Remplacée par une version plus récente

UNION INTERNATIONALE DES TÉLÉCOMMUNICATIONS

UIT-T Z.100 Appendices I et II

SECTEUR DE LA NORMALISATION
DES TÉLÉCOMMUNICATIONS
DE L'UIT

(03/93)

LANGAGE DE PROGRAMMATION

DIRECTIVES RELATIVES À LA MÉTHODOLOGIE, POUR LE LANGAGE DE DESCRIPTION ET DE SPÉCIFICATION

BIBLIOGRAPHIE

Recommandation UIT-T Z.100 – Appendices I et II
Remplacée par une version plus récente

(Antérieurement «Recommandation du CCITT»)

Remplacée par une version plus récente

AVANT-PROPOS

L'UIT-T (Secteur de la normalisation des télécommunications) est un organe permanent de l'Union internationale des télécommunications (UIT). Il est chargé de l'étude des questions techniques, d'exploitation et de tarification, et émet à ce sujet des Recommandations en vue de la normalisation des télécommunications à l'échelle mondiale.

La Conférence mondiale de normalisation des télécommunications (CMNT), qui se réunit tous les quatre ans, détermine les thèmes que les Commissions d'études de l'UIT-T doivent examiner et à propos desquels elles doivent émettre des Recommandations.

Les Appendices I et II à la Recommandation UIT-T Z.100, élaborés par la Commission d'études X (1988-1993) de l'UIT-T, ont été approuvés par la CMNT (Helsinki, 1-12 mars 1993).

NOTES

1 Suite au processus de réforme entrepris au sein de l'Union internationale des télécommunications (UIT), le CCITT n'existe plus depuis le 28 février 1993. Il est remplacé par le Secteur de la normalisation des télécommunications de l'UIT (UIT-T) créé le 1er mars 1993. De même, le CCIR et l'IFRB ont été remplacés par le Secteur des radiocommunications.

Afin de ne pas retarder la publication de la présente Recommandation, aucun changement n'a été apporté aux mentions contenant les sigles CCITT, CCIR et IFRB ou aux entités qui leur sont associées, comme «Assemblée plénière», «Secrétariat», etc. Les futures éditions de la présente Recommandation adopteront la terminologie appropriée reflétant la nouvelle structure de l'UIT.

2 Dans la présente Recommandation, le terme «Administration» désigne indifféremment une administration de télécommunication ou une exploitation reconnue.

TABLE DES MATIÈRES

Appendice I à la Recommandation Z.100

DIRECTIVES RELATIVES À LA MÉTHODOLOGIE POUR LE LANGAGE DE DESCRIPTION ET DE SPÉCIFICATION

(Helsinki, 1993)

I.1 Introduction

I.1.1 Objectif

L'objectif de ce appendice est de fournir des directives pour l'usage effectif du language de description et de spécification (SDL) (*specification and description language*) dans le cadre d'une méthodologie générale.

Il est reconnu qu'il existe de nombreuses façons d'utiliser le SDL, que celles-ci soient liées à la préférence de l'utilisateur, ou bien à l'application cible, ou alors à des règles propres de l'organisation, etc. Les directives publiées dans cet appendice ne sont pas normatives. Il est laissé à l'appréciation des utilisateurs du SDL dans quelle mesure elles doivent être suivies. La raison de les publier est que l'on croit qu'elles sont utiles et que c'est aussi important pour la promotion du SDL et pour l'usage unifié du langage et un support par des outils.

I.1.2 Change d'application

Cet appendice fournit des directives pour les sujets suivants:

- l'usage du SDL pour différents domaines d'application;
- la production progressive d'une spécification SDL;
- l'usage des diagrammes de séquences de messages;
- la dérivation d'une mise en œuvre à partir d'une spécification SDL;
- les approches formelles de la validation, de la vérification et du test;
- les diagrammes auxiliaires;
- les aspects relatifs à la documentation.

Les directives ne constituent pas une méthodologie unique, cohérente et complète. Elles n'ont pas la prétention d'être exhaustives, ni spécifiques, ni d'un grand niveau de détail, ni de restreindre exagérément la liberté des utilisateurs. Elles sont destinées à être sélectionnées et incorporées par les utilisateurs du SDL dans leurs propres méthodologies générales et adaptées pour leurs systèmes et pour leurs besoins spécifiques.

I.1.3 Conventions

Les spécifications formelles ne doivent pas être rendues confuses par du texte en langue naturelle et des figures. Ceci est obtenu dans cet appendice normalement soit en traitant les spécifications formelles comme des exemples soit en les faisant figurer dans des paragraphes séparés. Les exemples sont numérotés si nécessaire, par exemple lorsqu'ils sont référencés. Une police de caractères courier est utilisée pour les spécifications formelles textuelles.

Une exemple peut couvrir une partie seulement d'une spécification SDL complète. Les parties manquantes sont indiquées par exemple par une ligne pointillée ou par une ligne continue non limitée.

I.1.4 Cadre

Selon un dictionnaire standard, une méthodologie est un système de méthodes et de principes utilisé dans une discipline particulière. Une méthode est soit une façon systématique de faire quelque chose soit les techniques ou l'aménagement du travail pour un domaine ou un sujet particulier. Dans le contexte de cet appendice, les disciplines concernées sont le développement de systèmes de télécommunication, les protocoles, etc appelés *systèmes d'application* (ou simplement *systèmes* pour des raisons de concision, le terme qualifiant pouvant être déduit du contexte).

La spécification et la conception sont des *activités*. Plusieurs méthodes différentes peuvent exister pour la même activité. Ainsi, l'évaluation des approches concurrentes et le choix de l'une d'entre elles constituent un point important lorsque l'on développe une méthodologie. Celle-ci peut-être basée sur plusieurs autres choses tirées des caractéristiques du système d'application.

Une activité est caractérisée par ses entrées et ses sorties. La conduite d'une activité est gouvernée par quelques règles qui sont déduites de la politique de l'entreprise concernée. Une activité est habituellement réalisée par des personnes appartenant à l'entreprise et faisant usage d'outils (voir la Figure I.1-1). Une activité peut être considérée comme une machine ou un système à états finis et peut être décrite en utilisant des techniques semblables (par exemple en utilisant SDL).

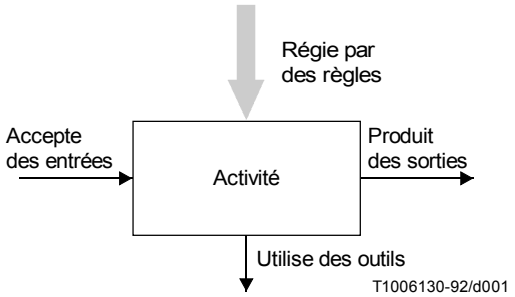


FIGURE I.1-1/Z.100

Spécification d'une activité

Une activité est généralement conduite sur la représentation (d'une partie) du système d'application. Par conséquent, la décomposition d'une activité en sous-activités doit se fonder sur un modèle d'information et sur l'architecture du système d'application.

Chaque élément d'information doit être exprimé dans un langage approprié (naturel, semi-formel, formel). Le monde réel est décrit finalement dans un langage naturel quelconque, qui convient pour exprimer des objectifs et des intentions et ne convient pas pour exprimer les détails de façon précise et non ambigüe. Un langage formel possède les propriétés contraires. Une méthodologie peut utiliser différentes catégories de langages, en faisant en sorte qu'ils se complètent les uns et les autres et qu'on exploite leurs avantages respectifs.

S'appuyant sur la précédente discussion, une méthodologie (dans le contexte de ce document) contient généralement les composants suivants:

- modèle d'entreprise;
- modèle d'activité;
- architecture de système;
- modèle d'information;
- langages;
- procédures opérationnelles;
- règles de gestion de produit et de documentation;
- utilisation des outils.

Le *modèle d'entreprise* décrit les objectifs généraux d'un système d'application en terme d'actions, de buts et de règles de conduite. Il spécifie les activités qui prennent place dans l'organisation utilisant le système d'application, les rôles que jouent les personnes dans l'organisation, et les interactions entre l'organisation, le système d'application et l'environnement dans lequel se situent le système d'application et l'organisation.

Le *modèle d'activité* identifie toutes les activités concernées par le système d'application et les relations entre elles. Chaque activité est décrite comme indiqué ci-dessus. Noter que le modèle d'activité est habituellement appelé modèle de cycle de vie. Ce terme devrait être évité, car il implique, de façon incorrecte, un enchaînement linéaire d'activités. Un modèle d'activité possède généralement deux dimensions, autorisant l'exécution parallèle d'activités.

L'*architecture d'application* fournit en principe une décomposition du système d'application en parties complètes par elles-mêmes et en règles définissant les interactions entre ces parties.

Le *modèle d'information* identifie toutes les informations pertinentes pour le système d'application à solliciter, produire, maintenir et livrer aussi bien que toutes les relations entre les différents informations.

Les *langages* fournissent les moyens d'exprimer le modèle d'information avec la précision voulue, en permettant de réaliser des contrôles et d'utiliser des outils informatiques.

Les *procédures opérationnelles* renforcent certaines règles communément admises et devant être suivies par tous les membres du projet, de façon à faire avancer réellement un travail et à produire des résultats de haute qualité.

Les *règles de gestion du produit et de la documentation* complètent les procédures opérationnelles et assurent la maintenabilité de la documentation de l'application.

Les *outils* augmentent la productivité des membres du projet et la qualité des produits en exécutant des tâches bien définies et réalisant un travail important.

NOTE – Une méthodologie couvre plusieurs niveaux d'abstraction. Le modèle d'activité couvre tous les niveaux d'abstraction, mais quelques autres composants de méthodologie peuvent ne pas s'appliquer à tous les niveaux d'abstraction. Les procédures opérationnelles par exemple s'appliquent seulement aux niveaux les plus bas.

I.1.5 Les implications de l'usage du SDL

SDL est un langage de spécification formel. Pour des raisons de concision, le terme *formel* sera normalement omis et sous-entendu dans la suite, lorsque l'on considérera les langages de spécification.

Il est probablement très largement accepté que la clé du succès d'un système se trouve à travers sa spécification et sa conception. Ceci exige, d'une part, un langage de spécification approprié, satisfaisant les besoins suivants (le résultat de la spécification du système et de l'activité de conception est appelée ici *spécification* pour des raisons de concision):

- un *ensemble de concepts* bien défini;
- des spécifications *non ambigües*, *claires*, *précises* et *concises*;
- une base pour *analyser* la *complétude* et la *correction* des spécifications;
- une base pour déterminer la *conformité* des mises en œuvre par rapport aux spécifications;
- une base pour déterminer la *cohérence* intrinsèque des spécifications;
- l'utilisation d'*outils* informatiques pour créer, maintenir, analyser et simuler les spécifications.

Pour un système, il peut exister des spécifications de différents niveaux d'abstraction. Une spécification est une base pour déduire des *mises en œuvre*, et elle doit s'abstraire de tout détail de mise en œuvre de façon à:

- donner une vue d'ensemble d'un système complexe;
- de différer les décisions relatives à la mise en œuvre; et
- de ne pas exclure de mises en œuvre correctes.

En contraste avec un programme, une spécification formelle (c'est-à-dire une spécification écrite avec un langage de spécification) n'est pas destinée à être exécutée par un ordinateur. En plus de servir de base à la dérivation de mises en œuvre, une spécification formelle peut être utilisée pour assurer une communication précise et non ambigüe entre personnes, en particulier pour la commande et la livraison.

L'utilisation d'un langage de spécification rend possible l'analyse et la simulation de différentes solutions pour un même système; ce qui est, en pratique, impossible pour des raisons de coûts et de délais lorsque l'on utilise un langage de programmation. Un langage de spécification offre un ensemble bien défini de concepts pour l'utilisation du langage améliorant sa capacité à produire une solution à un problème et à raisonner sur la solution.

I.1.5.1 Compréhension d'une spécification formelle

Comme indiqué ci-dessus, un langage de spécification offre un ensemble bien défini de concepts. Ces concepts forment une sorte de modèle mathématique, et par conséquent peut sembler étrange à certaines personnes qui ne sont pas familières avec la théorie mathématique sous-jacente. Par conséquent, une discussion préliminaire de la façon d'appliquer le modèle formel d'un langage de spécification est donnée ci-dessous.

Le champ de l'application d'un système est interprété finalement en terme de concepts d'un langage naturel. Nous acquérons ces concepts à travers un long processus d'apprentissage et d'expérience de la vie réelle. La description d'une application en langage naturel est par nature descriptive, les phénomènes sont décrits tels qu'ils sont perçus par un observateur. La description d'un système en langage naturel fait usage de concepts qui se déduisent directement de l'application et de la mise en œuvre du système.

Lorsqu'un système est spécifié en utilisant un langage de spécification, la spécification formelle ne fait usage ni des concepts de l'application ni de ceux de la mise en œuvre; elle définit plutôt un *modèle* qui représente les propriétés significatives (principalement le comportement) du système. Pour comprendre ce modèle, il faut lui faire correspondre la compréhension intuitive de l'application en terme des concepts du langage naturel (voir Figure I.1-2). Cette mise en correspondance peut être faite de différentes façons, l'une est de choisir les noms des concepts introduits dans la spécification formelle qui présentent une bonne association avec ceux de l'application, une autre est de commenter la spécification formelle. Ceci relève sensiblement de la même problématique que la compréhension d'un programme ou d'un algorithme qui résoud un problème pris dans la vie courante.

Un modèle doit posséder une bonne *puissance d'analyse*, comme indiqué au I.1.5, et une bonne *puissance d'expression* pour faciliter sa mise en correspondance avec l'application. Malheureusement, la puissance d'analyse et la puissance d'expression sont généralement contradictoires. Plus un modèle est expressif et plus il est difficile à analyser. Lorsque l'on conçoit un langage de spécification, il faut évidemment faire un compromis entre ces deux propriétés. De plus il faut insister sur le fait qu'un modèle représente toujours une vue simplifiée de la réalité et qu'il possède, par conséquent, toujours des limitations.

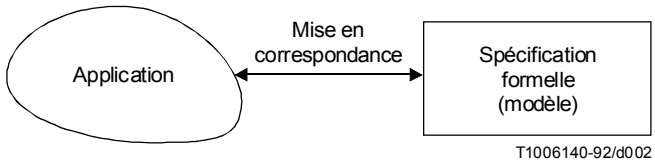


FIGURE I.1-2/Z.100

Comment comprendre une spécification formelle

I.1.5.2 Relations avec une mise en œuvre

Normalement, une distinction est faite entre les langages *déclaratifs* et les langages *constructifs*. SDL est un langage constructif, ce qui signifie qu'une spécification SDL définit un *modèle* qui représente les propriétés significatives d'un système (comme mentionné ci-dessus). Une question importante est de déterminer quelles sont ces propriétés significatives. Ce point est laissé de côté en SDL, c'*est à l'utilisateur de décider quelles sont ces propriétés.*

Si une décision est prise de représenter seulement le comportement du système (tel qu'on le voit à sa frontière), alors on parlera normalement d'une *spécification*, et la structure donnée du modèle n'est qu'une aide pour structurer la spécification en éléments gérables. Si la décision est prise de représenter aussi la structure interne du système, alors on parlera d'une *description*. C'est la raison pour laquelle le SDL ne fait aucune distinction entre spécification et description.

Remarquer que la structure interne d'un système est normalement représentée en SDL par des *blocs*. Les *processus* et la signalisation interne à l'intérieur des blocs n'exigent pas de représenter la structure interne du système, et ainsi n'ont pas besoin d'être considérés comme des exigences d'une mise en œuvre, comme certaines personnes le considèrent à tort. La conformité d'une mise en œuvre à des spécifications est normalement traitée par les organismes de normalisation par une *règle de conformité explicite*. Ceci est évidemment nécessaire aussi lorsqu'on utilise un langage constructif.

I.1.6 Le champ d'application du SDL

Aujourd'hui le SDL est principalement connu dans le secteur des télécommunications, mais il possède un champ d'application plus vaste. Le champ d'application du SDL peut être caractérisé comme suit:

- *type de système:* temps réel, interactif, réparti;
- *type d'information:* comportement et structure;
- *niveau d'abstraction:* du niveau général jusqu'au niveau détaillé.

Le SDL a été développé pour être utilisé dans les systèmes de télécommunication comprenant la communication de données, mais peut actuellement être utilisé pour tous les systèmes temps réel et systèmes interactifs. Il a été conçu pour la spécification de comportement d'un tel système, c'est-à-dire la coopération entre le système et son environnement. Il est aussi adapté à la description de la structure interne d'un système, de sorte que le système peut être développé et être compris une seule partie à la fois. Cette caractéristique est essentielle pour les systèmes répartis.

Le SDL couvre différents niveaux d'abstraction, allant de la vue d'ensemble la plus large possible jusqu'au niveau de la conception détaillée. On n'a pas voulu que ce soit un langage destiné à la mise en œuvre. La traduction plus ou moins automatique de spécifications SDL en un langage de programmation est, cependant, possible.

La Figure I.1-3 montre un échantillon des utilisations possibles du SDL, dans le contexte de l'achat ou de l'acquisition de systèmes de commutation. Dans cette figure, les rectangles représentent des activités caractéristiques, dont les noms précis peuvent varier d'une organisation à une autre. Chaque ligne orientée (ligne de flux) représente un ensemble de documents transmis d'une activité à une autre; les spécifications SDL peuvent être utilisées comme partie de chacun de ces documents. La figure est destinée à être simplement illustrative et ne possède pas de caractère définitif et exhaustif.

Lorsque l'on considère particulièrement les systèmes de commutation, on peut citer quelques exemples de fonctions qui peuvent être documentées en utilisant le SDL, ce sont: le traitement d'appel (par exemple: l'établissement d'appel, l'acheminement, la signalisation, le comptage, etc.), la maintenance, le traitement des fautes (par exemple: les alarmes, la relève automatique de panne, la configuration du système, les tests, etc.), le contrôle du système (par exemple: le contrôle de surcharge) et les interfaces homme-machine.

La spécification de protocoles utilisant le SDL est traitée dans les Recommandations de la série X.

I.1.6.1 A propos de la nature du logiciel de commutation

Le logiciel de commutation peut être vu comme ayant des caractéristiques qui lui sont propres. Le plus souvent le logiciel s'exécute comme un *système spécialisé*, sur une plate-forme matérielle dédiée à une tâche particulière (par exemple un commutateur). Naturellement, le matériel impose des limites sur ce que peut faire le logiciel. Comparé à d'autres nombreux domaines du développement du logiciel, les *besoins fonctionnels* sont bien définis dans le sens qu'ils sont précis, plutôt simples, non ambigüs et très souvent même définis formellement. Très rarement les besoins changent pendant le développement du logiciel. Une grande part du logiciel de commutation peut être considérée comme *réactive* – un message est envoyé au système qui est supposé fournir une réponse. Le logiciel de commutation est habituellement *temps réel* mais on devrait souligner que les besoins en termes de performance sont plus souvent *statistiques* qu'absolus. En règle générale, le logiciel de commutation est *parallèle* et *réparti*.

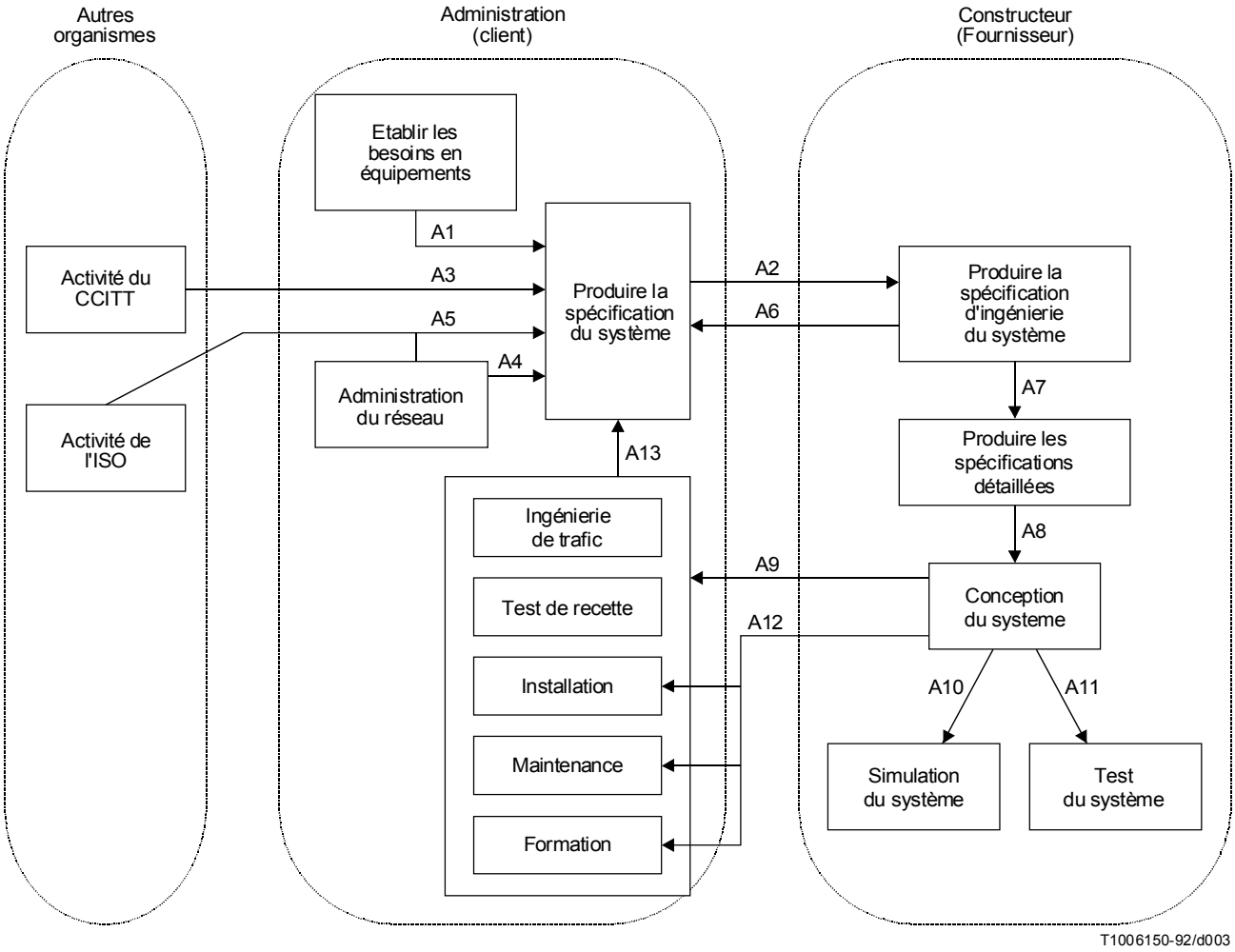
I.1.6.2 Le rôle du SDL dans le logiciel de commutation

De nombreux concepteurs concernés par le développement de logiciel de commutation perçoivent le SDL moins comme un *langage de spécification* (exprimant ce que le système doit faire) que comme un *langage de description* (exprimant comment le système le fait réellement). Pour le concepteur, l'activité de développement du logiciel commence par la réception de la spécification des besoins, éventuellement écrite en SDL, à partir de laquelle le système d'application se déduit d'une manière orientée vers la mise en œuvre.

L'application du SDL comme langage de description pour les systèmes de commutation est une activité organisée qui intègre plusieurs expertises exigeant des compétences, des aspects multi-facettes de l'état de l'art en génie logiciel. Le SDL fournit un cadre conceptuel suffisant pour supporter différents niveaux d'abstraction allant de la spécification des besoins jusqu'à la mise en œuvre.

Le problème du test d'une spécification SDL n'est pas trivial. D'abord, les logiciels des systèmes de commutation sont très volumineux et le développeur, la plupart du temps, n'a qu'une compréhension globale de l'environnement de son propre code. Il n'est pas inimaginable que les modules destinés à communiquer entre eux soient développés en même temps. Par conséquent, il faut construire un *environnement de test* pour chaque composant. Le *test d'intégration* et le *test de système* ne sont possibles qu'ensuite. Deuxièmement, les systèmes spécialisés ne fournissent pas toujours les outils de développement de logiciel de la qualité des systèmes informatiques ordinaires et le développeur doit se satisfaire du contrôle des échanges de messages et d'une simple mise au point. Souvent, une autre possibilité consiste à simuler le comportement de la spécification SDL réelle sur une plate-forme matérielle plus développée de façon à s'assurer de la correction fonctionnelle de la conception. Une autre possibilité consiste à engendrer les cas de test ou les entrées des tests directement à partir de la spécification SDL. Troisièmement, il est conseillé de contrôler la *conformité* de la conception SDL par rapport à la spécification des besoins. Cependant, ceci se fait la plupart du temps de manière non automatique, informelle.

Après la livraison, la conception du système devra être *maintenue* ce qui est une des raisons pour produire une spécification SDL lisible et bien documentée. Chaque action de maintenance devrait relancer un nouveau cycle de test, de préférence avec le même jeu de données de test que précédemment. Plusieurs parties des systèmes de commutation sont *personnalisée*s ce qui nécessite d'avoir un système adéquat de *contrôle de versions*.



A1 Spécification d'un équipement ou d'une fonctionnalité indépendant de la mise en oeuvre et du réseau
A2 Spécification de système indépendante de la mise en oeuvre mais dépendante du réseau, comprenant la description de l'environnement du système
A3 Recommandations et directives du CCITT
A4 Contributions à la spécification du système, montrant l'administration du réseau et les besoins opérationnels
A5 Autres normes pertinentes
A6 Description d'une proposition de mise en oeuvre
A7 Une spécification de projet
A8 Une spécification de conception détaillée
A9 Une description du système complet
A10 Description du système et de l'environnement convenant à la simulation du système
A11 Description du système et de l'environnement convenant au test du système
A12 Manuels d'installation et d'exploitation
A13 Contributions à la spécification de système à partir d'activités spécialisées à l'intérieur d'une Administration

NOTES

1 L'itération est possible à tous les niveaux.

2 Dans certaines circonstances, les spécifications SDL qui sont présentées ici comme étant internes à une organisation, par exemple A1, A7 et A8, peuvent être fournies à une autre organisation.

FIGURE I.1-3/Z.100

Scénario général pour l'utilisation du SDL

I.2 Modélisation des applications

I.2.1 Protocoles et services OSI

I.2.1.1 Introduction

Les concepts OSI utilisés au I.2.1 sont définis dans [1] et [2]. Ceux-ci sont expliqués ici, pour rendre le paragraphe complet par lui même.

Un *service de couche* est offert par un *fournisseur de service* pour une couche donnée. Le fournisseur de service est une machine abstraite offrant un moyen de communication aux *utilisateurs* de la couche immédiatement au-dessus. Le service est accédé par les utilisateurs aux *points d'accès au service* au moyen de *primitives de service*, comme le montre la Figure I.2-1. Une primitive de service peut être utilisée pour la gestion de la connexion (connexion, déconnexion, réinitialisation, etc.) ou être une donnée (donnée normale ou donnée expresse). Il existe seulement quatre types de primitives de service:

- demande (de l'utilisateur au fournisseur);
- indication (du fournisseur à l'utilisateur);
- réponse (de l'utilisateur au fournisseur);
- confirmation (du fournisseur à l'utilisateur).

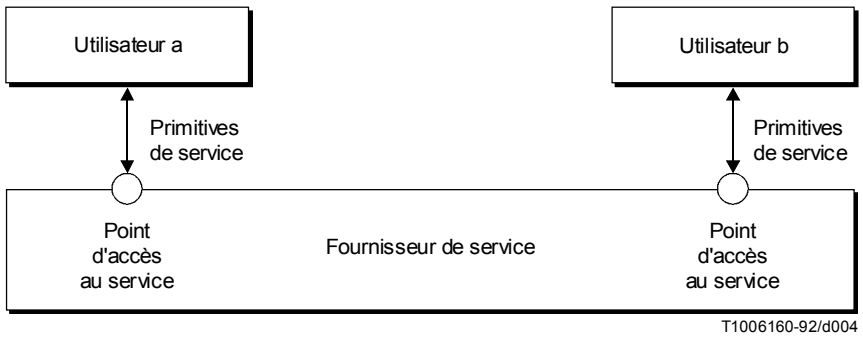


FIGURE I.2-1/Z.100

Un fournisseur de service en couche

Une *spécification de service* est un moyen de caractériser le comportement du fournisseur de service, à la fois localement: en fixant les séquences correctes de primitives de service transmises à un point d'accès au service, et de bout en bout: en fixant la relation correcte entre les primitives de service transmises à différents points d'accès du service. Une spécification de service ne concerne pas la structure interne du fournisseur de service; n'importe quelle structure interne fournie lors de la spécification du service n'est qu'un modèle abstrait pour décrire le comportement observable à l'extérieur du fournisseur du service.

Excepté pour la couche la plus haute, les utilisateurs d'un service en couche sont des *entités de protocoles* pour la couche immédiatement au-dessus, qui coopèrent pour augmenter les fonctionnalités du service de la couche, en fournissant ainsi un service à la couche au-dessus. La coopération est conduite en respectant un ensemble prédéfini de règles de comportement et de formats de messages qui constituent un *protocole*. Conformément à cette optique, les entités de protocole de la couche (N) et le fournisseur de service (N – 1) fournissent ensemble un raffinage pour le fournisseur du service (N) (voir la Figure I.2-2).

Le raffinage du fournisseur de service (N) présenté à la Figure I.2-2 peut naturellement être beaucoup plus compliqué. Par exemple, il peut y avoir des entités de protocole de relai (N) qui ne sont connectés à aucune entité de protocole de la couche (N + 1). Par souci de concision, de tels cas ne sont pas considérés ici.

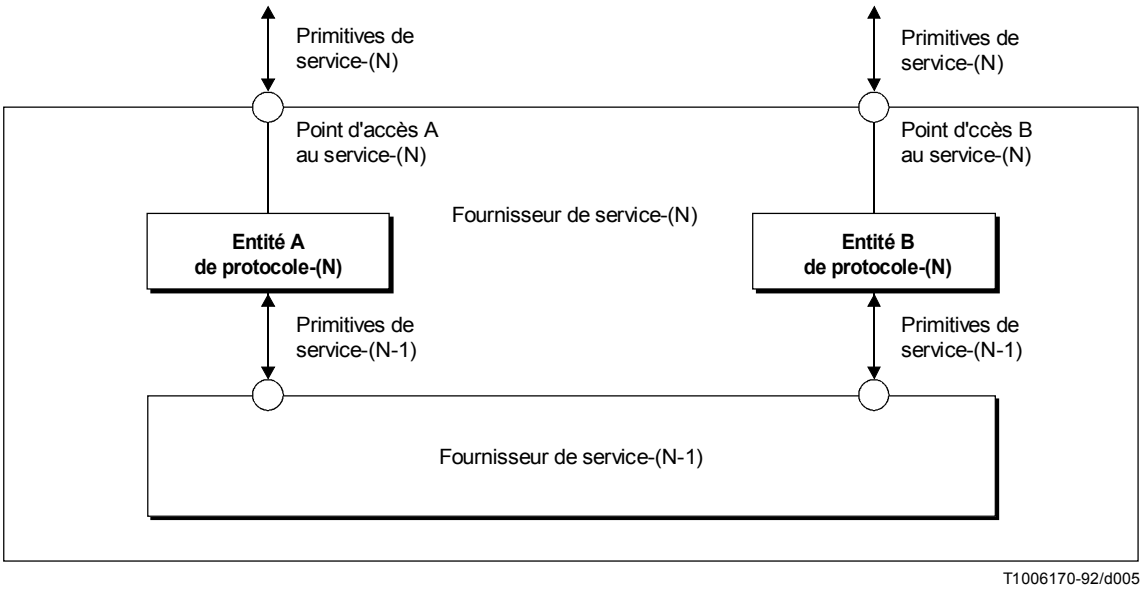


FIGURE I.2-2/Z.100

Raffinage du fournisseur de service de la couche (N)

Les entités de protocole communiquent par échanges d'*éléments d'information de protocoles*. Ceux-ci sont transférés en tant que paramètres des primitives de service de la couche sous-jacente. L'entité du protocole émettrice code les éléments d'information de protocole dans les primitives de service, l'entité du protocole réceptrice décode les éléments d'information de protocole à partir des primitives de service reçues. Un protocole se fonde sur les propriétés du fournisseur de service sous-jacent. Le fournisseur de service sous-jacent peut par exemple perdre, altérer des messages ou en modifier l'ordre, en pareil cas le protocole doit disposer de mécanismes de détection et de correction d'erreur, de resynchronisation, de retransmission, etc. de façon à fournir un service fiable et généralement plus puissant à la couche immédiatement au-dessus.

Les concepts architecturaux OSI peuvent être modélisés en SDL de différentes façons, dépendant principalement des aspects que l'on souhaite mettre en valeur. D'abord, une approche simple est décrite, ensuite d'autres approches sont esquissées comme variante de cette approche de base.

Dans les exemples, la syntaxe graphique du SDL est utilisée autant que faire se peut. Noter, cependant, que pour une raison pratique certaines informations imposées par les règles syntaxiques peuvent être omises, ou représentées par une suite de points (...) qui bien sûr ne font pas partie de la syntaxe.

I.2.1.2 Approche de base

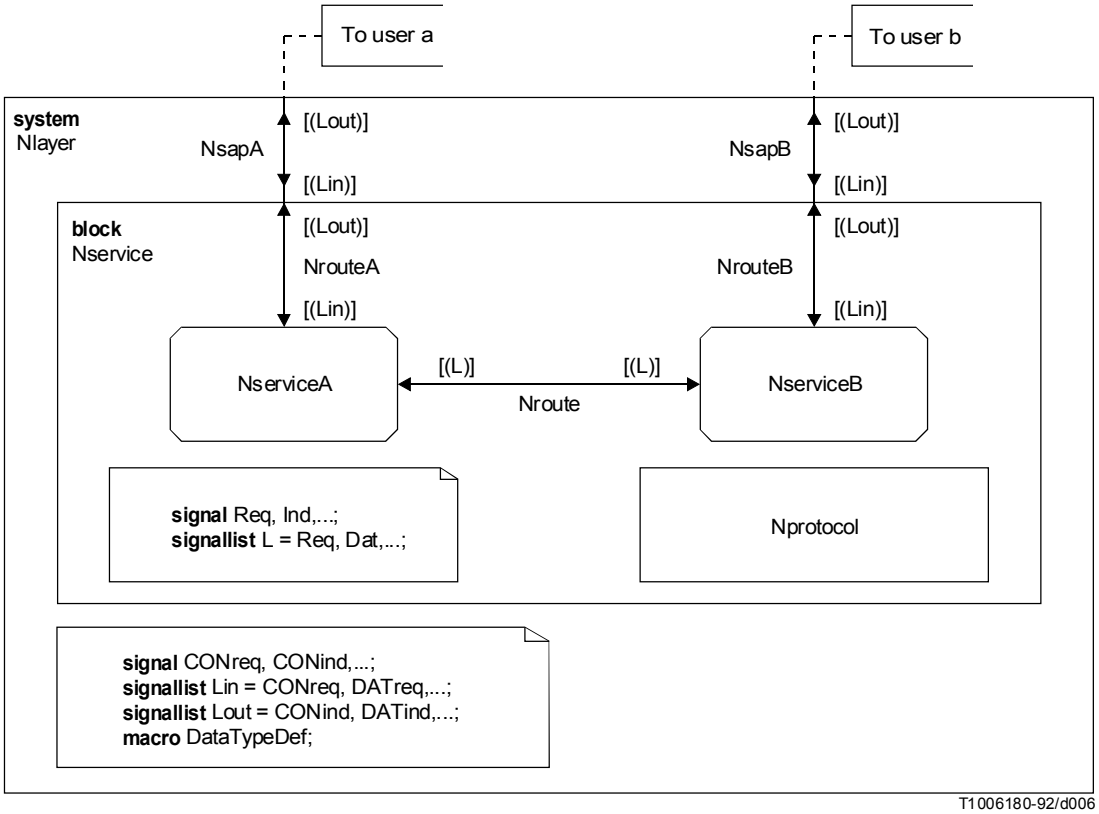
Spécification de service

Une spécification de service pour la couche N peut être modélisée de manière directe comme un bloc *Nservice* contenant deux processus *NserviceA* et *NserviceB,* comme le montre l'exemple I.2-1.

Dans l'exemple I.2-1, les utilisateurs de ce service sont dans l'environnement du système et peuvent être considérés comme des processus, capables de communiquer avec le système selon les termes de ce système.

Un point d'accès au service est représenté par un canal (*NsapA*, ou *NsapB*) transportant des signaux, qui représentent les primitives du service. Un signal peut transporter des valeurs des sortes données dans la spécification du signal. Les spécifications de sorte (autres que celles des sortes prédéfinies) sont contenues dans la spécification distante de *macro DataTypeDef*, qui est omise ici car non pertinente pour l'objet du présent exposé.

Naturellement, dans le cas le plus général, il peut y avoir plus de deux processus concernés par la spécification du service. Nous allons considérer ici seulement deux processus, un pour chaque point d'accès au service, par souci de concision. L'exposé suivant s'applique aussi, cependant, au cas où il y a plusieurs processus pour un point d'accès au service.



EXEMPLE I.2-1

Spécification d'un service de couche (N) en SDL

Dans l'exemple I.2-1, on montre quelques exemples de spécifications de signaux. Comme quelques noms de signaux le suggèrent, on suppose qu'il s'agit d'un service orienté connexion. Dans le cas d'un service sans connexion, un grand nombre de simplifications peuvent être faites. Cependant, ce cas ne sera pas traité plus tard, pour des raisons de concision.

On considère ici, à la fois les aspects locaux et «de bout en bout» d'une spécification de service. Le comportement local s'exprime de façon indépendante par les processus *NserviceA* et *NserviceB*. Ces processus communiquent entre eux par des signaux (*Req*, *Ind*, ...) qui sont internes au bloc et transportés par l'acheminement de signaux *Nroute*. Le comportement de bout en bout s'exprime par la mise en correspondance (effectuée par chaque processus) entre les primitives de service et les signaux internes sur *Nroute*. Les processus *NserviceA* et *NserviceB* sont des images miroir l'un de l'autre. Il y en a deux, au lieu d'un seul, pour permettre de modéliser fidèlement une situation de collision possible dans le fournisseur de service.

Le comportement non déterministe est une caractéristique inhérente au fournisseur de service, parce qu'il peut refuser les tentatives de connexion et qu'il peut interrompre les connexions établies de sa propre initiative.

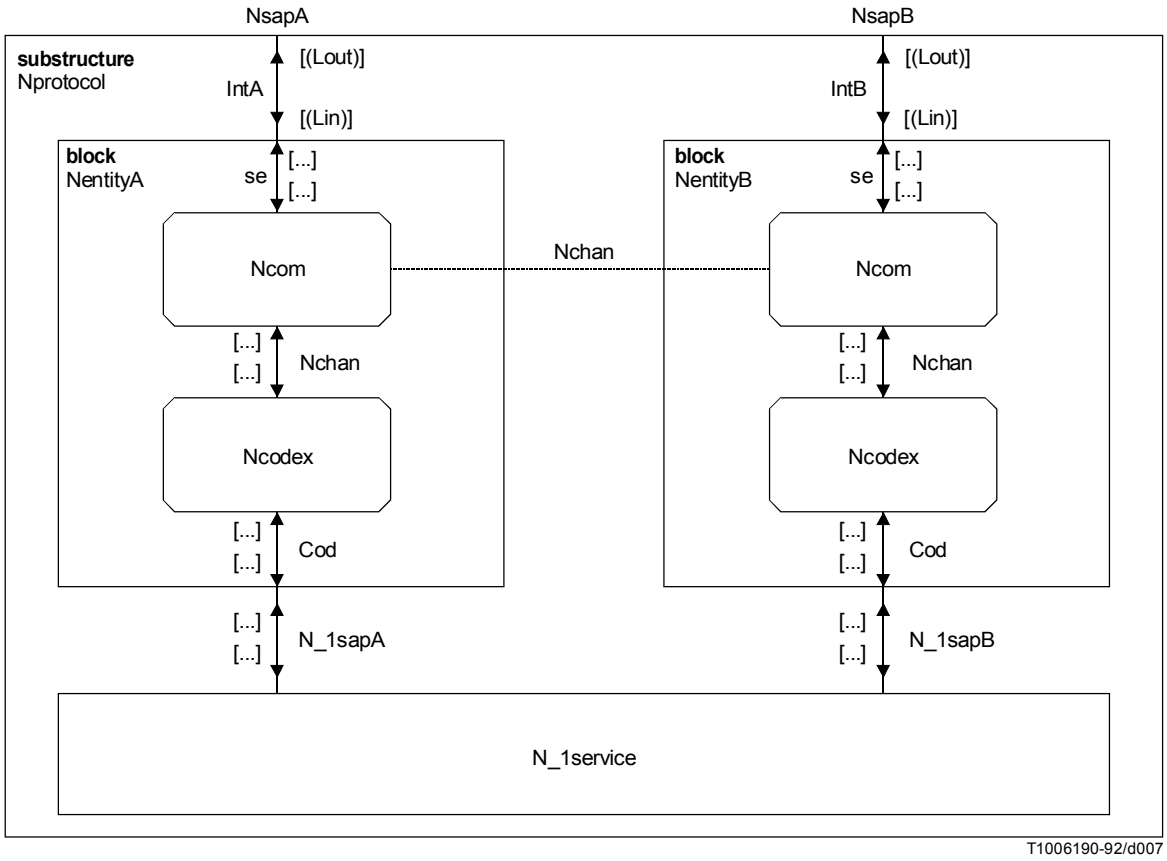
Il faut noter que la spécification du bloc *Nservice* contient seulement une référence aux processus *NserviceA* et *NserviceB*; ces processus sont spécifiés par des *spécifications distantes*, placées en dehors de la spécification du bloc, et ils ne sont pas présentés ici, parce qu'ils pourraient attirer l'attention du lecteur sur des caractéristiques nécessairement spécifiques d'un service donné.

Spécification de protocole

La spécification de protocole pour la couche N est modélisée par la sous-structure *Nprotocol* du bloc *Nservice*, comme le montre l'exemple I.2-1.

Dans le *diagramme de bloc* de l'exemple I.2-1, une référence de sous-structure de bloc (un symbole de bloc contenant le nom *Nprotocol* de la sous-structure de bloc) a été introduite. La spécification de la sous-structure de bloc est donnée dans un *diagramme de sous-structure de bloc* distant (voir l'exemple I.2-2) contenant 3 blocs *NentityA*, *NentityB* et *N_1service*. Les deux premiers blocs représentent les entités de protocole (N), tandis que le bloc *N_1service* représente le fournisseur de service (N – 1). La spécification de *N_1service* est analogue à la spécification de *Nservice* et ne figure pas dans le diagramme (il s'agit d'une spécification distante).

Un bloc entité de protocole contient un ou plusieurs processus, selon les caractéristiques du protocole. Dans le cas présent, deux processus ont été choisis, *Ncom* et *Ncodex*. Le processus *Ncom* gère l'envoi et la réception d'éléments d'information de protocole; tandis que le processus *Ncodex* s'assure de la transmission des éléments d'information du protocole en utilisant le service sous-jacent. Conceptuellement, les processus *Ncom* communiquent directement via un canal implicite *NChan* (transportant les éléments d'information du protocole), mais en réalité ils communiquent indirectement via le processus *Ncodex* et le fournisseur du service sous-jacent.



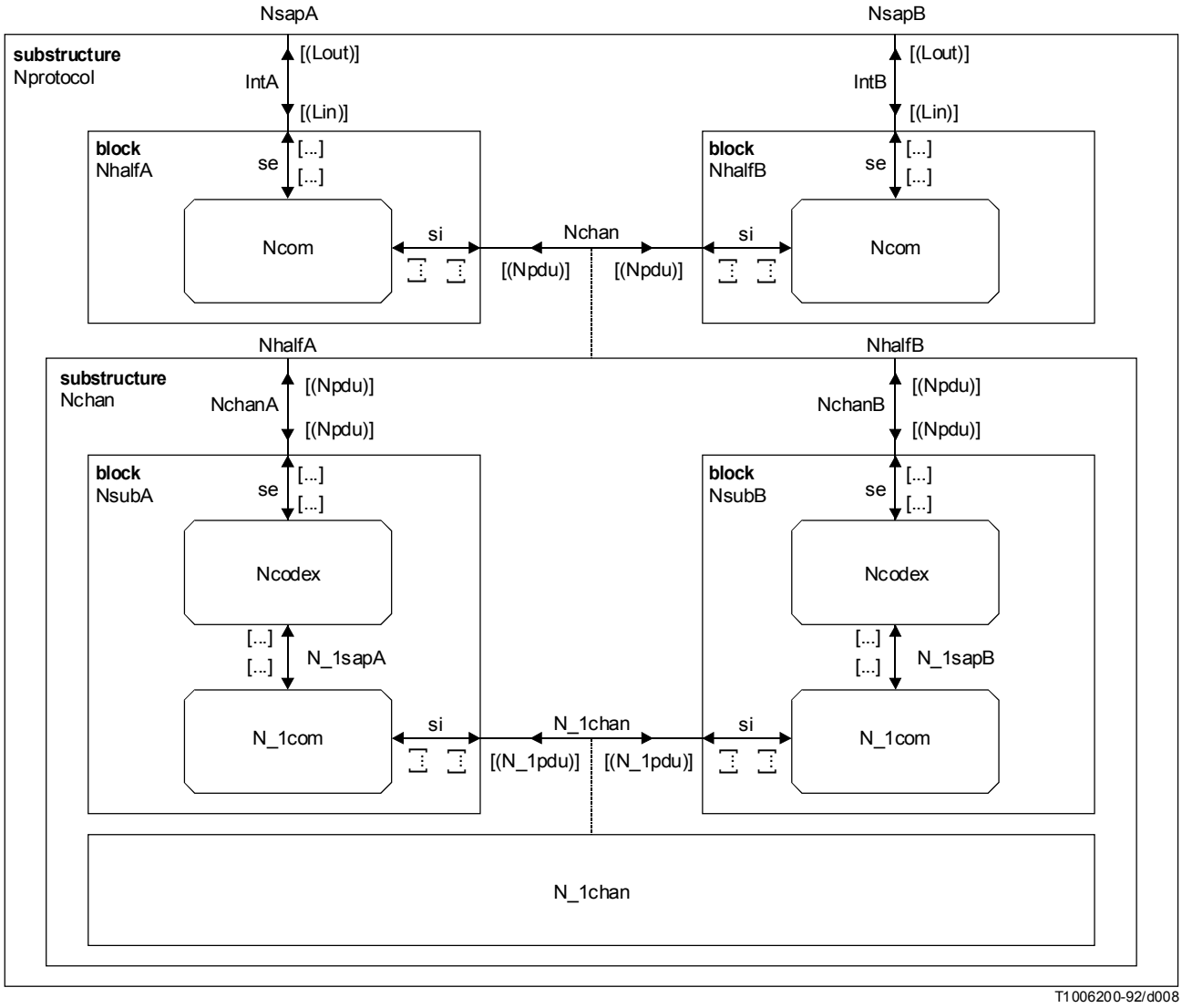
EXEMPLE I.2-2

Spécification d'un protocole de couche (N) en SDL

I.2.1.3 Autre approche utilisant la sous-structure de canal

Cette approche s'obtient à partir de l'approche de base de l'exemple I.2-2 en groupant les processus différemment, en introduisant le canal réel *Nchan* et en utilisant une sous-structure de canal, comme le montre l'exemple I.2-3. Le canal *Nchan* transporte les éléments d'information du protocole, figurant dans la liste des signaux *Npdu*. Cette approche fait ressortir la vue du protocole et l'orientation en couche de l'OSI.

Noter que dans cette approche les blocs à l'intérieur de la sous-structure de canal ne représentent pas les entités de protocole et se superposent aux deux couches adjacentes. Les primitives de service sont cachées dans ces blocs et sont transportées dans les acheminements de signaux *N_1sapA, N_1sapB, N_2sapA, N_2sapB*, etc. Cependant, la plus haute couche choisie (N) doit être traitée séparément, comme indiqué dans l'exemple. Noter aussi que le diagramme de système (voir l'exemple I.2-1) n'est pas affecté par cette approche.

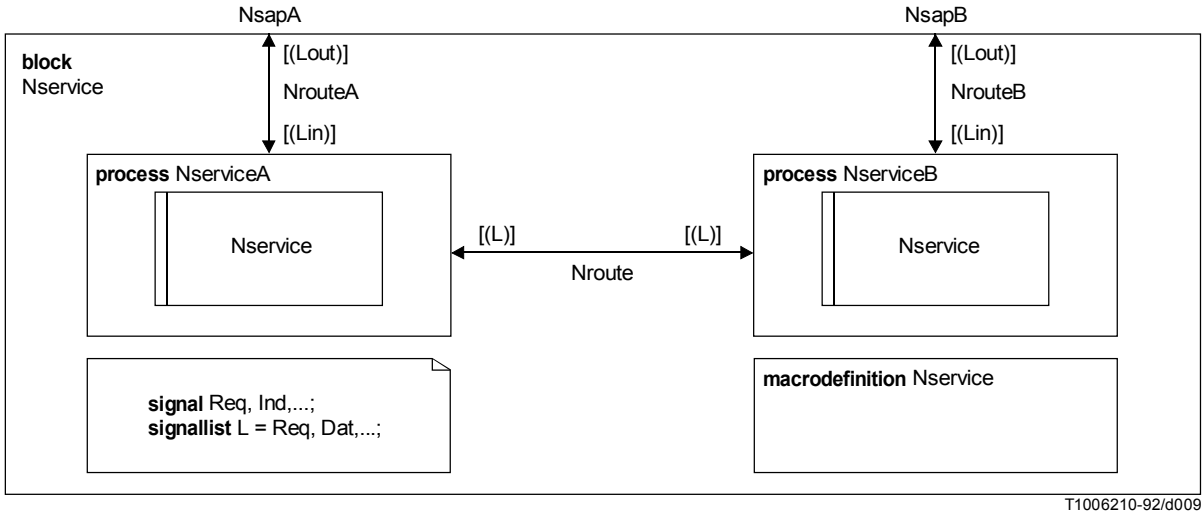


EXEMPLE I.2-3

La vue de protocole de l'OSI

I.2.1.4 Architecture OSI symétrique

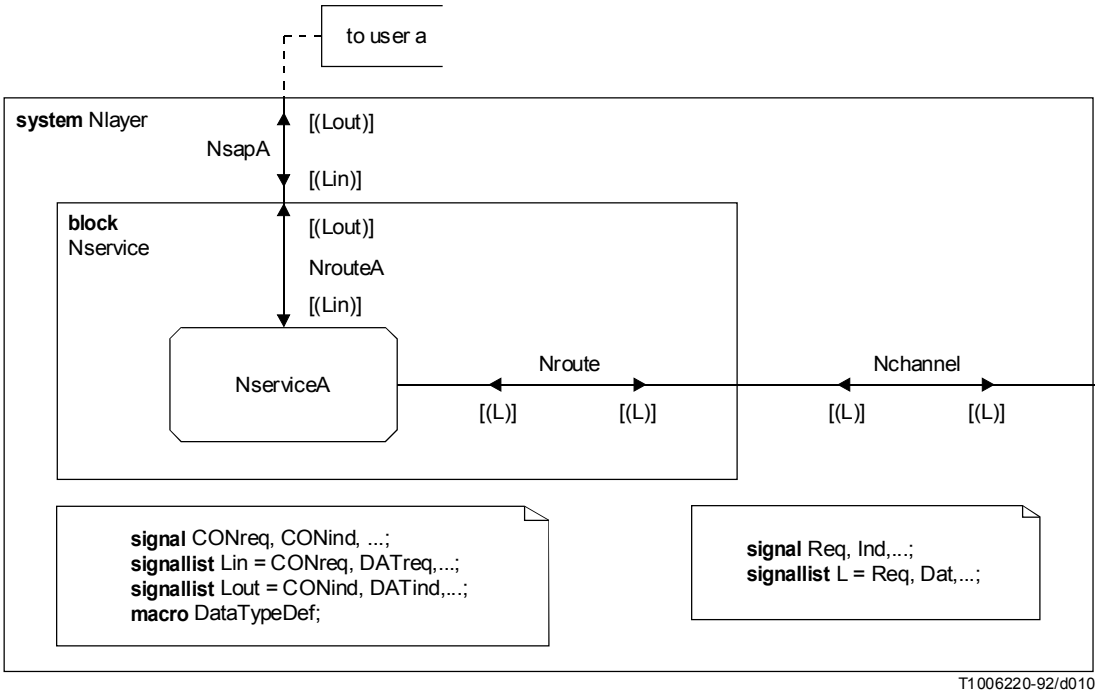
Lorsque l'architecture OSI est symétrique, ce qui est le cas lorsque les entités des deux cotés de l'architecture OSI sont les images miroir l'une de l'autre, les spécifications de ces entités sont identiques, sauf le nom de l'entité. La spécification commune peut être établie une seule fois en utilisant une macro (voir l'exemple I.2-4). Dans cet exemple seul figure le diagramme de bloc de *Nservice* de l'exemple I.2-1. Noter que les spécifications de service sont toujours identiques, seules les spécifications de protocole peuvent être asymétriques.



EXEMPLE I.2-4

Utilisation d'une macro pour représenter une architecture OSI symétrique

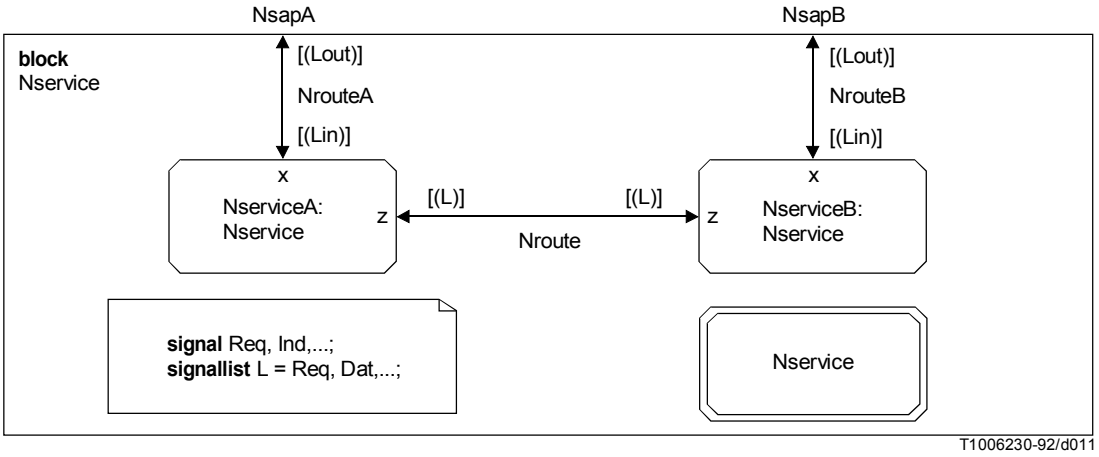
Une autre approche consiste à ne représenter qu'un coté, comme le montre l'exemple I.2-5, qui est une modification du diagramme du système de l'exemple I.2-1. Le canal *NsapB* a été remplacé par le canal *Nchannel*, transportant les signaux internes *L*. Ces signaux sont maintenant au niveau du système, et leurs spécifications ont été déplacées en conséquence. Noter que cette approche ne peut pas être utilisée conjointement avec la sous-structure de canal (montrée à l'exemple I.2-3).



EXEMPLE I.2-5

Représentation d'un seul côté d'une architecture OSI symétrique

Une troisième approche peut être donnée par instanciation d'un type. L'exemple I.2-4 est modifié en introduisant un type de processus *Nservice* dans le bloc *Nservice*, qui est alors instancié en *NserviceA* et *NserviceB,* comme le montre l'exemple I.2-6. x et z sont les paramètres réels correspondant aux portes (non montrées ici) du type de processus *Nservice*.



EXEMPLE I.2-6

Utilisation d'un type de processus pour représenter une architecture OSI symétrique

I.2.2 RNIS conforme à I.130 et Q.65

Les services RNIS sont décrits en utilisant une méthodologie conforme aux Recommandations I.130 [3]. Cette méthodologie comprend 3 phases :

- Phase 1: Les aspects liés aux services
- Phase 2: Les aspects fonctionnels du réseau
- Phase 3: Les aspects physiques du réseau

La phase 2 est décrite plus en détail dans la Recommandation Q.65 [4] et comprend, pour chaque service, les étapes suivantes:

- Etape 1: Modèle fonctionnel – identification des entités fonctionnelles et de leurs relations;
- Etape 2: Diagrammes de flux d'information;
- Etape 3: Diagrammes SDL pour chaque entité fonctionnelle;
- Etape 4: Actions des entités fonctionnelles;
- Etape 5: Allocation des entités fonctionnelles à des emplacements physiques.

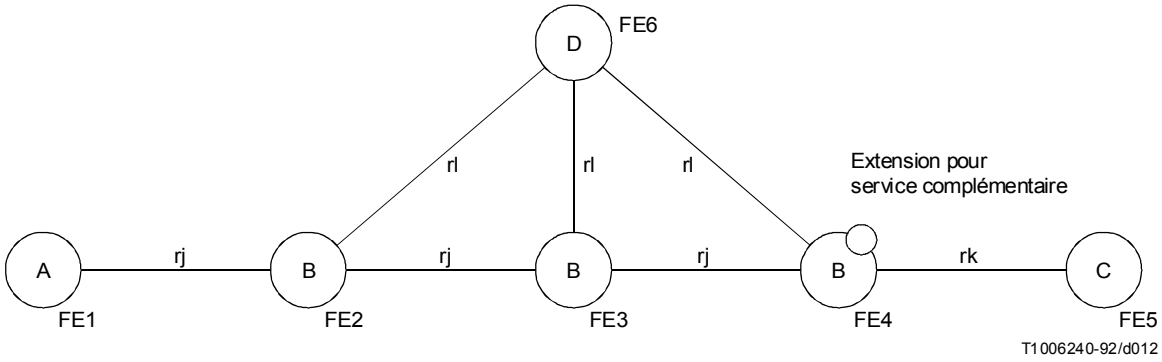
Jusqu'en 1988, le SDL n'a été utilisé que pour l'étape 3. Cette utilisation du SDL était basée sur le SDL 80. L'intention est d'utiliser le SDL 88 pour les nouvelles Recommandations.

Idéalement, il devrait être possible d'utiliser les concepts structurels de SDL à l'étape 1 et les concepts comportementaux dans les étapes 2 à 4 avec une introduction progressive des données. L'étape 5 reste en dehors de la portée du SDL. L'argument général en faveur de l'utilisation du SDL dans les étapes 1 à 4 de Q.65 est principalement que l'adhésion à des standards accroît la lisibilité et rend possible le support par des outils.

On montrera ci-dessous comment les concepts issus de la plupart des étapes de Q.65 peuvent être représentés en SDL.

I.2.2.1 Structure

Etape 1: modèle fonctionnel, identification d'entités fonctionnelles et leurs relations correspondant à l'usage des concepts structurels du SDL. La Figure I.2-3 montre un exemple de modèle fonctionnel conforme à Q.65.



NOTE – Les éléments de la figure sont:
– les noms à l'intérieur des cercles (ex. *A*): Types d'entités fonctionnelles;
– les noms en majuscules adjacents aux cercles (ex. *FE1*): Noms des entités fonctionnelles (instances);
– les noms en minuscules entre les cercles (ex. *rj*): Relations entre les types d'entités fonctionnelles;
– le petit cercle ajouté: Extensions pour service complémentaire.

FIGURE I.2-3/Z.100

Exemple d'un modèle fonctionnel conforme à la Recommandation Q.65

Noter que ce modèle établit une nette distinction entre les types des entités fonctionnelles (ex. *A*) et leurs instanciations (ex. *FE1*) lorsqu'est décrite la structure du système. Q.65 mentionne également qu'il est commode de décrire les deux types d'entités fonctionnelles comme des sous-ensembles de la même entité fonctionnelle, si les deux types d'entités fonctionnelles ont beaucoup de points communs. Ces caractéristiques peuvent être exprimées en utilisant les extensions «orientées objet» de SDL 92, mais ne peuvent pas l'être en SDL 88.

L'analyse a montré que les «entités fonctionnelles» dans Q.65 peuvent être représentées par les concepts structurels du SDL de la façon suivante:

- Modèle → système;
- Entité fonctionnelle → bloc avec un processus;
- Relations entre entités fonctionnelles → canal;

De plus, en utilisant les constructions «orientées objet»:

- Type d'entité fonctionnelle → type de bloc.

Une différence entre le modèle fonctionnel conforme à Q.65 et les diagrammes de structure du SDL est que les diagrammes de structure du SDL modélisent normalement des systèmes ouverts.

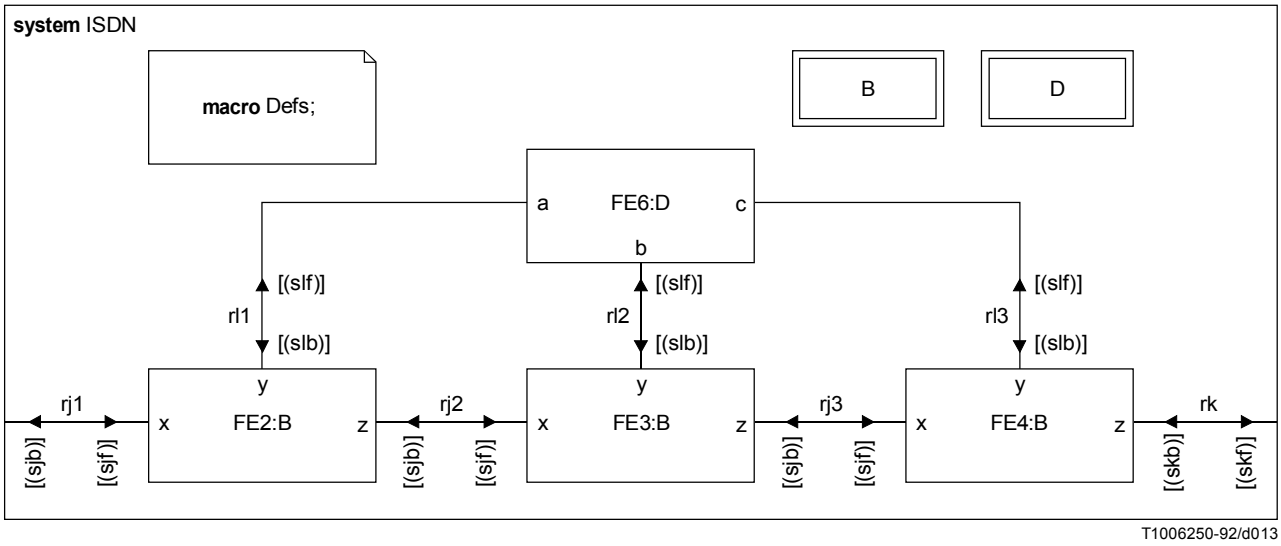
L'exemple I.2-7 présente le modèle fonctionnel de la Figure I.2-3 exprimé en SDL. Remarquer que *FE1* et *FE5* sont absents de l'intérieur du système. La communication avec ces entités fonctionnelles est modélisée comme une communication avec l'environnement. En laissant *FE1* et *FE5* dans l'environnement on anticipe le fait que l'on ne souhaite pas décrire le comportement de *FE1* et *FE5*.

La distinction entre relation et type de relation peut être modélisée par l'usage de définitions de listes de signaux dans les diagrammes de structure du SDL. Conformément à SDL 92, *B* et *D* sont des types de bloc.

Les avantages d'utiliser des diagrammes de structure du SDL résident principalement dans le fait que les diagrammes de structure du SDL contiennent les définitions de signaux, de types de données, etc. qui sont importants pour les étapes suivantes de Q.65. Dans l'exemple I.2-7, ces définitions sont indiquées par la macro *Defs*.

I.2.2.2 Comportement

Le comportement est décrit dans les étapes 2 à 4 de Q.65. L'interaction entre entités fonctionnelles est décrite au moyen de diagrammes de flux d'information. A partir de ces derniers, un diagramme SDL (par exemple, un diagramme de processus) est produit pour chaque type d'entité fonctionnelle.



EXAMPLE I.2-7

Version SDL de l'exemple de la Figure I.2-3

Les diagrammes de flux d'information correspondent aux diagrammes de séquences de messages (MSC) (*message sequence charts*) qui ont été normalisés par le sous-groupe de travail X/3 dans la Recommandation Z.120 [5]. Les diagrammes de flux d'information sont produits pour les cas «normaux», et le SDL est utilisé pour décrire le comportement des entités fonctionnelles.

L'étape 4 de Q.65 concerne la définition d'un certain nombre d'actions de base pour chaque entité fonctionnelle. L'idée est alors de réutiliser ces actions de base dans différentes spécifications de service. Cette idée est encore à l'étude. Pour cette approche, le SDL peut offrir le concept de macro ou de procédure: chaque action de base peut être «encapsulée» dans une macro ou dans une procédure. L'étape 4 ne sera pas décrite plus à fond dans la suite.

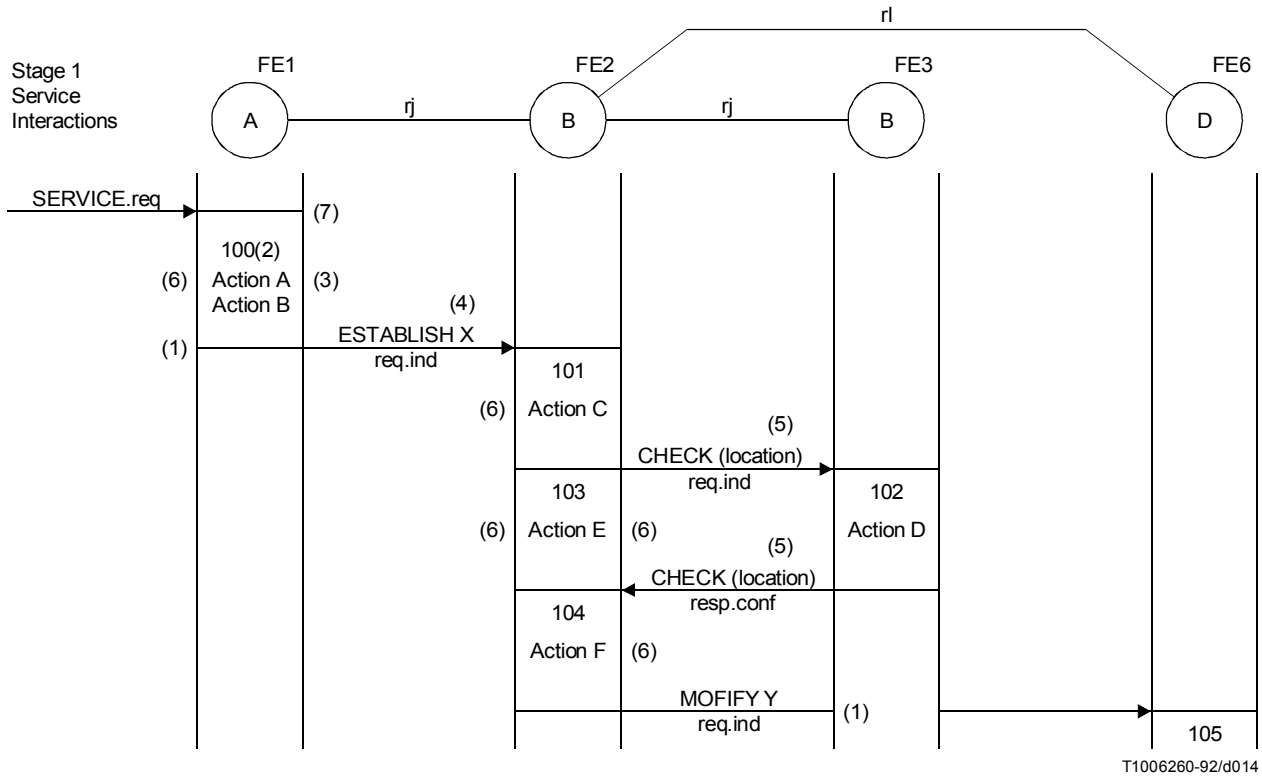
Diagrammes de flux d'information

Un exemple de diagramme de flux d'information est présenté à la Figure I.2-4.

L'exemple I.2-8 montre le MSC qui correspond au diagramme de flux d'information de la Figure I.2-4. La correspondance entre les deux est évidemment:

- nom d'entité fonctionnelle → instance de processus (une instance de processus par bloc);
- flux d'information → message;
- informations additionnelles → paramètres de message;
- action de base de l'entité fonctionnelle → action;
- étiquettes des diagrammes SDL → commentaires.

Les diagrammes de flux d'information recommandés dans Q.65 ainsi contiennent fondamentalement les mêmes informations que les MSC. Les spécifications SDL peuvent être généralement vérifiées par rapport aux MSC pendant une simulation, et dans quelques cas simples (pas de création dynamique de processus, pas d'adressage dynamique des sorties). Ce contrôle peut être réalisé facilement en dehors de toute simulation. Les MSC peuvent être dérivés des spécifications SDL, mais normalement un sous-ensemble seulement du comportement («le comportement normal») est exprimé dans un MSC, de sorte qu'il faut une intervention humaine pour déduire les MSC adéquats à partir d'une spécification SDL. Une autre possibilité est de déduire automatiquement des squelettes de diagrammes de processus SDL à partir des diagrammes de flux d'information. Voir aussi l'**étape 5** en I.3.2.



NOTE – Les éléments de ce diagramme sont:
– les noms au sommet des colonnes (ex. *FE1*): Entités fonctionnelles;
– les noms en minuscules entre les cercles (ex. *rj*): Relations entre types d'entités fonctionnelles;
– les noms sur les flèches entre les colonnes (ex. *ESTABLISH X req.ind*): Flux d'information;
– les noms placés entre parenthèses et adjacents aux noms de flux d'information (ex. *location*): Information additionnelle transmise par le flux d'information;
– les noms dans les colonnes (ex. *Action B, 100*): Noms des actions de base des entités fonctionnelles;
– les nombres placés entre parenthèses (ex. *(5)*): Etiquettes utilisées dans les diagrammes SDL;

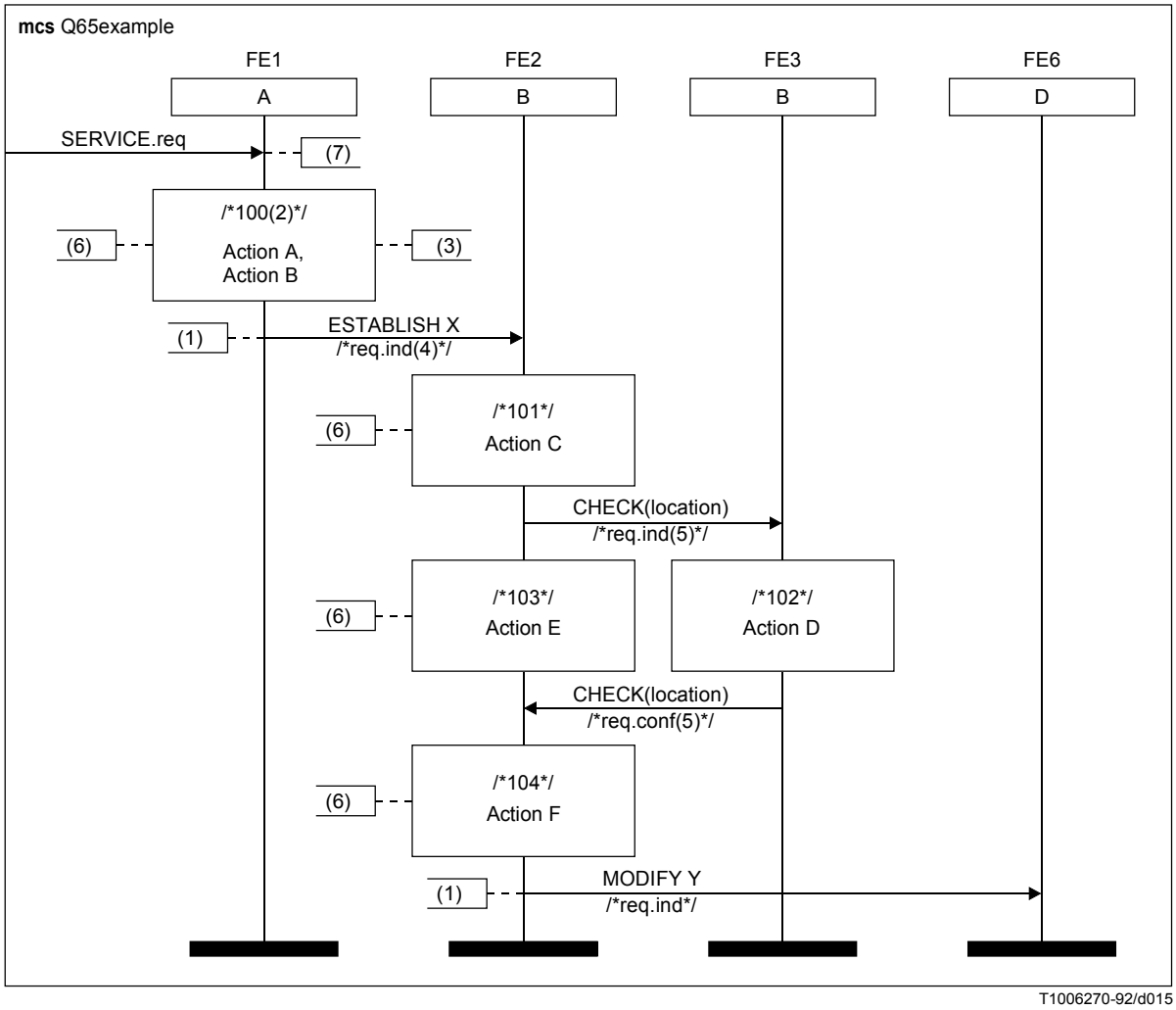
FIGURE I.2-4/Z.100

Exemple de diagramme de flux d'information extrait de la Recommandation Q.65

Diagrammes de processus

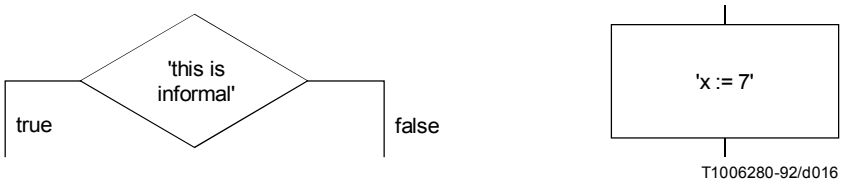
C'est la partie du SDL qui a été réellement utilisée dans les Recommandations du RNIS. La plupart du travail a maintenant été investi dans la mise des diagrammes SDL en conformité au SDL88. Quelques observations ont été tirées de cette mise à jour: tous les textes situés dans les symboles étaient informels à partir du début. Il était plus facile de convaincre les gens du besoin de formalisme à l'intérieur des symboles d'**entrée** (**input**) et de **sortie** (**ouput**), car ces symboles se rapportent aux interfaces des processus (c'est-à-dire à la précédente étape dans la méthodologie préétablie. Il était (et il est) ennuyeux pour beaucoup de personnes d'avoir à placer des apostrophes (") autour du texte informel dans les **tâches** (**task**) et les **décisions** (**decision**), car le contenu de ces boîtes est la plupart du temps du texte informel. Quelques exemples sont présentés à l'exemple I.2-9. De nombreuses suggestions créatives ont été faites de manière informelle pour définir une règle plus conviviale pour l'utilisateur et peut-être moins pour l'outil pour faire la distinction entre texte informel et texte formel. Cependant, il semble difficile de formuler des règles simples et non ambigües pour les textes informels sans invalider l'usage courant des chaînes de caractères *Charstring* comme texte informel.

La tradition d'affecter des suites descriptives de noms à un état ou à un signal a créé un autre désaccord. La première solution suggérée par la Commission X était d'utiliser des blanc-soulignés pour lier un certain nombre de mots en un seul, par exemple «hang up req.ind» devient «hang_up_req.ind». Plus tard les règles lexicales du SDL88 ont été reformulées rendant «hang up req.ind» un nom acceptable en SDL. L'usage des noms en SDL88 correspond maintenant à l'usage établi.



EXEMPLE I.2-8

Version MSC de l'exemple de la Figure I.2-4

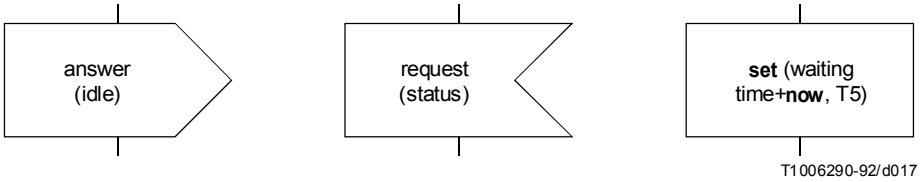


EXEMPLE I.2-9

Texte informel

I.2.2.3 Données

Les **tâches** (**task**) et les **décisions** (**decision**) sont principalement informelles dans les diagrammes existants pour le RNIS. Par conséquent, il y a un léger besoin de définir des opérateurs pour les types de données. Les données apparaissent principalement dans les constructions **entrée** (**input**), **sortie** (**output**) et **temporisateur** (**timer**), comme le montre l'exemple I.2-10.



EXEMPLE I.2-10

Utilisation des données

Dans la plupart des cas, les types «booléens» *Booléan* et «entier» *Integer* sont suffisants, par exemple pour représenter la valeur de quelque drapeau ou compteur. Le type «chaînes de caractères» *Charstring* est aussi largement utilisé, parce qu'il permet de manipuler les paramètres de façon similaire à celle du texte informel.

L'utilisation des types énumérés devrait être utile. Un **newtype** ne comprenant que des littéraux correspond à un type de données énuméré (comme par exemple en Pascal):

```
newtype Indication
literals idle, busy, congestion;
endnewtype Indication;
...
signal Check location (Indication) /*resp.conf */;
...
output Check location (busy) /*resp.conf*/;
...
```

Les données sont introduites dans les étapes 1 à 4 en formalisant les spécifications, par exemple les spécifications de signal s'appuient sur les types de données. Les données peuvent être introduites dans plusieurs niveaux de raffinage. Le bénéfice de l'introduction des données est naturellement qu'il permet de vérifier que les paramètres des entrées et des sorties, les questions et les réponses etc. sont cohérents dans l'ensemble de la spécification SDL.

I.3 Production progressive d'une spécification SDL

Ce paragraphe présente une méthode pour produire une spécification SDL. Le texte n'implique pas une recommandation d'une méthode particulière par le CCITT, puisqu'il est reconnu qu'une spécification SDL peut être obtenue par de nombreuses méthodes différentes. La méthode peut être considérée comme des directives pour développer une spécification SDL à travers des étapes bien définies, bien que le résultat de chaque étape peut ne pas être une spécification complète et formelle. Des adaptations de la méthode peuvent très bien inclure des réarrangements de quelques étapes, etc.

Les étapes sont identifiées du point de vue de l'utilisateur, et formellement en exposant les constructions introduites. La méthode est illustrée par un exemple simple.

La méthode présentée ici introduit un sous-ensemble du SDL, qui est nécessaire pour l'exemple simple utilisé. Une méthode plus élaborée devrait prendre en considération davantage de concepts du langage, par exemple les *services* pour réduire la complexité des *processus*.

I.3.1 Introduction

La production d'une spécification formelle débute par une spécification informelle et incomplète. Le processus de production s'achève lorsque la spécification est à la fois formelle et complète.

Le terme «formel» est pris dans le sens «suit les règles du langage formel» c'est-à-dire que la spécification SDL est conforme à la syntaxe concrète et à la sémantique statique du SDL. De plus, elle ne contient pas de *texte informel*.

Le terme «complet» est pris dans le sens «comporte toutes les informations concernant le système tel que l'a voulu l'auteur».

La tâche de production d'une spécification formelle complète est considérée comme difficile, spécialement par les personnes ayant une formation limitée dans ce domaine. La situation de la formation en matière de Techniques de Description Formelle (FDT) (*formal description technique*) est très différente de celle qui concerne les langages de programmation par exemple et il faut prendre en considération le soutien méthodologique pour une tâche de spécification formelle.

Puisque chaque étape s'en tient strictement aux règles du SDL, les outils dédiés au SDL peuvent être utilisés très largement.

Les spécifications SDL peuvent être produites en ce qui concerne:

- la structure;
- le comportement;
- les données.

La spécification est naturellement incomplète, jusqu'à ce que tous ces trois aspects soient complètement couverts. Cependant, cette spécification peut être une spécification SDL formellement valide avant cette étape.

Il est souhaitable de couvrir les trois aspects en parallèle. Les principes suivis pour la définition des étapes sont que chaque étape:

- constitue un point d'arrêt naturel dans le processus de production;
- doit produire un résultat complet par lui-même qui apparaît compréhensible et peut être vérifié, de préférence par des outils;
- n'est pas (ou seulement dans une faible mesure) dépendant des étapes suivantes.

Chaque étape est décrite par:

- **Une description** – Le processus conceptuel de l'étape. Elle est formulée en utilisant des concepts généraux et les concepts SDL correspondants (en *italique*). Dans la mesure du possible, certaines heuristiques sont fournies. Elle comprend un résumé d'actions et une partie discussion.
- **Un résultat** – Le résultat de l'accomplissement de l'étape.
- **Un exemple** – L'étape appliquée à la spécification d'un petit système d'ascenseur (la spécification complète est donnée dans I.3-3). L'*exemple* a été omis dans certaines étapes de façon à réduire la taille du paragraphe.

I.3.2 Les étapes

Etape 1 – Les limites du système

Description

- Délimiter le *système* de son environnement. Trouver un nom convenable pour le *système*. Identifier les agents de l'environnement du *système* (c'est-à-dire les entités avec lesquelles interagit le *système*). Décrire l'objet et les caractéristiques du *système* informellement dans un *commentaire*.
- Spécifier un *canal* pour chaque agent identifié de l'environnement et lui donner un nom convenable correspondant au nom de l'agent.
- Introduire un *bloc* fictif dans le *système*. Ce *bloc* sera remplacé plus tard par la structure du système réel.

– Identifier le flux d'information, en termes d'événements discrets, à la frontière du *système*. Ces événements constituent l'alphabet du système et sont modélisés par des *signaux*. Les signaux destinés à la communication externe sont spécifiés au niveau du *système*. Indiquer le rôle de chaque *signal* dans un commentaire pour chaque *spécification de signal*. Identifier les informations qui doivent être transportées par les *signaux* et indiquer les *sortes* des *paramètres des signaux*. Utiliser des syntype, des *sortes* énumérées et des *sortes* prédéfinies dans la mesure du possible. Associer les *signaux* aux *canaux*, soit directement soit en utilisant des *listes de signaux*.

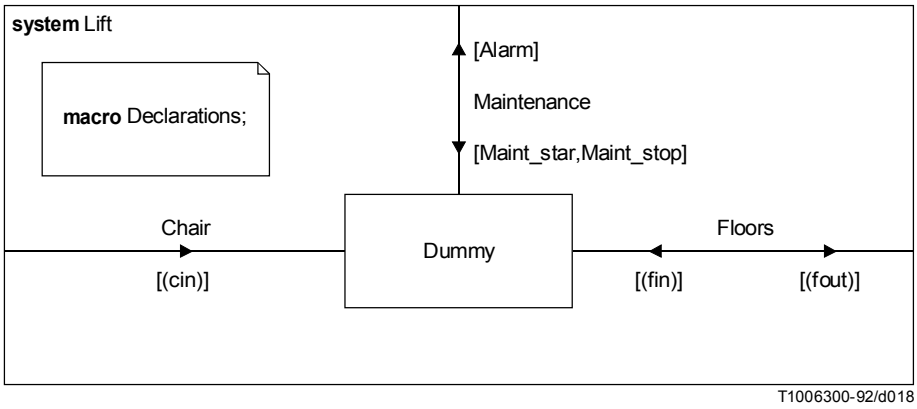
– Produire un squelette de spécification (sans signature) pour chaque nouvelle *sorte* introduite.

On doit garder présent à l'esprit que le nombre de *signaux* peut être réduit par qualification des *signaux* avec des *paramètres de signaux*.

Il faut se rendre compte que cette étape constitue une très grande simplification des efforts considérables nécessaires dans les phases préliminaires de recensement des besoins, d'analyse des systèmes, etc. Les projets peuvent bénéficier de l'usage de techniques autres que le SDL, en prélude à cette étape. L'analyse orientée-objet est présentée plus en détail au I.4.

Résultat – La spécification de la frontière du système, avec sa structure interne non définie.

Exemple




```
macrodefinition Declarations;

/*  Un ascenseur comprend une cabine avec des boutons pour aller à l'étage désiré et
des commandes d'étage où à chaque étage les usagers peuvent demander à descendre ou
à monter.

Cette spécification ne modélise pas la partie mécanique de l'ascenseur.

La demande de service est: s'il y a encore des étages à desservir dans la direction
courante, alors continuer jusqu'au prochain de ces étages. S'il y a un étage à
desservir dans la direction opposée, alors changer de direction. S'il n'y a aucun
étage en attente de desserte, rester à l'étage courant.

Un utilisateur peut commander l'ascenseur par:

– une pression sur un bouton à n'importe quel étage pour aller dans la direction
souhaitée;

– une pression sur un bouton dans la cabine pour un étage déterminé.

De plus, la spécification couvre aussi quelques cas inhabituels, tels que démarrage
et arrêt du fonctionnement de l'ascenseur.        */
```

```
block Dummy;
/* syntaxe de bloc minimale */
process Dummy;start;stop;endprocess;
endblock;
signal
  Goto(Floor),                /* requête pour aller à une destination donnée */
  Floor_req(Direction,Floor), /* requête de l'ascenseur à partir d'un étage */
  Open_door(Floor),           /* ouvrir la porte à l'étage                */
  ..........
signallist cin = Goto, .......;
signallist fin = Floor_req, .......;
signallist fout = Open_door, .......;
syntype Floor
......
endsyntype;
newtype Direction
......
endnewtype;

endmacro Declarations;
```

Etape 2 – Structure du système

Description

- Identifier les principaux constituants conceptuels internes au système et leur donner un nom. Ce sont les *blocs* du *système*. Trouver un nom convenable pour chaque *bloc* et décrire le *bloc* et ses relations avec son environnement (sa structure englobante) de façon informelle dans un *commentaire* dans la *spécification de bloc*.
- Connecter les *blocs* à la frontière du *système* à l'aide des *canaux* déjà introduits à l'**étape 1**.
- Identifier le flux d'information (*canaux* et *signaux* associés) entre les *blocs*. Spécifier les nouveaux *signaux* introduits, comme à l'**étape 1**, au niveau du système.
- Fournir un squelette de spécification pour chaque nouvelle sorte introduite, comme à l'**étape 1**, au niveau du système.

Il est conseillé de ne pas avoir trop de *blocs* dans la même structure. Si le nombre de *blocs* est trop grand, c'est-à-dire qu'il faut différer la vue d'ensemble et procéder à des imbrications (examiné à l'**étape 3**). Conformément aux heuristiques: un *bloc* doit avoir une forte cohésion interne et doit être facilement compréhensible par lui-même. Un bloc doit correspondre à un concept en soi, comme par exemple «commutateur local», «commande d'ascenseur».

Lorsque l'on identifie les *blocs*, il faut prendre en considération les propriétés suivantes:

- Un *bloc* détermine la visibilité : les *signaux* et *sortes* locaux peuvent être spécifiés à l'intérieur d'un *bloc*;
- La communication entre *blocs* (c'est-à-dire les *canaux*) peut se faire avec délai.

Un *bloc* est considéré comme un système en soi : Il définit un alphabet externe pour le sous-système à l'intérieur du *bloc* et la frontière de son environnement.

Un bloc peut être spécifié directement comme une instance de bloc. Cependant, le SDL orienté-objet introduit une distinction nette entre *type* et *instance*. Si le bloc ou un autre bloc qui lui est similaire, doit être utilisé quelque part ailleurs dans le système, on doit d'abord identifier les caractéristiques les plus fondamentales des blocs semblables dans un type de bloc. Les blocs réels peuvent alors être instanciés à partir du type général ou peuvent être des spécifications du type général. Cette observation constitue une simplification des méthodes orientées objet qui préconisent la définition des concepts en *hiérarchies de types* etc., de plus amples informations concernant ce sujet sont fournies en I.4.

Un délai de transmission peut être affecté à un *canal*, de sorte que les effets spécifiques d'une modélisation peuvent être exprimés en ayant plusieurs *canaux* entre deux constituants. Normalement, deux *blocs* à l'intérieur du système n'ont besoin que d'un seul *canal* pour être interconnectés, mais la modélisation faisant appel à plusieurs *canaux* peut être utile dans des cas particuliers.

L'usage des diagrammes de séquences de messages (MSC) peut être déjà considéré à cette étape, de façon à obtenir une vue d'ensemble utile des scénarios caractéristiques de la communication entre les constituants du système (l'usage des MSC est discuté à l'**étape 5** et est présenté en détail en I.6).

Résultat – Identification des *blocs* au niveau du *système*

Exemple – Dans cet exemple, un seul bloc est nécessaire et il est spécifié directement.

```
block Control;
/* Le bloc va décrire les commandes du système complet d'ascenseur */
/* syntaxe de processus minimale */
process Dummy; start; stop; endprocess;
endblock;
```

Etape 3 – Partitionnement des blocs

Description

- Découper chaque *bloc* complexe en *sous-blocs* (même chose que l'**étape 2** pour le *système*)
- Répéter l'opération jusqu'à ce qu'il n'y ait plus de *blocs* complexes.

Si le système est important, certains *blocs* peuvent être considérés comme des systèmes par eux-mêmes et être partitionnés de nouveau en respectant les règles données pour le *système*. Dans la structure résultante, chaque *bloc non partitionné* est un contenant pour des spécifications de comportement. (Le comportement est décrit en étapes successives par un certain nombre de *processus* à l'intérieur de chaque *bloc.*)

Résultat – Un arbre de *blocs* ayant le *système* pour racine et les blocs non partitionnés pour feuilles.

Exemple – Aucun, puisque nous affirmons que le système *Lift* est si petit qu'aucun partitionnement n'est nécessaire.

Etape 4 – Les constituants des blocs

Description

- Identifier les activités dans chacun des blocs non partitionnés. Ce sont les *ensembles de processus* du *bloc*. Choisir un nom convenable pour chaque *ensemble de processus* et le spécifier avec ses relations le liant à son environnement (le *bloc* englobant) de manière informelle dans un *commentaire* à l'intérieur de la *spécification de processus.*
- Connecter les *ensembles de processus* aux *canaux* à la frontière de *bloc* avec des *acheminements de signaux* (*signal routes*).
- Identifier le flux d'information (*acheminement de signaux* et *signaux* associés) entre les *ensembles de processus*. Spécifier les nouveaux *signaux* introduits comme à l'**étape 1**.
- Fournir un squelette de spécification pour chaque nouvelle *sorte* introduite, comme à l'**étape 1**.

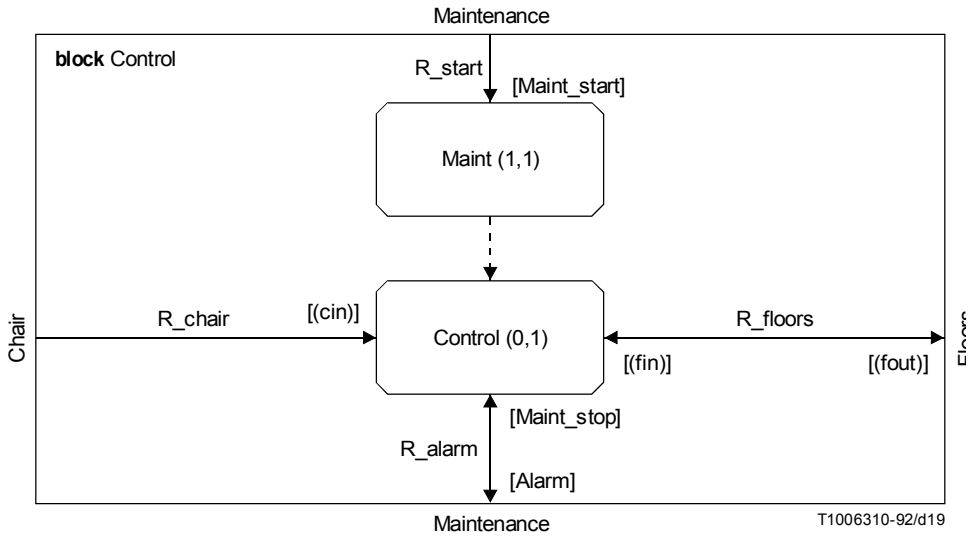
Spécifier pour chaque ensemble de processus le *nombre d'instances* (c'est-à-dire le *nombre initial* et le *nombre maximal* d'instances. Pour chaque *bloc,* au moins un *ensemble de processus* doit avoir un *nombre initial* plus grand que zéro. Il peut être utile de fixer le *nombre maximal* si chaque *instance de processus* correspond à une ressource limitée (par exemple quelque dispositif matériel). Les considérations relatives à l'identification de *type* énoncées à l'**étape 2**, s'appliquent aussi à cette étape pour les *types de processus*.

Des directives pour l'identification des *ensembles de processus* sont: chaque *ensemble de processus* représente un modèle d'activité et ce modèle peut exister en un certain nombre d'instances simultanées. La plupart des heuristiques énoncées à l'**étape 2** s'appliquent également à cette étape.

L'alphabet d'entrée d'un *ensemble de processus* peut être spécifié soit par des *acheminements de signaux (signal routes)* soit par un *ensemble de signaux (signalset*). Un *acheminement de signal* est un chemin de communication entre des *ensembles de processus* à l'intérieur d'un *bloc* ou entre des *ensembles de processus* et l'environnement d'un *bloc* (comparer avec un *canal* qui est chemin de communication entre des *blocs* et leur environnement).

Résultat – Identification d'un *ensemble de processus* à l'intérieur de *blocs* non partitionnés.

Exemple




```
process Control;
/* commande le fonctionnement de l'ascenseur */
start; stop; /* syntaxe de processus minimale */
endprocess;
process Maint;
/* initialise le fonctionnement de l'ascenseur */
start; stop; /* syntaxe de processus minimale */
endprocess;
```

Etape 5 – Spécification d'un squelette de processus

Description

- Identifier les cas typiques d'utilisation et les décrire par exemple à l'aide des diagrammes de séquences de messages (MSC).
- Prendre les décisions complémentaires concernant la façon de modéliser le comportement. Ceci peut exiger l'introduction de nouveaux *signaux* et *sortes*, ceux-ci sont spécifiés comme à l'**étape 1**.
- Ecrire un squelette de *spécification de processus* couvrant les cas d'utilisation, mais ne pas déjà considérer la combinaison de ceux-ci.
- Considérer l'usage de *procédures* pour cacher les détails et les *temporisateurs* pour la gestion du temps. Introduire les *synonymes externes* pour les valeurs non spécifiées.

L'ordre des événements sur un axe vertical (voir l'exemple ci-dessous) dans un MSC donne un ordre des événements dans une *spécification de processus*. Cet ordre constitue une (partie de) squelette de *spécification de processus*. Les MSC peuvent même être utilisés dans des étapes plus en amont pour spécifier la communication à l'intérieur du *système*. Dans ce cas un axe de MSC peut représenter un *bloc* entier. L'utilisation des MSC est décrite en détail en I.6.

En partant du symbole de début de chaque *spécification de processus*, construire un arbre d'*états* en considérant la traversée «normale» de la *spécification de processus*. Introduire si nécessaire la *création* dynamique d'*instances de processus*, sans y inclure les *paramètres*. Indiquer les *paramètres* dans les *entrées* et dans les *sorties*.

La supervision du temps peut aussi être exprimée avec les MSC. Conformément aux heuristiques: la supervision du temps est utilisée pour modéliser une mesure de temps à l'intérieur du modèle, pour superviser la libération d'une ressource et pour superviser les réponses en provenance de sources non fiables (par exemple un autre noeud du réseau). La supervision du temps est traitée par l'introduction de *temporisateurs (timers*) et par les actions d'*armement* (*set*) et de *désarmement (reset*).

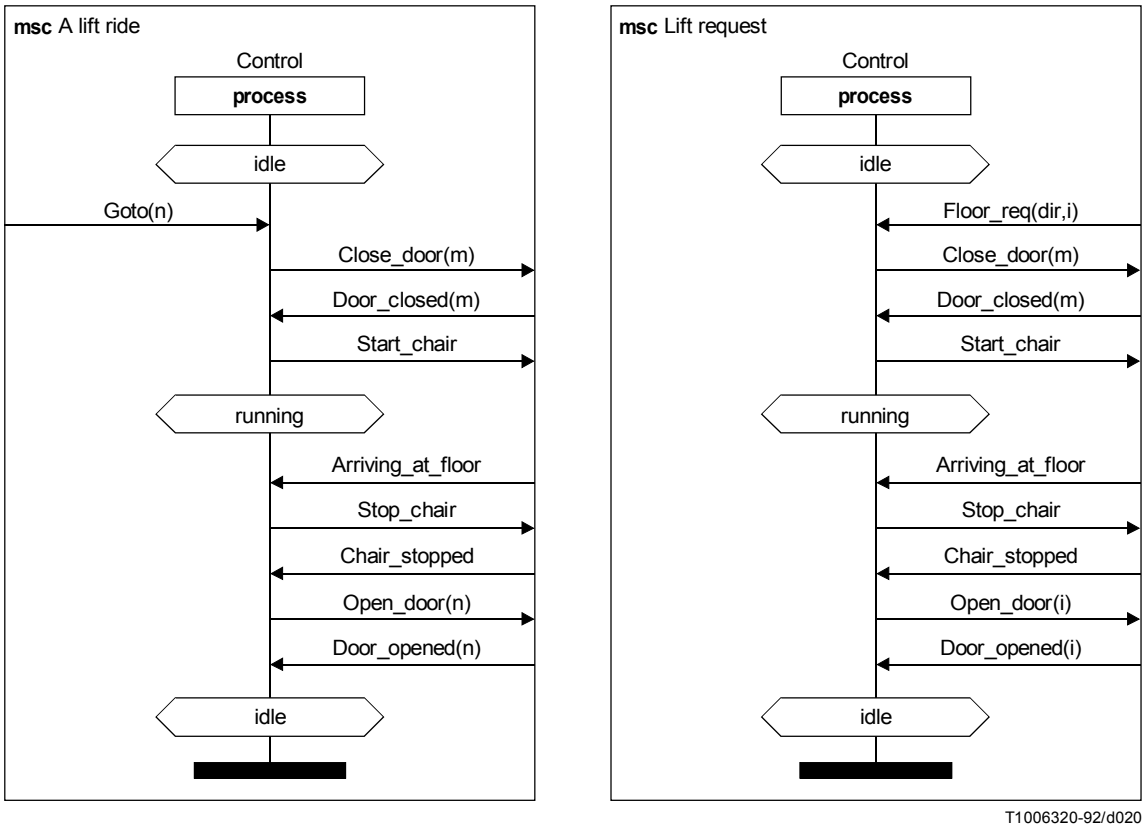
Si certaines valeurs sont naturellement laissées non spécifiées (par exemple le dernier étage d'un ascenseur), le système décrit est *générique*. Cet aspect peut être exprimé, par exemple, par l'utilisation de *synonymes externes*. La prise en compte de celles des parties d'une spécification qui devraient être génériques est un point important pour accroître la réutilisation d'une spécification (une spécification SDL d'un ascenseur pour une maison ayant un nombre quelconque d'étages est plus utile que celle donnée pour une maison de 14 étages).

Résultat – Les MSC et les squelettes de *spécifications de processus*

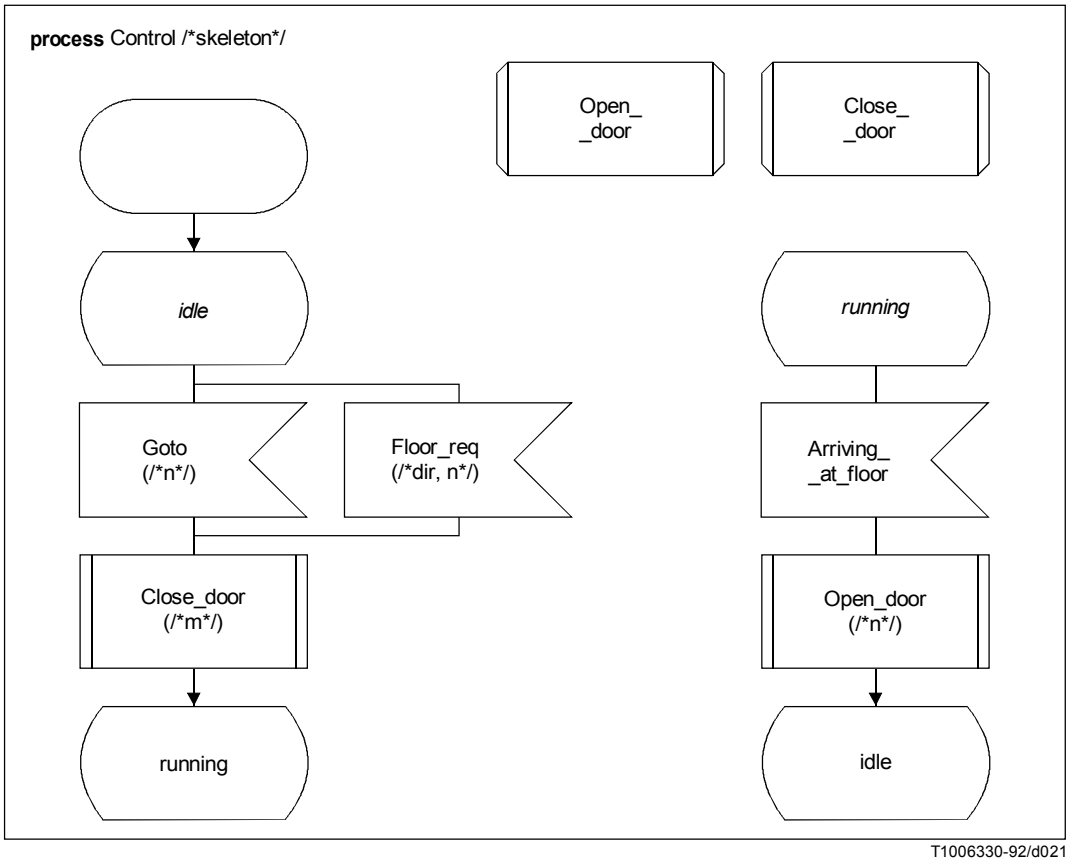
Exemple – Certaines décisions concernant la modélisation du comportement doivent être faites, par exemple, comment indiquer l'arrivée de la cabine d'ascenseur à un étage. Ici, ceci est fait à la réception du signal *Arriving_at_floor* en provenance de l'environnement.

Le MSC *A lift ride* décrit la situation simple suivante: un utilisateur se tenant au $m^{ième}$ étage entre dans la cabine d'ascenseur (qui est supposée l'attendre à l'étage), appuie sur le bouton *goto(n)* et est transporté au $n^{ième}$ étage où la porte s'ouvre à son arrivée.

Maintenant, s'il se trouve que la cabine de l'ascenseur est en attente à un autre étage, alors l'utilisateur doit d'abord appeler l'ascenseur. Cette situation est décrite dans le MSC *Lift request* (L'utilisateur se trouve au $i^{ème}$ étage).



Le squelette de la *spécification de processus* couvre le cas d'un utilisateur unique et se déduit du MSC ci-dessus (noter que l'étage *i* est traité de la même manière que l'étage *n*). Certains détails de signalisation sont cachés dans la spécification de processus par l'utilisation des procédures *Open_door* (ouvrir la porte) et *Close_door* (fermer la porte).



Etape 6 – Spécifications informelles de processus

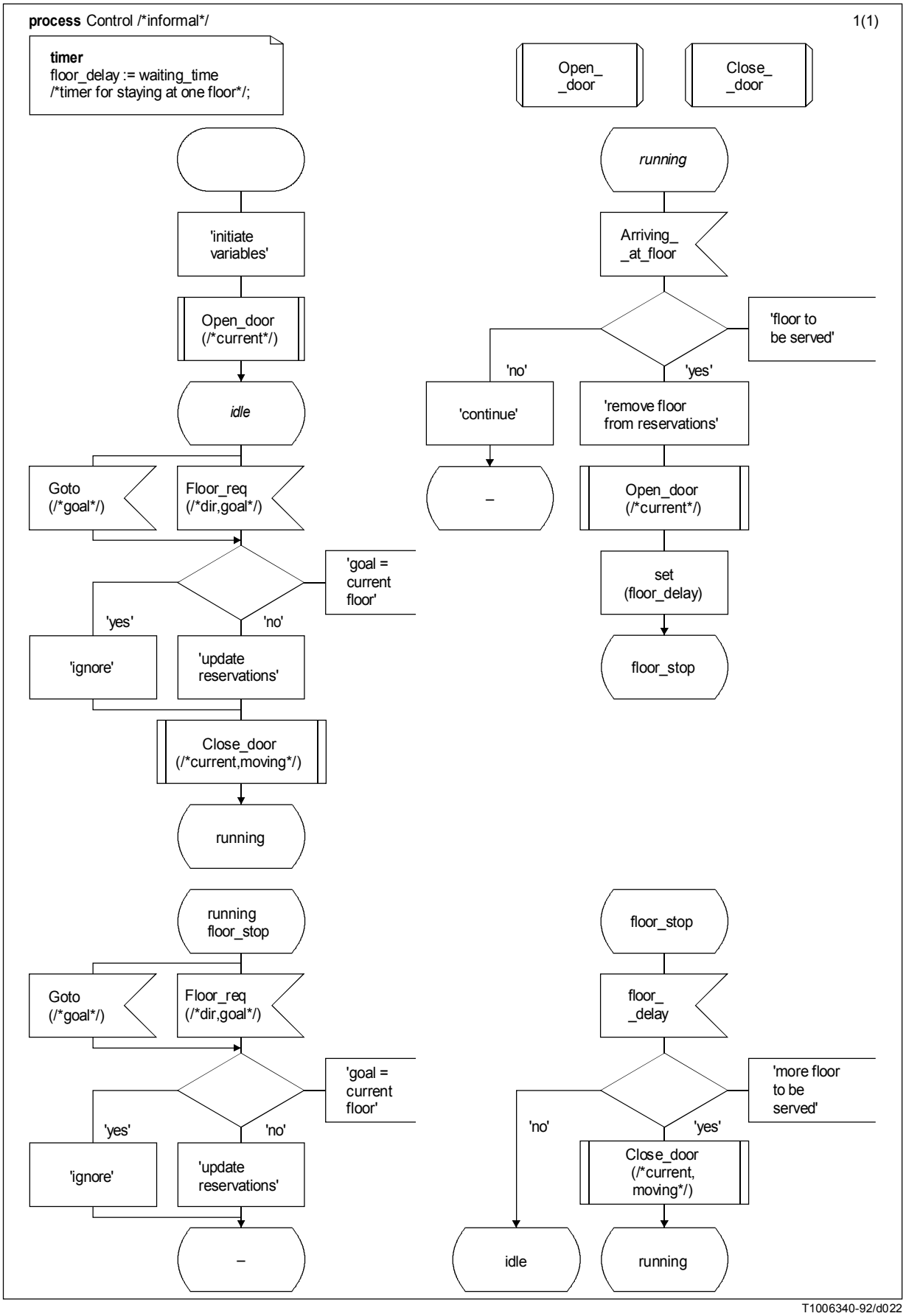
Description

- Considérer la combinaison de cas d'utilisation et les décrire, le cas échéant, en utilisant les MSC.
- Identifier les informations qui doivent être mémorisées dans les *processus*.
- Introduire les *tâches* et les *décisions*, mais n'utiliser que des *textes informels*. Introduire, éventuellement, de nouveaux *états*.
- Ecrire un squelette de spécification pour chaque *procédure* et indiquer la *sorte* des *paramètres de procédure*.
- Indiquer la sorte des *paramètres formels de processus*.
- Spécifier chaque nouvelle *sorte* introduite, comme à l'**étape 1**.

Le choix entre différentes *entrées* est réalisé pour un *état* donné. Le choix entre les *sorties* est réalisé au moyen d'une *décision informelle*. Les choix entre les *entrées* et les *sorties* sont réalisés par une combinaison de *décisions informelle* et d'*état*.

Résultat – Les spécifications de processus informelles**.**

Exemple – Dans le cas d'utilisations multiples, les requêtes *Goto* et *Floor_req* peuvent être reçues pendant le déplacement de la cabine d'ascenseur et doivent être mémorisées dans une table de réservation. Un nouvel état *floor_stop* est introduit pour un arrêt temporaire à un étage. La durée de l'arrêt temporaire est déterminé par le temporisateur *floor_delay*.


process Control /*informal*/
1(1)
timer
floor_delay := waiting_time
/*timer for staying at one floor*/;
'initiate variables'
Open_door (/*current*/)
idle
Goto (/*goal*/)
Floor_req (/*dir,goal*/)
'goal = current floor'
'yes'
'no'
'ignore'
'update reservations'
Close_door (/*current,moving*/)
running
running floor_stop
Goto (/*goal*/)
Floor_req (/*dir,goal*/)
'goal = current floor'
'yes'
'no'
'ignore'
'update reservations'
–
Open_ _door
Close_ _door
running
Arriving_ _at_floor
'floor to be served'
'no'
'yes'
'continue'
'remove floor from reservations'
–
Open_door (/*current*/)
set (floor_delay)
floor_stop
floor_stop
floor_ _delay
'more floor to be served'
'no'
'yes'
Close_door (/*current, moving*/)
idle
running
T1006340-92/d022

Le cas de l'utilisation multiple peut être aussi décrit par des MSC (voir I.9.1.2).

```
macrodefinition Declarations;
......
synonym waiting_time Duration = external;
       /* durée de l'arrêt à l'étage */

endmacro Declarations;
```

Etape 7 – Spécifications complètes de processus

Description

- Considérer également les cas inhabituels, tels que les situations d'erreur.
- Compléter les spécifications de procédures.
- Vérifier que toutes les combinaisons état/signal sont traitées.

Cette étape se termine lorsque aucun *état suivant(nextstate*) non couvert n'est introduit dans les *transitions*. Un *état suivant* non couvert correspond à un *état* qui n'a pas encore été pris en compte. Une *matrice signal/état* peut être utile ici, (voir I.9.4).

Résultat – *Spécifications de processus* complètes mais informelles

Etape 8 – Spécifications formelles de processus

Description

- Identifier les *sortes* nécessaires pour mémoriser les informations. Spécifier la *signature* de chaque *sorte* introduite jusqu'ici et utiliser des *textes informels* à la place des *équations*.
- Spécifier les variables pour les informations mémorisées et les paramètres d'entrée. Spécifier les *paramètres formels de processus*.
- Remplacer le *texte informel* dans les *tâches*, les *décisions* et les *réponses* par des *affectations* et des *expressions*.
- Introduire des paramètres dans les *entrées*, les *sorties*, les *créations de demandes* et les *appels de procédure*.

La *signature* d'une *sorte* comprend les *opérateurs* avec le typage et les *littéraux*. Hériter autant que faire se peut des *sortes* prédéfinies. Identifier (dans un *commentaire*) l'ensemble des *opérateurs* et des *littéraux* qui peuvent être utilisés pour représenter toutes les valeurs possibles. Il s'agit des constructeurs de la *sorte* (voir I.5).

Résultat – *Spécifications de sortes* semi-formelles et *spécifications de processus* formelles

Exemple

```
macrodefinition Declarations;
......
newtype Direction
  literals up, down;/* constructeurs:up,down */
  operators
   change_dir: Direction -> Direction;
  axioms
   'if direction is up then down, else up'
endnewtype;
........
endmacro Declarations;

process Control;
dcl
  goal Floor,               /* l'étage cible */
  towards Direction,        /* la direction indiquée */
  here Floor,               /* étage courant */
  moving Direction,         /* direction courante de la cabine */
  table Reservations,       /* table des réservations */
  ........
endprocess;
```

De plus:

```
task 'initiate variables';
```

devient:

```
task here := bot_floor;
task moving := up;
task table := empty_serv;
```

et

```
decision 'goal = current floor';
```

devient:

```
decision goal = here;
```

Etape 9 – Spécifications de sortes formelles

Description

- Formaliser les *équations* en remplaçant les *équations* informelles par des *équations* formelles.
- Ajouter les *équations* aux spécifications de *sortes*, jusqu'à ce que l'ensemble des *équations* soit complet.
- En complément aux *équations*, spécifier les opérateurs au moyen de *spécifications d'opérateur*.

Spécifier d'abord les *équations* pour les constructeurs. Ceci fournit les valeurs possibles de la *sorte*. Ensuite spécifier les *équations* pour les *opérateurs* et les *littéraux* restants. L'ensemble des équations est complet lorsque toutes les expressions contenant un *opérateur* qui n'est pas un constructeur et des *littéraux* peuvent être réécrites en des expressions ne contenant que des *opérateurs* constructeurs et des littéraux. Voir I.5 pour plus de détails.

Les spécifications d'opérateur sont similaires aux procédures retournant une valeur (en fait, elles se définissent par une transformation en de telles procédures) et sont recommandées pour ceux qui ne sont pas familiers avec les équations.

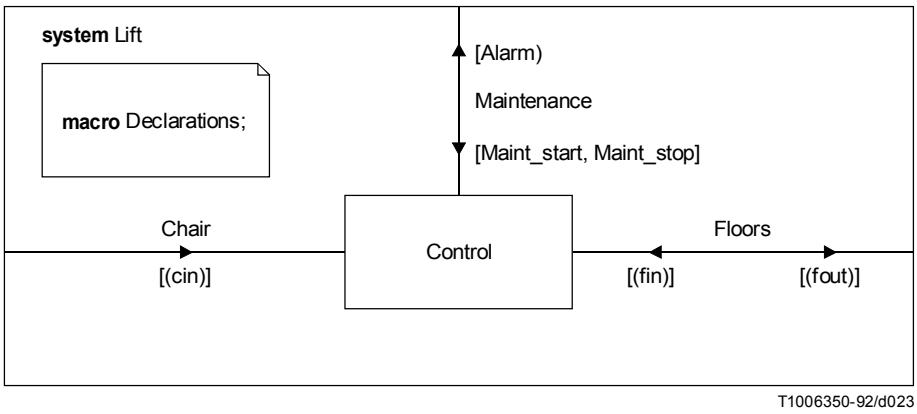
Résultat – *Spécifications de sorte* complète et formelle**.**

Exemple

```
macrodefinition Declarations;
......
newtype Direction
  literals up, down;/* constructeurs:up,down */
  operators
   change_dir: Direction -> Direction;
  axioms
   change_dir(up) == down;
   change_dir(down) == up;
endnewtype;
synonym waiting_time Duration = external;
       /* durée de l'arrêt à l'étage */
newtype Reservations array(Floor,Floor_indicator) adding
  literals empty_serv;
  operators
   goto_req: Reservations, Floor, Floor -> Reservations;
    /* note une demande pour aller de l'étage courant à un autre étage */
..........
  operator goto_req;
   fpar table Reservations, here Floor, goal Floor;
   returns Reservations;
   start;
    decision goal > here;
      (true):
        task table(goal)!upwards := true;
      (false):
        decision goal < here;
         (true):
```

```
            task table(goal)!downwards := true;
          (false):
             /* pas d'action */
         enddecision;
     enddecision;
     return table;
  endoperator;
.........
endnewtype Reservations;
.........
endmacro Declarations;
```

I.3.3 Exemple: l'ascenseur

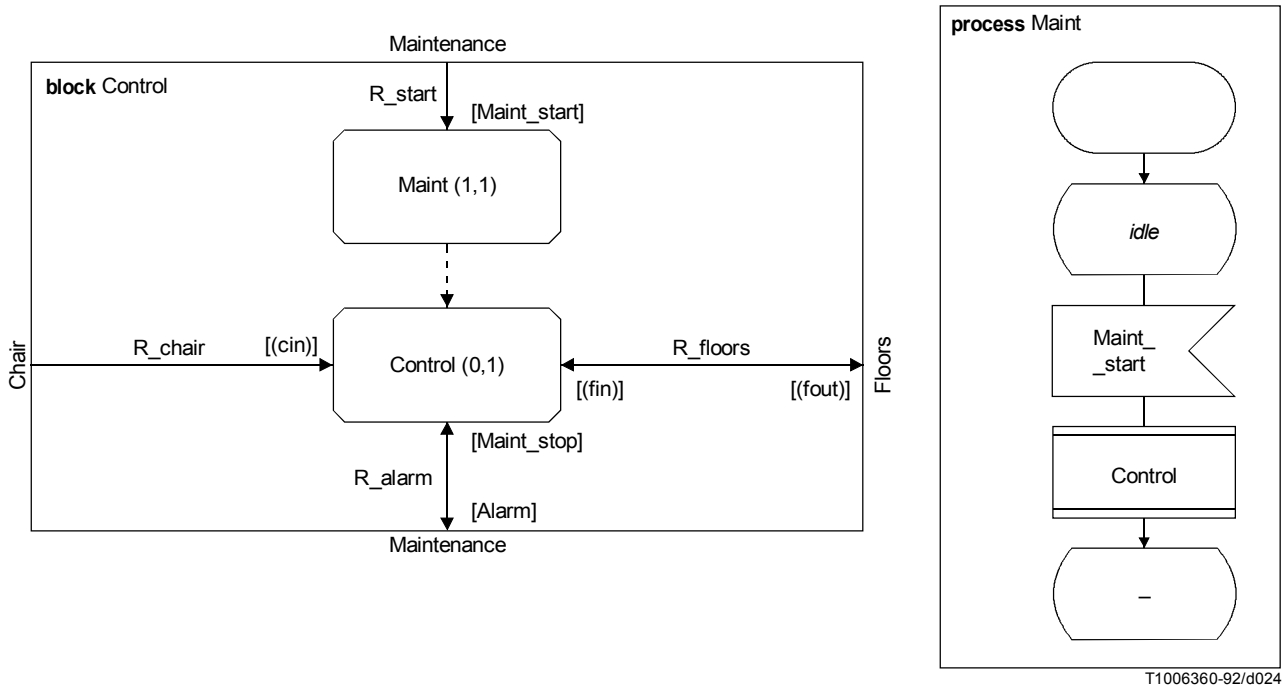


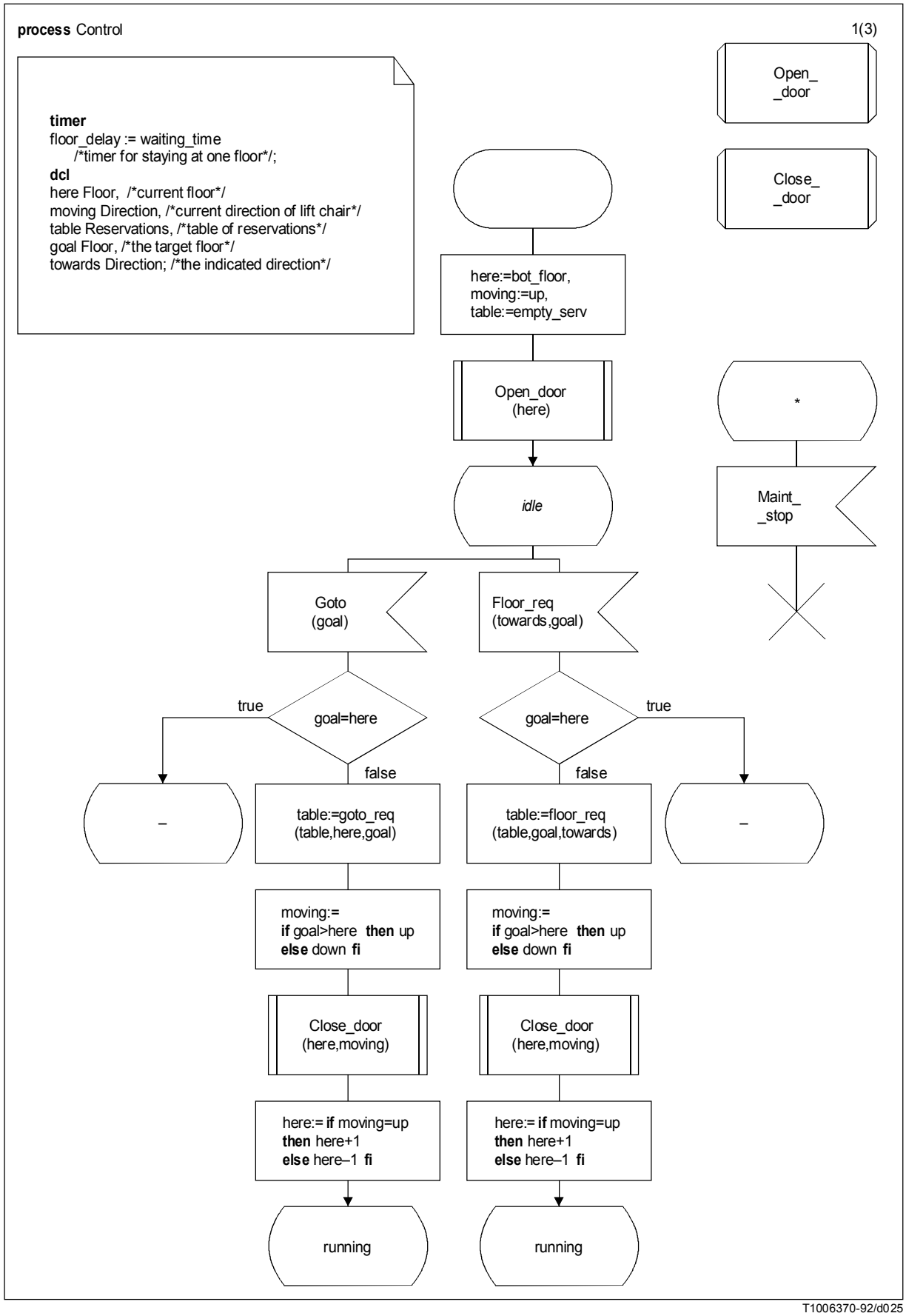
T1006350-92/d023

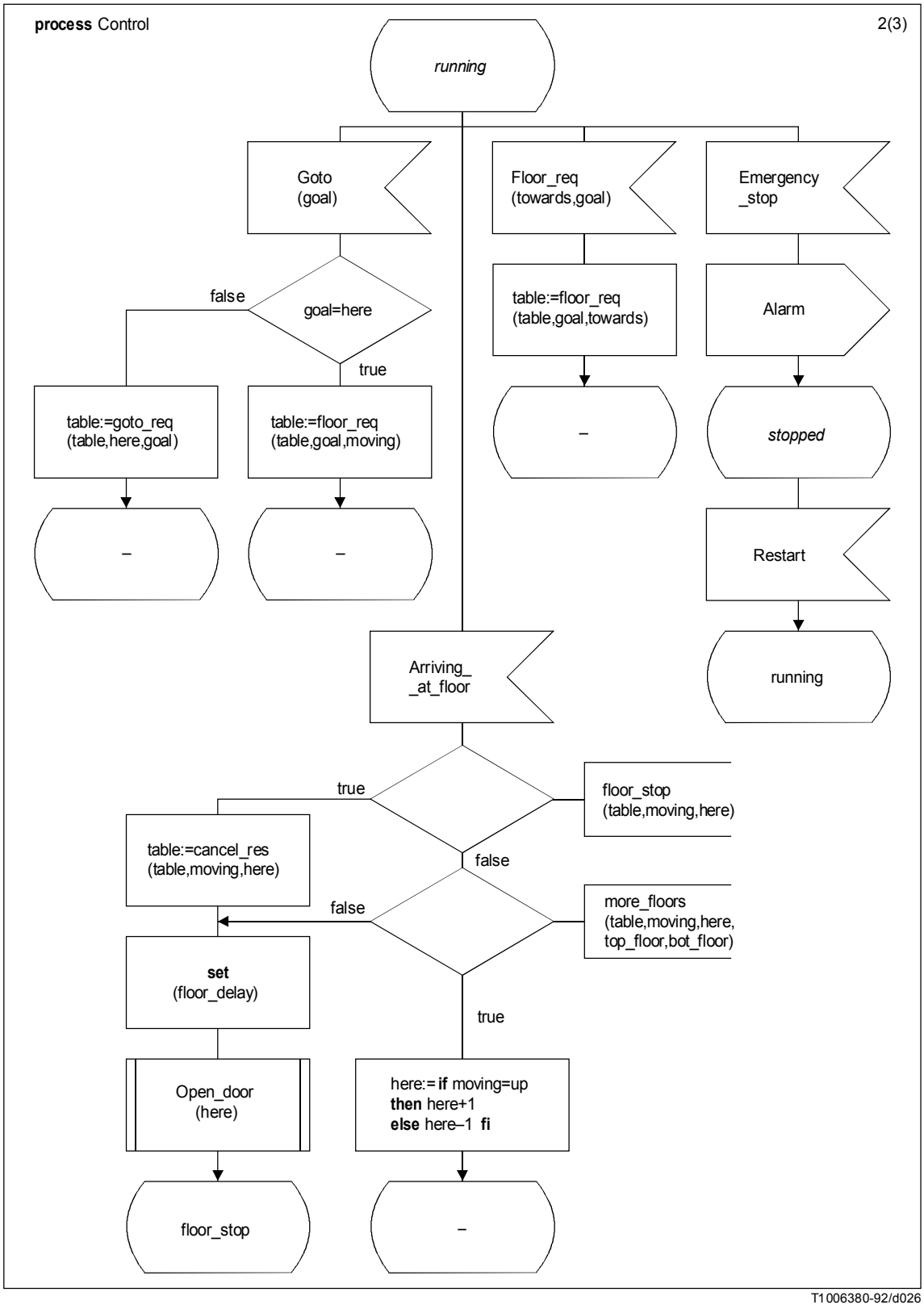
```
macrodefinition Declarations;

/* Un ascenseur comprend une cabine avec des boutons pour aller à l'étage désiré et
des commandes d'étage où à chaque étage les usagers peuvent demander à descendre ou
à monter.

Cette spécification ne modélise pas la partie mécanique de l'ascenseur.

La demande de service est : S'il y a encore des étages à desservir dans la direction
courante, alors continuer jusqu'au prochain de ces étages. S'il y a un étage à
desservir dans la direction opposée, alors changer de direction. S'il n'y a aucun
étage en attente de desserte, rester à l'étage courant.

Un utilisateur peut commander l'ascenseur par:
– une pression sur un bouton à n'importe quel étage pour aller dans la direction
souhaitée;
– une pression sur un bouton dans la cabine pour un certain étage.

De plus, la spécification couvre aussi quelques cas inhabituels, tels que démarrage
et arrêt du mouvement de l'ascenseur.       */

signal
  Alarm,                      /* message d'alarme                */
  Arriving_at_floor,          /* arrivée à l'étage       */
  Chair_stopped(Floor),       /* cabine arrêtée à l'étage              */
  Close_door(Floor),          /* fermer la porte à l'étage              */
  Door_closed(Floor),         /* porte fermée à l'étage             */
  Door_opened(Floor),         /* porte ouverte à l'étage            */
  Emergency_stop,             /* arrête la cabine immédiatement           */
  Floor_req(Direction,Floor), /* requête de l'ascenseur à partir d'un étage */
  Goto(Floor),                /* requête pour aller à une destination donnée  */
```

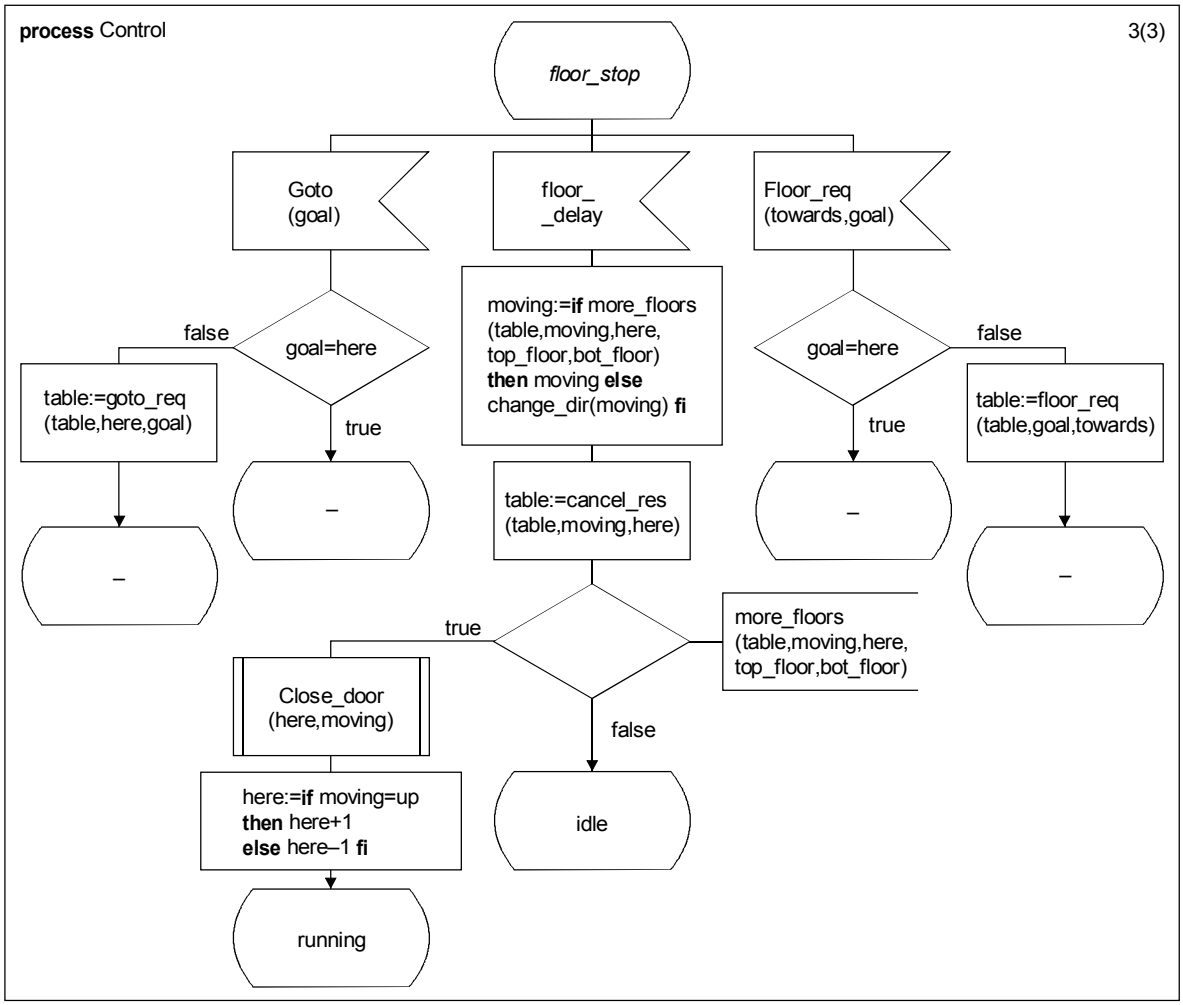
```
  Maint_start,                /* mise en route de l'ascenseur          */
  Maint_stop,                 /* arrêt de l'ascenseur          */
  Open_door(Floor),           /* ouvrir la porte à l'étage                */
  Restart,                    /* redémarrer la cabine arrêtée             */
  Start_chair(Direction),     /* démarrer la cabine dans une direction donnée */
  Stop_chair(Floor);          /* arrêter la cabine à l'étage               */
synonym top_floor Natural = external    /* étage le plus haut */;
synonym bot_floor Natural = external     /* étage le plus bas */;
synonym waiting_time Duration = external;
       /* durée de l'arrête à un étage */
signallist cin = Goto, Emergency_stop, Restart;
signallist fin = Floor_req, Door_closed, Door_opened, Chair_stopped,
  Arriving_at_floor;
signallist fout = Open_door, Close_door, Stop_chair, Start_chair;
syntype Floor = Natural
  constants bot_floor : top_floor
endsyntype;
newtype Direction
  literals up, down;
  operators
   change_dir: Direction -> Direction;
  axioms
   change_dir(up) == down;
   change_dir(down) == up;
endnewtype;
newtype Floor_indicator
  struct upwards Boolean; downwards Boolean;
endnewtype Floor_indicator;
newtype Reservations array(Floor,Floor_indicator) adding
  literals empty_serv;
  operators
   goto_req: Reservations, Floor, Floor -> Reservations;
    /* note une demande pour aller de l'étage courant à un autre étage  */
   floor_req: Reservations, Floor, Direction -> Reservations;
    /* note une demande pour aller dans une direction donnée à partir d'un étage  */
   floor_stop: Reservations, Direction, Floor -> Boolean;
    /* vérifie si l'étage a demandé le service pour une direction donnée */
   cancel_res: Reservations, Direction, Floor -> Reservations;
    /* retire une demande de service à partir d'un étage dans une direction
...donnée */
   more_floors: Reservations, Direction, Floor, Floor, Floor -> Boolean;
    /* vérifie s'il y a encore des étages à desservir à partir de l'étage
     courant jusqu'en bas selon la direction courante   */
   more_floors!: Reservations, Direction, Floor, Floor, Floor -> Boolean;
    /* opérateur supplémentaire nécessaire pour faire une itération sur un
...intervalle  */
  operator goto_req;
   fpar table Reservations, here Floor, goal Floor;
   returns Reservations;
   start;
    decision goal > here;
      (true):
        task table(goal)!upwards := true;
      (false):
        decision goal < here;
         (true):
           task table(goal)!downwards := true;
```

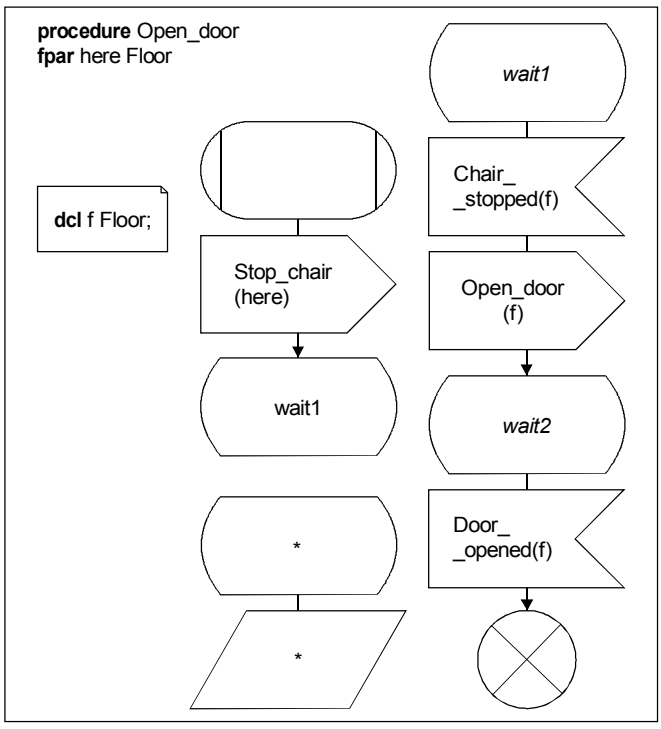
```
          (false):
             /* no action */
          enddecision;
       enddecision;
       return table;
    endoperator;
    axioms
     empty_serv == Make!(Make!(false,false));
     floor_req(r,goal,d) ==
        Modify!(r,goal, if d = up
             then upwardsModify!(Extract!(r,goal),true)
             else downwardsModify!(Extract!(r,goal),true) fi);
     floor_stop(r,d,f) ==
       if d = up then upwardsExtract!(Extract!(r,f))
                 else downwardsExtract!(Extract!(r,f)) fi;
     cancel_res(r,d,f) ==
        Modify!(r,f, if d = up
            then upwardsModify!(Extract!(r,f),false)
            else downwardsModify!(Extract!(r,f),false) fi);
     more_floors(r,d,f,top,bot) ==
        if d=up then
           if f>=top then false else more_floors!(r,d,f+1,top,bot) fi
        else
           if f<=bot then false else more_floors!(r,d,f-1,top,bot) fi
        fi
     more_floors!(r,d,f,top,bot) ==
        upwardsExtract!(Extract!(r,f))
        or downwardsExtract!(Extract!(r,f))
        or if d = up then
              if f>=top then false else more_floors!(r,d,f+1,top,bot) fi
           else
              if f<=bot then false else more_floors!(r,d,f-1,top,bot) fi
           fi
endnewtype Reservations;
endmacro Declarations;
```

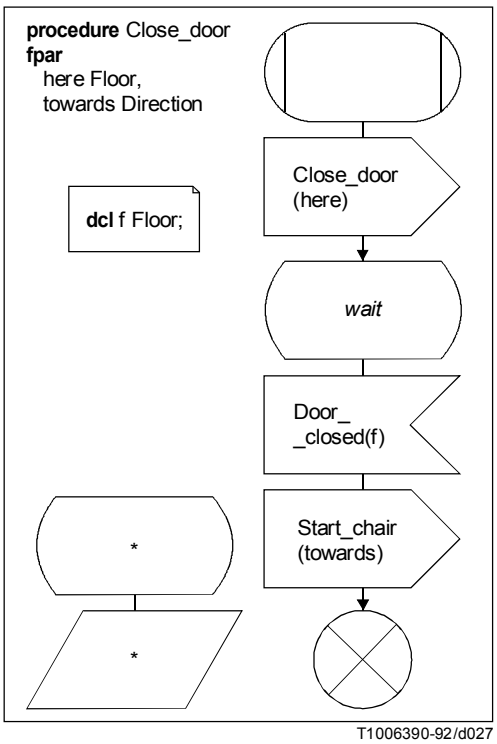



process Control
1(3)
timer
floor_delay := waiting_time
/*timer for staying at one floor*/;
dcl
here Floor, /*current floor*/
moving Direction, /*current direction of lift chair*/
table Reservations, /*table of reservations*/
goal Floor, /*the target floor*/
towards Direction; /*the indicated direction*/
Open_ _door
Close_ _door
here:=bot_floor, moving:=up, table:=empty_serv
Open_door (here)
*
idle
Maint_ _stop
Goto (goal)
Floor_req (towards,goal)
goal=here
true
false
goal=here
true
false
–
table:=goto_req (table,here,goal)
table:=floor_req (table,goal,towards)
–
moving:= if goal>here then up else down fi
moving:= if goal>here then up else down fi
Close_door (here,moving)
Close_door (here,moving)
here:= if moving=up then here+1 else here–1 fi
here:= if moving=up then here+1 else here–1 fi
running
running
T1006370-92/d025


process Control
2(3)
running
Goto
(goal)
Floor_req
(towards,goal)
Emergency
_stop
false
goal=here
true
table:=floor_req
(table,goal,towards)
Alarm
table:=goto_req
(table,here,goal)
table:=floor_req
(table,goal,moving)
–
stopped
–
–
Restart
Arriving_
_at_floor
running
true
floor_stop
(table,moving,here)
table:=cancel_res
(table,moving,here)
false
false
more_floors
(table,moving,here,
top_floor,bot_floor)
set
(floor_delay)
true
Open_door
(here)
here:= if moving=up
then here+1
else here–1 fi
floor_stop
–
T1006380-92/d026







T1006390-92/d027

I.4 Approche orientée objet et SDL

I.4.1 Analyse «orientée objet»

L'approche orientée objet est une perspective concernant la façon de structurer les systèmes et les descriptions de systèmes et peut suggérer même quelques façons d'organiser le processus de développement de système mais ne fournit en aucun cas une méthode complète.

L'approche orientée objet est souvent associée à la programmation, aux interfaces utilisateurs graphiques et aux bases de données. Appliquée à la programmation, elle est souvent associée seulement à la réutilisation du code. Il s'est avéré, cependant, que l'approche orientée objet convient aussi bien dans les phases préliminaires du processus de développement de système, où il faut insister sur la compréhension du champ de l'application et sur la première spécification et analyse d'une application. L'approche orientée objet dans ce cas n'est pas appliquée principalement en vue de la réutilisation mais parce qu'elle implique une façon de penser nouvelle et fructueuse. Les deux principales raisons pour appliquer l'approche orientée objet pendant l'analyse sont les suivants:

- L'analyse doit conduire à une meilleure *compréhension* de ce qu'est supposé faire le système. Ce que les clients veulent et ce que sont les besoins qui s'appliquent au système.
- On considère comme un avantage de *ne pas avoir besoin de changer de paradigme* lorsque l'on passe de la spécification à la mise en œuvre.

Une partie de cette compréhension est d'identifier les *phénomènes* pertinents spécifiques à l'application et de les classer en *concepts*. Les phénomènes sont représentés par des *objets*, les concepts sont représentés par les *types* des objets. Les concepts sont organisés en hiérarchies de concepts qui contribuent à la compréhension des similitudes et des différences entre les phénomènes. Les types représentant les concepts éprouvés de l'application sont candidats à la réutilisation dans de nombreuses applications différentes. A cet égard, il est important d'avoir une vue générale de l'approche orientée objet qui permette de couvrir la définition des concepts pour les aspects concernant à la fois les données et les actions. L'ensemble des concepts concernés de l'application est de grande importance pour permettre aux personnes concernées par le développement de système de comprendre le système et de raisonner à propos du système. L'ensemble des concepts concernés de l'application va constituer une base de connaissance commune pertinente pour les phases suivantes du cycle de vie du système (spécification fonctionnelle, conception, mise en œuvre et exploitation/maintenance). Le développement de système met en jeu différents groupes de personnes et ces concepts vont faciliter la communication et l'interaction entre ces groupes.

Lorsque les spécifications fonctionnelles et formelles peuvent être réalisées selon l'approche orientée objet, ceci conduit à des spécifications où les éléments essentiels sont préservés dans la mise en œuvre:

- en reflétant les éléments externes du système par des composants du système spécifié, cela va conduire également à des mises en œuvre pertinentes;
- en identifiant les concepts clés spécifiques (au domaine) de l'application, en le représentant par des types et en représentant les hiérarchies de concepts par des hiérarchies de sous-types, ceci conduit à des définitions de type qui valent aussi d'être préservées dans les mises en œuvre. Les hiérarchies de sous-types qui sont le résultat des hiérarchies de concepts de l'application seront plus stables que des hiérarchies de sous-types inventées juste dans le but de la mise en œuvre. Les hiérarchies de concepts spécifiques de l'application sont souvent le résultat de nombreuses années d'expérience d'utilisateurs et d'experts dans le domaine;
- La structure du système spécifié peut être rendue très proche de la structure du système réalisé.

Le fait que les parties essentielles de la spécification puissent rester stables en allant de la conception à la mise en œuvre, ne sera pas seulement utile pour maintenir le système, mais il fera aussi apparaître la qualité des spécifications comme un *contrat* entre le fournisseur du système et le client.

L'analyse orientée objet est illustrée par un exemple à demi-réaliste: un système bancaire (*Bank*). L'exemple est construit à partir de concepts déjà développés (en relation avec un système de contrôle d'accès (*AccessControl*) rangé dans une bibliothèque). Ceci reflète la pratique industrielle, où les nouveaux systèmes ne sont pas développés à partir de rien, mais plutôt construits industriellement à partir de composants et de systèmes existants.

L'analyse peut grossièrement être divisée en quatre phases: *compréhension*, *recherche*, *généralisation* et *spécialisation*.

Etape 1 – Description textuelle

- Faire une esquisse du système à spécifier en utilisant du texte et des dessins.

Dans la plupart des cas, quelques descriptions textuelles sont disponibles dans le domaine que vous êtes en train de modéliser. Il peut y avoir des spécifications de besoins provenant d'un demandeur, des manuels et des descriptions informelles de systèmes plus anciens ou simplement des esquisses informelles destinées à des objectifs complètement différents.

Une banque (*bank*) est un lieu où l'on gère de l'argent (*money*). L'argent peut être géré de différentes façons, il y a l'argent liquide (*cash*) et les chèques (*cheques*). L'argent peut aussi être géré sans support physique pour représenter sa valeur, par exemple, par les moyens électroniques, des transactions internes des comptes courants (*accounts*). On mettra en évidence les parties de la banque où la représentation physique de l'argent est manipulée.

Dans la banque, il y a un certain nombre de guichets (*counters*). Certains de ceux-ci (appelés caisses (*cashiers*)) ont un employé (*clerk*), les autres sont des caisses automatiques, appelées mini-banques (*minibanks*).

Certains des clients (*customers*) préfèrent la mini-banque où ils insèrent leur carte (*card*) et tapent leur code confidentiel pour avoir accès à leur compte bancaire et recevoir de l'argent liquide. Ils peuvent aussi avoir accès à d'autres services tels que l'impression (*printout*) de leur relevé de compte. La mini-banque dispose d'un système d'affichage (*display*) qui sert de moyen d'interaction avec le client.

Certains des clients préfèrent les caisses manuelles auxquelles on peut s'adresser et communiquer d'une manière humaine. L'employé, cependant, dispose de son propre affichage des données et son propre clavier. La caisse peut fournir plus de services que les mini-banques automatiques.

Cette étape conduit à la toute première compréhension de tout ce qui concerne le domaine de l'application. Elle aide à définir les limites du système et elle donne une idée des concepts à introduire dans le dictionnaire (voir ci-dessous).

Etape 2 – Dictionnaire

- Réaliser ou obtenir un dictionnaire couvrant le domaine du sujet. Le dictionnaire devra être mis à jour en permanence pendant le développement.

Dans cette étape, on améliore la compréhension en établissant une liste des concepts pertinents. Certains de ces concepts peuvent se trouver dans des dictionnaires, les autres doivent être définis.

La notion classique d'un concept se caractérise par:

- *Extension*, la collection des phénomènes que couvre le concept.
- *Intension*, une collection de propriétés qui caractérise d'une certaine manière les phénomènes dans l'extension du concept.
- *Désignation*, la collection des noms sous lesquels le concept est connu.

La représentation des concepts par les types d'instances suit ce modèle: les instances appartiennent à l'extension, la définition de type donne l'intension et le nom du type constitue sa désignation.

On isole les *noms* qui, dans la description textuelle, représentent les concepts clés du sujet en cours d'analyse. Les noms sont donnés en italique dans l'**étape 1**. Dans la terminologie orientée objet ces concepts clés sont des candidats pour les *types*, à partir desquels les objets sont engendrés. Pendant les étapes suivantes d'analyse, le dictionnaire devrait être mis à jour et complété.

La banque pourrait avoir le dictionnaire initial suivant:

Account (Compte)	Une représentation informatique de l'argent associé à un possesseur de compte
Bank (Banque)	Un lieu où l'argent est manipulé
Card (Carte)	Un moyen d'identification personnel
Cash (Argent liquide)	Représentation physique de l'argent garanti par un état
Cashier (Caisse)	Guichet avec employé
Cheque (Chèque)	Représentation de l'argent avec des garanties variées
Counter (Guichet)	Ports de communication avec le système bancaire
Customer (Client)	Processus externes fournissant de l'argent à la banque ou recevant de l'argent de la banque
Display (Affichage)	Ecran pour fournir des messages écrits volatils au client ou à l'employé
Key (Touche)	Partie d'un clavier
Minibank (Mini-banque)	Guichet automatique
Money (Argent)	Les signaux fondamentaux d'une banque
Printout (Imprimé)	Message écrit sur support papier (fourni au client)

Etape 3 – Agrégation

- Faire une ébauche d'une instance type du système en cours d'analyse. Etiqueter les éléments de l'ébauche.

- De cette instance type, décrire les relations entre concepts du type «consiste-en».

Lorsqu'on observe une banque, on peut voir quels sont les éléments qui la composent et quels sont les éléments qui composent lesdits éléments, etc. L'agrégation signifie la collection de parties séparées pour former un tout et concerne donc les relations du type «consiste-en», ce qui constitue une importante relation dans la plupart des paradigmes d'analyse, et c'est également le cas pour l'approche orientée objet. Mais lorsqu'on spécifie les relations «consiste-en», il faut faire attention de faire la distinction entre les concepts (types) et les instances de ces types. Pour éviter toute confusion, une majuscule sera utilisée en initiale pour les concepts (par exemple *Bank*) et les minuscules pour les instances (par exemple *bank*). La Figure I.4-1 montre les relations «consiste-en» pour une instance particulière de banque ayant quatre instances particulières de guichets.

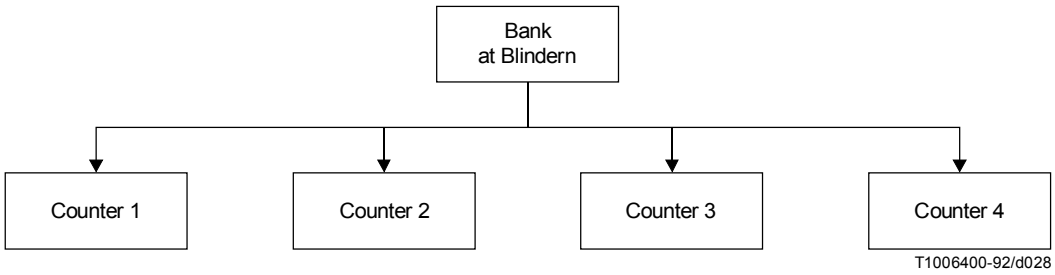


FIGURE I.4-1/Z.100

Relations «consiste-en» entre les instances

Une analyse de système s'intéresse la plupart du temps aux systèmes en général, et le moins souvent à des instances spécifiques de système. La même chose se produit pour les banques et les guichets. En conséquence, la relation «consiste-en» entre *Bank* et *Counter* est en général intéressante (voir la Figure I.4-2).

Dans cette figure, nous avons actuellement des types de relation qui sont des associations entre concepts (types d'objets). Un type de relation a une cardinalité qui indique le nombre maximal d'instances de chaque type concerné. Il est aussi possible d'indiquer que certaines instances sont facultatives (flèches tiretées), c'est-à-dire qu'elles peuvent ne pas faire partie de certaines instances de banques.

Etape 4 – Similitudes

- Pour chaque concept dans le dictionnaire, demander si tous les objets qui «tombent» dans l'extension du concept possèdent les mêmes propriétés. Si ce n'est pas le cas, trouver les spécialisations qui peuvent ou ne peuvent pas étendre le dictionnaire.

- Pendant la spécialisation, étendre la description dans le dictionnaire avec les propriétés qui complètent la compréhension du concept.

Dans l'étape précédente, les relations entre les instances ont été analysées. Cette étape concerne les relations entre les concepts.

Ainsi qu'il a été établi dans les étapes précédentes, *Minibank* et *Cashier* sont des spécialisations de *Counter*, signifiant que toutes les caisses et toutes les mini-banques sont des guichets et toutes les choses générales qui peuvent être dites pour les guichets sont valables à la fois pour les mini-banques et pour les caisses. Nous considérerons la hiérarchie de spécialisation conformément à la Figure I.4-3.

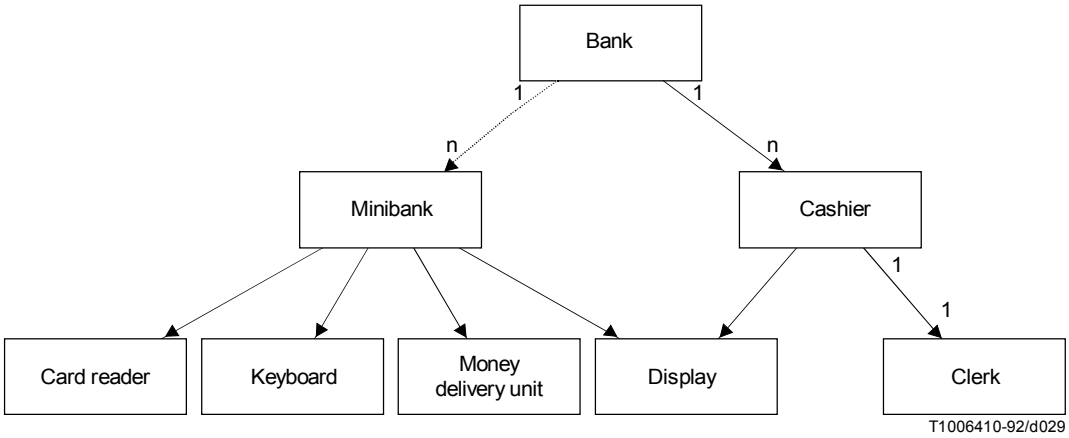


FIGURE I.4-2/Z.100

Relations «consiste-en» entre les concepts

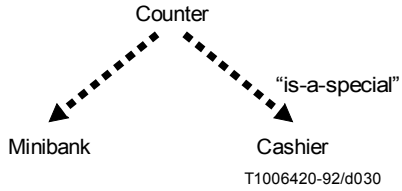


FIGURE I.4-3/Z.100

Hiérarchie de spécialisation de *counter*

Lorsque nous trouvons de telles hiérarchies de spécialisation, nous devons considérer plus en détail les définitions de ce qui caractérise une mini-banque (*Minibank*) ou une caisse (*Cashier*) et ce qui les distingue l'une de l'autre à partir de la notion générale de guichet (*Counter*). L'idée sous-jacente de telles relations de spécialisation est de décrire les aspects généraux et communs en une seule fois. La description d'un guichet (*Counter*) comprendra toutes les caractéristiques communes, tandis que les descriptions de mini-banque (*Minibank*) n'incluront que les caractéristiques spécifiques des mini-banques.

L'exemple de la banque possède davantage de hiérarchies de spécialisation, comme le montre la Figure I.4-4.

Etape 5 – Recherche en bibliothèque

– Regarder s'il y a, dans le dictionnaire, un ***Y*** qui est similaire à un ***X***. Lorsqu'une similitude est rencontrée, soit faire de ***X*** une spécialisation directe de ***Y***, soit restructurer la bibliothèque en créant ***Z*** comme généralisation commune de ***X*** et ***Y***.

En suivant notre parfaite compréhension de la banque, nous souhaitons voir s'il existe des éléments résidant dans la bibliothèque qui peuvent être utilisés pour la conception du système bancaire (*Bank*). Nous trouvons qu'il y a beaucoup de similitudes avec le système de contrôle d'accès (*AccessControl*) (voir la Figure I.4-5). Une station locale (*LocalStation*) est très semblable à un guichet (*Counter*).

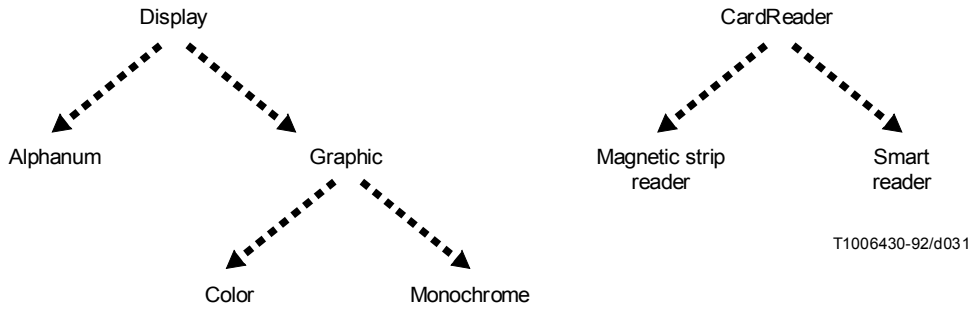


FIGURE I.4-4/Z.100

Hiérarchies de spécialisation de *Display* et *CardReader*

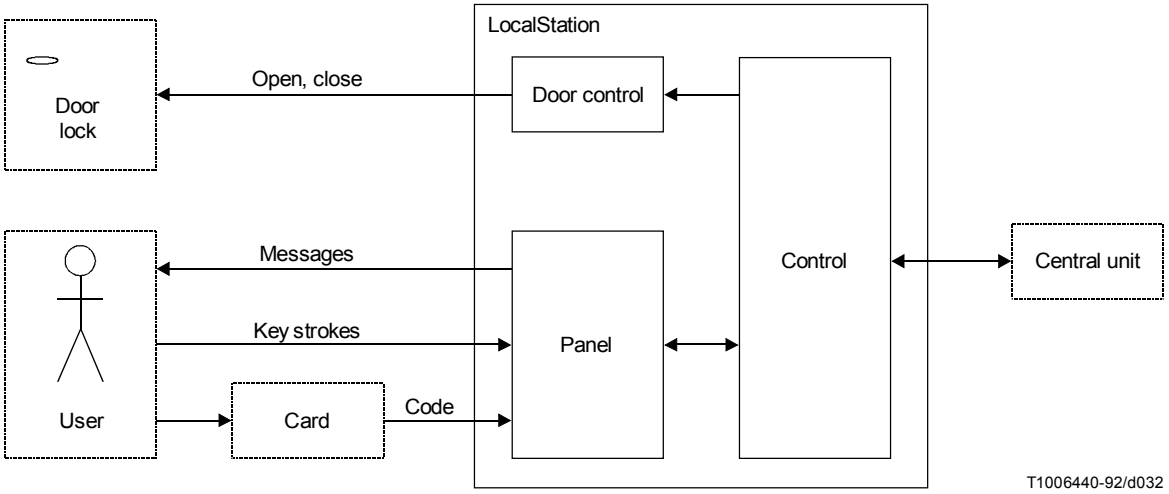


FIGURE I.4-5/Z.100

Le système AccessControl

Comment pouvons-nous tirer profit des similitudes entre *AccessControl* et *Bank*? Ce sont deux approches différentes qui dépendent de la relation de spécialisation entre le concept de la bibliothèque et celui de l'application: soit le concept de l'application est une spécialisation directe du concept de la bibliothèque, soit il n'y a pas de relation directe de spécialisation entre eux, mais ils restent néanmoins très semblables.

Si nous rencontrons le cas de relation de spécialisation ci-dessus, alors nous sommes plutôt satisfaits. Nous hériterons de grandes parties de la description de la bibliothèque et nous n'aurons besoin de spécifier que les fonctionnalités supplémentaires (voir la Figure I.4-6). Le problème potentiel réside dans le fait que même si la relation de spécialisation paraît convenir au premier abord, il se peut qu'elle ne convienne pas complètement après une investigation plus poussée.

Les unités de bibliothèque, qui n'ont pas été conçues avec l'idée de définir une banque, peuvent posséder quelques structures internes qui ne sont pas exactement ce que nous souhaitons pour une banque. Elles peuvent ne pas être «virtuelles» et alors elles ne peuvent pas être redéfinies. Si c'est le cas, nous devons conclure que même si les concepts généraux paraissaient être liés par une relation directe de spécialisation, ils ne l'étaient finalement pas.

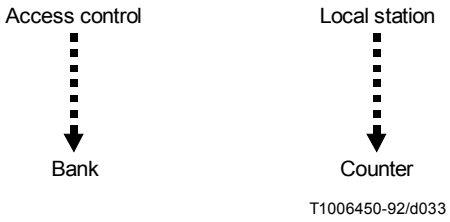


FIGURE I.4-6/Z.100

Relation de spécialisation directe entre application et bibliothèque

Le deuxième cas est que nous savons que *AccessControl* et *Bank* sont semblables, mais qu'ils ne sont pas liés par une relation de spécialisation directe. Alors ils sont tous les deux des spécialisations d'un concept commun plus général (voir la Figure I.4-7).

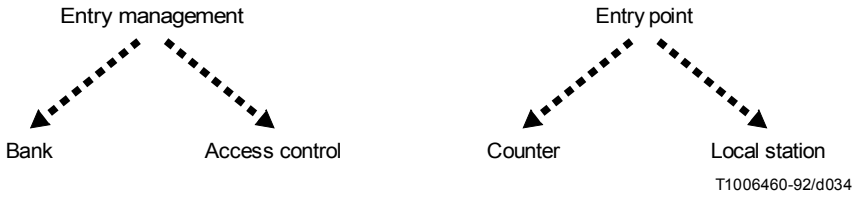


FIGURE I.4-7/Z.100

Relation de spécialisation indirecte entre application et bibliothèque

L'analyse devient maintenant quelque chose de plus compréhensible. La bibliothèque devrait être structurée en introduisant les nouveaux concepts plus généraux *Entry Management* et *Entry Point* et les anciennes unités de bibliothèque *AccessControl* et *LocalStation* devront devenir des spécialisations directes des nouvelles unités de bibliothèque.

Nous avons maintenant terminé deux choses. La première est que la bibliothèque est devenue plus générale et aussi plus facilement applicable à de nouvelles applications autres que *AccessControl* et *Bank*. La deuxième chose est que nous avons créé des unités de bibliothèque à partir desquelles les concepts de *Bank* sont des spécialisations directes.

De plus, nous n'avons rien perdu puisque *AccessControl* est en effet identique à ce qu'il était avant la restructuration. Nous avons assuré la compatibilité à la fois en amont et en aval.

Cette restructuration de la bibliothèque engendre quelques effets. Nous noterons que plus les concepts sont généraux, plus ils incluent de types virtuels. (Ceci bien sûr décroît la puissance d'analyse, mais pas nécessairement la puissance de description). En rendant *AccessControl* identique à ce qu'il était auparavant (de telle sorte que l'analyse antérieure est encore valable pour ses spécialisations), nous devons faire attention pour «mettre au point» les types virtuels hérités des nouveaux concepts généraux qui n'étaient pas virtuels dans la version originale de *AccessControl*.

Un autre effet très caractéristique et très intéressant de la recherche en bibliothèque est que lorsque l'on réalise la similitude entre *AccessControl* et *Bank*, nous pouvons trouver les caractéristiques de *AccessControl* qui peuvent s'appliquer à *Bank*, mais qui n'ont pas été conçues dans le contexte de *Bank*. Un exemple possible de cet effet est la *LoggingStation* dans *AccessControl*. Une station de «logging» pour les transactions bancaires est peut être une bonne idée!

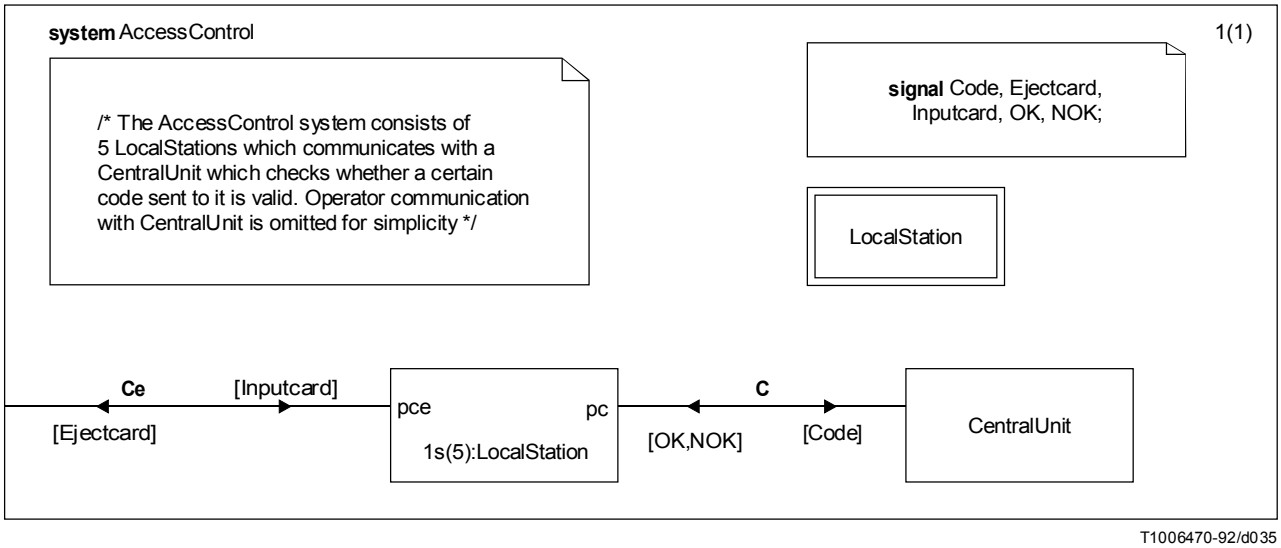
I.4.2 Application des concepts «orienté objet» du SDL

Le paragraphe précédent décrit un langage indépendant de l'analyse orientée objet. Ce paragraphe précise les **étapes 4** et **5** en appliquant les concepts orientés objet du SDL.

Etape 6 – Généralisations

- Rendre les composants «adaptables» en introduisant les *types virtuels*. S'assurer que de tels types disposent des noms généraux convenables. Equilibrer l'adaptabilité des types virtuels en utilisant la clause **atleast** pour contraindre les possibilités de redéfinitions.
- Assurer l'indépendance des signaux et des sortes en introduisant des *paramètres de contexte*. Equilibrer cette indépendance en contraignant les paramètres de contexte.

Notre point de départ est un système de contrôle d'accès (*AccessControl*) avec quelques types vraiment généraux (voir l'exemple I.4-1).



EXEMPLE I.4-1

Le diagramme du système *AccessControl*

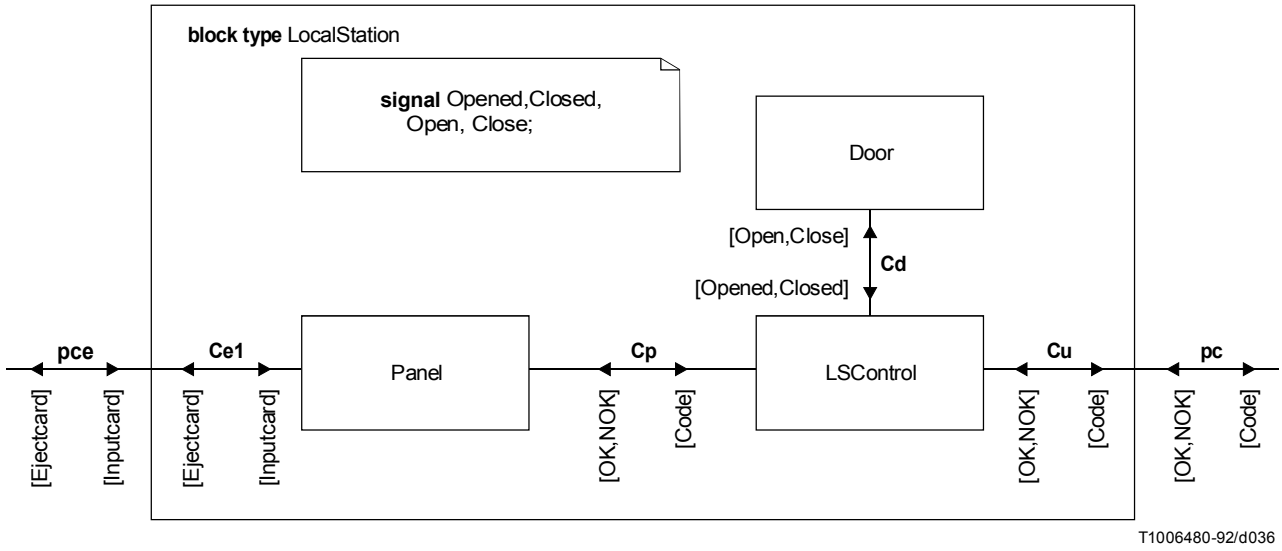
Un des types centraux est le type de bloc *LocalStation* qui (dans une version) est défini comme dans l'exemple I.4-2.

Même si nous sommes capables de voir qu'un guichet (*Counter*) peut être modélisé à peu près de la même manière, les noms des composants ne conviennent pas et la définition n'est pas assez souple pour notre nouvel objectif.

Nous devrons généraliser pour obtenir une définition de type plus générale qui peut être spécialisée à la fois en *LocalStation* (de *AccessControl*) et en *Counter* (de *Bank*). Nous avons besoin ici de souplesse plus que de facilités d'analyse.

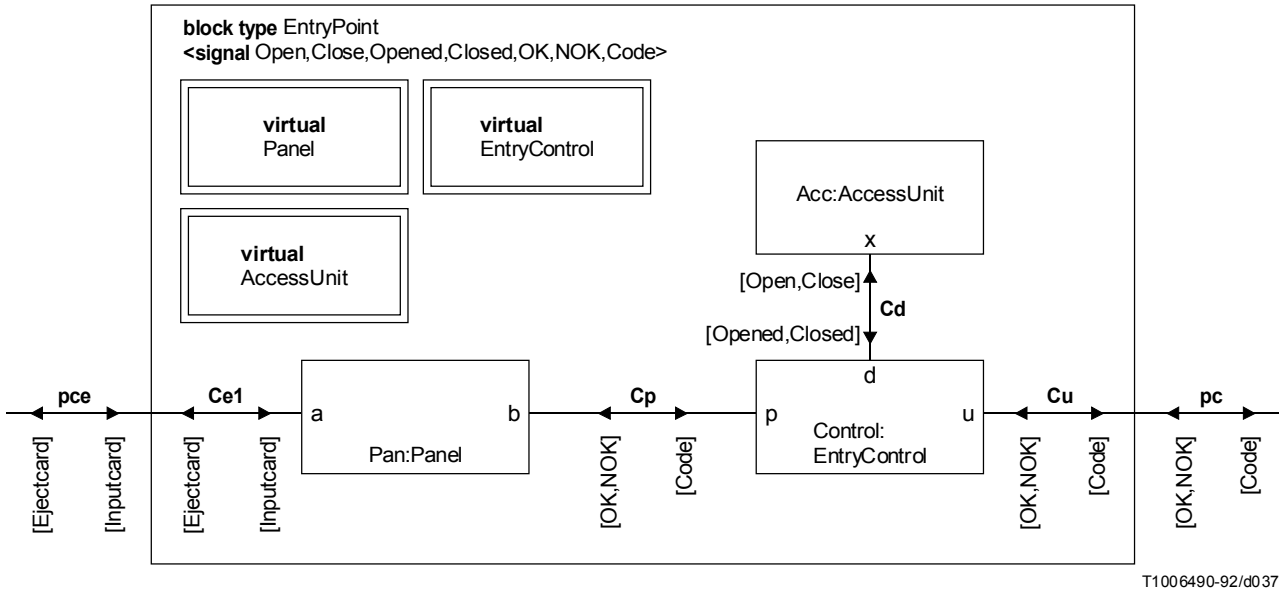
Il y a deux dimensions sur ce que nous souhaitons généraliser. D'abord, nous voulons rendre les composants adaptables c'est-à-dire redéfinissables par des spécialisations. Ceci est rendu possible par des types virtuels. Deuxièmement, nous devons supposer que les signaux d'un système bancaire (*Bank*) sont différents de ceux d'un système de contrôle d'accès

(*AccessControl*). Ainsi, nous voulons réaliser la généralisation au-dessus des noms de signaux. Ceci est obtenu en utilisant des paramètres de signaux contextuels (voir l'exemple I.4-3). Noter que le type de bloc *EntryPoint* est paramétré et qu'il ne permet pas de produire directement les instances.



EXEMPLE I.4-2

LocalStation* à partir de la version (ancienne) de *AccessControl



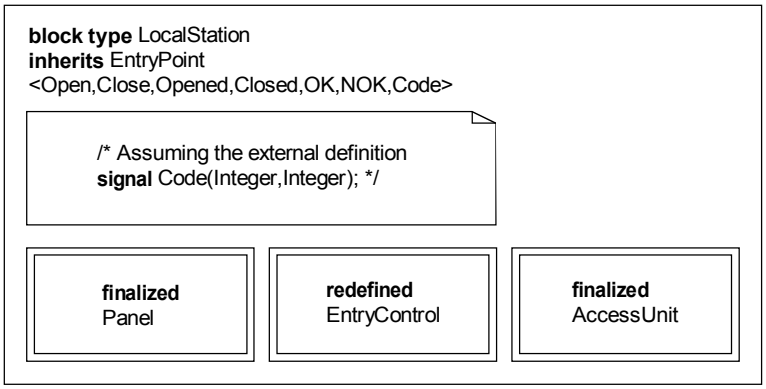
EXEMPLE I.4-3

Le type de bloc général *EntryPoint*

Etape 7 – Spécialisation

– En spécialisant, prendre garde de maintenir l'équilibre entre les possibilités d'adaptation et les possibilités d'analyse de façon convenable en utilisant les clauses **atleast** et **finalized** pour contraindre les types virtuels.

En utilisant le type de bloc *EntryPoint*, *LocalStation* peut être définie comme une spécialisation, comme le montre l'exemple I.4-4.

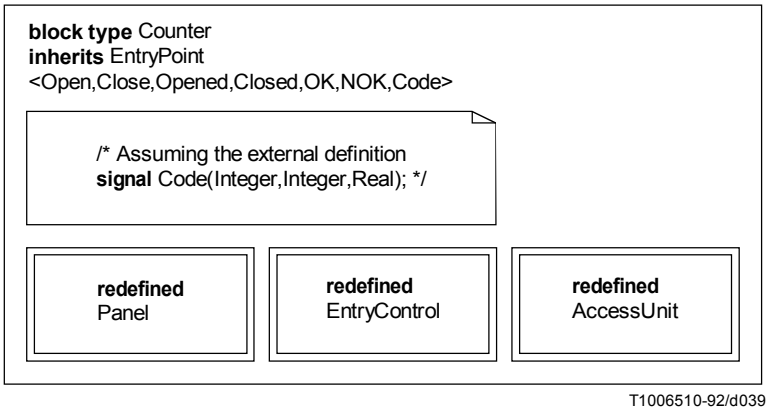


EXEMPLE I.4-4

Le (nouveau) type de bloc *LocalStation*

Nous remarquons que *Panel* et *AccessUnit* possèdent la clause **finalized** de façon à limiter la souplesse et augmenter les possibilités d'analyse. De plus, ceci est réalisé dans ce cas pour rendre la nouvelle version de *LocalStation* fonctionnellement compatible avec l'ancienne. Nous n'avons pas ici montré le niveau système mais indiqué qu'il y a, au niveau du système, une définition de signal des paramètres réels et que la définition du signal *Code* possédant deux paramètres entiers est spécialement intéressante.

Le guichet de banque (*Counter*) peut être défini de façon analogue, comme le montre l'exemple I.4-5.

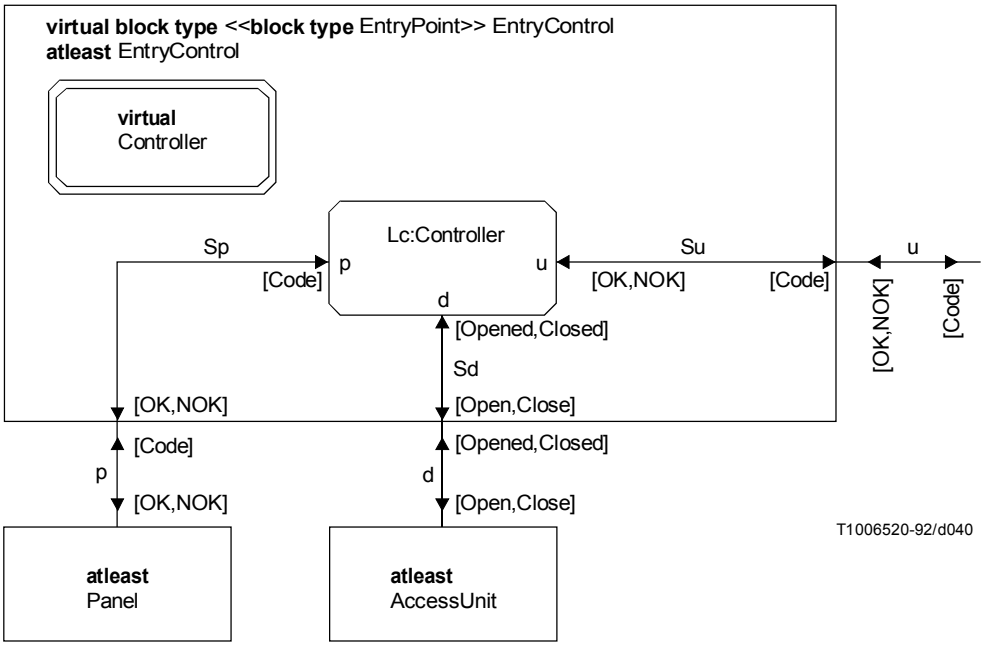


EXEMPLE I.4-5

Le type de bloc *Counter*

Noter que le système bancaire *Bank* est supposé posséder un signal *Code* qui comprend trois paramètres (contrairement à celui du système de contrôle d'accès *AccessControl* qui n'en possède que deux).

Regardons maintenant la définition du type de bloc *EntryControl*, présenté à l'exemple I.4-6.

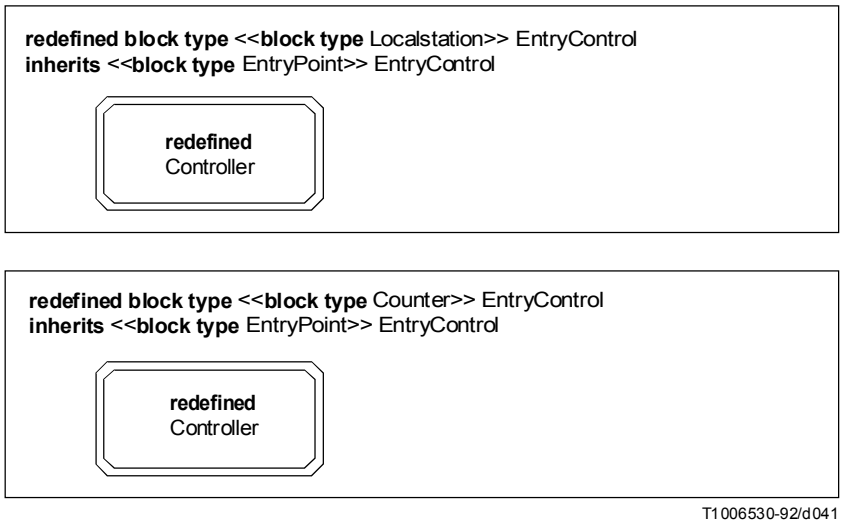


EXEMPLE I.4-6

Le type de bloc *EntryControl*

Nous avons ici la définition par défaut du type virtuel et nous trouvons qu'il spécifie des restrictions en utilisant la clause **atleast.** La restriction porte sur le fait que toutes les redéfinitions doivent être des spécialisations de celle-ci. Ce type de bloc lui-même possède un type de processus virtuel.

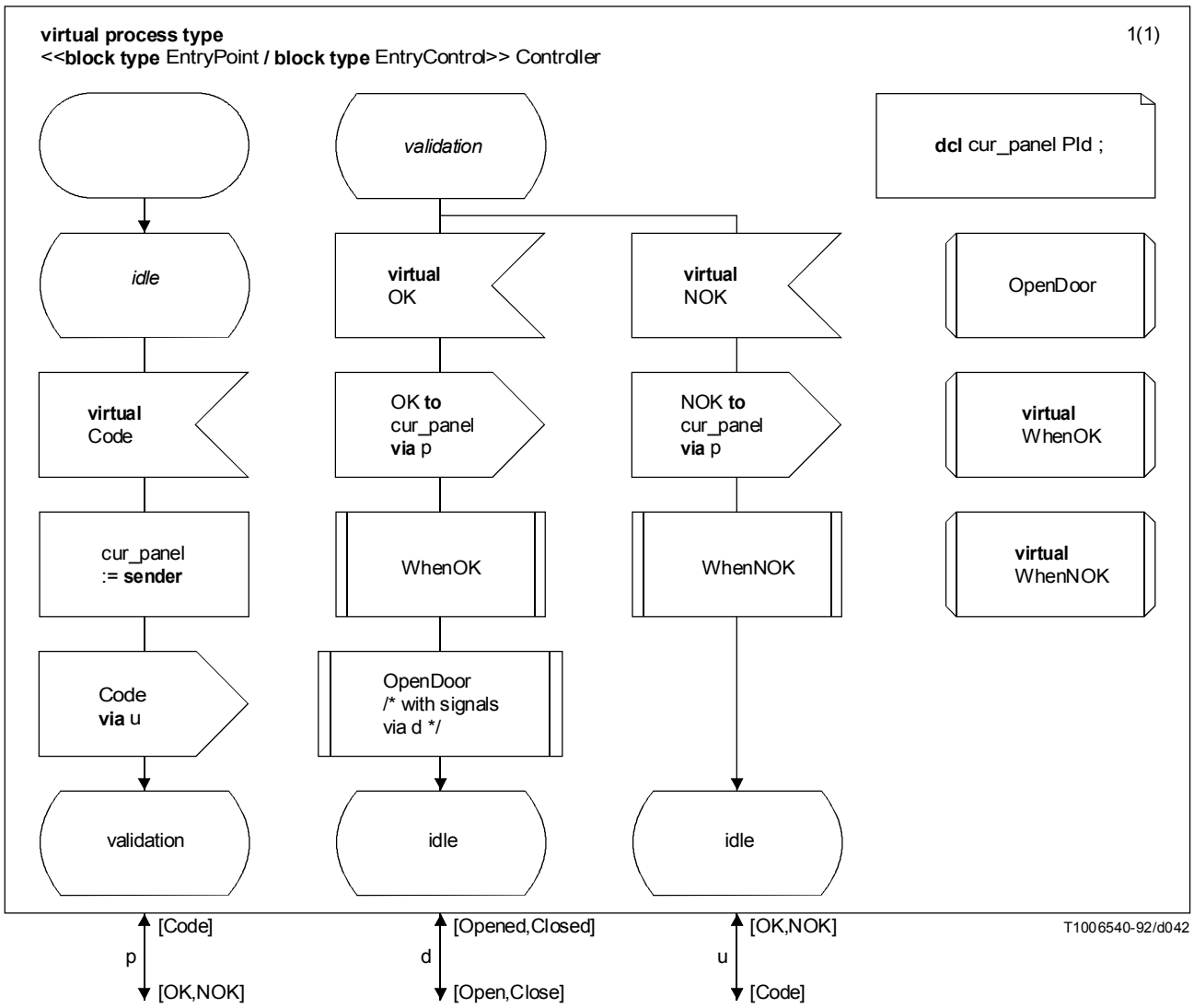
Les redéfinitions correspondantes des systèmes *AccessControl* et *Bank* sont évidemment très simples, comme le montre l'exemple I.4-7. Noter les longues qualifications nécessaires à l'identification des diagrammes. Ceci s'explique par le fait que l'utilisation de types virtuels accroît le nombre de collisions de noms. Les utilisateurs du SDL n'auront pas à considérer ce problème dans la mesure où il sera sûrement pris en charge par les outils.



EXEMPLE I.4-7

Redéfinitions de *EntryControl*

Considérons maintenant la définition du type de processus *Controller*, présenté à l'exemple I.4-8.



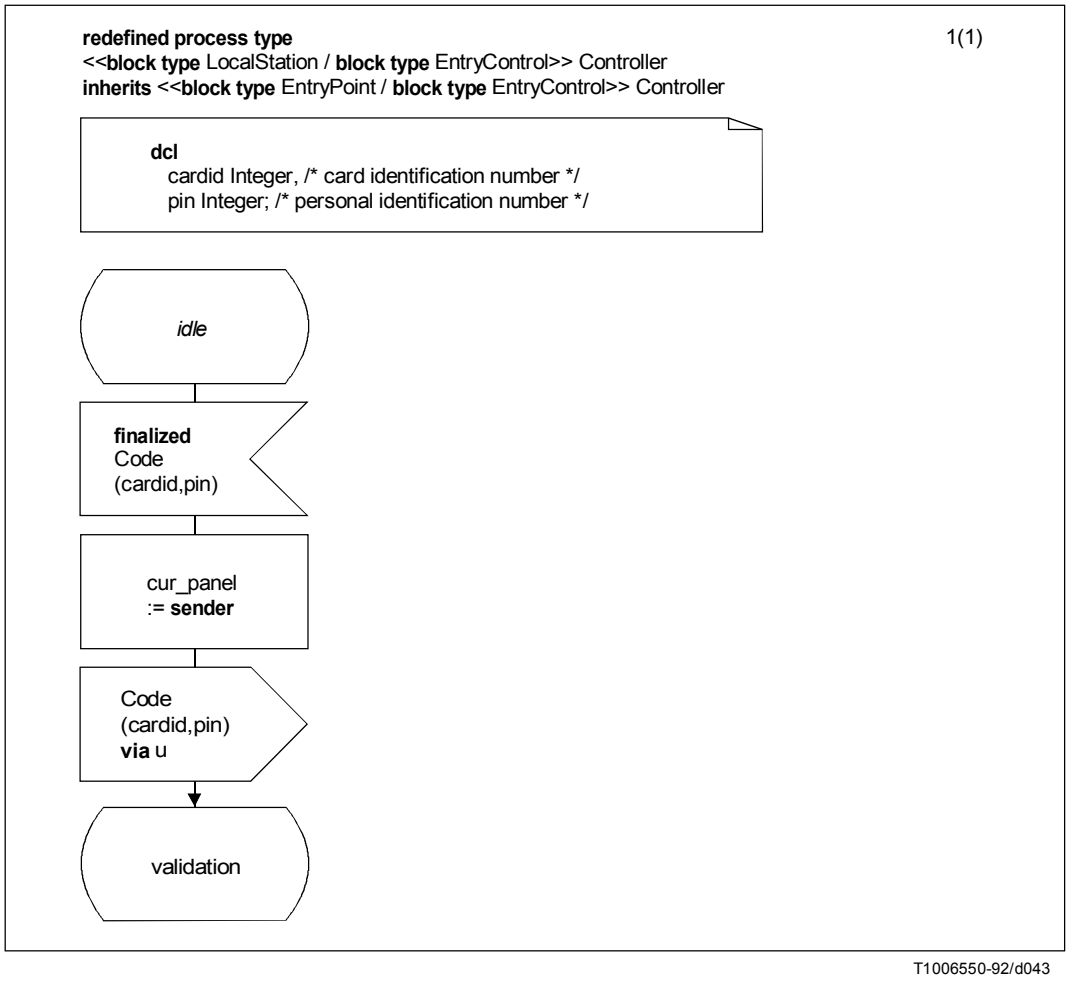
EXEMPLE I.4-8

Le type de processus *Controller*

Nous constatons de nouveau que les parties les plus internes de la définition est «assouplie» en les déclarant **virtuels**. Ici nous avons rendu toutes les transitions virtuelles. La façon dont celles-ci sont redéfinies en spécialisations est présentée dans les exemples I.4-9 et I.4-10.

La redéfinition dans l'exemple I.4-9 concerne la réception du signal *Code*. La transition utilise les deux paramètres du signal: *cardid* et *pin* (la définition du signal peut être la même puisqu'on autorise d'avoir moins de paramètres dans une entrée qu'il n'est spécifié dans la définition de signal, les informations supplémentaires transportées par l'instance de signal n'étant pas prises en compte).

Cette redéfinition dans l'exemple I.4-10 concerne de nouveau la réception des signaux: *Code* et *OK*. Les transitions utilisant de nouveaux paramètres de signaux: *cardid*, *pin*, *amount* et *leftonaccount*.



EXEMPLE I.4-9

Redéfinition de *Controller* dans *AccessControl*

I.5 Production progressive d'une spécification complète de types abstraits de données

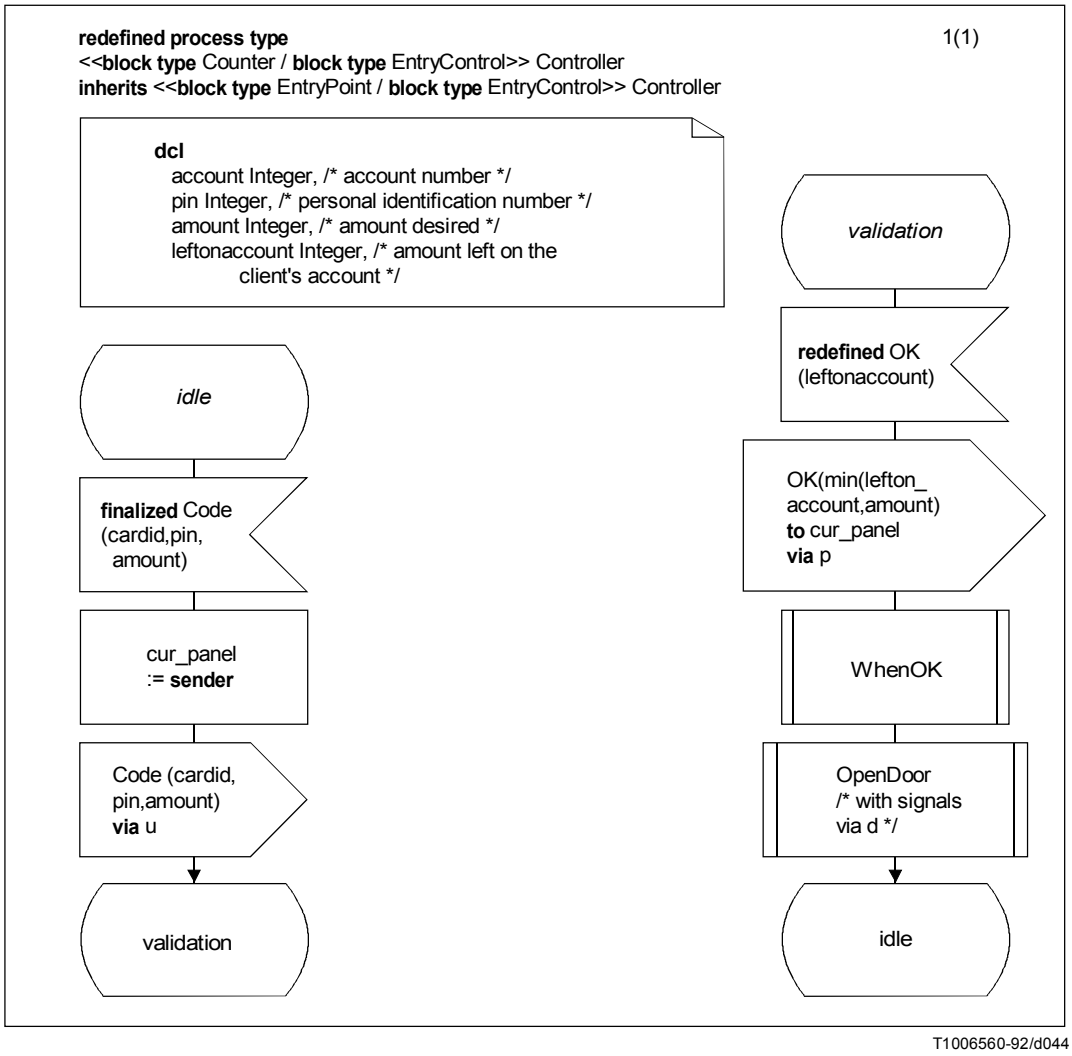
On prétend que les sortes prédéfinies dans la présente Recommandation sont complètes. Pourquoi ces définitions sont-elles complètes? Afin de répondre à cette question, il faut déterminer d'abord ce qu'est la complétude. Dans ce paragraphe, on présentera et on expliquera la méthode appelée méthode fonction-constructeur (CFM) (*constructor function method*). Cette méthode CFM fournit un support pour écrire des spécifications complètes de types abstraits de données (ADT) (*abstract data types*).

Dans la suite, on suppose que le lecteur est familier avec les notions mathématiques de types abstraits de données (les signatures engendrent des termes séparés en classes d'équivalence par les équations; les classes d'équivalence définissant les valeurs de la sorte).

I.5.1 Complétude d'une spécification complète de types abstraits de données

Une définition absolue de la complétude *ne* peut *pas* être donnée. Une sorte est spécifiée complètement lorsque les valeurs de la sorte correspondent aux valeurs que le spécifieur avait l'intention de définir. Comme l'intention d'une personne est dure à mesurer, la complétude d'une spécification de sorte ne peut être définie formellement[1)].

1) Dans la théorie des systèmes de réécriture, «complet» et «complétude» ont un sens formel. Dans ce paragraphe cependant, «complet» est utilisé avec le sens non formel.



EXEMPLE I.4-10

Redéfinition de *Controller* dans *Bank*

Nous pouvons faire quelque chose avec une définition relative (relative à l'intention de quelqu'un) de la complétude. Pour commencer, on doit s'assurer que deux termes qui sont supposés représenter des valeurs différentes ne soient pas placés dans la même classe d'équivalence. Deuxièmement, il faut s'assurer que tous les termes sont placés dans les classes d'équivalence (= valeurs) supposées, autrement ces termes deviendraient des valeurs nouvelles non désirées.

L'intention de la spécification de la sorte *Boolean* est de créer deux valeurs dans la sorte *Boolean*, de telle manière que les deux littéraux appartiennent à deux classes d'équivalence différentes. Ceci signifie que dans les équations on devrait s'assurer que *true* et *false* ne soient pas placés dans la même classe d'équivalence, et qu'il ne soit créé aucune autre classe d'équivalence, en plus des classes d'équivalence auxquelles appartiennent *true* et *false*. Les équations sont écrites de telle sorte qu'une expression booléenne quelconque puisse se réduire à partir de l'intérieur.

Dans toute expression, la sous-expression la plus interne, qui n'est pas un littéral, peut grâce à l'application d'une ou plusieurs équations, se réduire à un littéral. L'application répétée de cette procédure réduit toute expression *booléenne*, sans considération de complexité, à un littéral. Ainsi, on peut conclure qu'il y a au plus deux valeurs.

Il n'est, en principe, pas possible de montrer que *true* et *false* ne sont pas dans la même classe d'équivalence. Cependant, les fonctions définies dans la sorte *Boolean* sont des fonctions mathématiques très connues qui se comportent «bien».

Il faut noter que même avec des fonctions mathématiques ayant un «bon» comportement, il faut faire attention. Il existe une façon de définir tous les nombres naturels (sorte *Nat*) avec le littéral *0* et l'opérateur successeur *succ*[2]. Pour exprimer l'égalité de deux nombres naturels avec l'opérateur *eq*, les expressions sont «pelées»:

```
succ( x ) = succ( y )   ==   x = y ;
       succ( x ) = 0   ==   false ;
              0 = 0    ==   true ;
              x = y    ==   y = x ;
```

Jusqu'ici aucun problème. Supposons maintenant qu'un opérateur *fac: Nat –> Nat* ait été défini avec les équations:

```
   fac( 0 ) == succ( 0 ) ;
fac( succ(n) ) == succ( n ) * fac( n ) ;
```

où * est l'opérateur mathématique de multiplication et qu'il faille déterminer si le terme *fac(6) = fac(7)* appartient à la même classe d'équivalence que *true*. Ceci exige un nombre important d'applications d'équations (d'abord un nombre important de multiplications exigeant encore davantage d'additions suivies par la *réduction* de *succ fac(6)* fois selon la première équation ci-dessus). Un spécifieur averti ajoute donc l'équation

```
fac( x ) = fac( y )  ==  x = y ;
```

Les conséquences sont sévères: *true* et *false* représentent la même valeur. La preuve s'établit de la manière suivante:

```
fac( 0 ) = fac(succ( 0 )) == succ( 0 ) = succ( 0 ) == true
```

mais d'un autre côté

```
fac(0) = fac(succ(0)) == 0 = succ(0) == false
```

et il s'ensuit que

```
true == false
```

I.5.2 La méthode de base utilisant fonction et constructeur

Dans ce paragraphe-section on présente la méthode constructeur fonction (CFM). Cette méthode se base sur une «déclaration d'intention» de la part du spécifieur à l'égard des valeurs des sortes. Dans cette méthode, les opérateurs et les littéraux sont divisés entre «constructeurs» et «fonctions» et c'est de cette division que la méthode tire son nom. La terminologie «fonction» et «constructeur» n'a aucune relation avec une quelconque construction SDL; ces termes n'ont de sens que dans cette méthode.

Dans les deux premiers paragraphes, on donne les définitions de «constructeur» et de «fonction». Dans le troisième paragraphe on présente les quatre premières étapes de la méthode CFM.

I.5.2.1 Constructeurs

Lorsqu'on étudie les spécifications de types abstraits de données, on note qu'en général il y a un petit nombre d'opérateurs[3] qui *peut* engendrer ensemble au moins un terme dans *chaque* classe d'équivalence *prévue*. Les opérateurs qui «déterminent» une sorte sont appelés les «constructeurs» de cette sorte.

Exemple

Sort	Constructors
Boolean	true, false
Integer	0, 1, plus, neg
Tree	empty, leaf, node
IntSet	EmptyIntSet, Insert

La première phrase de cette sous-section comporte trois mots essentiels:

chaque

L'opérateur *neg* pour *Integer* ne devrait pas être omis; sans lui, il n'y aurait aucun terme ne comportant que des constructeurs (terme constructeur) dans la classe d'équivalence du terme *minus(0,1)*.

---

2) Ce n'est pas la façon utilisée pour les sortes prédéfinies.

3) Dans la suite, les littéraux sont considérés comme des opérateurs sans arguments.

prévue

La subjectivité subsiste mais une fois que les constructeurs ont été déterminés, il y a une «déclaration d'intention» qui dit que tous les autres opérateurs n'engendrent pas de valeurs qui ne peuvent être décrites par un terme constructeur.

peut

En général, l'ensemble des constructeurs n'est pas unique. Pour la sorte *Boolean* on pourrait avoir choisi comme constructeurs { *true, Not* } ou {*false, Not* }.

Il n'est pas possible de donner plus de directives pour le choix de l'ensemble des constructeurs. En général, on préférera un petit nombre de constructeurs à un grand nombre. Le nombre de constructeurs devrait au moins être fini. Habituellement, quelques expériences sont nécessaires pour trouver l'ensemble des constructeurs à partir duquel la méthode engendre des équations «intuitivement intéressantes» (les experts en ADT sont des personnes qui possèdent quelque expérience pour choisir un ensemble de constructeurs qui fournit de bonnes équations, le reste est la plupart du temps de l'intimidation).

Il ne semble pas que chaque valeur d'une sorte se décrive de façon unique avec quelque terme constructeur, bien que cela simplifie l'utilisation de la méthode. Pour illustrer le fait, il est probable que l'on veuille que les deux termes

```
Insert( 1, Insert( 0, EmptyIntSet ) )    et
Insert( 0, Insert( 1, EmptyIntSet ) )
```

représentent le même ensemble d'entiers (c'est-à-dire *{ 0, 1}*)

I.5.2.2 Fonctions

La définition des fonctions est simple: chaque opérateur qui n'est pas un constructeur est une fonction. Ne pas être induit en erreur par le nom: dans la méthode CFM, un littéral (et par conséquent une constante) peut être une fonction!

Pendant l'utilisation de la méthode CFM, il est important pour toutes les sortes de ne pas changer les ensembles de constructeurs et de fonctions. Par exemple, il n'est pas recommandé d'utiliser les constructeurs {*true, false*} dans la spécification de la sorte *Boolean*, tandis que dans la spécification de certaines autres sortes on considérera les constructeurs {*true, Not*}.

I.5.2.3 La méthode

Dans I.5.2.1, on a indiqué que le choix des constructeurs est une déclaration d'intention: une fonction (ou une expression avec plusieurs fonctions) ne devrait pas engendrer les valeurs qui ne peuvent pas être décrites avec un terme constructeur. Cette déclaration d'intention va être exploitée.

On appliquera les étapes suivantes pour chaque fonction, une fonction à la fois.

Etape 1

– Pour chaque argument d'une fonction, déterminer toutes les formes possibles que peut prendre cet argument seulement en utilisant des constructeurs avec seulement des variables comme argument(s).

Noter que, au sens strict, «identificateur de valeur» devrait être utilisé à la place de «variable», car la valeur de la variable ne change pas.

Exemple – (A référencer dans les autres étapes) Considérer qu'il existe une fonction

```
X: Intset, Tree -> IntSet
```

(on pourrait penser à l'ensemble des entiers (résultat) qui apparaît à la fois dans l'ensemble des entiers (argument) et dans le nœud-feuille de l'arbre). Le premier argument de *X* peut avoir l'une des formes suivantes:

```
EmptyIntSet,
Insert( j, s )    avec j de sorte Integer et s de la sorte IntSet.
```

Le deuxième argument peut prendre l'une des formes suivantes:

```
empty,
leaf( j )    avec j de la sorte Integer,
node( b1, b2 )    avec b1 et b2 de la sorte Tree.
```

L'expression *Insert(j1, EmptyIntSet)* n'appartient pas à la liste des formes pour le premier argument parce que le constructeur *Insert* a un constructeur comme argument (c'est-à-dire *EmptyIntSet*). Pour la même raison, *node(b3, leaf(0))* ne peut être une forme valable pour le deuxième argument.

Etape 2

– Produire une liste d'applications de la fonction avec pour arguments toutes les combinaisons possibles des formes. Les variables dans différents arguments devraient avoir des noms différents (les renommer si nécessaire) et toutes les variables utilisées devraient être placées dans une spécification.

Exemple – Pour la fonction *X* de l'exemple ci-dessus, l'**étape 2** produit la liste quantifiée suivante d'applications.

```
for all  j, j1, j2 in Integer,
         s  in IntSet,
         b1, b2 in Tree
    X( EmptyIntSet, empty )
    X( EmptyIntSet, leaf( j ) )
    X( EmptyIntSet, node( b1, b2 ) )
    X( Insert( j, s ), empty )
    X( Insert( j1, s ), leaf( j2 ) )
    X( Insert( j, s ), node( b1, b2 ) )
```

Noter que dans l'avant-dernière application *j* a été renommé.

Dans une variante de la méthode CFM, l'**étape 2** est remplacée par:

– Produire une application en donnant des variables à tous les arguments d'une fonction.

Ensuite, l'**étape 3** et l'**étape 4** (voir ci-dessous) restent inchangées, mais en particulier l'**étape 4b** sera appliquée plus souvent. Le choix entre la méthode CFM telle qu'elle est présentée ici et cette variante est affaire de goût.

Etape 3

– Prendre chaque application dans le résultat de l'**étape 2** comme la partie gauche d'une équation et écrire la partie droite en utilisant seulement:

1) des constructeurs;
2) des variables provenant de la partie gauche;
3) des fonctions complètement définies; et
4) la fonction elle-même, à condition que les expressions pour les arguments soient plus petites ou égales à celles de la partie gauche, et qu'au moins une des expressions soit strictement plus petite.

– Si cela ne réussit pas aller à l'**étape 4**.

Ce que signifie *plus petit ou égal* ou *strictement plus petit* n'est pas facile à expliquer. En général, cela signifie une expression obtenue par élimination de certains opérateurs, mais ce n'est pas toujours le cas. Le point essentiel est d'éviter les définitions circulaires.

Exemple – Si l'interprétation de *X* est telle que celle suggérée à l'**étape 1** [l'ensemble des entiers (résultat) qui apparaît à la fois dans l'ensemble des entiers (argument) et dans le nœud-feuille de l'arbre], alors les équations suivantes peuvent être données. La quantification est omise pour des raisons de concision.

```
X( EmptyIntSet, empty )             ==  EmptyIntSet ;
X( EmptyIntSet, leaf( j ) )         ==  EmptyIntSet ;
X( EmptyIntSet, node( b1, b2 ) )    ==  EmptyIntSet ;
X( Insert( j, s ), empty )          ==  EmptyIntSet ;
X( Insert( j1, s ), leaf( j2 ) )    ==  ? ;
X( Insert( j, s ), node( b1, b2 ) ) ==  Union(X(Insert( j, s ), b1),
                                        X(Insert( j, s ), b2 ) ) ;
```

Les quatre premières équations ne posent pas de problème. Dans la cinquième, on doit distinguer entre les cas où *j1* et *j2* sont égaux et non égaux, et par conséquent l'**étape 4** est attendue pour cela.

La sixième équation utilise la fonction *Union* complètement définie auparavant et l'appel récursif de *X*. Ces appels récursifs sont autorisés parce que le premier argument de *X* reste «égal» et que le deuxième argument est strictement plus petit (également à l'égard de la partie gauche).

Etape 4

– Il y a deux raisons possibles pour que la seconde moitié d'une équation ne puisse pas être fournie à l'**étape 3**:

1) la partie droite dépend d'une relation entre variables. Dans ce cas, aller à l'**étape 4a.**
2) la partie droite dépend de la structure d'une ou plusieurs variables. Dans ce cas, aller à l'**étape 4b**.

Exemple – Dans le cas de la cinquième équation pour *X*, la partie droite dépend de la relation entre les variables *j1* et *j2*.

Etape 4a

- Dans cette étape, une équation est décomposée en plusieurs équations conditionnelles en mettant la relation requise entre les variables dans une condition. La partie gauche de l'équation obtenue à l'**étape 2** n'est pas changée.
- Ecrire les relations entre les variables comme des équations (non conditionnelles) et/ou des expressions booléennes en utilisant seulement:
  1) des constructeurs,
  2) des variables,
  3) des fonctions complètement définies,
  4) la fonction elle-même, à condition que les expressions pour les arguments soient plus petites ou égales à celles de la partie gauche et qu'au moins une des expressions soit strictement plus petite.
- Vérifier que les conditions pour la même application sont complémentaires ou que, en cas de recouvrement des conditions, la partie droite est égale.

Exemple – Les conditions requises pour fournir les parties droites pour l'application

```
X( Insert( j1, s ), leaf( j2 ) )
```

sont

```
j1 = j2
j1 /= j2
```

Ces conditions sont complémentaires. De plus, les équations suivantes peuvent être écrites.

```
j1 = j2 ==> X( Insert( j1, s ), leaf( j2 ) )    ==  Insert( j1, EmptyIntSet )
j1 /= j2 ==> X( Insert( j1, s ), leaf( j2 ) )   ==  X( s, leaf( j2 ) )
```

La première équation conditionnelle satisfait les équations de l'**étape 3** puisque la partie droite utilise seulement des constructeurs et une variable. La seconde équation conditionnelle satisfait les conditions parce que l'appel récursif possède comme premier argument une expression qui est strictement plus petite et parce que le second argument ne grossit pas.

Etape 4b

- Dans cette étape, une équation est décomposée en plusieurs équations en remplaçant une des variables par plusieurs termes. Déterminer toutes les formes de la variable dont dépend la structure de la partie gauche comme à l'**étape 1**. Remplacer la variable par chacune de ces formes, de sorte que les noms de variables dans ces formes ne coïncident pas avec les noms de variables apparaissant déjà dans le terme de l'application (c'est-à-dire la partie gauche). Ajouter les variables nouvellement introduites dans la quantification et compléter les équations.

Exemples – Supposer qu'il existe un opérateur *Fib : Nat –> Nat* (Fib pour Fibonacci). Pour *Nat*, les constructeurs sont supposés être le littéral *0* et *succ : Nat –> Nat*. L'application des **étapes 1**, **2** et **3** conduit à:

```
for all n  in  Nat
        Fib( 0 )          ==  1 ;
        Fib( succ( n ) )  ==  ? ;
```

Puisque *Fib* est défini par *Fib(0)=Fib(1)=1* et *Fib(n)=Fib(n-1)+Fib(n-2)* pour *n > 1*, la seconde équation pour *Fib* dépend de la structure de *n* et l'**étape 4b** s'applique. On obtient le résultat suivant:

```
for all n  in  Nat
  Fib( succ( 0 ) )          ==  1 ;
  Fib( succ( succ( n ) ) )  ==  plus( Fib( succ( n)), Fib(n ) ) ;
```

La partie droite de la dernière équation satisfait les conditions de l'**étape 3** parce que plus est une fonction complètement définie et que les arguments sont des appels récursifs de *Fib* avec un argument plus petit que l'argument de la partie gauche.

Dans certains cas, l'**étape 4a** et/ou l'**étape 4b** doit être appliquée plusieurs fois.

Les quatre étapes ci-dessus constituent la méthode CFM de base complète. Il faut insister sur le fait que la méthode CFM de base est sûre vis-à-vis de la complétude, mais coûte du temps et du papier. Souvent un groupe d'équations peut être remplacé par une seule équation.

Un second point à noter est que la méthode CFM de base n'engendre pas d'équations où, à la fois, dans la partie gauche et dans la partie droite, l'opérateur le plus englobant est un constructeur. Il arrive que de telles équations soient cependant utiles.

I.5.3 Quatre étapes supplémentaires

Tel qu'il a été dit auparavant, la méthode CFM de base est sûre, mais elle produit plus de texte qu'il n'est strictement nécessaire. Une bonne méthode ne doit pas être seulement sûre, elle doit aussi conduire à un résultat lisible. Les étapes décrites dans ce paragraphe ont pour objectif d'aller dans cette direction.

I.5.3.1 Réduction des équations par les variables supplémentaires

Dans l'exemple courant du I.5.2.3, il apparaît que le premier argument de la fonction *X* est égale à l'ensemble vide et que le second argument n'influence pas la partie droite. Cela signifie que les équations

```
X( EmptyIntSet, empty )           ==  EmptyIntSet ;
X( EmptyIntSet, leaf( j ) )       ==  EmptyIntSet ;
X( EmptyIntSet, node( b1, b2 ) )  ==  EmptyIntSet ;
```

peuvent se résumer à

```
X( EmptyIntSet, b ) ==  EmptyIntSet ;
```

où *b* est une variable de la sorte *Tree*.

Etape 5

– Remplacer l'ensemble des équations possédant une partie droite invariante et des parties gauches ayant un argument variant par une équation dans laquelle une variable se substitue à l'argument soumis à variation.

La fusion de l'**étape 5** et de l'**étape 2** est possible, mais habituellement elle se fait seulement au prix de la sûreté. Si toutes les formes ne peuvent être remplacées par une variable, on peut terminer avec, par exemple:

```
X( EmptyIntSet, leaf( j ) )  ==  Insert( j, EmptyIntSet) ;
X( EmptyIntSet, b )          ==  EmptyIntSet ;
```

Dans ce cas on peut, en substituant *leaf(j)* par *b*, obtenir l'équivalence

```
Insert( j, EmptyIntSet ) ==  EmptyIntSet
```

Ceci peut être interprété comme «Insert est une fonction», ce qui contredit la «déclaration d'intention». Ceci a un sens car l'équivalence ci-dessus signifie qu'il y a seulement une valeur dans *IntSet*.

I.5.3.2 Réduction des équations grâce à la commutativité

S'il existe une fonction *f* qui est commutative dans une paire d'arguments, alors le nombre d'équations peut être réduit.

Etape 6

– S'il existe une fonction *f* pour laquelle l'équation

f(... ,x,... ,y,...) == f(... ,y,... ,x,...) ;

(commutativité) est vérifiée pour certains arguments *x* et *y*, alors

1) ajouter l'équation de commutativité;
2) rechercher les paires d'équations où une partie gauche correspond à une partie gauche de l'équation de commutativité et une autre partie gauche correspond à une partie droite de l'équation de commutativité et supprimer une équation pour chaque paire.

Exemple – Supposer qu'un opérateur *eq: Tree, Tree –> Boolean* doit être défini pour l'égalité entre arbres définie par l'utilisateur. Après l'**étape 2**, il y a les équations suivantes:

```
eq( empty, empty )                  (1)
eq( empty, leaf( j ) )              (2)
eq( empty, node( b1, b2 ) )         (3)
eq( leaf( i ), empty )              (4)
eq( leaf( i ), leaf( j ) )          (5)
eq( leaf( i ), node( b1, b2 ) )     (6)
eq( node( b1, b2 ), empty )         (7)
eq( node( b1, b2 ), leaf( j ) )     (8)
eq( node( b1, b2 ), node( b3, b4 ) )  (9)
```

Après l'**étape 3**, il apparaît que *eq(t1, t2) == eq (t2, t1)* et cette équation est ajoutée à l'**étape 6** tandis que les équations 4, 7 et 8 sont supprimées car elles forment des paires avec les équations 2, 3 et 6 respectivement.

L'**étape 6** peut être combinée avec l'**étape 2**, mais de nouveau au prix d'une certaine sécurité.

I.5.3.3 Réduction des équations par les fonctions supplémentaires

Si nous regardons un peu plus les équations pour la fonction *X* dans *IntSet*, on peut noter que son premier argument apparaît souvent dans la partie droite sans changement. Ceci suggère un changement possible pour une situation où d'abord quelque fonction agit sur le second argument avant que son résultat soit combiné au premier argument. Pour être plus concret: pour *X*, nous recherchons les fonctions *X1* et *X2* telles que

```
X( s, b ) = X1( s, X2( b ) ) ;
```

Dans ce cas, les fonctions *Intersection: IntSet, IntSet –> IntSet* pour *X1* et *Projection: Tree –> IntSet* pour *X2* devraient convenir. En supposant que *Intersection* est spécifiée de toute façon, cette partition réduit le nombre d'équations de sept (pour *X*) à quatre (une pour *X* et trois pour *Projection*).

Etape 7

- S'il existe une fonction *f* où un ou plusieurs arguments restent (la plupart du temps) inchangés dans les parties droites, alors essayer de trouver de nouvelles fonctions *f1, ..., fn, C* telles que

  1) *f1, ..., fn* agissent sur (des sous-ensembles des) arguments changeant de *f*,

  2) *f* peut être exprimé dans *C* appliqué aux arguments de *f* ne changeant pas et aux résultats de *f1, ..., fn*.

- Les nouvelles fonctions qui sont «artificielles» peuvent être cachées aux utilisateurs des types abstraits de données en utilisant le point d'exclamation comme dernier caractère des noms de fonction.

Exemple – Voir ci-dessus.

I.5.3.4 Combinaison d'équations conditionnelles

Dans de nombreux cas il y a deux équations conditionnelles pour une application et les conditions sont des expressions booléennes et leurs négations. Celles-ci peuvent être combinées en utilisant un terme conditionnel pour la partie droite.

Etape 8

- Si pour certaines applications, des équations conditionnelles sont utilisées et qu'il y a deux équations conditionnelles de la forme

  c ==> *lhs* == *rhs*1 ;

  not(*c* ***)*** ==> *lhs* == *rhs2* ;

alors ces équations conditionnelles sont remplacées par l'équation non conditionnelle

lhs == ***if*** *c* ***then*** *rhs*1 ***else*** *rhs*2 ***fi;***

Le terme conditionnel de la partie droite d'une équation ne doit pas être utilisé en d'autres circonstances que celles décrites à l'**étape 8**. La raison est que l'on a tendance à programmer dans les équations, souvent avec des termes conditionnels imbriqués, ce qui rend les équations difficiles à lire et à comprendre.

I.5.4 Equations pour constructeurs

Avec la méthode CFM de base, aucune équation ne peut être engendrée avec un constructeur comme opérateur le plus englobant (dénommées équations de constructeurs) des deux cotés du signe ==. Cependant, dans certains cas, on a besoin d'exprimer l'équivalence entre des termes constructeurs.

Considérons l'exemple de la sorte *Integer* avec ses constructeurs *0* (littéral), *succ*, *neg: Integer –> Integer*. Ceci prend en compte des termes tels que *succ(neg(...))*. Il est, cependant, possible de placer un terme constructeur dans chaque classe d'équivalence si *neg* est utilisé au plus une fois comme opérateur le plus englobant. Ce fait peut même être utilisé pour simplifier des équations pour les fonctions.

Etape 9

- Pour les constructeurs *C* pour lesquels on souhaite écrire des équations de constructeur, on crée une fonction C^f. On applique la méthode CFM sans cette **étape 9** à ces fonctions. Ensuite on supprime les marques ^{f} et les équations ayant même partie gauche et partie droite.

Le résultat de l'**étape 9** est sûr, mais habituellement des équations superflues (mais sans conséquence) sont engendrées.

L'**étape 9** est présentée comme la dernière étape de la méthode CFM puisque commencer par marquer les constructeurs comme des fonctions peut être plutôt peu clair. Cependant, une fois que le lecteur est un peu familier avec la méthode CFM, il est recommandé de produire les équations pour les constructeurs avant la définition des fonctions. La raison est que les équations pour les constructeurs donnent une bonne perception de la structure des termes constructeurs.

Exemple – Pour *Integer*, les équations de constructeurs pour *neg* et *succ* ont un sens. Une application normale de la méthode CFM conduit à

```
negf( 0 )                 ==  0 ;
negf( succ( n ) )         ==  neg( succ( n ) ) ;
negf( neg( n ) )          ==  n ;
negf( 0 )                 == succ( 0 ) ;
succf( succ( n ) )        ==  succ( succ( n ) ) ;
succf( neg( 0 ) )         ==  succ( 0 ) ;
succf( neg( succ( n ) ) ) ==  neg( n ) ;
succf( neg( neg( n ) ) )  ==  succ( n ) ;
```

Les autres sous-étapes de l'**étape 9** conduisent à

```
neg( 0 )                  ==  0 ;
neg( neg( n ) )           ==  n ;
succ( neg( 0 ) )          ==  succ( 0 ) ;
succ( neg( succ( n ) ) )  ==  neg( n ) ;
succ( neg( neg( n ) ) )   ==  succ( n ) ;
```

La troisième et la cinquième équation de la dernière liste sont superflues puisqu'elles sont impliquées respectivement par la première et la deuxième équation.

I.5.5 Limitations

La présentation de la méthode CFM dans ces directrices débute au point où on décide de construire une nouvelle sorte à partir de rien. La méthode ne donne aucune directive sur le moment où il faut le faire et quand baser une spécification de sorte sur des sortes prédéfinies ou sur le constructeur de sorte *Structure*. Ceci est simplement en dehors de la portée de la méthode CFM.

Dans la portée de la méthode CFM devrait être abordée l'utilisation de, par exemple, **error!** et **nameclass**, mais ce n'est pas actuellement le cas. On notera également comme point important que la méthode CFM présentée se base sur la théorie des systèmes de réécriture réductifs, voir [6]. Ceci est plutôt restrictif. Considérer par exemple la fonction de tri rapide (ou Quick Sort) *Qsort: IntList –> IntList* avec les équations

```
Qsort( EmptyIntList ) == EmptyIntList ;
split( il, i ) == pair( l1, l2 ) ==>
   Qsort( MkString( i ) // il ) == Qsort( l1 ) // MkString(i ) // Qsort( l2 );
```

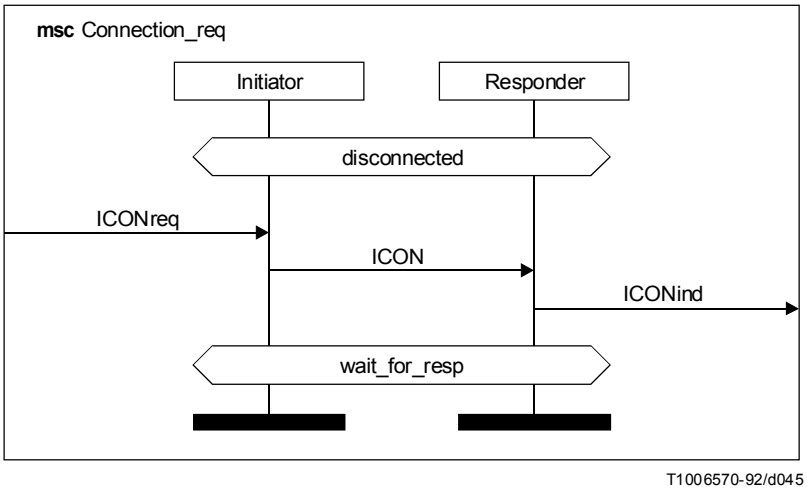
où *pair* est un constructeur de paires de listes d'entiers *IntLists* et *split* sépare son premier argument en une liste d'entiers plus petits que son deuxième argument et en une liste avec des entiers plus grands que son deuxième argument.

Cette définition de *Qsort* pourrait être parfaitement en accord avec la théorie des systèmes de réécriture quasi réductifs mais avec la méthode CFM, cette sorte d'équations «élégantes» ne sera probablement jamais produite.

I.6 Utilisation des diagrammes de séquences de messages

I.6.1 Introduction

Dans le cycle de vie d'un système, il est rentable d'attacher une plus grande attention à la phase de spécification de système puisque la qualité de toutes les phases suivantes en dépend. En particulier, dans le domaine des télécommunications, cet aspect a été pris en compte grâce à l'utilisation du langage SDL. En plus d'une preuve générale de correction (comme l'absence de blocage), la cohérence de la spécification SDL avec le comportement souhaité du système doit être vérifiée. Une façon adéquate de décrire le comportement attendu d'un système est offerte par les traces du système qui sont convenablement présentées sous la forme de diagramme de séquences de messages (MSC) (*message sequence charts*). Les MSC sont des moyens très répandus pour la description et particulièrement la visualisation graphique de traces de systèmes choisies dans les systèmes répartis, spécialement les systèmes de télécommunication. Un diagramme de séquences de messages montre les séquences des messages échangés entre les entités (telles que des services SDL, des processus, des blocs) et leur environnement (voir l'exemple I.6-1). Formellement, un diagramme de séquences de messages décrit un ordre partiel d'événements, comme l'envoi et la consommation de message [5].



EXEMPLE I.6-1

Un diagramme de séquences de messages

Puisque chaque séquence d'un MSC décrit une trace de système, un MSC peut être déduit d'une spécification de système SDL existante. Cependant, un MSC est généralement créé avant la spécification du système et il constitue alors:

- une formulation des besoins pour les spécifications SDL;
- une base pour la génération automatique de squelette de spécifications SDL;
- une base pour la sélection et la spécification de cas de test;
- une spécification semi-formelle de la communication; ou
- une spécification d'interface

Le MSC de l'exemple I.6-1 décrit un «morceau choisi» d'une trace de l'établissement de la connexion dans la spécification du service INRES [12]. Il pourrait aussi bien être représenté en utilisant les diagrammes de processus SDL avec certaines extensions ou modifications, comme le montre la Figure I.6-1, où les branches non traversées sont dessinées en pointillés et où le flux des signaux est indiqué par des flèches en gras.

Le diagramme présenté à la Figure I.6-1 contient au moins les mêmes informations que le MSC de l'exemple I.6-1. Cependant, le MSC est évidemment le plus utile dans ce contexte, puisqu'il met en évidence les informations pertinentes, à savoir les entités (*Initiator*, *Responder*) et les signaux concernés par le «morceau choisi» de la trace (*ICONreq*, *ICON*, *ICONind*). De plus, un MSC peut décrire l'interaction entre des entités appartenant à des niveaux d'abstraction différents.

Une comparaison de l'exemple I.6-1 et de la Figure I.6-1 donne une bonne idée de la signification d'un MSC. Elle montre aussi qu'un MSC décrivant un scénario possible peut aussi être vu comme un squelette de spécification SDL.

Les constructions du langage les plus élémentaires des MSC sont les *instances* et les *messages* échangés entre elles. Dans la forme graphique, les instances et les environnements sont représentés par des lignes verticales ou par des colonnes. Les messages sont représentés par des lignes orientées horizontales avec une inclinaison possible pour admettre le dépassement ou le croisement de message. La flèche du message représente la consommation du signal, l'extrémité opposée (origine du message) représente l'envoi du signal.

Un MSC décrit en général une petite partie d'une trace de système complet. Par conséquent, le MSC doit être caractérisé par la spécification de ses *conditions* initiales et finales et éventuellement par des conditions intermédiaires. Graphiquement, une condition est représentée par un hexagone contenant le nom de la condition (voir la condition initiale *disconnected* dans l'exemple I.6-1). De plus, un MSC peut contenir des *actions* contrôlées par la consommation de signal ou par des *débordements de temporisations* (*timeouts*).

I.6.2 Spécification de système utilisant les mécanismes de composition des MSC

Les MSC sont utilisés principalement comme un langage d'expression de besoins pour décrire l'objet d'un système sous forme d'exemples de traces. Dans la suite, on présente une méthodologie «MSC» systématique basée sur des mécanismes de composition. Les conditions introduites à cette intention peuvent être employées aussi pour la dérivation de squelettes de spécifications SDL.

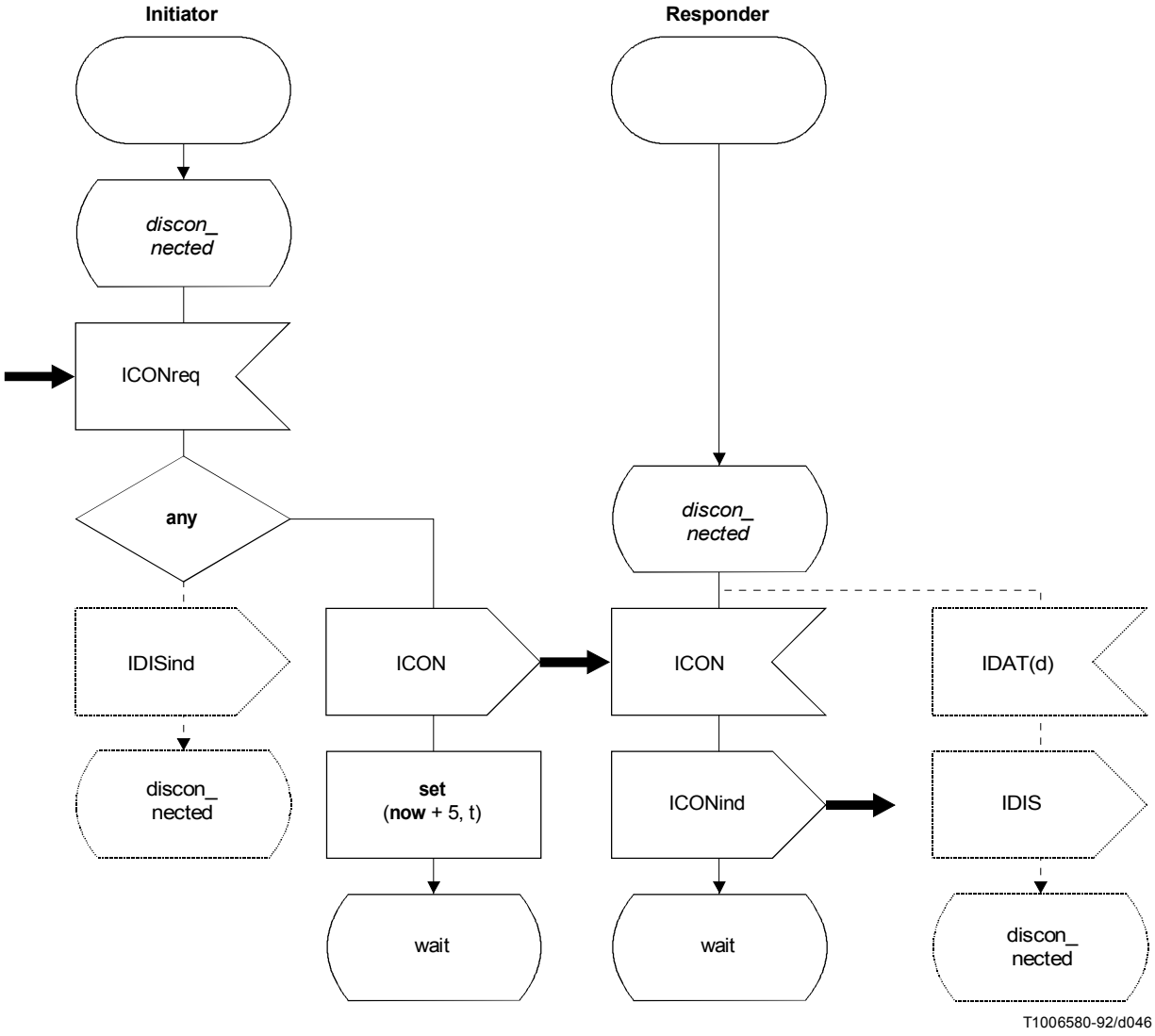


FIGURE I.6-1/Z.100

Diagramme combiné SDL – Flux de signaux (informel)

Puisqu'un MSC décrit seulement un comportement partiel de système, il est intéressant de disposer d'un certain nombre de MSC simples pouvant être combinés de différentes façons. (Ceci peut actuellement être réalisé, ce qu'on appelle ici composition ou qu'on considère seulement comme un ordre d'interprétation). Les MSC peuvent être composés par identification (de nom) des conditions initiale et finale. D'un autre côté, les MSC peuvent être décomposés aux conditions intermédiaires.

La composition et la décomposition de MSC suit les règles suivantes pour les conditions globales et non globales, par ce moyen les conditions globales se réfèrent à toutes les instances concernées dans les MSC, tandis qu'au contraire les conditions non globales sont liées à un sous-ensemble de ces instances.

I.6.2.1 Composition de MSC

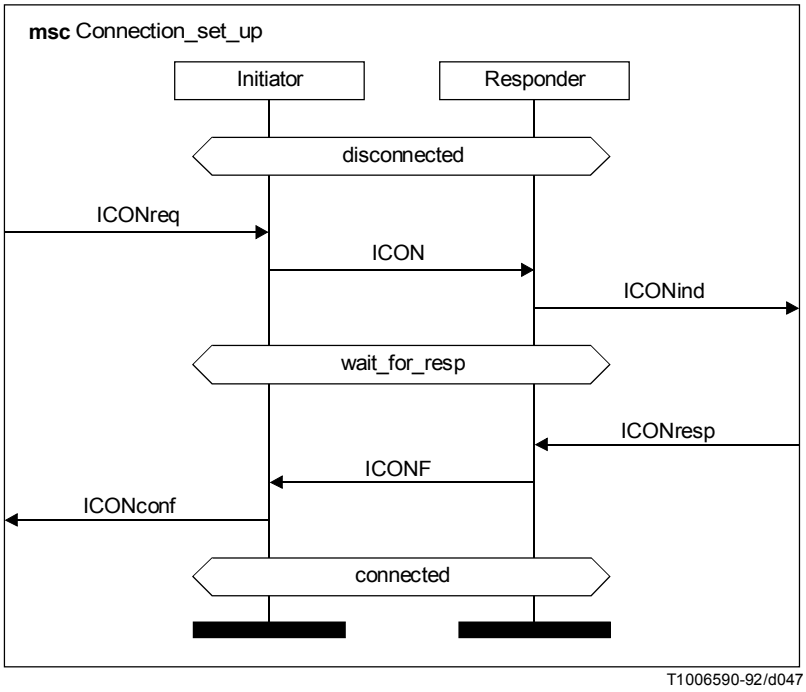
Conditions globales

Deux MSC, *MSC1* et *MSC2* peuvent être composés, si les deux MSC contiennent le même ensemble d'instances et si la condition initiale de *MSC2* correspond à la condition finale de *MSC1* en respectant la même identification des noms. La condition finale de *MSC1* et la condition initiale de *MSC2* deviennent la condition intermédiaire du MSC composé. On écrira symboliquement:

MSC1 = MSC1' Condition
MSC2 = Condition MSC2'
MSC1 * MSC2 = MSC1' Condition MSC2'

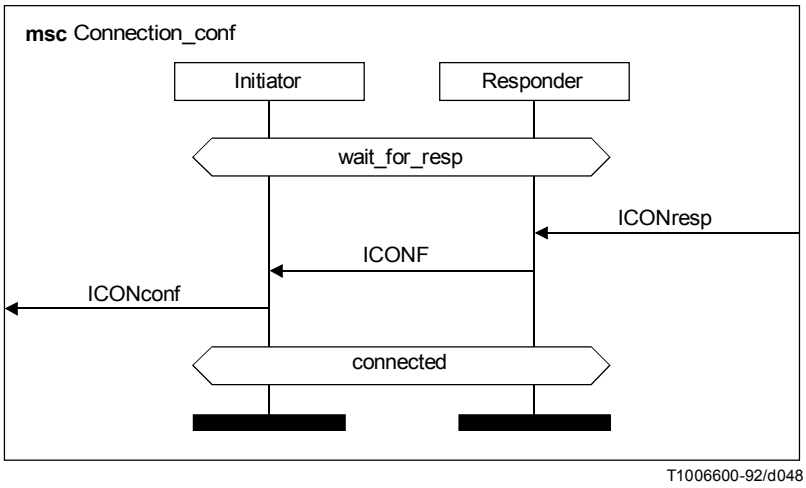
La première équation signifie que *MSC1* peut s'écrire comme une section de MSC *MSC1'* et suivie d'une condition finale. La seconde équation indique que *MSC2* débute par une condition initiale suivie par une section de MSC *MSC2'*. La troisième équation indique la composition de *MSC1* et *MSC2* (en utilisant le symbole astérisque pour représenter la composition). Le MSC composé peut s'écrire sous la forme d'une section de MSC *MSC1'*, suivie d'une condition intermédiaire puis d'une section de MSC *MSC2'*.

Le MSC *Connection_set_up* de l'exemple I.6-2 est une composition des MSC *Connection_req* de l'exemple I.6-1 et *Connection_conf* de l'exemple I.6-3.



EXEMPLE I.6-2

MSC composé basé sur les exemples I.6-1 et I.6-3



EXEMPLE I.6-3

Le MSC Connection_conf

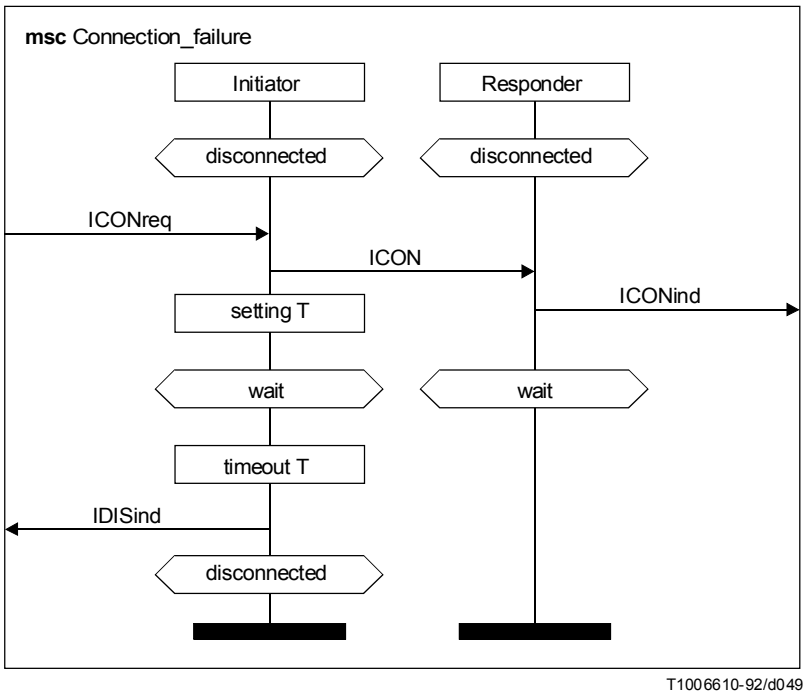
Conditions non globales

Deux MSC *MSC1* et *MSC2* peuvent être composés grâce aux conditions non globales si pour chaque instance (*I*) que les deux MSC ont en commun *MSC1* se termine par une condition non globale et *MSC2* débute par la condition non globale correspondante. De plus, chaque condition non globale de *MSC2* doit avoir une condition non globale correspondante dans *MSC1*. Si *I(MSCi) (i = 1,2)* représente la restriction d'un *MSCi* applicable aux événements de l'instance *I*, ceci peut s'écrire symboliquement:

I(MSC1) = I(MSC1)' Condition
I(MSC2) = Condition I(MSC2)'

I(MSC1) * I(MSC2) = I(MSC1)' Condition I(MSC2)'

Par exemple, le MSC *Connection_failure* (voir l'exemple I.6-4) est une composition des MSC *Connection_request* (voir l'exemple I.6-5a) et *Timeout* (voir l'exemple I.6-5c) via la condition locale *wait*. Le MSC *Connection_request* contient deux instances *Initiator* et *Responder*. A chacune de ces instances est associée une condition locale (finale) *wait* . Il faut noter que les deux conditions locales *wait* avec des noms identiques sont différentes et sont distinguées par les instances auxquelles elles sont attachées. Le MSC *Timeout* ne contient qu'une seule instance, *Initiator*, à laquelle est associée la condition locale *wait*. La composition du MSC *Connection_request* avec le MSC *Timeout* ne réfère qu'à l'instance, *Initiator,* c'est-à-dire que le MSC *Connection_request* se poursuit le long de l'instance *Initiator* par le MSC *Timeout*. Ceci montre aussi l'utilité des conditions non globales, qui rend possible une composition pour un sous-ensemble d'instances concernées dans les MSC. Remarquer également que dans l'exemple I.6-4 l'armement du temporisateur *T* et le débordement de la temporisation sont représentés par des actions, à des fins de compatibilité avec les exemples I.6-5a et I.6-5c.



EXEMPLE I.6-4

Le MSC Connection_failure

I.6.2.2 Décomposition de MSC

Conditions globales

Une condition intermédiaire rend possible la décomposition d'un MSC, *MSC3'* en le découpant à la condition intermédiaire en *MSC1* et *MSC2*, la condition intermédiaire pouvant être convertie en une condition finale pour *MSC1* et en une condition initiale pour MSC2:

MSC3 = MSC1' Condition MSC2'

MSC1 = MSC1' Condition
MSC2 = Condition MSC2'

Conditions non globales

Un sous-ensemble de conditions non globales autorise la décomposition d'un MSC, *MSC3'* en *MSC1* et *MSC2*, si toutes les conditions non globales de ce sous-ensemble réfèrent aux différentes instances et qu'aucun message n'est «coupé» par cette décomposition, c'est-à-dire que l'entrée et la sortie du message appartiennent toutes les deux soit à *MSC1* soit à *MSC2*.

I(MSC3) = I(MSC1)' Condition I(MSC2)'

I(MSC1) = I(MSC1)' Condition
I(MSC2) = Condition I(MSC2)'

Par exemple, le MSC *Connection_failure* de l'exemple I.6-4 peut être décomposé, à la condition locale *wait*, selon les MSC *Connection_request* de l'exemple I.6-5a et *Timeout* de l'exemple I.6-5c .

I.6.2.3 Normalisation de MSC

Un ensemble de MSC développé dans une étape «amont» de conception est en général plutôt non structuré. Il est difficile d'avoir une bonne vue d'ensemble et d'estimer la couverture de toutes les traces possibles du système. Aussi, de grands MSC partagent souvent de nombreuses sous-traces cycliques en commun, qui pourraient être évitées en appliquant une règle de composition adéquate.

Pour surmonter ces imperfections, ce que l'on appellera des *MSC normalisés* peuvent être construits à partir d'un ensemble de MSC, en appliquant les règles de décomposition et de composition définies ci-dessus. Ces MSC normalisés représentent des blocs de construction «normalisés». Ils décrivent le même comportement de système que l'ensemble original de MSC à partir duquel ils sont obtenus, c'est-à-dire que les mêmes traces de système peuvent en être déduites, en prenant en compte les règles de composition. De cette manière, l'ensemble des MSC normalisés dérivés est équivalent à l'ensemble des MSC original.

Les blocs de construction normalisés et la méthode correspondante de composition structurée ont été suggérées déjà pour les réseaux de Petri au moyen de ce qui est appelé des *process periods*.

Les MSC normalisés sont essentiellement des MSC maximaux qui sont insécables, c'est-à-dire qu'ils apparaissent toujours comme un tout sans insertion de cycles. Le terme maximal signifie, dans ce contexte, qu'il n'existe pas de plus grand MSC ayant les propriétés énoncées qui contienne le MSC comme une sous-trace. Les MSC normalisés décrivent soit des traces séquentielles soit des traces cycliques qui ne contiennent pas de sous-traces cycliques apparaissant à partir des états intermédiaires.

Les MSC normalisés sont définis de telle manière qu'il est possible de les engendrer automatiquement à partir d'un ensemble donné de MSC à chaque étape de la spécification d'un système. Ainsi, les MSC normalisés représentent des blocs de construction qui peuvent être utilisés séparément et en parallèle avec l'ensemble des MSC spécifié à l'origine. Il est recommandé de spécifier les MSC dès le tout début d'une manière structurée en suivant les règles des MSC normalisés dans la mesure du possible. En pratique, cependant, une telle approche peut imposer des restrictions non nécessaires et inopportunes pour l'utilisateur, puisque l'idée des MSC normalisés considère le système comme un tout, tandis que, au contraire, dans les étapes en amont sont habituellement spécifiées seulement des parties de systèmes.

La composition de MSC normalisés assistée par ordinateur offre la possibilité d'une simulation immédiate du comportement spécifié. De plus, les MSC normalisés peuvent être utilisés pour la génération et pour la sélection de cas de test. Considérer les MSC normalisés comme les plus petites unités autorise l'abstraction des détails de communication et l'analyse du comportement essentiel du système. Les MSC normalisés sont intéressants aussi pour l'analyse de système en fournissant une autre possibilité au graphe des marquages accessibles, puisqu'ils représentent directement les aspects concurrents d'un système.

La procédure pour normaliser les MSC est donnée dans les étapes schématiques suivantes, en supposant l'existence d'un état initial observable pour le système.

Etape 1 – Ensemble initial de MSC

– Les MSC choisis sont spécifiés pour un système lors de l'étape de définition des besoins. Les relations entre ces MSC sont spécifiées au moyen de conditions qui autorisent les compositions et les décompositions correspondantes.

Pour illustrer ces étapes, nous utilisons la spécification de besoin du service INRES, défini dans [12], qui montre un établissement (simplifié) de connexion suivi d'un transfert de données entre *Initiator* et *Responder*. A cause d'un support de communication non fiable, le transfert de signal peut échouer. Pour des raisons pratiques, l'ensemble des MSC initialement choisi est identique à celui des MSC atomiques définis à l'**étape 2**.

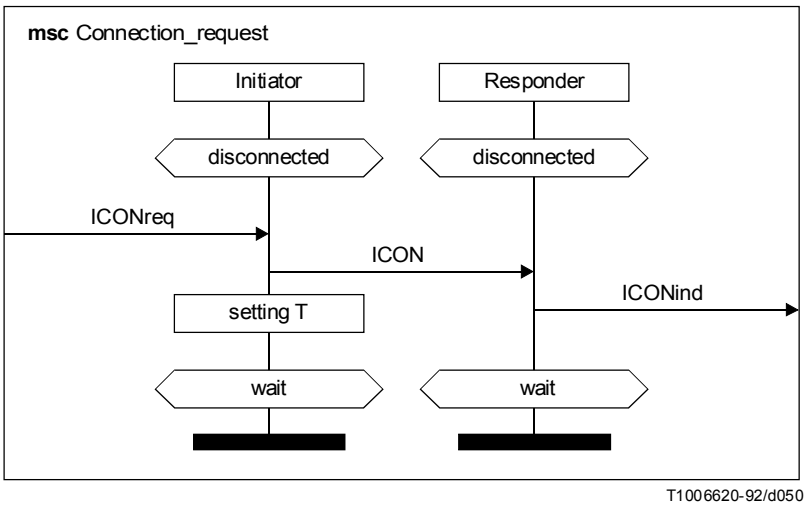
Etape 2 – Décomposition en MSC atomiques

– Décomposer les MSC donnés en MSC «atomiques», c'est-à-dire des MSC qui ne contiennent pas de conditions intermédiaires.

Les MSC atomiques dans les exemples I.6-5a à I.6-5g montrent seulement les cas de non-échec à l'égard du moyen de transmission, c'est-à-dire les cas où les signaux ne sont pas perdus. La situation de débordement de temporisation dans l'exemple I.6-5c provient du fait que le répondeur ne répond pas à temps, mais elle n'est pas causée par la perte de signaux.

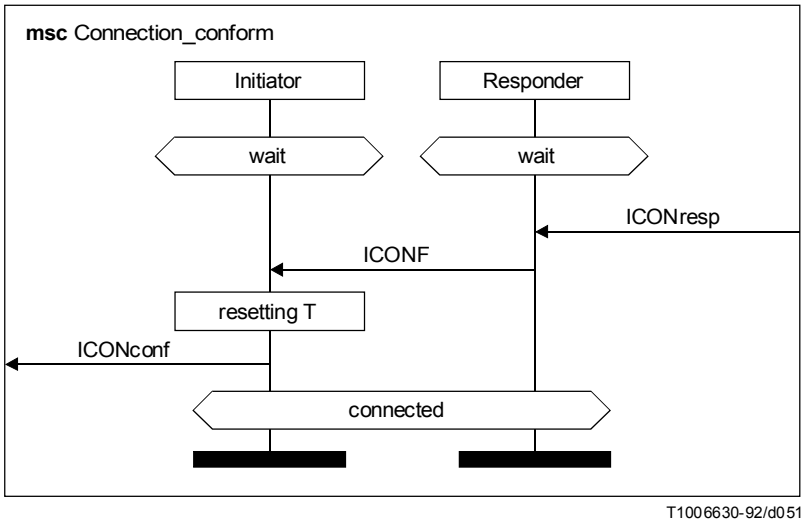
Les MSC sont très liés aux diagrammes des temps utilisés dans le modèle de référence de base OSI. Cependant, les MSC contiennent davantage d'informations: à côté des actions et des débordements de temporisation, ils contiennent aussi des *conditions* qui fournissent la base pour les mécanismes de composition.

Des conditions à la fois locales et globales sont utilisées dans les exemples. Les conditions locales sont employées pour faciliter la continuation d'un MSC par un autre en ce qui concerne un sous-ensemble d'instances. Par exemple, le MSC *Connection_request* de l'exemple I.6-5a peut se continuer par le MSC *Timeout* de l'exemple I.6-5c en ce qui concerne l'instance *Initiator*. Les conditions globales, par exemple *connected*, sont utilisées pour une composition globale, c'est-à-dire une composition concernant toutes les instances. Remarquer que l'armement d'un temporisateur *T*, le désarmement correspondant et le débordement de la temporisation apparaissent dans différents MSC et doivent, par conséquent, être mentionnés comme des actions.



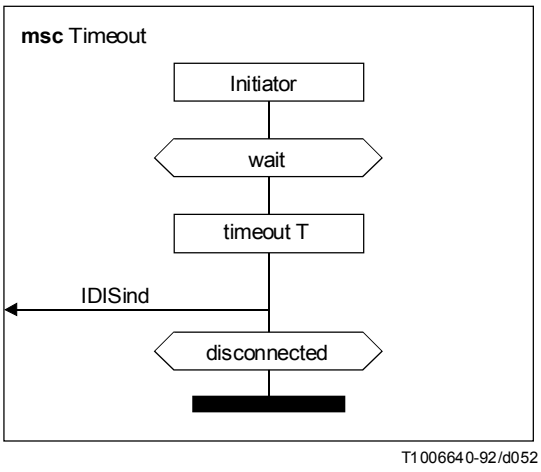
EXEMPLE I.6-5a

MSC atomique



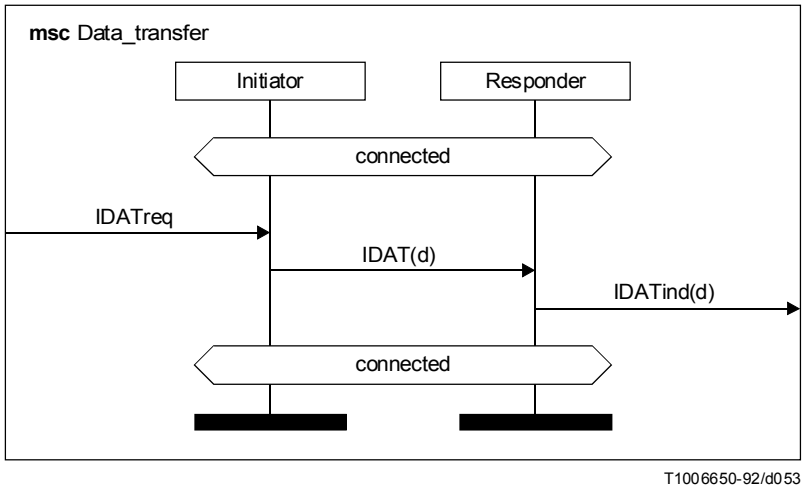
EXEMPLE I.6-5b

MSC atomique



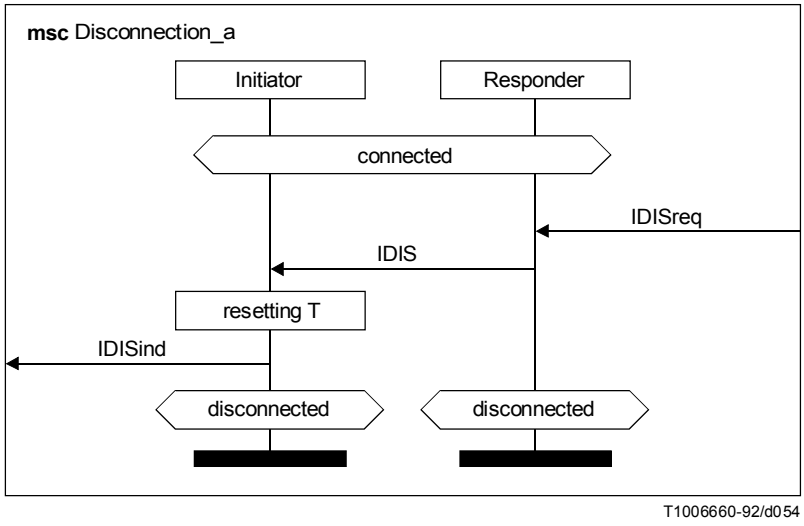
EXEMPLE I.6-5c

MSC atomique



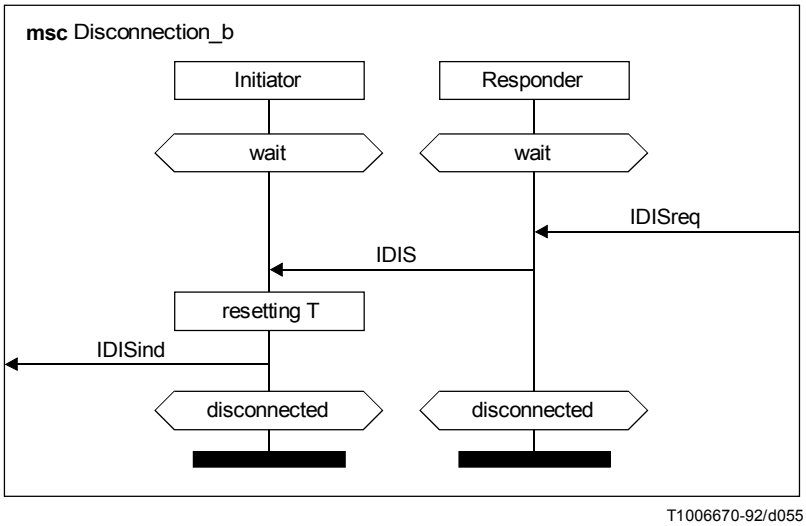
EXEMPLE I.6-5d

MSC atomique



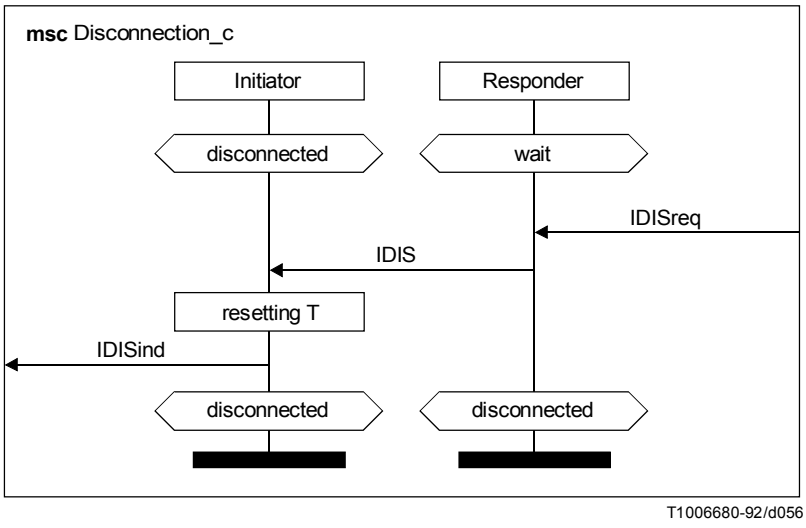
EXEMPLE I.6-5e

MSC atomique



EXEMPLE I.6-5f

MSC atomique



EXEMPLE I.6-5g

MSC atomique

Etape 3 – Création d'un diagramme de vue d'ensemble de MSC

– En commençant par l'état initial et en suivant les règles de composition des MSC, on peut relier ensemble les MSC atomiques. L'ensemble des suites de MSC obtenu peut être présenté sous la forme d'un *diagramme de vue d'ensemble de MSC* en identifiant les états du système atteint, fournis par les conditions exprimées dans les MSC.

La Figure I.6-2 présente le diagramme de vue d'ensemble de MSC obtenu à partir de l'ensemble des MSC atomiques des exemples I.6-5a à I.6-5g. Les nœuds représentent les états globaux du système. L'état initial *disconnected* est marqué. Un tel diagramme de vue d'ensemble peut être utile généralement pour montrer les relations/connexions entre plusieurs MSC. Un diagramme de vue d'ensemble peut être considéré comme un diagramme auxiliaire pour les MSC, correspondant au diagramme de vue d'ensemble des états du SDL.

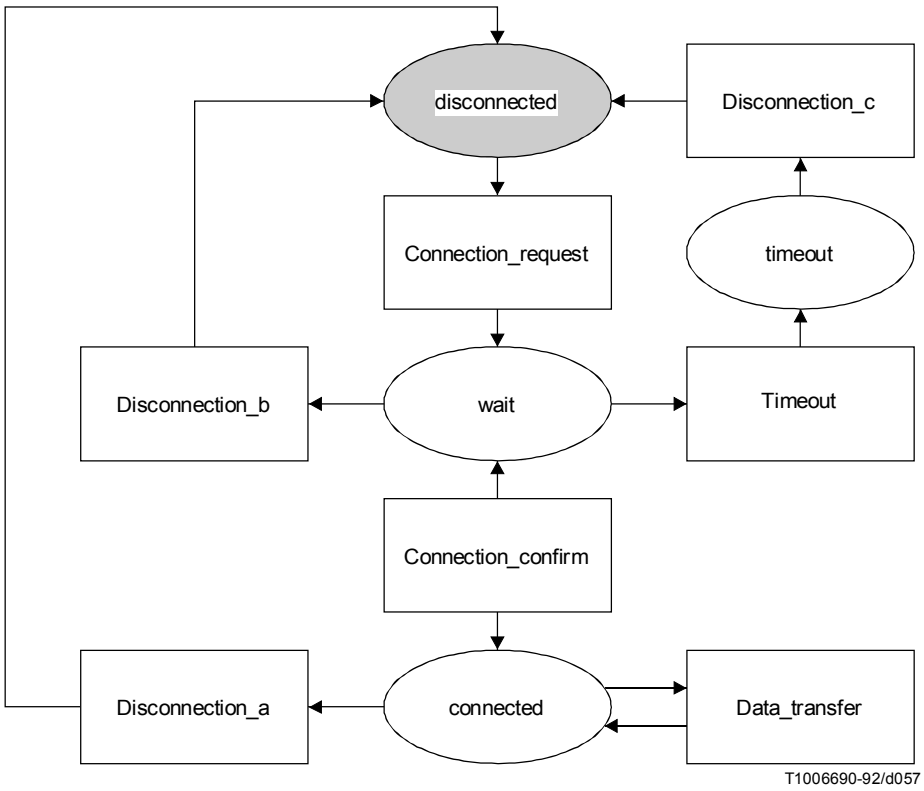


FIGURE I.6-2/Z.100

Un diagramme de vue d'ensemble de MSC

Etape 4 – Décomposition d'un diagramme de vue d'ensemble de MSC

– Le diagramme de vue d'ensemble de MSC est décomposé au nœud affecté de l'état initial. En partant de ce nœud, le diagramme est décomposé plus loin aux nœuds d'où émergent des sous-traces cycliques. A chaque arc connecté à un tel nœud, on attache la copie correspondante de ce nœud après décomposition.

Etape 5

– Les traces maximales à l'intérieur des fragments obtenus à partir du diagramme de vue d'ensemble de MSC forment des MSC normalisés.

Nous obtenons cinq MSC normalisés à partir de la Figure I.6-3, pour laquelle *Data_transfer* et *Disconnection_a* sont les mêmes que dans l'exemple I.6-5. Les nouveaux MSC normalisés sont présentés dans les exemples I.6-6a à I.6-6c.

De façon à obtenir une vue d'ensemble de toutes les traces possibles du système, il faut présenter la relation/connexion entre les MSC normalisés dérivés. Ceci est fait à la Figure I.6-4, où les MSC normalisés sont de nouveau représentés comme des transitions dans un diagramme d'ensemble de MSC.

Une comparaison des Figures I.6-2 et I.6-4 montre la possibilité de structuration des MSC normalisés. Naturellement, l'utilité de MSC normalisés est encore plus évidente dans le cas d'exemples plus réalistes contenant des traces importantes de système.

I.6.3 Une méthode pour le développement de système basée sur les MSC, le SDL et la décomposition fonctionnelle

Puisque l'importance principale des MSC réside dans la phase de définition des besoins, alors que dans la pratique industrielle le SDL est plutôt utilisé pour la conception de système, il semble évident de dériver des spécifications SDL à partir d'un ensemble de MSC. Cependant, une telle approche est réalisable automatiquement en pratique seulement jusqu'à un certain niveau. La spécification SDL doit être «raffinée» ensuite à la main. De plus, une telle chronologie stricte (cycle de vie en «cascade») présente une vue simplifiée de la conception de système. Le SDL et les MSC devraient être utilisés en parallèle dans différentes phases du développement de système, l'un complétant l'autre, et étant interdépendant de différentes façons. Dans la suite, on présente une méthodologie plus sophistiquée supportant la conception descendante basée sur les concepts de «raffinage».

Un concept de télécommunication est décrit selon trois points de vue différents mais interdépendants: *fonctionnel*, *comportemental* et *architectural*. L'aspect fonctionnel peut être modélisé par une hiérarchie de fonctions, l'aspect comportemental par des diagrammes de séquences de messages et l'aspect architectural par le SDL. Il faut noter que cette affectation schématique n'exprime que la principale ligne directrice, à savoir que chaque sorte de description fait ressortir un aspect de façon plus évidente.

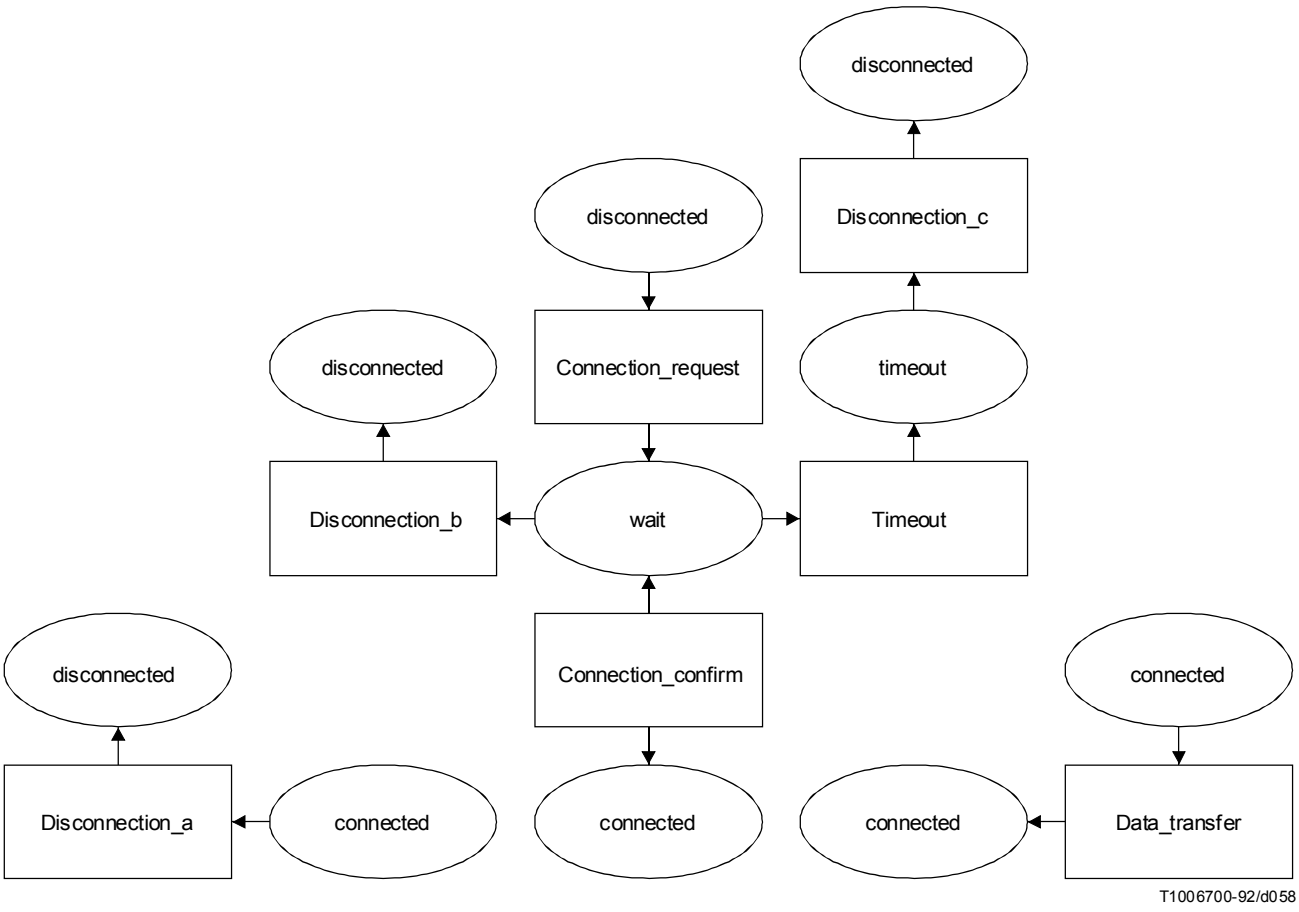
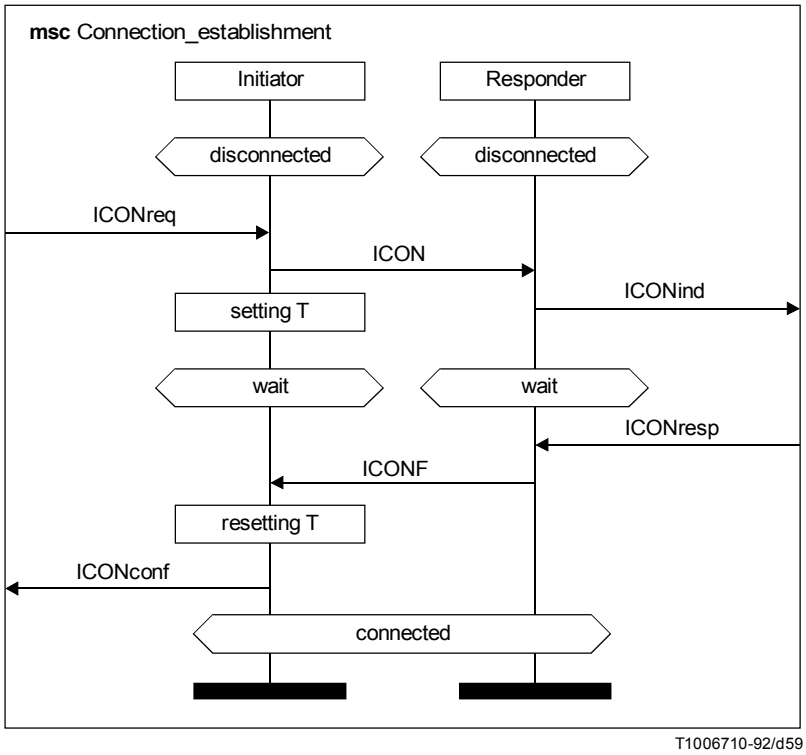


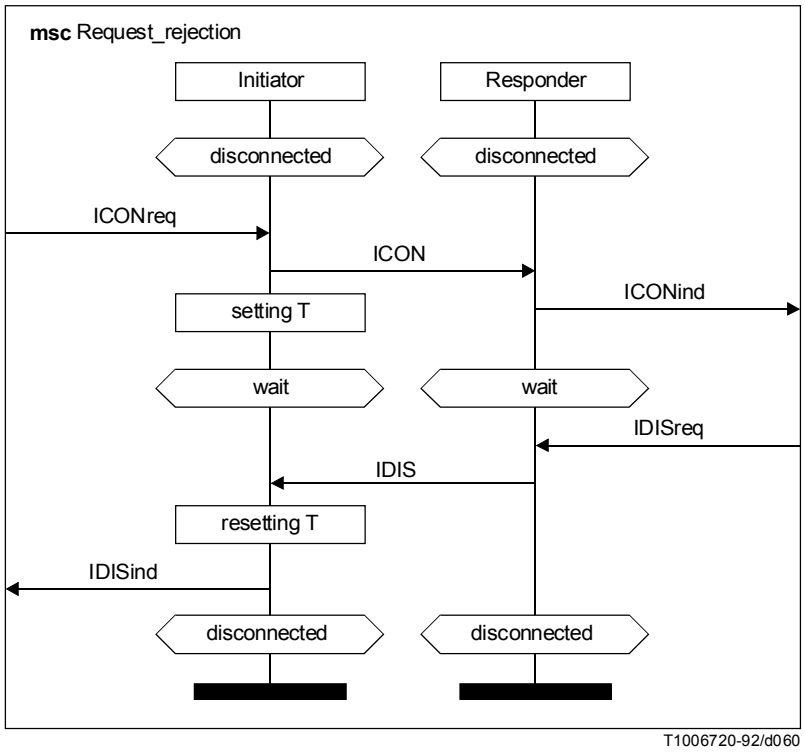
FIGURE I.6-3/Z.100

Décomposition du diagramme de vue d'ensemble de MSC



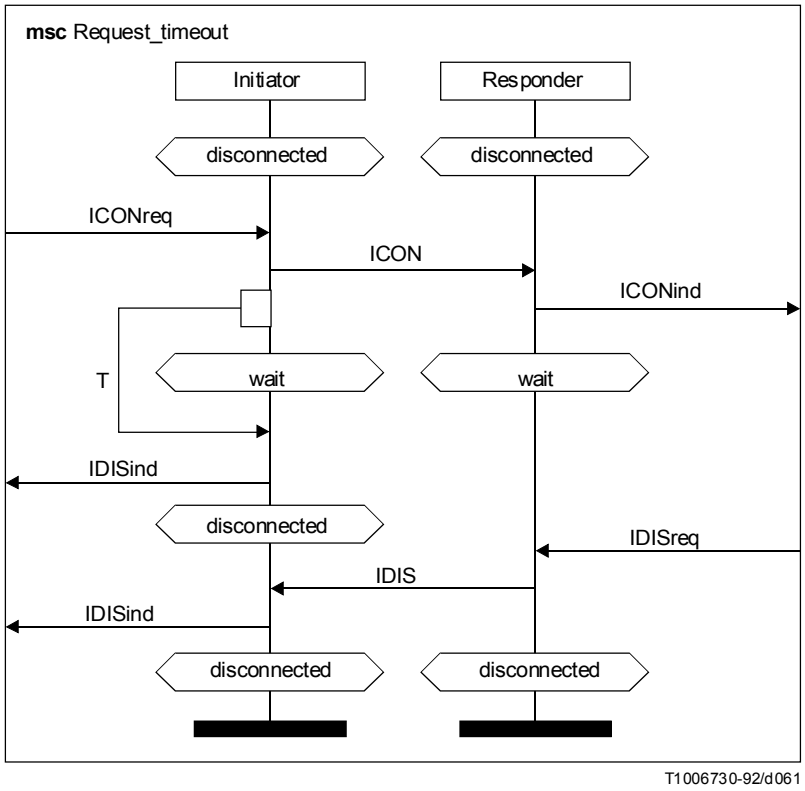
EXEMPLE I.6-6a

MSC normalisé



EXEMPLE I.6-6b

MSC normalisé



EXEMPLE I.6-6c

MSC normalisé

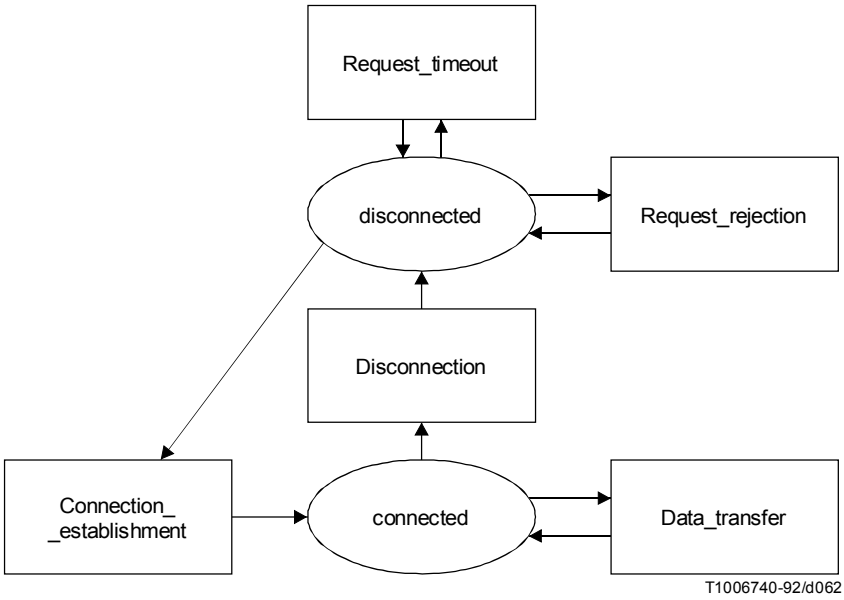


FIGURE I.6-4/Z.100

Diagramme de vue d'ensemble pour des MSC normalisés

I.6.3.1 Découpe fonctionnelle

Les fonctions sont utilisées pour la spécification fonctionnelle de système et nous pouvons distinguer:

- *Les fonctions globales*, chacune d'elles est décomposée en une hiérarchie de fonctions de niveau inférieur;
- *Les fonctions abstraites* qui représentent des nœuds intermédiaires des niveaux de la hiérarchie de fonctions;
- *Les fonctions terminales* qui forment les feuilles dans la hiérarchie de fonctions et ainsi ne sont pas davantage décomposées. Elles représentent des fonctions très précises et peuvent être décrites par une hiérarchie de diagrammes de séquences de messages.

Pour exprimer la relation temporelle entre les fonctions, on associe un opérateur de composition à chaque fonction globale et à chaque fonction abstraite, en indiquant comment les sous-fonctions sont ordonnées dans le temps et entre elles. Il y a six opérateurs:

- **is** (est) – la fonction résultante est une sous-fonction;
- **and** (et) – la fonction résultante est la séquence des sous-fonctions ordonnées dans le temps, de la gauche vers la droite;
- **parallel** (parallèle) – il n'y a pas d'ordre temporel entre les sous-fonctions;
- **or** (ou) – la fonction résultante est l'une des sous-fonctions;
- **repeat** (répéter) – la fonction résultante est la répétition de sa sous-fonction;
- **exception** – la fonction résultante est sa sous-fonction lorsqu'une erreur se produit.

La Figure I.6-5 montre un exemple de décomposition fonctionnelle. La fonction *normal call* (appel normal) correspond à la séquence *dialing* (numérotation), *conversation* et *disconnection* (déconnexion). Remarquer que les symboles utilisés pour les opérateurs ne font pas partie du SDL.

Le formalisme est utilisé pour exprimer ce que le système doit réaliser. Le SDL décrira comment le système doit le réaliser. Les fonctions définies sont ainsi mises en œuvre par la description SDL.

I.6.3.2 Spécification SDL

On spécifie le système complet au moyen du SDL. Un aspect important du SDL est sa capacité à décrire une décomposition hiérarchique représentant l'architecture du système. Chaque nœud SDL dans cette hiérarchie (système, bloc, processus) est appelé par la suite une *entité*.

I.6.3.3 Diagrammes de séquences de messages

Un ensemble de diagrammes de séquences de messages (MSC) est attaché à chaque fonction terminale. Cet ensemble est structuré de nouveau comme une hiérarchie de MSC. La hiérarchie de MSC doit correspondre à la hiérarchie SDL. Chaque MSC décrit comment un service terminal est fourni à un certain niveau de décomposition d'une architecture de système SDL. Un MSC peut être considéré comme la projection d'un service terminal sur un niveau de décomposition de la hiérarchie SDL.

I.6.3.4 Règles de cohérence entre descriptions

La méthodologie présentée exige que les trois sortes de descriptions soient maintenues cohérentes pendant toutes les phases de conception de système.

La Figure I.6-6 montre une vue globale des relations entre les 3 sortes de descriptions au moyen d'un exemple simple et illustre les règles de cohérence. Les règles suivantes doivent s'appliquer.

- La décomposition des MSC doit être cohérente avec la décomposition du SDL.
- Toutes les instances et tous les messages dans un MSC doivent être visibles dans l'entité correspondante dans la description SDL.
- Un MSC doit être cohérent avec la description SDL au niveau de décomposition correspondant.

Une instance de MSC montre un ensemble d'événements d'entrée et de sortie qui suivent un ordre entièrement linéaire. Par conséquent, une instance de MSC peut être interprétée comme une séquence d'événements. Cette séquence est appelée une trace MSC.

Une spécification de processus SDL peut se représenter par un graphe des marquages accessibles: les nœuds sont les états du processus et les arcs étiquetés représentent les événements d'entrée. Une trace SDL d'un processus est une suite d'événements dans le graphe des marquages accessibles.

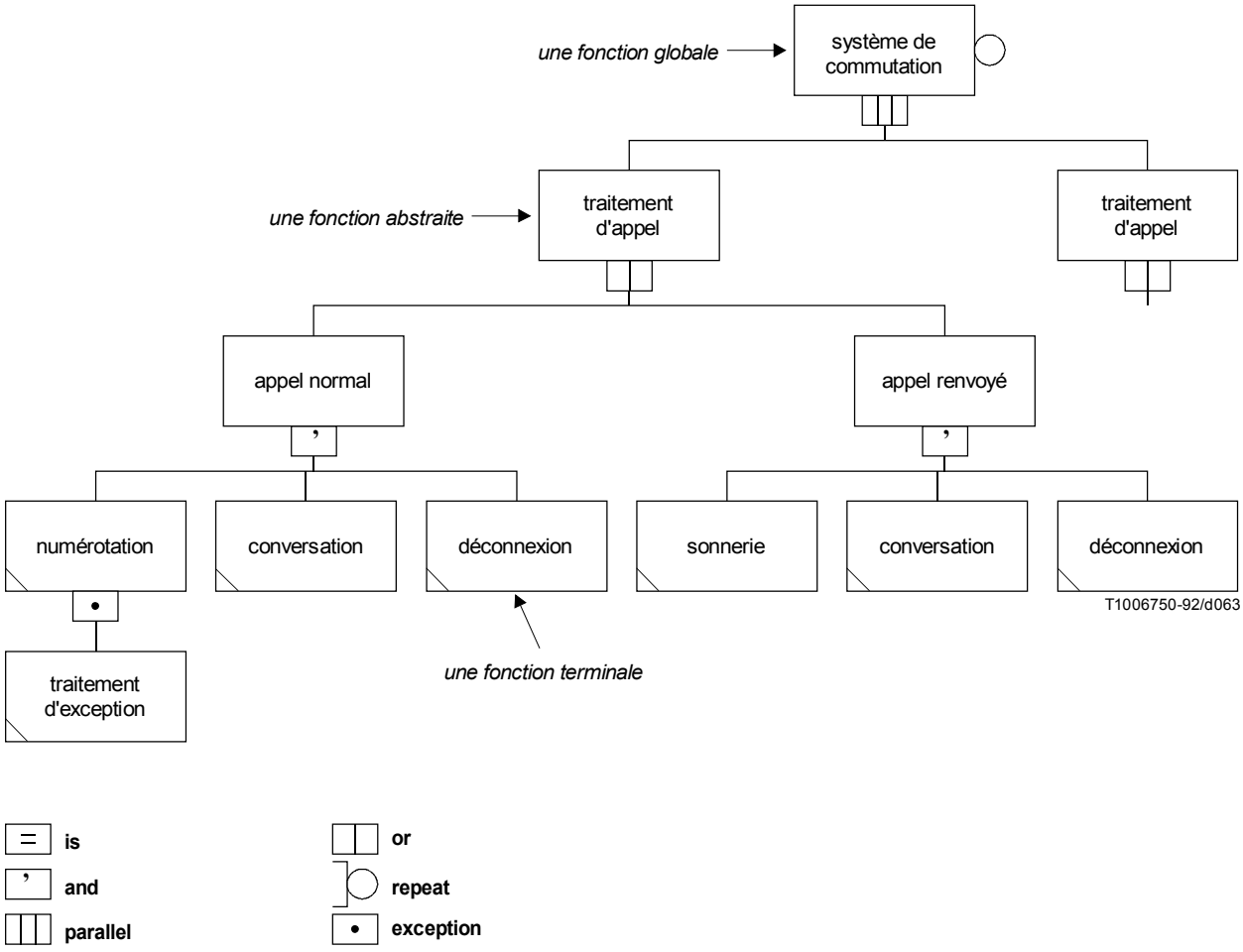


FIGURE I.6-5/Z.100

Exemple d'une décomposition fonctionnelle

Maintenant, la définition de la cohérence est réduite à un problème de la théorie des graphes. Une instance MSC est cohérente avec une spécification de processus SDL, s'il existe une trace SDL qui contient une trace MSC et ne fait apparaître aucun événement suplémentaire d'entrée et de sortie (voir la Figure I.6-7).

Cette règle de cohérence n'est pas suffisante pour toutes les sortes de MSC. On pourrait laisser de côté les informations relatives au flux des signaux entre les processus, par exemple en cas de dépassement de signaux. La définition des traces et une règle de cohérence correspondante peuvent être généralisées pour couvrir toutes sortes de MSC. En pratique, cependant, un contrôle automatique de cohérence est réalisable sur les instances MSC de taille réduite en raison du problème de l'explosion des états.

I.7 Dérivations de mise en œuvre à partir de spécifications SDL

Le problème abordé dans cet article concerne la façon de faire correspondre les systèmes abstraits spécifiés en SDL et les systèmes réels construits à partir de composants matériels et logiciels. Les composants du monde réel diffèrent normalement des composants abstraits tant par la structure que par le comportement. Par conséquent, des adaptations sont nécessaires à la fois sur les niveaux abstrait et concret pour s'assurer que la spécification SDL définit fidèlement les fonctionnalités du système réel. En plus du SDL pur, la documentation est nécessaire pour définir le système concret et ses relations avec le système SDL abstrait. Cet article met en évidence les étapes les plus importantes pour déduire des mises en œuvre et indique comment on peut documenter la mise en œuvre au niveau architectural. Les directives établies progressivement sont résumées à la fin de l'article.

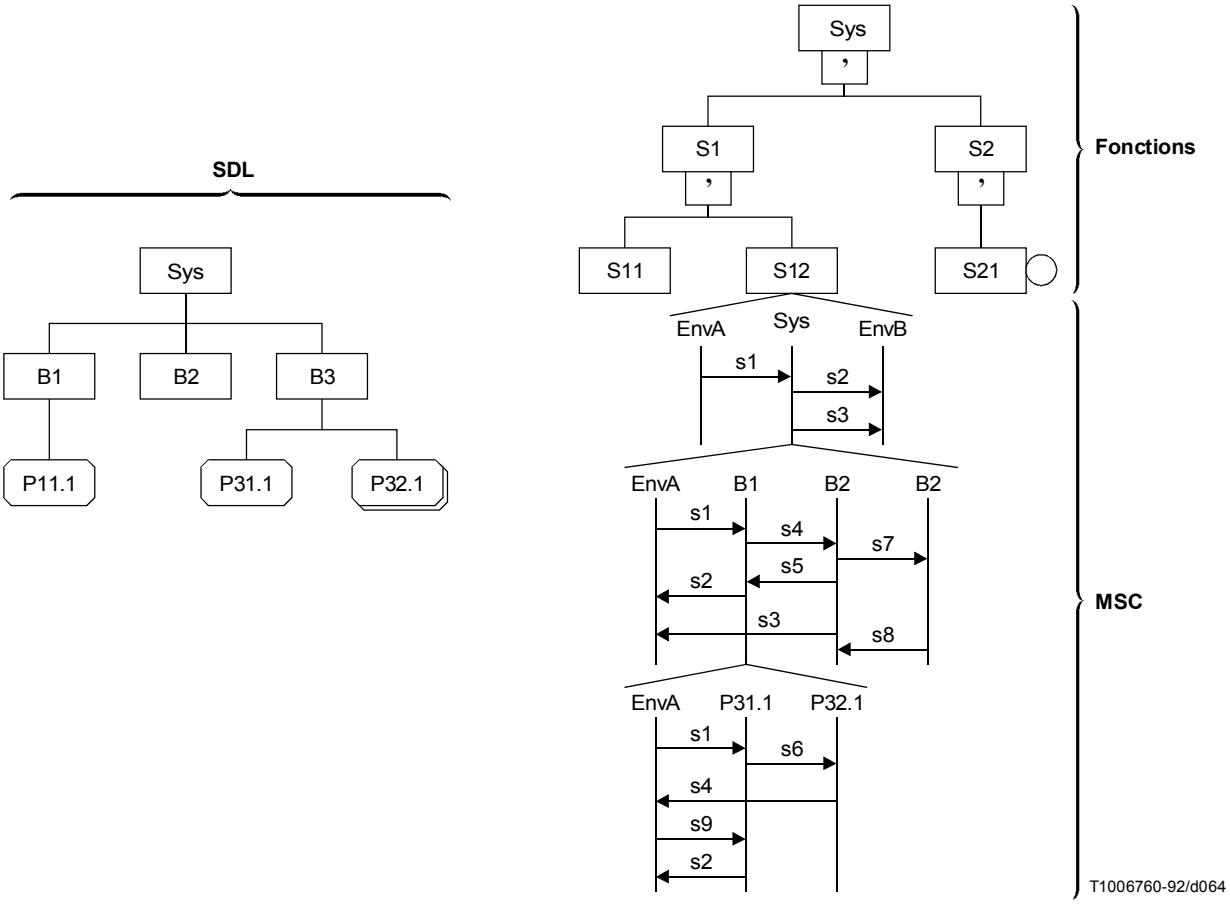


FIGURE I.6-6/Z.100

Les trois vues d'un système

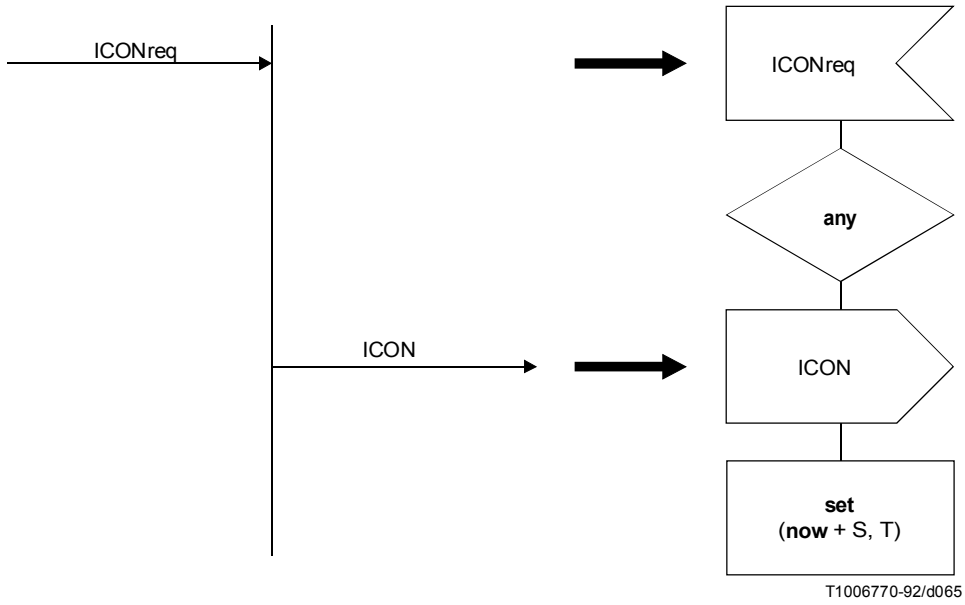


FIGURE I.6-7/Z.100

Cohérence entre trace SDL et trace MSC

I.7.1 Introduction

Pendant la durée de vie d'un système, les spécifications SDL seront utilisées avec au moins trois objectifs:

- A une étape très en amont, pour spécifier et pour valider les fonctionnalités (comportement) requises par l'environnement de l'utilisateur (spécification de haut niveau).
- Puis, pour fournir une base solide pour la conception (de la mise en œuvre), c'est-à-dire la conception optimale (spécification de bas niveau).
- Après la conception, pour décrire (documenter) les propriétés fondamentales complètes du système tel qu'il est réalisé (spécification orientée vers la mise en œuvre).

Le SDL est basé sur des concepts très bien adaptés au premier objectif mentionné ci-dessus: pour spécifier le comportement observable des systèmes d'une manière claire et non ambigüe. Pour cette raison, le comportement externe devrait être mis en valeur et les détails de conception interne non pertinents devraient être écartés.

Le SDL est aussi bien adapté au deuxième objectif: pour être une base pour la conception de la mise en œuvre. Le SDL a la bonne propriété de combiner l'indépendance vis-à-vis de la mise en œuvre et la capacité d'exprimer la mise en œuvre (sauf pour les types abstraits infinis). Pour cette raison, aucune décision prématurée de conception ne devrait être incluse dans les spécifications SDL, mais des besoins dits non fonctionnels, c'est-à-dire les propriétés que la mise en œuvre devra posséder en plus de celles exprimées dans la spécification SDL, peuvent être données comme directive complémentaire.

Si le système réel est fonctionnellement équivalent au système SDL, le troisième objectif ci-dessus est également atteint. Ceci vaut la peine de viser cet objectif. Non seulement parce qu'il économise l'effort de documentation, mais aussi parce que les spécifications SDL sont utiles pendant le test, l'exploitation et la maintenance du système. Lorsque les concepteurs et les programmeurs trouvent les spécifications SDL utiles dans leur travail quotidien, ils sont motivés pour les maintenir à jour et éviter les changements au niveau de la mise en œuvre qui altèrent la documentation. Pour cette raison, les spécifications SDL doivent normalement être quelque peu adaptées pour refléter les propriétés fonctionnelles qui sont spécifiques de la mise en œuvre sous-jacente.

Il y a deux aspects majeurs impliqués par la conception de la mise en œuvre d'un système SDL:

- l'aspect «aval» du choix parmi plusieurs solutions de mise en œuvre conformes sur le plan fonctionnel au système SDL. Ce choix existe habituellement et il appartient au concepteur de retenir la mise en œuvre optimale pour ce qui est des spécifications non fonctionnelles;
- l'aspect de rétroaction concernant l'adaptation de la spécification SDL dans le cas où la mise en œuvre choisie n'est pas fonctionnellement équivalente. Il y a d'importantes différences entre le monde abstrait du SDL et le monde réel qui va quelquefois se faire jour dans le comportement observable du système. Dans de telles situations, il faut adapter la spécification SDL complète du système pour maintenir l'équivalence fonctionnelle.

La Figure I.7-1 illustre l'aspect «aval». On utilise ensemble la spécification SDL et la spécification des besoins non fonctionnels pour choisir et spécifier une mise en œuvre. La spécification de mise en œuvre résultant de cette activité définit la correspondance entre la spécification SDL et la mise en œuvre du système réel. Elle est orthogonale à la spécification SDL et par conséquent apparaît comme une boîte séparée à la Figure I.7-1. La séparation ne constitue pas ici le point clé, mais la nature orthogonale des informations. En pratique, la spécification de la mise en œuvre peut être incluse en tant qu'annotation de la spécification SDL. Quelques observations peuvent être faites à propos de la Figure I.7-1:

- Une et une seule spécification SDL remplit, dans ce cas, les trois objectifs des spécifications SDL mentionnés plus haut.
- Les mises en œuvre différentes peuvent être déduites de la même spécification SDL en fournissant des spécifications de choix de mise en œuvre.
- Si l'activité de mise en œuvre est automatisée, la mise en œuvre restera constamment cohérente avec la spécification (jusqu'au point de correction de la traduction).

Dans beaucoup de situations pratiques, le dessin quelque peu idéalisé de la Figure I.7-1 sera modifié par les effets de la rétroaction. Si le système doit être réalisé par un réseau de calculateurs répartis, par exemple, certains des canaux SDL seront mis en œuvre en utilisant les protocoles du réseau. Ces protocoles sont nécessaires dans une mise en œuvre répartie, mais pas pour une solution centralisée. Etant donné que des parties d'un système réparti peuvent devenir non opérationnelles alors que simultanément d'autres parties restent opérationnelles, les traitement des erreurs est différent dans un système réparti comparé à un système centralisé. Par conséquent, pour définir la fonctionnalité complète réellement mise en œuvre, la spécification SDL d'un système réparti peut être différente de celle d'un système centralisé.

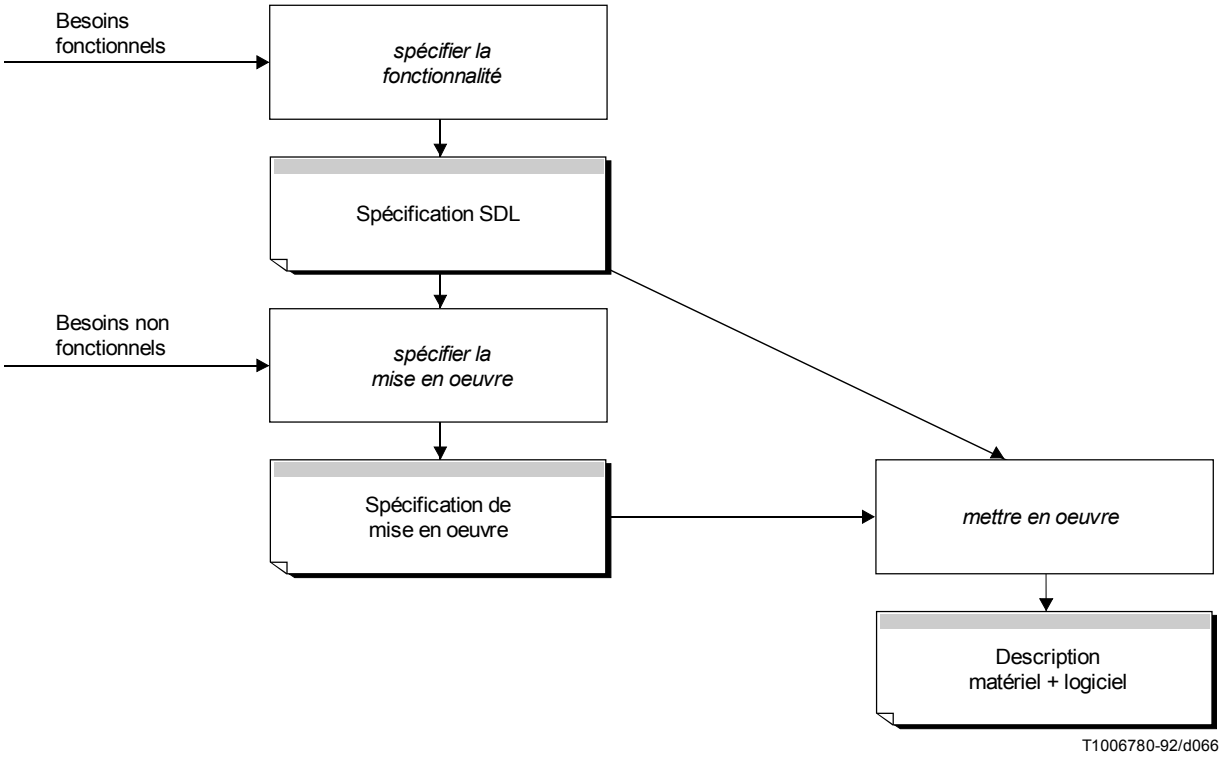


FIGURE I.7-1/Z.100

L'aspect de rétroaction; les besoins non fonctionnels sont utilisés pour choisir une mise en oeuvre fonctionnellement équivalente

La Figure I.7-2 illustre l'influence réciproque entre les aspects «aval» et de rétroaction. Comme à la Figure I.7-1, les propriétés non fonctionnelles sont utilisées pour dériver une spécification de mise en œuvre. La connaissance relative à la mise en œuvre est alors utilisée pour raffiner et restructurer la spécification SDL en une spécification SDL complète raffinée de bas niveau. Mais même cette dernière spécification SDL va laisser place à des choix de mises en œuvre fonctionnellement équivalents. Puisque la spécification de mise en œuvre est encore nécessaire pour conduire l'étape de mise en œuvre juste comme l'indique la Figure I.7-1.

Les observations suivantes s'appliquent à la Figure I.7-2:

- Deux spécifications SDL, une spécification de haut niveau et une spécification de bas niveau sont nécessaires pour remplir les trois objectifs du SDL.
- Pour une spécification SDL de haut niveau donnée, plusieurs spécifications SDL de bas niveau peuvent être dérivées en fonction de la spécification de la mise en œuvre.
- L'aspect de rétroaction et l'aspect «aval» peuvent être séparés et combinés de façon structurée.

Lorsqu'un nouveau système est développé à partir de rien, il est souhaitable que la spécification SDL soit suffisamment indépendante de la mise en œuvre pour répondre au deuxième objectif (mentionné au début de cet article). Ceci peut signifier une rétroaction considérable lorsque la mise en œuvre est choisie[4)]. Lorsqu'un système existant est étendu ou amélioré, beaucoup plus de choses sont connues concernant la mise en œuvre de sorte que l'on peut accéder plus directement à la spécification SDL de bas niveau et éviter la rétroaction.

4) Mais cela ne veut pas dire que chaque chose est changée dans la spécification. Souvent le raffinage et la restructuration nécessaires peuvent ne concerner que des parties limitées du système.

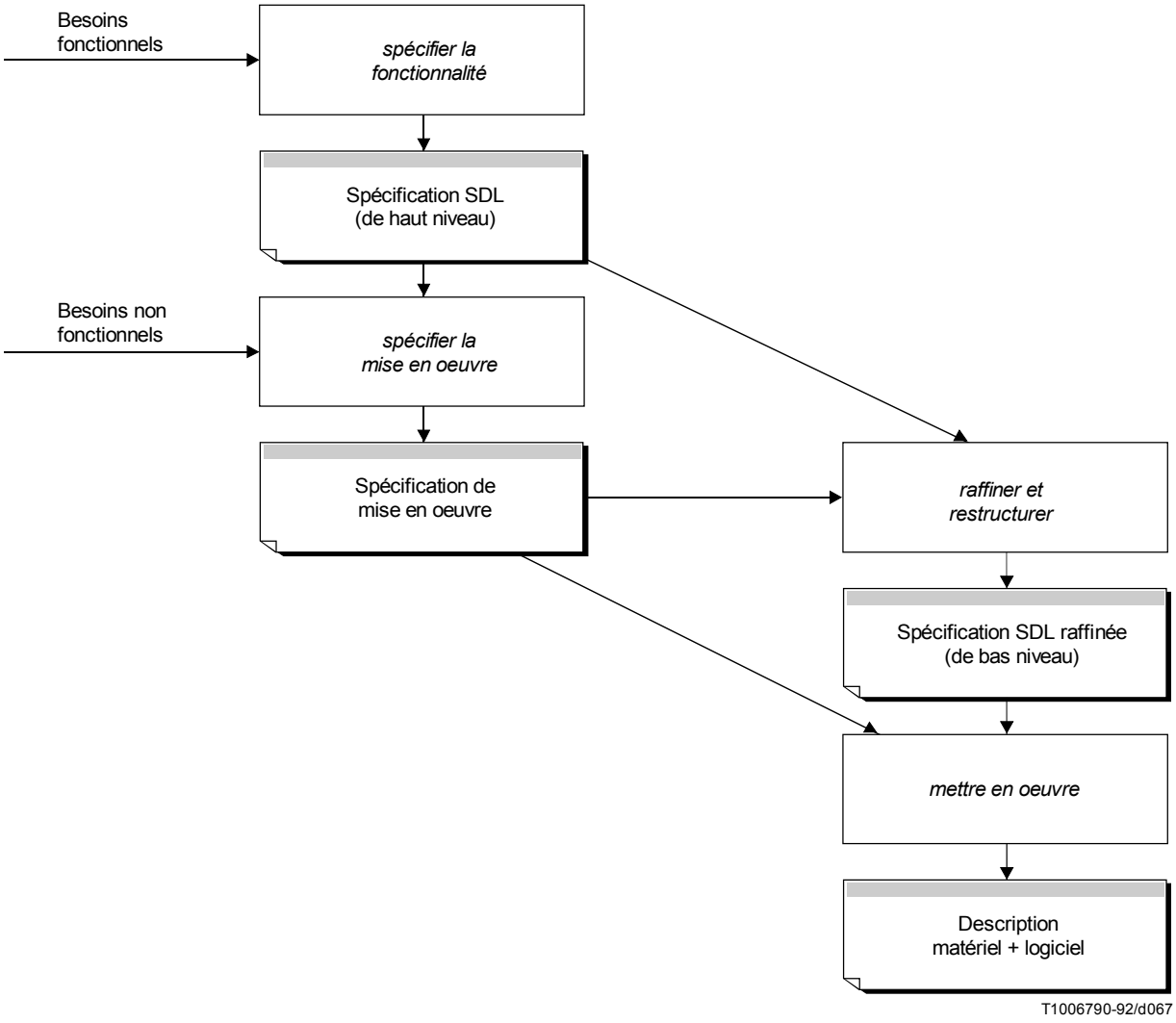


FIGURE I.7-2/Z.100

Les aspects «aval» et de rétroaction de la conception de mise en oeuvre

Trois questions peuvent dès lors se poser:

- Quelles sont les différences essentielles entre les systèmes SDL et les systèmes réels qui doivent être prises en considération?
- Comment devrait être exprimée la spécification de mise en œuvre: comme annotation à la spécification SDL de bas niveau ou de manière séparée à partir de la spécification SDL?
- Quels sont les principaux choix de mises en œuvre à partir desquels on peut choisir?

Ces questions seront discutées dans le reste de ce paragraphe.

I.7.2 Différences entre systèmes réels et systèmes SDL

Un bon concepteur doit être averti des différences entre le monde réel et le monde abstrait. Il y a deux catégories principales de telles différences:

- les différences fondamentales dans la nature des composants. Les composants physiques sont plutôt imparfaits comparés aux propriétés les plus idéales des composants SDL. Ils produisent des erreurs au cours du temps, ils sont sujets au bruit, et ils ont besoin de temps pour exécuter les tâches de calcul;
- des différences conceptuelles dans le fonctionnement des composants. Dans les deux mondes, il y a des concepts pour la concurrence d'accès, la communication, le comportement séquentiel et les données mais ils ne sont pas nécessairement les mêmes.

I.7.2.1 Différences fondamentales

Le temps de traitement

Un système SDL n'est pas limité par les ressources de calcul. En conséquence, on ne doit pas considérer l'équilibre entre la charge de trafic offerte au système et sa capacité de traitement. On suppose simplement que le système est suffisamment rapide pour traiter la charge qui est offerte.

Le monde réel est beaucoup plus différent sur ce point. Chaque transfert de signal et chaque transition d'un processus va prendre un certain temps et nécessiter certaines ressources pour le traitement. Par conséquent, une des questions les plus importantes dans la conception de mise en œuvre est d'établir l'équilibre entre la capacité de traitement de la mise en œuvre et la charge de trafic offerte.

Une question liée concerne la vitesse à laquelle le système doit répondre à certaines heures. De nouveau le système SDL n'a pas de problème mais la mise en œuvre peut être très contrainte sur le respect des temps de réponse.

En raison de ces différences, le point central de la conception de la mise en œuvre est complètement différent de celui de la spécification fonctionnelle. Le «challenge» est de trouver des réalisations pour les concepts SDL qui sont suffisamment efficaces pour respecter les besoins en termes de charge de trafic et de temps de réponse sans affecter la validité des spécifications SDL.

Puisque les spécifications SDL spécifient clairement les interactions externes et internes nécessaires à l'exécution de fonctions données, elles fournissent une excellente base pour estimer la capacité de traitement nécessaire pour respecter les contraintes de charge et de temps. Ceci est décrit de façon détaillée au I.7.4.3.

Erreurs et bruit

Les systèmes SDL peuvent souffrir d'erreurs de spécification, mais le monde abstrait du SDL ne souffre d'aucune erreur physique. On suppose simplement que les processus et les canaux se comportent toujours conformément à leur spécification. On ne considère pas que les processus vont s'arrêter de temps en temps ou que les canaux vont altérer le contenu des signaux. Mais dans le monde réel, de telles choses arrivent. De temps en temps, des erreurs vont se manifester elles-mêmes comme des fautes dans le fonctionnement des canaux et des processus.

Les erreurs matérielles, le bruit, les dommages physiques sont inévitables, puisqu'ils sont provoqués par des processus physiques entièrement en dehors du royaume de la logique. De plus, les erreurs logiques qui n'étaient pas dans la spécification SDL seront normalement introduites dans la mise en œuvre.

Les mécanismes qui provoquent les erreurs et le bruit sont en dehors de la portée de la spécification SDL. Mais l'effet des erreurs et du bruit va souvent devoir être manipulé explicitement dans la spécification SDL. On doit considérer ce qu'un processus devrait faire s'il n'obtient jamais de réponse d'un autre processus, ou s'il obtient une réponse erronée. Est-il possible qu'un canal tombe en panne ou que du bruit altère les signaux? Quelle devrait être la réaction à un canal tombant en panne? Et si un processus se met à produire des signaux «anormaux»? Ce sont de telles questions qu'il faut se poser et auxquelles il faut répondre.

Dans une certaine mesure, les réponses dépendent de la répartition physique des processus SDL dans le système réel et des distances physiques que les canaux SDL doivent couvrir. Des processus et des canaux physiquement séparés peuvent tomber en panne indépendamment les uns des autres. Les canaux couvrant de longues distances physiques sont sujets à plus de bruit et de fautes que des canaux réalisés en logiciel dans un même ordinateur.

Un effet positif de la séparation physique est que les erreurs sont isolées. Il ne faut pas que les erreurs dans une unité affectent les autres unités du système, à condition que les informations erronées ne leur soient pas propagées ou qu'elles soient capables de se protéger elles-mêmes. Ainsi, la séparation physique peut améliorer le traitement d'erreur. Mais ceci normalement ne viendra pas sans rien. Les erreurs doivent être détectées et isolées pour permettre aux parties opérationnelles de poursuivre leur traitement en présence de l'erreur. Un traitement approprié de cet aspect peut être très complexe et exiger normalement une fonctionnalité supplémentaire dans la spécification SDL.

Une spécification SDL ne renseigne en rien sur la distance physique couverte par un canal. En réalité cependant, il peut y avoir (ou pas) de grandes distances physiques. Ceci signifie qu'il faut un équipement de transmission et des protocoles pour mettre en œuvre la fiabilité du canal. Ainsi, il se peut que la distance physique nécessite d'introduire de nouvelles fonctions pour consolider la mise en œuvre des canaux.

Ressources finies

Dans un système réel, toutes les ressources sont finies. Il peut y avoir un nombre maximal de processus que le système d'exploitation peut traiter ou un nombre maximal de tampons pour les envois de messages. La taille du mot est limitée, comme l'est l'espace mémoire. Même les données primaires comme les entiers sont finies.

Le SDL, de son côté, possède une file d'attente non bornée dans le port d'entrée de chaque processus et permet de spécifier des données infinies. Par conséquent, le concepteur doit trouver les façons de mettre en œuvre les systèmes SDL potentiellement infinis en utilisant des ressources finies. Une façon de procéder consiste à restreindre l'utilisation du SDL de telle sorte que toutes les valeurs soient sûres d'être bornées. Une autre consiste à traiter les limitations de ressources lors de la mise en œuvre, de préférence de manière transparente au niveau SDL. Dans le cas où la transparence ne peut être assurée, on doit soit accepter une déviation de la sémantique du SDL, soit considérer explicitement les limitations de la spécification SDL.

I.7.2.2 Différences conceptuelles

Concurrence d'accès

Le modèle de concurrence d'accès utilisé dans le SDL suppose que les processus se comportent indépendamment les uns des autres et de manière asynchrone. Il n'y a pas d'ordre relatif des opérations dans les différents processus sauf ordre impliqué par l'envoi et la réception de signaux.

Ceci permet de mettre en œuvre les processus SDL soit vraiment en parallèle sur des unités matérielles séparées, soit en quasi-parallélisme sur du matériel partagé.

Les objets physiques du monde réel se comportent réellement en parallèle. Ceci signifie que les opérations dans les différents objets se déroulent en parallèle à la vitesse d'exécution du matériel. Nous pouvons distinguer deux cas:

- *Opération synchrone:* où les opérations d'objets parallèles sont réalisées en même temps. Ceci est la plupart du temps utilisé au niveau du circuit électronique où les opérations peuvent être commandées par les signaux d'une horloge commune. Selon la sémantique du SDL ceci peut être considéré comme un cas particulier.

- *Opération asynchrone:* où les opérations des objets parallèles sont exécutées indépendamment et vraisemblablement à des instants différents. A moins d'avoir une connaissance précise des vitesses de traitement, nous ne pouvons pas connaître l'ordre exact des opérations. Ceci correspond bien à la sémantique du SDL.

Le quasi-parallélisme signifie qu'un seul processus est actif à la fois et que le processus actif va bloquer le fonctionnement des autres processus aussi longtemps qu'il restera actif. Ceci va affecter les temps de réponse des processus bloqués. Il faudra un mécanisme supplémentaire pour assurer l'ordonnancement et le multiplexage des processus quasi-parallèles au-dessus d'une machine séquentielle. Normalement, cette facilité est prise en charge par un système d'exploitation, qui peut être vu comme une couche qui met en œuvre une machine virtuelle quasi parallèle au dessus de la machine physique.

Communication

De manière très simple, il y a deux différentes classes de besoins de communication:

- pour transmettre une séquence de symboles, ou de valeurs, dans un ordre donné;

- pour transmettre un symbole ou une valeur, de façon continue pendant tout le temps où celui-ci reste valable.

Dans le premier cas, l'ordre séquentiel est important. Dans le second cas, la séquence n'a pas d'importance, seule a de l'importance la valeur courante à chaque instant.

Lorsque l'on lit un livre, la séquence des lettres, des mots et des phrases est essentielle à l'interprétation. Lorsque l'on arrive à un feu tricolore, la couleur courante va déterminer si l'on va s'arrêter ou non. L'histoire des changements de couleur antérieurs n'a pas d'importance. Les symboles continus peuvent être lus maintes et maintes fois, tandis qu'au contraire les symboles figurant dans une séquence ne sont normalement lus qu'une seule fois.

Les signaux SDL appartiennent à la catégorie des symboles organisés en séquence. Ils sont lus seulement une fois et sont ensuite consommés à la fin de la réception. Ils conviennent bien pour représenter des séquences d'événements.

La construction view/reveal en SDL appartient à la catégorie des valeurs continues. La construction import/export y ressemble, mais il s'agit d'une représentation abrégée pour un protocole sous-jacent de signaux permettant au récepteur de garder trace d'une valeur continue.

Les deux formes de communication sont duales dans le sens qu'une forme peut être utilisée pour mettre en œuvre l'autre. Nous considérons d'abord le besoin de communiquer une valeur continue. La mise en œuvre la plus directe consiste à utiliser un moyen de communication qui transmettra la valeur continuellement, comme une variable partagée en logiciel, ou une connexion électrique en matériel. Mais on peut aussi bien utiliser un moyen de transmission de nature séquentielle, comme une file d'attente de messages, pour transmettre une séquence de symboles représentant la séquence des changements (événements) affectant la valeur continue. Pour permettre à ce modèle de fonctionner, l'émetteur doit dériver des événements et le récepteur doit intégrer ces événements pour regénérer la valeur continue. Ce principe est appliqué dans le mécanisme d'import/export du SDL.

Que dire à propos du besoin de transmettre une séquence de symboles? La mise en œuvre la plus directe est d'utiliser un moyen de communication de nature séquentielle tel qu'une file d'attente de messages. Mais on peut aussi bien utiliser un moyen permettant de transmettre une valeur continue. Dans ce cas, la séquence doit être représentée par les changements (événements) affectant la valeur continue. Pour permettre à ce modèle de fonctionner, l'émetteur doit intégrer une séquence d'événements pour former la valeur continue et le récepteur doit dériver ces événements à partir des changements de la valeur continue.

Bien qu'il soit possible de mettre en œuvre une forme de communication au moyen de l'autre, on est pénalisé par le changement de paradigme.

Le moyen de communication pour une mise en œuvre «naturelle» des signaux SDL est de nature séquentielle. Mais ceci ne va pas toujours être la meilleure forme dans le système réel. Quelquefois, les signaux SDL doivent être mis en œuvre au moyen de valeurs continues.

L'entrée à partir d'un clavier peut servir d'exemple. Le signal émis à partir de chaque bouton est fondamentalement un «1» continu lorsque la touche est enfoncée et un «0» continu lorsque la touche n'est pas enfoncée. Mais le système a besoin de connaître la séquence des frappes de touches et non les valeurs instantanées. Ainsi, les changements de valeurs (événements) doivent être détectés et convertis en symboles représentant les frappes de touches complètes. La détection d'événements de ce type est souvent nécessaire aux interfaces des systèmes temps-réel. Elle peut être réalisée soit avec du logiciel, soit avec du matériel.

Les signaux visuels sur un écran d'affichage constituent un autre exemple. L'utilisateur veut que les informations soient présentées comme des valeurs continues, et non pas comme des messages furtifs sur l'écran. Par conséquent, le signal SDL doit être converti en une valeur continue sur l'écran.

D'autre part, lorsque l'on utilise les constructions export/import en SDL, il peut être plus judicieux d'utiliser des valeurs continues dans la mise en œuvre plutôt que le protocole de mise en séquence des symboles considéré en SDL[5)].

Les canaux traversant la frontière matériel-logiciel nécessitent une attention particulière. Un canal atomique, représenté par une ligne dans la représentation graphique peut se transformer en un mélange de lignes physiques, d'équipements électroniques et de logiciel dans le système réel. Les primitives de communication et de synchronisation utilisées du côté matériel vont souvent se distinguer de celles utilisées du côté logiciel de la frontière, voir dans les paragraphes suivants. La conversion d'une forme dans l'autre va être nécessaire. Ceci est souvent une tâche critique au niveau du temps ayant besoin d'une optimisation soigneuse.

Synchronisation

Considérer deux processus SDL qui communiquent. Le processus émetteur peut envoyer un signal n'importe quand parce qu'il va être mémorisé dans la porte d'entrée du processus récepteur. Le processus récepteur peut alors consommer le signal un instant plus tard.

Ceci constitue un modèle de communication avec tampon (buffer) dans lequel l'émetteur peut produire une infinité de signaux sans attendre que le récepteur les ait consommés. Ceci est souvent référencé comme *communication asynchrone.*

La communication asynchrone peut être opposée à ce que l'on appelle la *communication synchrone* dans laquelle l'opération d'envoi et l'opération de consommation se déroulent en même temps. Ceci est nécessaire lorsqu'il n'y a pas de tampons entre les processus.

L'action d'alignement des opérations des différents processus concurrents en relation les uns avec les autres est généralement appelée *synchronisation*. La synchronisation est nécessaire non seulement pour assurer la correction de la communication, mais aussi pour contrôler l'accès aux ressources partagées dans le système physique.

---

5) Ceci modifie la sémantique du SDL de manière sensible, aussi il faut être prudent. Mais normalement ceci sera acceptable par l'utilisateur et beaucoup plus efficace.

La synchronisation des interactions peut être classée selon les catégories suivantes:

a) *Interaction synchrone,* où les processus exécutent des opérations d'interaction en même temps. Il y a deux sous-catégories:

   1) *Dépendant du temps* – Dans ce cas le moyen de transmission lui-même est synchrone comme par exemple les canaux dans un système de MIC. L'émetteur et le récepteur doivent rester en phase avec l'horloge commune du moyen de transmission. Ceci impose des contraintes de temps réel aussi bien pour l'émetteur que pour le récepteur.

   2) *Indépendant du temps*, ou explicitement synchronisé – Dans ce cas, une synchronisation explicite, indépendante du temps, a lieu entre les processus, au moyen duquel ils sont verrouillés ensemble pendant l'interaction. Les processus peuvent avoir à attendre que l'interaction soit mise en service. Une fois qu'elle est en service, l'interaction a lieu par des opérations exécutées simultanément par les deux processus. La communication en LOTOS appartient à cette catégorie. C'est un modèle orienté séquence.

b) *Interaction asynchrone,* où les processus n'exécutent pas nécessairement les opérations d'interaction en même temps. Néanmoins les opérations doivent être alignées en un certain ordre relatif de façon à assurer une interaction correcte. Pour la communication basée sur une valeur continue le principal besoin est que les opérations soient *mutuellement exclusives*. Pour la communication par séquence de valeurs il y a deux sous-catégories:

   1) *Dépendant du temps* – Le moyen de transmission lui-même est asynchrone, sans mécanisme de synchronisation explicite sur le moyen de transmission. Ce modèle dépend de la vitesse relative du récepteur comparée à celle de l'émetteur. L'émetteur doit produire les sorties à un niveau que le récepteur doit pouvoir suivre. Un exemple caractéristique est la numérotation par impulsions décimales sur les anciennes lignes d'abonnés au téléphone. Ce modèle est très utilisé dans les liaisons de communication physiques (et c'est une cause notoire de problèmes de temps réel).

   2) *Indépendant du temps* ou explicitement synchronisé – Dans ce cas, il y a un tampon et une synchronisation explicite indépendante du temps entre les processus en interaction. Le tampon est normalement une file d'attente FIFO (premier entré, premier sorti). L'emetteur met une (séquence de) valeur(s) (signaux) dans le tampon et le récepteur retire les (séquences de) valeurs un instant plus tard. La capacité du tampon détermine combien de valeurs l'émetteur peut produire en avance sur le récepteur. La communication par signaux SDL appartient à cette catégorie. Elle est très utilisée dans les systèmes logiciels, mais elle est moins répandue dans le matériel. L'interaction synchrone peut être considérée comme un cas particulier où la capacité du tampon est nulle.

La sémantique du SDL suppose une interaction asynchrone avec des tampons de capacité infinie. En pratique, cependant, la capacité des tampons doit être bornée d'une manière ou d'une autre.

Quelquefois l'application est telle que le nombre de signaux que le producteur est capable de produire en avance sur le consommateur va être toujours limitée. Dans les autres cas, la limitation dépend du fait que le consommateur soit suffisamment rapide pour éviter que les files d'attente ne deviennent trop longues. En général, on doit prendre en considération le problème du débordement de tampon en mettant le producteur en attente lorsque le tampon atteint sa capacité maximale.

En conséquence, il se peut que la sortie d'un processus SDL doive être retardée jusqu'à ce que le récepteur soit prêt. Cet écart de la sémantique du SDL peut être durement évité dans une mise en œuvre finie. L'ingénierie doit être faite avec soin pour réduire à un minimum les problèmes pratiques que ceci peut provoquer.

On va souvent trouver des mécanismes qui diffèrent de ceux du SDL aux interfaces physiques du système. Il est très caractéristique de trouver une interaction dépendant du temps sur les canaux physiques par exemple. Ceci implique qu'il faudra surveiller les événements critiques et générer des événements.

Un concepteur sera confronté d'un côté aux primitives de synchronisation disponibles dans le système réel et de l'autre à la synchronisation impliquée par la spécification SDL. Une fonctionnalité supplémentaire sera souvent nécessaire pour «recoller» ensemble les diverses formes.

Données

Les données SDL sont basées sur la notion de types abstraits de données, où les opérations peuvent être définies au moyen d'équations. Une mise en œuvre va normalement nécessiter des types concrets de données où les opérations sont définies de manière opérationnelle. Par conséquent, le concepteur peut avoir besoin de transformer les types abstraits de données de SDL en types de données plus concrets convenables pour la mise en œuvre.

I.7.3 Spécifications de mise en œuvre

Il découle de l'indépendance vis-à-vis de la mise en œuvre, qu'une et une seule spécification SDL peut être réutilisée dans différentes mises en œuvre. Aussi, il s'ensuit que les structures physiques de ces mises en œuvre peuvent varier considérablement. Dans une instance, chaque processus SDL peut être mis en œuvre sur une puce physique séparée, dans une autre tous les processus peuvent être du logiciel s'exécutant sur la même machine. La façon dont ceci est réalisé a besoin d'être documentée.

I.7.3.1 Justification

Les spécifications de mise en œuvre définissent la mise en correspondance entre un système SDL abstrait et le système concret construit avec des composants logiciels et matériels. Clairement, un système SDL peut être restructuré et raffiné jusqu'au point où il reflète une grande part de la conception. Mais cependant, le système SDL va rester abstrait et il sera possible de le mettre en œuvre de différentes façons. Par conséquent, on a besoin de quelque chose en complément du SDL pur pour définir la mise en œuvre.

Comment devrait être exprimée la spécification de mise en œuvre?

Une possibilité est d'ajouter les spécifications de mise en œuvre comme des annotations aux spécifications SDL. L'avantage de cette solution est que toutes les informations décrivant le système peuvent être trouvées en une seule place. L'inconvénient est que les spécifications SDL sont liées à des conceptions de mise en œuvre particulières. En décrivant les mises en œuvre séparément, il sera plus facile de réutiliser les spécifications SDL dans des systèmes ayant des mises en œuvre différentes. Il sera aussi plus facile de mettre en évidence des aspects particuliers de la mise en œuvre pendant l'activité de conception de la mise en œuvre.

Les aspects suivants sont à prendre en considération:

- la structure globale du système réel en termes de composants matériels et logiciels;
- les propriétés non fonctionnelles des composants réels et le système résultant. En particulier, ses performances en termes de capacité d'écoulement de trafic, de temps de réponse et de traitement d'erreur;
- l'établissement d'une correspondance entre le système SDL abstrait et les composants du système réel.

Il peut être avantageux de spécifier et d'analyser ces aspects de manière indépendante de la structure fonctionnelle des systèmes SDL. Les principaux critères de structuration pour les systèmes réels sont relatifs aux performances, au coût et aux propriétés physiques, tandis qu'au contraire les critères des spécifications SDL sont la clarté et la complétude du comportement. Ces critères sont si différents par nature qu'ils ne conduiront jamais à des structures semblables. En outre, il existe des aspects de systèmes réels qui sont suffisamment complexes par eux-mêmes pour justifier une spécification séparée.

Pour discuter des aspects de la conception de la mise en œuvre indépendamment des spécifications SDL sans aller dans les détails de mises en œuvre particulières, une notation pour les spécifications de mise en œuvre va être informellement introduite dans le paragraphe suivant.

I.7.3.2 Notation pour une spécification de mise en œuvre

La notation pour les spécifications de mise en œuvre se focalise sur les structures de matériel et de logiciel et les mises en correspondance entre les spécifications de plusieurs niveaux d'abstraction. Noter que cette notation n'est pas normative.

C'est une notation très générale pour les diagrammes de blocs. A de nombreux égards, cette notation est syntaxiquement similaire à celle des diagrammes d'interaction de blocs, mais il y d'importantes différences dans la signification.

Les boîtes et les flèches des diagrammes d'interaction de blocs SDL représentent des blocs abstraits et des canaux avec une sémantique bien définie. On peut les comprendre et les analyser avec leurs propres termes indépendamment de la réalisation.

L'objet d'une spécification de mise en œuvre est de définir la correspondance entre les spécifications SDL et la réalisation. Dans ce but, on n'a pas besoin d'une sémantique propre dans le sens où le SDL possède une sémantique. Sa signification provient de ce qu'elle représente dans le monde réel. Pour le monde réel lui-même nous avons d'autres formalismes, tels que les langages de programmation et les langages de description du matériel avec des sémantiques très bien définies. Par conséquent, la portée d'une spécification de mise en œuvre peut se limiter à une mise en correspondance syntaxique.

En représentant la structure de la réalisation grâce à une notation graphique, nous obtenons une vue d'ensemble et pénétrons dans la structure du système physique. Un exemple de structure de matériel est décrit à la Figure I.7-3.

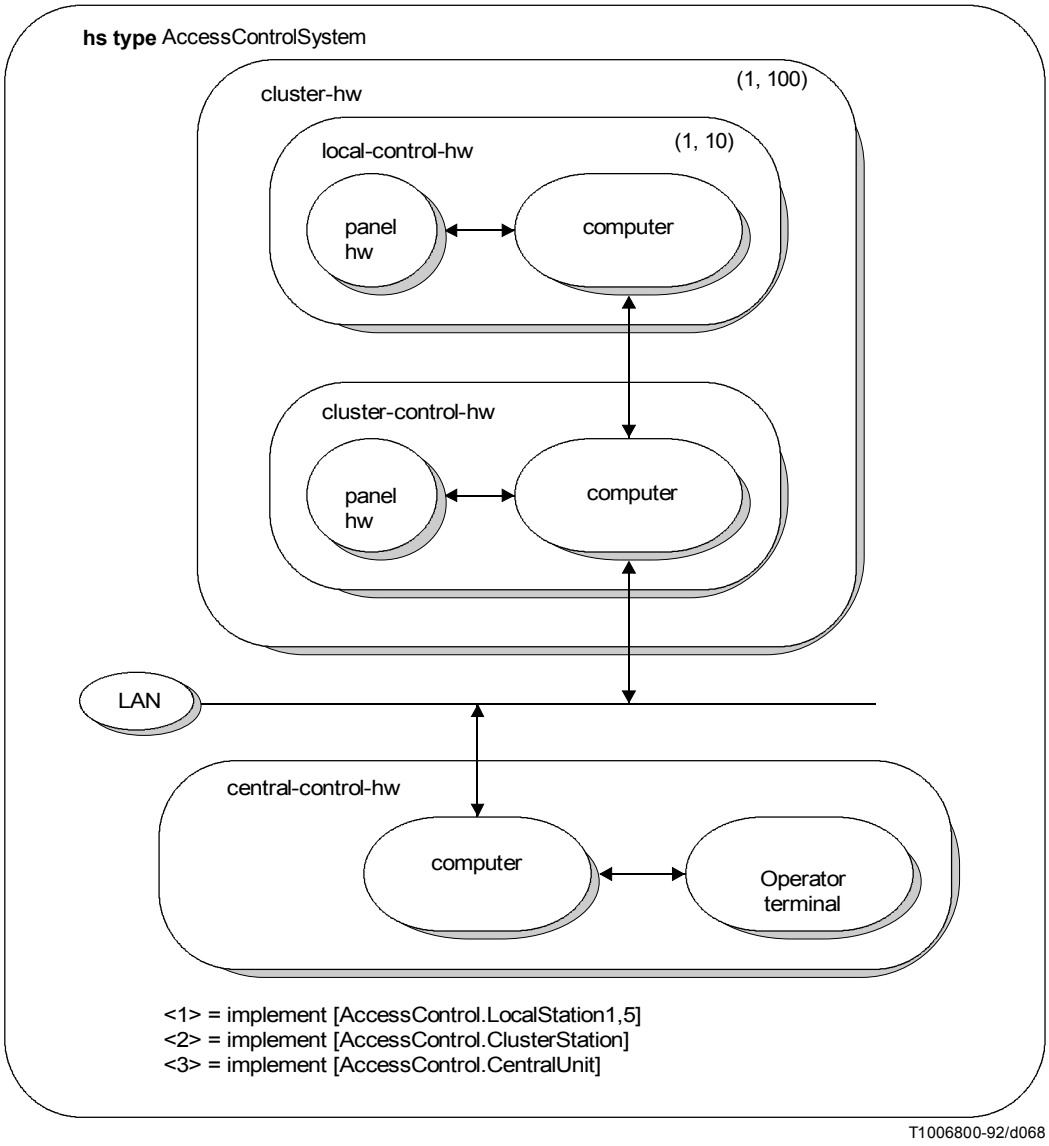


FIGURE I.7-3/Z.100

Un exemple de structure de matériel

Les boîtes représentent des unités concrètes de matériel telles que les ordinateurs et plaquettes. Les flèches représentent des connexions physiques, telles que les câbles. Avec cette notation, la spécification du matériel peut être décomposée pour fournir une approche progressive des détails. L'idée est d'utiliser cette notation pour définir la structure générale et puis de se référer à des notations spéciales au matériel telles que diagrammes de circuit pour les détails. De cette manière, on peut réaliser une spécification de matériel bien structurée. On trouvera la table des symboles pour les diagrammes relatifs au matériel à la fin de cette sous-section.

Les flèches indiquent la direction des signaux qui traversent une connexion. Les connexions bidirectionnelles sont possibles.

La notation pour les structures de logiciel se fonde sur les mêmes principes que ceux de la notation pour la structure du matériel, mais les boîtes ont des formes différentes. Ceci est utilisé pour distinguer de manière caractéristique différents types d'unités de logiciel, comme l'indique la Figure I.7-4.

La boîte à triples côtés représente le *processus logiciel* d'une unité de logiciel contenant au moins un programme ne se terminant pas. De tels programmes vont s'exécuter comme des processus quasi parallèles sur un système d'exploitation. Les boîtes à triples côtés représentent, par conséquent, des unités concurrentes. Les unités qui contiennent seulement des programmes qui terminent tels que les procédures se représentent par des boîtes à doubles côtés, et les données pures se représentent par des boîtes à côtés simples.

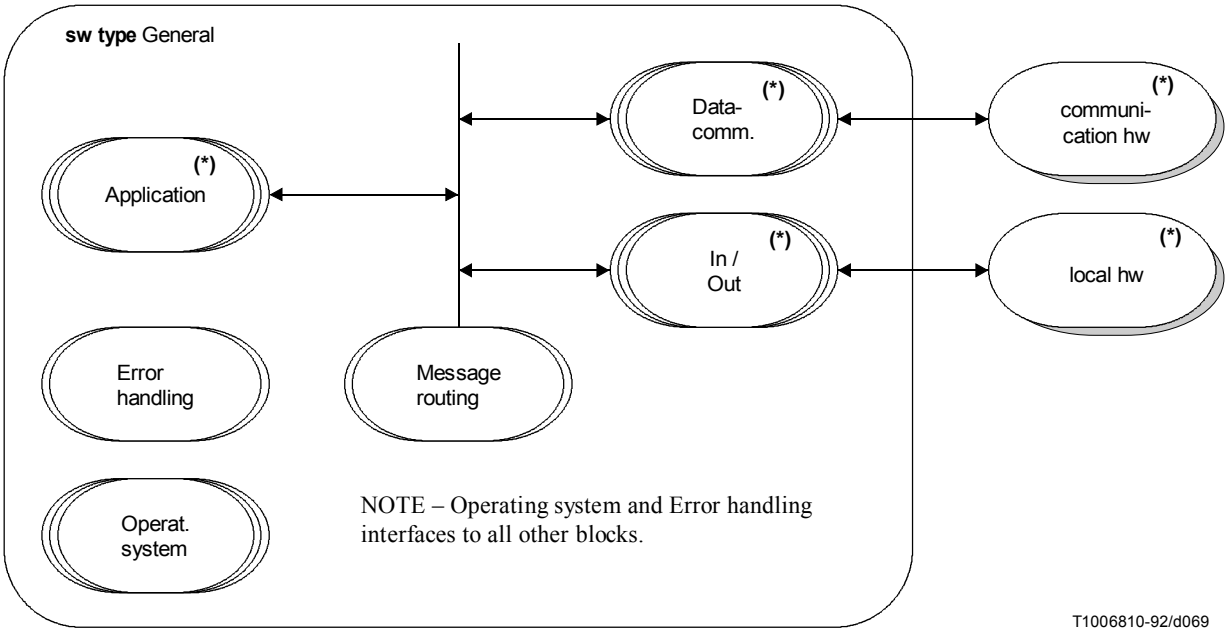


FIGURE I.7-4/Z.100

Un exemple de structure de logiciel

La structure du logiciel décrite à la Figure I.7-4 est très générale. Elle contient un certain nombre de processus logiciels, certains réalisent des fonctions de l'application tandis que d'autres réalisent des entrées, des sorties et communiquent en se transmettant des messages entre eux grâce à une procédure d'acheminement de messages.

La notation de structure de logiciel est utilisée pour donner une vue d'ensemble et fournir une approche progressive des détails du système logiciel. Cette notation combine les structures de données et les structures de programme dans une notation unifiée. Elle permet de représenter les relations structurées en réseau et peut exprimer la concurrence.

Les autres notations comme le pseudo-code ou le code source peuvent être utilisées pour décrire le système logiciel dans tous ses détails.

Parce que les relations sont difficiles à voir dans un texte de programme linéaire, il est utile de les représenter graphiquement. Puisque l'activation, ou le flux de contrôle, est très claire dans un texte de programme on considère que c'est la relation la moins importante à représenter dans la structure du logiciel. On fera ressortir les flots de données et les références.

Les flèches utilisées pour représenter les flux de données, les références et les activations entre les unités logicielles possèdent des formes différentes, comme le montre la vue générale des symboles à la fin de ce paragraphe.

Les diagrammes de structure de logiciel et les diagrammes de structure du matériel peuvent être décomposés hiérarchiquement. Ils possèdent des concepts semblables pour les types et les instances. Les deux sortes de diagrammes sont organisées comme la représentation graphique SDL avec un symbole de cadre représentant la frontière entre l'entité spécifiée et son environnement.

Dans le coin en haut et à gauche à l'intérieur du symbole de cadre, seront placés le type de diagramme et le nom de l'entité spécifiée:

hs type <nom> spécification d'un type de structure de matériel

ss type <nom> spécification d'un type de structure de logiciel

hs <nom> spécification d'une instance de structure de matériel

ss <nom> spécification d'une instance de structure de logiciel

Les boîtes et les flèches peuvent être rassemblées en ensembles ou tableaux. La taille d'un ensemble ou d'un tableau est indiquée par les nombres minimum et maximum d'éléments. Les nombres sont délimités par des parenthèses.

(<min>,<max>)	au moins <*min*>, au plus <*max*>
(<min>,)	au moins <*min*>, pas de limite supérieure
(+)	au moins un, pas de limite supérieure
(*)	n'importe quel nombre

Une boîte est identifiée par un nom de type et un nom d'instance facultatif:

lu:LocalUnit	instance *lu* de type *LocalUnit*
CentralUnit	type *CentralUnit*

Cette syntaxe est semblable à celle du SDL.

Les liens avec l'environnement sont indiqués par des flèches qui se prolongent jusqu'au symbole de cadre ou au-delà. Les entités et les liens dans l'environnement peuvent être représentés en dehors du symbole de cadre et être reliés aux entités situées à l'intérieur du cadre.

Les spécifications de mise en œuvre spécifient le système réel qui est une mise en œuvre du système SDL (voir la Figure I.7-5).

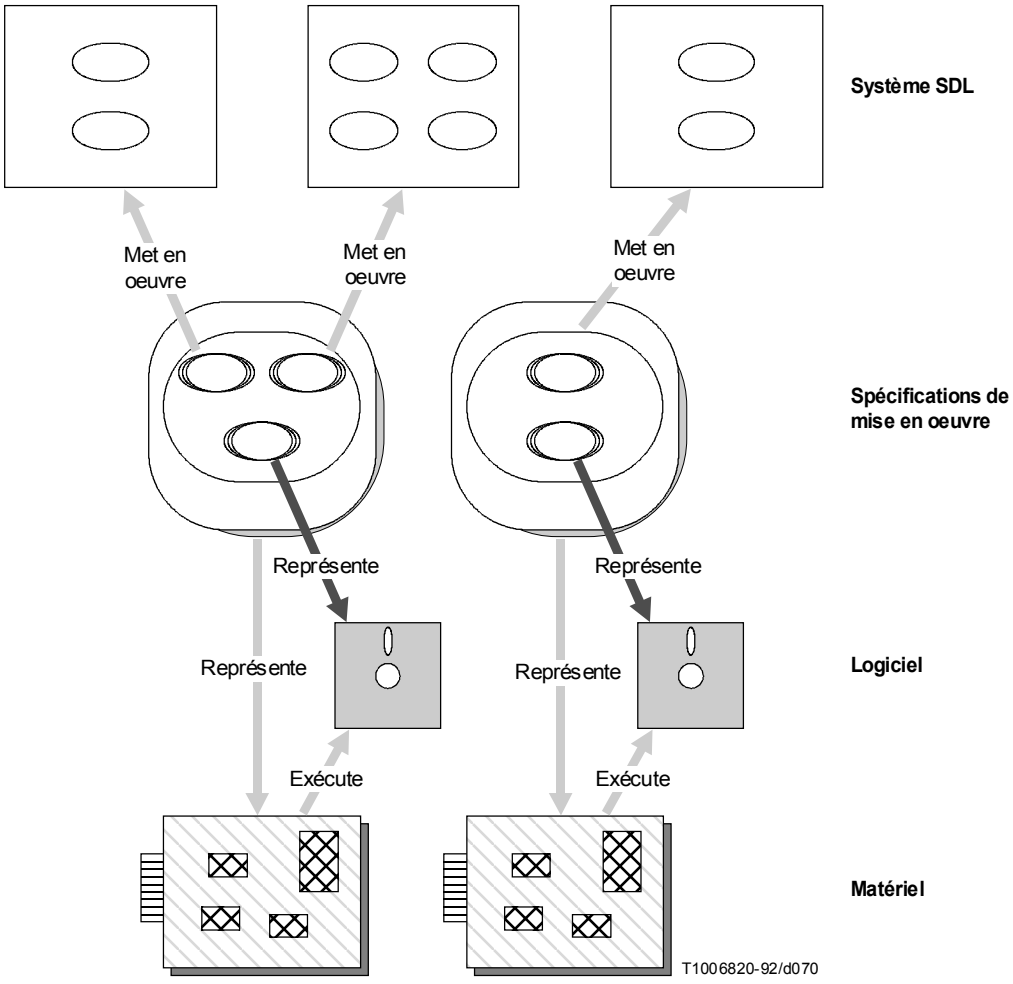


FIGURE I.7-5/Z.100

Les spécifications de mise en oeuvre spécifient le système réel

De façon à spécifier la mise en correspondance entre le système SDL et le système réel, il est nécessaire d'ajouter des informations de mise en correspondance entre les diagrammes de structure du matériel et du logiciel. Il y a plusieurs mises en correspondance à prendre en compte:

- Le code source *est traduit* de la spécification source.
- Le code objet *est compilé* à partir du code source.
- Le code exécutable *est chargé* à partir du code objet.
- Le matériel (physique) *exécute* le code exécutable.
- Le matériel (physique) + le code exécutable *mettent en œuvre* la spécification source.

Pour contrôler et documenter complètement la conception et la mise en œuvre, il faut considérer tous les niveaux. Pour la plupart des objectifs pratiques de la documentation, certains niveaux peuvent être omis.

Les boîtes et les flèches dans les diagrammes de mise en œuvre se rapportent d'une part aux expressions SDL et de l'autre aux objets du monde réel qu'ils représentent. Ceci est spécifié au moyen d'*expressions de mise en correspondance* dans le diagramme, comme l'illustre la Figure I.7-3.

Pour économiser l'espace dans les boîtes et sur les flèches, les expressions de mise en correspondance peuvent être référencées :

<1> référence de mise en correspondance

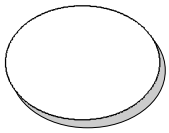
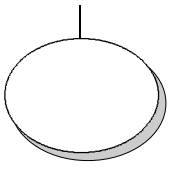
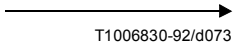
<1> = implement [**system** AccessControl.LocalStation] expression de mise en correspondance référencée

Les types de mise en correspondance suivants sont utilisés (l'*identificateur de nœud* identifie l'entité considérée):

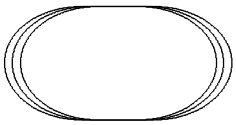
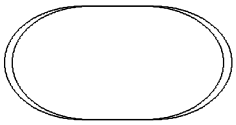
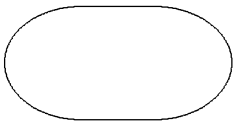
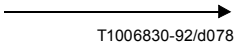
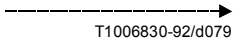
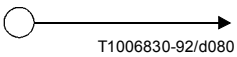
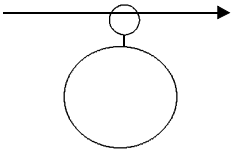
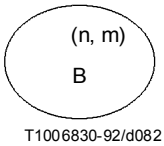
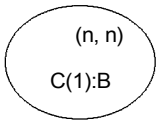
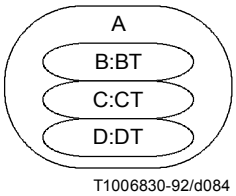
- Met-en-œuvre [identificateur de nœud]
- Est-mis-en œuvre-par [identificateur de nœud]
- Exécute [identificateur de nœud] (implique que le logiciel soit chargé)
- Est-exécuté-par [identificateur de nœud]

I.7.3.3 Résumé des symboles

Symboles des diagrammes de structure du matériel

 T1006830-92/d071	Bloc matériel	Symbole général pour un bloc matériel. Peut être décomposé récursivement.
 T1006830-92/d072	Commutateur	Symbole général pour un bloc matériel qui exécute la commutation ou l'acheminement. Le symbole peut être incliné dans n'importe quelle direction, et la ligne simple peut être allongée pour être attachée à de nombreux liens matériels.
 T1006830-92/d073	Connexion (flèches facultatives)	La flèche de la ligne de connexion est facultative. Les flèches montrent la direction des signaux. On peut avoir des flèches aux deux bouts. Au niveau le plus bas, les connexions sont des liaisons physiques.
 T1006830-92/d074	Ensemble de blocs	Les blocs de matériel peuvent être groupés en un ensemble de blocs de même type. Les blocs devraient avoir un type et peuvent avoir un identificateur d'instance.

Symboles de diagrammes de structure de logiciel

 T1006830-92/d075	Bloc de processus logiciel	Une unité de logiciel contenant au moins un programme ne se terminant pas. Peut contenir des procédures et des données.
 T1006830-92/d076	Bloc de procédure	Une unité de logiciel contenant au moins un programme se terminant mais aucun programme ne se terminant pas. Peut contenir des données.
 T1006830-92/d077	Bloc de données	Un élément de données pures ou un groupe d'éléments de données. Ne peut pas contenir de programmes.
 T1006830-92/d078	Flux de données	Flèche représentant la direction du flux.
 T1006830-92/d079	Activation	Activation ou appel. Ne peut être connecté aux blocs de données.
 T1006830-92/d080	Référence	Référence ou pointeur.
 T1006830-92/d081	Flux de données par référence à un tampon de messages	La ligne représente le flux de données et le carré référencé représente le tampon des messages.
 T1006830-92/d082	Ensemble d'éléments de données de type B	Un ensemble d'au moins *n*, et au plus *m* d'éléments de type B.
 T1006830-92/d083	Tableau indexé d'éléments de données	Un tableau de *n* éléments appelés C(1)-C(n) de type B.
 T1006830-92/d084	Structure d'éléments de données; enregistrement	Une structure de données composite (A) comprenant des éléments de types différents. Chaque donnée a un identificateur de type et un identificateur d'instance.

I.7.4 Compromis entre matériel et logiciel

I.7.4.1 Introduction

Ces directives mettent principalement l'accent sur la conception du logiciel. Mais ni le logiciel ni le matériel ne peuvent être complètement conçus indépendamment l'un de l'autre. La capacité à respecter les contraintes de temps, par exemple, dépend à la fois de la vitesse du calculateur et de la quantité de logiciel nécessaire pour réaliser une fonction donnée. En fait, il y a une influence réciproque et il faut faire de nombreux compromis entre le matériel et le logiciel.

Une fois qu'on a sélectionné une structure adéquate de matériel, on peut alors considérer les conséquences sur la conception du logiciel. Le matériel périphérique, par exemple, a des conséquences sur le logiciel d'entrée-sortie. Si les fonctions sont réparties sur plusieurs calculateurs, nous aurons besoin de logiciel supplémentaire pour assurer la communication entre les machines et le traitement d'erreur réparti. Par conséquent, avant de concevoir tout le logiciel, il est nécessaire de connaître la structure générale du matériel.

Le matériel ne devrait pas être conçu sans considérer aussi la structure du logiciel. Par conséquent, une coordination et des compromis entre la conception du matériel et du logiciel sont nécessaires. La première étape de la conception de la mise en œuvre est de réaliser ce compromis et cette conception de l'architecture globale du matériel et du logiciel en prenant en compte les besoins non fonctionnels. Les considérations les plus importantes sont :

- la répartition physique et les interfaces physiques;
- les contraintes de temps vis-à-vis de la capacité de traitement;
- les besoins relatifs au traitement d'erreurs, par exemple la détection, l'isolement et le recouvrement des erreurs;
- la sécurité contre les accès non autorisés aux informations;
- l'extensibilité, l'exploitation et la maintenance du matériel;
- le coût de développement, le coût de production, le coût de maintenance;
- l'utilisation de composants existants.

I.7.4.2 Répartition physique et interfaces

La spécification SDL ne devrait pas supposer prématurément une répartition physique interne du système. Mais quelquefois, l'emplacement physique des interfaces implique une répartition physique du système. Les abonnés à un système télématique, par exemple, seront physiquement dispersés. En conséquence, au moins les interfaces utilisateur seront physiquement dispersées.

Les signaux SDL sont définis indépendamment des distances physiques. On est, par conséquent, libre de séparer physiquement les processus. Mais il y aura toujours un certain temps de transmission et un coût associé à un transfert de signal sur certaines distances. On devrait par conséquent considérer les canaux véhiculant un trafic faible sans contraintes de temps strictes. De tels canaux peuvent quelquefois se trouver aux interfaces externes du système, mais le plus souvent ils se trouveront à l'intérieur du système.

Ceci se généralise en une règle qui dit que nous devrions répartir le long des canaux avec de faibles interactions et des contraintes de temps peu sévères (bande passante étroite). Nous devrions garder les processus fortement couplés ensemble. Ceci va souvent signifier qu'une bonne partie du traitement devrait être physiquement réalisée très près des interfaces externes.

En conséquence, la mise en œuvre peut répartir le système SDL en blocs et canaux physiquement distincts. Une fonctionnalité supplémentaire est probablement nécessaire pour supporter la répartition, par exemple des protocoles de transfert de signaux, de traitement d'erreurs.

I.7.4.3 Charge de trafic et temps de réponse

Le but est de trouver le matériel et le logiciel qui peuvent répondre à la charge de trafic et aux besoins en temps de réponse à un coût minimal. Dans ce but, nous avons besoin d'estimer les temps de traitement et ensuite de calculer la charge de traitement représentée par l'application SDL et ses temps de réponse.

Les calculs de moyennes simples peuvent se réaliser de la manière suivante :

a) pour chaque processus SDL, P, calculer le temps moyen de transition t_p. Cette situation dépend de la taille du logiciel et de la vitesse d'exécution du matériel et va devoir être estimée. On peut, par exemple, calculer le nombre moyen d'opérations o_p par transition (envoi de signaux, opérations concernant les temporisateurs, opérations concernant les données) et estimer ensuite un nombre moyen d'instructions i_p par opération. Si la vitesse d'exécution du matériel est de S instructions par seconde, alors $t_p = i_p * o_p * S$;

b) calculer le nombre moyen de transitions n_p que chaque processus va exécuter par seconde à l'heure de pointe. Cette situation peut se déterminer en comptant le nombre de transitions nécessaires pour réaliser une fonction donnée, par exemple la prise en charge d'un appel téléphonique, multipliée par le nombre de telles fonctions que le processus devra exécuter par seconde;

c) calculer la charge normalisée, à l'heure de pointe, pour chaque processus:

$$l_p = n_p * t_p$$

Ceci constitue la mesure de la quantité de temps dont aura besoin le processus par seconde, c'est-à-dire, la fraction de la capacité de traitement nécessaire au processus (mesure d'Erlang);

d) calculer la situation de charge correspondante pour chaque canal *C* et acheminement de signal *R* :

$$l_c = n_c * t_c$$

$$l_r = n_r * t_r$$

Où n_c et n_r sont les nombres de signaux par seconde, et t_c et t_r les temps de traitement par transfert de signal;

e) calculer la charge moyenne du système comme la somme des charges correspondant aux canaux, acheminements de signaux et processus. Si cette valeur est plus grande que un, la charge moyenne est supérieure à la capacité de traitement d'un simple calculateur. Dans ce cas, la capacité doit être augmentée soit en optimisant du logiciel, soit en augmentant la vitesse du matériel, soit en répartissant le système. Comme règle empirique, il est bon de limiter la charge d'un seul calculateur à 0,3 de manière à laisser la place aux pointes de charge statistiques.

Remarquer que la charge réelle va varier statistiquement et va comporter des pointes dont les valeurs vont être considérablement plus hautes que les valeurs moyennes que nous avons considérées. Le système peut, par conséquent, être surchargé pendant des laps de temps bien que la charge moyenne soit assurée avec une bonne marge. Il faudra néanmoins qu'une stratégie de contrôle de charge fasse partie de la conception de la mise en œuvre.

Si le système peut s'exécuter sur un seul calculateur, cette solution devrait être préférée (à moins que d'autres considérations exigent quelque chose d'autre). S'il ne peut s'exécuter sur une seule machine pour des raisons de performances, il doit être réparti. Ceci va ajouter une charge supplémentaire qui doit être prise en compte dans des données de charge révisées.

En plus des calculs de charge ci-dessus, il est nécessaire de vérifier que le système va respecter les contraintes temps réel. Le nombre de transitions et de transferts de signaux par transaction à durée critique doit être déduit des diagrammes SDL, et les temps de traitement correspondants doivent être calculés. Encore une fois, il est nécessaire d'admettre une tolérance pour les variations statistiques.

L'interface matériel/logiciel nécessite une considération particulière. Il n'est pas habituel que la plus grande partie de la capacité d'un calculateur soit dépensée à faire des entrées-sorties. Par conséquent, on peut obtenir beaucoup grâce à une conception soignée de l'interface d'entrée-sortie. Une classe particulière de contraintes de temps provient des canaux assurant une synchronisation dépendant du temps, voir I.7.2.2.

I.7.4.4 Fiabilité, sûreté et sécurité

Les besoins de fiabilité peuvent avoir un impact sur la structure du matériel de différentes manières:

- l'insensibilité aux défaillances signifie redondance: au moins deux unités matérielles, et les facilités de détection d'erreur, de diagnostic et de commutation sont nécessaires pour mettre en œuvre la tolérance aux pannes;
- la segmentation des défauts signifie répartir les fonctions sur des unités matérielles séparées de manière à limiter le nombre de processus SDL qui peuvent être bloqués par une seule panne de matériel;
- la sûreté en cas de panne signifie que le système doit tomber en panne toujours à un état de sortie sûr où il ne fait aucun tort à son environnement. Il faut normalement se servir d'un dispositif matériel de supervision.

La sécurité contre les accès non autorisés ou la modidification des informations peut aussi exiger quelques mesures particulières au sein du matériel.

I.7.4.5 Modularité et réutilisation

La conception de matériel, plus que la conception de logiciel, doit prendre en considération le coût de production. Ceci va dépendre à la fois de la complexité du matériel lui-même et du volume de la production. Un grand volume va signifier à un coût unitaire plus bas. On a besoin, par conséquent, de minimiser le nombre des différentes conceptions de manière à profiter d'un grand volume de production. On va souvent être capable de trouver une conception de matériel générique qui peut être utilisé pour mettre en œuvre une large gamme de systèmes SDL.

I.7.4.6 Architecture du matériel

A partir des considérations mentionnées ci-dessus, l'architecture globale du matériel nécessaire à la mise en œuvre du système SDL devrait être définie. Celle-ci peut être documentée au moyen de diagrammes de structure de matériel, comme l'illustre la Figure I.7-3. Ce qu'il ressort à cette étape est la structure globale du matériel en termes de calculateurs, équipements périphériques et canaux de communication.

Les protocoles, les formats des signaux et les modèles de synchronisation utilisés sur toutes les interconnexions physiques devraient être définis, car ceci constitue un élément important pour la conception du logiciel.

Finalement, devrait être documentée l'allocation des processus SDL aux unités physiques. Une fois cela réalisé, nous connaissons les fonctionnalités aussi bien que l'environnement physique du logiciel dans le système.

I.7.5 Conception de l'architecture du logiciel

I.7.5.1 Principes de conception

Il y a des différences sémantiques considérables entre le SDL et la plupart (si ce n'est tous) des langages de programmation:

a) **Concurrence** – Les langages de programmation séquentielle comme C et PASCAL ne fournissent aucun support pour la concurrence de SDL. Certains langages comme CHILL et ADA admettent la concurrence mais la traitent différemment du SDL.

b) **Temps** – Très peu de langages de programmation traitent le temps complètement. Le temps comme le considère le SDL n'est traité par aucun langage, excepté peut-être par CHILL?

c) **Communication** – La communication par signaux comme la considère le SDL n'est supportée par aucun langage.

d) **Comportement séquentiel** – Un graphe de processus SDL spécifie un comportement par états/ transitions à la manière d'une machine à états finis étendue. Les langages de programmation spécifient des séquences d'actions.

e) **Données** – Les données SDL sont abstraites et éventuellement infinies. La mise en œuvre dans un langage de programmation doit être opérationnelle et finie.

Une manière de surmonter de telles différences est d'adapter la machine sous-jacente et le langage de programmation à la sémantique du SDL au moyen de logiciel de support. Trois niveaux de support sont d'un usage commun, comme le montre la Figure I.7-6:

a) Aucun – Les concepts SDL sont mis en correspondance directement avec les concepts supportés par le langage de programmation séquentiel.

b) Des facilités de base pour supporter la concurrence, le temps et la communication sont fournies par un système d'exploitation temps réel – celui-ci comprend également le support d'exécution fourni pour les langages concurrents comme CHILL et ADA.

c) Support supplémentaire pour les concepts du SDL au sommet de b).

Bien que l'utilisation d'un système d'exploitation soit l'approche la plus commune aujourd'hui, il y a des cas où le temps système introduit est inacceptable soit pour des raisons de contraintes de vitesse, soit de contraintes de taille mémoire ou soit des contraintes de coût.

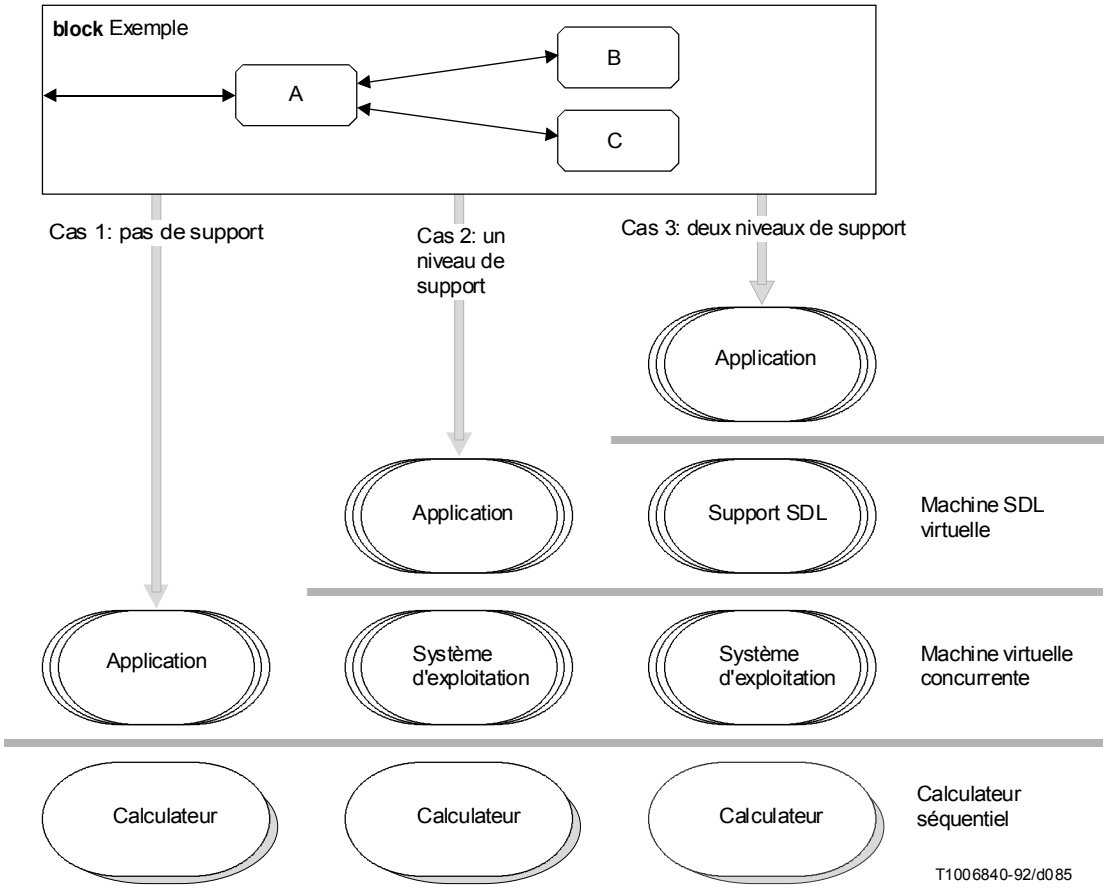


FIGURE I.7-6/Z.100

Niveaux de logiciel de support

I.7.5.2 Concurrence et temps

Niveaux de soutien

Lorsque plusieurs processus SDL sont mis en œuvre sur le même calculateur en utilisant un langage de programmation séquentiel comme C, la concurrence d'accès du SDL doit être approchée par un comportement séquentiel. Ceci peut, en principe, être obtenu en transformant une spécification SDL contenant plusieurs processus en une spécification équivalente n'en contenant qu'un seul, et ensuite en mettant en œuvre ce processus comme un programme séquentiel. En raison du problème de l'«explosion des états» et du manque de modularité de cette approche, celle-ci n'est applicable que dans des cas très restreints. Par conséquent, on va normalement rechercher une mise en œuvre qui garde la structure de processus originelle de la spécification SDL. Ceci implique qu'il doit y avoir un certain support pour l'ordonnancement des processus et leur communication.

Dans un langage purement séquentiel, il n'y a pas de support général pour la concurrence. Mais les appels de procédure fournissent une communication et un mécanisme d'ordonnancement combinés qui peuvent mettre en œuvre des cas particuliers de communication SDL et de concurrence. (La communication asynchrone du SDL est alors mise en œuvre au moyen de la communication synchrone.) Le niveau le plus bas du support est par conséquent d'utiliser les appels de procédure comme mécanisme de base pour l'ordonnancement et la communication. Cette approche est développée dans I.7.5.3.

Pour mettre en œuvre des systèmes SDL plus généraux, il est nécessaire d'introduire un modèle de communication asynchrone avec tampon. Ceci peut être obtenu à l'intérieur du cadre d'un programme purement séquentiel utilisant, par exemple, un programme principal pour organiser l'activation des procédures mettant en œuvre les processus SDL qui communiquent par des tampons de messages. La limitation de cette approche réside dans sa capacité à traiter le temps et les contraintes temps réel.

Un système d'exploitation à vocation générale qui assure l'ordonnancement des processus concurrents selon des règles de priorité, supporte la communication et gère le temps tels qu'ils sont dans le SDL, va fournir la plate-forme la plus facile et la plus générale pour mettre en œuvre la concurrence des systèmes SDL. Les facilités de base exigées du système d'exploitation temps réel sont les suivantes:

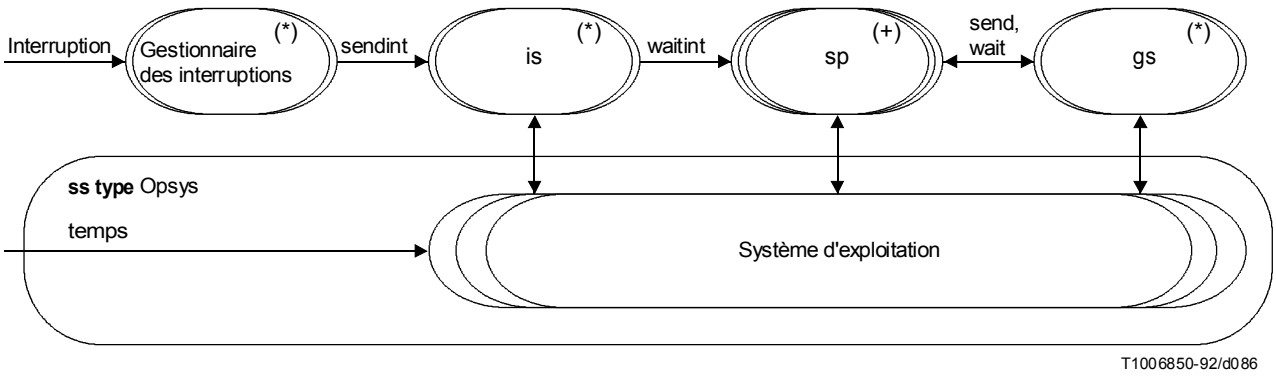
- Multiplexage et ordonnancement des processus, c'est-à-dire
  1) commutation de contexte entre les processus;
  2) ordonnancement avec priorité préemptive et non préemptive pour respecter les contraintes relatives aux temps de réponse;
  3) traitement d'interruptions pour fournir de l'attente passive sur les événements externes.
- Synchronisation des interactions, c'est-à-dire
  1) communication;
  2) accès aux ressources partagées.
- Mesure du temps.

Ces facilités sont fournies d'une façon ou d'une autre par beaucoup de systèmes d'exploitation commerciaux. Elles sont également supportées par des langages de programmation concurrente comme CHILL et ADA. Un petit système d'exploitation qui supporte ces fonctions sera décrit dans la suite pour fournir un exemple et un cadre de référence.

Au sommet du système d'exploitation de base, on aura besoin de facilités pour la prise en compte du temps et pour la communication «à la SDL» de façon à apporter un soutien général pour le SDL.

Un exemple de système d'exploitation

Le système d'exploitation voit le logiciel comme une collection de *processus logiciels* et de *sémaphores* généraux. Il effectue le multiplexage et l'ordonnancement des processus sur la base des événements externes et internes. Ces événements sont soit des interruptions externes (comprenant les interruptions liées au temps) ou des opérations internes sur les sémaphores (voir la Figure I.7-7).



sp Processus logiciel (*Software process*)
gs Sémaphore général (*General semaphore*)
is Sémaphore d'interruption (*Interrupt semaphore*)

FIGURE I.7-7/Z.100

Un exemple de système d'exploitation

Les sémaphores gèrent les tampons et les processus en attente. Les tampons sont utilisés à la fois pour la communication de messages et pour représenter les ressources partagées allouées par un sémaphore. Les tampons sont également des ressources partagées. Avant qu'un message puisse être envoyé, un tampon libre doit être alloué à partir d'un pool de tampons libres. Le sémaphore général est un mécanisme qui peut être utilisé à la fois pour allouer des tampons libres, représentant éventuellement quelques autres ressources et pour fournir la communication asynchrone.

Les opérations qui portent sur les sémaphores généraux sont les suivantes:

- **send**(semaphore, buffer);
- **wait**(semaphore, max-time) –> (buffer, time).

L'opération *wait* peut spécifier un temps maximal d'attente. Si aucun tampon n'est rendu disponible dans le délai imparti, l'opération va retourner une indication de débordement de temporisation. En attendant sur un sémaphore particulier, *suspend*, qui ne renvoie jamais de tampon, le processus peut s'interrompre lui-même pendant un temps déterminé.

On utilise un type particulier de sémaphores pour signaler les interruptions et pour attendre les interruptions:

- **sendint**(semaphore);
- **waitint**(semaphore, max-time) –> (time).

Le sémaphore général peut être vu comme un type abstrait de données avec deux opérations *send* et *wait*, mises en œuvre par des appels de procédures. Il va utiliser une structure de données, c'est-à-dire une liste liée, pour contenir une file d'attente de tampons. Il va aussi garder une file d'attente des références aux processus logiciels en attente dans le cas où il n'y aurait plus de tampons dans la file d'attente des tampons (voir la Figure I.7-8).

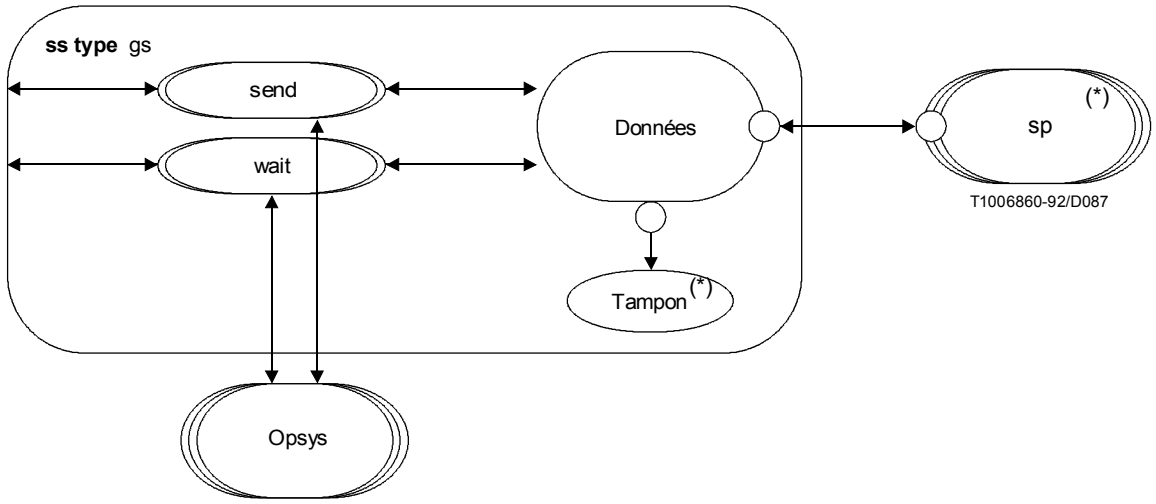


FIGURE I.7-8/Z.100

Le sémaphore général

Cette sorte de sémaphore est une généralisation des sémaphores classiques décrits dans la littérature [17]. Les tampons de messages ou les tampons libres sont placés dans la file d'attente par une opération *send* et sont retirés par une opération *wait*.

Afin de mettre en œuvre la communication de type SDL, les tampons peuvent être codés avec des signaux SDL.

La gestion du temps SDL est différente de l'interruption des processus, puisque un processus SDL peut être actif en exécutant des transitions pendant que se déroule une temporisation. Pour permettre ceci, le système d'exploitation fournit une facilité de gestion de temporisation de type SDL:

- **starttimer**(time, timer-id, PId) –> message(PId, timer-id);
- **stoptimer**(timer-id, PId);
- **now** –> time.

La facilité de gestion de temporisation agit essentiellement sur une liste de temporisateurs et les processus associés en attente de réception d'un message de débordement de temporisation. En utilisant *starttimer*, un processus demande en réalité la facilité de gestion de temporisation pour s'inscrire dans sa liste de temporisateurs et pour commencer à décompter le temps. Lorsque le temps spécifié est atteint, le message de débordement de temporisation est envoyé et l'inscription est retirée de la liste.

L'opération stoptimer va de manière similaire demander que l'entrée soit retirée de la liste. Que va-t-il se passer si une commande *stoptimer* est effectuée, juste après que le débordement de temporisation soit envoyé au processus? Le processus va alors être dans un nouvel état dans lequel il ne va pas attendre le débordement de temporisation, tandis que dans la liste des temporisateurs il n'y aura plus aucun compteur à retirer. La meilleure solution est de laisser *stoptimer* retirer le message de débordement de temporisation quel que soit l'endroit où il se trouve dans ce cas.

Chaque processus logiciel possèdera une description de processus que le système d'exploitation utilisera pendant l'ordonnancement et la commutation de contexte. Quelques informations de cette description de processus sont présentées à la Figure I.7-9.

L'ordonnanceur va conserver une liste des processus prêts à s'exécuter, et va lancer toujours celui qui possède la priorité la plus haute. (En cas de priorités égales, il doit choisir en alternance.) Toutes les fois qu'une opération *send* ou *wait* est exécutée, un processus peut devenir prêt à s'exécuter, et/ou le processus courant peut devoir attendre. Chaque opération de sémaphore peut par conséquent provoquer l'activation d'un nouveau processus et l'arrêt du processus courant. Les interruptions peuvent avoir le même effet par l'utilisation des opérations *sendint* et *waitint*. De plus, les interruptions d'horloge peuvent provoquer des débordements de temporisations.

En plus de l'ensemble des opérations de base déjà décrites, on peut avoir besoin dans certaines applications d'opérations pour créer et détruire dynamiquement des processus et des sémaphores.

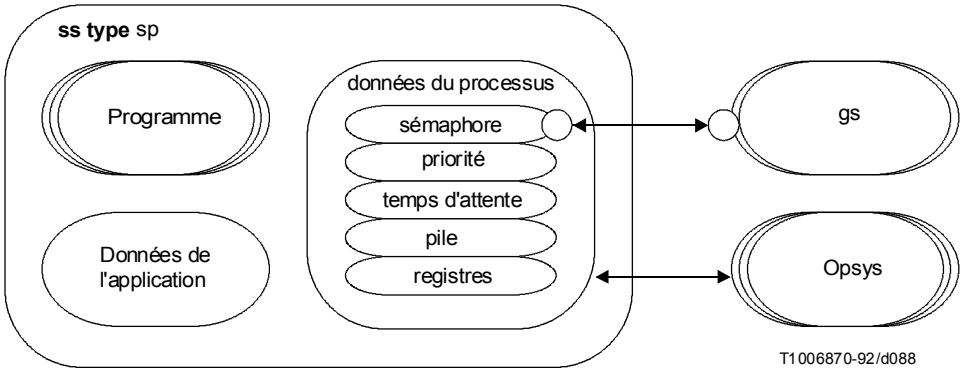


FIGURE I.7-9/Z.100

Un processus logiciel

Priorité

Dans le SDL il n'y a pas de priorité entre les processus. Mais ceci ne veut pas dire que l'on ne peut pas utiliser la notion de priorité dans la mise en œuvre. On a besoin de la notion de priorité pour diverses raisons:

- il y a des contraintes temps réel externes qui ne peuvent être tenues qu'en donnant une forte priorité à certains processus;
- il existe des situations d'erreurs internes qui devraient être traitées avec la plus grande priorité;
- il existe des situations où la priorité aide à simplifier et à accélérer le calcul.

Normalement, les contraintes temps réel doivent être respectées en donnant une priorité aux processus à durée critique. Par conséquent, il faut normalement traiter les entrées/sorties à durée critique avec une priorité élevée, tandis que le traitement interne peut se faire avec une priorité plus basse. C'est une des raisons pour laquelle les entrées/sorties devraient être séparées des processus de l'application.

Dans les situations de faute on doit réagir rapidement de façon à isoler l'erreur, à réduire les dommages et donner l'alerte. Ceci signifie qu'il faut réaliser immédiatement un certain traitement et qu'il faut rendre compte de l'erreur aussi rapidement que possible. De façon à établir ce compte rendu rapidement, il ne suffit pas que les processus aient une forte priorité. Nous avons besoin également de priorité sur le transfert de messages. Ceci peut être obtenu dans le système d'exploitation soit en donnant à chaque message un attribut de priorité, ou en envoyant des messages à haute priorité via les sémaphores qui sont traités avec une haute priorité.

Les services SDL agissent de manière quasi parallèle et communiquent au moyen de signaux prioritaires. Ceci peut être mis en œuvre en donnant des priorités égales aux services et laisser leurs signaux internes avoir une priorité supérieure à celle des signaux externes.

Si les messages internes à l'intérieur d'un système logiciel ont priorité sur les messages externes en provenance de l'environnement, l'ordre de traitement interne va devenir plus déterministe. Ceci permet de réduire le nombre de cas où les processus vont réellement s'entrelacer. Ceci réduit également la probabilité que se produisent certaines erreurs d'interaction. Plus important encore, ceci aide quelquefois à réduire le temps système et accroît la vitesse des communications internes. Finalement, on améliore le contrôle de charge si les demandes de service déjà acceptées ont la priorité sur les demandes de nouveaux services.

I.7.5.3 Communication

Dans le compromis entre le matériel et le logiciel (voir I.7.4) nous avons commencé par considérer les interfaces physiques du système et nous avons ensuite abordé l'architecture interne. Dans la conception du logiciel nous allons faire un peu la même chose en commençant par les interfaces matériel-logiciel et nous allons traiter progressivement la structure interne du logiciel.

Le format de communication utilisé aux interfaces physiques a une influence importante sur la structure globale du logiciel. La communication interne et externe a besoin également d'agir sur la façon dont la communication peut être mise en œuvre à l'intérieur d'un système logiciel. Elle agit également sur les besoins concernant l'ordonnancement, la préemption et la synchronisation.

Entrées/sorties

Il faut souvent réaliser des conversions à l'interface entre le matériel et le logiciel, pas seulement pour convertir des données d'un format en un autre, mais aussi pour faire des conversions entre valeurs continues et séquences de valeurs. On peut, par exemple, avoir besoin de parcourir les variables externes à certains intervalles pour détecter les événements et engendrer des messages de type SDL pour la communication interne.

Les vraies restrictions liées à la concurrence d'accès et à la gestion du temps présentes à l'interface entre le matériel et le logiciel doivent s'harmoniser avec le quasi-parallélisme et les priorités dans le système logiciel. Une haute priorité est normalement nécessaire pour les interactions d'entrées/sorties de façon à:

- s'assurer que tous les événements d'entrée sont détectés en cas d'interaction dépendant de la vitesse;
- faire un usage efficace des canaux d'entrées/sorties à bas débits;
- réduire les temps de réponse.

Les interruptions sont les moyens fondamentaux pour fournir une priorité préemptive et pour permettre l'attente passive sur les événements externes. L'attente passive économise les ressources de calcul, mais les interruptions introduisent un ordre d'exécution non déterministe dans le système logiciel, qui rend les conditions similaires à celles d'un traitement vraiment parallèle. En conséquence, la synchronisation est nécessaire sur les interactions internes entre les unités de logiciel interrompues et interrompantes (d'où les sémaphores d'interruption).

A cause des priorités et de la vitesse nécessaire pour répondre aux besoins du temps réel et des performances, les entrées/sorties doivent souvent être mises en œuvre par des processus logiciels séparés comme l'illustre la Figure I.7-10.

Ceci sert à cacher les particularités de l'interface d'entrée/sortie de la partie applicative mettant en œuvre les processus SDL. La mise en œuvre de processus SDL est discutée au I.7.5.4.

A la Figure I.7-10 la communication de type SDL est mise en œuvre par transfert de messages via des sémaphores. Mais ce n'est pas la seule façon possible de le faire.

Appels de procédure

Les appels de procédure peuvent être utilisés pour mettre en œuvre directement les signaux SDL. Chaque type de signal peut être représenté comme une procédure appartenant au processus récepteur. Le signal «Open_door(here)» par exemple, peut être mis en œuvre par la procédure OPEN_DOOR(HERE). Ceci implique que le processus SDL récepteur soit mis en œuvre par un ensemble de procédures et de données.

A titre d'exemple, considérons les trois processus SDL *P1*, *P2* et *P3* représentés à la Figure I.7-11.

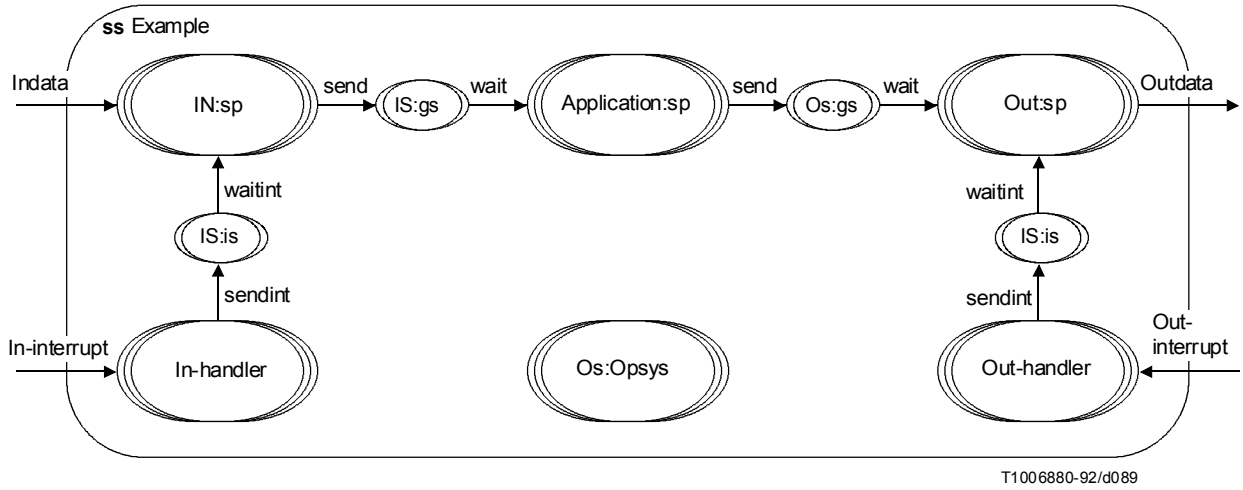


FIGURE I.7-10/Z.100

Mise en oeuvre des entrées/sorties par des processus logiciels séparés

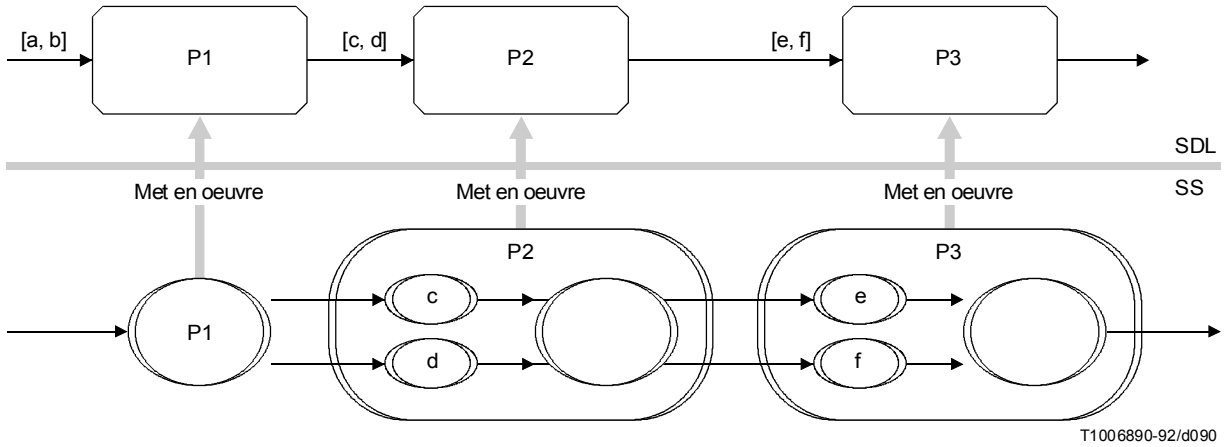


FIGURE I.7-11/Z.100

Signaux mis en oeuvre par des procédures

Pour chaque processus, on peut avoir une procédure pour chaque signal d'entrée, ou on peut avoir une procédure commune avec le type du signal codé sous forme de paramètre. Dans tous les cas, l'activation des processus va suivre les appels de procédure. Lorsqu'un signal est émis, le récepteur va prendre une priorité préemptive sur l'émetteur et terminer ses transitions avant que le contrôle soit rendu à l'émetteur.

Si cette solution convient, le système SDL doit être structuré et doit se comporter comme un arbre d'appel de procédures. Il faut suffisamment de temps entre chaque signal d'entrée ou débordement de temporisation pour faire tout le traitement et l'attente qui doit être fournie en réponse. Il ne doit pas falloir donner une priorité préemptive aux processus autre que celle qui découle implicitement de l'arbre d'appels. Pour chaque signal envoyé en bas de l'arbre d'appels, pas plus d'un signal de réponse doit être retourné (mis en œuvre comme une valeur de retour de procédure). En pratique, toutes les attentes non déterministes pour une «nouvelle» entrée et tous les débordements de temporisation doivent être pris en charge par un processus. Le rôle des autres processus est d'exécuter le traitement secondaire et la sortie qui découle d'une nouvelle entrée.

Les appels de procédure peuvent être considérés comme un cas particulier de communication synchrone, où l'ordonnancement est tel que le récepteur prend une priorité préemptive sur l'émetteur. Ceci constitue une restriction sévère sur l'utilisation du SDL. Par conséquent, cette solution simple et rapide ne va pas s'appliquer dans tous les cas. La communication SDL est plus générale et plus souple que les appels de procédure.

Communication avec tampons

La synchronisation SDL est infiniment élastique, signifiant que l'émetteur peut être infiniment en avance sur le récepteur d'une grande quantité de signaux. Comme nous l'avons déjà annoncé, cette situation ne peut pas se produire dans la réalité. En pratique, les files d'attente doivent être limitées et en cas de file d'attente pleine, soit l'émetteur va devoir attendre, soit les signaux vont être perdus. Une ingénierie soignée est nécessaire sur ce point pour assurer un contrôle de charge régulier sans dégradation des performances dans les situations de débordement.

Le modèle de communication décrit à la Figure I.7-12 utilise les sémaphores généraux introduits au I.7.5.2. Les signaux SDL sont codés dans des tampons et transmis comme des messages via le sémaphore *S*. Les tampons libres sont gérés par un autre sémaphore *F*. Le modèle est très général et a été utilisé pour mettre en œuvre beaucoup de systèmes temps réel spécifiés avec SDL. Deux aspects méritent considération:

– *Le contrôle de charge* – Les tampons circulaires fournissent une opportunité d'exécuter une forme simple de contrôle de charge. Dans les situations de surcharge la file d'attente des messages entre l'émetteur et le récepteur va grossir jusqu'à ce que tous les tampons libres soient utilisés. Ceci empêche l'émetteur de produire plus de messages avant que le récepteur soit arrivé à traiter certains des messages déjà dans la file d'attente. En dimensionnant soigneusement les pools de tampons libres, on peut contrôler la charge interne dans un système logiciel.

– *Les possibilités de blocage* – Lorsque plusieurs processus se font concurrence pour l'accès aux mêmes ressources, un blocage peut être possible. Considérer une situation de surcharge où le pool des tampons libres de la Figure I.7-12 est vide. Si le récepteur a maintenant besoin d'un tampon supplémentaire du pool des tampons libres, le système peut se bloquer avec tous les processus en attente d'un tampon libre. On devrait par conséquent utiliser de préférence des pools de tampons séparés pour éviter ce problème.

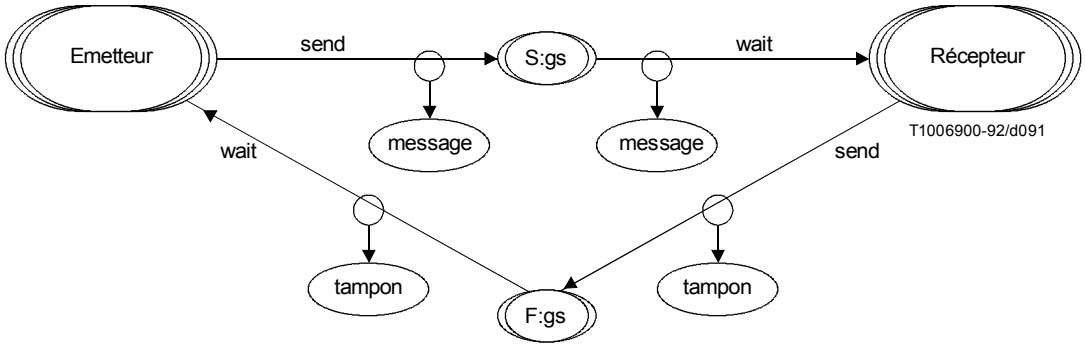


FIGURE I.7-12/Z.100

Communication par tampons de messages

A la Figure I.7-12, les messages sont déplacés par transfert de pointeurs et la taille du tampon est déterminée par le pool des tampons libres. Un modèle différent consiste à laisser les tampons à l'intérieur du sémaphore de communication et à recopier les messages. Lorsqu'un débordement se produit, l'opération d'envoi doit retarder l'émetteur.

L'inconvénient de la communication par tampon est qu'elle est plus lente, dans la plupart des cas, que les appels de procédure directs, et qu'elle n'est pas supportée par beaucoup de langages de programmation (CHILL est une exception). On doit par conséquent acquérir le logiciel supplémentaire nécessaire pour supporter la communication par tampons, et avoir des ressources de calcul pour le traitement supplémentaire. Les primitives pour la communication asynchrone fournies par beaucoup de systèmes d'exploitation seront souvent suffisantes.

Comme il est expliqué au I.7.2.2, il y a deux classes d'informations que l'on souhaite communiquer: les valeurs continues et les séquences de valeurs.

La communication à travers un seul tampon, c'est-à-dire une variable partagée est la façon la plus directe de transmettre des valeurs continues. Le producteur peut affecter la valeur lorsqu'elle doit être changée et les utilisateurs peuvent la lire lorsqu'ils en ont besoin, comme le montre la Figure I.7-13. Lire une valeur courante peut seulement être fait quand l'utilisateur est actif et exécute une opération de lecture. Il n'y aura pas d'attente induite, parce que l'utilisateur veut connaître la valeur au moment de la lecture.

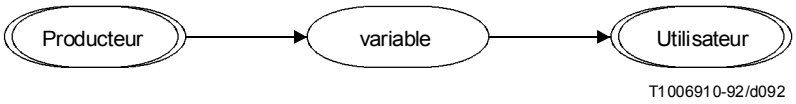


FIGURE I.7-13/Z.100

Communication via une variable partagée

Dans le cas où le producteur et l'utilisateur sont des processus concurrents, les opérations d'écriture et de lecture doivent être mutuellement exclusives. Ceci peut être obtenu de différentes manières:

a) S'assurer que les opérations de lecture et d'écriture sont atomiques l'une par rapport à l'autre:

   1) utiliser les instructions primitives;

   2) inhiber les interruptions;

   3) faire un ordonnancement tel que l'émetteur et le récepteur ne puissent jamais s'interrompre mutuellement.

b) Contrôler l'accès à la variable partagée au moyen d'un allocateur de ressource. Le producteur et le récepteur doivent demander à l'allocateur de fournir le droit d'accès avant de débuter une opération sur la variable partagée, et retourner le droit d'accès après la fin de l'opération. Ceci peut être obtenu par des opérations *wait* et *send* comme l'illustre la Figure I.7-14.

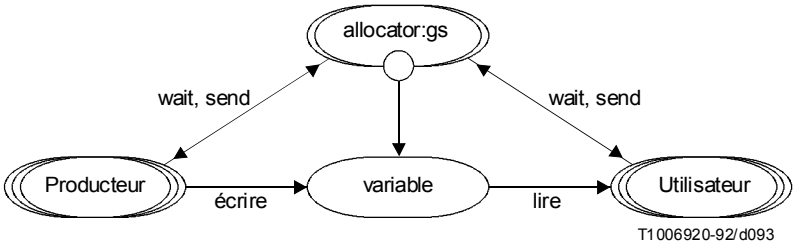


FIGURE I.7-14/Z.100

Communication via une variable partagée contrôlée par un allocateur de ressource

Acheminement

Lorsqu'un processus SDL envoie un signal, celui-ci doit d'une façon ou d'une autre être acheminé jusqu'au récepteur. Les signaux SDL identifient leurs émetteur et récepteur par des valeurs PId et les signaux sont acheminés par les blocs sur la base de *qui* est le récepteur, et pas *où* il est, sauf peut-être lorsque l'itinéraire est spécifié par la clause **via**. Au niveau abstrait, ceci est suffisant, mais dans la réalisation nous devons connaître l'emplacement physique du récepteur pour acheminer correctement le signal du processus émetteur jusqu'au processus récepteur.

L'emplacement physique du processus est déterminé par la machine physique, le processus logiciel à l'intérieur de la machine, et peut-être l'adresse locale à l'intérieur du processus logiciel.

Il faudrait, cependant, garder le logiciel aussi indépendant que possible de l'emplacement physique du processus. Un processus devrait connaître aussi peu que possible du chemin que le signal parcourt pour arriver à sa destination. Idéalement, il devrait seulement connaître *qui* est le récepteur, mais pas *où* il est.

Dans une certaine mesure, ce point est résolu par le concept des variables PId. La structure de données d'un processus va contenir les variables PId qui représentent les processus auxquels il peut envoyer des signaux. Ces variables sont affectées soit lors de la création du processus, soit dynamiquement durant le déroulement du processus. Grâce au concept de types abstraits de données, la représentation réelle des valeurs de PId peut être cachée du code du processus.

Mais la mise en œuvre du type de données PId va dépendre de la façon dont les valeurs de PId sont représentées. Le SDL ne définit pas la façon dont les valeurs de PId sont engendrées et représentées. La décision est laissée au concepteur. Une des questions majeures de la conception est donc de déterminer comment les valeurs de PId seront représentées et allouées.

Une solution commune est de représenter les valeurs de PId (adresses de signaux) par un identificateur pour le type de processus et un autre pour l'instance du processus. Mais ceci désignera une adresse logique et non une adresse physique. Nous allons, par conséquent, avoir besoin d'un certain support pour établir la correspondance entre les adresses logiques et les emplacements physiques.

A la Figure I.7-4, il y a un bloc de logiciel séparé, appelé *Message routing* (*Acheminement de messages*), qui effectue l'acheminement réel au niveau du processus logiciel. Son but est de cacher les adresses physiques des processus de l'application, en permettant aux messages d'être acheminés sur la base d'adresses de destination logiques. Par conséquent, un processus SDL n'a pas besoin de savoir où sont situés les autres processus. Il est suffisant de connaître leurs identificateurs logiques. D'ailleurs, la mise en œuvre du type de données PId est indépendante de la structure de l'adresse physique. Le système d'acheminement va utiliser l'identificateur logique et une table de correspondance d'adresses pour sélectionner un sémaphore par lequel le message est envoyé, comme l'illustre la Figure I.7-15. Ainsi, la connaissance de l'acheminement physique est centralisée dans la table de correspondance d'adresses.

S'il n'y a pas besoin de cacher les adresses physiques, un système d'acheminement séparé n'est pas nécessaire.

Un avantage de cette approche est qu'il est simple de réacheminer les messages dans les cas d'extensions en ligne du système.

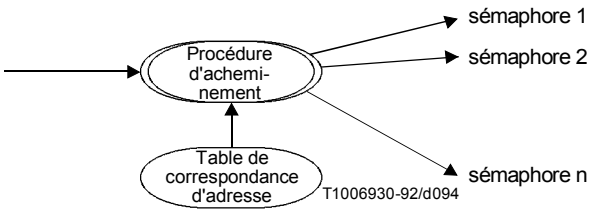


FIGURE I.7-15/Z.100

La procédure d'acheminement

I.7.5.4 Comportement séquentiel

Dans ce paragraphe, nous allons d'abord considérer la mise en œuvre de machines à états finis (FSM) (*finite state machines*) en général. Ensuite, nous considérerons les caractéristiques particulières du SDL, comme les services, les procédures, les décisions et la création dynamique de processus.

Le comportement de machine à états finis des processus SDL peut être mis en œuvre de nombreuses façons. Une consiste à coder la partie contrôle directement comme des expressions **if-then-else** ou des expressions **case** dans le langage de programmation. Une autre consiste à coder la partie contrôle dans des tables état/transition. Entre les deux, il existe un choix important de solutions intermédiaires.

Mise en œuvre de machines à états finis directement dans le code

Dans cette approche, nous pouvons distinguer deux façons de représenter l'état :

- par une position dans le texte du programme;
- par une valeur rangée dans une variable.

Considérons d'abord le cas de la position dans le programme. Le principe est de coder le diagramme état-transition directement dans le langage de programmation. Puisque le diagramme est une bonne définition de la logique, une correspondance directe peut être trouvée. En conséquence, on ne devrait pas essayer de transformer le diagramme dans un programme sans **goto**, mais utiliser le **goto** pour accéder à l'état suivant.

Une mise en œuvre caractéristique est la suivante:

```
begin
    state-1: wait(input);
        case input of
            signal-a:       call Action-1; goto state-3;
            signal-b:       call Action-2; goto state-5;
            else:           call Action-3; goto state-1;
        end case;
    state-2: wait(input);
        case input of
            signal-c:...
...
end
```

A cause de la vitesse d'exécution nécessaire, ceci peut être la solution préférable pour les processus d'entrées/sorties. Si des décisions peuvent affecter l'état suivant, des instructions «goto state» doivent être ajoutées aux procédures «Action». Les procédures SDL peuvent se traduire directement en procédures du langage de programmation.

Puisque l'état est représenté par une position dans le programme, cette technique implique que chaque processus SDL soit mis en œuvre comme un processus logiciel séparé, comme un processus CHILL. Ceci peut conduire à une perte importante d'espace si le nombre de processus est élevé.

Dans l'autre cas, l'état peut être rangé dans une variable. Ceci va conduire typiquement à un programme qui attend l'entrée en un seul endroit :

```
repeat forever
begin
    wait (input);
    case state of
        state-1:
            case input of
              signal-a:     call Action-1; state:= state-3;
              signal-b:     call Action-2; state:= state-5;
              else:         call Action-3; state:= state-1;
            end case;
        state-2:
            case input of
              signal-c:...
              ...
        end case;
    end;
```

Si les décisions peuvent modifier l'état suivant, l'affectation de l'état suivant doit être déplacée à l'intérieur des procédures Action ci-dessus. Les procédures SDL peuvent être facilement supportées ici également dans la mesure où on peut accepter l'attente d'une entrée à l'intérieur des procédures.

Cette approche peut être étendue pour manipuler beaucoup de processus SDL à l'intérieur d'un processus logiciel simplement en introduisant un tableau d'états indexé par l'identificateur de processus. Le multiplexage d'un grand nombre de processus SDL dans un seul processus logiciel peut aider à réduire la consommation supplémentaire d'espace et de temps induite par le système d'exploitation. La communication interne entre les processus SDL s'exécutant dans le même processus logiciel peut être considérablement plus rapide qu'une communication entraînant synchronisation et commutation de contexte entre les processus logiciels.

Un inconvénient est que la mise en œuvre des procédures SDL générales devient un peu plus difficile. Le problème est de garder trace des adresses de retour lorsque les procédures SDL contiennent des états. Si le programme est partagé par de nombreuses instances de processus, celui-ci doit être manipulé explicitement en utilisant une pile d'adresses de retour.

Dans le cas d'une communication par appels de procédures, l'état doit être mémorisé dans une variable. Dans cette approche, il y aura une procédure correspondant à chaque signal d'entrée du processus. Chaque procédure doit d'abord tester l'état et ensuite exécuter la transition correspondante. Les procédures SDL générales nécessitent une attention particulière également dans cette approche.

Mise en œuvre de machines à états finis gérée par tables

Puisque le modèle de processus du SDL est basé sur le modèle de la machine à états finis (EFSM) (*extended finite state machine*), tous les processus SDL partagent les caractéristiques communes des machines à états finis étendues.

De telles machines conviennent très bien à la mise en œuvre par des tables. Leurs comportements peuvent, utilement, être définis dans une table état/transition, qui est équivalente à la forme graphique utilisée dans les diagrammes état/transition. Une telle table peut être mise en œuvre par un tableau à deux dimensions indexé par l'état courant et le signal d'entrée, où chaque élément spécifie un état suivant et une action à faire pour chaque combinaison de l'état courant et de l'entrée courante (voir la Figure I.7-16).

	Signal	
Etat	Signal 1	Signal 2
Etat 1	Etat 2 / Action 1	Etat 3 / Action 3
Etat 2	Etat 3 / Action 5	Etat 1 / Action 2
Etat 3	Etat 2 / Action 3	Etat 1 / Action 2

FIGURE I.7-16/Z.100

Une table état/transition

On peut facilement concevoir un programme général qui utilise une table pour déterminer l'état suivant et l'action en fonction d'un état courant et d'un signal d'entrée. Ceci est l'idée générale qui est derrière la mise en œuvre des FSM gérée par table.

Normalement, il va y avoir beaucoup de cellules inutilisées parce qu'on n'attend pas que tous les signaux d'entrée arrivent dans tous les états. Il est mieux par conséquent de mettre en œuvre la table de manière à optimiser l'espace utilisé.

Une façon de concevoir une mise en œuvre de machines à états finis étendues gérées par table est mise en valeur par le diagramme de structure de logiciel de la Figure I.7-17. Elle contient un programme général appelé *FSM-support* qui interprète une structure de données *ST-table* représentant la table état/transition de la machine à états finis.

Le logiciel décrit à la Figure I.7-17 fournit une plate-forme de mise en œuvre concurrente pour les processus décrits comme des machines à états finis étendues. L'ordonnancement est effectué sur la base des messages d'entrée. Chaque message contient une adresse qui identifie le processus récepteur. Le programme *FSM-Support* va utiliser l'adresse pour activer le processus en sélectionnant le type de processus *Process-Type* et les données de processus *Process-Data* appropriés. Les données de processus *Process-Data* contiennent l'état courant et les objets qu'une machine à états finis étendue peut manipuler en plus de l'état (c'est-à-dire l'*extension* de la machine à états finis pure). Le message d'entrée transporte un nom de type de signal, qui est utilisé pour sélectionner une transition. Pour chaque message d'entrée, le processus adressé est autorisé à exécuter une transition. Il s'agit par conséquent d'une mise en œuvre quasi concurrente, où l'ordonnancement des processus est déterminé par l'adresse qui se trouve dans le message d'entrée.

En bref, *FSM* exploite le fait que le programme de support peut être réentrant pour tous les processus *FSM*, et que la table *ST-table* est réentrante pour tous les processus du même type. S'il y a beaucoup d'instances provenant du même type de processus, il doit y avoir pour chaque instance un élément *Process-Data*, par contre *ST-table* et *Actions* peuvent être partagées. S'il y a beaucoup de types de processus, chacun doit avoir sa propre *ST-table* et ses *Actions* correspondantes.

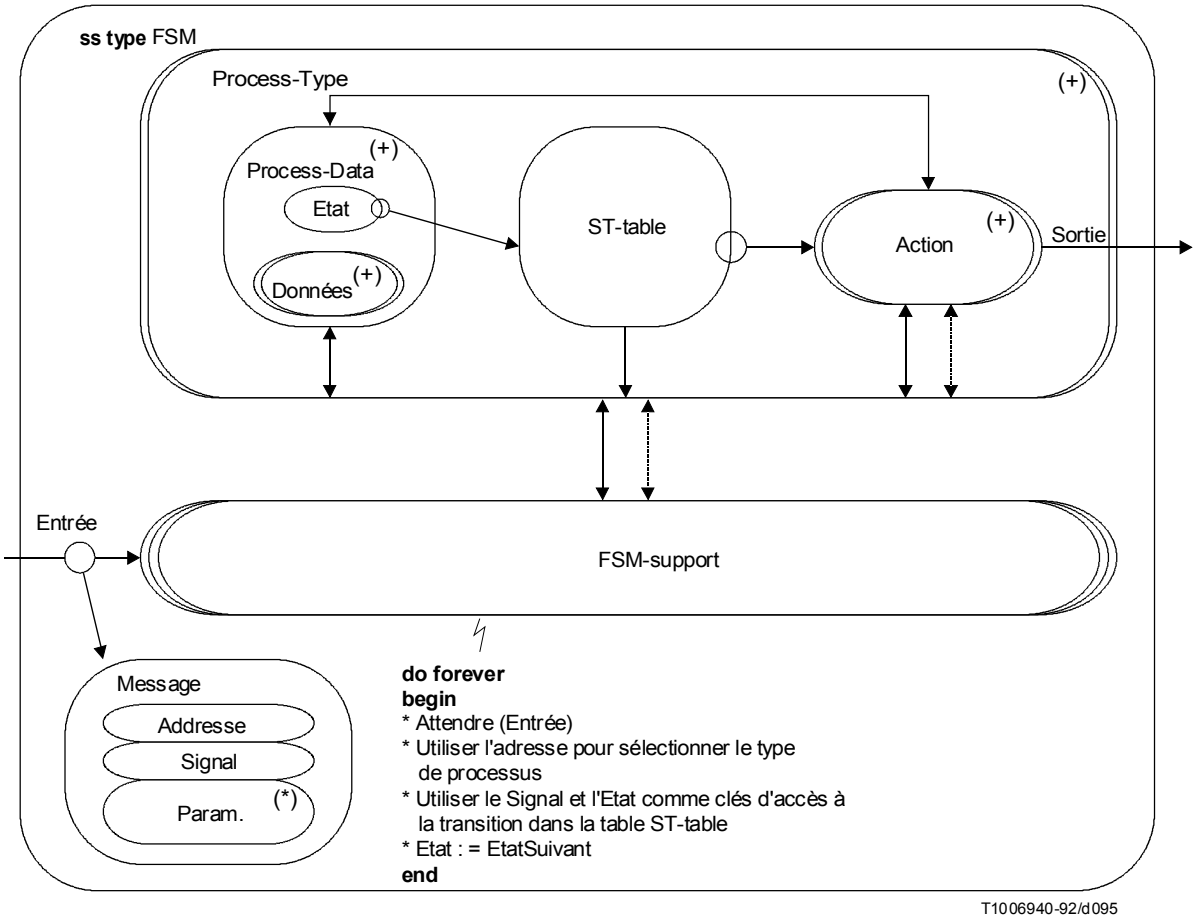


FIGURE I.7-17/Z.100

Logiciel de support pour les machines à états finis étendues

Dans la *ST-table*, chaque état est représenté par un enregistrement qui contient un nombre (variable) d'enregistrements de transitions, un pour chaque transition à partir de l'état, comme le montre la Figure I.7-18. Chaque enregistrement de transition spécifie les signaux d'entrée qui peuvent déclencher la transition, l'action correspondante et l'état suivant. Voir l'exemple à la Figure I.7-19.

Le programme *FSM-support* va attendre les messages d'entrée et répondre aux messages en respectant l'ordre de leur arrivée. Pour chaque message, il va accéder à l'enregistrement de l'état dans la table *ST-table* référencée par l'état courant dans *Process-Data*, et rechercher l'enregistrement de transition qui correspond au message d'entrée. Si on trouve un enregistrement de transition avec le même nom de signal que le message d'entrée, la transition est sélectionnée.

Si ce n'est le cas, la transition pour la réception non spécifiée, marquée avec un «*», est choisie (voir la Figure I.7-18). Le programme *FSM-support* va alors exécuter la procédure *Action* référencée dans l'enregistrement de transition sélectionné et finalement, affecter la valeur de l'état suivant à l'état courant. Ceci termine la transition, et *FSM-support* va attendre un nouveau message d'entrée.

Lorsqu'un nouveau message arrive, la transition suivante est exécutée de la même manière.

Les actions peuvent, en principe, être codées soit comme du code interprété, soit comme du code directement exécutable dans le langage de programmation. Nous allons utiliser la dernière approche ici. Par conséquent, l'interprétation est limitée à la table état/transition qui représente la partie contrôle. La partie action est ensuite exécutée par appel de la procédure d'action référencée dans l'enregistrement de la transition.

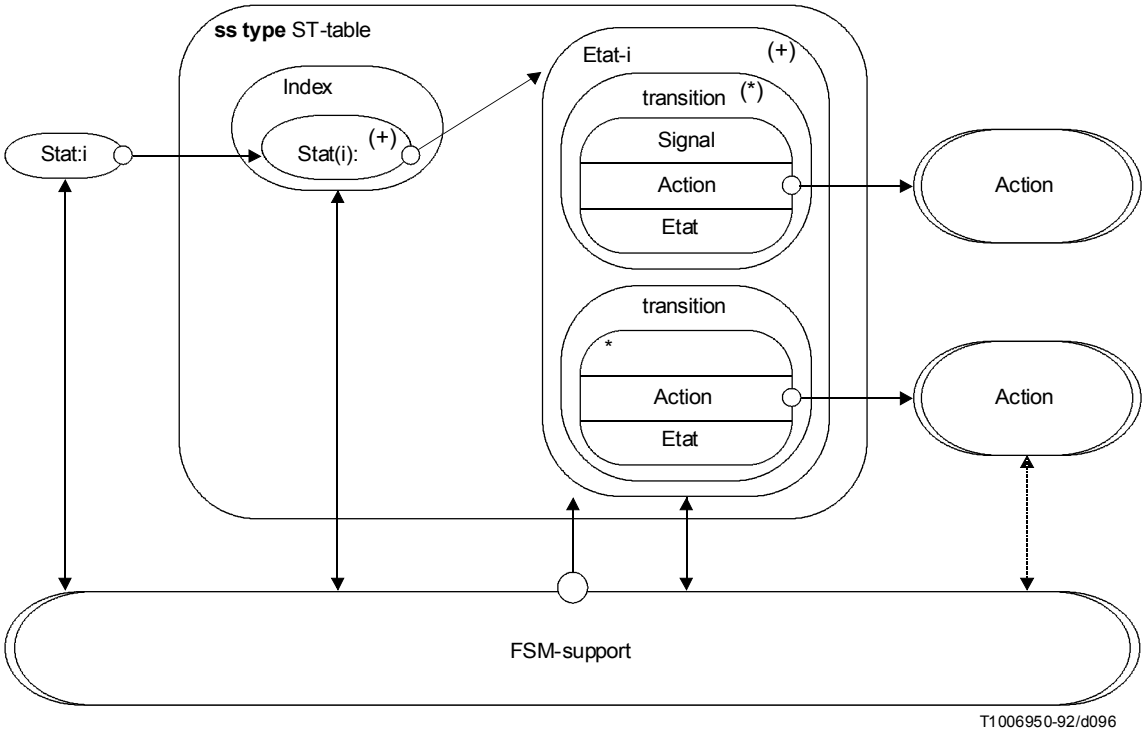


FIGURE 1.7-18/Z.100

Définition de ST-table

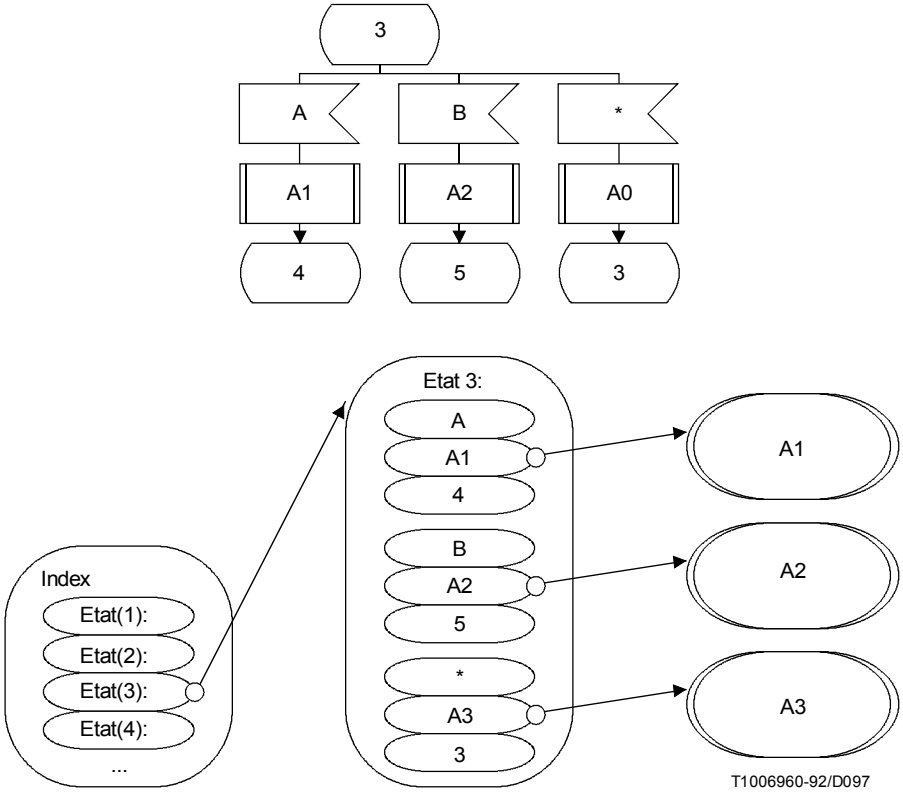


FIGURE I.7-19/Z.100

Exemple montrant comment un état est codé dans ST-table

Normalement, un certain nombre d'opérations distinctes vont être menées à bien pendant une transition, comme l'envoi des signaux de sortie, le démarrage et l'arrêt de temporisateurs, l'affectation de valeurs à des objets. Les procédures *Action* exécutent toutes ces opérations. Chaque opération va être normalement mise en œuvre comme une procédure. Par conséquent, la procédure *Action* consiste principalement en appels de procédure à des procédures d'action. De plus, elles peuvent consister en de simples affectations et expressions correspondant aux opérations spécifiées pour les données, par exemple:

```
procedure action-1
begin
    send-signal(open_door, here);
    send-signal(door_bell, here);
    starttimer(now+waiting_time, floor_delay, self);
    table:= cancel_res(table, moving, here);
end;
```

On peut se rendre compte que les procédures d'action peuvent être conçues pour constituer un niveau de langage très proche des spécifications SDL[6]. Ceci réduit l'effort nécessaire pour engendrer et pour lire le code correspondant aux actions de manière considérable. En général, les procédures vont correspondre aux opérations que la machine à états finis étendue doit réaliser:

- l'envoi de signaux aux autres processus ou aux équipements externes;
- les opérations sur les données locales;
- l'armement et le désarmement de temporisateurs.

Le format des signaux a besoin d'être normalisé pour chaque mise en œuvre particulière. Il n'est pas suffisant que les interfaces soient corrects au niveau de la conception fonctionnelle; les interfaces doivent être compatibles également au niveau concret. Ceci signifie que les processus qui communiquent doivent coder les signaux de la même manière.

Les messages en provenance d'un processus pour un autre s'exécutant sur la même instance de *FSM-support*, peuvent être affectés d'une priorité supérieure à celle des messages externes en transmettant de tels messages via une file d'attente interne. Ceci va accélérer la communication interne puisqu'on évite la synchronisation externe et les appels au système d'exploitation (opérations **wait**, **send**). Les messages aux processus à l'extérieur sont passés via des sémaphores de communication externe, vraisemblablement par une procédure d'acheminement.

La solution basée sur un contrôle par table possède plusieurs vertus:

a) **Elle facilite la mise en œuvre et la documentation** – Le codage des structures de contrôle, qui est normalement difficile et sujet aux erreurs, est fait facilement et d'une manière fiable dans la table *ST-table*. Les actions de transition sont habituellement structurées en petites procédures bien définies. Ces procédures ont une taille caractéristique de 10 lignes de code, et sont composées principalement d'appels à des sous-programmes standards pour l'envoi de signal, la gestion du temps et les opérations relatives aux données. Il est facile de voir la relation entre la conception fonctionnelle et sa mise en œuvre logicielle dans les tables *ST-tables*. Si facile que le codage peut être réalisé par des non-programmeurs. Un livre de recettes faciles peut être réalisé.

   Le nom de la procédure *Action* peut être enregistré comme un commentaire dans le graphe du processus SDL pour fournir une référence facile de la spécification SDL au code réel.

   Si le code est raisonablement bien structuré et commenté, le graphe de processus SDL et le listage du code source constituent une documentation suffisante. Ceci est normalement le cas des personnes utilisant réellement les diagrammes SDL comme documentation principale, et qui parcourent seulement le code lorsqu'ils doivent corriger des erreurs. De cette façon, l'approche encourage les personnes à tenir à jour la conception fonctionnelle.

b) **Elle améliore la modularité et facilite la maintenance** – Le programme standard *FSM-support* lui-même peut être utilisé dans de nombreux systèmes avec différentes applications. Par conséquent, c'est un module hautement réutilisable. Il est aussi plus facile de réutiliser les processus dans de nouvelles applications, lorsqu'elles sont mises en œuvre de manière standard.

---

[6] Ceci n'est pas particulier de l'approche gérée par table. Les mises en œuvre par codage direct peuvent utiliser une approche similaire pour le code des actions.

De plus, il est plus facile de faire des modifications et il est possible de contrôler leurs conséquences. Une pratique normale est de modifier d'abord le graphe de processus (*ST-table*) et ensuite d'ajouter les procédures *Action* qui sont nécessaires. L'astuce consiste à éviter les effets de bord si une procédure *Action* est appelée à partir de plusieurs transitions. Pour atteindre cet objectif, on devrait soit garder une référence amont des procédures *Action* vers les transitions, soit utiliser une procédure *Action* séparée pour chaque transition.

c) **Elle est fiable** – Le programme standard *FSM-support* va être minutieusement testé dans de nombreuses applications et va avoir par conséquent de grandes chances d'être très fiable. L'application est aussi susceptible d'être fiable puisqu'elle dérive directement d'une conception fonctionnelle vraisemblablement correcte.

d) **Elle est robuste** – La robustesse provient en partie du mécanisme de communication par messages, et en partie de l'approche du contrôle par table. Lorsque les signaux sont passés comme des messages, le récepteur peut toujours vérifier l'entrée avant qu'elle soit utilisée. Ainsi, *FSM-support* peut vérifier la cohérence des messages entrants avant de les accepter. De plus, la transition «*» dans les *ST-tables* assurent que toutes les entrées non attendues peuvent être reçues et traitées de façon adéquate.

e) **Elle supporte le test** – Le programme *FSM-support* peut être mis en œuvre dans une version de test qui permet au testeur de simuler des messages d'entrée et de tracer les actions des transitions et les messages de sortie qui sont engendrés. Ainsi un processus peut être testé convenablement dans un environnement qui ressemble exactement à l'environnement réel du processus lui-même.

De toute évidence, le programme *FSM-support* introduit une charge supplémentaire de traitement (overhead). La recherche des enregistrements de transitions peut consommer un certain temps. Pour réduire cet effet, on devrait organiser les enregistrements selon l'ordre des signaux les plus fréquents.

Lorsque tous les aspects d'une application sont pris en considération, il s'avère que le temps supplémentaire consommé est négligeable. Lorsque l'on met en œuvre plusieurs processus avec le même programme *FSM-support*, il peut même y avoir un gain net en raison de l'économie réalisée en commutation de contexte dans le système d'exploitation.

Le programme *FSM-support* est un candidat bien défini pour l'optimisation en temps, si on le juge nécessaire. On peut choisir de coder la table comme un tableau à deux dimensions, où l'état et le signal d'entrée servent d'indices. Ceci va fournir un accès rapide, mais peut conduire à une consommation trop importante d'espace mémoire.

Puisque la table état/transition *ST-table* code les structures de contrôle de manière efficace sur le plan de l'espace mémoire, on va économiser l'espace dans les plus grandes applications.

Quelles sont les limites de cette approche? Puisque les autres processus sont bloqués pendant une transition, les *Actions* ne devraient pas attendre les événements externes ou exécuter des opérations consommant beaucoup de temps. Pour la même raison, les processus mis en œuvre sur la même instance de *FSM-support* devraient avoir la même priorité. (Ils ne peuvent pas s'interrompre entre eux.)

Mise en œuvre des constructions caractéristiques du SDL

Le SDL possède certaines caractéristiques qui sont différentes des machines à états finis ordinaires, comme les décisions, les mises en réserve, les procédures, les services et la création dynamique de processus. Celles-ci peuvent toutes être mises en œuvre au-dessus des mises en œuvre des machines à états finis décrites jusqu'ici.

Les décisions dans la spécification SDL peuvent être codées comme des états lorsqu'on utilise la technique correspondant à FSM-support. Ceci n'est pas strictement nécessaire, mais elle rend les procédures Action indépendantes d'états particuliers.

Lorsqu'une décision affecte l'état suivant, la traiter comme s'il s'agissait d'un état. Faire l'opération correspondant à la question avant d'entrer dans l'état associé à la décision, et envoyer la réponse comme un signal d'entrée pour qu'il soit reçu dans l'état associé à la décision. De cette manière, le branchement dépendant des valeurs internes et des signaux externes est traité de façon similaire. Pour accélérer la prise de décision et pour terminer les décisions avant de nouvelles entrées, les signaux correspondant aux décisions devraient être traités de manière interne dans *FSM-support*, c'est-à-dire de façon prioritaire sur les signaux externes. Lorsqu'un signal de décision a été produit, la transition correspondante doit être exécutée immédiatement. Lorsqu'une transition contient une décision qui n'affecte pas l'état suivant, elle doit être codée comme une décision interne dans la procédure *Action* de la transition.

Le mécanisme de mise en réserve exige que chaque processus possède une file d'attente logique de signaux mis en réserve. Cette file d'attente peut être mise en œuvre comme partie des données du processus. Un symbole de mise en réserve peut alors être codé comme une transition retournant à l'état courant (état-suivant := état-courant), où l'action consiste à placer le signal dans la file d'attente des mises en réserve. Toutes les fois qu'un processus réalise une transition vers un nouvel état, les signaux de la file d'attente des mises en réserve sont déplacés dans la file d'attente des entrées et sont traités comme des signaux d'entrée normaux. Par conséquent, la mise en réserve peut être mise en œuvre en ajoutant une file d'attente des mises en réserve aux données de chaque processus, et en déplaçant les entrées de cette file d'attente à la file d'attente des entrées normales à la fin de chaque transition conduisant à un nouvel état. Puisque ceci peut être plutôt consommateur de temps, les mises en réserves devront être évitées dans les applications à durée critique.

L'idée d'une file d'attente des mises en réserve est de garder certains signaux de côté jusqu'à ce que ce soit le moment de les traiter. Ceci peut être mis en œuvre également d'autres façons, par exemple en utilisant des files d'attente de signaux séparées. Le lecteur devrait réfléchir au but poursuivi en utilisant des mises en réserve avant de réaliser la mise en œuvre, et rechercher des solutions qui évitent les activités qui consomment du temps.

Les procédures en SDL peuvent contenir des états. Dans de pareils cas, la procédure introduit une structure à niveau entre les graphes de processus. Ceci se met facilement en œuvre par codage direct lorsqu'on peut faire correspondre directement les procédures SDL et les procédures des langages de programmation.

Mais dans les cas où plusieurs instances de processus SDL correspondent à une seule instance de processus logiciel, ceci n'est pas possible. Pour mettre en œuvre ce cas, chaque processus va avoir besoin d'une pile pour sauvegarder les états, les adresses de tâches et les données locales aux procédures. Lorsqu'une procédure est appelée, le contexte courant (état, adresse de tâche et données) est empilé, et la procédure est entrée. Au retour, l'opération opposée a lieu. Ceci est plus simple à mettre en œuvre, si les appels de procédure se produisent à la fin d'une transition, parce que le retour se fera à un état et non à une adresse arbitraire de tâche.

Les services en SDL sont fondamentalement des machines à états finis qui s'exécutent en quasi parallélisme et qui utilisent des signaux prioritaires pour communiquer entre eux. Ils sont facilement mis en œuvre dans un seul processus logiciel en utilisant soit la méthode du contrôle par table, soit l'approche du codage direct. Pour identifier le service qui doit recevoir un signal d'entrée donné, il faut considérer le nom du type de signal en plus de l'identification du processus PId.

Dans le cas où il y a une instance de processus logiciel par instance de processus SDL, la création dynamique de processus exige de faire appel au système d'exploitation. Dans un processus logiciel, la création dynamique de processus a pour effet l'allocation et l'initialisation d'un enregistrement de mémoire disponible pour ranger les données du processus.

I.7.5.5 Données

Maintenant nous arrivons au problème non trivial de la mise en œuvre des données SDL. Les données SDL couvrent trois aspects:

- les sortes, définies en termes d'opérateurs et d'équations;
- l'instanciation des sortes comme variables de processus SDL;
- l'instanciation des opérateurs spécifiés dans des expressions de transitions de processus SDL.

Dans les spécifications SDL, les sortes aident à se concentrer sur la fonctionnalité et à laisser de côté les détails non pertinents d'une mise en œuvre. Dans une mise en œuvre logicielle, la notion de type abstrait de données favorise l'accroissement de la modularité et de la réutilisation. A ce niveau, le principal aspect est l'*encapsulation* et le *masquage d'informations* [18].

L'idée consiste à encapsuler des données dans des modules qui fournissent uniquement des opérateurs bien définis à l'environnement. Ceci signifie qu'on ne peut accéder aux structures de données directement, mais seulement par invocation des opérateurs, souvent mis en œuvre comme des procédures. De cette façon, les structures de données internes sont encapsulées par les procédures représentant les opérateurs et sont cachées de l'environnement (voir la Figure I.7-20).

L'instanciation des opérateurs spécifiés dans les graphes de processus SDL va donc être mise en œuvre par appels des procédures représentant les opérateurs. Ceci signifie que les *Actions* des transitions sont indépendantes d'une structure particulière de données utilisée pour mettre en œuvre les types de données.

Même si les données sont définies informellement au niveau du SDL, la notion de type abstrait de données est utile parce qu'elle aide à obtenir indépendance et modularité.

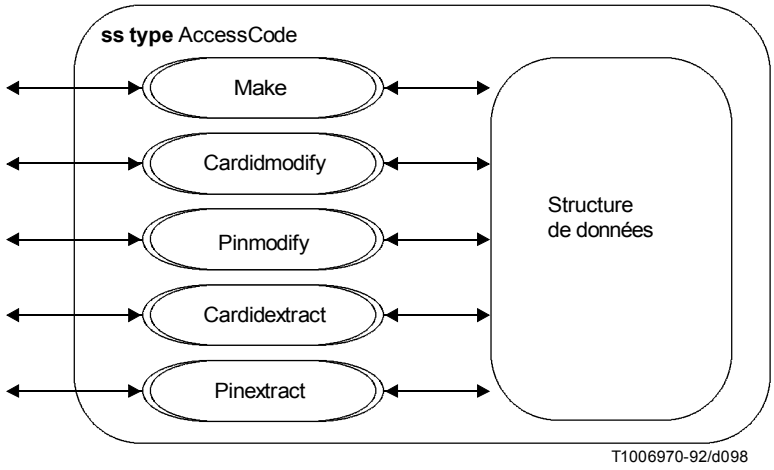


FIGURE I.7-20/Z.100

La mise en oeuvre d'une sorte est un module encapsulé

La première étape dans la conception du logiciel pour une sorte consiste à spécifier sa signature (l'interface de l'opérateur). Considérer, comme exemple, la sorte SDL *AccessCode*:

```
newtype AccessCode
operators
    Make             : Integer, Integer           ->     AccessCode;
    cardidModify     : AccessCode, Integer        ->     AccessCode;
    pinModify        : AccessCode, Integer        ->     AccessCode;
    cardidExtract    : AccessCode                 ->     Integer;
    pinExtract       : AccessCode                 ->     Integer;
axioms
/* nous les avons omis ici */
endnewtype AccessCode;
```

L'interface des opérateurs correspondants en termes d'en-têtes de procédures peut être:

```
Make(integer, integer)
Cardidmodify(accesscode, integer)
Pinmodify(accesscode, integer)
Cardidextract(accesscode)
Pinextract(accesscode)
```

La prochaine étape consiste à choisir une structure de données appropriée. Le choix est très important puisqu'il va avoir une influence majeure sur les algorithmes utilisés pour mettre en œuvre les opérateurs et donc sur les performances. Le choix va dépendre des contraintes de conception non fonctionnelles. Il sera possible d'avoir plusieurs mises en œuvre correspondant à différentes contraintes de conception, toutes accessibles via des interfaces identiques d'opérateurs.

Dans les spécifications SDL formelles, la signification des opérateurs peut être définie au moyen d'équations ou de spécifications d'opérateur. Cependant, il y a jusqu'ici aucune façon générale et directe de transformer une définition axiomatique en une mise en œuvre efficace. Nous devons donc être pragmatiques en matière de conception des structures de données et d'algorithmes. Nous avons un cas beaucoup plus simple lorsque l'on utilise des spécifications d'opérateur, puisqu'une spécification d'opérateur est similaire à une procédure retournant une valeur (en fait, elle se définit par une transformation en une telle procédure).

D'abord, nous devons considérer d'autres structures de données, et en sélectionner une qui répond aux besoins de performance, comme par exemple, une liste liée. Ensuite, nous concevons une procédure correspondant à chaque opérateur.

L'approche de conception de cet exemple a consisté à définir une procédure pour chaque opérateur de la sorte. On peut également considérer d'utiliser une macro ou même une simple instruction, si la sorte est directement traitée dans le langage de programmation. Ceci va être le cas de beaucoup des sortes prédéfinies du SDL. Dans les systèmes temps réel, beaucoup de structures de données et d'algorithmes sont simples, aussi on peut traiter une bonne partie du problème en utilisant des sortes prédéfinies.

Dans les cas complexes, il faut un plus grand effort pour réaliser cette partie de la conception. Pour les sortes comportant des structures de données complexes, il est utile de représenter la structure de données logique par des descriptions de données conceptuelles, par exemple des descriptions entités/relations comme partie des besoins fonctionnels ou de la conception fonctionnelle.

L'étape suivante devrait alors consister à faire correspondre la description conceptuelle et les données SDL. Puisque l'interface d'opérateur des objets cache la structure interne des données à l'environnement, un système de gestion de base de données peut être utilisé sans que l'environnement le connaisse. En fait, la notion de type abstrait de données peut s'utiliser avec succès pour des structures de données très élaborées.

Les types abstraits de données sont supportés par SDL, mais la notion n'est pas spécifique du SDL. En fait, les directives ci-dessus vont être utiles dans n'importe quelle conception de logiciel, indépendamment de la méthode de spécification. Ils facilitent la décomposition d'une conception en modules avec un faible couplage et une forte cohésion.

I.7.5.6 Conception de l'architecture globale du logiciel

Dans les paragraphes précédents, nous avons considéré quelques problèmes essentiels de conception et des solutions. Dans ce paragraphe, nous allons voir comment les solutions peuvent être utilisées dans une approche globale de la conception de l'architecture du logiciel. Le but est de trouver et documenter une structure logicielle qui va mettre en œuvre le système SDL et satisfaire les besoins non fonctionnels. Les informations d'entrée pour la conception de l'architecture logicielle sont:

- la spécification SDL;
- la conception de l'architecture matérielle;
- les besoins non fonctionnels.

A partir de la spécification SDL nous connaissons les interfaces fonctionnelles et les fonctionnalités du système logiciel. A partir de la conception de l'architecture du matériel nous connaissons les interfaces physiques. A partir des besoins non fonctionnels nous connaissons les restrictions relatives au temps et aux performances.

La première étape consiste à examiner la communication interne et externe, et à concevoir une solution pour chaque interface.

D'un côté, nous avons les problèmes des interfaces physiques, de l'autre nous avons les besoins de communication interne. Commencer par considérer les interfaces d'entrées/sorties et leurs exigences en termes de temps de réponse, de mesure de temps, de détection d'événement, de priorités et de synchronisation. Ensuite considérer les besoins de communication interne entre processus SDL. Le temps constitue-t-il une ressource critique? Peut-on faire des appels de procédures? Doit-on utiliser la communication par tampon?

A cette étape, on doit prendre une décision quant au besoin d'utiliser un système d'exploitation.

L'étape suivante consiste à relier les modules d'interfaces aux modules de processus internes. Ceci va conduire à une structure globale de logiciel, telle que celle présentée à la Figure I.7-4 ou à la Figure I.7-10.

La meilleure façon de mettre en œuvre une machine à états finis étendue définie avec le SDL dépend des contraintes de conception. Si les contraintes de vitesse le permettent, on peut utiliser la mise en œuvre par table. C'est une solution compacte, souple et fiable. Si la vitesse de traitement est critique, on peut considérer l'utilisation de la mise en œuvre par codage direct et la mise en œuvre des signaux par des procédures.

La partie relative aux données peut être mise en œuvre dans des modules correspondants aux sortes.

On peut mettre en œuvre chaque processus SDL dans un seul processus logiciel. Mais ceci n'est nécessaire que lorsqu'on a besoin de priorité préemptive. Une autre solution consiste à exécuter plusieurs processus SDL par dessus un seul processus logiciel, par exemple en utilisant la solution du contrôle par table présenté au I.7.5.4.

Examiner de nouveau les temps d'exécution moyens estimés, voir I.7.4.3. On peut maintenant améliorer les estimations. Recalculer la moyenne de la charge de pointe du calculateur. Si cette valeur est au-dessus de la limite de charge, reconsidérer la conception, par exemple, en appliquant l'un des cas suivants:

- allouer les processus SDL différemment aux calculateurs;
- utiliser un calculateur plus rapide; ou
- optimiser le logiciel pour obtenir une grande vitesse d'exécution.

Si les valeurs de charges semblent convenables, continuer à vérifier les contraintes temps réel par rapport aux temps d'exécution maximaux qu'il faut respecter pour répondre aux besoins. Lorsque tout semble convenable, continuer par la conception de chaque processus SDL et du logiciel de support.

Les processus logiciels qui interagissent en utilisant une communication par tampons fournissent une interface par message à leurs environnements et cachent la structure interne. Ce sont, par conséquent, des modules relativement indépendants, pratiques à manipuler et faciles à intégrer.

I.7.6 Conception de l'architecture du matériel

Nous avons déjà discuté au I.7.4, de la conception de l'architecture matérielle. Le matériel, en contraste avec le logiciel, est fondamentalement différent du SDL. Le matériel est physique. De sorte qu'il produit des erreurs au cours du temps, qu'il a besoin de temps pour exécuter des tâches et qu'il est sujet au bruit.

Il y a des similitudes conceptuelles entre le SDL et le matériel. La concurrence, par exemple, vient naturellement avec le matériel, et il y a une longue tradition pour spécifier et mettre en œuvre le matériel au moyen des machines à états finis.

Il existe des langages de descrition du matériel, comme VHDL, qui ressemblent aux langages de programmation. Par conséquent, jusqu'à un certain point, la conception de l'architecture matérielle est semblable à celle du logiciel. C'est un problème que de faire correspondre la spécification SDL avec une description correspondante dans le langage de description du matériel. Ceci peut s'obtenir par des outils de traduction automatique de la même manière que pour le logiciel. Mais le concepteur de matériel doit résoudre un ensemble de contraintes différentes rendant la tâche différente de celle de conception de logiciel.

Concurrence et temps

Au niveau d'un circuit, les composants matériels agissent souvent de manière synchrone et interagissent de manière synchrone en utilisant des valeurs continues. Ces mécanismes doivent être utilisés pour mettre en œuvre l'opération synchrone et l'interaction asynchrone avec les séquences de valeurs existant dans le système SDL.

Par conséquent, soit le système SDL doit être restreint de manière à respecter les mécanismes matériels, soit le matériel doit fournir les mécanismes utilisés dans le SDL.

En principe, il est possible de mettre en œuvre la plupart des mécanismes du SDL en matériel. Nous pouvons construire des composants asynchrones correspondant aux processus SDL communiquant de manière asynchrone grâce à des canaux «à la» SDL.

Pour des raisons techniques et économiques, cependant, il ne sera normalement pas pratique de mettre en œuvre les systèmes SDL généraux en matériel. Par conséquent, il est plus commun de restreindre l'usage du SDL de manière convenable pour la mise en œuvre du matériel. Ceci peut signifier de restreindre la structure de communication de sorte qu'on puisse utiliser la communication synchrone, sans mise en réserve.

Dans un système synchrone, le signal d'horloge donne une référence de temps naturelle. Les mesures de temps peuvent alors être mises en œuvre en comptant les tic-tac de l'horloge.

Comportement séquentiel

Il n'y a pas de problème fondamental pour mettre en œuvre les machines à états finis en matériel. En fait, les machines à états finis étaient utilisées pour la conception du matériel avant de l'être pour la conception du logiciel.

Pourvu que l'usage des données soit restreint aux types de données supportés dans le langage de description du matériel, il n'y a pas de problème fondamental dans la mise en œuvre des tâches et des décisions.

Les procédures peuvent causer des problèmes. En particulier, si on a besoin de l'allocation dynamique de données pour les traiter. Par conséquent, l'usage des procédures devrait être restreint.

La création dynamique de processus, dans le contexte du matériel, n'est pas réalisable, s'il signifie créer une unité physique. Mais il peut être simulé en activant et relaxant des processus construits dans le matériel depuis le début.

I.7.7 Guide progressif pour la conception de l'architecture

Les principales étapes de la conception de mise en œuvre sont les suivantes:

- Etape 1: Compromis entre matériel et logiciel.
- Etape 2: Conception de l'architecture du matériel.
- Etape 3: Conception de l'architecture du logiciel.
- Etape 4: Restructuration et raffinage des spécifications SDL.
- Etape 5: Conception détaillée du matériel et du logiciel.

Remarquer que ce sont uniquement des étapes principales de conception. En pratique, on peut avoir besoin d'itérations. Tandis que les principales étapes de conception s'appuient sur des décisions de haut niveau qui sont susceptibles de rester manuelles, la conception détaillée et la mise en œuvre sont soumises à une automatisation progressive.

S'intéresser à la communication. Trouver des solutions pour les interfaces externes d'abord et ensuite pour les interfaces internes. Aller ainsi jusqu'à la conception de mise en œuvre de processus. Utiliser le principe de généralité; préférer utiliser les techniques les plus générales et les plus souples chaque fois que c'est possible.

Etape 1 – Compromis entre matériel et logiciel

- Analyser les besoins concernant la répartition physique des interfaces et des services. Choisir une structure de système physique pour la supporter. Minimiser la bande passante dont ont besoin les canaux couvrant des distances physiques.
- Considérer les implications des besoins relatifs aux performances. Calculer les charges moyennes de calcul pour chaque canal SDL, chaque acheminement de signal et chaque processus. Allouer les processus aux calculateurs de sorte que la charge moyenne sur un calculateur n'excède pas 0,3 Erlang de sa capacité totale.
- Considérer les implications des besoins relatifs au temps réel. Calculer les temps de réponse pour les fonctions à durée critique et vérifier que les besoins seront satisfaits. Utiliser la priorité pour assurer les réponses rapides. Isoler les parties à durée critique autant que faire se peut.
- Considérer les implications des besoins relatifs à la fiabilité. Considérer le besoin de redondance. Ajouter les unités redondantes et restructurer le système jusqu'à ce que les besoins soient couverts.
- Considérer les implications des besoins relatifs à la sûreté et à la sécurité.
- Considérer les implications sur les coûts réels de production.
- Considérer les implications sur les coûts réels des procédures de modification.

Etape 2 – Architecture du matériel

- Décrire la structure globale des calculateurs et des autres unités matérielles.
- Décrire les interconnexions physiques entre les unités matérielles.
- Décrire les formats de signaux, les modèles de synchronisation et les protocoles à utiliser sur les interconnexions physiques dans la mesure où ils ne sont pas déjà couverts dans la spécification des besoins ou dans la conception fonctionnelle.

Etape 3 – Architecture du logiciel

- Faire correspondre la spécification SDL de manière aussi directe et fiable que possible dans la mise en œuvre de logiciel.
- Considérer les interfaces physiques et concevoir les modules de logiciel qui vont prendre en charge la couche physique, c'est-à-dire la synchronisation, la détection d'événements, la gestion du temps et la conversion de format.
- Considérer les interfaces internes et choisir un mécanisme de communication convenable pour chacun, c'est-à-dire appels de procédures, tampons de messages, valeurs continues.
- Utiliser le modèle de communication le plus général et le plus souple pour les signaux SDL, c'est-à-dire la communication par tampons, à moins d'être certain d'avoir besoin des appels directs de procédures et que ceux-ci vont fonctionner correctement.
- Préférer utiliser un système d'exploitation général qui supporte les processus concurrents et la communication par tampons, sauf si évidemment une structure simple de programme séquentiel suffit.
- Sélectionner la méthode de mise en œuvre pour chaque processus SDL. Utiliser des systèmes de support généraux autant que faire se peut pour faciliter la mise en œuvre des fonctions de l'application et pour accroître la fiabilité.

Etape 4 – Restructuration et raffinage des spécifications SDL

- Restructurer et raffiner la spécification SDL de haut niveau en une spécification SDL de bas niveau, si nécessaire, de sorte qu'elle reflète la structure concrète du système.

La spécification SDL de bas niveau devrait refléter toutes les propriétés fonctionnelles de la mise en œuvre. Ceci sert à assurer l'équivalence.

La spécification SDL devrait refléter la structure concrète du système. Ceci sert à simplifier la mise en correspondance entre la spécification SDL et la mise en œuvre.

Il faudra normalement des fonctions supplémentaires pour supporter le système concret. Ces fonctions vont dépendre des décisions prises lors de la conception, et ne pourront être définies avant que la conception globale soit effectuée. Alors que certaines de ces fonctions seront invisibles aux utilisateurs, les autres seront visibles.

Sont normalement visibles à l'utilisateur les quelques fonctions suivantes:

- traitement d'erreur, c'est-à-dire comptes rendus d'erreurs services indisponibles;
- exploitation et maintenance de la réalisation, c'est-à-dire unités de blocage, unités de test;
- accès aux ressources limitées comme les imprimantes.

Sont normalement invisibles à l'utilisateur les quelques fonctions suivantes :

- multiplexage des calculateurs et des canaux;
- synchronisation et exclusion mutuelle;
- services de communication;
- contrôle de charge.

Noter que la restructuration ne veut pas dire que tout doit être redéfini. Une majorité des processus de la spécification SDL de haut niveau va rester inchangée. S'ils sont définis comme des types séparés, c'est une opération simple que de mettre les instances dans un nouveau contexte structurel avec quelques processus nouveaux.

Etape 5 – Conception détaillée du matériel et du logiciel

- Concevoir complètement chaque unité de matériel et de logiciel identifiée dans les étapes précédentes, en utilisant des composants et une technologie standardisés autant que faire se peut.

I.8 Approches formelles pour la validation, la vérification et le test

I.8.1 Introduction

Ce paragraphe fournit une introduction aux approches formelles de la validation, de la vérification et du test. C'est un vaste domaine, sujet à d'intenses activités de recherche. On prévoit de fournir des directives plus détaillées dans un futur proche.

Selon la terminologie communément admise, la *validation* concerne l'établissement d'une correspondance entre d'une part les besoins informels et d'autre part la spécification et/ou la mise en œuvre d'un système. En d'autres termes, l'objectif de la validation est de répondre à la question «est-ce que le système est celui attendu, est-ce qu'il répondra aux attentes de ses futurs utilisateurs?».

La *vérification* concerne l'établissement de la *correction*, selon certains critères (de correction), de la spécification et/ou de la mise en œuvre d'un système. En d'autres termes, l'objectif de la vérification est de répondre à la question «le système est-il correct?». Naturellement, une des attentes des futurs utilisateurs est que le système soit correct, ainsi la vérification peut être considérée comme faisant partie de la validation.

Le *test* concerne l'établissement de la *conformité*, selon certains critères (de conformité), d'une mise en œuvre relativement à sa spécification associée. Les critères de conformité sont normalement exprimés dans les normes au moyen d'énoncés de conformité, couvrant entre autres choses un ensemble minimal de caractéristiques supportées par les mises en œuvre conformes. Le test peut être considéré comme une technique à utiliser à la fois en validation et en vérification.

Les propriétés requises qu'il faut valider ou vérifier sont habituellement classées en propriétés de *sûreté* et en propriétés de *vivacité*. Les propriétés de sûreté doivent être satisfaites à tout instant dans le système spécifié, il s'agit par exemple de l'absence de blocage. Les propriétés de vivacité doivent être satisfaites à certains instants, comme «après la demande de connexion, le système doit répondre soit par une indication de connexion, soit par une indication de déconnexion».

Puisque le SDL, comme d'autres techniques normalisées de descriptions formelles, concerne essentiellement le comportement des systèmes, cette section se focalise sur la validation et la vérification des propriétés fonctionnelles, c'est-à-dire le comportement spécifié en termes de communication de signaux. La validation et la vérification de propriétés non fonctionnelles, comme les besoins temps réel, les critères de performance, etc. ne sont pas ici considérées plus en détail. De plus, la portée se limite finalement à la dérivation de cas de test; les procédures de test n'étant pas couvertes.

Un avantage d'utiliser un langage de spécification formelle, comme le SDL, est la possibilité de réaliser une validation plus précise et une vérification formelle des spécifications. La spécification peut aussi être utilisée pour dériver des cas de test de façon à tester les mises en œuvre de la spécification. Ceci constitue une conséquence directe de la sémantique formelle du SDL, puisque la sémantique définit une interprétation non ambigüe et permet le raisonnement formel sur les propriétés comportementales des spécifications.

Le modèle sémantique formel défini pour le SDL est un modèle *dénotationnel*, qui définit le comportement d'un système SDL en termes de machine abstraite qui interprète la spécification SDL. Dans un tel modèle, les propriétés comportementales de la spécification, importantes pour la validation et la vérification, ne sont pas exprimées explicitement; à la place, on met l'accent sur le comportement de la machine abstraite interprétant la spécification. Ceci fait que le modèle sémantique courant pour le SDL n'est pas directement applicable dans le contexte de la validation et de la vérification.

Une autre approche consiste à utiliser un modèle *opérationnel*, [19], qui définit le comportement du système spécifié en termes de son comportement observable en utilisant la notation des *Systèmes de Transitions Etiquetées*. Un modèle supportant cette vue est donné dans [20] et [21], où le comportement d'un système SDL est défini par rapport à l'endroit où les événements sont observés. Par exemple, les événements considérés comme observables dans l'environnement du système sont l'entrée de signaux dans le système et la sortie de signaux à partir du système, les événements considérés comme observables au niveau du bloc sont l'entrée de signaux dans le bloc observé et la sortie de signaux du bloc observé, etc. Un des avantages majeurs d'un tel modèle est la possibilité de développer des outils informatiques qui supportent la validation et la vérification de spécifications. De tels outils sont habituellement basés sur la technique d'*exploration des états*, [22], où le comportement du système est calculé et représenté en termes de *graphe des marquages accessibles* (*reachability graph*) ou d'un *arbre de communication asynchrone* (*Asynchronous Communication Tree ou ACT*), [23]. De manière informelle, le processus de calcul du graphe des marquages accessibles d'un système peut être décrit de la manière suivante:

> *Etant donné un état initial pour le système, tous les événements possibles qui peuvent se produire dans cet état sont réalisés, ceci fournit en résultat un nouvel ensemble d'états pour le système. Les états sont connectés à l'état initial avec un arc étiqueté identifiant l'événement réalisé. Ce processus est alors répété pour chaque nouvel état, jusqu'à ce qu'aucun nouvel état ne puisse être dérivé.*

Cependant, tous les outils basés sur cette technique ont en commun le problème de l'*explosion des états*. Le nombre d'états du graphe des marquages accessibles dépasse la capacité du système réalisant le calcul du comportement. Le problème de l'explosion des états est une caractéristique inhérente au comportement complexe des systèmes communicants, plutôt qu'une imperfection dans une technique particulière. Par exemple, un système SDL possède généralement un comportement infini, et il n'est donc pas possible de calculer le comportement complet du système. Néanmoins, il existe un certain nombre de techniques différentes pour réduire les effets de l'explosion des états:

- Les techniques qui *réduisent* le nombre d'états nécessaire pour engendrer:

  1) un *ordre partiel* des événements au lieu de l'entrelacement, comme par exemple [24];

  2) l'*abstraction*, en réduisant les états sémantiquement équivalents dans le graphe des marquages accessibles en un seul état, comme par exemple [25] et [28];

  3) l'*analyse partielle*, en n'engendrant qu'une partie du comportement possible, comme par exemple [26];

  4) le *partitionnement*, en divisant le système en sous-systèmes indépendants qui sont analysés séparément, comme par exemple [27].

- Les techniques de prise en compte des *performances* pour réaliser un stockage efficace du graphe des marquages accessibles, comme par exemple [26].

Les différentes techniques peuvent être combinées et utilisées dans un même outil.

I.8.2 Validation et vérification

Etape 1 – Analyse

- S'assurer que la spécification est syntaxiquement et sémantiquement correcte en utilisant un outil d'analyse SDL conventionnel.

Etape 2 – Simulation

- Valider/vérifier le comportement normal de la spécification au moyen d'un outil de simulation.

Le but de cette étape est principalement de vérifier que le comportement normal décrit dans la spécification des besoins est correctement couvert par la spécification SDL. Ceci est réalisé en exécutant l'outil de simulation selon l'«usage normal» prescrit, décrit par exemple par des diagrammes de séquences de messages, en vérifiant que ceci est cohérent avec la spécification SDL. La simulation va mettre en évidence les imperfections les plus importantes de la conception (s'il y en a) et va aussi aider à détecter certaines erreurs n'ayant pas été détectées à l'**étape 1**.

Les outils de simulation fournissent une possibilité d'exécuter une spécification SDL, en examinant interactivement le comportement du système SDL spécifié. Les signaux peuvent être envoyés au système et les réponses du système peuvent être vérifiées. Habituellement, il est aussi possible de placer le système dans un certain état prédéfini, par exemple en changeant les valeurs de variables ou en créant des instances de processus, et de continuer la simulation à partir de cet état.

Les outils de simulation pour SDL sont disponibles commercialement de divers vendeurs et doivent être considérés pour constituer une technique reconnue pour la validation et la validation de spécification SDL.

Etape 3 – Exploration de l'espace des états

- Vérifier certaines propriétés spécifiques de la spécification au moyen d'un outil d'exploration de l'espace des états.

Le but de cette étape est de vérifier qu'une spécification SDL donnée de système possède certaines propriétés spécifiques. Ces propriétés peuvent être des propriétés générales comme l'absence de blocage et d'autres situations générales de faute, ou elles peuvent être des propriétés spécifiques du système définies par le concepteur du système SDL. L'analyse avec un outil d'exploration de l'espace des états va révéler des erreurs de conception subtiles, mettant en cause par exemple des signaux inattendus ou des débordements de temporisation peu fréquents, qui sont très difficiles à trouver en faisant appel à la simulation ou à l'inspection manuelle. Ceci quelquefois aussi permet de faire quelques modifications ou extensions à la spécification SDL du système de manière à réduire l'espace des états qu'il faut explorer. En pratique, ceci peut comprendre la limitation du nombre des instances autorisées pour quelques-uns des types de processus, la limitation de la longueur des files d'attente des canaux ou la réalisation d'une spécification plus ou moins détaillée de l'environnement du système.

Pour de petits systèmes SDL, il est possible de vérifier l'absence d'une propriété donnée, mais en général ce n'est pas possible, en raison des problèmes de l'explosion des états. Cependant, même pour les systèmes grands et complexes ce type d'outil s'est révélé très utile comme outil d'analyse automatique, qui réalise une analyse plus importante que celle qui est possible avec les outils de simulation.

Plusieurs outils basés sur l'exploration de l'espace des états existent et sont utilisés, mais ils ne sont pas, pour l'instant, disponibles commercialement et ce domaine peut être encore considéré comme faisant partie du domaine de la recherche.

I.8.3 Test de conformité

Le sujet du *test de conformité* tel qu'il est défini dans [29] met en jeu le test à la fois des possibilités et du comportement d'une mise en œuvre. Les *besoins de conformité statique* définissent les possibilités minimales qui doivent être prises en charge par une mise en œuvre conforme. C'est le comportement externe d'une mise en œuvre qui présente un intérêt, autrement dit le comportement d'une mise en œuvre se définit par rapport à la façon dont on observe dans l'environnement de la mise en œuvre les événements qu'elle peut traiter. Ceci signifie que même si les comportements à la fois interne et externe sont décrits par une norme, c'est seulement les besoins relatifs au comportement externe auquel doit répondre la mise en œuvre. Ces besoins sont considérés comme les *besoins de conformité dynamique*, et sont définis comme une relation portant sur le comportement externe entre la mise en œuvre et la norme de spécification.

De façon courante, les normes sont habituellement établies en langue naturelle, dans certains cas accompagnées de tables d'états, et les suites de test sont développées de façon manuelle. Ceci crée des problèmes lorsque l'on développe les suites de test, puisque il n'y a pas de possibilité de vérifier formellement qu'une suite de test spécifie le même ensemble de besoins sur une mise en œuvre comme le fait une norme.

On croit que l'introduction de langages formels comme SDL dans les normes de spécification est un moyen de résoudre ces problèmes. Lorsque des spécifications formelles sont disponibles, des possibilités vont se faire jour pour la dérivation de suites de test et/ou la validation de suites de test supportées par ordinateur. Beaucoup d'efforts de recherche ont été dépensés dans le domaine de la dérivation de suites de test par ordinateur durant les quelques années passées, et on peut identifier principalement deux approches:

- la dérivation de test automatique;
- la dérivation de test basée sur la simulation.

I.8.3.1 Dérivation de tests automatique

La dérivation de tests automatique utilise soit des techniques disponibles dans le domaine du test du matériel soit des techniques similaires à celles utilisées en validation, c'est-à-dire l'exploration de l'espace des états. La principale caractéristique de l'approche est qu'on calcule une représentation d'un comportement du système observable de manière externe. De cette représentation, la suite de tests est alors automatiquement engendrée, basée sur une description du comportement qui devrait être testé. Ces informations peuvent se baser sur les diagrammes de séquences de messages, [30], s'ils existent, ou sur les besoins informels concernant le système. On devrait noter qu'il est possible de spécifier des besoins portant sur des mises en œuvre qui ne sont pas testables. Il faut en tenir compte lorsque l'on prépare des normes de spécifications. De façon à obtenir des spécifications testables, il est nécessaire d'apporter certaines restrictions à l'usage des constructions du SDL.

Des exemples de techniques utilisées pour calculer le comportement observable sont des techniques basées sur l'exploration de l'espace des états et des techniques basées sur le concept de machines à états finis. Dans une approche plus récente, les processus du système sont transformés en un ensemble de machines à états finis, qui sont alors composées de manière à calculer le comportement observable. Puisque la sémantique du SDL se fonde sur le concept des machines à états finis étendues, il n'est pas, en général, possible de réaliser cette transformation.

I.8.3.2 Dérivation de tests assistée par simulateur

L'utilisation d'un simulateur peut assister la dérivation de tests de différentes manières. D'abord, le concepteur peut devenir familier du comportement de la spécification en simulant la spécification. Lorsqu'est acquise la connaissance des cas de comportement qu'il est pertinent de tester, le simulateur peut assister la dérivation de cas de test en permettant au concepteur des tests de parcourir interactivement la spécification et de définir les cas de test selon le comportement de la spécification. Plusieurs simulateurs pour le SDL sont disponibles commercialement provenant de différents vendeurs. Cependant, la simulation interactive de spécifications peut devenir une tâche difficile et coûteuse en temps si le comportement du système est complexe.

La dérivation de tests peut aussi être partiellement automatisée en utilisant un simulateur spécifique. Dans ce cas, le concepteur des tests guide la génération de test en prenant des décisions de manière interactive sur le comportement à partir duquel on engendre les tests, mais la génération réelle de tests est réalisée automatiquement.

I.9 Documents auxiliaires

Les documents auxiliaires contiennent des informations qui peuvent être extraites (manuellement ou au moyen d'un outil) d'une spécification SDL de système. Le but est de fournir une vue d'ensemble. Cependant, ces documents peuvent aussi être créés avant la spécification SDL de système et peuvent alors servir de base pour elle.

I.9.1 Communication et spécification d'interface

I.9.1.1 Introduction

Dans de nombreuses situations, on souhaite spécifier l'interface d'une entité, comme partie des spécifications de besoins de cette entité. Ce paragraphe contient une discussion générale de la spécification de communication et d'interface, qui sert de base aux paragraphes suivants, où différentes approches de ce type de spécifications seront présentées.

Pour commencer, on doit faire une distinction entre *communication* et *interface* (dynamique). La communication est supposée prendre place entre deux ou plusieurs parties, et toutes les parties concernées ont une certaine influence sur le comportement combiné qui en résulte.

Une interface, d'un autre côté, s'applique à une seule entité[7)]. Une entité peut avoir plus d'une interface. Une spécification d'interface exprime le comportement possible de l'entité à l'interface donnée, sans considérer le comportement aux autres interfaces de l'entité. Le comportement à une interface donnée est dit possible, puisqu'il n'est influencé que par l'entité. Lorsqu'une interface est connectée à une autre entité quelconque, alors la communication entre les deux entités va consister en un sous-ensemble du comportement possible à l'interface.

Lorsqu'on considère une interface, il est habituel de considérer que l'entité est connectée à l'*environnement* via l'interface. L'environnement, cependant, n'impose aucune restriction sur le comportement de l'entité, observé à l'interface.

---

7) Cette terminologie est «en ligne avec» les concepts de modélisation utilisés par l'ISO et le CCITT pour le traitement réparti ouvert (ODP) (*open distributed processing*). Il est reconnu que le terme «interface» est aussi utilisé dans un sens différent dans d'autres travaux de normalisation.

Dans le contexte du SDL, une telle situation est illustrée par un bloc *A* connecté à l'environnement par des canaux *I1* et *I2*, ou un type de bloc *A* ayant comme interfaces des portes *I1* et *I2*, comme le montre la Figure I.9-1. Noter qu'un canal connectant deux blocs est un moyen de communication et non une interface. L'entité pourrait aussi être un type de processus, mais cette possibilité n'est pas considérée explicitement dans la suite.

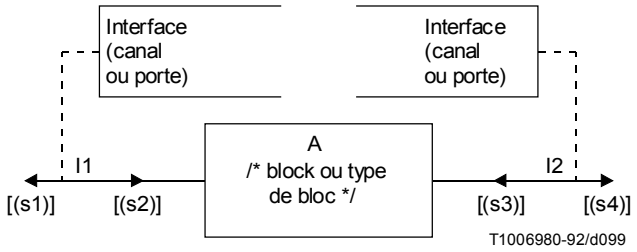


FIGURE I.9-1/Z.100

Interface d'un bloc ou d'un type de bloc

Les spécifications des canaux/portes (en utilisant les listes de signaux *s1*, *s2*, *s3* et *s4* et les spécifications de signaux correspondantes qui ne sont pas montrées ici) sont aussi la spécification de l'interface *statique*. Pour la spécification d'une interface *dynamique* ou communication nous pouvons utiliser :

- un diagramme de séquences de messages (MSC);
- une algèbre de processus; ou
- une sous-structure de canal.

Ces trois approches vont être décrites ci-dessous. La spécification d'interface *dynamique* ou de la communication est applicable lorsqu'un MSC peut être utilisé, comme par exemple à l'**étape 2** et à l'**étape 5** de I.3.

Noter que normalement chaque interface est spécifiée séparément. Ceci est une des raisons pour laquelle la spécification du bloc ou type de bloc n'est pas très utile dans ce contexte, puisqu'une telle spécification couvre le comportement *combiné* satisfaisant simultanément tous les interfaces du bloc ou du type de bloc.

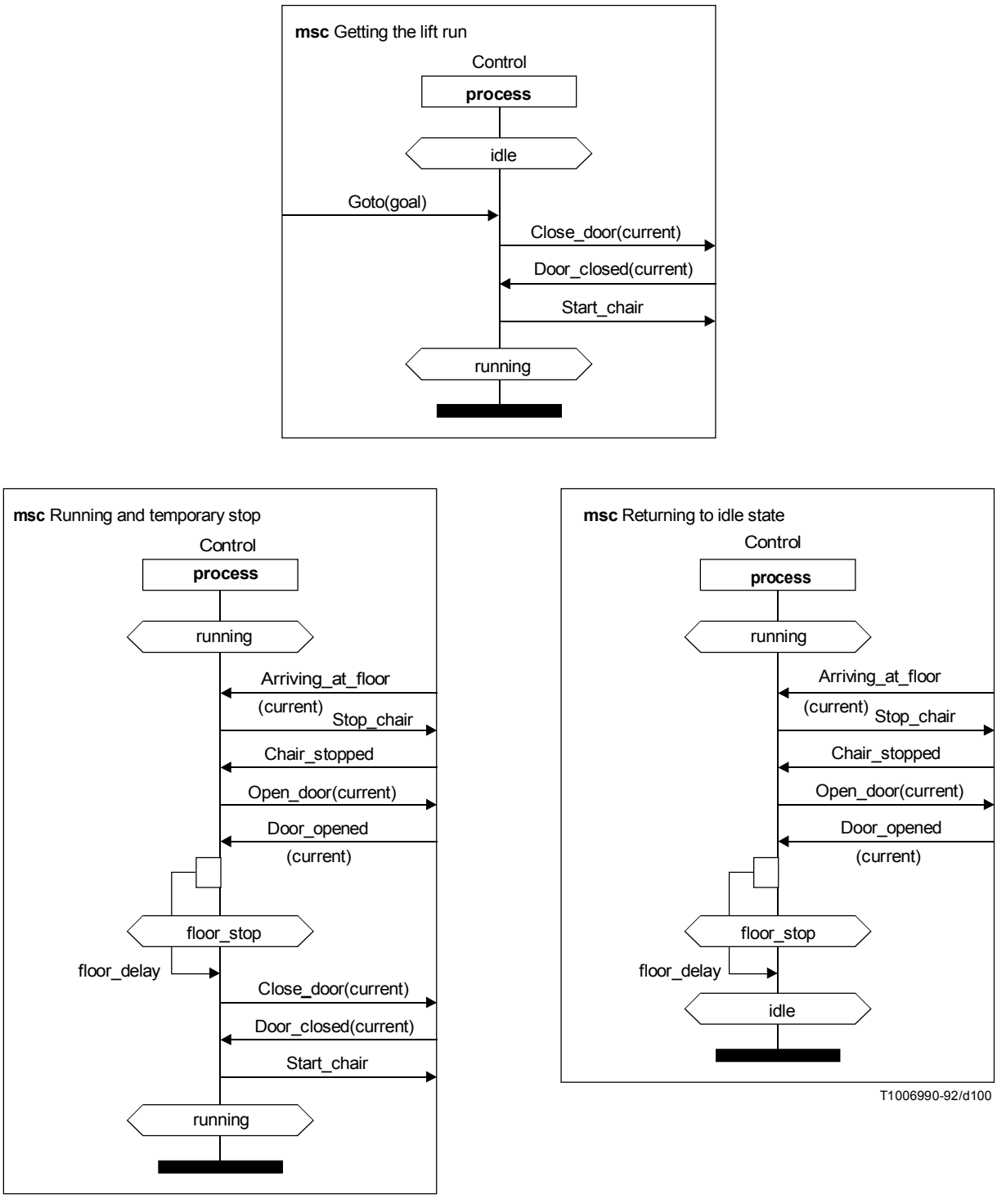
I.9.1.2 Utilisation de diagrammes de séquences de messages

L'utilisation de diagrammes de séquences de messages est traitée en détail en I.6. Comme il a été énoncé dans cette section, un MSC peut être utilisé dans plusieurs cas, incluant l'énoncé des besoins pour les spécifications SDL, la spécification semi-formelle de communication et la spécification d'interface.

Les différentes utilisations d'un MSC sont corrélées, une spécification d'interface peut par exemple être considérée comme faisant partie d'une spécification de besoin. En fait, il n'y a pas de différences entre utiliser un MSC pour une spécification d'interface et pour la spécification d'une communication. Ceci en raison du fait qu'un MSC couvre un comportement partiel (une des nombreuses réponses possibles d'une entité ou de l'environnement à un certain stimulus).

Pour faciliter la comparaison des trois approches mentionnées ci-dessus pour la spécification d'une interface dynamique ou d'une communication, on présente ici l'utilisation de MSC pour l'exemple de l'ascenseur (*Lift*). Quelques MSC avaient déjà été introduits en I.3, à l'**étape 5**, pour le cas de l'utilisateur unique (*Lift ride* et *Lift request*).

Le cas d'utilisateurs multiples peut aussi être décrit par des MSC, comme le montre l'exemple I.9-1. Puisqu'un MSC décrit seulement le cas d'un utilisateur unique, il est avantageux d'avoir un certain nombre de petits MSC pouvant se combiner de différentes façons. Les MSC *Running et temporary stop* peuvent évidemment être «ajoutés» au MSC *Getting the lift run* un certain nombre de fois avant d'ajouter le MSC *Returning to idle state*.



EXEMPLE I.9-1

MSC pour le cas d'utilisateurs multiples

I.9.1.3 Utilisation des algèbres de processus

Introduction

Dans ce paragraphe est présentée l'utilisation d'une algèbre de processus pour la spécification de communication et d'interface. L'algèbre de processus est basée sur LOTOS, et est désignée par LOTOS*. Le principal avantage d'utiliser LOTOS* pour la spécification de communication et d'interface comparée aux MSC est l'obtention d'une spécification complète et plus compacte. On devrait noter que l'usage combiné de LOTOS* et de SDL, comme il est annoncé dans ce paragraphe, ne possède pas une base théorique solide. De plus, LOTOS* n'est pas complètement conforme avec la définition officielle du langage LOTOS.

Lorsqu'on utilise LOTOS* pour la spécification de communication et d'interface, aucun nouveau document n'est nécessaire. A la place, les expressions comportementales LOTOS* peuvent être incluses dans les spécifications de porte ou de canal sous forme de commentaires. Une expression comportementale LOTOS* définit par sa sémantique opérationnelle un ensemble de traces, c'est-à-dire les séquences possibles de signaux traversant la porte ou le canal.

L'utilisation de LOTOS* pour la spécification de communication et d'interface suppose quelques hypothèses particulières. Le mécanisme de communication du SDL est asynchrone tandis que celui de LOTOS est synchrone. Par conséquent, les mécanismes de synchronisation de LOTOS ne doivent pas être employés. De plus, la communication peut toujours être considérée comme si elle se tenait entre deux parties, le bloc et son environnement. Lorsqu'on ne considère que le comportement observable, ces parties peut être supposées contenir chacune seulement un processus SDL. Donc, aucun parallélisme n'est présent et il suffit de ne considérer que la partie séquentielle de LOTOS.

Dans les spécifications d'interface il y a deux sortes d'événements; l'entrée et la sortie d'un signal asynchrone. Pour exprimer cet aspect en LOTOS*, nous avons besoin de déclarations de valeur et de variable à une certaine porte. Ensuite, nous avons les expressions suivantes qui se correspondent:

SDL	LOTOS*
input a;	? a
output a;	! a

Noter que le nom de porte est omis dans l'expression LOTOS* pour des raisons pratiques. Puisqu'une porte ne se synchronise pas avec autre chose, rien ne nous empêche de considérer les offres de valeur et de variable à la porte comme la consommation et l'envoi d'un signal asynchrone.

Opérateurs LOTOS*

LOTOS* offre plusieurs opérateurs qui peuvent être utilisés dans les spécifications de communication et d'interface. Quelques-uns d'entre eux s'expriment directement en SDL, tandis que d'autres offrent une représentation plus compacte que les constructions correspondantes du SDL.

- **Préfixe d'action**: Le préfixe d'action (;) peut être utilisé pour préfixer une expression comportementale existante par une action, comme *a;exp*. Ceci signifie que l'exécution de l'action *a* est immédiatement suivie de l'exécution de l'expression comportementale existante *exp*. Dans le SDL, le préfixe d'action correspond à l'ordre implicite des actions dans une transition.

- **Choix**: L'opérateur de choix ([ ]) signifie un choix non déterministe entre deux expressions comportementales. Il peut s'exprimer de deux façons différentes en SDL. S'il existe un choix entre signaux à recevoir, alors celui-ci est pris en charge par un état auquel les différentes entrées sont connectées. Dans les autres cas, on utilise une décision.

- **Entrelacement**: Comme il a déjà été mentionné, nous n'avons pas besoin de considérer la synchronisation dans les spécifications d'interface. Ceci nous laisse seulement l'entrelacement pur (| | |), où les actions des expressions qu'il relie peuvent s'entrelacer de manière arbitraire.

- **Validation**: L'opérateur de validation (>>) signifie un ordre séquentiel de deux expressions comportementales de telle sorte que l'expression comportementale de la partie droite n'est exécutée qu'après la terminaison avec succès (avec un **exit**) de l'expression comportementale de la partie gauche. Ceci correspond à l'état suivant en SDL, ou à l'ordre normal des actions dans une transition.

- **Invalidation**: L'opérateur d'invalidation ([>) signifie que l'expression comportementale de la partie droite, lorsqu'elle est exécutée, va interrompre immédiatement l'expression comportementale de la partie gauche. L'invalidation peut s'exprimer en SDL d'une manière habile par un état «astérisque», suivi par la transition qui interrompt le comportement normal.

- **Sortie** (**Exit**): Signifie la terminaison avec succès d'une expression comportementale et correspond à l'état suivant du SDL.
- **Arrêt** (**Stop**): A la même signification que **stop** en SDL.
- **Récursivité** (**Recursion**): Correspond à l'entrée dans un état antérieur de manière répétée.

Exemple:

Considérer le système SDL *Lift* décrit au I.3.3. Le bloc *Control*, possède trois interfaces, un pour chaque canal qui lui est connecté. Ceux-ci sont représentés par des portes en LOTOS*. Les spécifications d'interface pour *Chair* et *Floors* sont présentées dans l'exemple I.9-2.

```
Chair_interface :=
        ?Goto; Chair_interface
   [ ]
        ?Emergency_stop; ?Restart; Chair_interface
Floor_interface :=
        Open_door >> Floor_request >> Close_door >> Running
Running :=
        ?Floor_req; Running
   [ ]
        ?Arriving_at_floor; (Running [ ] Open_door >> Floor_stop)
Floor_stop :=
        ?Floor_req; Floor_stop
   [ ]
        i; (Close_door; Running [ ] Floor_request >> Close_door >> Running)
Floor_request :=
        ?Floor_req; (Floor_request [ ] exit)
Open_door :=
        !Stop_chair; ?Chair_stopped; !Open_door; ?Door_opened; Exit
Close_door :=
        !Close_door; ?Door_closed; !Start_chair; Exit
```

EXEMPLE I.9-2

Spécifications d'interface LOTOS*

Comparer ceci avec la spécification du processus *Control* au I.3.3. Comme on peut le voir, les exécutions répétées des transitions dans un état s'expriment en LOTOS* par des processus récursifs, comme *Running*.

Comme il est mentionné plus haut, la spécification d'interface peut être utilisée comme spécification de besoins pour un bloc SDL. Lorsqu'on spécifie le bloc (et ses processus), il faut normalement prendre des décisions supplémentaires quant au comportement détaillé et à la combinaison de comportements à chaque interface. Des exemples de telles décisions sont: Que se passe-t-il lorsque *Floor_req* ou un autre *Goto* arrive lorsque l'ascenseur est en mouvement? Quelles informations faut-il transmettre dans les signaux et quelles informations faut-il ranger dans le bloc? Néanmoins, une spécification de squelette de processus SDL peut être automatiquement extraite de la spécification d'interface.

Sémantique et conformité de spécifications LOTOS*

Maintenant nous allons discuter brièvement de la relation entre des spécifications exprimées en LOTOS* et une spécification utilisant le SDL complet.

Pour commencer, il nous faut construire un modèle pour la sémantique dynamique d'un système composé de processus SDL. Dans ce but, on utilisera les arbres de communication asynchrone (ACT) (*asynchronous communication trees*). Ce modèle se base sur les systèmes d'acteurs et leurs diagrammes d'événements [23]. La définition de l'arbre indique quelles sont les séquences possibles des entrées et des sorties observables de signaux. La définition d'un arbre de communication asynchrone d'un système SDL et de son environnement fournit la base sémantique nécessaire qu'il faut pour formaliser la conformité d'une spécification de processus SDL à une spécification d'interface.

L'étape suivante consiste à définir un arbre similaire pour les spécifications d'interface. Ceci n'est guère que la sémantique opérationnelle de LOTOS* et les graphes de transitions étiquetées basés dessus [31]. Intuitivement, les chemins de ce graphe de transitions représentent les séquences d'actions autorisées par l'expression comportementale correspondante de la même façon qu'un arbre de communication asynchrone.

La conformité d'une spécification de processus SDL à une spécification d'interface peut ensuite être définie comme une relation binaire entre les nœuds des arbres de communication asynchrone d'une spécification de processus SDL et la spécification d'interface. Intuitivement, il faut remplir trois conditions:

- tout ce que peut faire le processus SDL doit être permis par la spécification d'interface, qui exprime le besoin normal pour réaliser l'opération de raffinage;
- le processus SDL doit, à tout instant, être prêt à accepter n'importe quel signal qui peut lui être envoyé en respectant la spécification d'interface;
- le processus SDL doit effectuer ses tâches jusqu'au bout, c'est-à-dire qu'il ne peut les interrompre à tout moment selon sa propre décision.

Un point intéressant de la conformité concerne la distinction entre les signaux d'entrée et les signaux de sortie. Il est intuitivement clair que si, à un instant quelconque, plusieurs signaux différents peuvent être envoyés au processus en respectant la spécification d'interface, le processus doit être capable de les considérer tous. Si quelques-uns d'entre eux sont mis de côté et ne conduisent pas à une réponse anticipée selon la spécification d'interface, la spécification SDL détaillée correspondant à la spécification d'interface peut ne pas être correcte. Cependant, si plusieurs signaux de réponse à un seul signal d'entrée sont possibles, la liberté de choix est laissée au spécifieur qui choisit celui d'entre eux à utiliser. A cet égard, les besoins exprimés par les signaux d'entrée et de sortie sont doubles.

I.9.1.4 Utilisation des sous-structures de canal

Une sous-structure de canal n'est pas un document auxiliaire, mais elle peut être utilisée pour la spécification de communication et d'interface dynamique. On montre ici comment compléter cette illustration. Pour les canaux *Floors* du système *Lift* dans I.3.3, on introduit la sous-structure de canal *Floorsub*, comme le montre l'exemple I.9-3.

La partie dynamique de la spécification d'interface est donnée par la spécification de processus *Floor*. Celle-ci devrait être équivalente à la spécification de processus LOTOS* *Floor_interface* de l'exemple I.9-2.

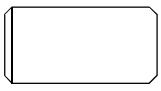
Le débordement de temporisation s'exprime par une transition spontanée et les décisions non spécifiées par des décisions non déterministes. Les procédures *Open_door* et *Close_door* sont spécifiées dans I.3.3 comme faisant partie de la spécification du processus *Control*.

I.9.2 Diagramme d'arbre

Un diagramme d'arbre montre les composants d'un système SDL. Les composants (également appelés nœuds dans la suite) forment une structure hiérarchique, ayant le système pour racine, selon les relations entre contenu et contenant exprimées par les règles syntaxiques. Les nœuds du diagramme d'arbre peuvent être (en plus du nœud système) :

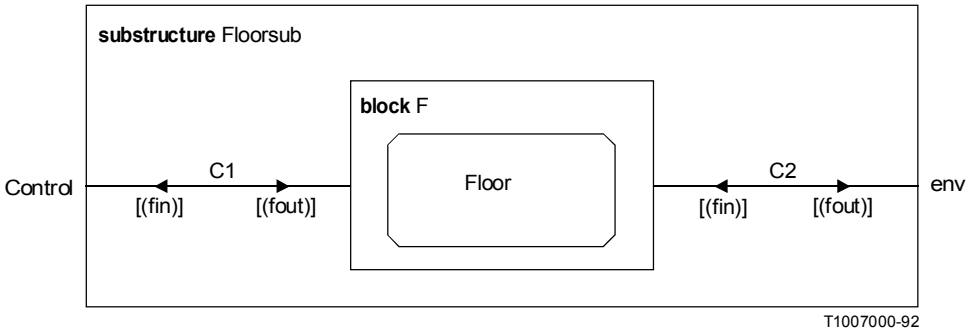
- les blocs seulement, dans ce cas il s'agit d'un *diagramme d'arbre de blocs*, dont un exemple est fourni à la Figure I.9-2;
- les blocs, les processus et les services, dans ce cas il s'agit d'un *diagramme d'arbre de base*, dont un exemple est fourni à la Figure I.9-3;
- en plus des nœuds du *diagramme d'arbre de base* aussi des procédures et des macros etc., dans ce cas il s'agit d'un *diagramme d'arbre général*, dont un exemple est fourni à la Figure I.9-5;

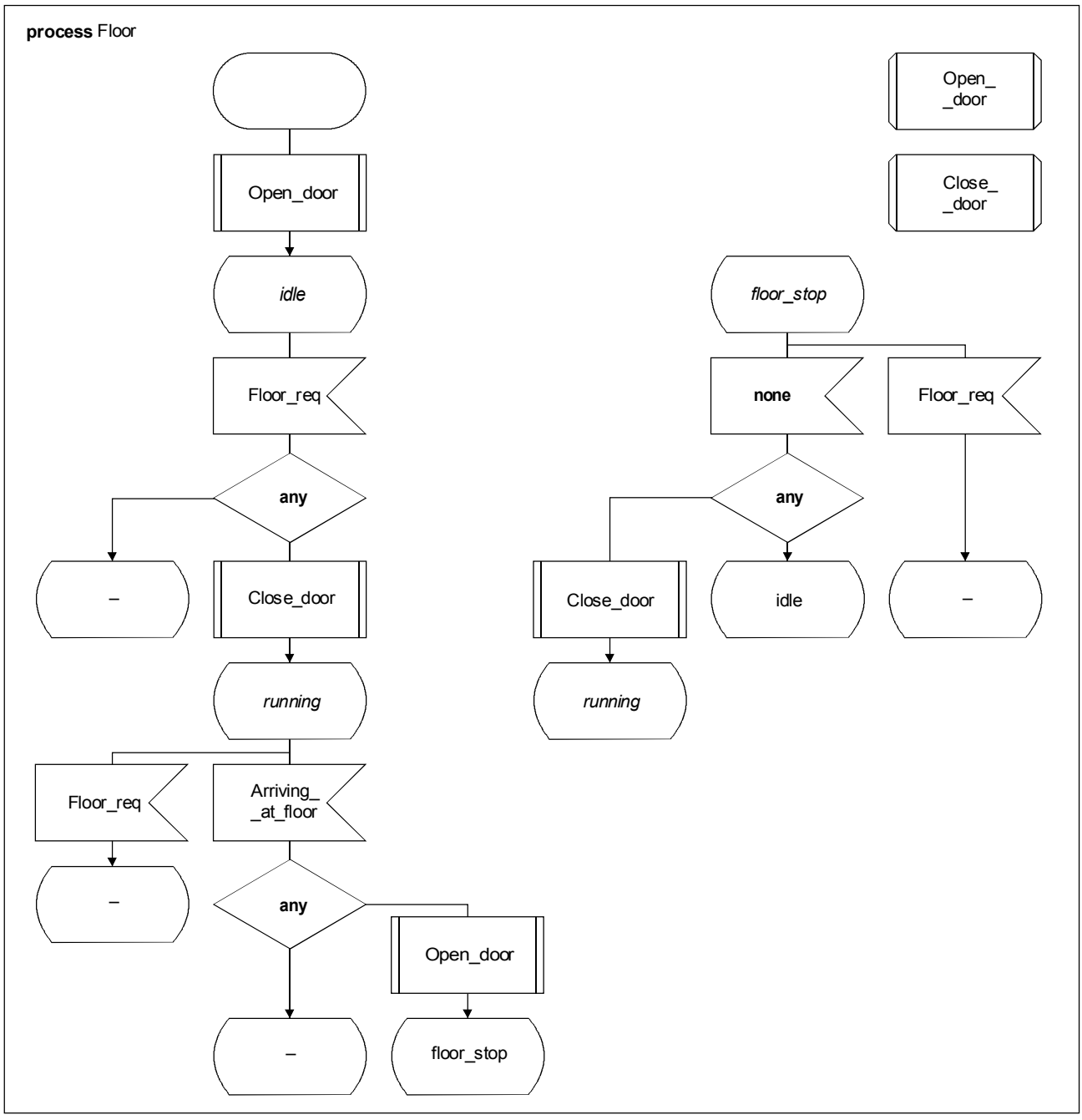
Noter que les différentes branches d'un diagramme d'arbre n'ont pas besoin d'avoir le même nombre de nœuds. Tous les nœuds représentent des définitions d'entités. En ce qui concerne le nœud correspondant à une macro, un nouveau symbole a été introduit



T1007020-92/d102

puisque aucun symbole n'est défini dans la présente Recommandation pour la définition de macro. Noter que le symbole n'est pas normatif.





EXEMPLE I.9-3

Sous-structure du canal *floors*

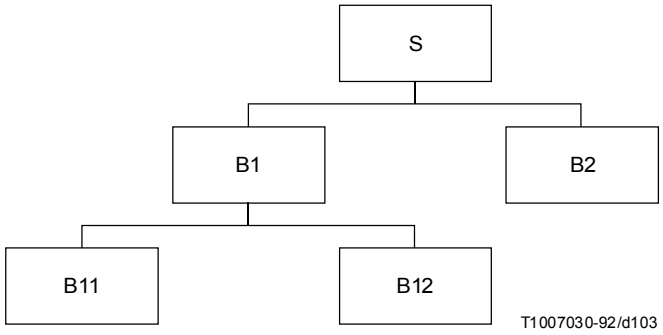


FIGURE I.9-2/Z.100

Un diagramme d'arbre de blocs

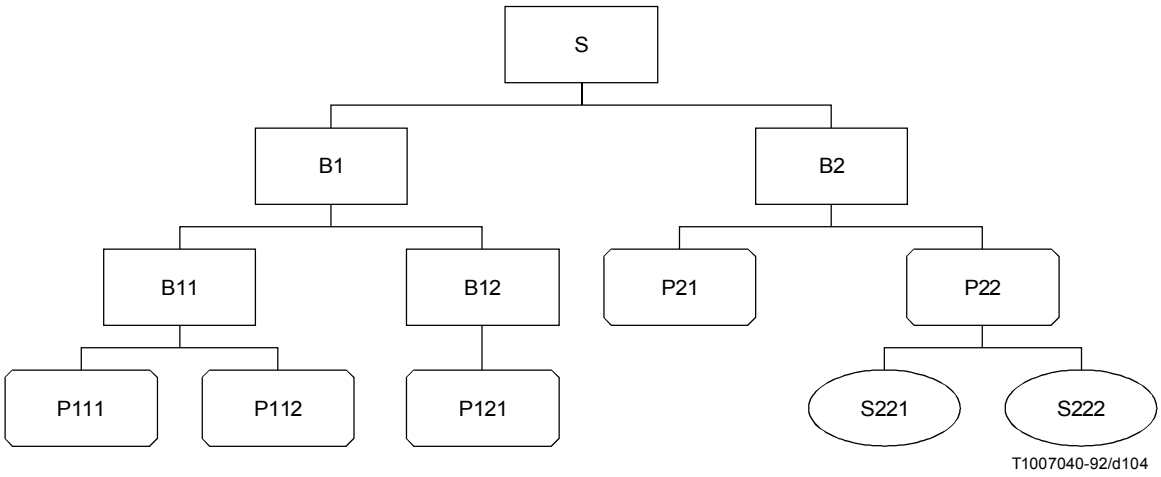


FIGURE I.9-3/Z.100

Un diagramme d'arbre de base

Les nœuds de macro sont toujours des nœuds terminaux et peuvent être attachés à tout autre nœud. Un nœud de procédure peut être attaché à tout autre nœud, sauf un nœud de macro.

Le diagramme devrait être dessiné de préférence en donnant une taille uniforme à tous les symboles de nœud. Ceci permet aux nœuds de même niveau de partitionnement d'apparaître à un niveau uniforme dans le diagramme.

Un diagramme d'arbre général peut aussi contenir une sous-structure de canal, comme le montre la Figure I.9-4. Dans cet exemple, se trouve une sous-structure de canal bidirectionnel *C1* entre les blocs *B1* et *B2*. Le nœud de sous-structure de canal est analogue à un nœud de bloc, et peut ainsi être la racine d'un arbre de nœuds similaires.

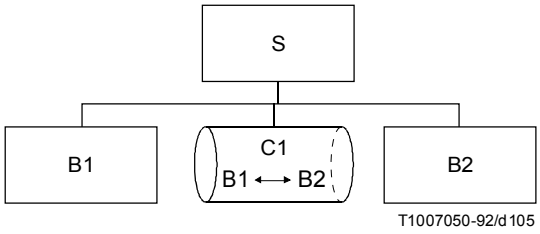


FIGURE I.9-4/Z.100

Un diagramme d'arbre général contenant une sous-structure de canal

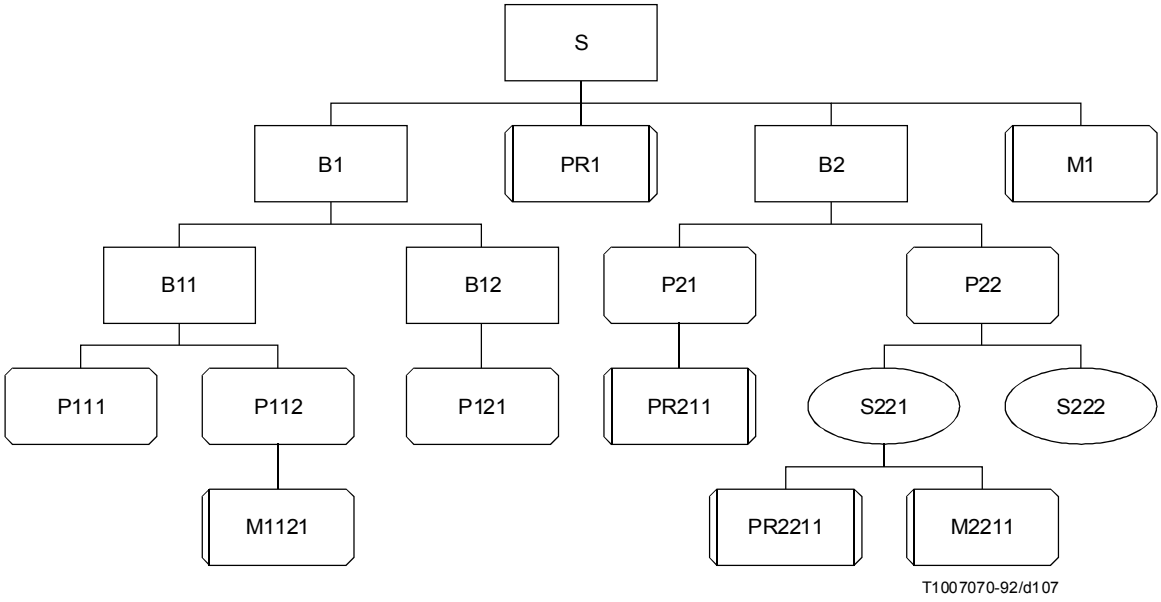
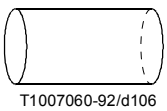


FIGURE I.9-5/Z.100

Un diagramme d'arbre général

Pour le nœud de sous-structure de canal un nouveau symbole a été introduit



puisque aucun symbole particulier n'est défini dans la présente Recommandation pour la définition de sous-structure de canal. Noter que ce symbole n'est pas normatif.

Il est souvent utile de partitionner un diagramme d'arbre en diagrammes partiels, par exemple si le diagramme est si grand qu'il nécessite plus d'une page. La découpe en plusieurs diagrammes partiels se fait de telle manière que les racines des diagrammes supplémentaires apparaissent en tant que nœuds terminaux du premier diagramme (voir la Figure I.9-6).

Comme il n'est pas évident qu'un nœud terminal de diagramme soit partitionné davantage dans d'autres diagrammes et/ou trouver les diagrammes assurant la continuation, on devrait insérer des références en utilisant le symbole de commentaire.

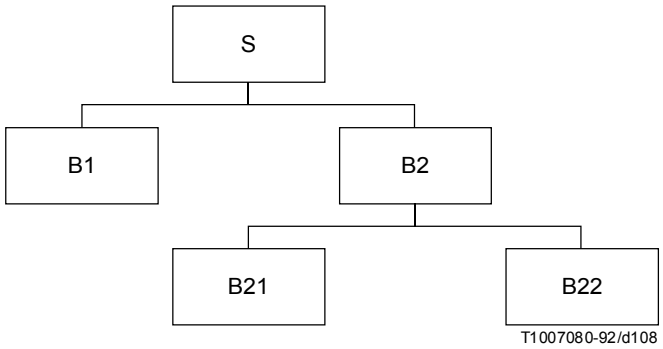


FIGURE I.9-6a/Z.100

Partitionnement d'un arbre général – Diagramme complet

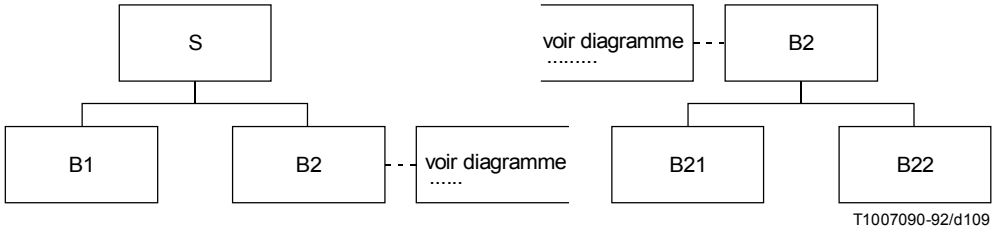


FIGURE I.9-6b/Z.100

Partitionnement d'un arbre général – Diagramme partiel

I.9.3 Diagramme de vue d'ensemble des états

L'objectif du diagramme de vue d'ensemble des états est de fournir une vue d'ensemble des états d'un processus et de montrer les transitions possibles entre elles. Le diagramme devient très vite compliqué lorsque le nombre d'états et de transitions augmente. Par conséquent, il ne s'applique qu'aux cas simples, ou lorsqu'il existe des états ou des transitions «si peu importants» qu'ils peuvent être omis.

Les diagrammes sont composés de symboles d'états, d'arcs orientés représentant les transitions, et éventuellement des symboles de début et d'arrêt.

Le symbole d'état devrait contenir le nom de l'état auquel il se rapporte. Le symbole peut contenir plusieurs noms d'état ou une notation astérisque (*).

A chacun des arcs orientés on peut associer le nom du signal, ou l'ensemble des signaux qui provoque la transition aussi bien que les signaux émis pendant la transition. La liste des signaux envoyés est précédée du caractère /. Les signaux de temporisateur et l'armement de temporisateurs peuvent être traités comme des signaux ordinaires. Un exemple de diagramme de vue d'ensemble d'états pour le processus *Control* défini au I.3.3 est donné à la Figure I.9-7.

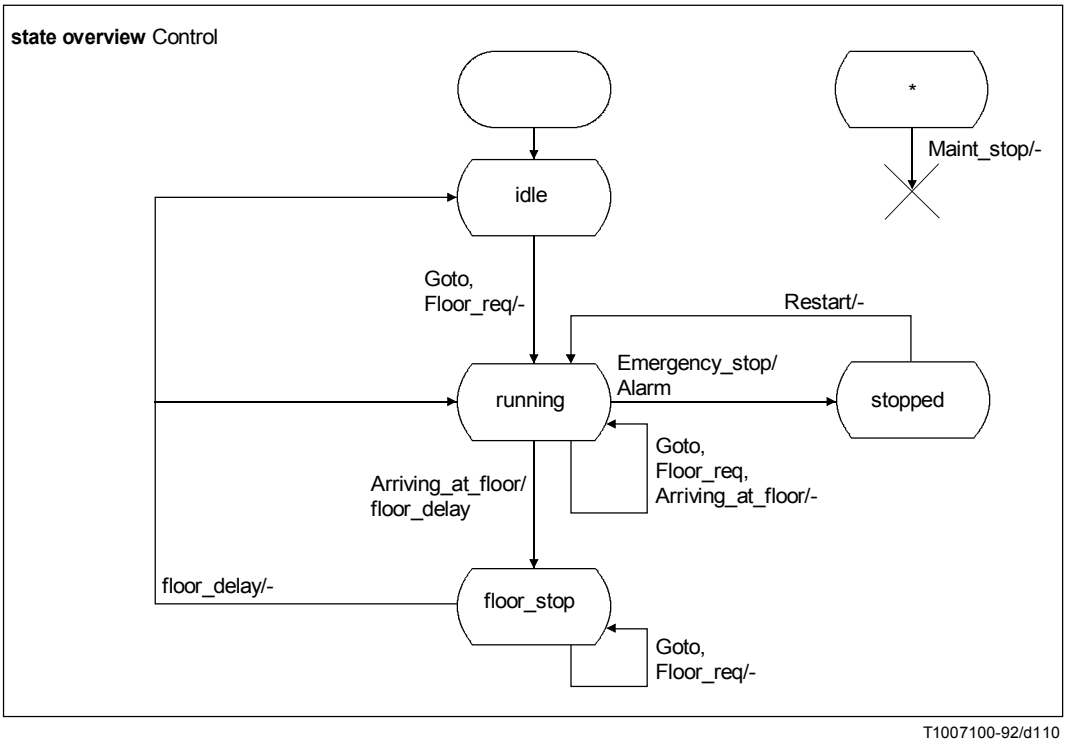


FIGURE I.9-7/Z.100

Un diagramme de vue d'ensemble d'états

I.9.4 Matrice signaux/états

La matrice signaux/états constitue une alternative au diagramme de vue d'ensemble d'états contenant exactement les mêmes informations. Elle peut être utilisée, cependant, aussi lorsque le nombre d'états et de transitions est grand. De plus, elle offre la possibilité de vérifier que toutes les combinaisons entre les états et les signaux d'entrée sont considérées. Consulter la Figure I.9-8, qui correspond au diagramme de vue d'ensemble des états de la Figure I.9-7.

Le diagramme consiste en une matrice à deux dimensions indexée sur un axe par tous les états du processus et de l'autre par tous les signaux d'entrée valables pour le processus. Dans chacun des éléments de la matrice l'état suivant est fourni avec les sorties possibles durant la transition. On peut donner une référence sur l'endroit où trouver la combinaison donnée par les indices, si celle-ci existe.

L'élément correspondant à l'état fictif «début» et au signal vide est utilisé pour montrer l'état initial du processus.

La matrice peut être partitionnée en matrices partielles présentées sur différentes pages. Les références sont les références normales utilisées par l'utilisateur dans la documentation.

De préférence, les signaux et les états devraient être groupés ensemble, de sorte que chaque matrice partielle couvre un aspect du comportement du processus.

I.10 Documentation

I.10.1 Introduction

L'ISO définit un document comme une *quantité d'information limitée et cohérente rangée sur un support sous une forme récupérable*. Par conséquent, un document devrait être considéré comme une unité logique qui est strictement délimitée. Les documents sont utilisés pour transporter toutes les informations relatives à un système qui est spécifié en utilisant le SDL.

	signal state	Control					
	state → signal ↓	'start'	*idle*	*running*	*floor_stop*	*stopped*	
		idle/-					
	Goto		running/-	-/-	-/-		
	Floor_req		running/-	-/-	-/-		
	floor_delay				idle, running/-		
	Arriving_ _at_floor			floor_stop/ floor delay; -/-			
	Emergen_ cy_stop			stopped/ Alarm			
	Restart					running/-	
	Maint_stop		'stop'/-	'stop'/-	'stop'/-	'stop'/-	

FIGURE I.9-8/Z.100

Une matrice signaux/états

Lorsque le papier est utilisé comme support physique pour ranger un document, le terme document s'applique plutôt injustement aux feuilles de papier qu'à leur contenu logique. Avec l'usage croissant des moyens de stockage magnétiques, le terme a repris son sens originel.

Ce paragraphe concerne l'organisation logique des documents plutôt que leur organisation physique. Ce dernier aspect est laissé à l'appréciation de l'utilisateur. La similitude des besoins concernant à la fois l'organisation logique et l'organisation physique des documents signifie que quelques conseils utiles peuvent être proposés dans le texte suivant pour aider un utilisateur à mettre en place une organisation physique pour les documents.

En répartissant l'information en un nombre satisfaisant de documents, la description du système peut être rendue plus facile à lire et à gérer. Une structure de documentation devrait fournir à la fois la vue d'ensemble et les détails.

Les propriétés d'un document sont les suivantes:

- identification unique;
- désignation de la révision;
- taille gérable;
- fait (généralement) partie d'une structure de documentation;
- ne fait pas partie d'un autre document (c'est-à-dire que les documents ne doivent pas être imbriqués);
- est (généralement) découpé en pages.

Le SDL ne recommande pas de manière intangible des documents ou des structures de documentation. Cependant, quelques constructions du langage sont fournies pour aider l'utilisateur à manipuler les documents. Ceux-ci sont aussi couverts dans ce paragraphe de façon à le rendre complet par lui-même.

I.10.2 Support du langage pour la documentation

Spécifications imbriquées

Une spécification SDL contient, comme beaucoup d'autres langages, une hiérarchie de composants qui sont organisés selon une structure arborescente. Ceci amène la spécification de système à posséder un nombre de niveaux de hiérarchie ou d'abstraction. La structure syntaxique traditionnelle d'une spécification de système SDL est illustrée à la Figure I.10-1. Des spécifications imbriquées y sont montrées avec les spécifications de rang le plus bas contenues dans les spécifications du rang immédiatement supérieur. Ceci peut être comparé à un diagramme de circuit qui tient entièrement sur une simple feuille.

Bien que l'imbrication des spécifications soit sûrement souhaitable du point de vue du réalisateur d'outils et pour de très petites spécifications, il pose à la personne utilisatrice les problèmes suivants:

- il ne fournit pas de vue d'ensemble;
- il ne fournit pas de séparation entre les niveaux d'abstraction;
- il y a trop d'information dans un seul endroit;
- il est difficile d'établir une correspondance entre les documents et les spécifications.

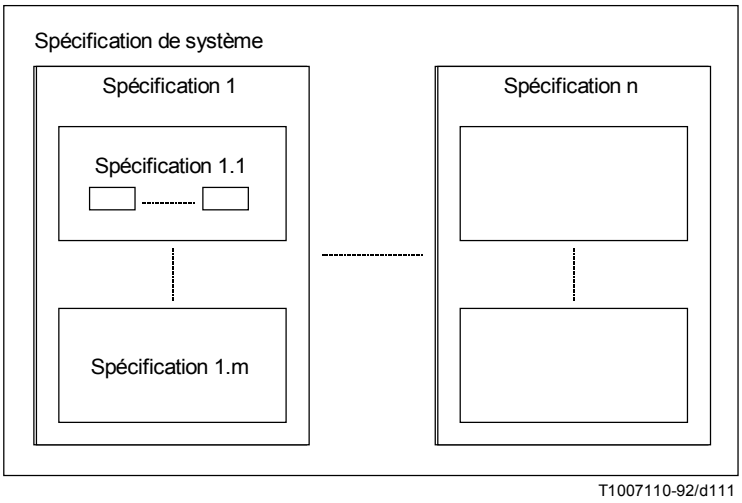


FIGURE I.10-1/Z.100

Structure syntaxique traditionnelle d'une spécification de système

Spécifications distantes

En SDL, ces problèmes ont été résolus en introduisant la construction dite de **spécification distante (remote specification)**. Une spécification distante est une spécification qui a été retirée de son contexte de définition pour mettre en évidence la vue d'ensemble. Ceci est similaire à l'appel et à la définition de procédure, mais l'«appel» n'est réalisé qu'à un seul endroit (le contexte de définition) en utilisant une **référence (reference)**. En d'autres termes, il y a une correspondance bi-univoque entre la référence et la spécification distante (voir la Figure I.10-2). Ceci peut être comparé à un diagramme de circuit qui est présenté comme un diagramme de bloc, où les blocs sont décrits dans des pages séparées comme des circuits avec des composants électroniques.

Une spécification de système qui fait usage de spécifications distantes est une représentation «à plat», que l'on oppose à la représentation hiérarchique lorsque l'on utilise des spécifications imbriquées.

Mélange de représentation graphique et textuelle

En utilisant des spécifications distantes, les formes graphiques et textuelles peuvent être mélangées selon le souhait de l'utilisateur, comme le montre la Figure I.10-3.

C'est une bonne pratique de commencer par un diagramme de système pour donner une vue d'ensemble. Les spécifications qui nécessitent beaucoup de texte devraient être fournies sous la forme d'une représentation textuelle. Ceci est illustré, par exemple, dans l'exemple de l'ascenseur en I.3.

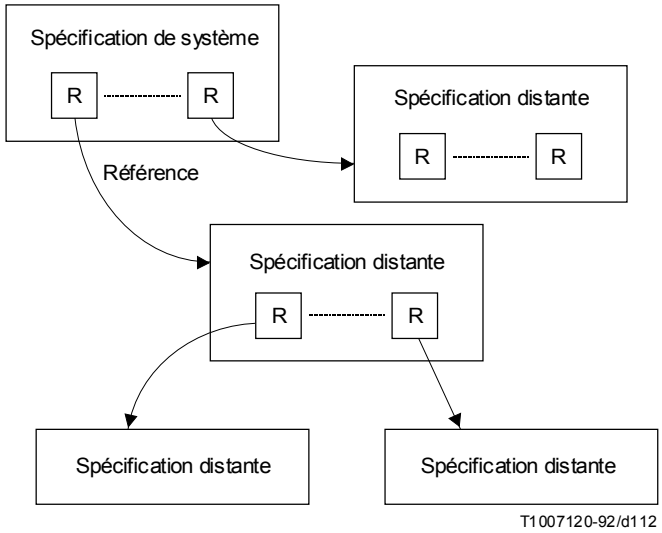


FIGURE I.10-2/Z.100

Spécifications distantes

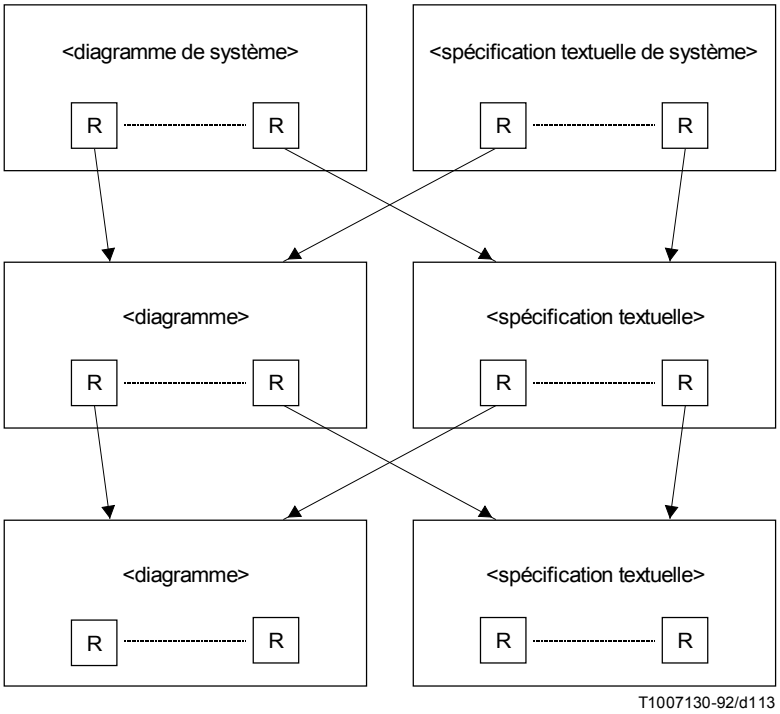


FIGURE I.10-3/Z.100

Mélange de représentation graphique et textuelle

I.10.3 Correspondance entre spécifications et documentation

En considérant une spécification de système comme un ensemble de spécifications distantes, on peut assimiler un document à un contenant de une ou plusieurs de ces spécifications:

<document> ::=

<spécification de système>

| {<spécification distante> }+

Notez qu'une spécification distante peut contenir des spécifications imbriquées.

Si la spécification de système est de petite taille et imbriquée, un seul et unique document est suffisant. Si l'on utilise une représentation «à plat», on peut utiliser davantage de documents, par exemple, un document par spécification (diagramme).

Le cas normal est probablement un mélange de représentations «à plat» et de représentations imbriquées. Lorsque l'on décide de ce mélange, on applique les règles suivantes:

- une spécification ne devrait pas être découpée entre plusieurs documents;
- si une spécification doit être placée dans un document séparé, ce doit être une spécification distante;
- lorsque l'on utilise le concept de page logique pour répartir un diagramme sur plusieurs pages de diagramme, les pages de diagramme doivent correspondre aux pages physiques du document;
- si un diagramme occupe plus d'une page, ce doit être une spécification distante;
- une zone allouée à un bloc, à un processus et à une interaction de service doit tenir sur une seule page.

La syntaxe graphique fournit les moyens de numéroter chaque page logique d'un diagramme, bien que ceci ne soit pas absolument nécessaire. L'ordre des pages n'a normalement aucune importance, ou peut être déduit de n'importe quelle manière. La chose importante est de répéter l'en-tête sur chaque page de façon à voir à quel diagramme la page appartient.

Dans la représentation textuelle, le langage ne fournit aucun moyen de numéroter les pages, bien que ceci soit crucial pour éviter les ambiguïtés. Il paraît raisonnable de suivre la convention adoptée par les langages de programmation, qui laisse le mécanisme de numérotation des pages en dehors du langage.

I.10.4 Les résultats de la documentation

I.10.4.1 Une structure de documentation

L'ensemble des documents couvrant un système complet doit être structuré de sorte que la relation entre un document et les autres documents soit claire. Un document contient normalement des informations se rapportant au système entier ou à un de ses composants, et ceci doit également être clair. Ceci est obtenu généralement en attachant un document de vue générale à chaque composant, contenant une référence à tous les autres documents qui appartiennent au composant. La relation entre les composants est donnée par la structure du système (voir la Figure I.10-4).

Si la spécification SDL de bas niveau (voir I.7) reflète la structure du système mis en œuvre, alors la description de la structure du système peut être engendrée automatiquement par un outil à partir de la spécification du système SDL.

Cependant, la spécification SDL couvre seulement une photographie du système, qui peut changer au cours du temps, vraisemblablement plusieurs fois par an. Pour la plupart des systèmes de la vie courante, il faut fournir des informations supplémentaires pour la gestion de la configuration et de la révision, pour le classement, pour la production, etc. Aussi la nature et les sortes de composants de système nécessitent une élaboration plus poussée. Ces questions sont discutées dans les paragraphes suivants.

I.10.4.2 Composants de système

Un composant de système (comprenant le système lui-même) évolue selon un nombre de paliers de révision bien définis. Un nouveau palier de révision devrait assurer une compatibilité ascendante avec les paliers de révision précédents en respectant quelques critères, qui font partie des normes de l'organisation. La raison d'un nouveau palier de révision se trouve dans la correction de fautes et/ou l'introduction de nouvelles fonctionnalités. Si la compatibilité ascendante ne peut pas être assurée, alors un nouveau composant (variante) doit être créé au lieu d'un nouveau palier de révision. Un composant de système devrait avoir une identité et une désignation uniques pour le palier de révision.

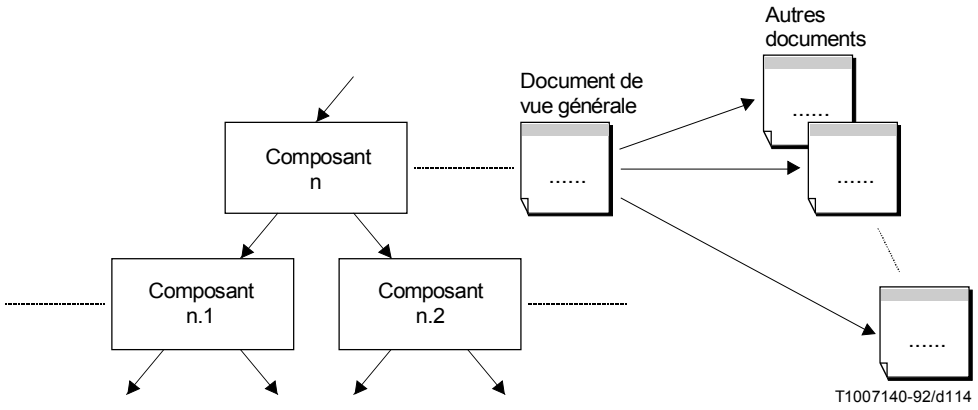


FIGURE I.10-4/Z.100

Une structure de documentation

Les caractéristiques générales d'un composant de système indiquées ci-dessous peuvent être renforcées de manière plus ou moins stricte selon la façon dont le composant est utilisé. A partir du point de vue déterminé par l'usage, nous pouvons distinguer:

- les composants locaux;
- les composants standards; et
- les composants faisant l'objet d'une livraison.

Un composant local devrait être une procédure utilisée à l'intérieur d'un système par une équipe de développement. Il est clair que la sûreté de manipulation (tâches d'identification, de documentation, de changement, etc.) d'un composant local ne constitue pas un problème majeur.

Un composant standard est généralement disponible pour tous les systèmes d'une organisation. A titre d'exemple on trouve des composants matériels et des types abstraits de données (sortes). Il est clair que les composants standards ne devraient pas être changés fréquemment ou ne devraient pas être changés du tout.

Un composant faisant l'objet d'une livraison est utilisé par les organisations clientes, dans différents paliers de révision, peut-être mondialement. Il est clair qu'un composant faisant l'objet d'une livraison doit être manipulé en suivant des règles strictes.

De quelle manière les entités SDL devraient être traitées à cet égard? Les *types* sont des candidats naturels pour être des composants standards. Les *blocs* ne devraient pas être utilisés en tant que composants faisant l'objet d'une livraison, si ceux-ci font partie d'une spécification SDL (de bas niveau). Il y a, cependant, des composants faisant l'objet d'une livraison qui ne peuvent pas être couverts de manière parfaite dans la spécification SDL. Toutes les autres entités SDL devraient être considérées comme des composants locaux.

Les identificateurs SDL ne couvrent pas le palier de révision et sont insuffisants pour les composants faisant l'objet d'une livraison. Pour les composants standards et les composants faisant l'objet d'une livraison, on fera appel aux techniques et aux outils existants pour la gestion de la configuration et de la révision.

I.10.4.3 Sortes de documentation

Les sortes de documents nécessaires pour un système dépendent de l'objet du système et des activités réalisées pour celui-ci. En général, les activités suivantes sont considérées comme pertinentes:

- commande;
- production;
- exploitation;
- développement;
- correction des défauts.

En règle générale, un élément d'information devrait être contenu dans un seul et unique document, de façon à éviter les incohérences engendrées par des changements. Dans certains cas, il n'est pas possible ou il n'est pas souhaitable, que l'un des documents soit considéré comme le document de base et les autres documents soient considérés comme des documents dérivés.

Les documents nécessaires au *Développement* constituent normalement la base pour les autres documents. Une spécification SDL couvre la structure et le comportement, et devrait constituer le noyau de cette classe de documents.

Les documents pour l'*Exploitation* contiennent la plupart du temps de l'information dérivée. Un problème spécifique relatif à ces documents est la façon dont ils peuvent être particularisés pour couvrir les fonctionnalités qui ont été livrées au client et uniquement celles-ci. Quelques constructions du SDL qui peuvent être utiles dans ce cas, ce sont l'*option*, le *synonyme externe* et le *sous-typage* (spécialisation).

I.10.4.4 Gabarit de document

Pour chaque type de document un gabarit devrait être fourni, de manière à renforcer l'uniformisation du style de la documentation et aussi pour faciliter la préparation de documents individuels. Un gabarit spécifie le type des informations qui doivent être considérées, la table des matières, la mise en page et éventuellement d'autres aspects de la documentation dont la normalisation est jugée nécessaire.

Différentes catégories de composants nécessitent des types différents d'ensembles de documents. Ici, nous considérons seulement la documentation pour un composant de système faisant l'objet d'une livraison. On utilise normalement les types de documents suivants; certains sont obligatoires, les autres sont utilisés lorsque cela paraît opportun.

- **Spécification de structure** – Fournit la liste des composants qui sont inclus et éventuellement aussi le plus ancien palier de révision jugé nécessaire.
- **Information de révision** – Est préparée pour chaque palier de révision et décrit les différences avec le palier de révision précédent.
- **Document de vue générale** – Fournit la liste de tous les autres documents qui appartiennent au composant, et indique aussi leur palier de révision approprié.
- **Information de commande** – Décrit les propriétés du composant pour le point de vue de la commande, couvrant aussi les fonctionnalités optionnelles et les codes (valeurs des synonymes externes) utilisés pour les sélectionner.
- **Description** – Fournit une structure complète et aussi éventuellement la description du comportement en SDL. La description du comportement ne doit concerner que les composants du niveau le plus bas du système (voir I.3). La spécification SDL peut être donnée en tant qu'annexe au document principal, qui fournit une brève description informelle et qui constitue alors une introduction à la spécification SDL.
- **Spécification de test** – Spécifie les tests que le composant doit satisfaire avant d'être livré.
- **Description de commande** – Décrit la syntaxe et la sémantique d'une commande, fournie pendant l'*Exploitation*, qui est exécutée principalement par le composant.
- **Description de sorties (listing)** – Décrit la syntaxe et la sémantique d'un message imprimé qui est engendré par le composant pendant l'*Exploitation*.
- **Manuel d'installation** – Décrit la procédure d'installation relative au composant.
- **Manuel d'exploitation** – Décrit comment le composant devrait être manipulé de manière correcte par un utilisateur /exploitant pendant l'*Exploitation*.

Bibliographie

[1] *Basic Reference Model* – International Standard, ISO/IS 7498, 1984.

[2] *OSI Service conventions* – Technical Report ISO/TR 8509, 1987.

[3] Recommendation I.130 du CCITT – *Méthodes de caractérisation des services de télécommunications assurés sur un RNIS et des possibilités réseau d'un RNIS.*

[4] Recommendation Q.65 du CCITT – *Etape 2 de la méthode de caractérisation des services de télécommunications assurés par un RNIS.*

[5] Recommendation Z.120 du CCITT – *Diagramme de séquencement des messages*, Genève, 1993.

[6] HARTMUNT (B): *Fundamentals of algebraic specification,* Volume 1, EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1985.

NACHUM: *Termination of rewriting*, Journal on Symbolic Computation, (3) pp 69-116, 1987.

HARALD: *A completion procedure for conditional equations*; Proc. 1st Int. Workshop on Conditional Term Rewriting, LNCS 308, p 62-83, Orsay, 1988, Springer-Verlag.

[7] CHONG-YI: *Process periods and system reconstruction,* Lecture Notes in Computer Science 222, Advances in Petri Nets, (G. Rozenberg ed.) Springer-Verlag, 1986.

[8] ENCONTRE, DELBOULBE, GAVAUD, LEBLANC, BOUSSALEM: *Combining services*, *Message Sequence Chart and SDL – Formalism, method, tools*, [10].

[9] FAERGEMAND, MARQUES EDITORS: *SDL '89 The language at work,* Proceedings of the 4th SDL Forum, North-Holland, Amsterdam, 429 p., 1989.

[10] FAERGEMAND, REED EDITORS: *SDL '91 Evolving methods*, Proceedings of the 5th SDL Forum, North-Holland, Amsterdam, 1991.

[11] GRABOWSKI (R): *Putting extended sequence charts to practice*, [9].

[12] HOGREFE: *OSI formal specification case study: the INRES protocol and service* – Technical Report, University of Berne, Avril 1991.

[13] KRISTOFFERSEN: *Message Sequence Charts and SDL specification consistency check,* [10].

[14] NAHM: *Consistency analysis of Message Sequence Charts and SDL systems*, [10].

[15] REISIG: *Petri Nets*, EATC Monographs on Theoretical Computer Sciences, Vol. 4, Springer Publ. Comp., 1985.

[16] GRAUBMANN, GRABOWSKI (R): *Towards an SDL design methodology using sequence chart segments*, [10].

[17] HOARE: *Communicating sequential processes*, Prentice-Hall International, 1985.

[18] PARNAS: *On the criteria to be used to decompose systems into modules*, Comm. ACM, vol. 15, 1972.

[19] PLOTKIN: *A structural approach to denotational semantics*, Report DAIM-FN-19, Computer Science Dept, Arhus University, Denmark, 1981.

[20] *A common semantics representation for SDL and TTCN* – European Telecommunications Standards Institute (ETSI) Technical Report, 1992.

[21] GODSKESSEN: *A compositional operational semantics for Basic SDL*, [10].

[22] WEST: *General technique for communications protocol validation,* IBM J. Res. Develop., 22(4), pp 393-404.

[23] AGHA: *ACTORS: A model of concurrent computation in distributed systems.* The MIT Press, Cambridge, Massachusetts, 1986.

[24] VALMARI: *A stubborn attack on state explosion*, Proc. Computer-Aided Verification, New Brunswick, New-Jersey, 1990.

[25] GRAF (S): *Compositional minimization of finite state processes*, in Proc. Computer-Aided Verification, 1990.

[26] HOLZMANN: *On limits and possibilities of automated protocol analysis*, Protocol Specification, Testing and Verification VII. Elsevier Science Publishers B V (North Holland), 1987.

[27] YOUNG, TAYLOR, FORESTER, BRODBECK: *Integrated cuncurrence analysis in a software development environment*, Proc. SCM SIGSOFT '89 Third Symp on Software Testing, Analysis, and Verification, Key West, Florida, 1989.

[28] EK, ELLSBERGER: *A dynamic analysis tool for SDL,* [10].

[29] *Information Technology – Open Systems Interconnection – Conformance testing methodology and framework*, International Standard IS 9646, 1991.

[30] HOGREFE: *Conformance testing based on formal methods, Proc. FORTE 90 3'rd Int Conf on Formal Description Techniques,* Madrid, 1990.

[31] BOLOGNESI, BRINKSMA: *Introduction to the ISO Specification Language LOTOS,* Computer Networks and ISDN Systems, 14, 1987, pp 25-59.

[32] BRAEK, HASNES, HAUGEN: *Engineering real-time systems with an object-oriented methodology based on SDL*, SISU Project Report 1992, Norwegian Computing Center, P.O. Box 114 Blindern, N-0314 Oslo, Norway.

[33] HAUGEN, MÖLLER-PEDERSEN: *Tutorial on object-oriented SDL*, SISU Project Report 91002, Norwegian Computing Center, P.O. Box 114 Blindern, N-0314 Oslo, Norway.

Remplacée par une version plus récente

Appendice II

Bibliographie pour le SDL

(Cet appendice fait partie intégrante de la présente Recommandation)

(Helsinki, 1993)

[1] BELINA (editor): *SDL Newsletter* Telia Research AB, P.O. Box 85, S-201 20 Malmö, Sweden.
/* Gratuit, Publié environ une fois par an sous les auspices du CCITT */.

[2] BELINA, HOGREFE, SARMA: *SDL with Applications from Protocol Specification*, Prentice-Hall International (UK), 270 p., (1991). Une version allemande sera publiée par Carl Hanser Verlag en 1992.
/* un livre sur SDL, qui peut aussi être utilisé comme référence. Peut aussi être utilisé comme introduction à la spécification de protocoles */.

[3] BELINA, HOGREFE: *The CCITT-Specification and Description Language SDL*, Computer Networks and ISDN System, 16, pp.311-341, (1988/89), North-Holland, Amsterdam.
/* un papier didactique */.

[4] BRAEK, HASNES, HAUGEN: *Engineering real-time systems with an object-oriented Methodology based on SDL* SISU Project Report 1992, 320 p, Norwegian Computing Center, P.O. Box 114 , N-0314 Oslo 3, Norvège.
/* un livre sur la méthodologie comprenant les techniques de mise en œuvre et de conception. Mis à jour pour SDL 92 */.

[5] Manuel du CCITT: Directives pour l'application de Estelle, LOTOS et SDL, UIT, 350 p. (1988); est aussi disponible comme document ISO DTR 10167 (Décembre 1989).
/*Contient une série d'exemples, dont chacun est spécifié en trois langues pour faciliter l'évaluation subjective et la comparaison. Présente aussi des raisons de l'utilisation des langages formels de spécification */.

[6] FÆRGEMAND, MARQUES (éditeurs): *SDL '89 The Language at work*, Proceedings of the Fourth SDL Forum, North-Holland, Amsterdam, 429 p., (1989).

[7] FÆRGEMAND, REED (éditeurs): *SDL '91 Evolving Methods*, Proceeding of the 5th SDL Forum, North-Holland, Amsterdam, 524 p., (1991).

[8] HAUGEN, MÖLLER-PEDERSEN: *Tutorial on object-oriented SDL*, SISU Project Report 91002, Norwegian Computing Center, P.O. Box 114, N-0314 Oslo 3, Norvège.

[9] HOGREFE, SARMA: *The application of SDL to ISDN and OSI*, Proceeding of the Seventh International Conference on Software, Engineering for Telecommunication Switching Systems (1989).

[10 HOGREFE: *OSI formal specification case study: the INRES protocol and service*, Technical Report, Université de Berne, Avril 1991.

[11] HOGREFE: *Protocol and Service Specification with SDL: the X.25 case study*, Bericht Nr. FBI-HH-B-134/88, Universität Hamburg, (1988).

[12] SARACCO, SMITH, REED: *Telecommunications system engineering using SDL*, North-Holland, Amsterdam, 631 p., (1989).
/* un livre sur SDL. Contient également des directives pour divers domaines d'application avec de nombreux exemples */.

[13] SARACCO, TILANUS (editors): *SDL '87 State of the art and future trends*, Proceedings of the Third SDL Forum, North-Holland, Amsterdam, 463 p., (1987).

[14] TURNER (editor): *Using Formal Description Techniques – An Introduction to Estelle, LOTOS and SDL*, sera publié par John Wiley & Sons en 1992.
/* est basé sur la référence 5 */.