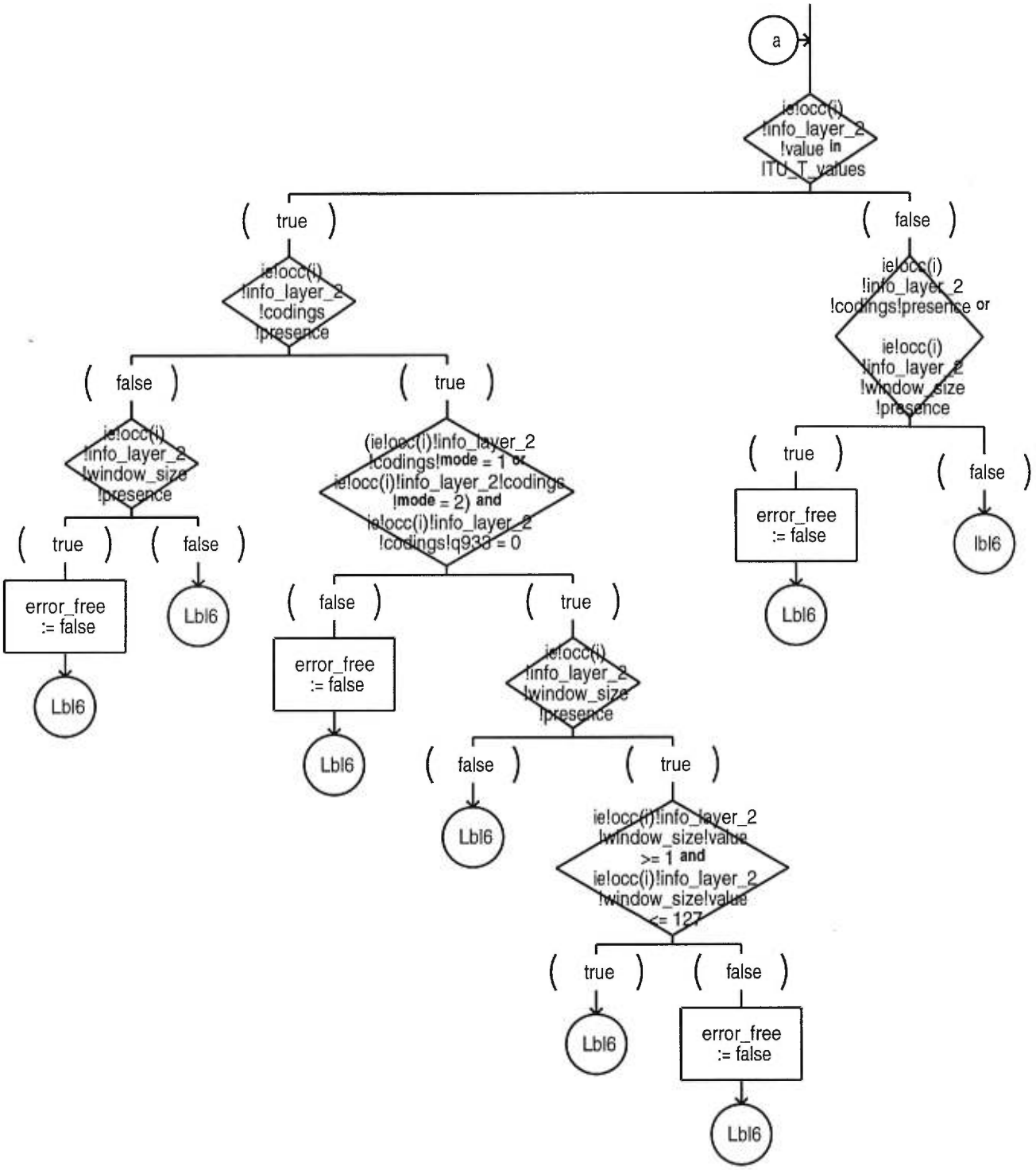

lbl5
not ie!fscr_0 !presence
true
false
Peak Cell Rate CLP=0+1
Sustainable Cell Rate CLP=0+1
Maximun Burst Size CLP=0+1
(not ie!fscr_0_1!presence or ie!fmbs_0_1!presence) and (not ie!fmbs_0_1!presence or ie!fscr_0_1!presence)
not (ie!fscr_0_1!presence or ie!fmbs_0_1!presence)
true
false
true
false
ie!tagging !presence
error_free := false
Lbl3
Lbl1
error_free := false
Lbl3
true
false
error_free := false
Lbl3
Lbl1
Same verifications in backward direction
not (ie!bpcr_0!presence or ie!bscr_0!presence or ie!bscr_0_1!presence or ie!bmbs_0!presence or ie!bmbs_0_1!presence or ie!tagging!presence)
false
true
lbl6
Lbl3

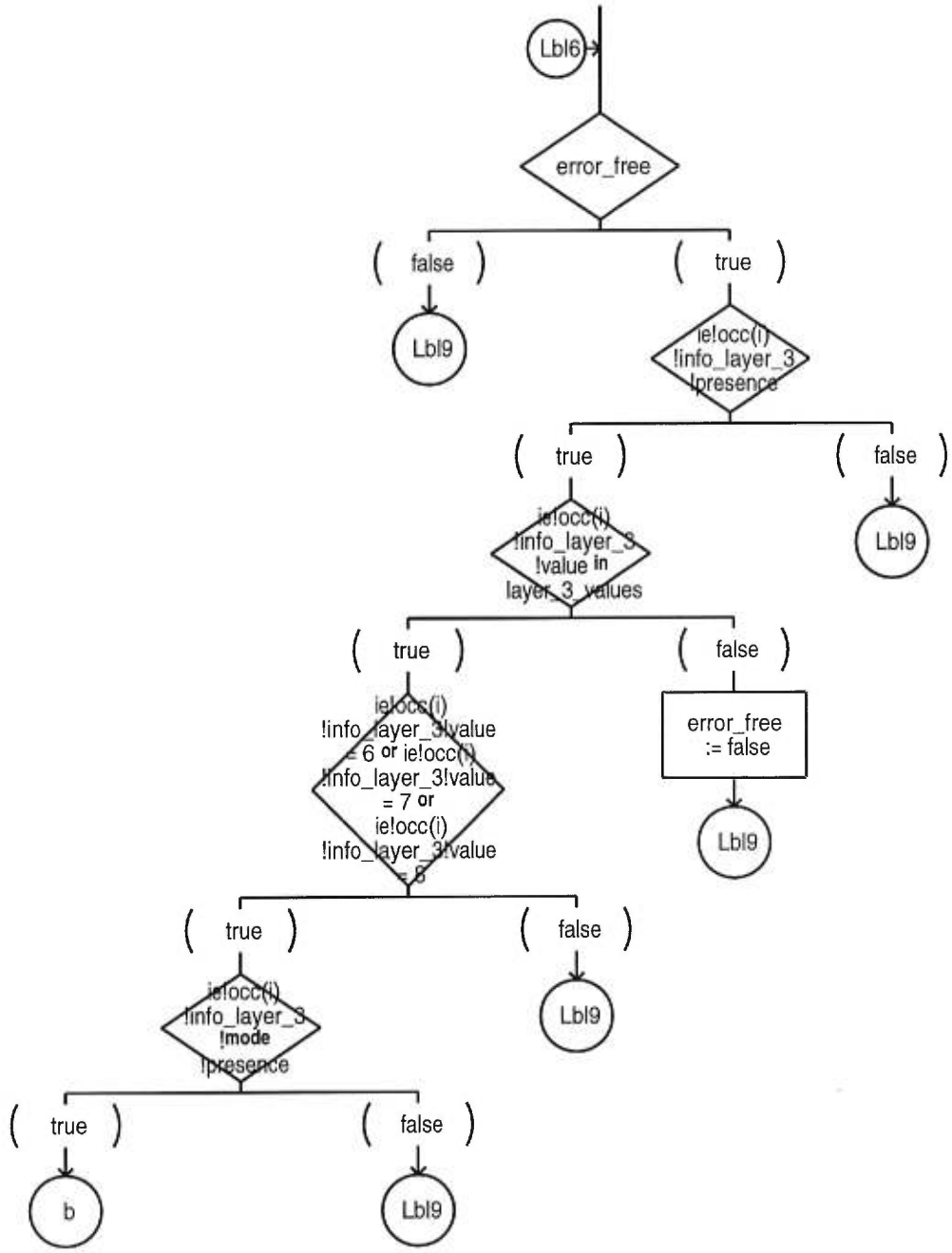

lbl6
ie!bpcr_0 !presence
false
true
(not ie!bscr_0!presence or ie!bmbs_0!presence) and (not ie!bmbs_0!presence or ie!bscr_0!presence)
not (ie!bscr_0 !presence or ie !bscr_0_1 !presence or ie!bmbs_0 !presence or ie!bmbs_0_1 !presence)
true
false
not ie!bscr_0 !presence
error_free := false
Lbl3
true
false
Lbl3
error_free := false
(not ie!bscr_0_1 !presence or ie !bmbs_0_1!presence) and (not ie!bmbs_0_1 !presence or ie !bscr_0_1!presence)
not (ie !bscr_0_1 !presence or ie!bmbs_0_1 !presence)
true
false
Lbl3
error_free := false
Lbl3
ie !tagging !presence
true
false
error_free := false
Lbl3
error_free := false
Lbl3
Lbl3

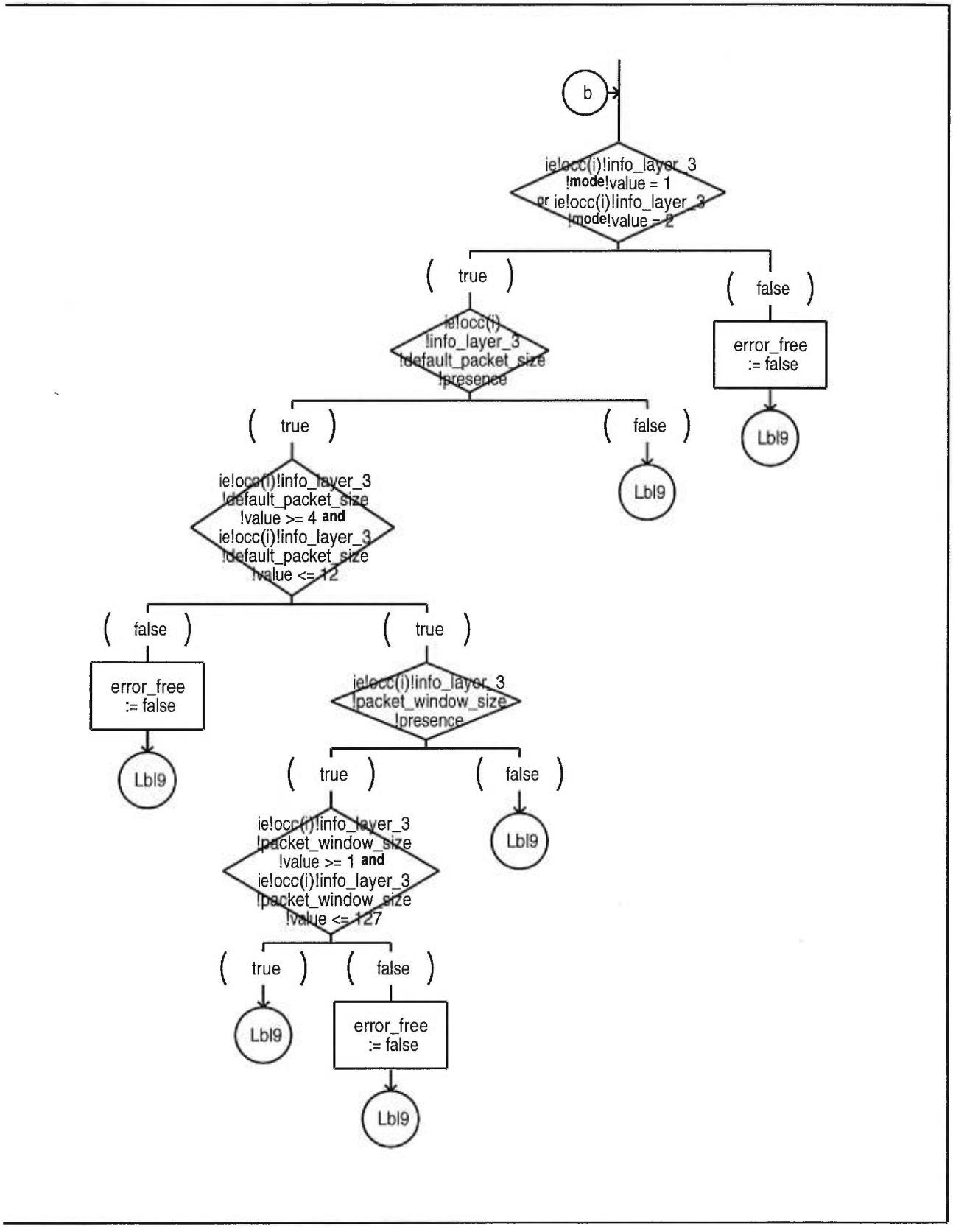

procedure B_HLI_verif
fpar
in ie B_HLI_type,
in/out error_free Boolean
error_free := true
ie !coding_standard = 0
true
false
ie!hl_type = 0 or ie !hl_type = 1 or ie!hl_type = 3
error_free := false
true
false
error_free := false

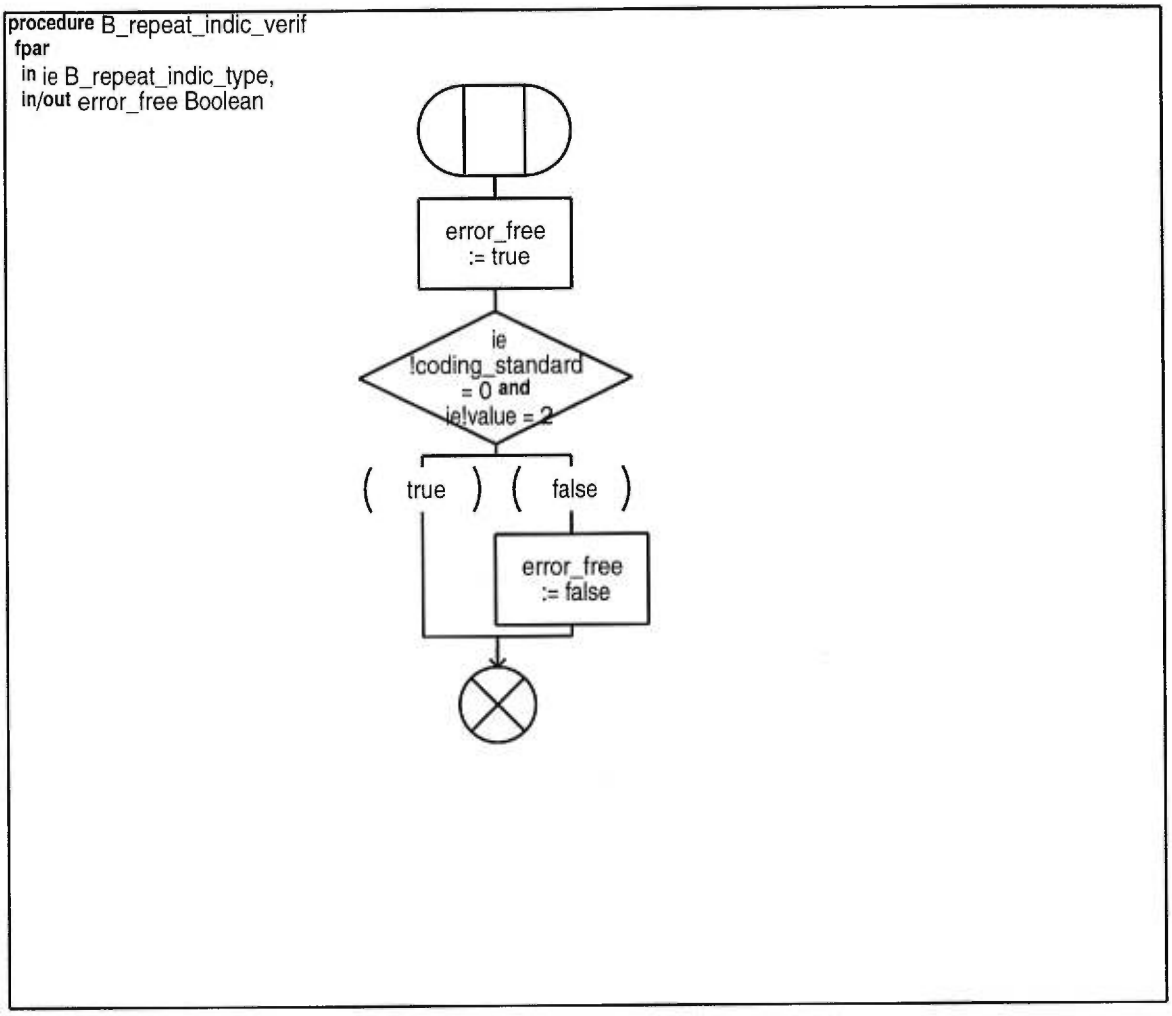

procedure B_LLI_verif
fpar
in ie B_LLI_type,
in/out error_free Boolean
dcl
layer_2_values,
ITU_T_values,
layer_3_values int_5_set;
dcl
i Natural;
error_free := true
layer_2_values := incl(1, layer_2_values),
layer_2_values := incl(2, layer_2_values),
layer_2_values := incl(6, layer_2_values),
layer_2_values := incl(7, layer_2_values),
layer_2_values := incl(8, layer_2_values),
layer_2_values := incl(9, layer_2_values),
layer_2_values := incl(10, layer_2_values),
layer_2_values := incl(11, layer_2_values),
layer_2_values := incl(12, layer_2_values),
layer_2_values := incl(13, layer_2_values),
layer_2_values := incl(14, layer_2_values),
layer_2_values := incl(16, layer_2_values),
layer_2_values := incl(17, layer_2_values)
layer_3_values := incl(6, layer_3_values),
layer_3_values := incl(7, layer_3_values),
layer_3_values := incl(8, layer_3_values),
layer_3_values := incl(9, layer_3_values),
layer_3_values := incl(10, layer_3_values),
layer_3_values := incl(11, layer_3_values),
layer_3_values := incl(16, layer_3_values)
ITU_T_values := incl(6, ITU_T_values),
ITU_T_values := incl(7, ITU_T_values),
ITU_T_values := incl(9, ITU_T_values),
ITU_T_values := incl(10, ITU_T_values),
ITU_T_values := incl(11, ITU_T_values),
ITU_T_values := incl(14, ITU_T_values),
ITU_T_values := incl(17, ITU_T_values)
i := 1
r


r
ie!occ(i)
!coding_standard
= 0
true
false
ie!occ(i)
!info_layer_1
!presence
error_free
:= false
Lbl9
true
false
error_free
:= false
ie!occ(i)
!info_layer_2
!presence
Lbl9
false
true
Lbl6
i = ie!nb_occ
ie!occ(i)
!info_layer_2
!value in
layer_2_values
false
true
true
false
i := i + 1
a
error_free
:= false
r
Lbl6

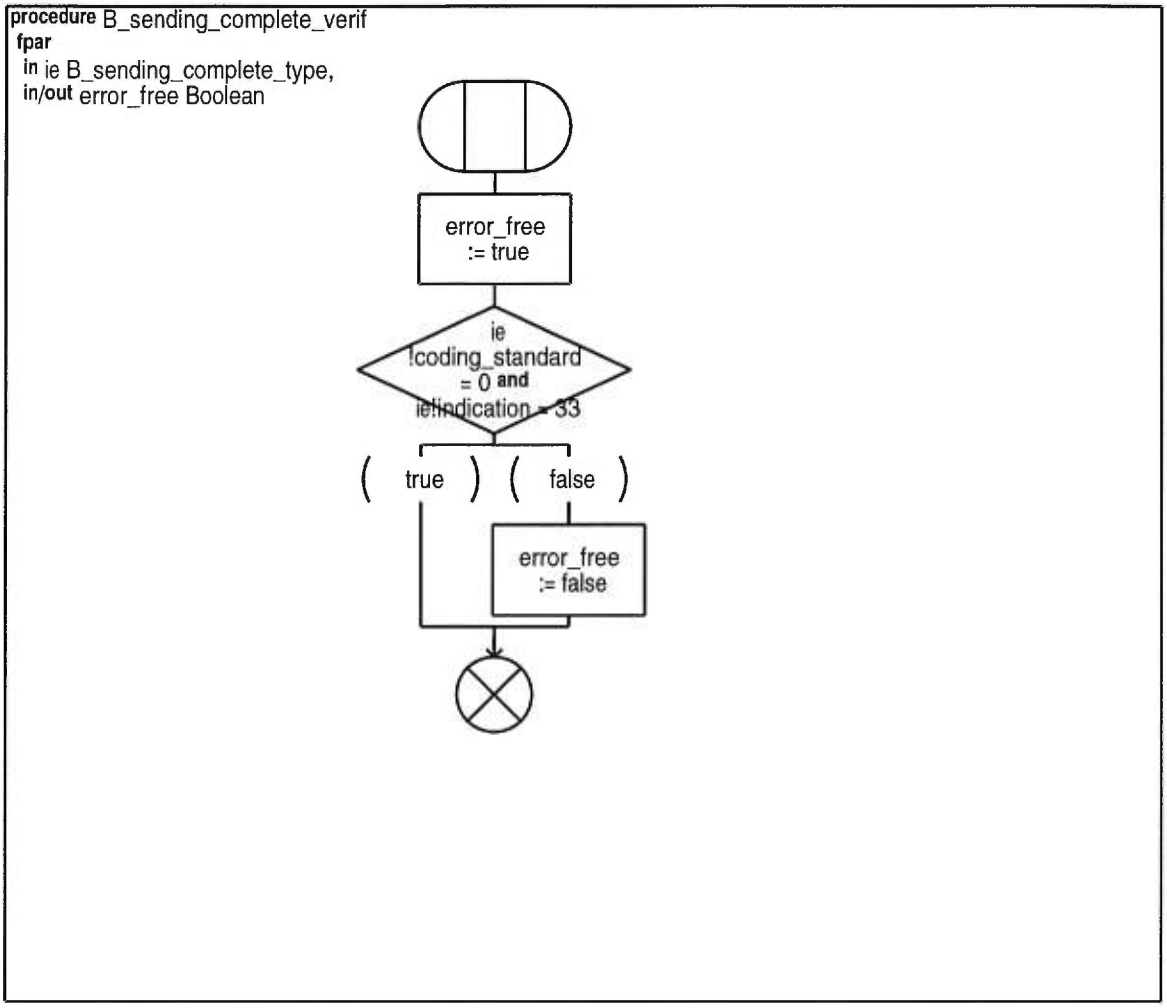

a
ie!occ(i)
!info_layer_2
!value in
ITU_T_values
true
false
ie!occ(i)
!info_layer_2
!codings
!presence
false
true
ie!occ(i)
!info_layer_2
!window_size
!presence
(ie!occ(i)!info_layer_2
!codings!mode = 1 or
ie!occ(i)!info_layer_2!codings
!mode = 2) and
ie!occ(i)!info_layer_2
!codings!q933 = 0
ie!occ(i)
!info_layer_2
!codings!presence or
ie!occ(i)
!info_layer_2
!window_size
!presence
true
false
true
false
error_free
:= false
Lbl6
false
true
error_free
:= false
Lbl6
ie!occ(i)
!info_layer_2
!window_size
!presence
false
true
Lbl6
ie!occ(i)!info_layer_2
!window_size!value
>= 1 and
ie!occ(i)!info_layer_2
!window_size!value
<= 127
true
false
Lbl6
error_free
:= false
Lbl6

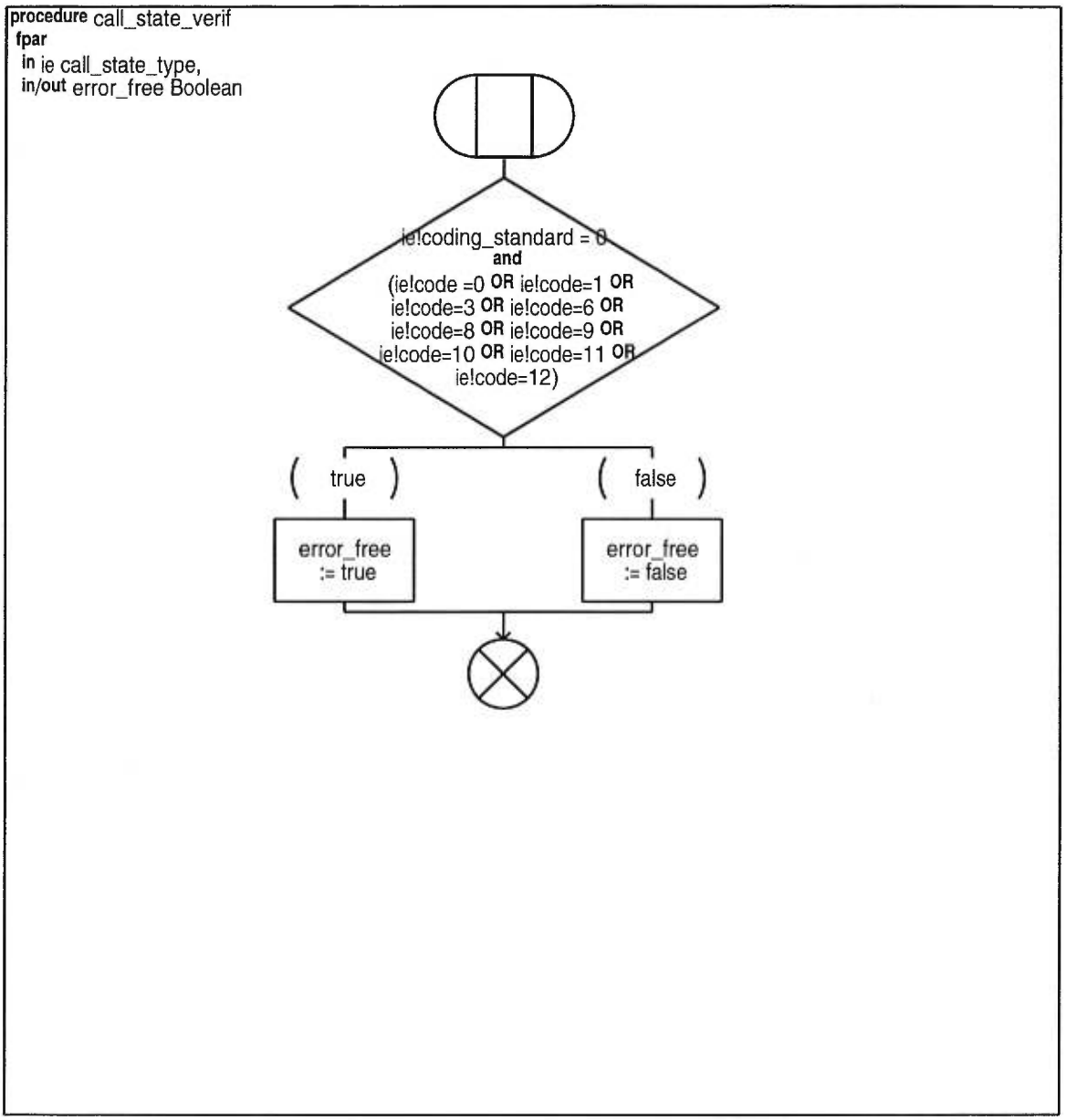

Lbl6
error_free
false
true
Lbl9
ie!occ(i)
!info_layer_3
!presence
true
false
Lbl9
ie!occ(i)
!info_layer_3
!value in
layer_3_values
true
false
ie!occ(i)
!info_layer_3!value
= 6 or ie!occ(i)
!info_layer_3!value
= 7 or
ie!occ(i)
!info_layer_3!value
= 8
error_free
:= false
Lbl9
true
false
ie!occ(i)
!info_layer_3
!mode
!presence
Lbl9
true
false
b
Lbl9

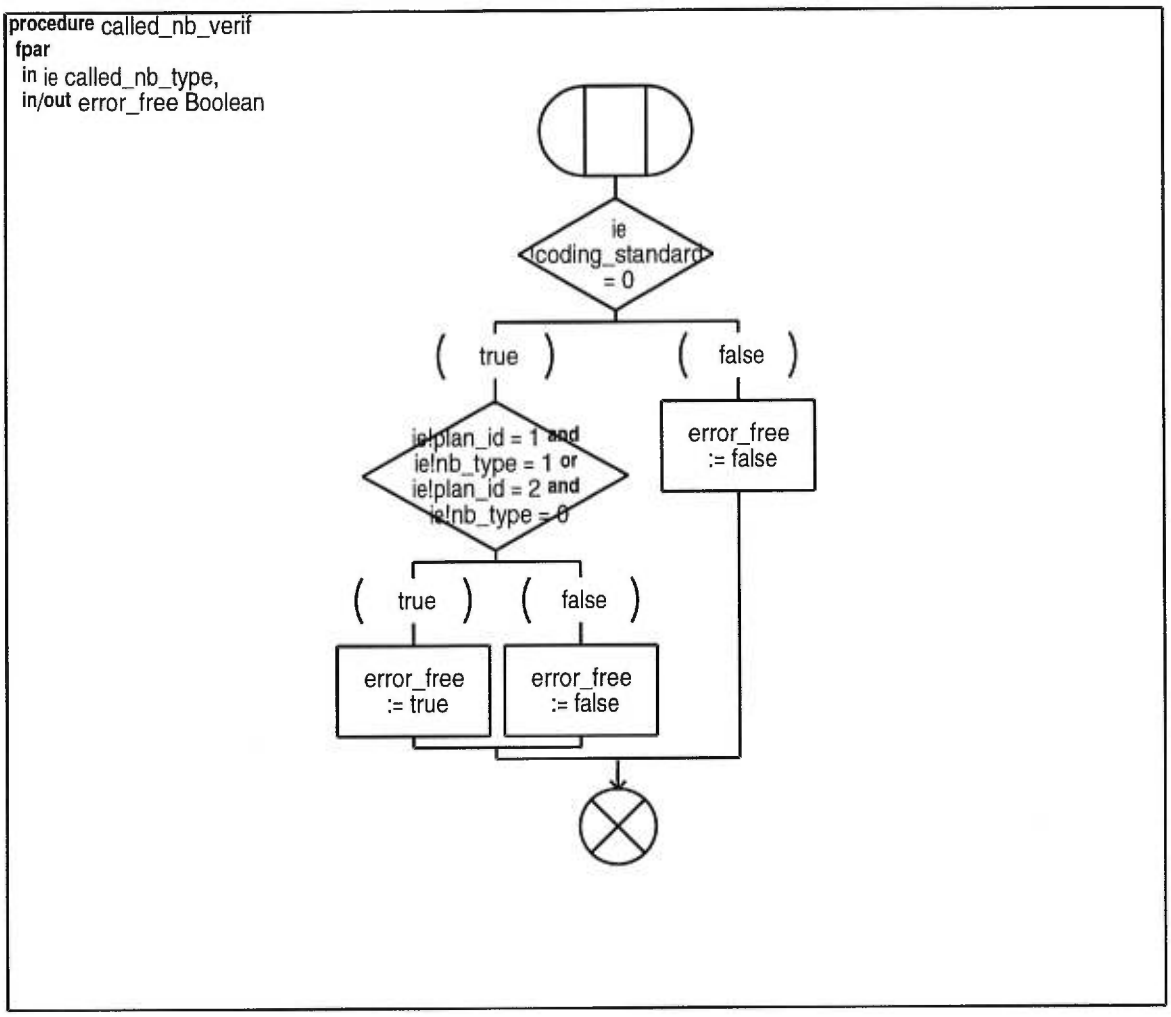

b
ie!occ(i)!info_layer_3
!mode!value = 1
or ie!occ(i)!info_layer_3
!mode!value = 2
true
false
ie!occ(i)
!info_layer_3
!default_packet_size
!presence
error_free
:= false
Lbl9
true
false
Lbl9
ie!occ(i)!info_layer_3
!default_packet_size
!value >= 4 and
ie!occ(i)!info_layer_3
!default_packet_size
!value <= 12
false
true
error_free
:= false
Lbl9
ie!occ(i)!info_layer_3
!packet_window_size
!presence
true
false
Lbl9
ie!occ(i)!info_layer_3
!packet_window_size
!value >= 1 and
ie!occ(i)!info_layer_3
!packet_window_size
!value <= 127
true
false
Lbl9
error_free
:= false
Lbl9

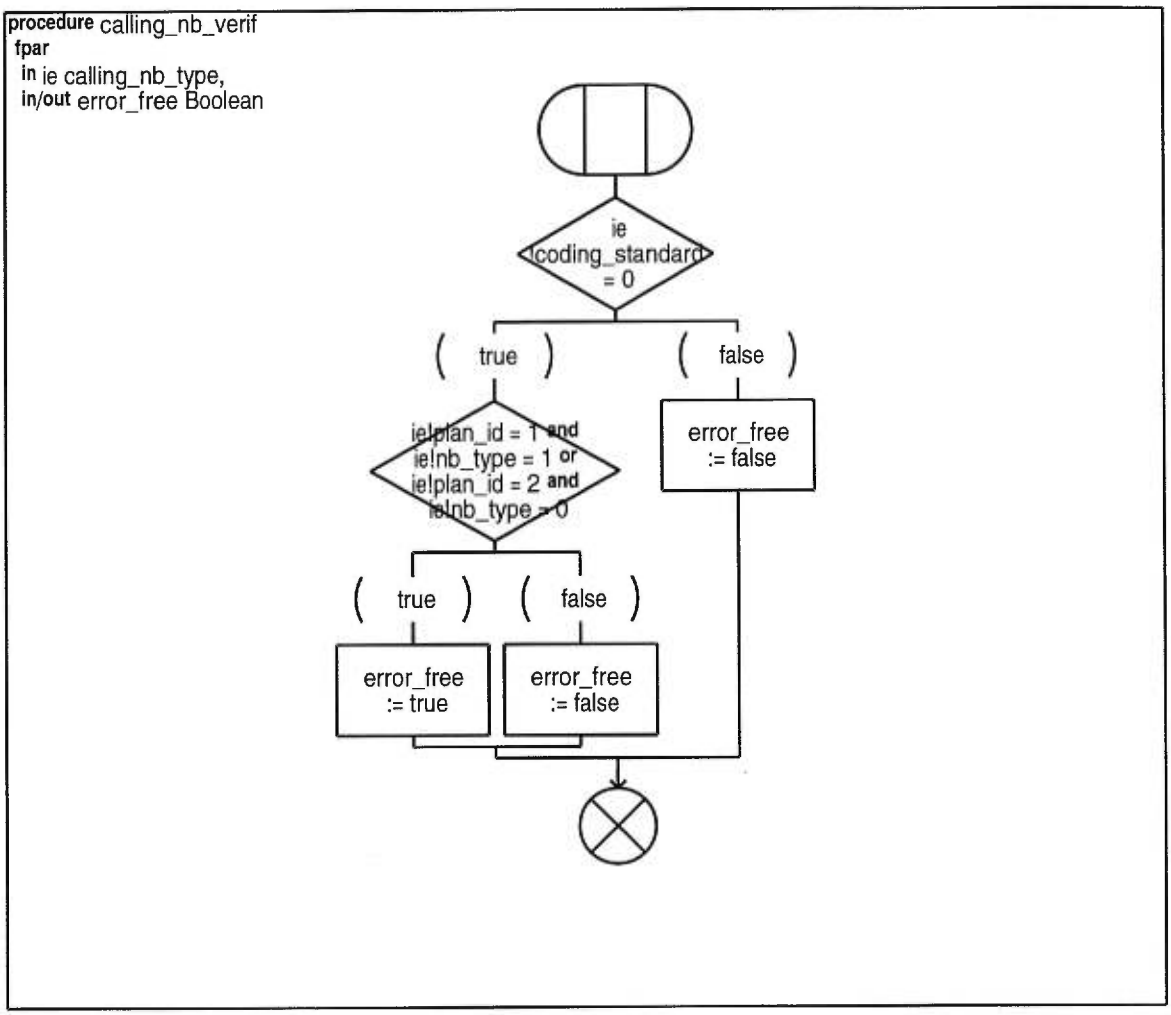


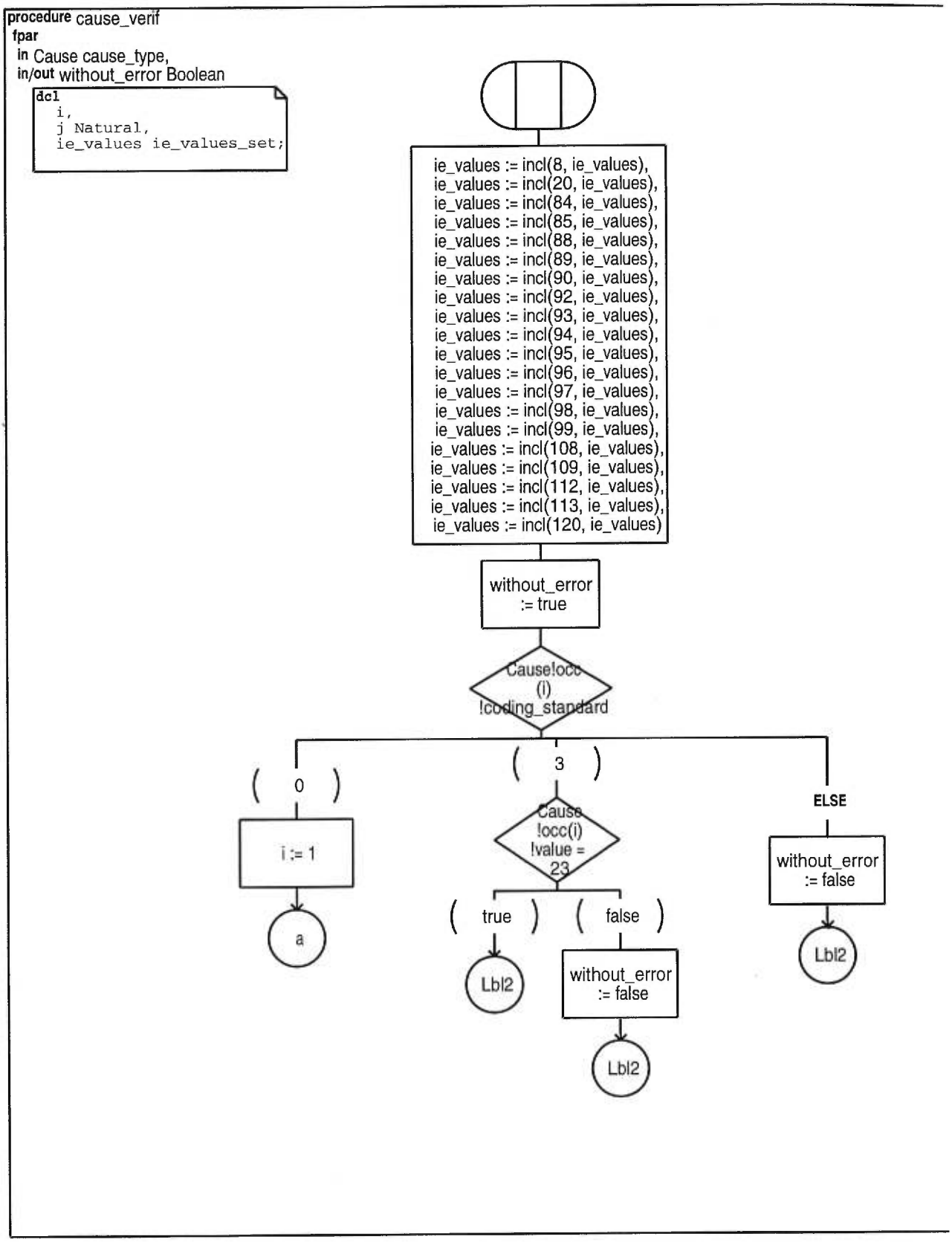
Author: Roxana Marcoci	File name: atm_sign.pr	B_repeat_indic_verif, page: 1

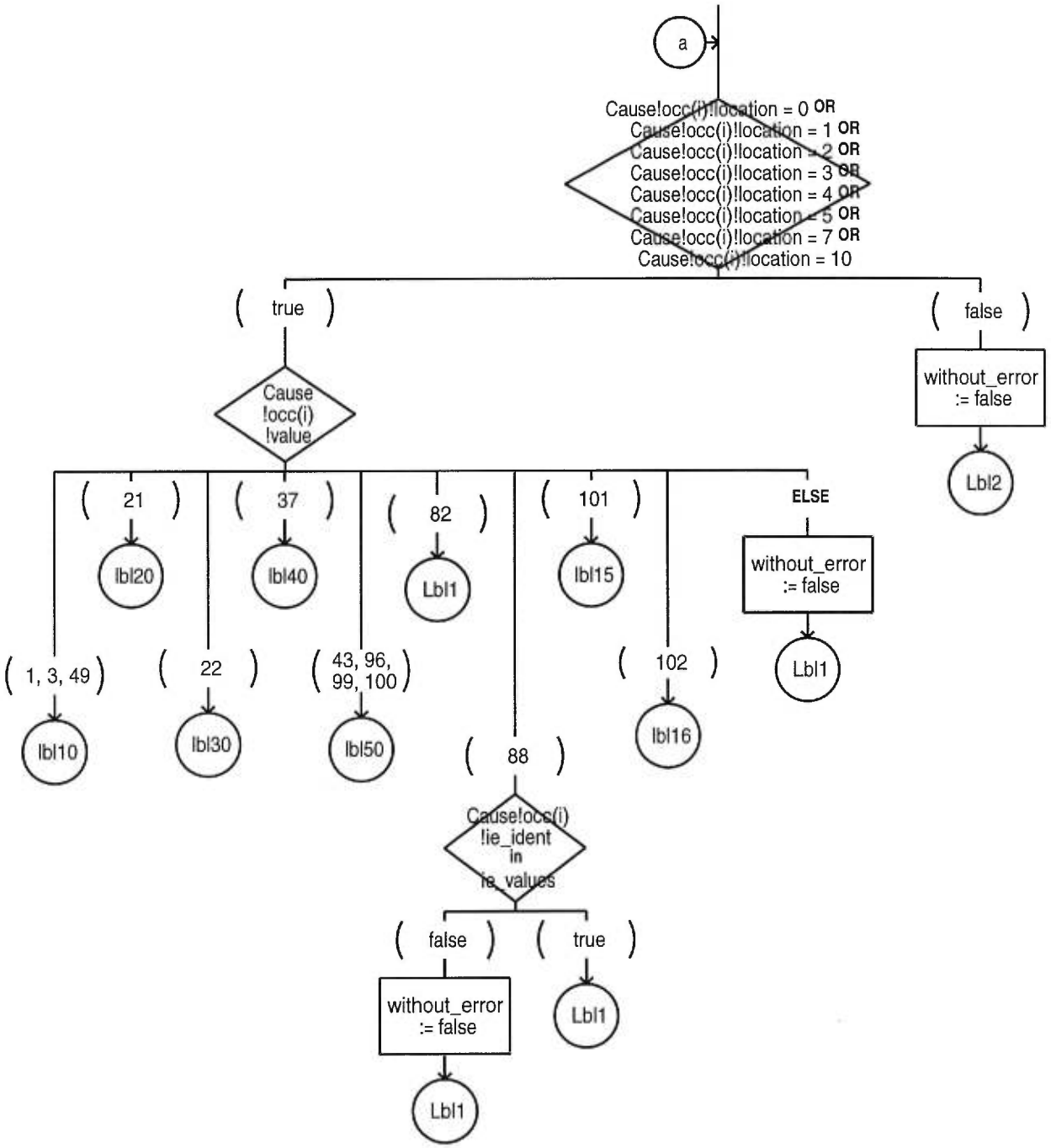

procedure B_sending_complete_verif
fpar
in ie B_sending_complete_type,
in/out error_free Boolean
error_free := true
ie !coding_standard = 0 and ieIndication = 33
true
false
error_free := false

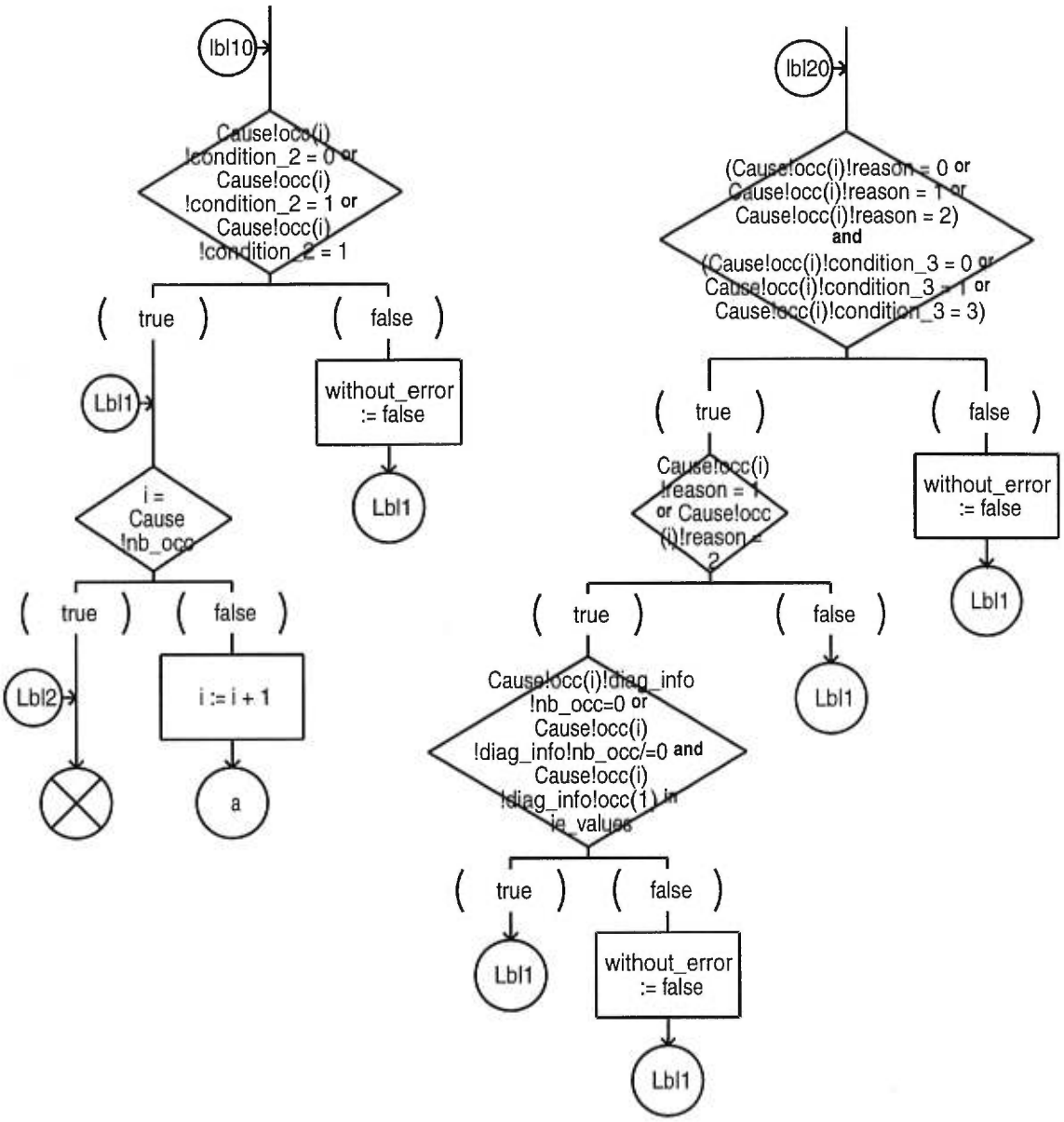

procedure call_state_verif
fpar
in ie call_state_type,
in/out error_free Boolean
ie!coding_standard = 0
and
(ie!code =0 OR ie!code=1 OR
ie!code=3 OR ie!code=6 OR
ie!code=8 OR ie!code=9 OR
ie!code=10 OR ie!code=11 OR
ie!code=12)
true
false
error_free
:= true
error_free
:= false

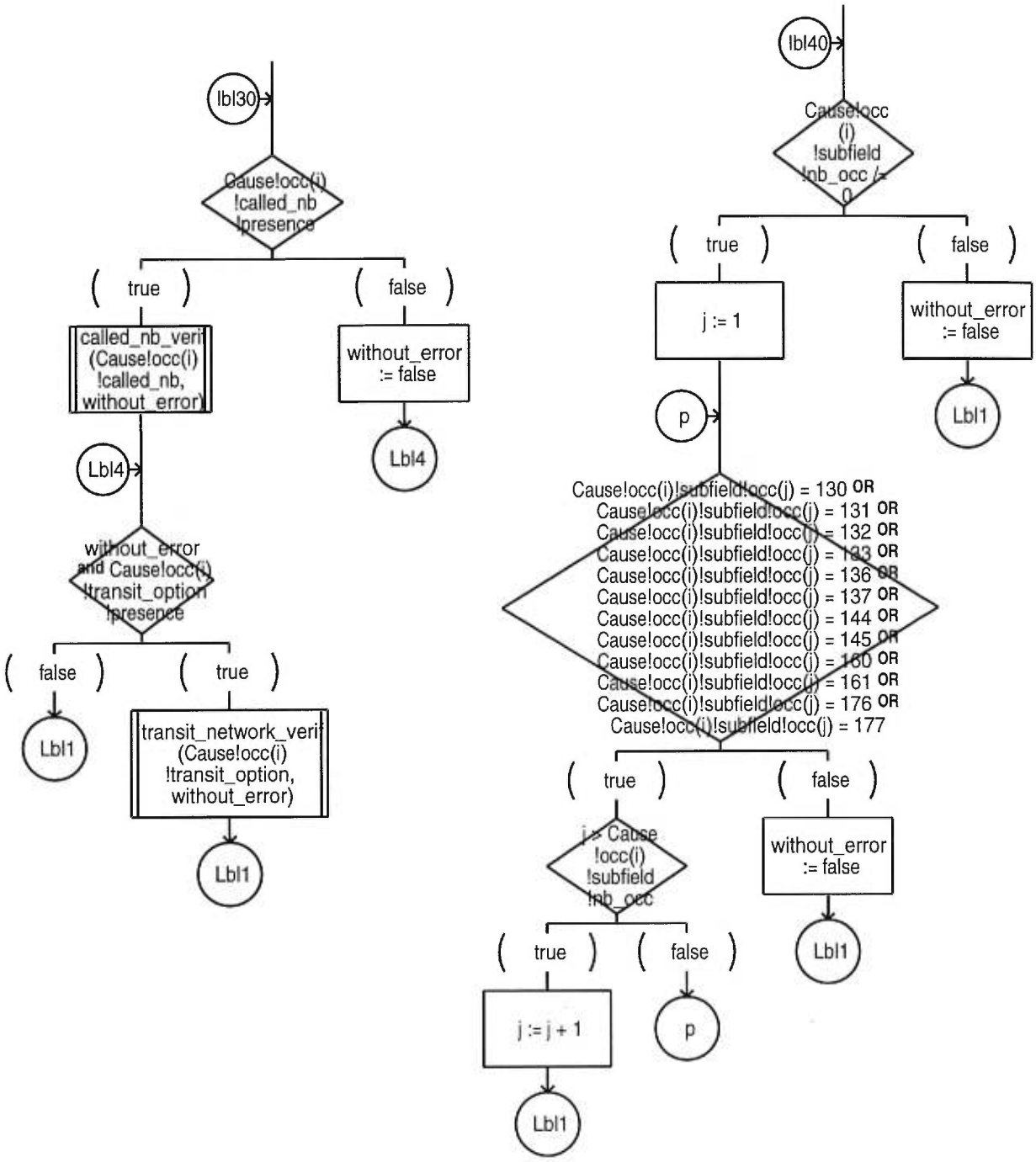

procedure called_nb_verif
fpar
in ie called_nb_type,
in/out error_free Boolean
ie
coding_standard
= 0
true
false
ie!plan_id = 1 and
ie!nb_type = 1 or
ie!plan_id = 2 and
ie!nb_type = 0
error_free
:= false
true
false
error_free
:= true
error_free
:= false

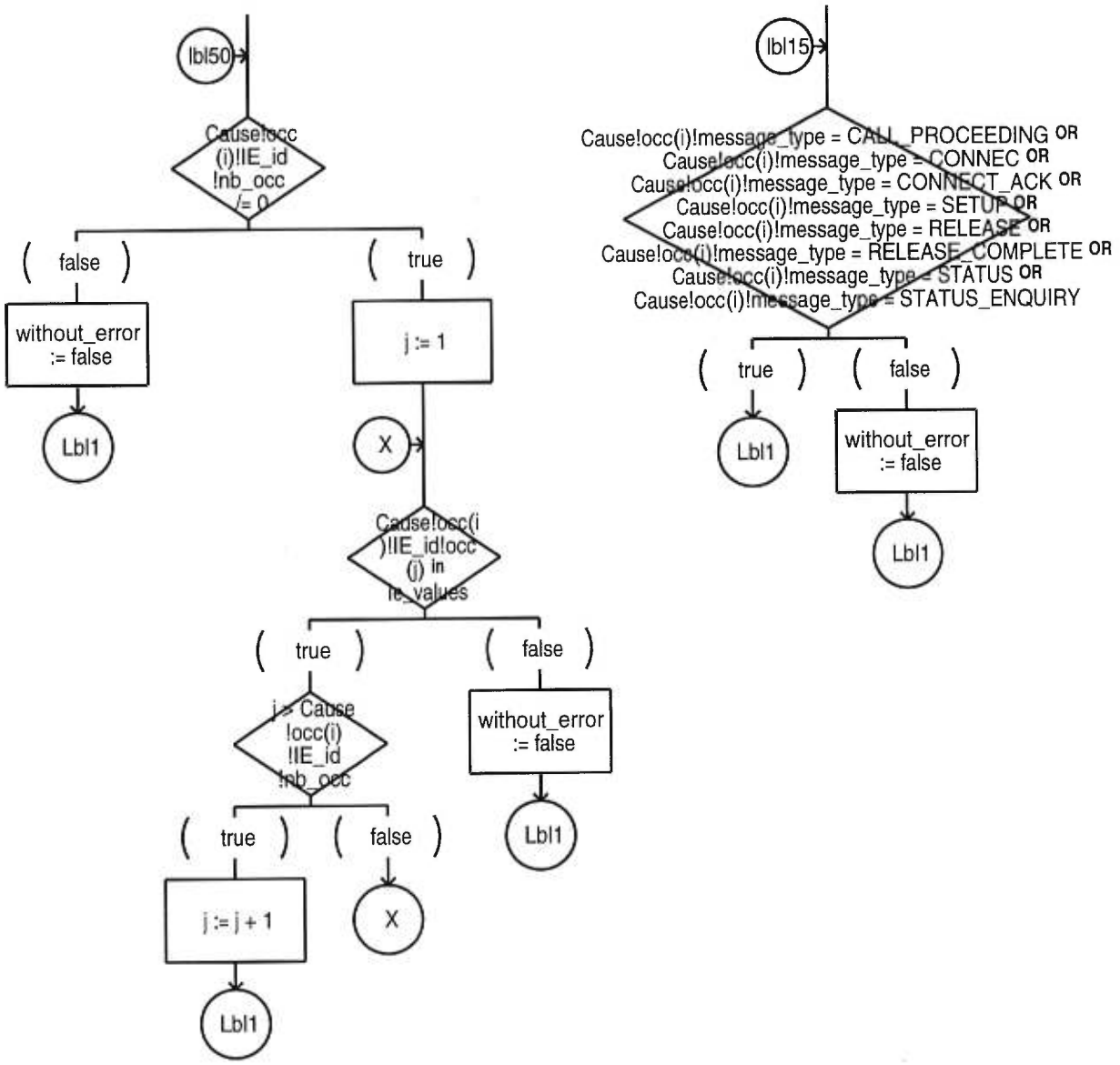

procedure calling_nb_verif
fpar
in ie calling_nb_type,
in/out error_free Boolean
ie
coding_standard
= 0
true
false
ie!plan_id = 1 and
ie!nb_type = 1 or
ie!plan_id = 2 and
ie!nb_type = 0
error_free
:= false
true
false
error_free
:= true
error_free
:= false

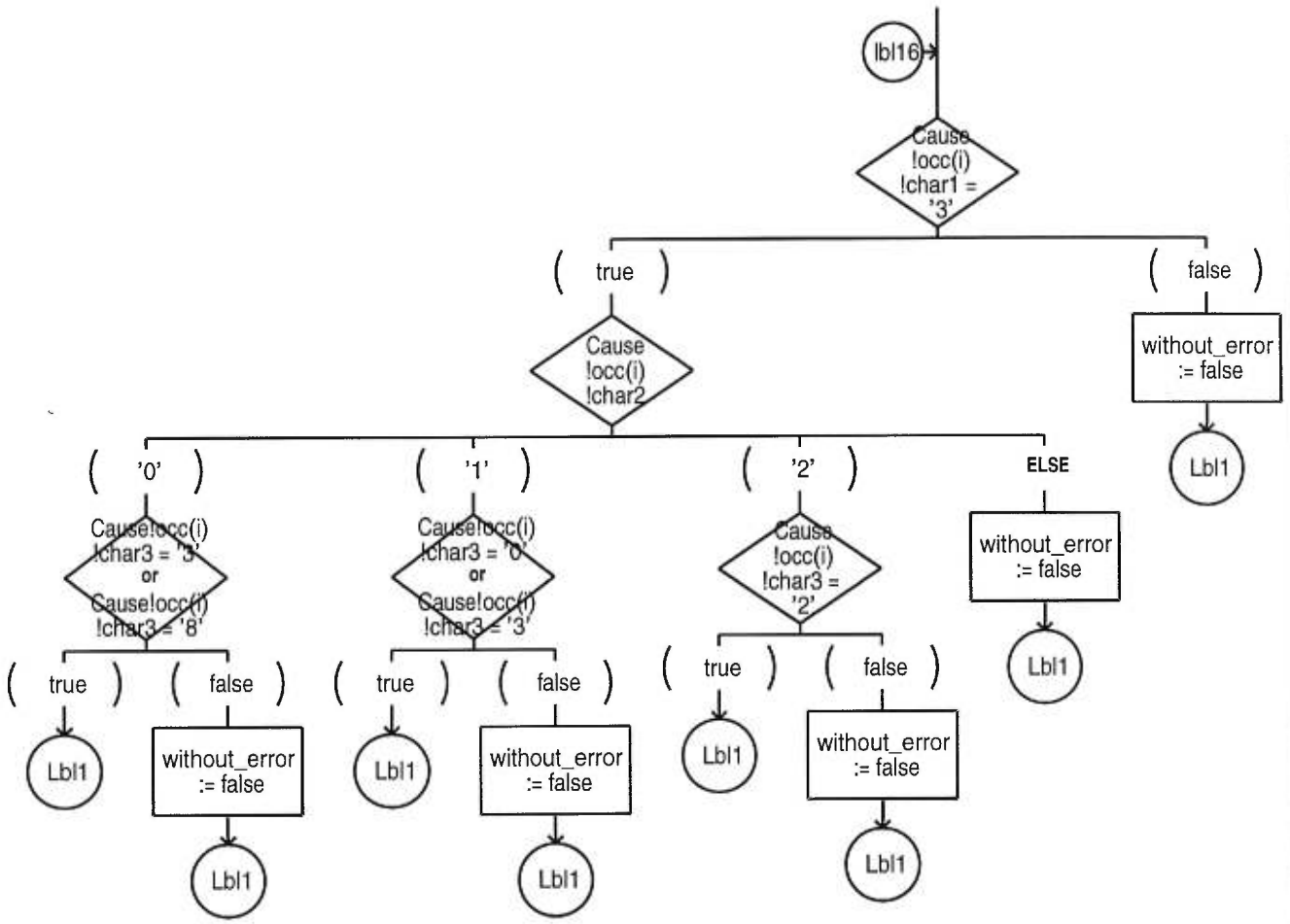

procedure cause_verif
fpar
in Cause cause_type,
in/out without_error Boolean
dcl
i,
j Natural,
ie_values ie_values_set;
ie_values := incl(8, ie_values),
ie_values := incl(20, ie_values),
ie_values := incl(84, ie_values),
ie_values := incl(85, ie_values),
ie_values := incl(88, ie_values),
ie_values := incl(89, ie_values),
ie_values := incl(90, ie_values),
ie_values := incl(92, ie_values),
ie_values := incl(93, ie_values),
ie_values := incl(94, ie_values),
ie_values := incl(95, ie_values),
ie_values := incl(96, ie_values),
ie_values := incl(97, ie_values),
ie_values := incl(98, ie_values),
ie_values := incl(99, ie_values),
ie_values := incl(108, ie_values),
ie_values := incl(109, ie_values),
ie_values := incl(112, ie_values),
ie_values := incl(113, ie_values),
ie_values := incl(120, ie_values)
without_error := true
Cause!occ (i) !coding_standard
(0)
i := 1
a
(3)
Cause !occ(i) !value = 23
(true)
Lbl2
(false)
without_error := false
Lbl2
ELSE
without_error := false
Lbl2

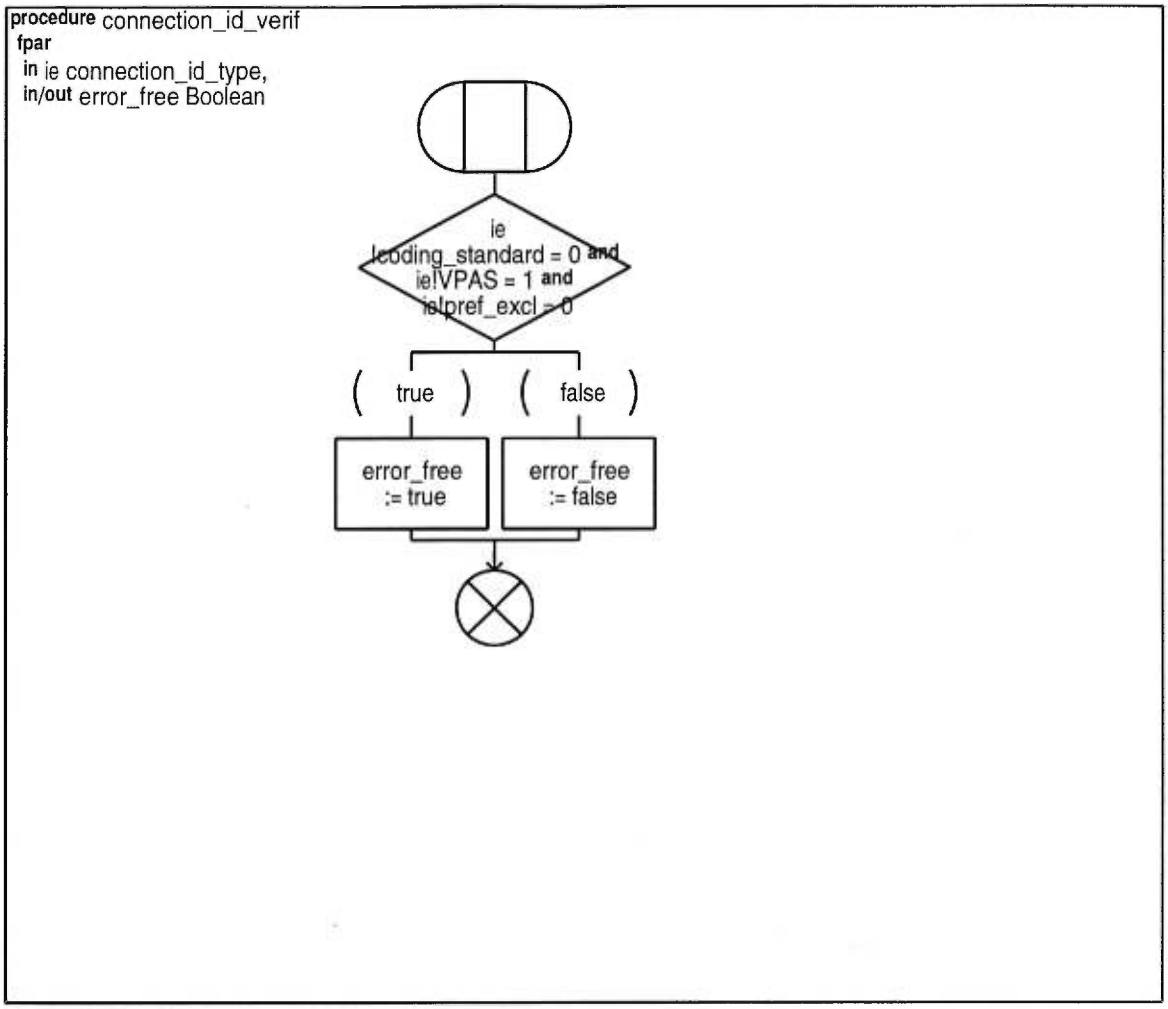

a
Cause!occ(i)!location = 0 OR
Cause!occ(i)!location = 1 OR
Cause!occ(i)!location = 2 OR
Cause!occ(i)!location = 3 OR
Cause!occ(i)!location = 4 OR
Cause!occ(i)!location = 5 OR
Cause!occ(i)!location = 7 OR
Cause!occ(i)!location = 10
true
false
without_error := false
Lbl2
Cause !occ(i) !value
21
37
82
101
ELSE
lbl20
lbl40
Lbl1
lbl15
without_error := false
Lbl1
1, 3, 49
22
43, 96, 99, 100
102
lbl10
lbl30
lbl50
lbl16
88
Cause!occ(i) !ie_ident in ie_values
false
true
without_error := false
Lbl1
Lbl1


lbl10
Cause!occ(i)
!condition_2 = 0 or
Cause!occ(i)
!condition_2 = 1 or
Cause!occ(i)
!condition_2 = 1
true
false
Lbl1
without_error
:= false
Lbl1
i =
Cause
!nb_occ
true
false
Lbl2
i := i + 1
a
lbl20
(Cause!occ(i)!reason = 0 or
Cause!occ(i)!reason = 1 or
Cause!occ(i)!reason = 2)
and
(Cause!occ(i)!condition_3 = 0 or
Cause!occ(i)!condition_3 = 1 or
Cause!occ(i)!condition_3 = 3)
true
false
Cause!occ(i)
!reason = 1
or Cause!occ
(i)!reason =
2
without_error
:= false
Lbl1
true
false
Lbl1
Cause!occ(i)!diag_info
!nb_occ=0 or
Cause!occ(i)
!diag_info!nb_occ/=0 and
Cause!occ(i)
!diag_info!occ(1) in
ie_values
true
false
Lbl1
without_error
:= false
Lbl1

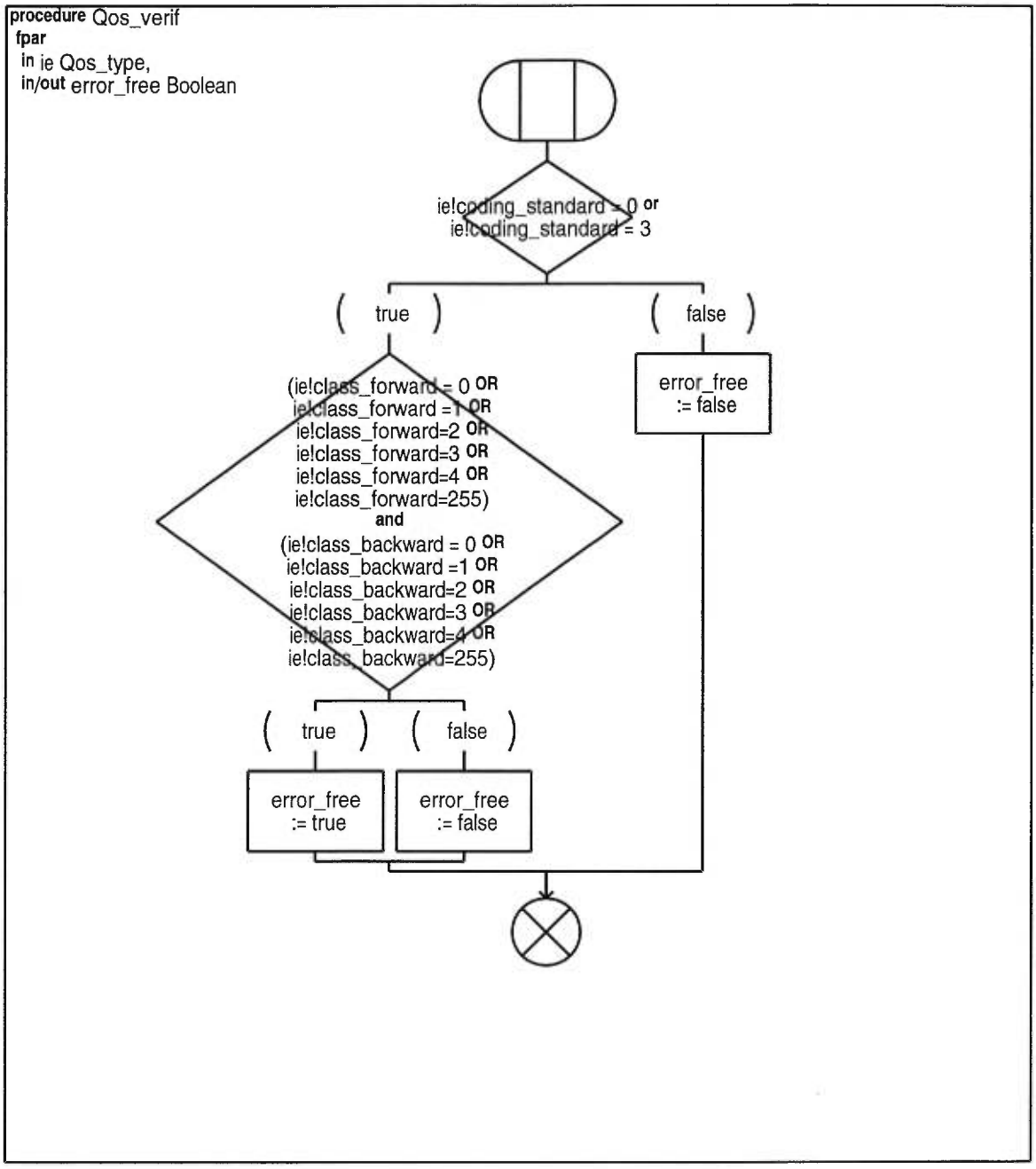

lbl30
Cause!occ(i)
!called_nb
!presence
true
false
called_nb_veri
(Cause!occ(i)
!called_nb,
without_error)
without_error
:= false
Lbl4
Lbl4
without_error
and Cause!occ(i)
!transit_option
!presence
false
true
Lbl1
transit_network_veri
(Cause!occ(i)
!transit_option,
without_error)
Lbl1
lbl40
Cause!occ
(i)
!subfield
!nb_occ /=
0
true
false
j := 1
without_error
:= false
p
Lbl1
Cause!occ(i)!subfield!occ(j) = 130 OR
Cause!occ(i)!subfield!occ(j) = 131 OR
Cause!occ(i)!subfield!occ(j) = 132 OR
Cause!occ(i)!subfield!occ(j) = 133 OR
Cause!occ(i)!subfield!occ(j) = 136 OR
Cause!occ(i)!subfield!occ(j) = 137 OR
Cause!occ(i)!subfield!occ(j) = 144 OR
Cause!occ(i)!subfield!occ(j) = 145 OR
Cause!occ(i)!subfield!occ(j) = 160 OR
Cause!occ(i)!subfield!occ(j) = 161 OR
Cause!occ(i)!subfield!occ(j) = 176 OR
Cause!occ(i)!subfield!occ(j) = 177
true
false
j > Cause
!occ(i)
!subfield
!nb_occ
without_error
:= false
Lbl1
true
false
j := j + 1
p
Lbl1

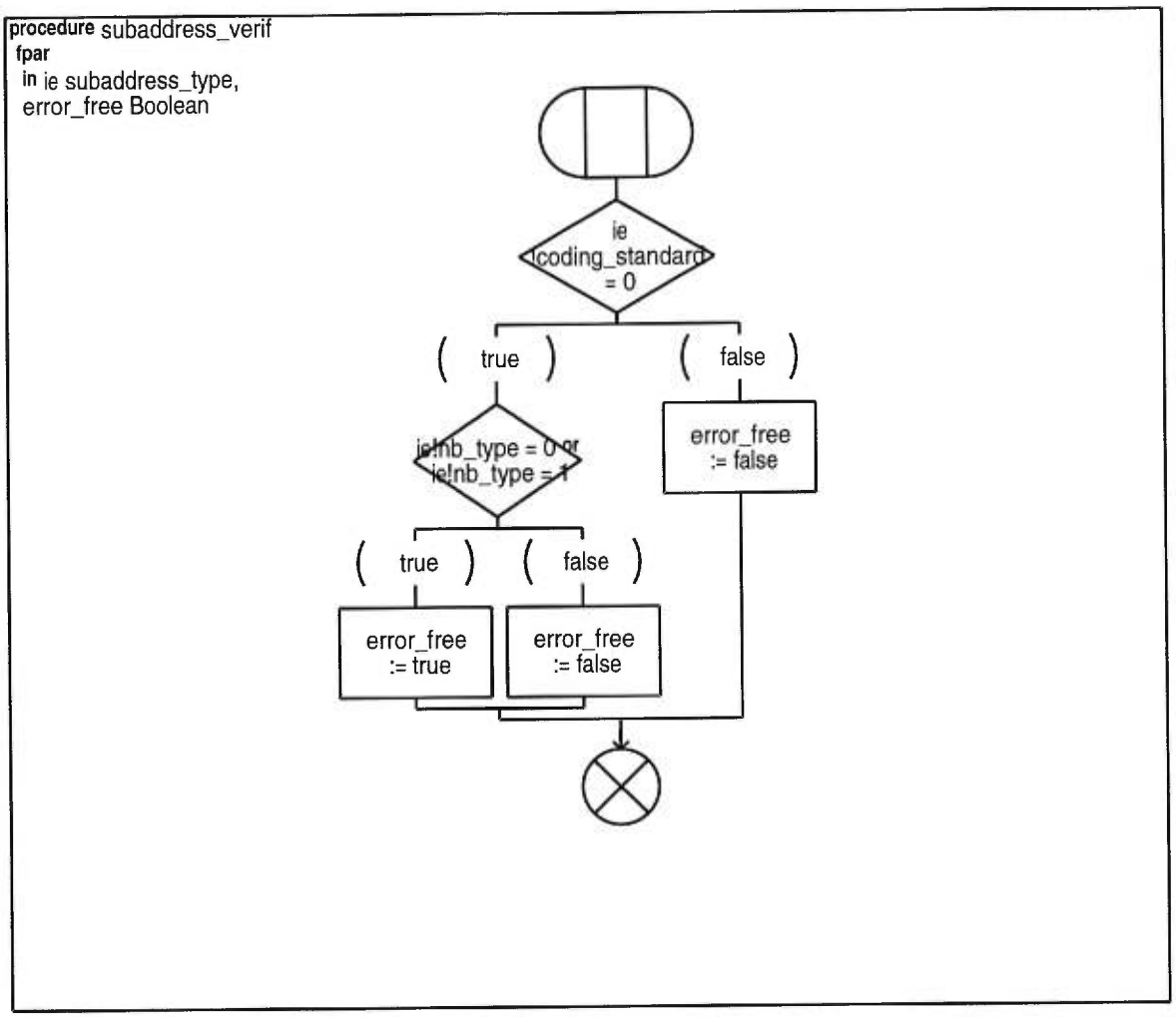

lbl50
Cause!occ (i)!IE_id !nb_occ /= 0
false
true
without_error := false
Lbl1
j := 1
X
Cause!occ(i)!IE_id!occ (j) in ie_values
true
false
j := Cause !occ(i) !IE_id !nb_occ
without_error := false
Lbl1
true
false
j := j + 1
X
Lbl1
lbl15
Cause!occ(i)!message_type = CALL_PROCEEDING OR
Cause!occ(i)!message_type = CONNEC OR
Cause!occ(i)!message_type = CONNECT_ACK OR
Cause!occ(i)!message_type = SETUP OR
Cause!occ(i)!message_type = RELEASE OR
Cause!occ(i)!message_type = RELEASE_COMPLETE OR
Cause!occ(i)!message_type = STATUS OR
Cause!occ(i)!message_type = STATUS_ENQUIRY
true
false
Lbl1
without_error := false
Lbl1

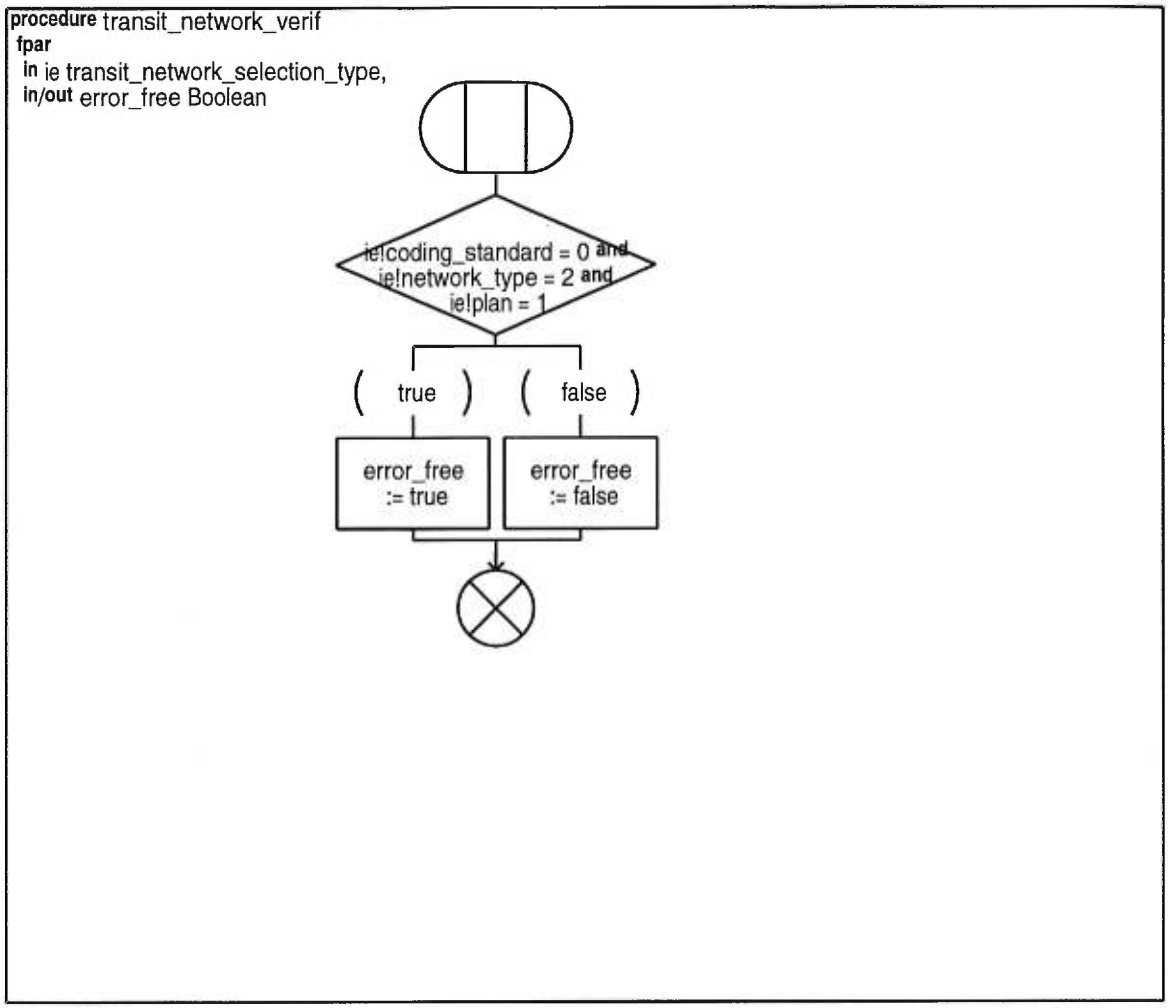

lbl16
Cause !occ(i) !char1 = '3'
true
false
without_error := false
Lbl1
Cause !occ(i) !char2
'0'
'1'
'2'
ELSE
Cause!occ(i) !char3 = '3' or Cause!occ(i) !char3 = '8'
Cause!occ(i) !char3 = '0' or Cause!occ(i) !char3 = '3'
Cause !occ(i) !char3 = '2'
without_error := false
Lbl1

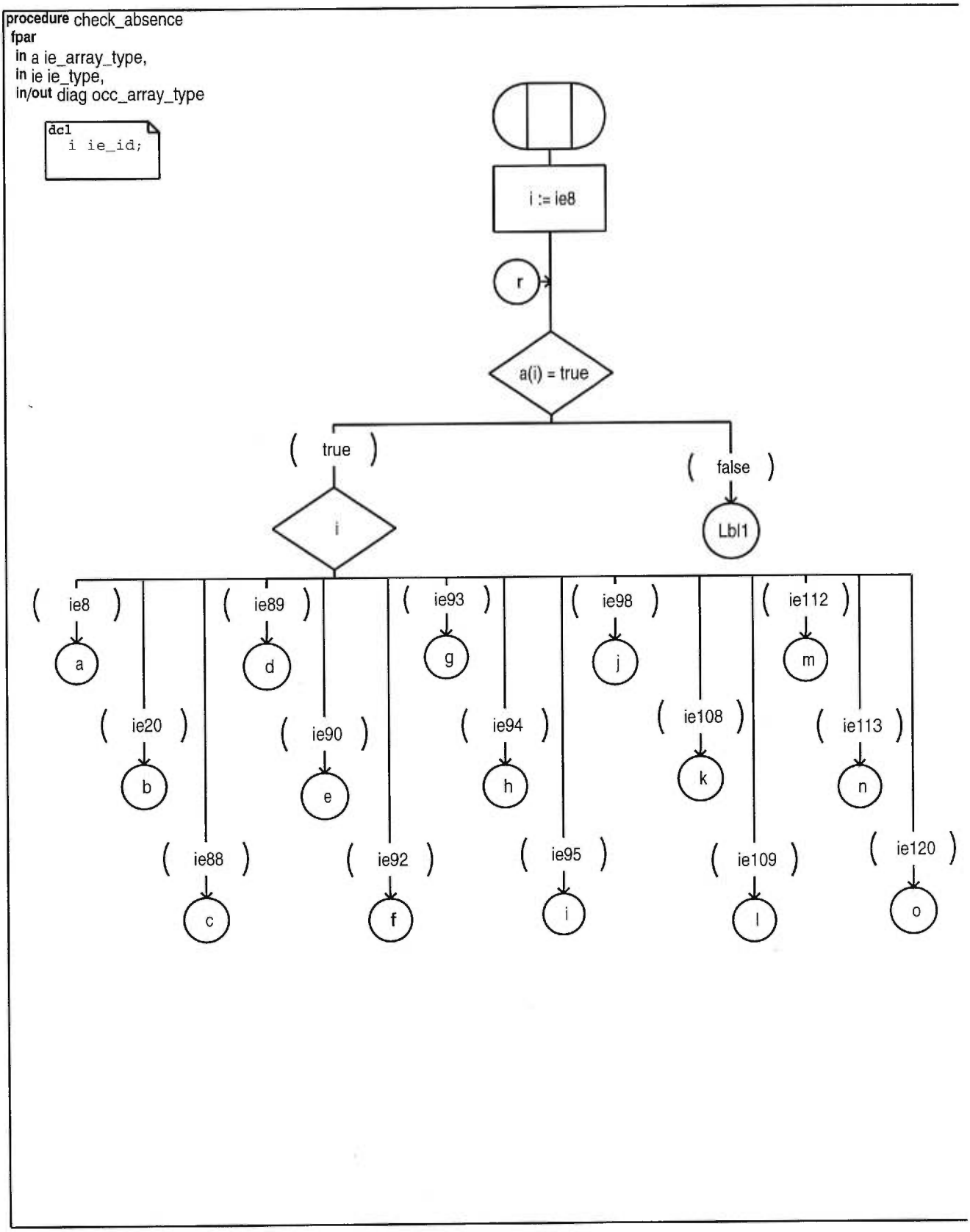


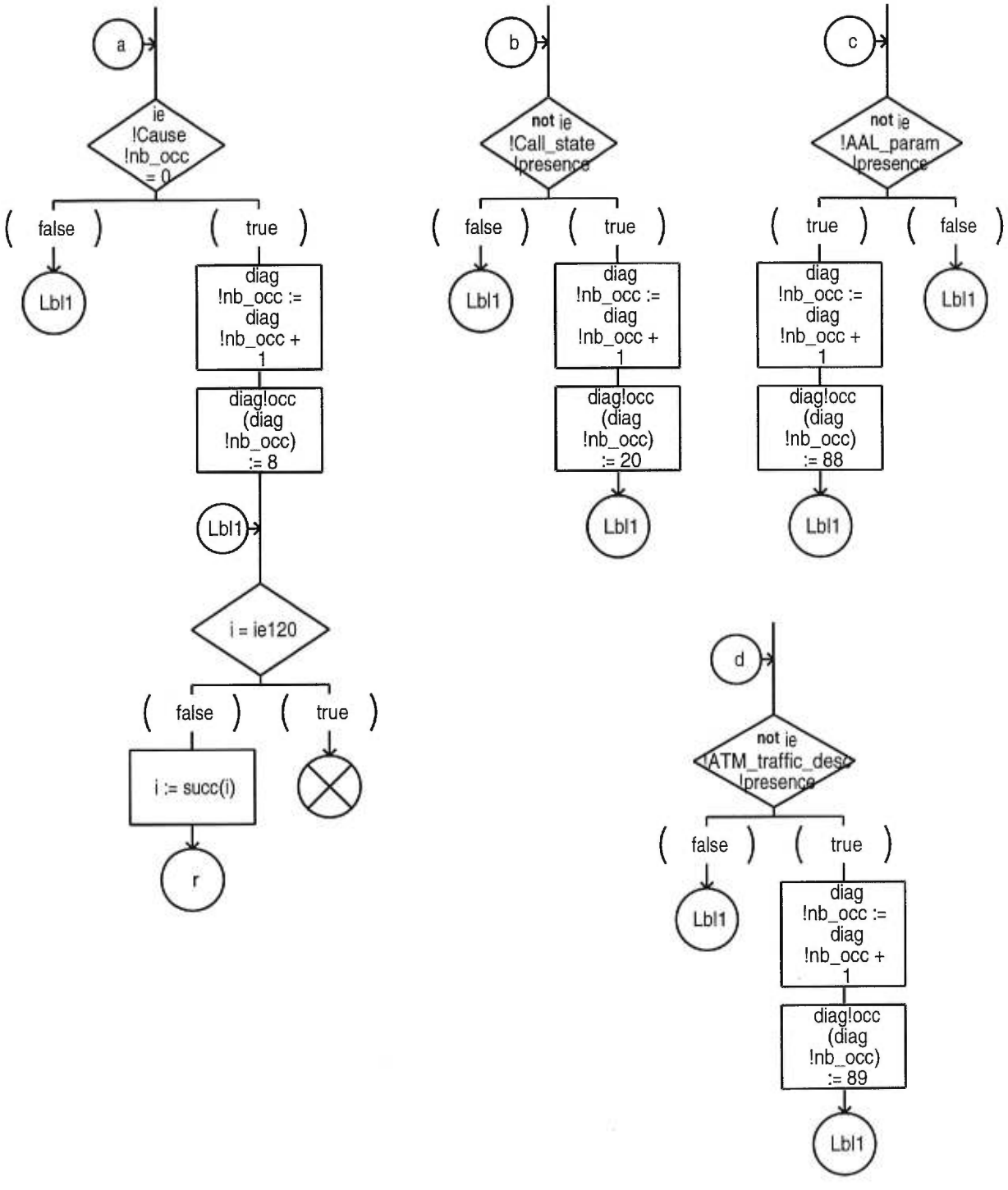
Author: Roxana Marcoci	File name: atm_sign.pr	connection_id_verif, page: 1

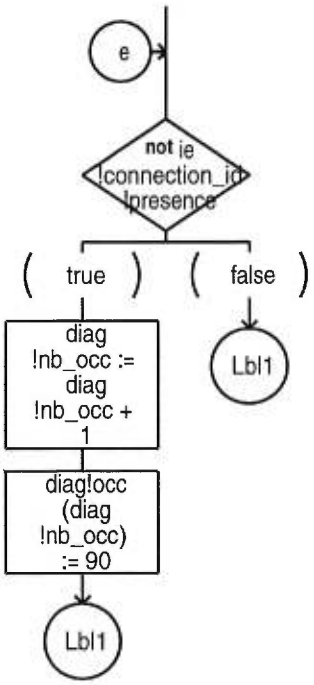

procedure Qos_verif
fpar
in ie Qos_type,
in/out error_free Boolean
ie!coding_standard = 0 or
ie!coding_standard = 3
true
false
error_free
:= false
(ie!class_forward = 0 OR
ie!class_forward =1 OR
ie!class_forward=2 OR
ie!class_forward=3 OR
ie!class_forward=4 OR
ie!class_forward=255)
and
(ie!class_backward = 0 OR
ie!class_backward =1 OR
ie!class_backward=2 OR
ie!class_backward=3 OR
ie!class_backward=4 OR
ie!class_backward=255)
true
false
error_free
:= true
error_free
:= false

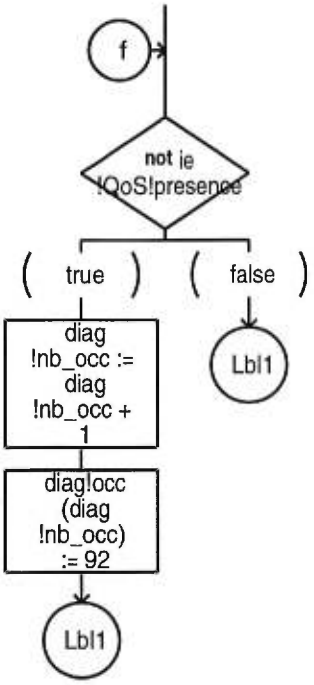

procedure subaddress_verif
fpar
in ie subaddress_type,
error_free Boolean
ie
coding_standard
= 0
true
false
ie!nb_type = 0 or
ie!nb_type = 1
error_free
:= false
true
false
error_free
:= true
error_free
:= false

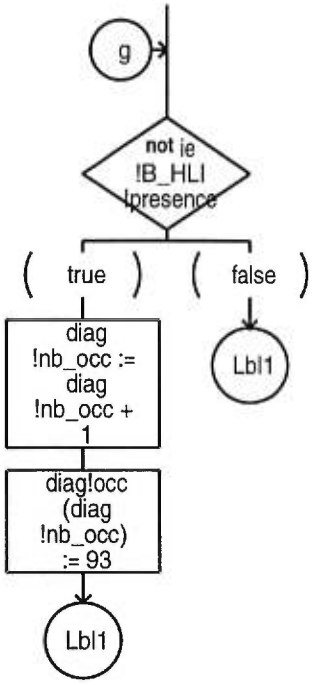

procedure transit_network_verif
fpar
in ie transit_network_selection_type,
in/out error_free Boolean
ie!coding_standard = 0 and
ie!network_type = 2 and
ie!plan = 1
true
false
error_free
:= true
error_free
:= false

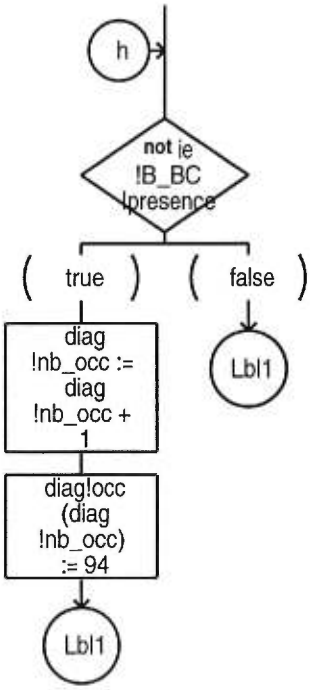

procedure check_absence
fpar
in a ie_array_type,
in ie ie_type,
in/out diag occ_array_type
dcl
i ie_id;
i := ie8
r
a(i) = true
(true)
(false)
i
Lbl1
(ie8)
(ie89)
(ie93)
(ie98)
(ie112)
a
d
g
j
m
(ie20)
(ie90)
(ie94)
(ie108)
(ie113)
b
e
h
k
n
(ie88)
(ie92)
(ie95)
(ie109)
(ie120)
c
f
i
l
o

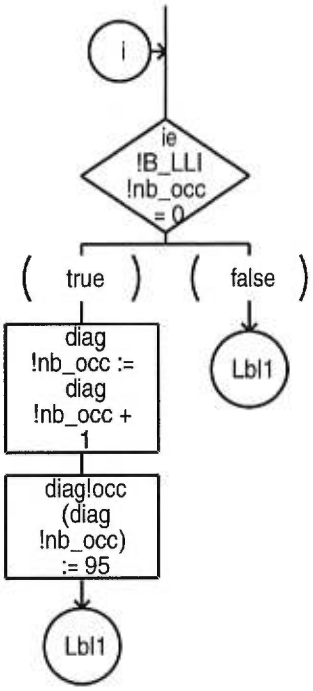

a
ie
!Cause
!nb_occ
= 0
false
true
Lbl1
diag
!nb_occ :=
diag
!nb_occ +
1
diag!occ
(diag
!nb_occ)
:= 8
Lbl1
i = ie120
false
true
i := succ(i)
r
b
not ie
!Call_state
!presence
false
true
Lbl1
diag
!nb_occ :=
diag
!nb_occ +
1
diag!occ
(diag
!nb_occ)
:= 20
Lbl1
c
not ie
!AAL_param
!presence
true
false
diag
!nb_occ :=
diag
!nb_occ +
1
diag!occ
(diag
!nb_occ)
:= 88
Lbl1
Lbl1
d
not ie
!ATM_traffic_desc
!presence
false
true
Lbl1
diag
!nb_occ :=
diag
!nb_occ +
1
diag!occ
(diag
!nb_occ)
:= 89
Lbl1

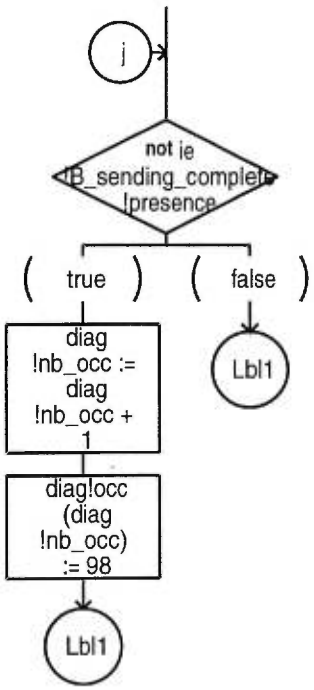

e
not ie
!connection_id
!presence
true
false
diag
!nb_occ :=
diag
!nb_occ +
1
diag!occ
(diag
!nb_occ)
:= 90
Lbl1
Lbl1

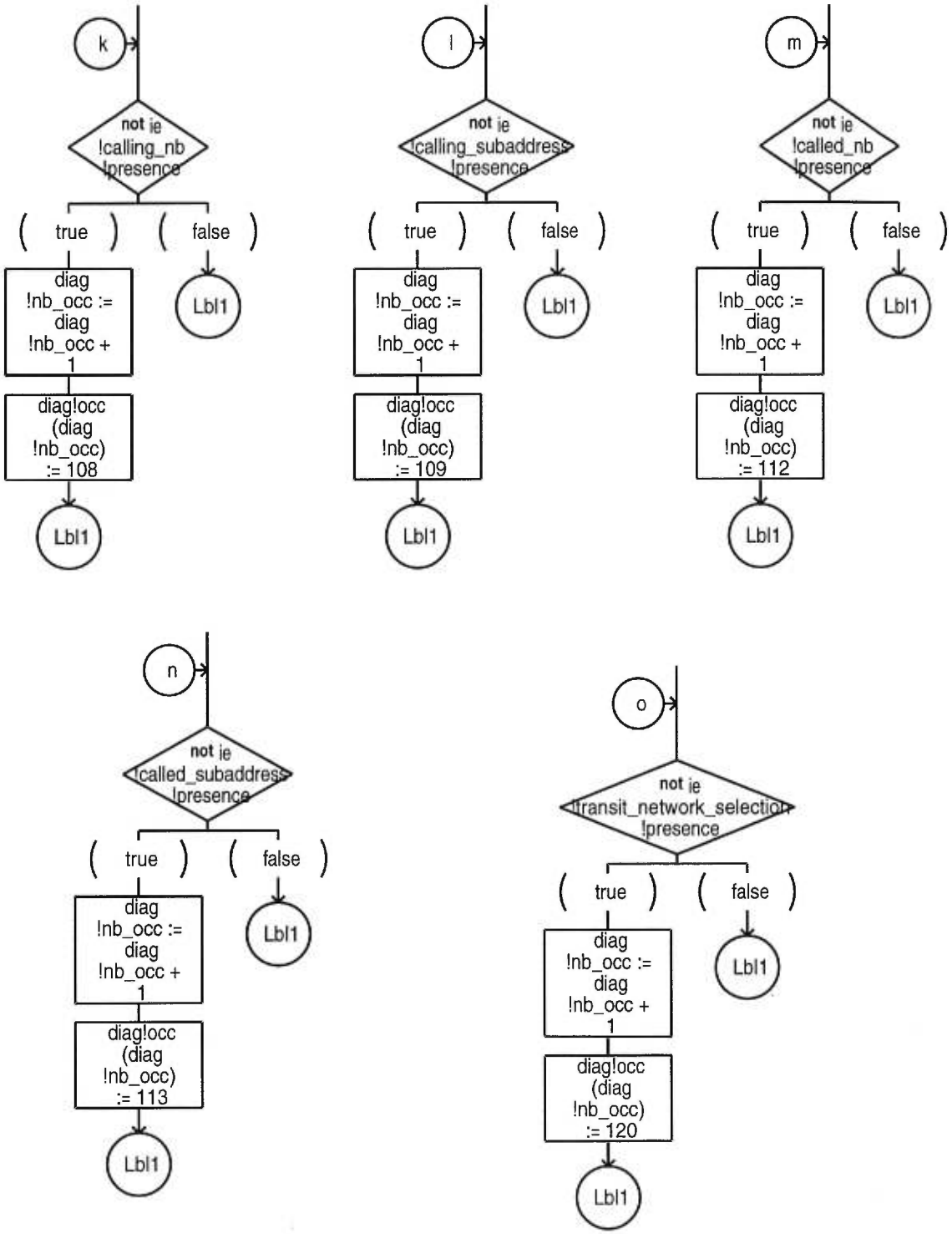

f
not ie
!QoS!presence
true
false
diag
!nb_occ :=
diag
!nb_occ +
1
diag!occ
(diag
!nb_occ)
:= 92
Lbl1
Lbl1

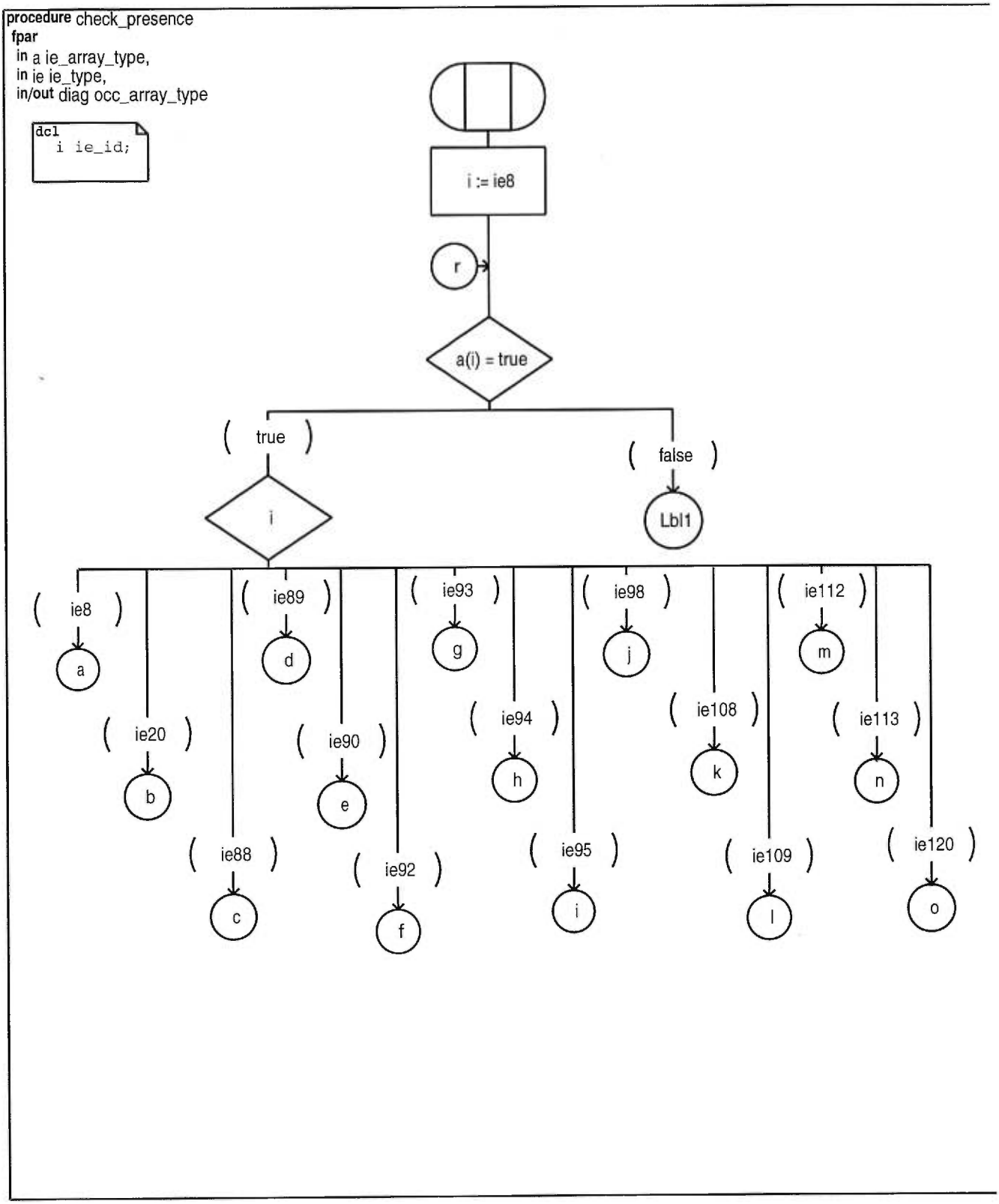

g
not ie
!B_HLI
!presence
true
false
diag
!nb_occ :=
diag
!nb_occ +
1
diag!occ
(diag
!nb_occ)
:= 93
Lbl1
Lbl1

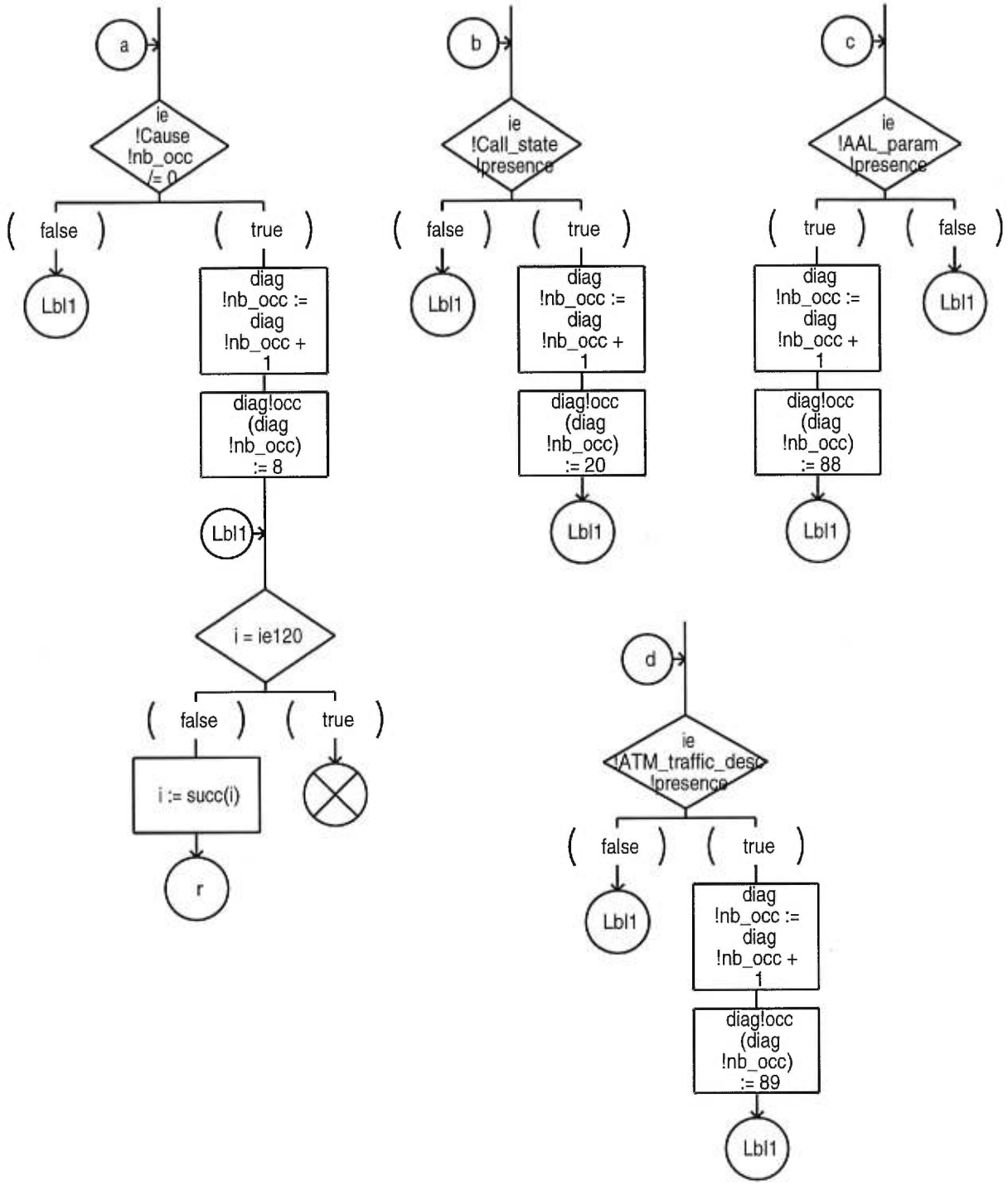

h
not ie
!B_BC
!presence
true
false
diag
!nb_occ :=
diag
!nb_occ +
1
diag!occ
(diag
!nb_occ)
:= 94
Lbl1
Lbl1

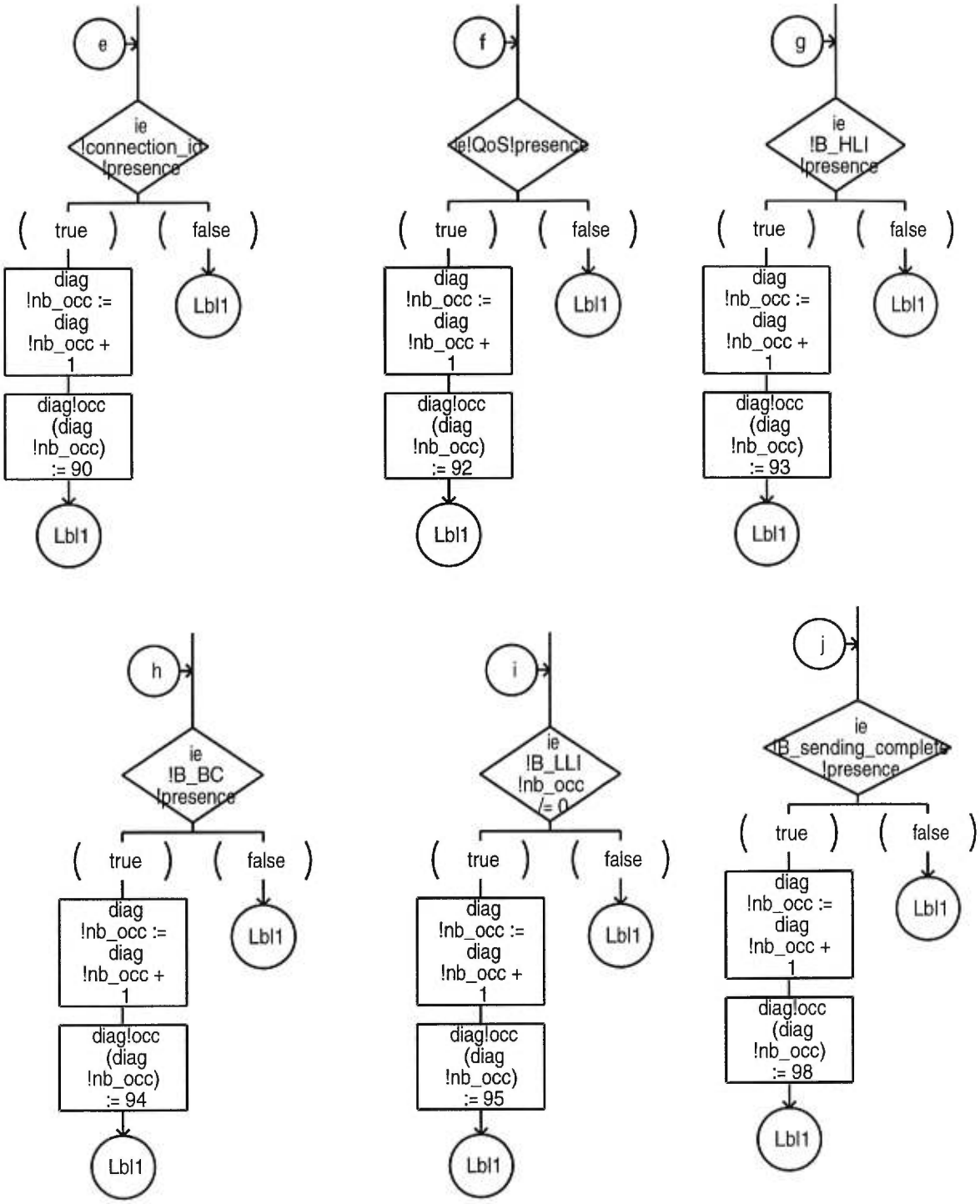

i
ie
!B_LLI
!nb_occ
= 0
true
false
diag
!nb_occ :=
diag
!nb_occ +
1
diag!occ
(diag
!nb_occ)
:= 95
Lbl1
Lbl1

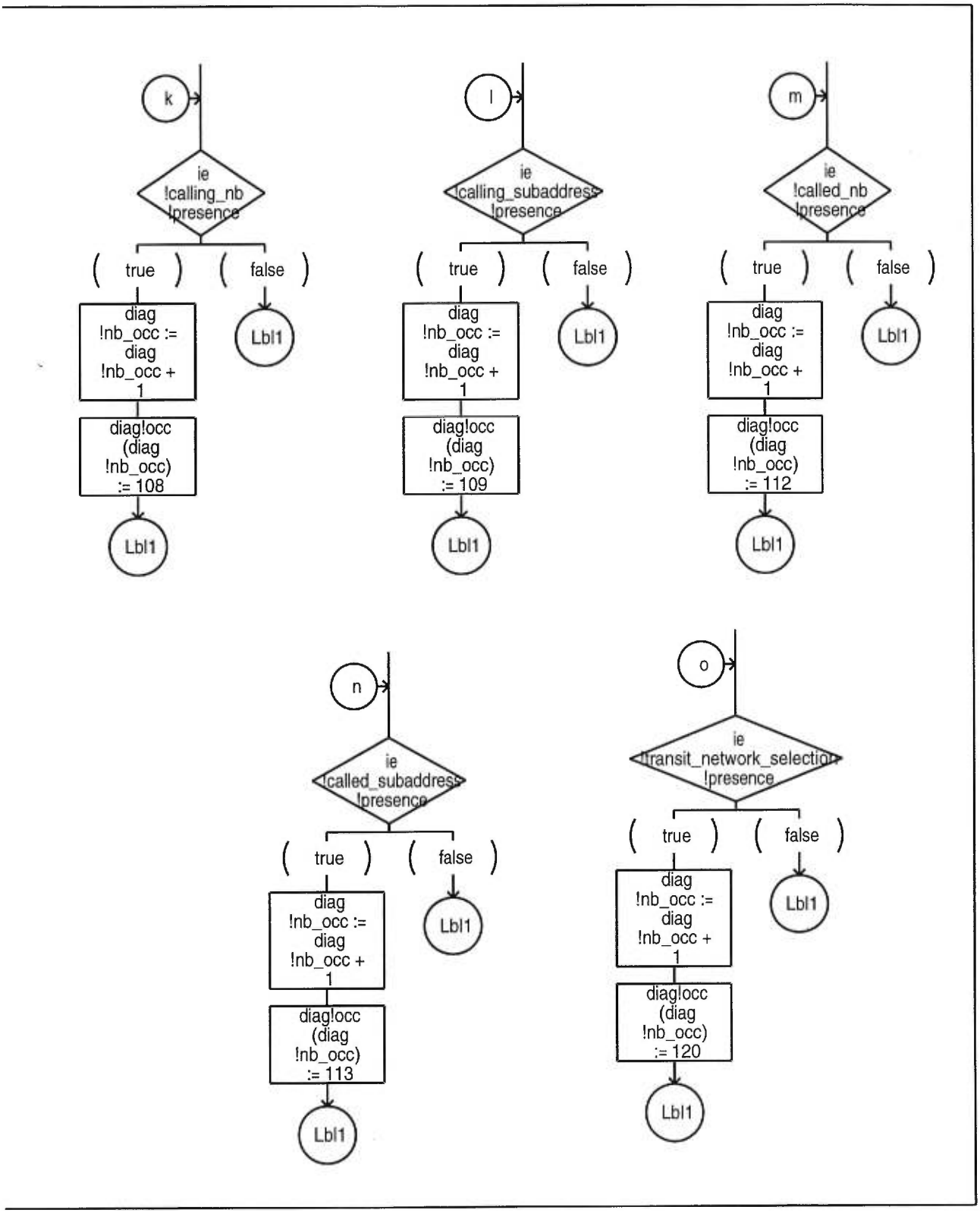

j
not ie
!B_sending_complete
!presence
true
false
diag
!nb_occ :=
diag
!nb_occ +
1
diag!occ
(diag
!nb_occ)
:= 98
Lbl1
Lbl1

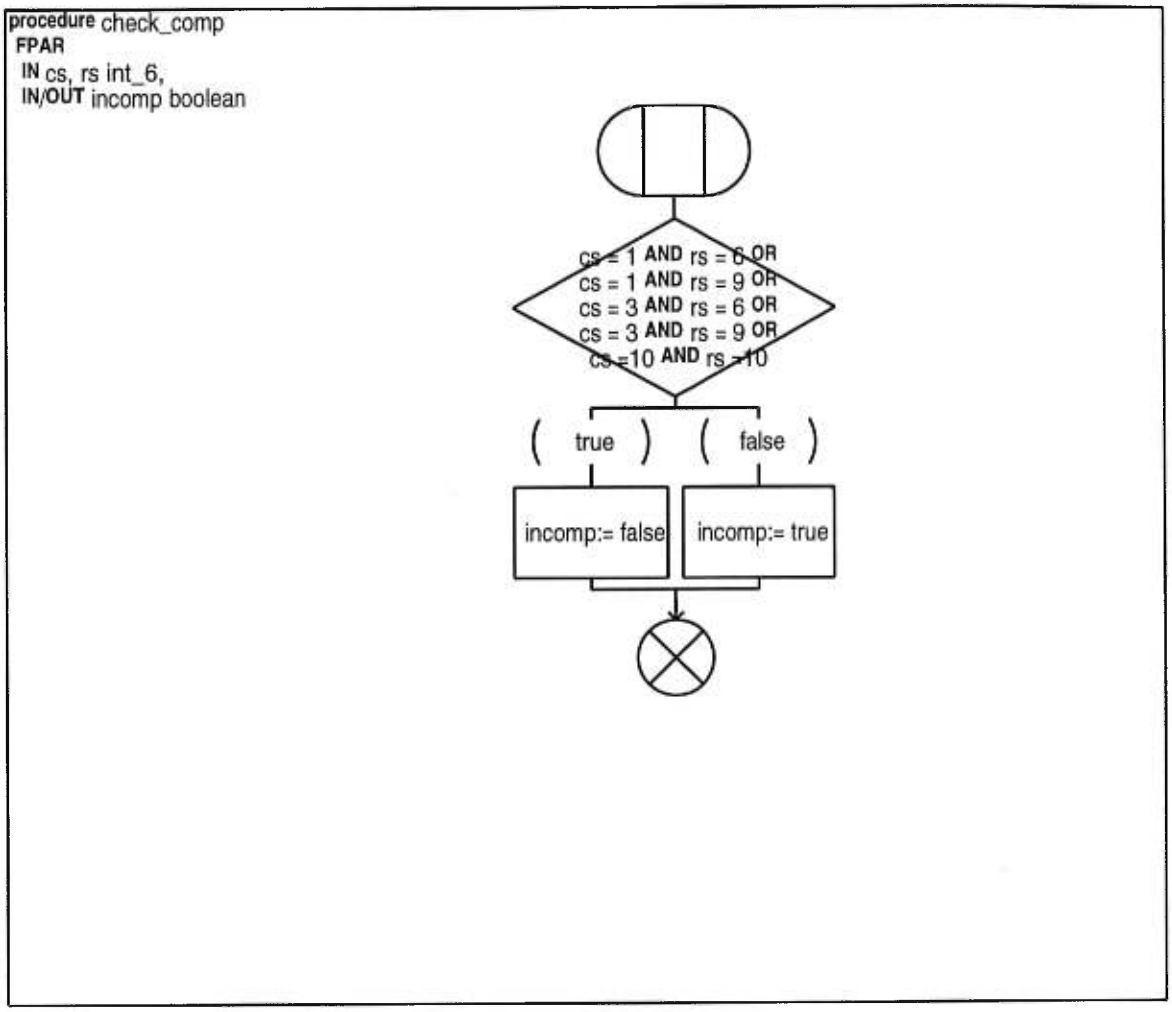

k
not je
!calling_nb
!presence
true
false
diag
!nb_occ :=
diag
!nb_occ +
1
diag!occ
(diag
!nb_occ)
:= 108
Lbl1
Lbl1
l
not je
!calling_subaddress
!presence
true
false
diag
!nb_occ :=
diag
!nb_occ +
1
diag!occ
(diag
!nb_occ)
:= 109
Lbl1
Lbl1
m
not je
!called_nb
!presence
true
false
diag
!nb_occ :=
diag
!nb_occ +
1
diag!occ
(diag
!nb_occ)
:= 112
Lbl1
Lbl1
n
not je
!called_subaddress
!presence
true
false
diag
!nb_occ :=
diag
!nb_occ +
1
diag!occ
(diag
!nb_occ)
:= 113
Lbl1
Lbl1
o
not je
!transit_network_selection
!presence
true
false
diag
!nb_occ :=
diag
!nb_occ +
1
diag!occ
(diag
!nb_occ)
:= 120
Lbl1
Lbl1

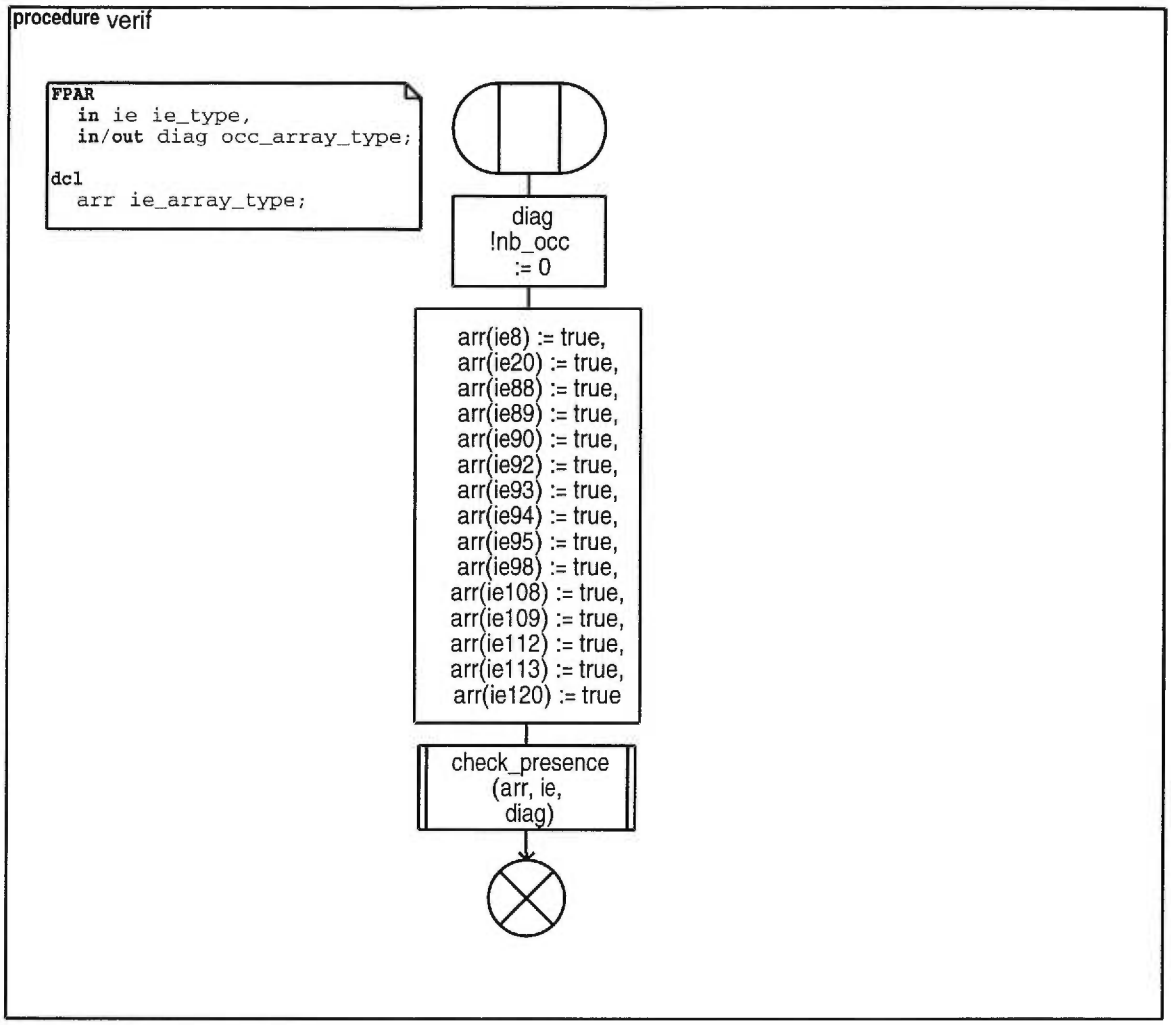

procedure check_presence
fpar
in a ie_array_type,
in ie ie_type,
in/out diag occ_array_type
dcl
i ie_id;
i := ie8
r
a(i) = true
true
false
i
Lbl1
ie8
ie89
ie93
ie98
ie112
a
d
g
j
m
ie20
ie90
ie94
ie108
ie113
b
e
h
k
n
ie88
ie92
ie95
ie109
ie120
c
f
i
l
o

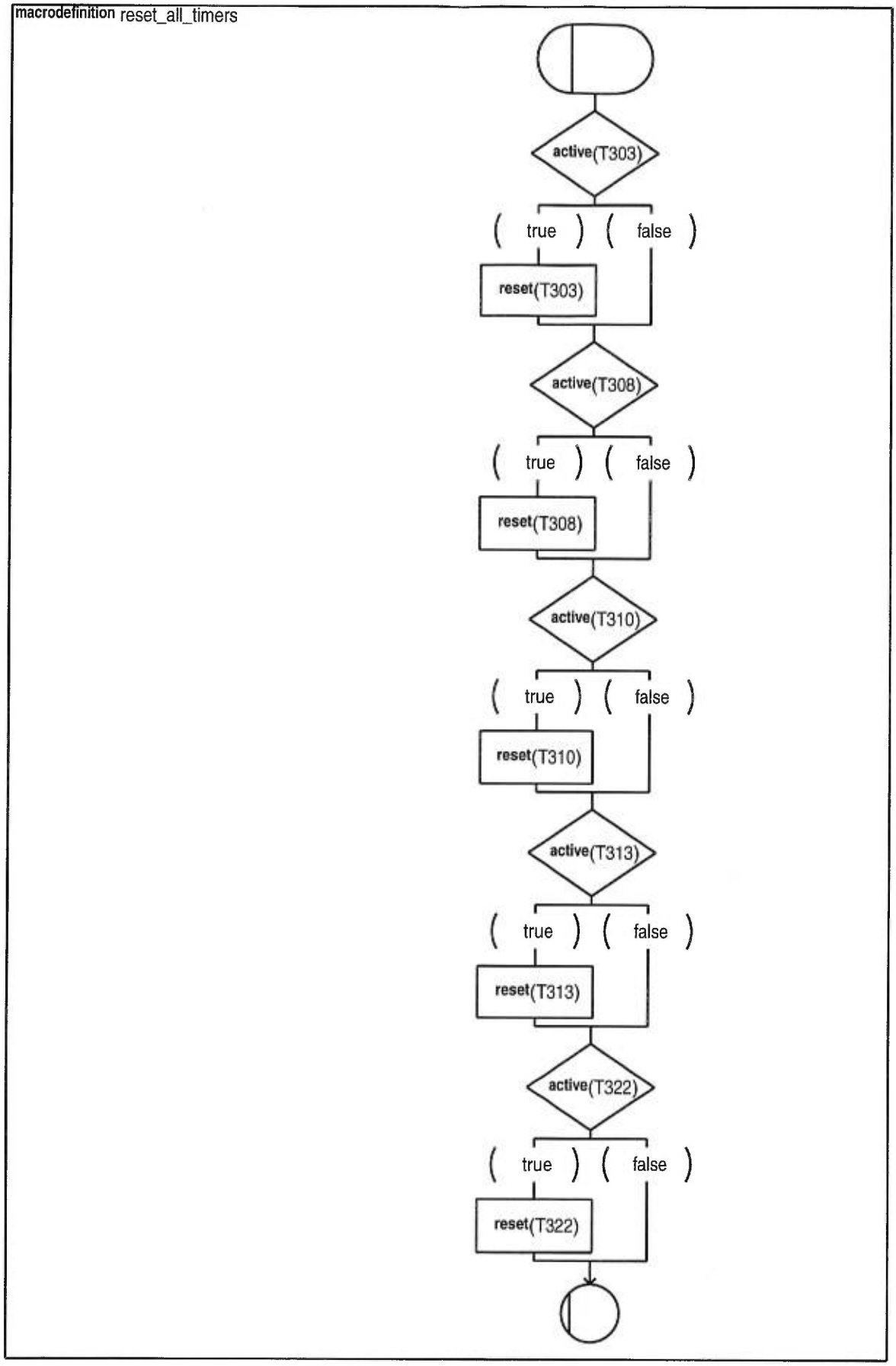

a
ie
!Cause
!nb_occ
/= 0
false
true
Lbl1
diag
!nb_occ :=
diag
!nb_occ +
1
diag!occ
(diag
!nb_occ)
:= 8
Lbl1
i = ie120
false
true
i := succ(i)
r
b
ie
!Call_state
!presence
false
true
Lbl1
diag
!nb_occ :=
diag
!nb_occ +
1
diag!occ
(diag
!nb_occ)
:= 20
Lbl1
c
ie
!AAL_param
!presence
true
false
diag
!nb_occ :=
diag
!nb_occ +
1
diag!occ
(diag
!nb_occ)
:= 88
Lbl1
Lbl1
d
ie
!ATM_traffic_desc
!presence
false
true
Lbl1
diag
!nb_occ :=
diag
!nb_occ +
1
diag!occ
(diag
!nb_occ)
:= 89
Lbl1


e
ie
!connection_id
!presence
true
false
diag
!nb_occ :=
diag
!nb_occ +
1
diag!occ
(diag
!nb_occ)
:= 90
Lbl1
Lbl1
f
ie!QoS!presence
true
false
diag
!nb_occ :=
diag
!nb_occ +
1
diag!occ
(diag
!nb_occ)
:= 92
Lbl1
Lbl1
g
ie
!B_HLI
!presence
true
false
diag
!nb_occ :=
diag
!nb_occ +
1
diag!occ
(diag
!nb_occ)
:= 93
Lbl1
Lbl1
h
ie
!B_BC
!presence
true
false
diag
!nb_occ :=
diag
!nb_occ +
1
diag!occ
(diag
!nb_occ)
:= 94
Lbl1
Lbl1
i
ie
!B_LLI
!nb_occ
/= 0
true
false
diag
!nb_occ :=
diag
!nb_occ +
1
diag!occ
(diag
!nb_occ)
:= 95
Lbl1
Lbl1
j
ie
!B_sending_complete
!presence
true
false
diag
!nb_occ :=
diag
!nb_occ +
1
diag!occ
(diag
!nb_occ)
:= 98
Lbl1
Lbl1


k
ie
!calling_nb
!presence
true
false
diag
!nb_occ :=
diag
!nb_occ +
1
diag!occ
(diag
!nb_occ)
:= 108
Lbl1
Lbl1
l
ie
!calling_subaddress
!presence
true
false
diag
!nb_occ :=
diag
!nb_occ +
1
diag!occ
(diag
!nb_occ)
:= 109
Lbl1
Lbl1
m
ie
!called_nb
!presence
true
false
diag
!nb_occ :=
diag
!nb_occ +
1
diag!occ
(diag
!nb_occ)
:= 112
Lbl1
Lbl1
n
ie
!called_subaddress
!presence
true
false
diag
!nb_occ :=
diag
!nb_occ +
1
diag!occ
(diag
!nb_occ)
:= 113
Lbl1
Lbl1
o
ie
!transit_network_selection
!presence
true
false
diag
!nb_occ :=
diag
!nb_occ +
1
diag!occ
(diag
!nb_occ)
:= 120
Lbl1
Lbl1


procedure check_comp
FPAR
IN cs, rs int_6,
IN/OUT incomp boolean
cs = 1 AND rs = 6 OR
cs = 1 AND rs = 9 OR
cs = 3 AND rs = 6 OR
cs = 3 AND rs = 9 OR
cs =10 AND rs =10
true
false
incomp:= false
incomp:= true


procedure verif
FPAR
in ie ie_type,
in/out diag occ_array_type;
dcl
arr ie_array_type;
diag
!nb_occ
:= 0
arr(ie8) := true,
arr(ie20) := true,
arr(ie88) := true,
arr(ie89) := true,
arr(ie90) := true,
arr(ie92) := true,
arr(ie93) := true,
arr(ie94) := true,
arr(ie95) := true,
arr(ie98) := true,
arr(ie108) := true,
arr(ie109) := true,
arr(ie112) := true,
arr(ie113) := true,
arr(ie120) := true
check_presence
(arr, ie,
diag)


macrodefinition reset_all_timers
active(T303)
true
false
reset(T303)
active(T308)
true
false
reset(T308)
active(T310)
true
false
reset(T310)
active(T313)
true
false
reset(T313)
active(T322)
true
false
reset(T322)