Université de Montréal
Faculté des arts et des sciences

Ce mémoire intitulé:

Développement d'un éditeur graphique pour les suites de tests de protocoles de communication

présenté par:

Abdelouahed Soukeur

a été évalué par un jury composé des personnes suivantes:

Alexander Petrenko	Président
Gregor V. Bochmann	Directeur de recherche
Pierre Poulin	Membre du jury

Mémoire accepté le: 30 mars 1998

Université de Montréal

Développement d'un éditeur graphique pour les suites de tests de protocoles de communication

par

Abdelouahed Soukeur

Département d'informatique et de recherche opérationnelle

Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures

en vue de l'obtention du grade de

Maître ès sciences (M.Sc.)

en informatique

Octobre, 1997

Sommaire

L'ISO ("International Organization for Standardization") et le CCITT (Comité Consultatif International Télégraphique et Téléphonique) ont défini et normalisé le langage TTCN("Tree and Tabular Combined Notation") pour la spécification de suites de tests abstraits de protocoles de communication. Le format, textuel et tabulaire, offert par ce langage n'étant pas désiré par plusieurs de ses utilisateurs.

Nous avons développé l'outil GETS (Graphical Editor for Test Suites) dans le but d'avoir un environnement convivial pour l'édition et la maintenance des suites de tests. Pour réaliser cet environnement, nous nous sommes inspirés de la représentation graphique du langage SDL ("Specification and Description Language"). Nous avons ainsi utilisé quelques-uns de ses constructeurs et nous avons conçu d'autres constructeurs spécifiques aux tests pour le compléter.

L'outil GETS comprend également deux traducteurs générant respectivement du code TTCN-MP et du code SDL-PR aa partir de la représentation interne des suites de tests. Le code TTCN-MP généré peut être importé par un des outils de TTCN tel que WORKBENCH. Ceci permet de valider les suites de tests éditées par GETS. Le but du deuxième traducteur (SDL-PR) est de pouvoir utiliser des outils supportant SDL tel que SDT pour effectuer d'autres fonctions sur les suites de tests tel que la validation et la simulation.

L'étude comparative que nous avons effectué entre les langages SDL et TTCN, nous a permis de voir la possibilité et les limites d'utiliser le langage SDL pour spécifier des suites de tests.

Mots Clés: TTCN, Suite de tests, Protocoles de communication, GETS, SDL,

Table des matières

Liste des sigles et abréviations

ASP	Abstract Service Primitive (Primitive de Service Abstraite)
CCITT	Comité Consultatif International Télégraphique et Téléphonique
GETS	Graphical Editor for Test Suites
ISO	International Organization for Standardization
IST	Implantation Sous Test (ou IUT pour Implementation Under Test)
ITU	International Telecommunication Union
LOTOS	Language Of Temporal Ordering Specification
LT	Lower Tester
OMT	Object Modeling Techniques
OSI	Open System Interconnection
PCO	Point of Control and Observation (Point de contrôle et d'observation)
PDU	Protocol Data Unit
PICS	Protocol Implementation Conformance Statement
PTC	Protocol Test Center
SDL	Specification and Description Language
SDT	Specification and Description Tool
TCP	Test Coordination Procedures
TM	Test Management
TTCN	Tree and Tabular Combined Notation
UT	Upper Tester

Liste des tables

Liste des figures

Remerciements

Je tiens à remercier mon directeur de recherche, le professeur Gregor v. Bochmann pour m'avoir proposé ce sujet et pour m'avoir suivi tout au long de ce projet. Je voudrais aussi remercier le coordonnateur de projets Daniel Ouimet pour son aide qu'il a apporté à ce projet et pour ses suggestions, commentaires et critiques qui m'ont été très précieux.

Aussi, je remercie Jean-Walter Guillery, étudiant au baccalauréat à l'Université de Montréal pour sa contribution à l'implantation de ce projet.

Un grand merci à tous les membres du groupe de téléinformatique, en particulier Lucie Lévesque pour l'excellente ambiance familiale.

Enfin, je tiens à exprimer ma reconnaissance envers la Chaire industrielle HP-NSERC-CWARC de l'université de Montréal pour sa contribution financière.

Introduction

Avec l'expansion rapide des réseaux de télécommunication, les protocoles de communication sont devenus de plus en plus nombreux et complexes. Ils ont ainsi suscité une grande attention de la part de la communauté scientifique.

Comme les logiciels, les protocoles de communication passent par plusieurs étapes lors de leur cycle de développement. Parmi ces étapes nous citons la description formelle et le test de conformité.

Dans la phase de description formelle, le protocole est décrit dans un langage formel, beaucoup plus clair et précis que le langage naturel. L'ISO ("International Organization for Standardization") et le CCITT (Comité Consultatif International Télégraphique et Téléphonique) ont développé et normalisé plusieurs langages de description formelle tel que le langage SDL ("Specification and Description Language"). Ce dernier possède deux formats, un format textuel (SDL-PR) et un format graphique (SDL-GR) qui consiste en un ensemble de symboles graphiques (ou constructeurs). Ce format graphique a fait de SDL un langage très convivial et facile à utilisé.

Quant au test de conformité, il permet de vérifier que l'implantation d'un protocole est conforme à sa spécification formelle. L'ISO et le CCITT ont défini dans la norme ISO 9694 des architectures et des méthodes qui permettent de réaliser des tests de conformité. Ils ont également défini un langage de spécification de test de protocoles appelé TTCN ("Tree and Tabular Combined Notation"). Le langage TTCN permet de spécifier des suites de tests abstraits pour les protocoles des systèmes ouverts OSI. Pour représenter les différents éléments d'une suite de tests, TTCN utilise une forme tabulaire (TTCN-GR) et une forme textuelle (TTCN-MP). Plusieurs utilisateurs ne trouvent pas cette représentation conviviale et facile à manipuler. Afin de palier à ce problème, le centre de test de protocoles (PTC) Hewlett Packard, qui est un des utilisateur de TTCN, nous a proposé de

développer un éditeur qui permet la représentation graphique des suites de tests (GETS: "Graphical Editor for Test Suites").

Pour réaliser cet éditeur, nous avons utilisé un sous-ensemble de symboles graphiques du langage SDL et nous les avons complétés par d'autres que nous avons conçus en collaboration avec le PTC. L'éditeur a été complété par deux traducteurs, le premier permet de générer des suites de tests sous le format TTCN-MP et le deuxième sous le format SDL-PR.

Dans le chapitre 1, nous donnons un aperçu sur le cycle de développement des protocoles. Le chapitre 2 contient un survol du domaine des tests des protocoles de communication. Ainsi nous décrivons la classification des tests et les différentes architectures de test. Dans le chapitre 3, nous donnons une description du langage TTCN. Dans le chapitre 4, nous présentons le langage SDL. Dans le chapitre 5, nous dressons une étude comparative entre le langage TTCN et le langage SDL. Le chapitre 6 sera consacré à la conception de l'outil GETS. Le chapitre 7 concerne l'implantation de cet outil. Dans ce chapitre, nous décrivons le langage JAVA que nous avons utilisé pour réaliser ce projet, les classes d'objets que nous avons implantées ainsi que leurs méthodes et enfin nous expliquons la manière dont a été réalisée l'implantation des traducteurs et nous l'illustrons par quelques résultats que nous avons obtenus.

1. Le cycle de développement des protocoles de communication

Comme tout logiciel, les protocoles de communication passent par plusieurs étapes lors de leur cycle de développement. Ces étapes devront aboutir à la production d'un système qui possède des caractéristiques répondant aux besoins et aux exigences préalablement requis. Omar Rafik propose un cycle de développement comportant sept étapes décrites dans la figure 1-1 [Rafi 91]:

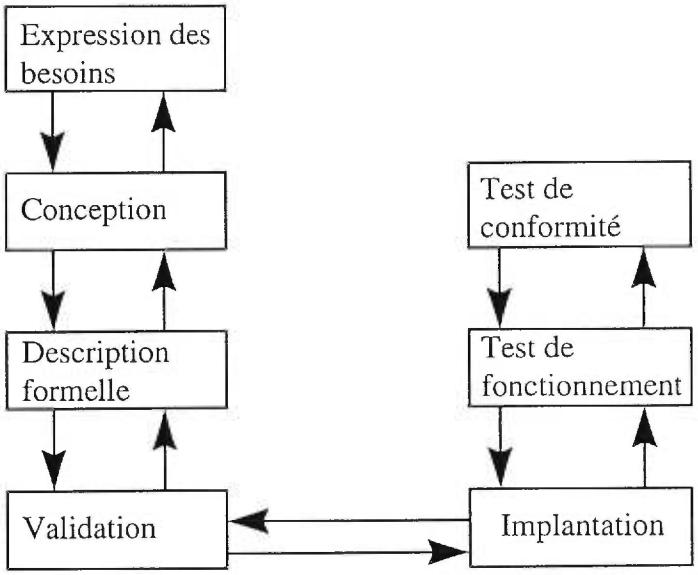


Figure 1-1: Cycle de développement des protocoles

1.1 Expression des besoins

L'expression des besoins consiste à établir un cahier de charges présentant les besoins de l'utilisateur en matière de communication et ceci en termes de services disponibles et de services désirés.

1.2 Conception

A partir de l'expression des besoins, on conçoit un protocole qui permet de fournir le service demandé en fonction du service disponible. En général, dans cette étape, le

protocole est décrit dans un langage naturel. Des schémas, des tableaux et des diagrammes sont souvent utilisés pour plus d'explication et de clarté. On parle alors d'une spécification informelle [Boch 89a] [Dsso 86] [Bell 89].

La spécification informelle est facilement compréhensible mais elle présente quelques inconvénients tels que :

- la présence d'ambiguïtés amenant à des interprétations différentes parfois contradictoires,
- elle ne peut pas être utilisée pour générer automatiquement des implantations et des tests de conformité,
- elle ne peut pas être validée automatiquement.

Malgré ces inconvénients, la spécification informelle demeure essentielle et servira de référence pour les étapes ultérieures du cycle de développement.

1.3 Description formelle

Afin d'éliminer les ambiguïtés de la spécification informelle, une spécification formelle est effectuée dans un formalisme approprié. A part les techniques formelles usuelles (automate à états finis, réseau de Petri, graphes,...), trois langages de description formelle sont maintenant normalisés par l'ISO et l'ITU ("International Telecommunication Union"): ESTELLE (prononciation française de STL "State Transition Language"), LOTOS("Language Of Temporal Ordering Specification") et SDL, et peuvent être utilisés pour écrire de telles spécifications formelles.

Cette étape est très importante puisqu'elle permet une automatisation, même partielle, de la suite du cycle de développement. Cependant, elle reste non obligatoire et peut être omise dans le cycle de développement.

1.3.1 Validation

La validation consiste à vérifier que la spécification formelle est correcte avant son utilisation pour une implantation. Dans cette phase, des comportements anormaux du protocole, dus à une spécification incomplète ou à des erreurs de conception, peuvent être

détectés en exécutant la spécification formelle. L'exécution de la spécification formelle est généralement effectuée en utilisant soit la technique d'analyse d'accessibilité [Boch 87, Rafi83, Zafi 80] pour une validation exhaustive, soit une technique de simulation [Jard 88] pour une validation non exhaustive.

1.4 Implantation

L'implantation consiste à générer, en un langage approprié, le code correspondant à la spécification validée.

1.5 Test de fonctionnement

Le test de fonctionnement est un test sur la structure interne de l'implantation. Il est appelé test boîte blanche. Dans cette phase, on essaie de couvrir tous les aspects de la structure interne (chemins, conditions, instructions,...). Ce type de test est réalisé par les programmeurs.

1.6 Test de conformité

Le test de conformité permet de vérifier si une implantation d'un protocole satisfait la spécification de référence. Dans ce type de test, seul le comportement externe est testé; aucune référence n'est faite à la structure interne de l'implantation du protocole . Il est appelé test boîte noire. Ce test consiste à effectuer des échanges d'ASP ("Abstract Data Primitive") et de PDU ("Protocol Data Unit") entre un testeur et l'IST (Implantation Sous Test). Ces échanges sont effectués à l'aide des interfaces d'interaction appelées PCO ("Point of Control and Observation"). Le testeur envoie donc des stimulus à l'IST et compare ses réponses à celles attendues, c'est-à-dire à celles de la spécification formelle. Si les réponses de l'IST sont conformes aux réponses attendues, alors le test a réussi sinon le test a échoué et l'implantation est remise en cause. L'ensemble des interactions définies pour vérifier une propriété donnée de l'IST est souvent appelé *séquence de test*, jeu de test

ou simplement test, et l'ensemble des tests conçus pour vérifier la conformité de l'IST à la spécification du protocole est appelé suite de test.

2. Tests de protocoles de communication

Pour que des systèmes de télécommunication hétérogènes puissent communiquer, il faudrait que les protocoles utilisés respectent les exigences des standards. La plupart des protocoles sont très complexes, ce qui entraîne des erreurs lors du processus d'implantation. La preuve formelle qu'une implantation de tels protocoles est conforme à sa spécification formelle est pratiquement impossible.

Les tests sont actuellement le meilleur moyen pour contrôler la conformité des protocoles. L'implantation est exécutée avec des suites de tests, puis ses réactions seront analysées.

2.1 Tests de conformité dans le cadre de l'OSI

Les tests de conformité sont réalisés à l'aide de suites de tests, en général déjà existantes, par des organismes et centres de tests nationaux et internationaux. Il existe actuellement des suites de tests pour certains protocoles tels que X.25, MHS (X400), LAP-D, LAP-B, ACSE, FTAM et pour les protocoles ATM (développés par le forum ATM). Ces suites de tests définissent non seulement les entrées à injecter à l'IST et les sorties possibles, mais aussi les verdicts associés: test réussi, test échoué ou test inconclusif.

L'ISO a aussi défini tout un cadre de travail: définition de plusieurs types de tests, structures de tests, architectures de tests et un langage de spécification de tests TTCN.

2.2 Types de tests

Les considérations économiques et pratiques ne permettent pas de réaliser des tests de manière exhaustive. Il est donc important de cibler les caractéristiques que l'on veut tester. L'ISO a défini quatre types de tests:

1. tests d'interconnexion de base,
2. test de capacité,
3. test de comportement et
4. tests de résolution de questions de conformité.

1) Tests d'interconnexion de base:

Ces tests permettent de détecter les situations évidentes de conformité et de s'assurer que les principales fonctions de base du protocole sont effectivement réalisées tel que l'établissement d'une connexion. Une fois que ces tests sont réalisés avec succès, des tests plus profonds et plus coûteux sont initiés.

2) Tests de capacité:

Des énoncés de conformité (PICS) sont associés aux standards de protocoles. Ces énoncés décrivent les capacités de l'implantation telle que la classe du protocole, les options, le nombre et le type des primitives de service et PDUs utilisées.

Les tests de capacité permettent de vérifier que les capacités ainsi décrites, sont effectivement supportées par l'IST.

3) Tests de comportements:

Ces tests permettent de tester les aspects dynamiques de l'IST. Il s'agit de vérifier que les règles de fonctionnement définies dans la norme sont respectées. Les tests de comportement ne peuvent pas être exhaustifs car le nombre de combinaisons d'événements et leur occurrence sont dans le temps soit très grand, soit infinie. Les tests de comportement peuvent servir de base pour l'évaluation de la conformité s'ils sont combinés avec les tests de capacité, mais ils ne sont pas appropriés pour résoudre les problèmes qui surviennent durant la période d'exploitation.

4) Tests pour la résolution de questions de conformité:

Ces tests tentent d'établir un diagnostic aussi final que possible sur certains aspects spécifiques de l'implantation. Ils sont idéalement utilisés pour donner une réponse (oui/non) dans des situations limitées et bien identifiées au préalable (exemple: pour vérifier qu'un aspect particulier est bien implanté, ou pour détecter la cause d'une anomalie particulière).

2.3 Structure des tests

Les différents types de tests définis par l'ISO [ISO9646] peuvent être considérés comme des groupes de tests constituant la suite de tests. Chaque groupe peut contenir d'autres groupes de tests et/ou plusieurs cas de tests. Les objectifs des cas de tests sont plus précis et mieux définis que ceux des groupes de tests, par exemple établissement de la connexion, transfert de données, multiplexage-éclatement, etc. Un cas de test est constitué d'un certain nombre d'événements de test (unité atomique d'un test telle que l'émission et la réception de PDU).

2.4 Architectures de test

L'implantation d'un protocole est conçue pour fonctionner dans un système distribué en communicant avec une implantation du même protocole situé dans un autre système grâce à un service de communication. Par conséquent, pour tester une telle implantation , il est nécessaire de disposer d'un environnement matériel et/ou logiciel appelé architecture de test qui doit lui permettre de fonctionner de manière adéquate[Rafi 91].

Le test consiste à observer le comportement de l'implantation comme si elle se trouvait dans son environnement réel [Rayn 87]. La simulation/observation est effectuée à l'aide d'un système de test appelé testeur, en appliquant une suite de test sous forme d'émission/réception de primitives de services au niveau des points de contrôle et d'observation (PCO). Un PCO est une interface utilisée par un testeur afin de communiquer avec une IST, d'observer et de contrôler ses réactions. Le testeur est l'entité qui permet de décider de la conformité de l'IST par rapport à la spécification. En effet, si un comportement donné n'est pas conforme à la spécification, le testeur signale une erreur.

Le système de test est principalement composé d'un testeur inférieur (LT) et éventuellement d'un testeur supérieur qui permettent de contrôler et d'observer l'IST respectivement au niveau de ses interfaces inférieure et supérieure. A partir de l'interface inférieure, les demandes de services effectuées par l'IST sont observées et contrôlées (exemple: requête). A partir de l'interface supérieure, on teste si l'IST fournit bien les

services qu'on lui demande (exemple: confirmation de l'accomplissement d'une activité demandée).

L'ISO a défini deux architectures de test de conformité[ISO9646]: l'une pour le test local et l'autre pour les tests externes.

2.4.1 Test local

L'environnement de test est situé sur le même site que l'IST. Cet environnement est constitué d'un testeur inférieur (LT), d'un testeur supérieur (UT) et d'une procédure de coordination du test (TCP). Cette méthode est la plus puissante en terme de détection des erreurs étant donné que l'environnement de test accède directement aux deux interfaces de l'IST. La figure 2-1 montre l'architecture du test local.

La disponibilité des PCOs dans un environnement réel fait que la méthode de test local n'est vraiment applicable que chez le fournisseur du protocole à tester [Ray 87].

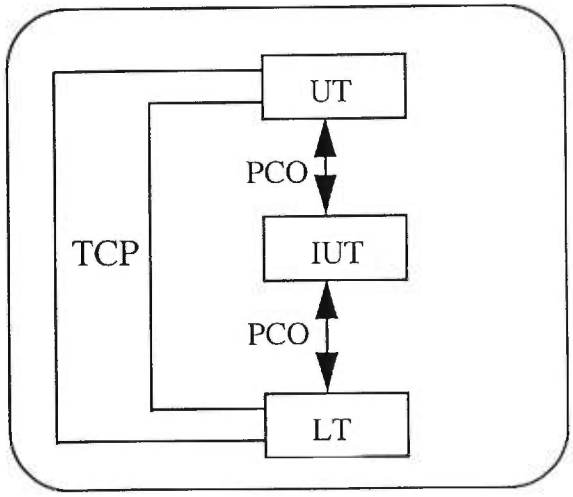


Figure 2-1: Le test local

2.4.2 Tests externes

Ces tests sont généralement utilisés par des centres de test indépendants du fournisseur du protocole. Il existe trois types de test externe:

Test distribué: Les testeurs inférieur et supérieur sont localisés dans deux sites différents. Le testeur inférieur réside dans le système de test, alors que le testeur supérieur réside dans le système sous test (figure 2-2). L'interface inférieure est contrôlée à distance via un service de communication, tandis que l'interface supérieure est directement observable. Les PCOs sont distribués entre le centre de test et le site sous test.

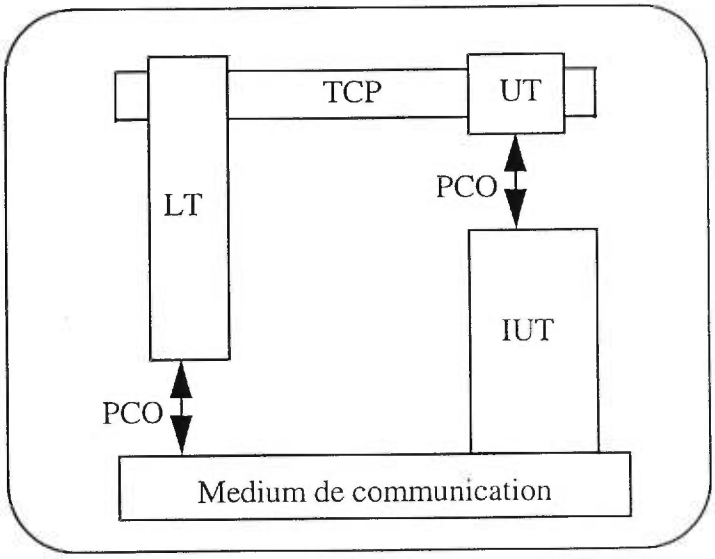


Figure 2-2: Test distribué

Test coordonné: Cette méthode de test est une amélioration de la méthode distribuée. Elle utilise un testeur supérieur standard et un protocole normalisé de gestion du test (TM-PDU). Il n'y a qu'un point de contrôle et d'observation, situé à l'opposé du service de communication (Figure 2-3).

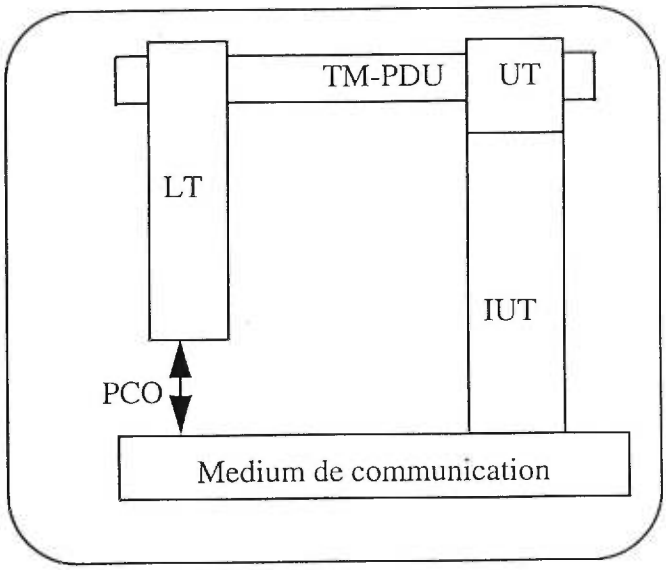


Figure 2-3: Test coordonné

Test à distance: Le système de test est composé d'un seul testeur: le testeur inférieur (Figure 2-4).

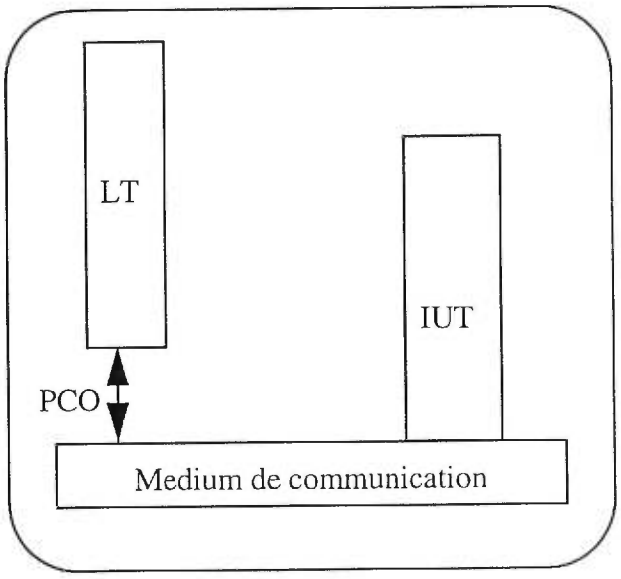


Figure 2-4: Test à distance

Cette architecture est appropriée pour les implantations dont le contrôle et l'observation des primitives de service prenant place à l'interface supérieure ne sont pas

possible. Les tests ne comprennent donc que les primitives du niveau inférieur contrôlées et observées à distance.

Remarque: Le principal avantage des architectures distribuées et à distance est la possibilité de tester avec un même testeur inférieur, situé dans un centre de test, un grand nombre d'implantations différentes situées sur des sites distincts. Ces deux types d'architecture sont les plus répandus. En effet, les tests de conformité sont souvent effectués par des centres de tests.

3. Le langage TTCN "Tree and Tabular Combined Notation"

L'organisation internationale de normalisation (ISO) a défini un langage de spécification pour décrire des suites de tests de protocoles de communication appelé TTCN [ISO 9646]. Le langage TTCN permet la spécification des suites de tests à un certain niveau d'abstraction. Dans ce chapitre, nous décrirons les différents aspects de ce langage.

3.1 Structure d'une suite de tests TTCN

Une suite de tests TTCN est constituée de plusieurs cas de tests qui permettent de tester différents aspects de la conformité d'une implémentation. Les cas de tests sont organisés en groupes de test, qui à leur tour peuvent être regroupés et ainsi de suite. Ce regroupement permet une classification et un ordre logique durant les tests. Certains tests sont construits sous forme de modules qui s'exécutent soit lors d'une invocation ou implicitement. Dans le premier cas, ces modules sont appelés étapes de tests et dans le deuxième cas, ils sont appelés défauts. Les groupes et les tests constituent le comportement dynamique d'une suite de tests TTCN.

Une suite de tests TTCN est composée des quatre parties suivantes:

1. La partie vue d'ensemble
2. La partie déclarations
3. La partie contraintes
4. La partie comportement dynamique

1) La partie vue d'ensemble: la suite de tests est présentée d'une façon générale. Toutes les références aux différents cas de test, étapes de test et défauts ainsi que celles des groupes qui les contiennent sont décrites à ce niveau. Cela permet de déterminer la structure hiérarchique de la suite de tests. C'est dans cette partie que sont décrits les objectifs des groupes de test.

2) La partie déclarations: tous les éléments utilisés dans la suite de tests sont définis à ce niveau, parmi ces éléments nous citons les ASPs , les chronomètres(Timers), les PCOs, les variables de la suite de tests et des cas de tests, les constantes, les types structurés et les types simples.

3) La partie contraintes: contient les contraintes sur les PDUs, les ASPs et les types structurés. Une contrainte spécifie les valeurs (ou des contraintes sur ces valeurs) des différents champs d'une PDU, d'une ASP ou d'un type structuré.

4) La partie comportement dynamique: contient le détail des comportements dynamiques des cas de test, étapes de test et défauts. Les échanges entre le testeur et l'IST sont définis à ce niveau. Les comportements dynamiques sont représentés d'une façon arborescente (Figure 3-1). Les événements qui sont représentés dans une même colonne sont des alternatives possibles selon le comportement de l'IST. Les événements consécutifs (séquentielles) sont représentés ligne par ligne de gauche à droite.

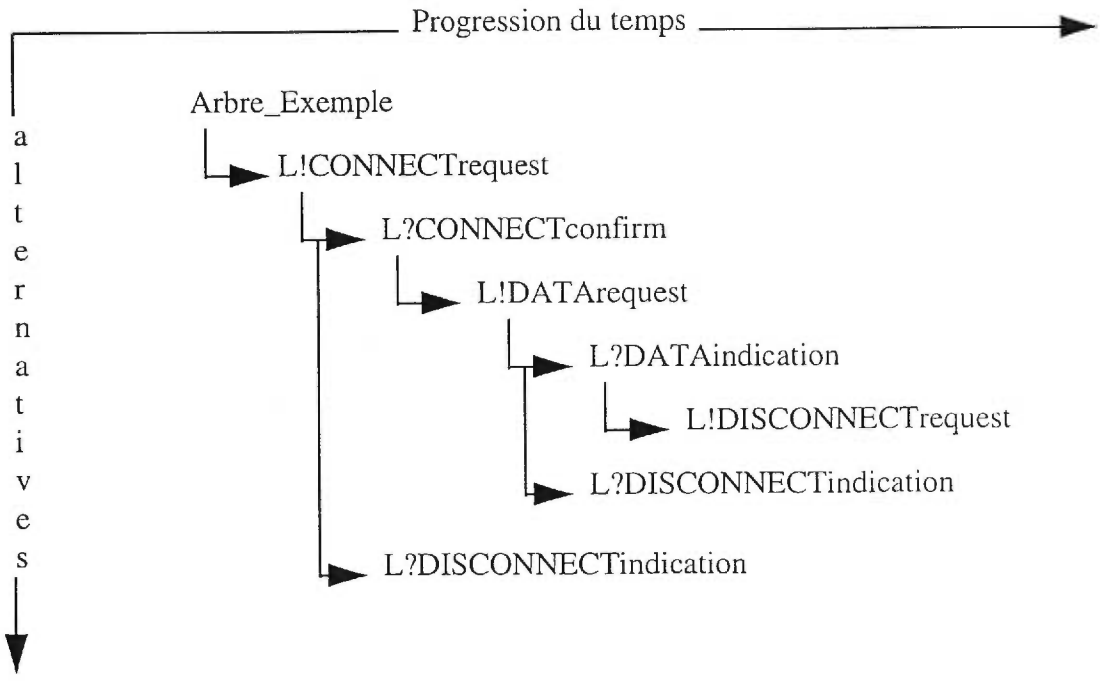


Figure 3-1: Conventions de la notation TTCN

Remarque:

Toutes les arborescences, sauf les cas de test, peuvent être paramétrées. Les paramètres peuvent être des PCOs, des contraintes, des variables de suite de tests, des variables de cas de test ou des constantes de suite de test.

3.1.1 Les cas de test

Un cas de test décrit les différents événements pouvant avoir lieu lors du test d'une IST, tous les scénarios possibles sont décrits à ce niveau. Un cas de test décrit également les verdicts associés à chaque comportement de l'IST. Les cas de test sont décrits sous forme d'un arbre ordonné, selon un séquencement temporel, dont les noeuds représentent les différentes instructions (Figure 3-2). Dans cette arborescence, tout chemin partant de la racine et aboutissant à une feuille représente une séquence de test complète et par conséquent, toute feuille doit être accompagnée d'un verdict final indiquant la façon avec laquelle s'est terminé l'exécution de test.

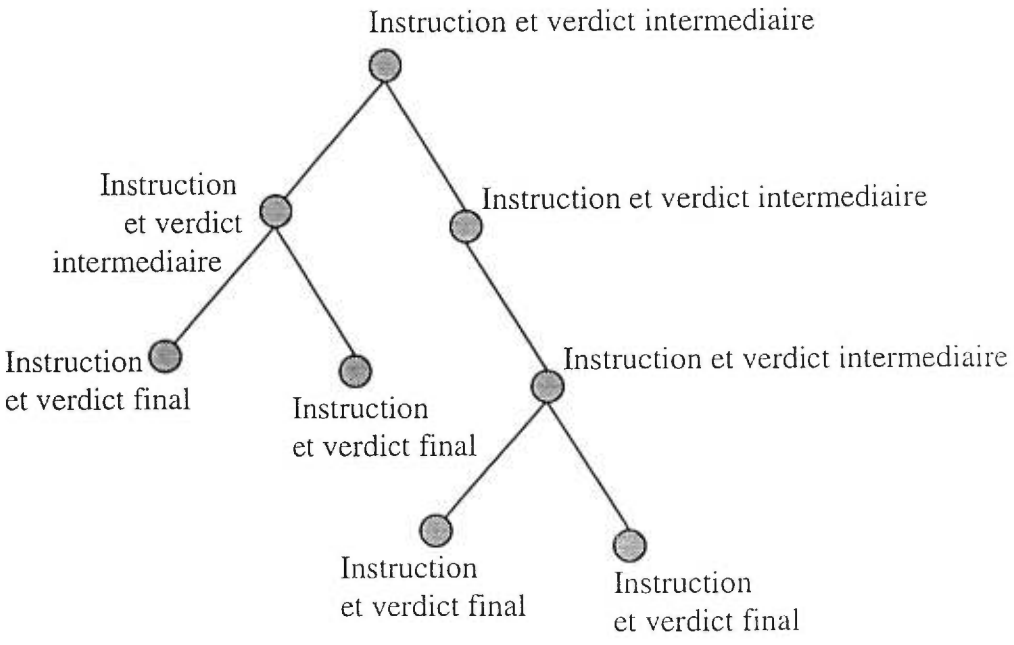


Figure 3-2: Structure d'un cas de test

3.1.2 Les étapes de test

Les étapes de test sont utilisées pour modulariser les cas de test ainsi que d'autres étapes de test. Elles jouent le même rôle que les procédures et les fonctions dans les langages de programmation. Elles ont la même structure que les cas de test (structure d'arbre), sauf que les étapes de test peuvent être paramétrées. Comme les cas de test, les étapes de test peuvent être organisées en groupes. L'appel d'une étape de test s'effectue d'une manière explicite à l'aide de l'instruction "Attach" qui permet l'attachement des arborescences des étapes de tests.

3.1.3 Les comportements par défaut

Les défauts sont utilisés par les cas et les étapes de test comme alternatives implicites. C'est-à-dire, lorsque dans un niveau donné de l'arborescence principale aucune alternative n'est satisfaite, alors le comportement par défaut s'exécutera. Les descriptions des comportements par défaut sont données dans la librairie des défauts.

Les défauts ont la même structure que les cas de test sauf que :

- Un défaut ne peut pas être spécifié dans un autre défaut.
- Un défaut ne peut pas être modularisé en utilisant des étapes de test.

Les défauts peuvent être organisés en groupe de défauts, et peuvent être paramétrés de la même façon que les étapes de test.

3.2 La notation d'arbre en TTCN

Chaque élément du comportement dynamique d'une suite de tests (cas de test, étapes de test et défauts) est décrit à l'aide d'une structure arborescente dont les noeuds représentent les instructions constituant cet élément. Les différentes instructions TTCN sont reliées entre elles soit d'une façon séquentielle soit d'une façon alternative.

Les instructions séquentielles sont exécutées l'une à la suite de l'autre selon un ordre temporel. Elles ont des niveaux d'indentation différents selon leur ordre d'exécution (Voir Figure 3-3).

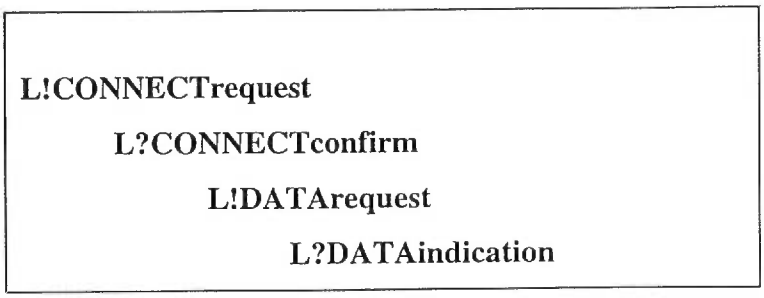


Figure 3-3: Instructions TTCN en séquence

Les alternatives ayant le même niveau d'indentation succèdent à un même prédécesseur. Elles représentent les alternatives possibles qui peuvent avoir lieu, de façon exclusive, au même moment (Figure 3-4).

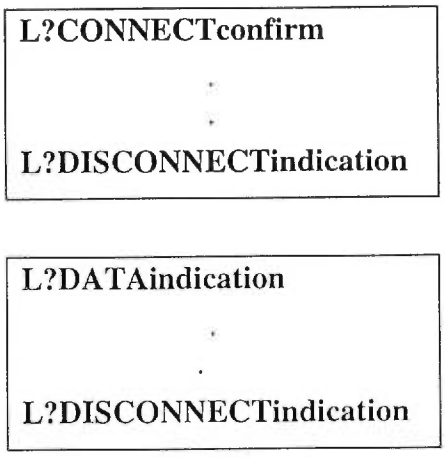


Figure 3-4: Alternatives TTCN

3.3 Les instructions TTCN

Le langage TTCN permet de spécifier les événements de tests en sortie et en entrée. Les événements en sortie permettent d'exprimer les envois de messages à partir des testeurs vers l'IST (Send et Implicit Send). Les événements en entrée permettent d'exprimer ce que les testeurs pourront recevoir (Receive, Otherwise et Timeout). TTCN permet également la spécification des structures de contrôle (Goto, Attach et Repeat) et des "pseudo-events"

comprenant des combinaisons d'expressions booléennes, des assignations et des opérations sur les "Timers". Ce sont les instructions TTCN du comportement dynamique.

3.3.1 Format des instructions TTCN

Une instruction TTCN du comportement dynamique est constituée de six champs (Voir Figure 3-5):

Numéro de ligne	Etiquette	Description du comportement	Contrainte	Verdict	Commentaire
1		L!CONNECTreq (START T1)	CR1		Request
2		L?CONNECTcon	CC1		Confirm
3		L!DATAreq	DTR1		Send data
4		L!DATAind	DTI1		Receive data
5		L!DISCONNECTreq	DSCR1	P	Accept
6		L?OTHERWISE			Not waited
7		L!DISCONNECTreq	DSCR1	F	Reject
8		L!DISCONNECTind	DSCI1		Premature
9		L!DISCONNECTreq	DSCR1	I	
10		L?TIMEOUT(T1)			
11		L!DISCONNECTreq	DSCR1	P	
12		L?OTHERWISE			
13		L!DISCONNECTreq	DSCR1	F	

Figure 3-5: Exemple de cas de test en format TTCN.GR

1. Numéro de ligne: Indique le numéro d'une instruction. Très utile lorsque une instruction s'écrit sur plusieurs lignes.

2. Étiquette: Permet particulièrement, l'utilisation de l'instruction de branchement "GOTO".

3. Description du comportement: Décrit les échanges entre l'IST et les testeurs. Un événement d'émission vers l'IST est identifié par "!", et un événement de réception à partir de l'IST est identifié par "?".

4. Référence des contraintes: Fait référence à une contrainte définie dans la partie des contraintes. Cela permet d'obtenir les valeurs des paramètres et champs des ASPs et PDUs à envoyer ou à recevoir selon le cas d'émission ou de réception.

5. Verdict: Spécifie la manière avec laquelle s'est achevé le test lorsqu'il passe par l'instruction contenant le verdict (voir section 3.5).

6. Commentaire: Explique et clarifie les interactions.

3.3.2 Règles d'exécution d'un cas de test

L'exécution d'un test commence à partir de la racine de l'arborescence. Dans le cas où l'instruction spécifie un événement à émettre, le testeur (supérieur ou inférieur) spécifié dans l'instruction, devra effectuer l'émission. Dans le cas où un ou plusieurs événements sont attendus en réception, le testeur se met en attente au niveau de l'indentation courante jusqu'à ce qu'il y ait correspondance entre l'une des alternatives et l'événement qui arrive de l'implantation.

Une fois qu'une instruction est exécutée, le testeur se positionne sur le niveau d'indentation suivant en séquence. Il ne peut y avoir retour au niveau précédent que dans le cas de l'instruction "GOTO".

3.3.3 L'instruction "Implicit Send"

Dans les méthodes de test à distance (voir section 2.4.2), il est nécessaire de disposer d'un moyen pour spécifier, lors du déroulement du test, que l'IST devrait émettre une PDU ou ASP particulière et cela malgré la non-existence de PCO au niveau supérieur de l'implantation sous test.

L'instruction "Implicit Send" permet de spécifier une telle demande, i.e. de spécifier la réaction requise par l'IST, mais en revanche ne spécifie pas comment faire pour provoquer l'IST à fournir ce qui est indiqué. En effet une instruction "Implicit Send" est toujours suivie d'une instruction de réception de l'interaction demandée, ceci pour vérifier l'émission effective de la part de l'IST de ce qui lui a été demandé.

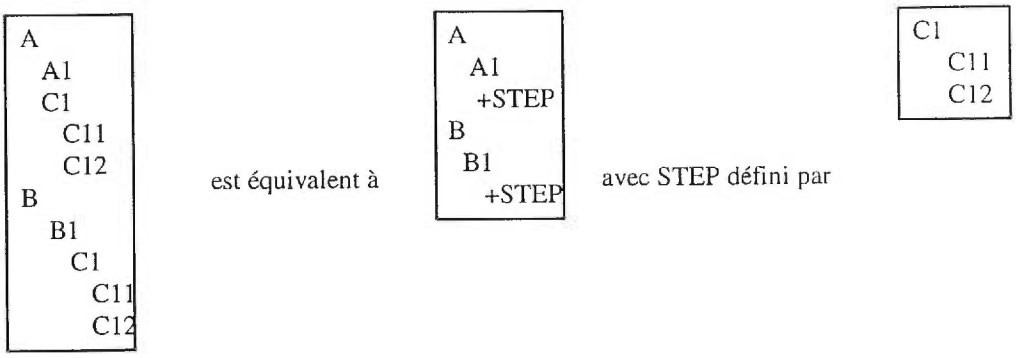
Exemple : Utilisation de l'Implicit Send(<IUT ! CR>)

```
<IUT ! CR>
        L? CR
```

3.3.4 L'instruction "Attach"

Cette instruction permet d'attacher des arborescences à une position donnée dans la spécification d'une suite de tests. Les seules arborescences qui peuvent être attachées à l'aide de cette instruction sont les étapes de test. Elles sont attachées à des cas de test ou à d'autres étapes de tests. Il est à noter que lors du déroulement de test, seuls les cas de test sont exécutables. Les étapes de test ne sont exécutées que lorsqu'elles sont attachées à un cas de test ou à une étape de test dont le point d'attachement remonte jusqu'à un cas de test, soit d'une façon directe ou indirecte via d'autres étapes de test.

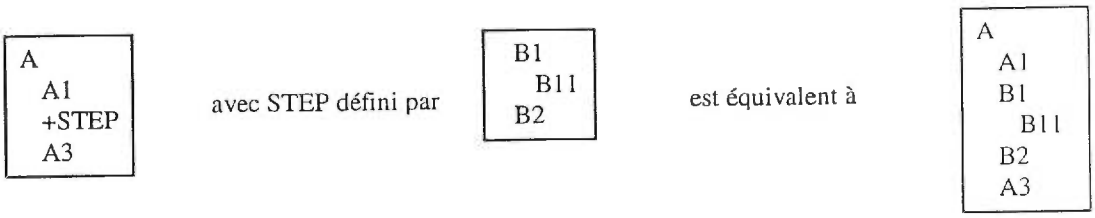
Exemple 1:



L'instruction "Attach" peut être utilisée comme alternative dans un ensemble ordonné d'alternatives au même titre qu'une réception d'événements. Les alternatives du premier niveau du sous-arbre à attacher sont insérées dans l'arbre appelant à la même position où l'appel a été fait.

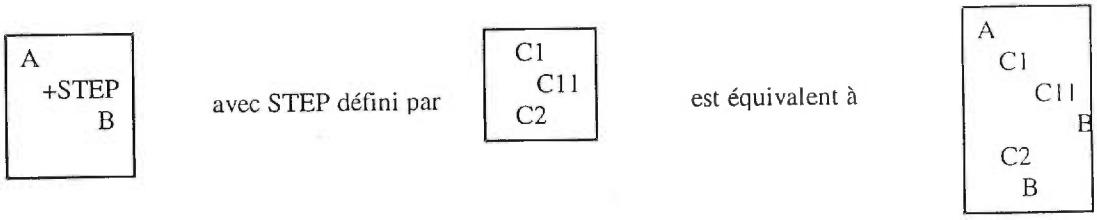
Remarque: Il est nécessaire que la première instruction du sous-arbre à attacher soit une réception d'événements.

Exemple 2:



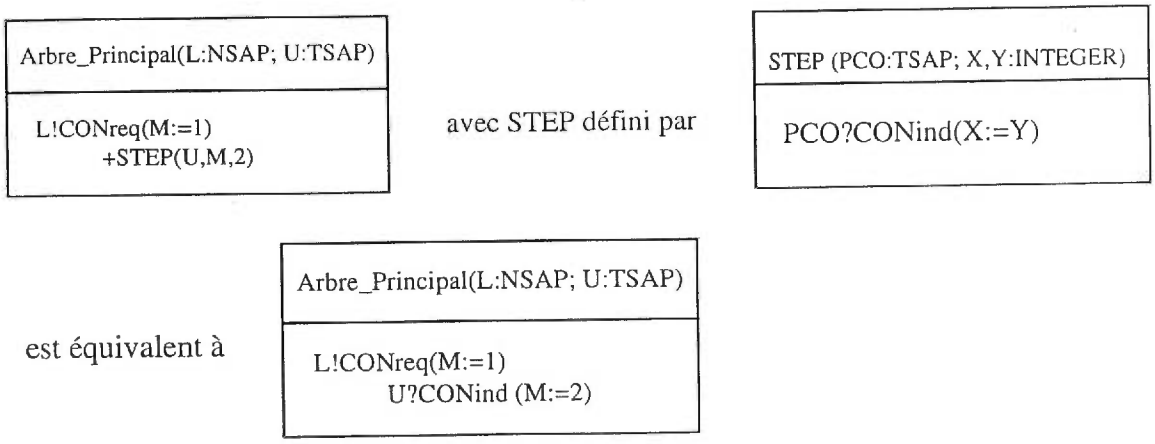
L'instruction "Attach" peut également être utilisée en séquence avec d'autres instructions. Dans ce cas, tout comportement en séquence à "Attach" devient un comportement en séquence de toutes les feuilles du sous-arbre attaché.

Exemple 3:



Lorsqu'une liste de paramètres est utilisée dans une instruction "Attach", chaque paramètre d'appel devra être substitué dans toutes les instructions du sous-arbre attaché dans lesquelles le paramètre correspondant est utilisé.

Exemple 4:



3.4 Les défauts

L'attachement implicite des défauts a le même impact que si l'on considérait le défaut comme une étape de test que l'on branche à la fin de chaque ensemble d'alternatives de tous les niveaux. Il ne peut y avoir un comportement par défaut à un autre défaut. Les attachements d'arbres ne peuvent être utilisés par les défauts, i.e. les comportements par défaut ne devraient pas attacher des étapes de test. Les cas de test et les étapes de test ne peuvent être utilisés comme défauts.

Afin d'assurer qu'aucun comportement ne peut avoir lieu à la suite de l'exécution des défauts, un verdict final devrait être assigné à chaque feuille du comportement par défaut.

3.5 Les verdicts

Tout au long du processus de test, il peut y avoir émission de verdicts en fonction du comportement de l'IST. A part les instructions "Attach", "Repeat", "Goto" et "Implicit Send", toutes les autres instructions TTCN peuvent contenir un verdict. Il existe deux types de verdicts:

- Verdict préliminaire,
- Verdict final.

Un verdict préliminaire peut être "Pass", "Inconclusive" ou "Fail"; il est représenté entre parenthèses. Un verdict (Pass) informe que certains aspects de l'objectif du test ont été vérifiés, (Inconclusive) signifie qu'un événement rendant le test inconclusif s'est produit, et (Fail) signifie que l'objectif de test a échoué suite à une erreur produite par l'IST.

Une variable prédéfinie "R" est utilisée pour stocker le verdict préliminaire courant, i.e. calculé jusqu'à présent. Au départ, cette variable est initialisée à "Null". Après chaque exécution de l'instruction correspondant au verdict préliminaire, la valeur de la variable "R" est recalculée en se basant sur la table suivante:

Verdict préliminaire de l'instruction / Ancienne valeur de R	(Pass)	(Inconclusive)	(Fail)
Null	Pass	Inconclusive	Fail
Pass	Pass	Pass	Fail
Inconclusive	Inconclusive	Inconclusive	Fail
Fail	Fail	Fail	Fail

Table 3-1: Calcul de la variable "R"

Un verdict final peut être "Pass", "Inconclusive", "Fail" ou "Error". Il est calculé en fonction de la valeur de la variable prédéfinie "R" et du verdict donné explicitement au niveau d'une feuille atteinte dans l'arborescence. Ce calcul est fait selon le tableau suivant:

Verdict rencontré / Valeur de R	Pass	Inconclusive	Fail
Null	Pass	Inconclusive	*Error*
Pass	Pass	Inconclusive	Pass
Inconclusive	*Error*	Inconclusive	Inconclusive
Fail	*Error*	*Error*	Fail

Table 3-2: Calcul du verdict final

4. Le langage SDL ("Specification and Description Language")

4.1 Historique

SDL [ITU Z.100] est un langage de spécification formelle des systèmes. Il a été développé et normalisé par l'ITU. Il offre un moyen de description formelle du comportement des systèmes et en particulier des systèmes de télécommunication, permettant ainsi leur analyse et leur interprétation d'une manière non ambiguë.

Le développement de SDL a commencé en 1972. La première version de ce langage a été publiée en 1976, suivie par de nouvelles versions en 1980, 1984, 1988 et 1992. La dernière version, SDL92, est étendue pour couvrir des concepts orientés objet.

4.2 Domaine d'application

SDL a été développé dans le but d'être utilisé dans les systèmes de télécommunication, mais actuellement il peut être utilisé dans tout système temps réel et système interactif [Bell 89] . De ce fait, il est le plus couramment utilisé dans l'industrie informatique.

4.3 Principaux concepts de SDL

Une spécification d'un système en SDL est vue comme une séquence de réponses suite à une séquence de stimulus. Le modèle de spécification en SDL est basé sur les concepts d'automates communiquants à états étendus.

SDL couvre différents niveaux d'abstraction en partant d'une vue globale jusqu'à un niveau de description détaillée. Une spécification d'un système en SDL est un ensemble de blocs interconnectés. Le comportement de chaque bloc est modélisé par un ou plusieurs processus. La communication entre les processus se fait de façon asynchrone par l'intermédiaire de files FIFO non bornées.

Dans SDL, on considère deux formats syntaxiques: un format graphique (SDL/GR) et un format textuelle (SDL/PR). Ces deux formats sont équivalents et dérivent de la même grammaire abstraite.

4.3.1 Structuration d'un système SDL

Un système est décrit en SDL en suivant une approche " Top-Down " dont le but est de surmonter sa complexité. À un plus haut niveau d'abstraction, un système est vu comme un ensemble de blocs. Un bloc peut contenir plusieurs sous-blocs. Les blocs sont interconnectés par des canaux (Figure 4-1). Ces canaux constituent le moyen de communication entre les blocs, et entre le système et son environnement. Un canal représente une communication unidirectionnelle. Une communication bidirectionnelle est représentée par deux canaux unidirectionnels indépendants. Un canal est typé, i.e. il ne peut contenir que des messages d'un certain type.

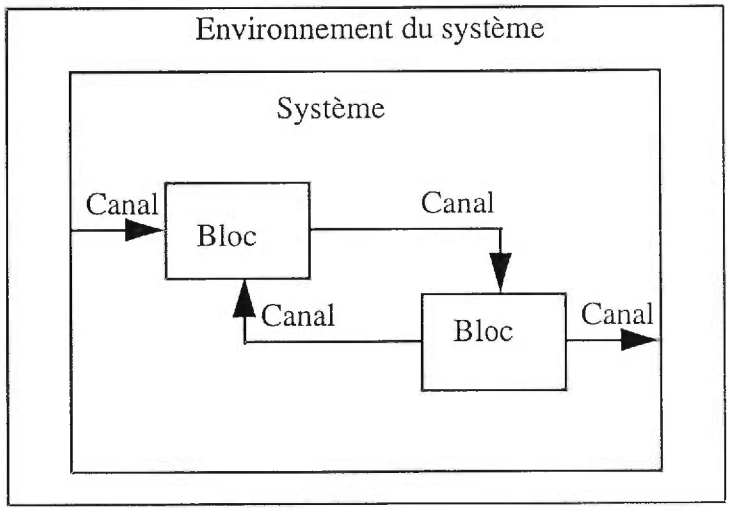


Figure 4-1: Structure d'un système SDL

Un bloc est constitué de plusieurs processus interconnectés par les chemins de signaux ('Signal-routes'). Ces chemins constituent un moyen de communication entre les

processus (Figure 4-2). De plus, ils peuvent être reliés aux canaux inter-blocs pour mettre en relation des processus appartenant à des blocs différents .

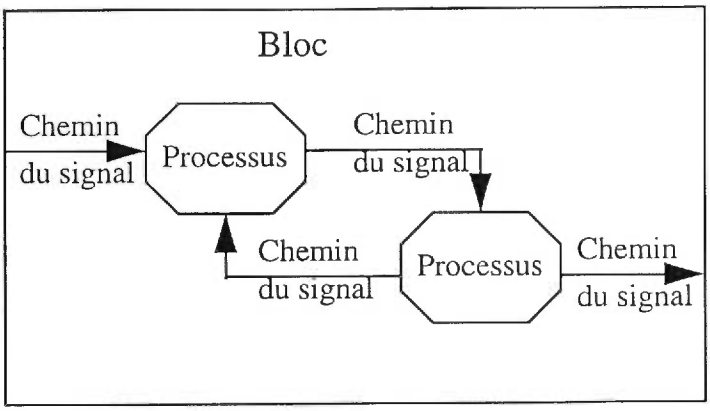


Figure 4-2: Structure d'un bloc SDL

4.3.2 Comportement d'un système SDL

Dans SDL, le comportement d'un système est la combinaison des comportements des processus dans le système. Un processus est une machine à états étendus, qui s'exécute d'une manière autonome et en concurrence avec les autres processus dans le système. La communication entre les processus se fait d'une façon asynchrone par des messages discrets appelés signaux. À chaque processus est associée une file d'entrée, dans laquelle tous les signaux qui arrivent via différents chemins de signaux sont stockés dans leur ordre d'arrivée. Ensuite, quatre cas de figure sont considérés:

• Le signal en tête de file correspond au signal spécifié au début d'une transition. Ce signal est retiré de la file, et le processus exécute la transition associée. Le signal est dit consommé par le processus.

• Le signal en tête de file est spécifié dans un "save construct", dans ce cas le signal reste dans la file et on examine le signal qui le suit.

• Le signal à la tête de la file n'est dans aucun des deux cas précédents, dans ce cas le signal est retiré de la file (ceci est appelé transition implicite) et on examine le signal suivant à moins qu'une transition spontanée soit présente; celle-ci est spécifiée par le constructeur *NONE*.

• La file est vide, dans ce cas le processus est suspendu jusqu'à l'arrivée d'un signal.

Une transition permet de passer d'un état à un autre en exécutant une suite d'actions. Les actions peuvent être des sorties, des 'Tasks' ou des décisions. Une action de sortie transmet un signal à un autre processus. Un 'Task' consiste en une manipulation des données internes et/ou des opérations sur les Timers. Enfin, une décision est un choix d'alternatives d'actions.

Pour la description d'un processus, SDL fournit cinq constructeurs de base :

Start, State, Input, Output et Nextstate. La figure 4-3 montre la représentation graphique (SDL/GR) de ces constructeurs.

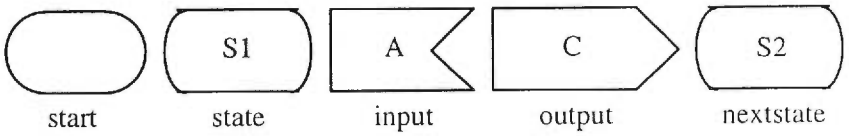


Figure 4-3: Les constructeurs de base pour la description d'un processus.

SDL fournit d'autres constructeurs parmi lesquelles nous citons l'appel de procédures et l'appel de macros. La figure 4-4 montre la représentation graphique (SDL/GR) de ces constructeurs.

Figure 4-4: Autres constructeurs de SDL.

4.3.3 Le temps dans SDL

Dans une spécification d'un système, on est souvent amené à définir des contraintes temporelles. Pour cela, SDL possède le type *timer*. Un *timer* est un objet appartenant à un processus. Il peut être actif ou inactif. La primitive *set* permet d'activer un *timer*. Celle-ci possède deux paramètres; le premier est le temps absolu avant l'expiration du timer, et le deuxième est le nom du timer. Pour la spécification du temps d'expiration du timer, l'expression NOW peut être utilisée et elle donne le temps courant du système. Un *timer* activé peut être désactivé par la primitive *reset* ayant comme paramètre le nom du timer en question. L'expiration d'un timer génère un signal qui sera déposé dans la file du processus.

4.3.4 Description et utilisation de données

En SDL, les variables sont déclarées à l'aide du mot clé *DCL*. Elles sont manipulées localement à l'intérieur du processus. Leurs valeurs peuvent être utilisées pour contrôler les transitions à l'aide des décisions.

La figure 4-5 montre un exemple de la déclaration et la manipulation des variables ainsi que la décision en SDL/GR. Dans cet exemple, le processus va transiter de S1 à S2, si la valeur de C est inférieure à 3, ou à S3 (sinon).

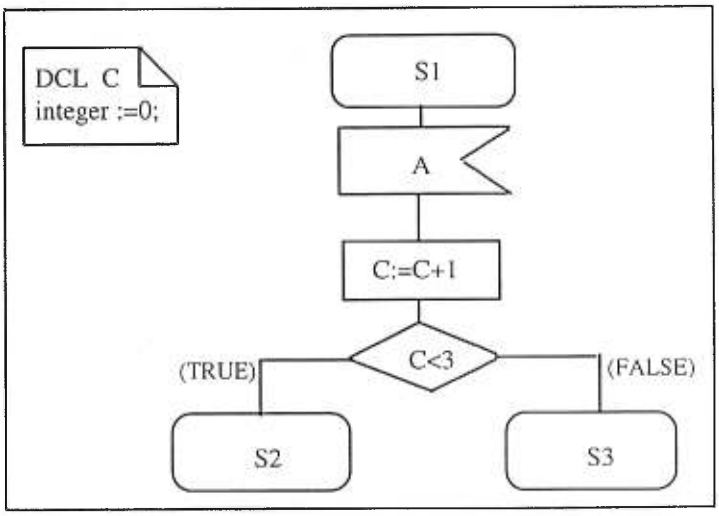


Figure 4-5: Les variables et la décision en SDL.

4.4 Outils supportant SDL

Le langage SDL est supporté par un ensemble d'outils, tels que SDT [SDT 96] et GEODE [Geod 95], permettant de faire certains traitements sur la spécification. En effet, à partir d'une spécification écrite en SDL, ces outils permettent de simuler et de valider la spécification ainsi que de générer une partie de l'implantation qui en découle.

5. Comparaison entre SDL et TTCN

Dans les chapitres précédents nous avons vu que le langage SDL permet la spécification formelle des systèmes et que le langage TTCN permet la spécification de suites de tests. Étant donné que SDL offre une représentation graphique et qu'il existe des outils qui permettent la validation de systèmes décrits en SDL, il serait intéressant d'étudier la possibilité d'utiliser le langage SDL pour spécifier des suites de tests. Pour cela nous allons effectuer une comparaison entre les langages SDL et TTCN. Dans ce qui suit, nous considérerons donc surtout les concepts spécifiques aux tests c'est-à-dire ceux qui peuvent être représentés par TTCN et qui ne le sont pas directement par SDL.

Les langages SDL et TTCN présentent un certain nombre de concepts communs et certaines différences. Dans certains cas et malgré ces différences, le passage de TTCN à SDL est possible en remplaçant une instruction TTCN par des instructions SDL.

5.1 Concordance entre SDL et TTCN

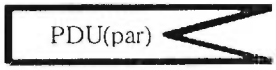
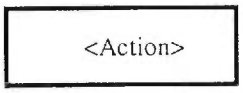
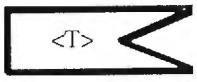
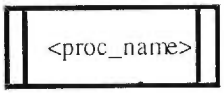
D'un point de vu fonctionnel, les langages SDL et TTCN permettent tous les deux de spécifier :

- des événements d'envois et de réceptions,
- des tâches permettant d'exprimer des affectations à des variables et des opérations sur des Timers,
- et des appels de procédures.

Cependant, d'un point de vue sémantique, ces deux langages présentent quelques différences illustrées par le tableau suivant:

Fonction	Sémantique dans TTCN	Sémantique dans SDL
Événements d'envoi	Envoi est avec contraintes: Procéder à l'application de ces contraintes avant l'envoi effectif.	Pas de notion de contraintes. Les signaux envoyés peuvent être paramétrés.
Événements de réception	Les PDU et ASP sont reçues dans des files FIFO correspondant aux différents PCO. Lorsque la PDU ou ASP reçue dans la file correspond à celle attendue et lorsque son contenu correspond aux contraintes posées, l'exécution continue sur le chemin qui contient l'événement de réception.	Le signal reçu n'est consommé que s'il correspond à celui attendu. Après la consommation du signal, le paramètre contenu dans le signal sera examiné si nécessaire.
Opération sur Timers	A l'expiration d'un Timer, son nom est mis dans la liste des Timeout.	L'expiration d'un Timer génère un signal qui sera déposé dans la file du processus comme tout autre signal reçu.
Passage de paramètres lors de l'appel de procédures et de macros	Un paramètre de type variable de suite de tests est transmis par référence, vu que la variable est globale. Les autres paramètres (variable de cas de test, constante de suite de test, valeur de littéral, contrainte ou PCO), sont transmis par valeur.	Le mode de transmission d'un paramètre est précisé dans la procédure.

Et d'un point de vue syntaxique, le tableau suivant montre les différents formats utilisés par les langages SDL et TTCN pour cet ensemble de concepts communs :

Fonction	TTCN	SDL.PR	SDL.GR
Réception	**?<PDU ou ASP>**	**Input <PDU(paramètre) ou ASP(paramètre)>**	
Envoie	**! <PDU ou ASP>**	**Output<PDU(paramètre) ou ASP(paramètre)>**	
Action	**<Action>**	**Task <Action>**	
Démarrage d'un Timer	**Start <T>**	**Set(<Duration>, <T>)**	Les opérations de démarrage et d'annulation des Timers s'effectuent à l'intérieur du symbole d'action défini ci-dessus.
Annulation d'un Timer	**Cancel <T>**	**Reset (<T>)**	
Lecture d'un Timer	**?TIMEOUT <T>**	**Input <T>**	
Appel de procédure	+	**Call**	

Fonction	TTCN	SDL.PR	SDL.GR
Étiquette	<label>	<label> :	
Branchement à une étiquette	GO TO	Join	
Décision	[<Condition>] ..... [NOT<Condition>]	Decision <expression booléenne>; (True): .....; (False): .....; EndDecision;	
Alternative par défaut	**?OTHERWISE**	**Input ***	

Note : T désigne l'identificateur du Timer

Exemple :

Dans le cas de test suivant, nous retrouvons toutes les fonctions qui existent dans les deux langages.

- **Représentation à l'aide de TTCN**

	+ Preambule_1
	! Envoi_1
	[Condition_1]
	START Timer_1
Label_1	? Reception_1
	CANCEL Timer_1
	? TIMEOUT Timer_1
	Action_1
	? OTHERWISE
	GOTO Label_1
	[Not (Condition_1)]
	START Timer_1
Label_2	? Reception_2
	CANCEL Timer_1
	? OTHERWISE
	GOTO Label_2

- **Représentation à l'aide de SDL:**

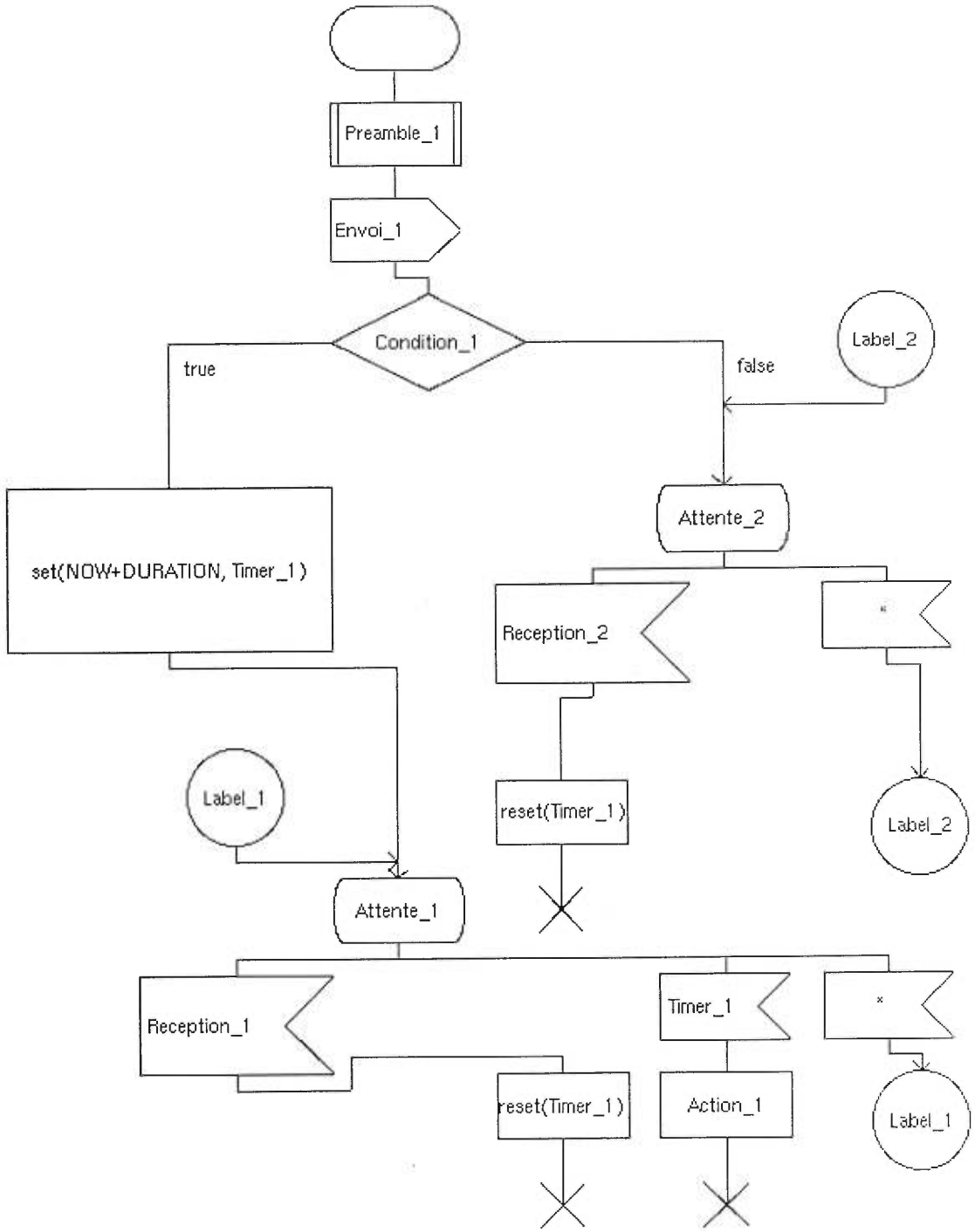
La représentation de ce cas de test en SDL nécessite le rajout d'un état initial et d'un état d'attente d'événements. Dans ce qui suit nous montrons respectivement les représentations SDL.PR et SDL.GR de ce cas de test :

```
start;

  call Preambule_1;
  output Envoie1;
  decision  Condition_1;

        (true) :
              task  Set(NOW+Duration, Timer_1);
Label_1:            state Attente_1;
                          input Reception_1;
                                reset Timer_1;
                          input Timer_1;
                                task Action_1;
                          input *;
                                join Label_1;
                    endstate Attente_1;

        (false) :
Label_2:            state Attente_2;
                          input Reception_2;
                                reset Timer_1;
                          input *;
                                join Label_2;
                    endstate Attente_2;
  enddecision;
```



5.2 Non concordance entre SDL et TTCN

La différence principale entre SDL et TTCN réside dans le fait que le premier est orienté vers la description de systèmes, alors que le deuxième est un langage pour la

description de suites de tests. De ce fait, les fonctions suivantes existent dans TTCN et n'existent pas directement dans SDL: émission de Verdicts, envoi implicite (Implicit send), envoi avec contrainte et réception avec contrainte. Ceci pose un problème lors du passage de TTCN à SDL. Pour palier à ce problème, nous proposons des solutions suivantes:

5.2.1 Solution pour l'émission de Verdicts :

Dans ce cas, on suppose que l'environnement de test contient un canal de gestion. Ce dernier permet à l'ingénieur de test de recevoir de l'information sur le déroulement du test telles que début/fin d'un cas de test, verdicts et messages d'erreurs.

La solution proposée consiste en la réalisation d'une macro-commande en SDL qui permet de calculer les verdicts, en respectant les règles de TTCN. Ces verdicts sont ensuite envoyés via le canal de gestion à l'aide des OUTPUTs.

5.2.2 Solution pour l'envoi implicite (Implicit send) :

Dans la solution proposée, le testeur devra envoyer un signal particulier à son utilisateur. Ce signal doit être envoyé sur le canal de gestion et doit avoir comme paramètre la PDU (ou ASP) que l'IST doit envoyer.

5.2.3 Solution pour l'envoi et la réception avec contraintes:

La solution proposée consiste à remplacer chaque contrainte par deux macro-commandes en SDL, une pour l'envoi et l'autre pour la réception.

Dans le cas d'un envoi avec contrainte, la macro consiste à affecter les différentes valeurs de la contrainte aux champs de la PDU (ou de l'ASP). Si l'une des valeurs dans la contrainte est une référence à une autre contrainte la macro-commande correspondante sera invoquée.

Ensuite, l'envoi effectif de la PDU (ou de l'ASP) est effectué.

Dans le cas d'une réception avec contrainte, la macro possède la PDU (ou l'ASP) reçu comme paramètre d'entrée et un booléen comme paramètre de sortie. Ce booléen indique si oui ou non les valeurs reçues correspondent aux constantes qui sont dans la contrainte.

Si l'une des valeurs dans la contrainte est une référence à une autre contrainte la macro-commande correspondante sera invoquée.

La réception avec contrainte s'effectuera alors comme suit :

1- Consommation du message dans la file d'entrée s'il correspond à celui attendu,

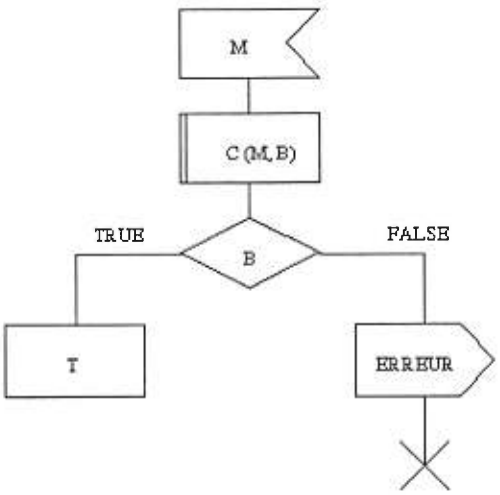
2- Appel de la macro-commande,

3- Si la macro retourne vrai alors la suite de la branche sera exécutée,

4- Sinon il y aura émission d'un message d'erreur et le test bloque, comme c'est le cas dans TTCN.

Exemple :

Soit la réception d'un message M avec une contrainte C en TTCN, et soit T le traitement à effectuer si la réception réussit. En supposant que le nom de la macro appelée en SDL est le même que celui de la contrainte correspondante en TTCN, la représentation en DL.GR obtenue est la suivante :



- **Remarque:**

S'il existe plusieurs alternatives avec des entrées (Inputs) portant le même nom et avec des contraintes différentes, nous proposons la solution suivante:

- regroupement de toutes les entrées en une seule entrée portant le nom en question,
- appel de la macro-commande qui correspond à la première contrainte,

- Si la macro-commande retourne vrai alors la séquence de traitement associée à cette contrainte sera exécutée,
- sinon, nous faisons appel à la macro qui correspond à la contrainte suivante et ainsi de suite,
- au cas où aucune alternative n'a réussi, il y aura émission d'un message d'erreur et le test bloque.

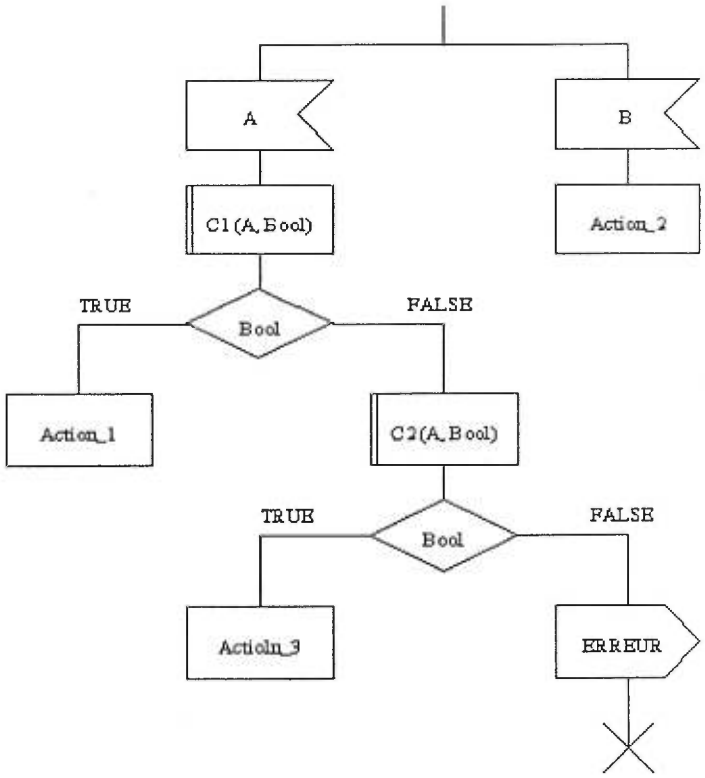
Les exemples suivants illustrent certains problèmes rencontrés lors du passage de TTCN à SDL dans le cas de l'envoi et réception avec contraintes.

Exemple1:

Soit l'alternative suivante représentée en TTCN :

? A		**C1**	**/*C1 est une contrainte*/**
	Action_1		
? B			
	Action_2		
? A		**C2**	**/*C2 est une contrainte*/**
	Action_3		

D'après la solution proposée, la représentation de cette alternative en SDL.GR est la suivante :



Supposons que dans cet exemple, nous recevons en même temps, A avec la contrainte C2, et B. Supposons également que nous les avons reçues dans deux PCOs différents.

D'après TTCN qui impose l'ordre d'exécution de gauche à droite, la branche qui commence par l'entrée B devrait être exécutée. Mais avec la solution proposée, c'est la branche qui commence par l'entrée A avec la contrainte C2 qui va être exécutée.

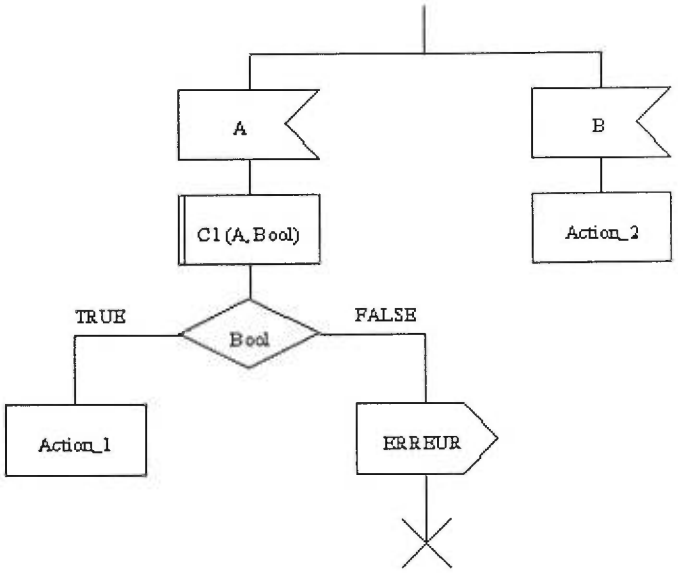
Nous remarquons que la solution ne respecte pas l'ordre d'exécution imposé par TTCN dans le cas de réception simultanée. Par contre, si nous faisons l'hypothèse que chaque message reçu sera aussitôt traité par le testeur, il n'y aura (en pratique) pas de réception simultanée. Donc nous concluons que notre proposition de traduction est acceptable.

Exemple2:

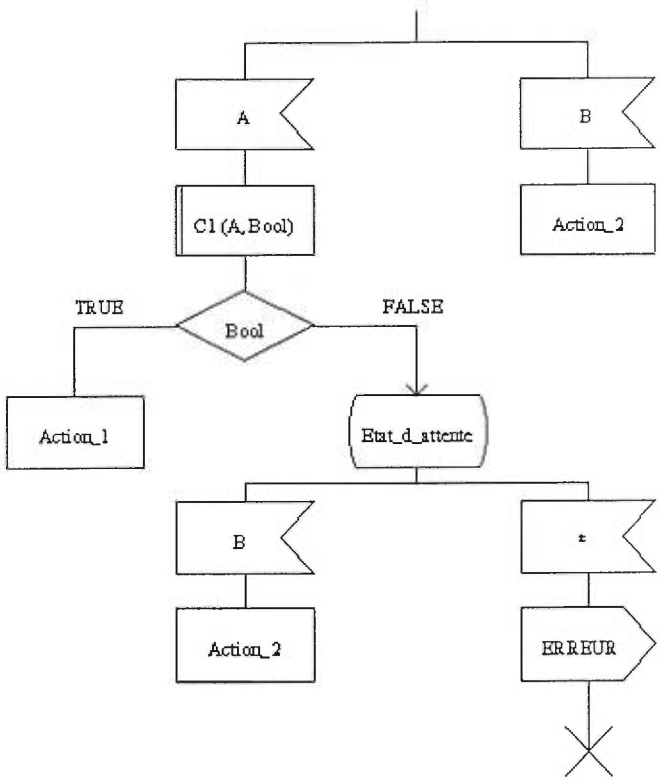
Soit l'alternative suivante représentée en TTCN :

? A Action_1 ? B Action_2	C1 /*C1 est une contrainte*/

La représentation en SDL.GR de cet exemple est la suivante :



Supposons que nous recevons une entrée nommée A avec une contrainte différente de C1 à partir d'un PCO, et une entrée nommée B à partir d'un autre PCO. Avec TTCN, la deuxième alternative s'exécutera. Par contre, d'après la solution proposée, le A est consommé et la macro retourne faux et il y aura message d'erreur. Pour résoudre ce problème, et quand la macro retourne faux, il faut ajouter une autre entrée B. Nous obtenons ainsi la représentation graphique suivante:



La représentation graphique devient plus complexe et par conséquent, cette solution n'est pas intéressante, et comme discuté lors de l'exemple 1, la réception simultanée peut être négligée en pratique.

5.2.4 Autres différences entre SDL et TTCN

Avec le langage SDL, si dans un état donné la file du processus est non vide, le premier signal est extrait et consommé. Si ce signal est attendu alors l'alternative correspondante est exécutée; sinon le signal sera détruit et le prochain signal sera consommé et ainsi de suite. Mais dans le cas où le constructeur *save* est présent, alors le signal est conservé pour la prochaine transition. Dans le cas du langage TTCN, il n'existe pas une fonction équivalente à la fonction *save* de SDL. Si les files d'entrée ne sont pas vides et leur premiers messages ne correspondent pas à ce qui est attendu, alors le test bloque.

Dans SDL, il n'y a pas de notion d'alternative par défaut commune à toutes les attentes d'événements comme le *Default* de TTCN. Mais elle peut être simulée en rajoutant cette alternative partout où une attente d'événements se présente.

Enfin, dans SDL, le seul constructeur qui suit un état doit être une entrée (Input). De ce fait, seul ce constructeur peut débuter une alternative. Par contre, dans TTCN, une alternative peut commencer par un appel de procédure (Attach).

6. Conception d'un éditeur graphique pour TTCN

6.1 Introduction

Le langage standard de spécification de suites de test de conformité, TTCN (Tree and Tabular Combined Notation) , utilise une représentation tabulaire des différents éléments d'une suite de tests, en particulier pour la partie dynamique. Plusieurs utilisateurs ne trouvent pas cette représentation assez intuitive et naturelle. Pour pallier à ce problème, nous proposons un éditeur qui permet une représentation graphique de la partie dynamique plus facile et naturelle à manipuler.

La réalisation de cet éditeur nécessite l'étude de deux aspects, un premier aspect externe qui concerne l'Interface Homme-Machine pour la manipulation de la suite de tests et un autre aspect qui concerne la représentation interne.

Le premier consiste donc en la conception d'une représentation graphique naturelle et facile à comprendre et à manipuler par l'usager. Celle-ci s'inspire essentiellement d'un sous-ensemble de la représentation graphique de SDL, auquel se sont ajoutés de nouveaux éléments conçus pour permettre de représenter des constructions propres aux suites de test. Le choix du langage SDL s'explique par son utilisation au sein d'une vaste communauté industrielle et universitaire. Sa représentation graphique semble intuitive, basée sur les anciens organigrammes, et plusieurs outils le supportent.

Le second aspect consiste en la conception d'une représentation interne d'une suite de tests basée sur l'approche orientée objet (voir section 7.2). Ce choix s'est fait naturellement puisque le langage choisi pour implanter l'outil est orienté objet.

L'outil sera complété par deux traducteurs. Le premier permet la génération d'une suite de tests en TTCN-MP à partir de la représentation interne. Ce traducteur permet donc de passer la suite de tests éditée à d'autres systèmes tels qu'un exécuteur de suites de test, ou d'autres outils pour des suites en TTCN-MP. Le second traducteur permet la génération d'une suite de tests en SDL-PR à partir de la représentation interne.

Notre choix d'un langage d'implantation s'est arrêté sur JAVA pour plusieurs raisons, principalement pour sa grande portabilité (l'outil doit pouvoir fonctionner au moins sur des HP-UX), sa grande librairie graphique pour créer l'Interface Homme-Machine, sa simplicité, et la représentation orientée objet (utile pour décrire les comportement dynamiques).

6.2 Architecture

6.2.1 Architecture conceptuelle de l'outil

La figure 6-1 présente l'architecture conceptuelle de notre outil GETS (Graphical Editor for Test Suite). La partie en ligne continue désigne le travail que nous avons réalisé. Cette architecture comprend les parties suivantes :

• La représentation interne de suites de test: C'est un ensemble de classes d'objets permettant de modéliser les différents éléments et constructions d'une suite de tests. Elle englobera les aspects possibles d'une suite de tests afin de permettre la réalisation d'éventuels traducteurs (en pointillé sur la figure 6-1), comme par exemple un traducteur de TTCN-MP à la représentation interne.

• La représentation graphique de suites de tests: C'est la représentation visible à l'utilisateur. Les parties déclaratives (variables, constantes, contraintes, etc.) sont représentées sous forme de texte en TTCN-MP, tandis que tout le comportement dynamique est montré dans un format graphique similaire à SDL-graphique.

• L'éditeur: Il permet de visualiser la représentation interne en représentation graphique et modifier la représentation interne à partir d'actions sur la représentation graphique.

• Le traducteur MP : Il permet de convertir de la représentation interne des suites de test en format TTCN-MP.

• Le traducteur PR : Il permet de convertir de la représentation interne des suites de test en format SDL-PR.

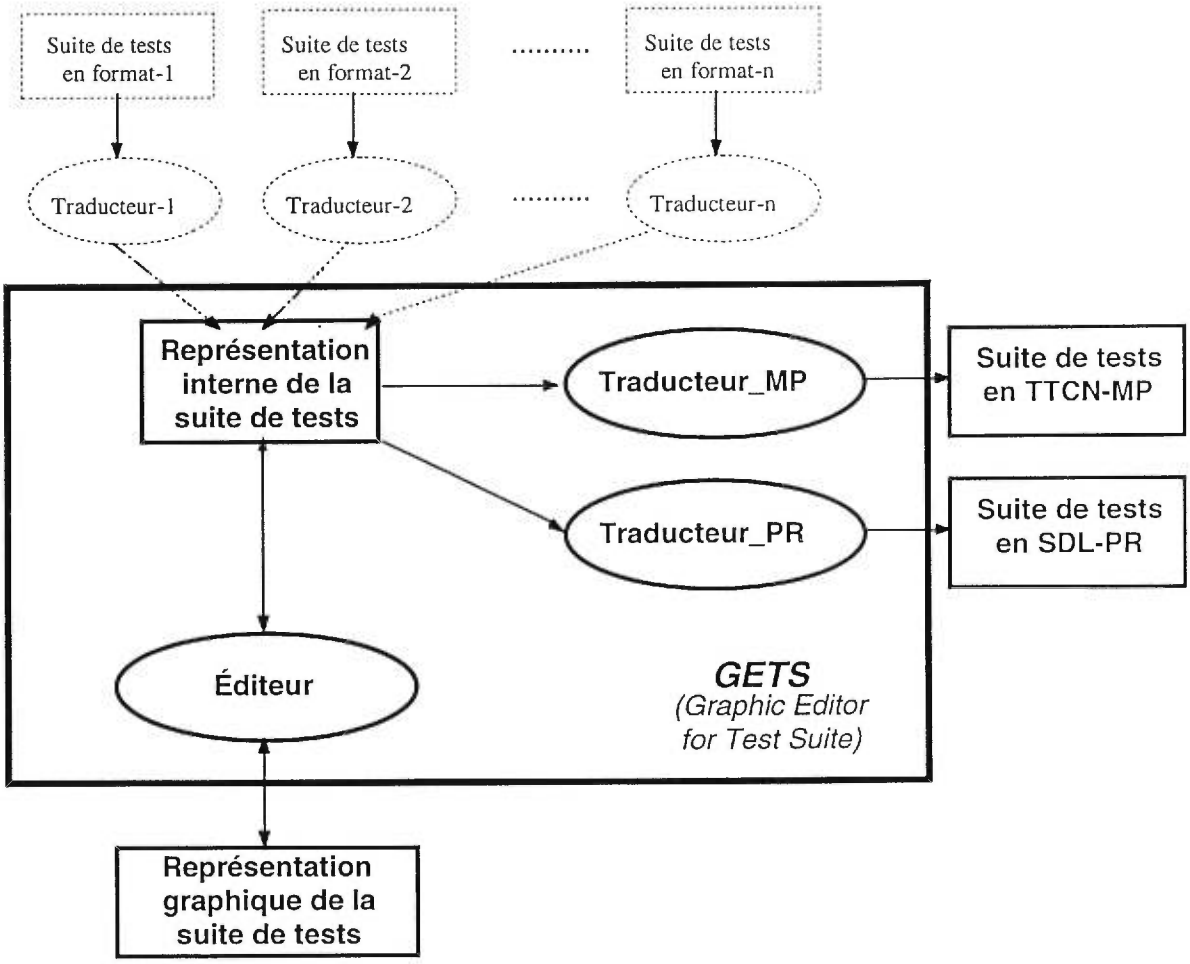


Figure 6-1: Architecture de l'outil

6.2.2 Organisation d'une suite de tests

Chaque suite de tests est décomposable en deux parties: les déclarations comme les PDUs ou les primitives de service (ASP) et le comportement dynamique comme les cas de

tests ou les comportements de défaut. En plus, certaines descriptions de comportement peuvent être regroupées sous certains noms, tels que les préambules, postambules, etc. Ces structures de regroupement des comportements et des déclarations suggèrent fortement une organisation hiérarchique de l'information pour chaque suite de tests. Pour en faciliter l'implantation, nous proposons d'utiliser le système de fichiers Unix pour stocker hiérarchiquement les différents objets d'une suite de tests.

Chaque déclaration peut être stockée dans un fichier sous format texte, en TTCN-MP. Il sera ainsi facile pour un usager plus expérimenté d'éditer une déclaration sans utiliser l'éditeur graphique. Par exemple, des outils comme "sed" (stream editor) pourrait être utilisés pour faire plusieurs modifications en même temps à toutes les déclarations, plus rapidement qu'en utilisant GETS.

Chaque comportement dynamique (cas de tests, étapes de tests, défauts) étant un objet, il peut être stocké dans un fichier sous forme d'objet JAVA "sérialisé" (suite d'octets représentant l'objet).

Comme mentionné auparavant, nous proposons d'utiliser le système de fichiers Unix pour hiérarchiser l'information d'une suite de tests. Le répertoire principal portant le nom de la suite de tests contient deux sous-répertoires: *Declarations* et *Dynamic* contenant respectivement les déclarations et le comportement dynamique de la suite de tests.

Le répertoire *Declarations* est divisé en sept sous-répertoires, représentant les différentes catégories de déclarations, telles que les *ASPs* (Abstract Service Primitives), *PDUs* (Protocol Data Units), *ASP_Constraints* (contraintes d'ASP), *PDU_Constraints*

(contraintes de PDU), *Timers* (horloges), *PCOs* (Points of Control Observation), et *OTHERS* (toutes les autres catégories telles les constantes, alias, etc.). Chacun de ces sous-répertoires contient les fichiers des déclarations (chaque fichier contenant une seule déclaration). Le nom du fichier correspondant au nom d'identificateur de la déclaration. Par exemple, on pourra retrouver dans le sous-répertoire *PDUs* des fichiers textes (contenant des déclarations en TTCN-MP) pour chaque PDU, chaque fichier portant le nom du PDU. De même pour toutes les autres catégories de déclarations.

Le répertoire *Dynamic* contient tous les comportements dynamiques dans chacun des cinq sous-répertoires suivants: *Test_Cases* (cas de test), *Test_Steps* (étapes de test), *Preambles* (préambules de test), *Postambles* (postambules de test), et *Defaults* (défauts de test). Chacun de ces sous-répertoires peut contenir plusieurs fichiers ou d'autres sous-répertoires qui sont des groupes de tests. Chaque fichier contient un comportement (sous forme d'objet JAVA sérialisé) avec comme nom l'identificateur du comportement (nom du cas de test par exemple). Récursivement, chaque sous-répertoire représentant un groupe de tests, peut aussi contenir des fichiers ou d'autres sous-répertoires qui sont des sous-groupes de tests. Et ainsi de suite (voir annexe pour l'organisation des fichiers et répertoires représentant l'information décrivant une suite de tests).

De plus, tous les noms de fichiers doivent être distincts pour respecter la règle de TTCN voulant que tous les noms de tests et de déclarations doivent être distincts, même si ces tests ou déclarations peuvent être distingués par leurs groupes ou catégories de déclarations. Le système GETS gérera une liste interne pour faire cette vérification.

6.3 Représentation graphique

Il existe deux catégories principales d'éléments d'une suite de tests à montrer à l'usager lors de l'édition d'une suite de tests: les déclarations et le comportement dynamique de chaque test. Le centre de test de protocoles (PTC) a mentionné que la partie intéressante à manipuler graphiquement par un usager type est le comportement dynamique, et qu'il

serait pratique de pouvoir utiliser les différentes déclarations sous forme de menus ou listes lors de l'édition du comportement dynamique. L'édition-même des déclarations étant beaucoup moins importante. C'est pourquoi nous avons mis l'emphase sur l'édition graphique du comportement, et moins sur l'édition des déclarations.

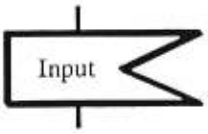
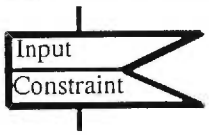
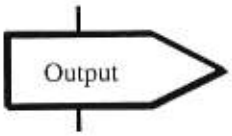
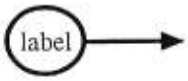
6.3.1 Représentation des déclarations

Il sera possible d'avoir une liste de toutes les déclarations de la suite de tests classées par catégorie et de voir la définition de chacune de ces déclarations. Chaque déclaration sera représentée dans une fenêtre individuelle en format TTCN-MP. Ce format est bien connu, et le texte de chaque déclaration est modifiable à l'aide d'un éditeur de texte simple du même style que TextEdit. Lorsqu'une déclaration est modifiée ou ajoutée à la suite de tests, il n'y a pas de vérification syntaxique ou sémantique, sauf pour en valider l'identificateur et l'ajouter à la liste des déclarations. Il sera toujours possible d'utiliser d'autres outils (tel que Workbench) après la traduction de la suite de tests en TTCN-MP pour faire une validation des modifications.

6.3.2 Représentation graphique du comportement dynamique

Chaque cas de test, étape de test ou défaut de test est représenté graphiquement par une suite de symboles (empruntés à SDL-GR) débutant par l'état initial, connectés de haut en bas pour indiquer l'ordre d'exécution du comportement.

La liste suivante contient les différents symboles utilisés pour décrire un comportement dynamique. Les symboles qui sont une extension à SDL-GR sont indiqués par une étoile "*".

Élément d'une suite de tests	**Représentation graphique**
État initial	
Arrivée d'un signal (Input)	
Arrivée d'un signal (Input) avec vérification des contraintes *	
Envoi d'un signal (Output)	
Étiquette (Label)	
État d'attente d'un événement (Wait State) *	
Connexion de deux symboles	
Branchement à une étiquette (Goto)	

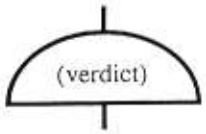
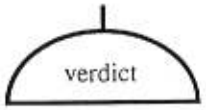
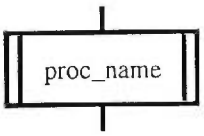
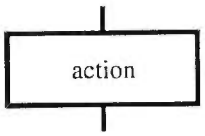
Élément d'une suite de tests	Représentation graphique
Verdict préliminaire *	
Verdict final *	
Appel de procédure (comportement)	
Action	
Décision (If Then Else)	

Figure 6-2: Liste des symboles graphiques

La figure 6-3 montre un exemple d'un comportement décrit en TTCN-GR (notation tabulaire) et la figure 6-4 montre sa représentation graphique:

+FRU0_PREAMBLE	
!ALERT	
START Ts	
(L1) ?REL_COM CANCEL Ts	
+FRU0_VERIFICATION	
+FR_POSTAMBLE	Pass
+FRU0_UNEXPECTED	
GOTO L1	
?TIMEOUT Ts	
FR_POSTAMBLE	Fail

Figure 6-3: Représentation d'un exemple de comportement décrit en TTCN-GR

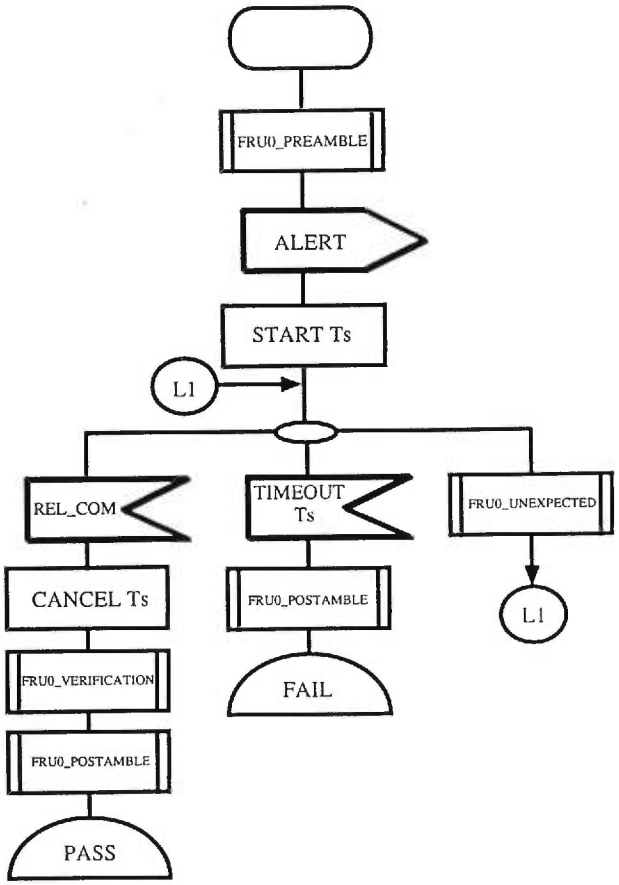


Figure 6-4: Représentation graphique d'un exemple de comportement

6.4 Éditeur graphique d'une suite de tests

Les sous-sections suivantes expliquent les éléments et les fonctionnalités du plus haut niveau du système: l'édition d'une suite de tests complète. Les sections qui suivent (Sections 6.4.1 et 6.4.2) expliquent les autres niveaux d'édition: l'édition des déclarations et l'édition des comportements dynamiques.

6.4.1 Fenêtre principale du système

Seule la fenêtre principale de GETS s'ouvre au démarrage. Comme aucune suite de tests n'est ouverte encore, elle ne contient que les boutons de fonctionnalités du système. La figure 6-5 montre cette fenêtre de démarrage.

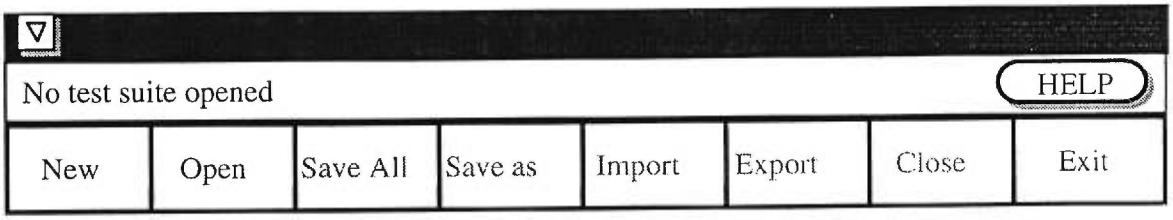


Figure 6-5: Fenêtre principale du système lors du démarrage

Cette fenêtre est composée d'une suite de boutons pour activer les différentes fonctions (*New*, *Open*, etc.). Les fonctions non disponibles sont toujours en gris. Entre les boutons et la barre d'entête se trouve une région indiquant la suite de tests en cours d'édition (dans le cas de la figure 6-5, un message indiquant qu'aucune suite de tests n'est ouverte), et un bouton pour obtenir de l'aide générale sur cette fenêtre.

Si une suite de tests est ouverte (par *Open*, *New* ou *Import*), le nom de la suite de tests apparaît dans l'entête et les fenêtres *Declarations* et *Dynamic Behavior* apparaissent. La figure 6-6 montre un exemple d'une suite de tests ouverte.

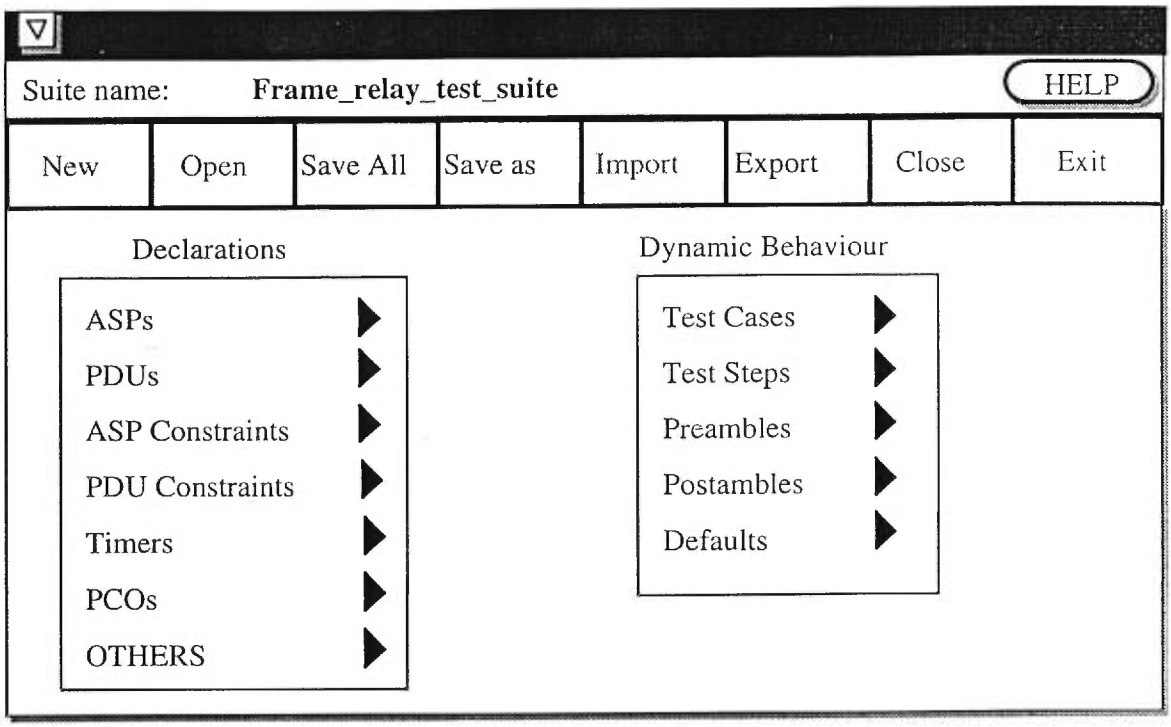


Figure 6-6: Fenêtre principale du système avec une suite de tests ouverte

Les deux listes d'éléments *Declarations* et *Dynamic Behaviour* sont organisées selon la même hiérarchie que celle définie à la section 2.2. En réalité, ces listes utilisent l'information hiérarchisée dans les différents sous-répertoires comme définis à la section 2.2.

Pour la liste *Declarations*, chaque double-clique sur une catégorie de déclarations (*PDUs*, *ASPs*, etc.) ouvre une fenêtre (si elle n'est pas déjà ouverte) contenant une liste de toutes les déclarations de cette catégorie avec une bande de défilement ("scroll-bar"). L'usager peut alors faire défiler les déclarations de cette catégorie et ouvrir la fenêtre correspondante à une déclaration spécifique en double-cliquant sur le nom de la déclaration dans la liste. Le comportement de la liste *Declarations* et des exemples de fenêtres sont donnés plus en détails à la section 6.5.

La liste *Dynamic Behaviour* est différente à cause de son organisation hiérarchique à multiples niveaux (contrairement aux déclarations qui n'ont que deux niveaux seulement).

Lorsqu'un usager double-clique sur un des éléments de la liste des *Dynamic Behaviour*, une fenêtre de navigation (si elle n'est pas déjà ouverte) est activée au niveau le plus élevé dans la hiérarchie des comportements de ce type d'élément. Un exemple de cette fenêtre est montré à la figure 6-7 si l'usager avait double-cliqué sur *Test Steps*, en supposant la hiérarchie donnée dans l'exemple de la section 2.2. Les différentes fonctions accessibles à partir de cette fenêtre sont décrites à la section 4.3.

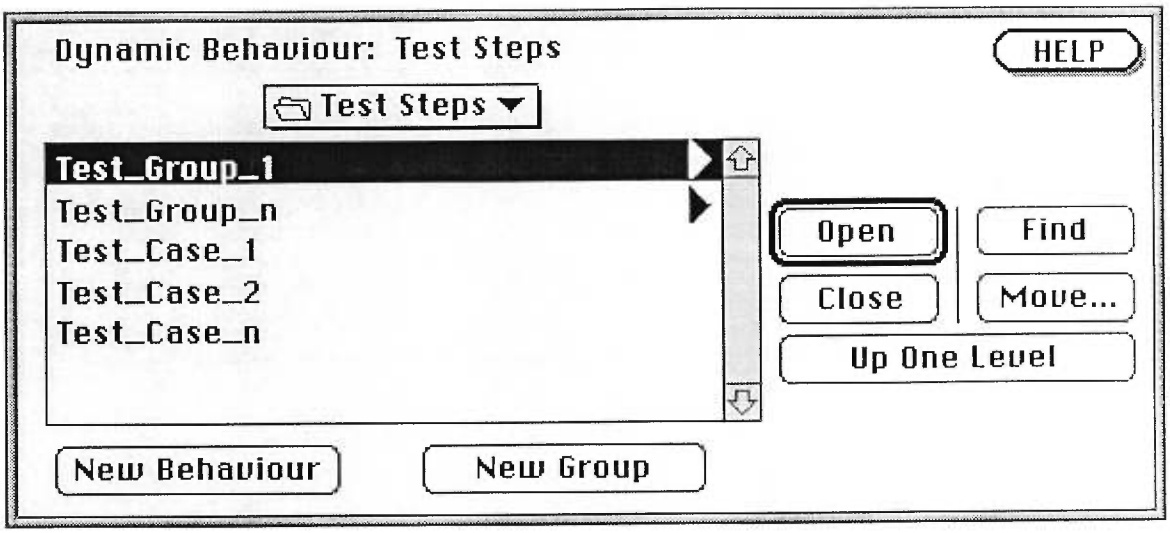


Figure 6-7: Fenêtre de navigation des comportements dynamiques

6.4.2 Fonctions de l'éditeur de la fenêtre principale

Les différentes fonctions disponibles à partir de la fenêtre principale d'édition de la suite de tests sont accessibles par des boutons situés en-dessous du nom de la suite de tests (voir figures 6-5 et 6-6). Ces fonctions sont:

•*New* Création d'une nouvelle suite de tests, une fenêtre de dialogue permet de donner un nom à la suite de tests.

•*Open* Ouverture d'une suite de tests existante, une fenêtre de dialogue permet de naviguer dans les répertoires pour trouver la suite de tests.

•***Save All*** Sauvegarde de la suite de tests complète, permet de sauvegarder toutes les déclarations ou comportements qui sont en cours d'édition et qui ne sont pas encore sauvegardés. C'est l'équivalent de faire un *Save* sur chaque fenêtre d'édition ouverte.

•***Save as*** Sauvegarde de la suite de tests complète (comme *Save All*) mais sous un nouveau nom ou/et dans un autre répertoire.

•***Import*** Appelle un des traducteurs (voir figure 6-1) pour traduire une suite de tests écrite dans un format donné vers notre représentation interne, et ouvre cette suite de tests dans la fenêtre principale. Cette fonction ne sera pas disponible à la première version du système GETS.

•***Export*** Traduction de la suite de tests qui est ouverte en format TTCN-MP. Une fenêtre de dialogue permet de donner un nom et un répertoire au fichier résultant.

•***Exit*** Quitte l'outil. Si une suite de tests est ouverte, une fenêtre de dialogue demande si l'usager veut la sauvegarder.

•***Help*** Aide générale sur l'utilisation de l'éditeur de suite de tests.

6.4.3 Fonctions de la fenêtre de navigation des comportements

La fenêtre de navigation permet de se déplacer dans la hiérarchie des comportements dynamiques. Cette hiérarchie est la même que celle décrite à la section 2.2 et chaque sous-répertoire (représentant un sous-groupe) est indiqué par un triangle à la fin pour montrer qu'il est possible de descendre encore dans la hiérarchie par ce sous-groupe. En haut de la liste, il y a un menu avec le chemin inverse parcouru dans la hiérarchie des comportements dynamiques depuis la racine et qui donne la possibilité de remonter dans la liste de plusieurs niveaux à la fois. La liste des éléments dans la fenêtre de navigation est ordonnée par ordre alphabétique, les sous-groupes en premier, puis les comportements dynamiques à la suite.

Lorsqu'un comportement dynamique est double-cliqué, la fenêtre d'édition de comportement est ouverte avec ce comportement, prêt à être édité, tel que décrit à la section 6.6.

Lorsqu'un sous-groupe (indiqué par un triangle) est double-cliqué, la fenêtre descend à ce niveau dans la hiérarchie. La liste est mise à jour pour montrer les sous-groupes et comportements de ce niveau, et le menu en haut de la liste montre le nouveau nom courant du niveau.

Voici la liste des fonctions disponibles dans cette fenêtre:

•***Open*** Ouverture du dernier élément sélectionné de la liste. L'équivalent d'un double-clique.

•***Find*** Permet de faire une recherche textuelle d'un nom de comportement ou de sous-groupe, à partir de ce niveau de la hiérarchie, vers le bas de l'arbre. La fenêtre déplace ses pointeurs automatiquement vers le nom trouvé, et le sélectionne.

•***Move...*** Permet de déplacer un comportement ou un sous-groupe vers un autre sous-groupe. Après avoir sélectionné l'élément à déplacer, il faut cliquer ce bouton. Le bouton *Move...* se change en *...HERE*, qui devra être cliqué lorsque l'usager aura déterminé l'endroit d'arrivée de l'élément dans la hiérarchie, en naviguant avec cette même fenêtre. Un message en haut des boutons indique que l'usager est en train de déplacer un comportement (*Moving behaviour Behaviour_Name to...*). Après avoir cliqué sur *...HERE*, l'élément est déplacé à l'endroit indiqué par la fenêtre de navigation et le bouton redevient *Move...*; si la fenêtre est fermée ou l'usager essaie de créer un nouveau comportement avant d'avoir cliqué sur *...HERE*, l'opération de déplacement est annulée.

•***Close*** Ferme la fenêtre de navigation.

•***UpOneLevel***

Permet de remonter d'un niveau dans la hiérarchie. Au niveau le plus élevé de la hiérarchie, ce bouton n'a aucun effet.

•*New Behaviour*

Permet de créer un nouveau comportement dynamique dans le groupe courant indiqué par la fenêtre de navigation. Un dialogue demande à l'usager le nouveau nom du comportement, et ensuite la fenêtre d'édition de comportement (voir section 6.6) est ouverte pour éditer ce nouveau comportement.

•*New Group*

Permet de créer un nouveau sous-groupe dans le groupe courant indiqué par la fenêtre de navigation. Un dialogue demande à l'usager le nouveau nom du sous-groupe, et ensuite la liste des sous-groupes et comportements est mise à jour.

•***Help*** Aide générale sur l'utilisation de la fenêtre de navigation des comportements.

6.5 Éditeur de déclarations

La liste des catégories de déclarations (figure 6-8) se retrouve sur la fenêtre principale d'une suite de tests et sur la fenêtre d'édition de comportement dynamique. Sa fonctionnalité est la même dans les deux cas. Chaque double-clique sur une catégorie de déclarations (*PDUs*, *ASPs*, etc.) ouvre une fenêtre (si elle n'est pas déjà ouverte) contenant une liste de toutes les déclarations de cette catégorie avec une bande de défilement ("scroll-bar").

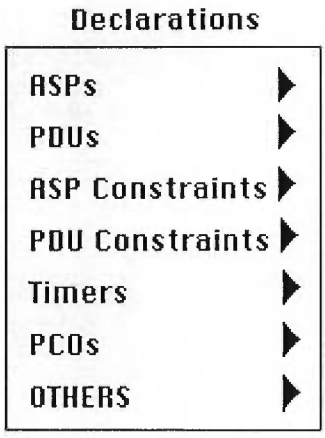


Figure 6-8: Liste des catégories de déclarations

La figure 6-9 donne un exemple si l'usager avait double-cliqué sur *PDUs*. L'usager peut alors faire défiler les déclarations de cette catégorie (tous les PDUs de la suite de tests dans l'exemple à la figure 6-9) et ouvrir la fenêtre correspondant à une déclaration spécifique en double-cliquant sur le nom de la déclaration dans la liste. Cette fenêtre d'édition de déclarations est décrite à la section 6.5.2.

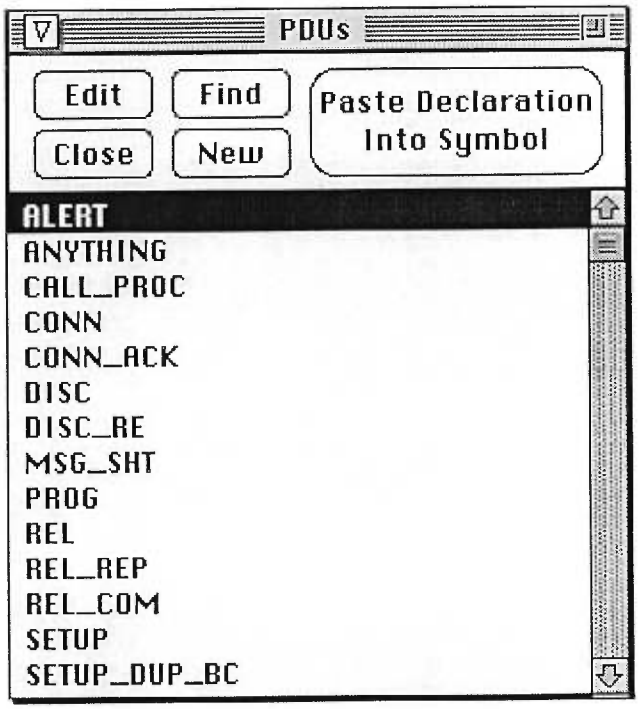


Figure 6-9: Fenêtre de liste de déclarations par catégorie

6.5.1 Fonctions de l'éditeur de la fenêtre des déclarations

Chaque fenêtre de déclarations (*PDUs*, *ASPs*, etc.) offre ces fonctions sous forme de boutons (comme l'exemple à la figure 6-9) pour manipuler sa catégorie de déclarations:

•***Edit*** Ouverture d'une fenêtre d'édition de texte de la dernière déclaration sélectionnée (fenêtre d'édition de texte comme expliquée à la section 6.5.2). Un double-clique sur la déclaration dans la liste a le même effet.

•***Find*** Permet de faire une recherche textuelle d'un nom de déclarations. La fenêtre déplace son pointeur automatiquement vers le nom trouvé en le sélectionnant.

•***Close*** Ferme la fenêtre de cette liste des déclarations.

•***New*** Permet de créer une nouvelle déclaration dans cette catégorie. Un dialogue demande à l'usager le nouveau nom de la déclaration, et ensuite ouvre une fenêtre d'édition de texte pour définir cette déclaration (fenêtre d'édition de texte comme expliquée à la section 6.5.2).

•***Paste Declaration Into Symbol***

Copie l'identificateur de déclaration sélectionné pour ensuite le coller dans le dernier symbole sélectionné dans la fenêtre d'édition de comportement dynamique. Dans le cas de certains identificateurs qui peuvent contenir des paramètres, le système vérifie à ce moment la définition de la déclaration pour extraire ces paramètres, et les paramètres sont aussi copiés et collés.

L'usager peut aussi créer une nouvelle déclaration à partir d'une ancienne en ouvrant la déclaration à copier avec *Edit*, puis en faisant un *Copy* sur le texte de la déclaration à copier. Dans un deuxième temps l'usager doit faire un *New* avec le nouveau nom et faire un *Paste* du texte de la déclaration copiée.

L'usager peut aussi renommer une déclaration en éditant son nom dans le texte de la déclaration. À la fermeture de la fenêtre du texte de la déclaration, le système vérifie le nom (la seule validation faite par le système) et change son nom dans la liste des déclarations.

6.5.2 Fenêtre d'édition de déclarations

L'édition d'une déclaration s'effectue dans une fenêtre d'édition de texte ordinaire. Cette fenêtre d'édition de texte permet de modifier le texte, et possède un bouton pour fermer et sauvegarder la déclaration et un autre pour annuler l'édition de déclaration en cours. Un menu dans cette fenêtre d'édition permet les opérations de *cut/copy/paste/clear* sur le texte. Comme chaque déclaration est sous format texte (TTCN-MP), l'usager modifie la déclaration directement en format texte. Aucune validation de ce texte n'est effectuée par l'éditeur, sauf pour le nom de la déclaration qui est validé (pour s'assurer que tous les identificateurs sont distincts et pour l'ajouter à la liste des déclarations). Lors de la traduction de la suite de tests au complet en TTCN-MP, ce texte est recopié sans aucune validation non plus. L'usager devrait prévoir un autre outil après la traduction en TTCN-MP pour valider et corriger s'il y a lieu les déclarations éditées avec GETS.

6.6 Éditeur graphique des comportements dynamiques

L'ouverture d'un comportement dynamique (cas de test, étape de test, défaut de test) démarre une fenêtre d'édition montrant le comportement avec la représentation graphique tel que décrit à la section 6.3. Cette fenêtre contient tous les éléments nécessaires pour

modifier ou créer un comportement dynamique. De plus, les déclarations de la suite de tests sont accessibles à partir de cette fenêtre. Les différentes composantes de la fenêtre d'édition de comportement sont décrites à la section 6.6.1, et les différentes fonctions accessibles à partir de ces composantes sont expliquées à la section 6.6.2.

6.6.1 Fenêtre d'édition de comportement dynamique

La fenêtre d'édition de comportement dynamique contient quatre régions principales. Il y a en haut les en-têtes, la palette des symboles de comportement est à droite dans le sens de la verticale, puis en bas une région de texte pour éditer ou voir le texte contenu dans le

symbole de comportement sélectionné, et finalement au centre à gauche occupant le plus grand de la fenêtre il y a la région avec le comportement sous forme graphique. Cette dernière région contient des bandes de défilement ("scroll-bar") pour permettre d'accéder à des comportements plus grands que la fenêtre d'édition. Un exemple d'une fenêtre d'édition d'un cas de test est montré à la figure 6-10, tandis que la liste des items dans les menus (menus déroulés) est montrée à la figure 6-11.

Région "en-tête"

L'en-tête de la fenêtre d'édition contient dans le haut la barre de titre qui porte le nom du comportement de la fenêtre, avec le groupe principal du comportement suivi des deux points (*Test case:* par exemple) et ensuite dans l'ordre hiérarchique le nom des sous-groupes du comportement (une barre oblique sépare les sous-groupes), puis le nom du comportement.

En dessous, il y a une barre de menu qui contient les menus *File Edit Declarations Behaviour* et *Help*. La figure 6-11 montre les items de chacun de ces menus (avec les menus déroulés).

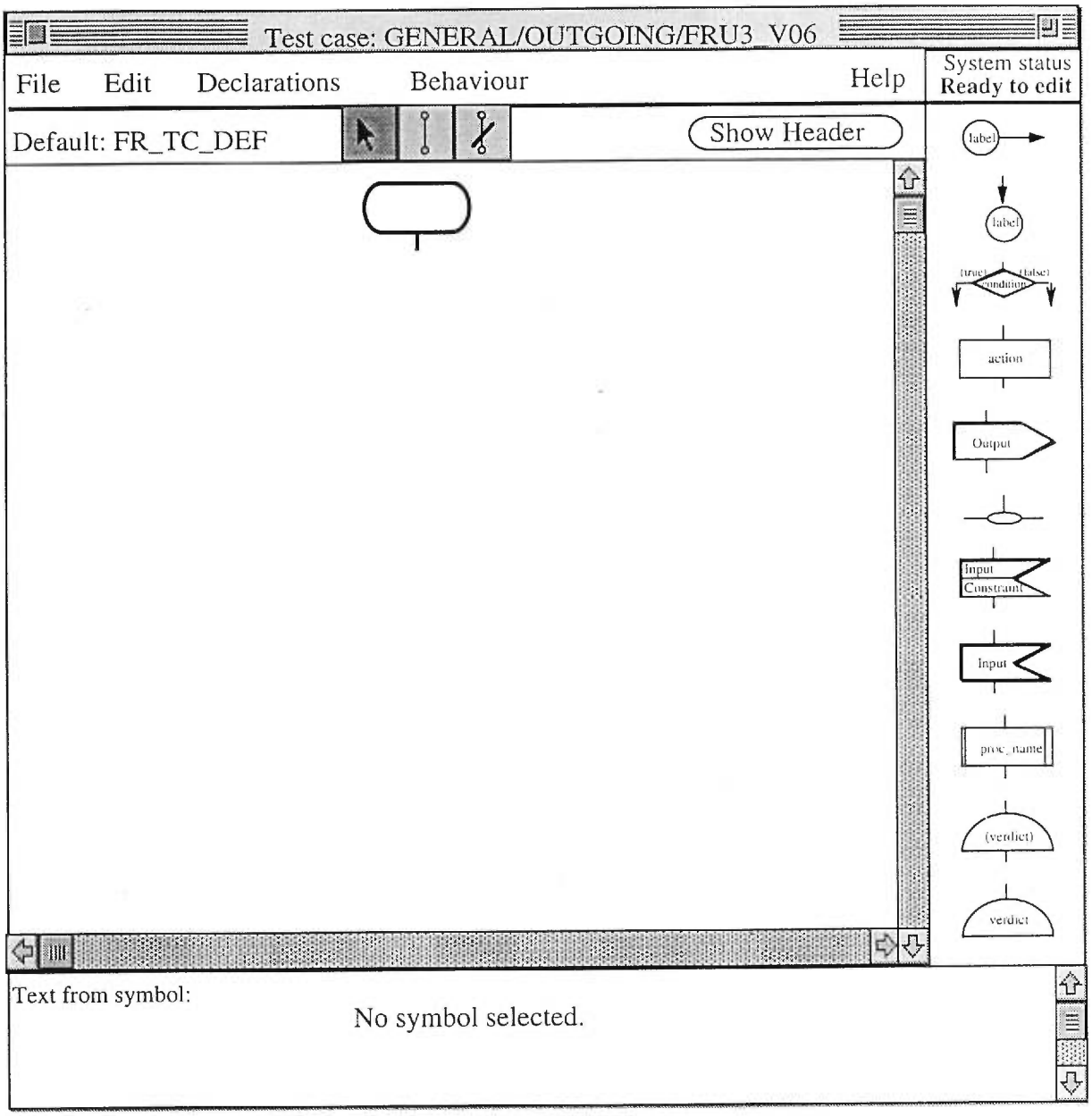


Figure 6-10: Fenêtre d'édition d'un comportement dynamique

Toujours dans la même région près de la barre de menus et du titre, une section contenant un bouton qui permet de voir et d'éditer l'en-tête du comportement dans une fenêtre appropriée (*Show Header*) et le nom du comportement de défaut (éditable avec le bouton *Show Header*). La partie de défaut apparaît vide dans la fenêtre d'édition d'un

comportement de défaut (un comportement de défaut ne peut pas avoir de comportement de défaut). L'en-tête ("Header") d'un comportement est composé d'un défaut de comportement, d'une description textuelle des objectifs de comportement (l'équivalent en TTCN de "Test Purpose" pour les cas de test et de "Objective" pour les étapes et défauts de test), et de commentaires plus détaillés (toujours sous forme textuelle). Un sélecteur de curseur occupe la place centrale qui permet de passer d'un mode d'édition à un autre en changeant le curseur: avec la flêche la souris permet de sélectionner/insérer/déplacer des objets, avec la barre verticale avec bouts arrondis la souris permet de connecter des symboles, et avec la barre verticale avec bouts arrondis mais traversée d'une barre oblique la souris permet de déconnecter des symboles. Puis une dernière région au-dessus de la palette des symboles indique le statut courant de l'éditeur de comportement: par exemple *Ready to edit* au début quand rien n'est sélectionné, ou bien *Inserting Input symbol* si le symbole d'Input vient juste d'être sélectionné.

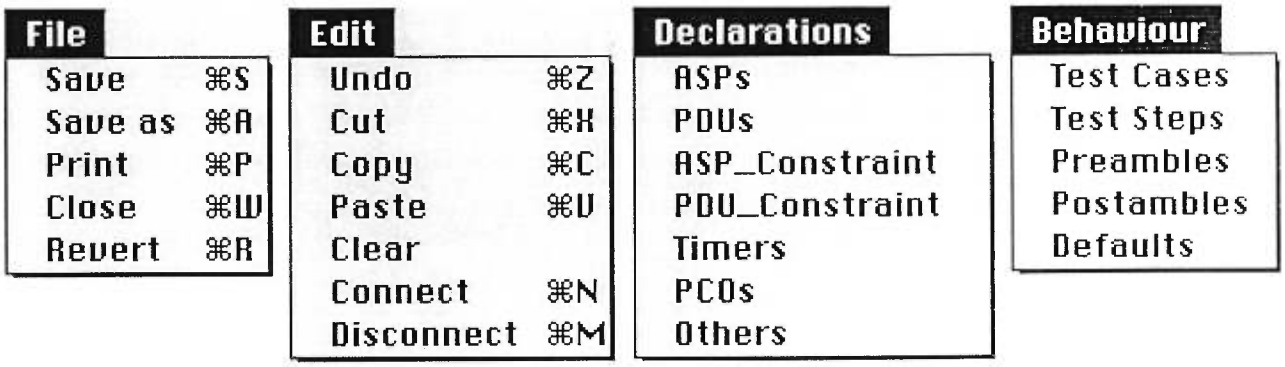


Figure 6-11: Menus déroulés de la fenêtre de comportement dynamique

Région "comportement sous forme graphique"

La région au centre à gauche contient le comportement sous format graphique (comme décrit à la section 6.3) et occupe le plus grand espace de la fenêtre. Des bandes de défilement ("scroll-bar") verticale et horizontale permettent de visualiser un comportement qui serait plus grand que la région d'édition.

Région "palette des symboles"

La région à droite contient la liste sous format graphique de tous les symboles pouvant être utilisés dans un comportement, sauf l'état initial qui est placé automatiquement au début de la page d'édition lorsqu'un nouveau comportement est créé.

Région "texte du symbole sélectionné"

La région en bas complètement de la fenêtre est une région de texte éditable. Quand un symbole est inséré dans le comportement ou qu'un symbole existant dans le comportement est sélectionné avec la souris, cette région s'active et représente le texte inclus dans le symbole choisi. Il est possible d'éditer le texte d'un symbole par cette fenêtre.

Le texte d'un symbole peut comporter des commentaires indiqués avec les symboles /* pour ouvrir le commentaire et */ pour le fermer. Chaque symbole n'affiche que les 15 premiers caractères du texte pour éviter que le texte déborde les frontières du symbole, mais la région du texte du symbole montre tout le texte grâce à la bande de défilement ("scroll-bar").

Dans le cas du symbole de réception de signal avec contrainte, il faut séparer le texte du signal (qui se place en premier dans le texte) de celui de la contrainte. Le mot réservé *CONSTRAINT* en début d'une nouvelle ligne est alors utilisé pour séparer le texte du signal du texte de la contrainte.

6.6.2 Fonctions de l'éditeur des comportements dynamiques

Tout les éléments d'un comportement dynamique peuvent être édités à partir de la fenêtre d'édition de comportements, sauf le titre et les sous-groupes auxquels appartiennent ce comportement. La seule façon de modifier le titre et les sous-groupes est d'utiliser la fenêtre de navigation de comportements (voir section 6.4.3) ou d'utiliser l'item *Save As* dans le menu *File* pour créer un nouveau comportement avec un nouveau nom.

6.6.2.1 Édition de "l'en-tête"

Le comportement de défaut (*Default*) affiché peut être modifié en cliquant sur *Show Header*. Une fenêtre d'édition de l'en-tête permet alors de modifier le comportement de défaut.

En cliquant sur le bouton *Show Header*, une fenêtre avec l'en-tête complète du comportement permet de consulter et modifier chacun des éléments de l'en-tête. Un exemple de cette fenêtre est montré à la figure 6-12. Le comportement de défaut est montré dans la première partie de la fenêtre (en haut), et deux boutons permettent de l'éditer: *Change Default* qui ouvre une fenêtre de navigation permettant la sélection d'un comportement parmi tous les comportements de défaut et *Clear Default* qui permet d'enlever le comportement de défaut complètement de cette en-tête. Les parties de l'en-tête *Purpose/Objective* et *Extended Comments* sont constituées de texte libre éditable à partir de cette fenêtre. Une barre de menu complète les fonctions de cette fenêtre (voir les menus déroulés dans le bas de la figure 6-12). Les fonctions accessibles par le menu sont:

- ***Save & Close***

  Sauvegarde les informations de l'en-tête et ferme cette fenêtre d'édition.

- ***Revert & Close***

  Ferme cette fenêtre d'édition sans sauvegarder les informations qui ont été modifiées (retour aux anciennes valeurs lors de l'ouverture de cette fenêtre).

- ***Revert*** Remet les anciennes valeurs de l'en-tête qui étaient valides lors de l'ouverture de cette fenêtre. Les modifications effectuées depuis l'ouverture de cette fenêtre sont alors perdues.

•***Print Header***

Imprime les informations de la fenêtre d'en-tête.

•***Undo / Cut / Copy / Paste / Clear***

Permet les opérations usuelles sur le texte sélectionné dans la partie *Purpose/Objective* ou *Extended Comments*. Ces opérations fonctionnent seulement sur le texte libre (par exemple l'opération *Undo* ne peut être utilisée pour remettre un comportement de défaut à sa valeur précédente).

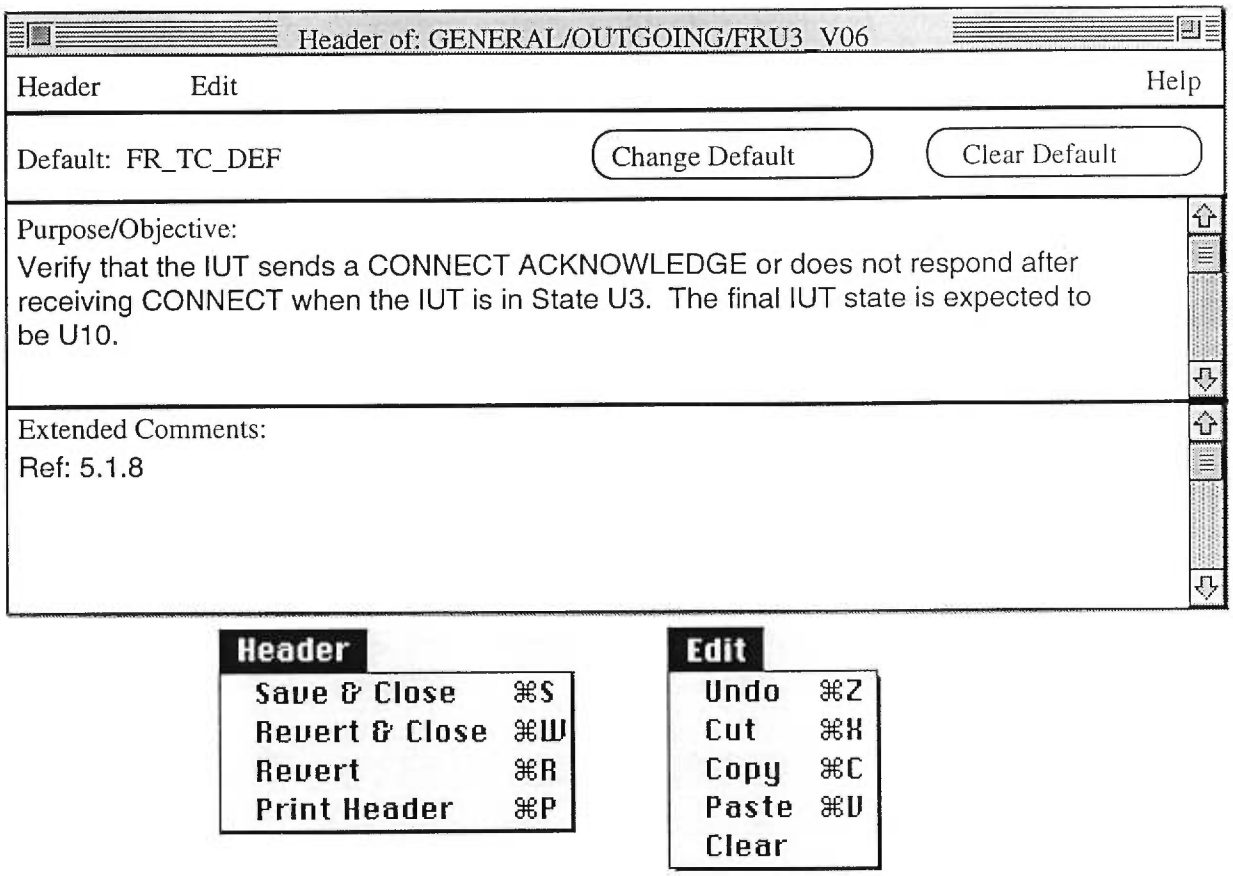


Figure 6-12: Éditeur d'en-tête d'un comportement dynamique

6.6.2.2 Utilisation de la "palette des symboles"

La palette des symboles est utilisée pour insérer des symboles graphiques dans la région du comportement sous forme graphique. Pour insérer un symbole, il suffit de cliquer une fois sur sa représentation dans la palette des symboles (le statut du système indique qu'il est en mode d'insertion de symbole, comme par exemple *Inserting Output*) et ensuite de cliquer

dans la région du comportement sur le symbole auquel il faut attacher le nouveau symbole. Toute opération autre que de cliquer sur un symbole déjà existant dans la région du comportement va annuler l'opération d'insertion et le système redevient en mode *Ready*. Après l'insertion du symbole, celui-ci devient comme s'il était sélectionné pour pouvoir insérer le texte qui doit aller dans le symbole nouvellement ajouté. Ce texte doit être écrit dans la région de "texte du symbole sélectionné", en bas de la fenêtre. Le système affiche en tout temps dans le symbole les 15 premiers caractères de la région de texte du symbole. Cet affichage dans le symbole se fait au fur et à mesure (en temps réel) que l'usager fait des modifications à la région de texte du symbole. Lorsque l'usager fait n'importe quelle opération autre que d'éditer ce texte, le système termine l'insertion du symbole.

Pour le symbole de verdict final, lorsque l'usager l'insère, une fenêtre de dialogue apparaît avec quatre boutons représentant les verdicts: *PASS*, *FAIL*, *INCONCLUSIVE*, et *RESULT*. L'usager peut choisir un des quatre verdicts en cliquant sur le bouton approprié, et le verdict s'insère dans le texte du symbole. Il est aussi possible à l'usager d'insérer des commentaires après ce verdict, mais le premier mot du symbole doit nécessairement être un des quatre verdicts valides. Pour le verdict préliminaire c'est la même chose sauf qu'il n'y a que trois verdicts: *PASS*, *FAIL* et *INCONCLUSIVE*.

Dans le cas du symbole d'état d'attente d'événements, il peut se séparer en plusieurs branches. Quand l'usager ajoute un symbole à la suite de ce symbole d'état d'attente, une nouvelle branche est commencée immédiatement à la droite des autres branches, et un réalignement automatique de toutes les branches est effectué (pour les rendre équidistantes

les unes des autres). De plus, le système vérifie que tous les symboles connectés en bas du symbole d'état d'attente sont des symboles d'attente de signal. Une seule exception est permise pour une et une seule branche qui devient un "OTHERWISE" (comme en TTCN). Cette branche "OTHERWISE" sera toujours la dernière à droite de toutes les branches, même si l'usager essaie d'insérer un symbole d'attente de signal à la suite, cette branche ajoutée ira s'insérer comme avant-dernière, avant le "OTHERWISE". Si l'usager essaie de déroger de ces règles pour le symbole d'état d'attente d'événements, un message approprié le lui indiquera. Une version plus sophistiquée du système pourrait permettre d'ordonner les branches d'un symbole d'état d'attente d'événements selon un ordre établi par l'usager, mais ce n'est pas prévu pour cette version.

Il existe deux exceptions quant à l'endroit à cliquer sur la région du comportement pour insérer un symbole. La première étant l'insertion d'une étiquette ("label") qui se fait en cliquant sur le connecteur directement à l'endroit où l'étiquette doit être ajoutée. La deuxième exception est l'ajout d'un symbole à la suite d'un symbole de décision qui pourrait être ambiguë à cause des deux branches qui sortent d'une décision. Pour régler ce problème, l'usager devra cliquer sur la partie gauche du symbole de décision pour l'ajouter au comportement du côté gauche de la décision ("True" de la décision) et sur la partie droite du symbole de décision pour l'ajouter au comportement du côté droite ("False" de la décision).

Par exemple, si on veut insérer un *Output* après l'état initial (et qu'il n'y a rien encore dans la région du comportement sauf l'état initial), il suffit de cliquer sur le symbole *Output* dans la palette des symboles, le *System status* indique alors *Inserting Output*, puis cliquer sur le symbole d'état initial dans la région du comportement. Le symbole d'*Output* se place tout de suite en bas de l'état initial, et se connecte automatiquement à l'état initial, et le *System status* indique ensuite *Editing Output* pour indiquer que l'*Output* est sélectionné et qu'il faut maintenant insérer le texte de l'*Output*. L'usager peut alors écrire dans la région indiquée par *Text from symbol* en bas le signal qui doit aller dans le symbole d'*Output*. Normalement l'usager se servira du menu Declarations pour insérer un indentificateur de

déclaration au lieu de le taper dans son entier. Ensuite il suffit d'un click avec la souris n'importe où dans la fenêtre pour terminer l'édition du texte. Si le dernier endroit cliqué ne démarre pas une nouvelle opération, le *System status* retourne à *Ready to edit*, dans le cas contraire le *System status* affiche la nouvelle opération.

6.6.2.3 "Sélection" de symboles

La sélection (cliquer avec la souris dessus) d'un symbole dans la région graphique permet d'activer le texte d'un symbole dans la région de texte du bas pour le visualiser au complet ou l'éditer. La sélection de symboles est possible et active seulement lorsque le curseur courant est la flèche. Pour changer le curseur courant, il faut cliquer avec la souris sur le symbole de la flèche dans la partie de l'en-tête de la fenêtre (voir figure 6-10 dans le haut).

Il est possible de sélectionner plusieurs symboles à la fois en utilisant le Shift-click sur chacun des symboles ou en sélectionnant une région complète contenant des symboles avec la souris. Dans les deux cas, l'effet est le même: plusieurs symboles sont sélectionnés et aucun texte n'apparaît dans la région de texte du bas. Les opérations effectuées le seront sur tous les symboles. Certaines opérations seront impossibles avec plusieurs symboles sélectionnés comme la connexion pour plus de deux symboles: il faut savoir quels couples de symboles il faut connecter ensembles, ce qui est impossible lorsque trois symboles ou plus sont sélectionnés.

Le symbole d'appel de procédure peut aussi être double-cliqué, ce qui fait ouvrir une nouvelle fenêtre d'édition de comportement graphique avec le comportement de la procédure qui a été double-cliqué. L'usager peut ainsi voir un autre comportement en même temps et possiblement l'éditer.

Le symbole de verdict peut aussi être double-cliqué pour avoir la fenêtre avec les trois ou quatre boutons de verdict. L'usager peut ainsi changer le verdict simplement en cliquant un des boutons sans être obligé de taper avec le clavier un autre verdict.

6.6.2.4 Opérations de "connexion/déconnexion" de symboles

Il y a deux façons de connecter ou déconnecter des symboles entre eux. Les symboles ainsi déconnectés peuvent être déplacés et reconnectés. Il est impossible de faire un *Undo* sur les opérations de connexion et déconnexion. Un comportement qui contient des symboles non connectés ne peut être sauvegardé. Un message d'erreur l'indique si l'usager tente de sauvegarder ou fermer la fenêtre.

Les connexions ont un sens puisqu'elles indiquent l'ordre d'exécution des activités d'un comportement dynamique. Ce sens va normalement de haut en bas, et de gauche à droite dans le cas où deux symboles sont à la même hauteur.

La première méthode de connexion/déconnexion consiste à changer le curseur avec le sélecteur de curseur dans la région de l'en-tête. Le curseur du centre dans ce sélecteur de curseur représente la connexion, tandis que le même symbole traversé d'une barre oblique représente la déconnexion. Pour déconnecter des symboles déjà connectés entre eux, il suffit de choisir le curseur de déconnexion, et de cliquer sur chacun des symboles à déconnecter (pour enlever la connexion entre deux symboles, il faut cliquer sur chacun des deux symboles et la connexion disparaît). Il est possible de déconnecter ainsi plus de deux symboles consécutifs en continuant de cliquer sur chacun des symboles à déconnecter, tant et aussi longtemps que l'usager n'a pas changé le curseur. Pour reconnecter, il faut choisir le curseur de connexion et cliquer sur chacun des symboles à connecter (le premier symbole cliqué se connectera au deuxième, qui se connectera au troisième, et ainsi de suite, le sens de la connexion allant du premier au dernier symbole cliqué).

Une deuxième méthode consiste à utiliser les items *Connect* et *Disconnect* du menu d'édition *Edit*. Pour déconnecter un groupe de symboles, il faut sélectionner (avec le curseur de sélection, comme expliqué à la section 6.2.3) un groupe de symboles à déconnecter, puis choisir l'item *Disconnect* dans le menu *Edit*. Tous les symboles sélectionnés enlèvent la connexion qu'ils ont entre eux. Pour connecter un groupe de

symboles entre eux par cette méthode, il faut sélectionner les symboles à connecter, puis choisir l'item *Connect* dans le menu *Edit*. Le sens de la connexion s'effectue du symbole le plus haut dans la fenêtre au second symbole le plus haut et ainsi de suite jusqu'au symbole le plus bas, ou de gauche à droite s'il y a plusieurs symboles à la même hauteur. S'il y a ambiguïté à cause des symboles de décision ou d'état d'attente d'événements, la connexion n'est pas effectuée et un message d'avertissement l'indique à l'usager.

6.6.2.5 Opérations de "copie/élimination/déplacement" de symboles

Il est possible de copier ou d'éliminer un ou plusieurs symboles sélectionnés avec les items *Cut Copy Paste Clear* du menu *Edit*. Les connexions reliant des symboles détruits sont évidemment détruites, et l'usager doit reconnecter les connexions non-branchées du graphique pour que le comportement soit conforme.

Pour copier un symbole, il faut le sélectionner, choisir l'item *Copy* du menu *Edit*, et choisir l'item *Paste* du même menu autant de fois qu'il est nécessaire d'avoir des duplicata. Le texte inclus dans le symbole est lui aussi copié et dupliqué. Ces duplicata ne sont pas connectés et doivent l'être avec les opérations de connexion telles qu'expliquées à la section 6.6.2.4. Il est possible de dupliquer un groupe sélectionné de symboles de la même façon: après la sélection, choisir *Copy*, puis *Paste* insère une copie sur la région graphique avec tout le groupe de symboles encore connectés de la même façon entre eux comme ils l'étaient auparavant, mais avec le symbole du haut non-connecté en haut et celui du bas non-connecté vers le bas.

L'usager peut déplacer un symbole non-connecté en tenant le bouton de la souris enfoncé sur un symbole et le déplace vers sa destination où il relâche le bouton (opération de "drag" standard). La même opération peut s'effectuer sur un groupe de symboles connectés entre eux, mais seulement s'ils sont sélectionnés et isolés du reste du comportement graphique.

6.6.2.6 Menus

La barre de menu contient cinq menus: *File*, *Edit*, *Declarations*, *Behaviour*, *Help*. Le menu *Help* donne de l'aide générale sur la fenêtre d'édition du comportement dynamique.

Le menu *File* est composé des items suivants:

•***Save*** Sauvegarde du comportement représenté dans cette fenêtre. Il faut que tous les symboles soient connectés logiquement ensemble et que le comportement soit conforme.

•***Save as*** Sauvegarde du comportement (comme *Save*) mais sous un nouveau nom. Une fenêtre demande le nouveau nom, et une vérification du nouveau nom est faite pour qu'il soit unique. Le sous-groupe reste le même; l'usager pourra le modifier avec la fenêtre de navigation comme expliqué à la section 6.3.

•***Print*** Impression du comportement graphique (s'il est plus grand qu'une page normale, un découpage selon la grandeur du papier choisi est fait avec numérotation des pages de haut en bas, puis de gauche à droite).

•***Close*** Fermeture complète de la fenêtre d'édition. Si la fenêtre d'édition du *Header* est ouverte, il y a une fermeture de celle-ci en premier, avec la question usuelle de demande de sauvegarde si l'usager a fait des modifications. Ensuite la fenêtre d'édition du comportement est fermée et si l'usager a fait des modifications, une fenêtre de dialogue s'occupe de demander à l'usager s'il désire sauvegarder les modifications (un *Save* comme décrit plus haut) ou les annuler.

•***Revert*** Retourne le comportement graphique de la dernière version sauvegardée et annule les modifications qui auraient pu être réalisées depuis cette version.

Le menu *Edit* est composé d'items connus comme *Undo Cut Copy Paste Clear* utilisés sur les symboles graphiques ou du texte selon le contexte. Les *Cut Copy Clear* s'effectueront sur le ou les derniers éléments sélectionnés (symboles graphiques ou texte selon le cas), et le *Paste* va insérer le ou les derniers éléments copiés avec *Copy* ou *Cut* (symboles graphiques ou texte selon le cas) au dernier endroit cliqué (sur la fenêtre du comportement graphique ou du texte selon le cas) par l'usager. Le *Undo* ne fonctionne que pour les opérations effectuées avec les items *Cut* et *Paste*. Les opérations *Connect* et *Disconnect* ont déjà été expliquées en détail à la section 6.6.2.4.

Le menu *Declarations* fontionne comme la liste *Declarations* décrite à la section 6.4, et ouvre une fenêtre avec les déclarations comme décrit à la section 6.5. Ces déclarations peuvent ensuite être utilisées avec le bouton *Paste Declaration Into Symbol* pour insérer un identificateur de déclaration dans le texte d'un symbole. Lorsque ce bouton est cliqué, l'identificateur de déclaration choisi est inséré dans le texte du dernier symbole sélectionné dans la fenêtre d'édition du comportement graphique.

Le menu *Behaviour* permet d'insérer le nom d'un comportement avec ses paramètres dans le texte d'un symbole d'appel de procédure. La sélection d'un item au menu *Behaviour* démarre une fenêtre de navigation comme celle montrée à la figure 6-13 et dont le comportement est semblable à celui décrit à la section 6.4.3, sauf qu'il est impossible de créer des nouveaux sous-groupes ou comportements et qu'il est impossible d'ouvrir une nouvelle fenêtre d'édition de comportement avec cette fenêtre. Les boutons pour créer les nouveaux sous-groupes/comportements sont remplacés par un bouton dans le bas de la fenêtre *Paste Behaviour Into Symbol* qui copie l'identificateur du comportement sélectionné avec ses paramètres et le place dans le texte du dernier symbole sélectionné de la fenêtre d'édition de comportement courante. La fenêtre se ferme automatiquement après que le comportement soit copié.

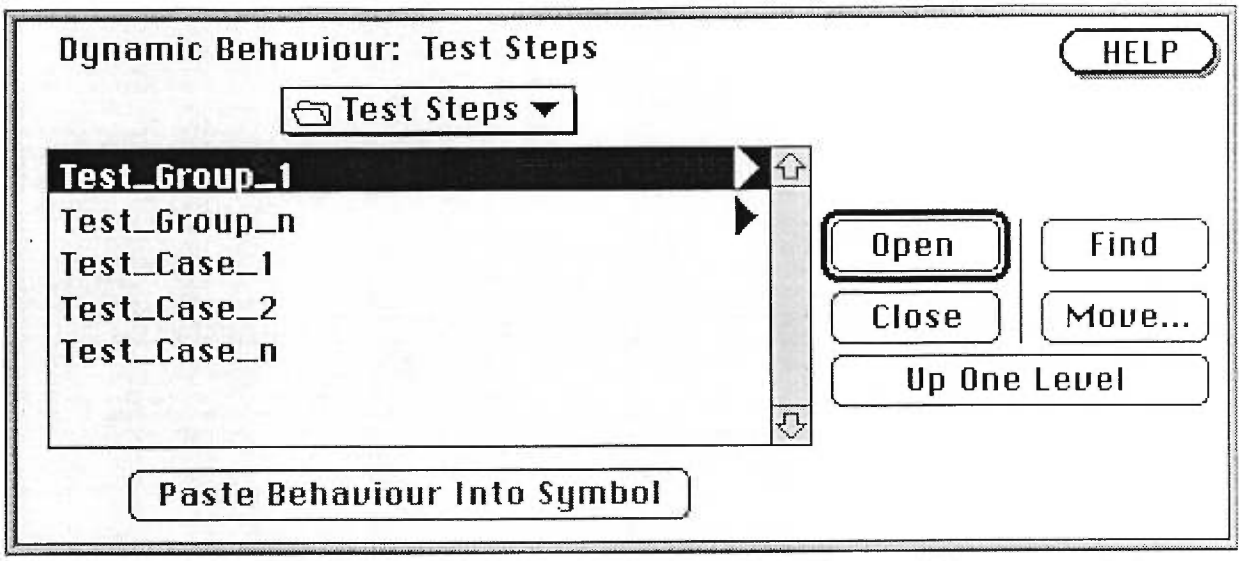


Figure 6-13: Fenêtre de navigation pour insérer un comportement dans un symbole

Si la sélection dans la fenêtre de navigation est un sous-groupe au lieu d'un comportement, le bouton *Paste Behaviour Into Symbol* reste en gris pour montrer qu'il ne peut être activé sur un sous-groupe. La même chose pour le bouton *Open* lorsque la sélection est sur un comportement puisqu'il est impossible d'ouvrir un fenêtre d'édition de comportement avec cette fenêtre de navigation.

6.7 Exemple d'un scénario d'édition d'une suite de tests

Pour mieux comprendre les fonctionnalités du système, voici un exemple de création d'un comportement. On suppose que les déclarations sont déjà dans la suite de tests avec les autres comportements, et qu'un usager veut créer un nouveau comportement *XXX* dans les *Test Steps* qui représente le comportement de la figure 6-4 dans la suite de tests *FR_SVC*. Voici les différentes opérations qu'un usager aurait à effectuer après avoir démarrer le système GETS.

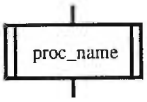
1- À partir de la fenêtre de départ du système GETS (voir figure 6-5), cliquer le bouton Open. Un dialogue d'ouverture de fichiers permet de choisir la suite de tests à éditer: *FR_SVC*.

2- La fenêtre de départ de GETS se transforme comme à la figure 6-6, avec le nom de la suite en haut.

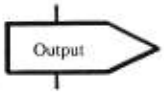
3- L'usager clique sur *Test Steps* dans la liste de *Dynamic Behaviour* de la figure 6-6. Une fenêtre de navigation de comportement s'ouvre (laissant toujours ouverte la fenêtre générale de GETS). Cette fenêtre est identique à la figure 6-7.

4- L'usager clique sur le bouton *New Behaviour* pour créer un nouveau comportement dans les *Test Steps* directement. Une fenêtre de dialogue s'ouvre pour demander le nom du nouveau comportement.

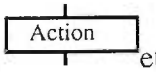
5- L'usager tape *XXX* dans la fenêtre de dialogue de nom, puis clique sur le bouton OK, et cette fenêtre de dialogue se referme. Le système conserve la fenêtre générale de GETS mais ferme la fenêtre de navigation de comportement pour la remplacer par une fenêtre d'édition de comportement identique à la figure 6-10 sauf pour le titre en haut qui se lit: *Test case: XXX*.

6- Il reste maintenant à l'usager à créer le comportement comme montré à la figure 6-4. Pour insérer le préambule *FRU0_PREAMBLE*, l'usager clique sur le symbole d'appel de procédure: 

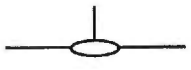
qui se trouve dans la palette des symboles à droite en bas, puis clique ensuite sur le symbole d'état initial dans la région du comportement. Le symbole de comportement se connecte automatiquement en bas de l'état initial et l'usager peut maintenant entrer le nom du comportement soit en tapant directement *FRU0_PREAMBLE* ou en utilisant la fenêtre de navigation de comportement en utilisant le menu *Behaviour* puis en sélectionnant l'item *Preambles*. Avec la fenêtre de navigation, il doit sélectionner *FRU0_PREAMBLE* et cliquer sur le bouton *Paste Behaviour Into Symbol*. La fenêtre de navigation se ferme et le nom du préambule *FRU0_PREAMBLE* s'insère dans le texte du symbole d'appel de procédure.

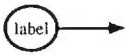
7- L'usager clique dans la palette des symboles sur: 

et clique ensuite sur le symbole d'appel de procédure avec *FRU0_PREAMBLE* dans la région du comportement graphique. Le symbole d'envoi de signal (Output) apparaîtra en-dessous, connecté à l'appel de procédure. Pour insérer le nom du signal, l'usager choisit dans le menu *Declarations* l'item *PDUs*. La fenêtre des PDUs s'ouvre, l'usager peut choisir *ALERT* et ensuite clique sur le bouton *Paste Declaration Into Symbol.* L'identificateur *ALERT* s'insère dans le texte du symbole d'envoi de signal, avec les différents paramètres et des commentaires appropriés pour aider l'usager à compléter le signal. À son choix, l'usager peut fermer la liste des PDUs ou la laisser ouverte pour d'autres insertions.

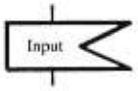
8- Puis l'usager clique dans la palette des symboles sur: 

et clique ensuite sur le symbole d'envoi de signal avec *ALERT* dans la région du comportement graphique. Le symbole d'action apparaîtra en-dessous, connecté à l'envoi de signal. Pour insérer l'action, l'usager tape *START* puis choisit l'item *Timers* dans le menu *Declarations*. La fenêtre des *Timers* s'ouvre, l'usager peut choisir un *Timer Ts* et ensuite clique sur le bouton *Paste Declaration Into Symbol.* L'identificateur du *Timer* (*Ts*) s'insère dans le texte du symbole à la suite du *START.* À son choix, l'usager peut fermer la liste des *Timers* ou la laisser ouverte pour d'autres insertions.

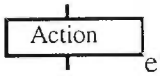
9- L'usager clique dans la palette des symboles sur:  et clique ensuite sur le symbole d'action avec *START Ts* dans la région du comportement graphique. Le symbole d'attente d'événements apparaîtra en-dessous, connecté à l'envoi de signal.

10- L'usager clique dans la palette des symboles sur: 

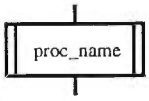
et clique ensuite sur la connexion entre les symboles d'action et d'attente d'événements dans la région du comportement graphique. Le symbole d'étiquette apparaîtra sur cette connexion. L'usager doit alors taper le nom de l'étiquette *L1*.

11- L'usager clique dans la palette des symboles sur: 

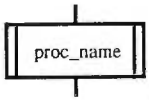
et clique ensuite sur le symbole d'attente d'événements dans la région du comportement graphique. Le symbole de réception de signal (Input) apparaîtra en-dessous, connecté au symbole d'attente d'événements. Pour insérer le nom du signal, l'usager choisit dans le menu *Declarations* l'item *PDUs*. La fenêtre des PDUs s'ouvre, l'usager peut choisir *REL_COM* et ensuite clique sur le bouton *Paste Declaration Into Symbol*. L'identificateur *REL_COM* s'insère dans le texte du symbole d'envoi de signal, avec les différents paramètres et des commentaires appropriés pour aider l'usager à compléter le signal.

12- L'usager clique dans la palette des symboles sur: 

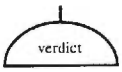
et clique ensuite sur le symbole de réception de signal avec *REL_COM* dans la région du comportement graphique. Le symbole d'action apparaîtra en-dessous, connecté à la réception de signal. Pour insérer l'action, l'usager tape *CANCEL* puis choisit l'item *Timers* dans le menu *Declarations*. La fenêtre des Timers s'ouvre, l'usager peut choisir *Ts* et ensuite clique sur le bouton *Paste Declaration Into Symbol*. L'identificateur *Ts* s'insère dans le texte du symbole à la suite du *CANCEL*.

13- Pour insérer l'étape de test *FRU0_VERIFICATION*, l'usager clique sur le symbole d'appel de procédure: 

dans la palette des symboles, puis clique sur le symbole d'action avec *CANCEL Ts* dans la région du comportement. Le symbole d'appel de procédure se connecte automatiquement en bas de l'action et l'usager peut maintenant entrer le nom du comportement soit en tapant directement *FRU0_VERIFICATION* ou en utilisant la fenêtre de navigation de comportement en utilisant le menu *Behaviour* puis en sélectionnant l'item *Test Steps*. Avec la fenêtre de navigation, il doit sélectionner *VERIFICATION* (parce qu'il est dans ce sous-groupe) et ensuite faire *Open*, puis sélectionner *FRU0_VERIFICATION* et cliquer sur le bouton *Paste Behaviour Into Symbol*. La fenêtre de navigation se ferme et le nom du préambule *FRU0_VERIFICATION* s'insère dans le texte du symbole d'appel de procédure.

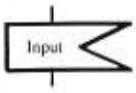
14- Pour insérer le postambule *FRU0_POSTAMBLE*, l'usager clique sur le symbole d'appel de procédure: 

dans la palette des symboles, puis clique ensuite sur le dernier symbole de procédure inséré avec *FRU0_VERIFICATION* dans la région du comportement. Le symbole de comportement se connecte automatiquement en bas de l'autre symbole de procédure et l'usager peut maintenant entrer le nom du comportement soit en tapant directement *FRU0_POSTAMBLE* ou en utilisant la fenêtre de navigation de comportement en utilisant le menu *Behaviour* puis en sélectionnant l'item *Postambles*. Avec la fenêtre de navigation, il doit sélectionner *FRU0_POSTAMBLE* et cliquer sur le bouton *Paste Behaviour Into Symbol*. La fenêtre de navigation se ferme et le nom du préambule *FRU0_POSTAMBLE* s'insère dans le texte du symbole d'appel de procédure.

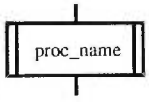
15- Pour terminer cette branche, l'usager doit insérer un verdict *PASS*. L'usager doit cliquer sur le symbole de verdict: 

dans palette des symboles. Ensuite il clique sur le symbole de procédure avec *FRU0_POSTAMBLE* pour insérer le verdict tout de suite

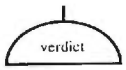
après, à la fin. La fenêtre avec les quatre boutons de verdict apparaît. L'usager doit cliquer le bouton *PASS* et cette branche est maintenant complètement terminée.

16- Pour débuter la deuxième branche à partir du symbole d'attente d'événements, l'usager clique dans la palette des symboles sur: 

et clique ensuite sur le symbole d'attente d'événements dans la région du comportement graphique. Le symbole de réception de signal (Input) apparaîtra en-dessous, connecté au symbole d'attente d'événements. La première branche qui était unique et au centre se déplacera vers la gauche pour que les deux branches maintenant existantes soit alignées. Pour insérer le nom du signal, l'usager choisit dans le menu *Declarations* l'item *Timers*. La fenêtre des Timers s'ouvre, l'usager peut choisir *Ts* et ensuite clique sur le bouton *Paste Declaration Into Symbol*. Le mot *TIMEOUT* suivi de l'identificateur *Ts* s'insère dans le texte du symbole d'envoi de signal.

17- Pour insérer le postambule *FRU0_POSTAMBLE*, l'usager clique sur le symbole d'appel de procédure: 

dans la palette des symboles, puis clique sur le symbole de réception de signal avec *TIMEOUT Ts* dans la région du comportement. Le symbole de comportement se connecte automatiquement en bas de la réception de signal et l'usager peut maintenant entrer le nom du comportement soit en tapant directement *FRU0_POSTAMBLE* ou en utilisant la fenêtre de navigation de comportement en utilisant le menu *Behaviour* puis en sélectionnant l'item *Postambles*. Avec la fenêtre de navigation, il doit sélectionner *FRU0_POSTAMBLE* et cliquer sur le bouton *Paste Behaviour Into Symbol*. La fenêtre de navigation se ferme et le nom du préambule *FRU0_POSTAMBLE* s'insère dans le texte du symbole d'appel de procédure.

18- Pour terminer la deuxième branche, l'usager doit insérer un verdict *FAIL*. L'usager doit cliquer sur le symbole de verdict: 

dans palette des symboles. Ensuite il clique sur le symbole de procédure avec *FRU0_POSTAMBLE* pour insérer le verdict tout de suite après, à la fin. La fenêtre avec les quatre boutons de verdict apparaît. L'usager doit cliquer le bouton *FAIL* et cette branche est maintenant complètement terminée.

19- La dernière branche à partir du symbole d'attente d'événements est l'équivalent d'un "OTHERWISE" en TTCN puisque le premier symbole de la branche n'est pas une attente de signal. Pour insérer l'appel de procédure *FRU0_UNEXPECTED*, l'usager clique sur le symbole d'appel de procédure: 

dans la palette des symboles, puis clique sur le symbole d'attente d'événements dans la région du comportement. Le symbole d'appel de procédure apparaîtra en-dessous, connecté au symbole d'attente d'événements. Les première et deuxième branches se déplaceront vers la gauche pour que les trois branches maintenant équidistantes. L'usager peut maintenant entrer le nom du comportement soit en tapant directement *FRU0_UNEXPECTED* ou en utilisant la fenêtre de navigation de comportement en utilisant le menu *Behaviour* puis en sélectionnant l'item *Test Steps*. Avec la fenêtre de navigation, il doit sélectionner *UNEXPECTED* (parce qu'il est dans ce sous-groupe) et ensuite faire *Open*, puis sélectionner *FRU0_UNEXPECTED* et cliquer sur le bouton *Paste Behaviour Into Symbol*. La fenêtre de navigation se ferme et le nom du test *FRU0_UNEXPECTED* s'insère dans le texte du symbole d'appel de procédure.

20- La dernière branche est terminée par un branchement à *L1*. Pour l'insérer, l'usager doit cliquer sur le symbole de branchement: 

dans la palette des symboles. Ensuite il clique sur le symbole de procédure avec *FRU0_UNEXPECTED* pour insérer le

branchement tout de suite après, à la fin. L'usager doit taper l'étiquette *L1* et cette branche est maintenant complètement terminée.

L'édition complète du comportement est terminée. Il ne reste plus qu'à le sauvegarder avec la suite de tests.

21- L'usager choisit dans le menu *File* les items *Save* puis *Close* par la suite. Le comportement est maintenant complètement sauvegardé, et l'usager peut au choix terminer l'exécution du programme en choisissant *Exit* sur la fenêtre principale ou continuer à éditer d'autres comportements.

7. Implantation de l'éditeur

Pour le développement de l'outil GETS, nous avons implanté les parties suivantes:

- un éditeur de suites de tests,
- un traducteur qui génère du code TTCN-MP à partir de la représentation interne,
- un traducteur qui génère du code SDL-PR à partir de cette représentation.

L'éditeur GETS a été réalisé avec le langage Java sous UNIX. Ce langage a été choisi en raison de sa portabilité et la richesse de ses bibliothèques, en particulier sa bibliothèque graphique.

Dans ce chapitre, nous présenterons le langage Java avec ses particularités, ses possibilités et particulièrement les paquetages utilisés pour la réalisation de ce projet. Nous décrirons également la représentation interne des différentes classes d'objets qui ont servi à la réalisation du projet. Enfin nous donnerons un aperçu sur la réalisation de l'éditeur de suites de tests et des traducteurs.

7.1 Le langage Java

7.1.1 Généralités

7.1.1.1 Historique

À l'origine Java était destiné au développement de logiciels embarqués ("embedded software"). Ce type de logiciel doit être, plus que tout autre, portable, robuste sans fuite mémoire, etc. Java est né en quelque sorte du constat de la part de la compagnie Sun de l'inadéquation des langages classiques tel que C++ vis-à-vis de ce type de logiciels (inadéquation due, pour l'essentiel, aux fonctionnalités de bas niveau que C++ a hérité de C, surtout la gestion mémoire). Une équipe de Sun s'est donc lancée dans la conception d'un nouveau langage intégrant les meilleures techniques utilisées par les langages à objets de haut niveau tels que Eiffel[Meyer 88], Smalltalk[Goldberg 83] et Objective C[Obj 93], tout en restant syntaxiquement proche de C++. Cette ressemblance avec C++ est très

importante, puisqu'elle permet à un grand nombre de développeurs d'assimiler Java très facilement.

7.1.1.2 Environnement de développement et d'exécution

La plupart des points forts de Java proviennent des fonctionnalités offertes par son environnement d'exécution et des techniques de compilation utilisées.

Une application écrite en Java se présente sous forme d'un ensemble de classes, chaque classe décrivant un type d'objet particulier et contenant les méthodes permettant de manipuler ces objets. Lorsque l'on compile une classe Java, on obtient un fichier dont le nom a le suffixe ".class" contenant, entre autres, le code de la classe traduit en un langage intermédiaire indépendant de toute architecture et pouvant être exécuté par une machine virtuelle Java (Figure 7-1).

Ce code peut être transporté d'une machine à une autre pour y être exécuté. Il suffit pour cela que la machine cible dispose d'un environnement d'exécution Java. Cet environnement est constitué d'une machine virtuelle Java et d'un ensemble de classes utilitaires gérant l'utilisation des ressources de la machine, le chargement des classes, etc.

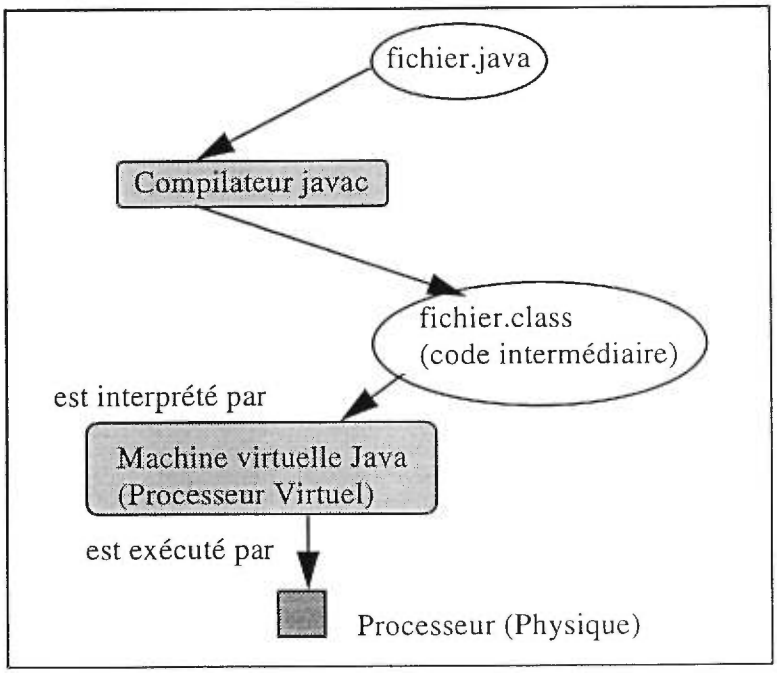


Figure 7-1: Vie d'une application Java

7.1.1.3 Robustesse

La gestion de mémoire est automatique, l'arithmétique des pointeurs est interdite et les conversions de types sont systématiquement vérifiées. En plus, contrairement à C++ qui permet d'adopter un style de programmation orienté objet tout en supportant d'autres styles, Java n'autorise que le style objet, ce qui le rend beaucoup plus simple.

7.1.1.4 Problème d'efficacité

Le code intermédiaire généré par le compilateur Javac doit être interprété. L'interprétation est par définition plus lente que l'exécution directe par un processeur. Ceci a fait de Java un langage moins efficace par rapport à d'autres langages de programmation tel que C++. Plusieurs travaux sont actuellement en cours pour palier à ce problème [Chau 97].

7.1.2 Librairie des classes de Java

Dans sa librairie, Java dispose d'un grand nombre de classes d'objets. Elles sont regroupées en paquetages ("packages"). Dans ce qui suit, nous mentionnons brièvement les paquetages que nous avons utilisés pour réaliser notre projet:

Java.lang: contient les classes permettant de représenter les types primitifs tels que entier, réel, booléen, caractère et chaîne de caractères.

Java.io: permet d'effectuer plusieurs types d'opérations sur les entrées et les sorties, y compris la gestion des fichiers.

Java.util: fournit, entre autres, la classe vecteur (Vector). Cette classe offre une structure de données et des méthodes qui permettent l'insertion, la suppression, l'ajout d'éléments (objets quelconques) dans un vecteur. Elle offre également des méthodes qui indiquent le nombre d'éléments dans le vecteur et quel élément se trouve dans une position donnée.

Java.awt: c'est un outil d'aide à la réalisation des interfaces graphiques. Les classes contenues dans ce paquetage peuvent être regroupées en trois catégories:

- Graphiques: ces classes définissent les polices, les couleurs, les images, les polygones, lignes, textes, etc.
- Composantes: ce sont les interfaces graphiques d'utilisateur ("GUI") tels que les boutons, les menus, les listes, les boîtes de dialogues, etc.
- Gestionnaires de disposition : ces classes permettent le contrôle de la disposition des composantes à l'intérieur d'autres qui les contiennent.

Chaque paquetage peut être importé à l'aide de l'instruction "import java.<package>". On peut également utiliser l'instruction "import java.*" pour importer tous les paquetages de Java.

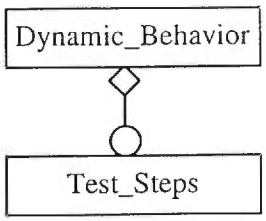
7.2 Représentation interne d'une suite de tests

7.2.1 Représentation graphique des structures de données :

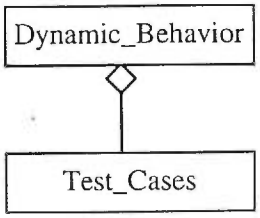
La représentation des connaissances est effectuée à l'aide du formalisme objet avec des classes représentant des types et des liens entre ces classes. Une telle représentation a été choisie en raison de sa simplicité et parce que Java est un langage orienté objet.

Nous avons utilisé la notation OMT ("Object Modeling Techniques") [Rumb 91]. Avec cette notation, chaque classe d'objets est représentée par un rectangle portant son nom. Les différents types de liens entre les classes sont:

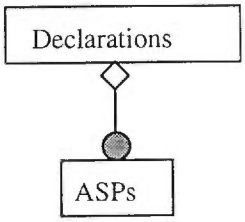
Composition au plus un: La présence du symbole "  " entre une classe A et une classe B, signifie qu'un objet la classe A est composée de zéro ou un seul objet de la classe B. Par exemple, un objet de la classe **Dynamic_Behavior** est composé de zéro ou d'un objet de la classe **Test_Steps** (car une suite de tests peut ne pas avoir d'étapes de tests).



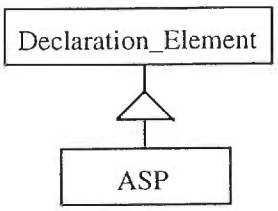
Composition simple: La présence du symbole "  " entre une classe A et une classe B, signifie qu'un objet de la classe A est composé d'un seul objet de la classe B. Par exemple, un objet de la classe **suite_de_test** est composé d'un objet de la classe **Declarations** et d'un objet de la classe **Dynamic_Behavior** (car une suite de tests doit avoir au moins un cas de test).



Composition multiple: La présence du symbole "  " entre une classe A et une classe B, signifie qu'un objet de la classe A est composé de zéro ou plusieurs objets de la classe B. Par exemple, un objet de la classe **Declaration** est composé de zéro ou plusieurs objets de la classe **ASP**.



Lien d'héritage: La présence du symbole "  " entre une classe A et une classe B, signifie que la classe B hérite de la classe A. Par exemple, la classe **ASP** hérite de la classe **Declaration_Element**.



7.2.2 Représentation d'une suite de tests

Pour plus de clarté, nous avons représenté les classes représentant une suite de tests sur deux diagrammes, un diagramme contenant les classes avec les liens de composition, appelé structure d'agrégation (Figure 7-2), et un diagramme contenant les classes avec les liens d'héritage appelé définition des types (Figure 7-3).

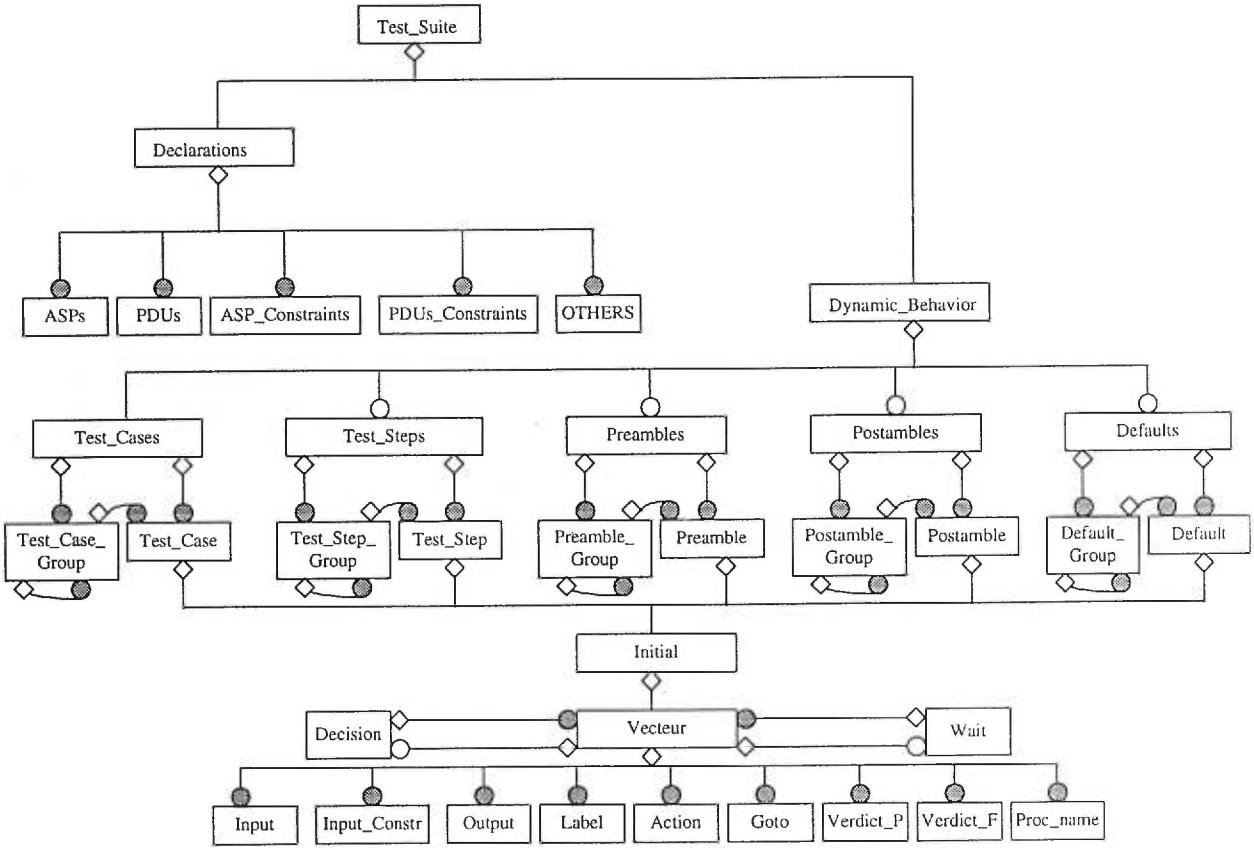


Figure 7-2: Structure d'agrégation

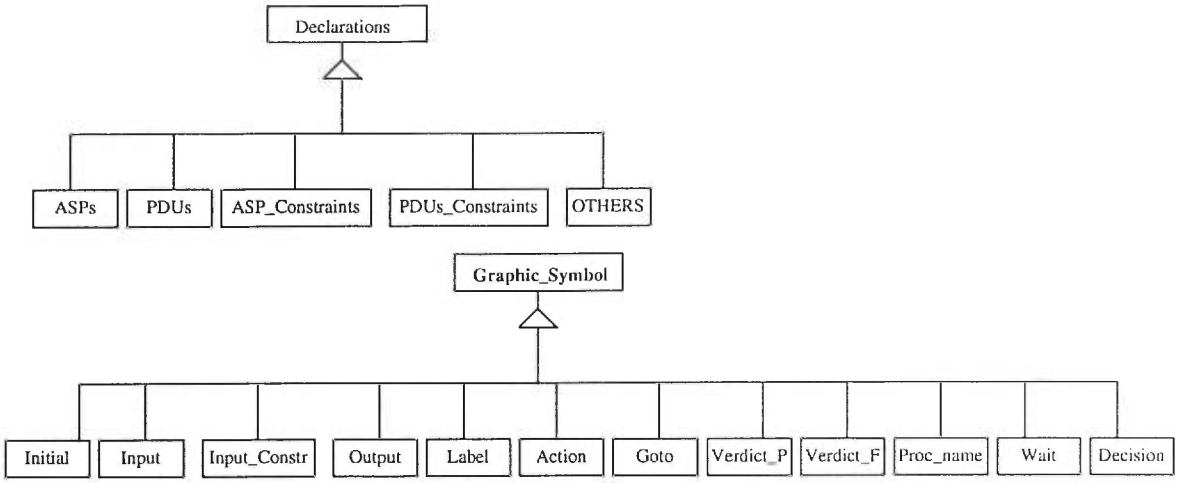


Figure 7-3: Définition des types

Comme nous l'avons représenté dans la figure 7-2, un objet de la classe suite de tests est composé d'un objet de la classe Declarations et d'un objet de la classe Dynamic_Behavior. La classe Declarations est composée de toutes les classes qui représentent les éléments de déclarations. Toutes ces classes héritent de la classe Declarations (Figure 7-3). Chacune de ces classes possède une variable identificateur et une variable texte. La première est considérée comme un identificateur de l'élément de déclarations, et la deuxième contient sa définition. Notons que la variable identificateur est également considérée comme le nom du fichier contenant la définition de l'élément de déclarations concerné.

Concernant les éléments de la classe Dynamic_Behavior, nous avons considéré que la classe Test_Case est composée d'un seul objet de type Initial (c'est la classe du symbole état initial). Ceci s'applique également pour les classes Test_step, Preamble, Postamble et Default. La classe Initial comprend un attribut de type vecteur. Les éléments de ce vecteur sont les symboles graphiques qui succèdent à l'état initial. Lorsque l'un des deux symboles Wait ou Decision est rencontré, il sera le dernier élément dans le vecteur. La classe Wait comprend un attribut de type vecteur de vecteurs. Ces derniers possèdent la même structure que le vecteur de l'état initial. De même la classe Decision possède deux vecteurs qui correspondent respectivement aux deux branches True et False de la decision.

Dans l'annexe nous donnons la définition de toutes ces classes.

Les classes d'objets graphiques ('Initial', 'Input', 'Output', etc.) ont quasiment la même définition. Elles diffèrent essentiellement par la méthode "Draw" qui permet de dessiner le symbole qui leur est associé. Elles héritent toutes de la classe "Graphic_Symbol" (Voir Figure 7-3). Pour cela, seule la définition de cette classe a été décrite dans ce document.

Dans la figure suivante (Figure 7-4), nous reprenons le cas de test décrit dans l'exemple de la figure 6-4 et nous représentons sa structure selon la méthode OMT. Cette représentation est appelée modèle d'objets [Rumb 91]. Chaque instance est représentée à l'intérieur d'un rectangle avec le nom de la classe à laquelle elle appartient entre parenthèses. Seuls les liens de composition simple sont représentés.

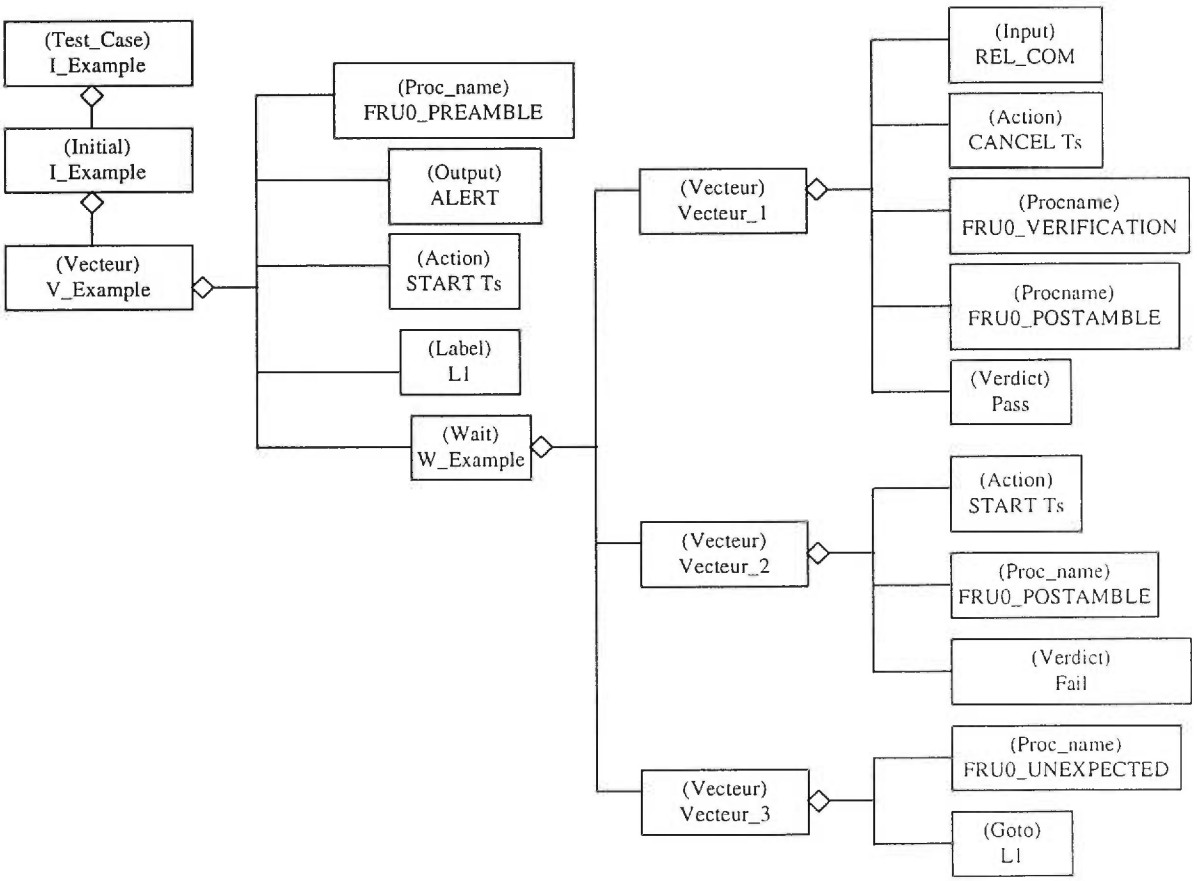


Figure 7-4: Structure d'un cas de test

7.3 Éditeur de suites de tests

Dans l'implantation de la partie déclaration, nous avons surtout utilisé les classes de *java.awt*, et plus précisément les composantes (boîtes de dialogue, boutons, champs de saisie, listes) et les gestionnaires de disposition pour bien disposer ces dernières. Ces classes ont été également utilisées pour l'implantation de la fenêtre de navigation des comportements dynamiques (Figure 6-7).

Nous avons également utilisé *Java.io* pour la création des fichiers et des répertoires, ainsi que pour la lecture et l'écriture des différents éléments de la partie déclaration.

Jean-Walter Guillery, étudiant au baccalauréat à l'Université de Montréal a contribué à l'implantation de la partie édition graphique du comportement dynamique (Figure 6-10). Cette implantation a été réalisée selon les étapes suivantes:

- implantation des éléments de base de la fenêtre d'édition graphique tels que fenêtre, menu, bouton, région (canevas), "scroll-bar", champ de saisie.

- implantation des classes de symboles graphiques définies dans la représentation interne (Action, input, output, condition, etc.). Pour cela, nous avons utilisé les classes graphiques de *Java.awt* telles que ligne, rectangle, polygone, cercle, texte, etc.

- implantation des méthodes qui permettent la manipulation des symboles graphiques (sélection, ajout, suppression, etc.). Pour cela, nous avons utilisé la classe vecteur de *Java.util*. Les possibilités offertes par cette classe ont été décrites dans la section 7.1.2.

7.4 Traducteur vers TTCN-MP

Ce traducteur génère des suites de tests sous le format TTCN-MP à partir de la représentation interne. Dans cette partie, nous avons considéré deux phases: la traduction de la partie des déclarations et la traduction de la partie du comportement dynamique.

Concernant les déclarations, la tâche de génération du code TTCN-MP est simple puisque le texte de la représentation est déjà essentiellement en format TTCN-MP. Afin de respecter la grammaire TTCN-MP, les délimitations de blocs sont ajoutés. Par exemple, "$Begin_SimpleTypeDefs" et "$End_SimpleTypeDefs" sont ajoutés au début et à la fin des déclarations des types simples.

Concernant la génération du comportement en TTCN-MP, il s'agit de parcourir tous les groupes et les cas de tests et à générer le code TTCN-MP correspondant. Pour chaque comportement, le vecteur contenant les symboles est parcouru et le traducteur génère les instructions TTCN-MP qui leur correspondent. Les indentations sont attribuées séquentiellement et commencent par zéro. Pour plus de détails, le pseudo-algorithme suivant explique la traduction vers TTCN-MP de la partie du comportement dynamique:

```
DEBUT
        Indentation = 0;
        Appel Traducteur (Vecteur de l'état initial, Indentation);
FIN

Traducteur (V,i)
DEBUT
        Pour tous les éléments de V
        Faire
                Générer l'instruction TTCN qui correspond au symbole rencontré;
                Attribuer l'indentation i;
                Incrémenter i;
                /* Traitement de Wait */
                Si le symbole rencontré est Wait
                Alors
                        Pour chacun des vecteurs de Wait
                        Faire
                                Appel Traducteur (Vecteur, i)       /*Appel récursif*/
                        Fin Faire
                Fin Si
                /* Traitement de Decision */
                Si le symbole rencontré est Decision
                Alors
                        Appel Traducteur (Vecteur_True, i);
                        Appel Traducteur (Vecteur_False, i);
                Fin Si
        Fin Faire
        FIN
```

Dans l'exemple suivant, nous présentons une suite de tests éditée par GETS et sa traduction en TTCN-MP. Nous insistons surtout sur la partie comportement dynamique vu que les éléments de la partie déclaration sont générés tels qu'ils ont été édités. Dans cet exemple, nous supposons que la suite de tests comprend le cas de test suivant:

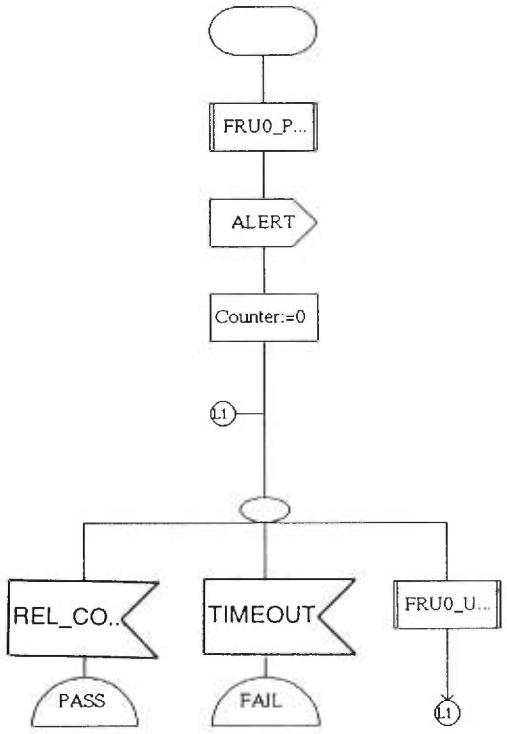


Figure 7-5: Exemple de représentation graphique d'un cas de test

Le code généré par le traducteur est le suivant:

$Suite	$End_ConstraintsPart	$BehaviourLine
$SuiteId Exemple	$DynamicPart	$LabelId L1
$SuiteOverviewPart	$TestCases	$Line [3]?REL_COM (4)
$Begin_SuiteStructure	$Begin_TestCase	$CRef
$SuiteId Exemple	$TestCaseId Test_Case1	$VerdictId PASS
$StandardsRef /* free text	$TestGroupRef	$Comment
*/	FR_SVC.st/Dynamic/Test_Cases/	$End_BehaviourLine
$PICSref /*free text	GENERAL	$BehaviourLine
*/	$TestPurpose /* */	$LabelId
$PIXITref /*freetext	$DefaultsRef	$Line [3]?TIMEOUT (5)
*/	$Comment	$CRef
$TestMethods /*freetext*/	$BehaviourDescription	$VerdictId FAIL
$Comment	$BehaviourLine	$Comment
$Structure&Objectives	$LabelId	$End_BehaviourLine
$End_Structure&Objectives	$Line [0]+ FRU0_PREAMBLE (1)	$BehaviourLine
$End_SuiteStructure	$CRef	$LabelId
$Begin_TestCaseIndex	$VerdictId	$Line[3]+FRU0_UNEXPECTED (6)
$CaseIndex	$Comment	$CRef
$TestGroupRef	$End_BehaviourLine	$VerdictId
Exemple/Dynamic/	$BehaviourLine	$Comment
$TestCaseId T_Case1	$LabelId	$End_BehaviourLine
$SelectExprId	$Line [1]!ALERT (2)	$BehaviourLine
$Description /*FreeText*/	$CRef	$LabelId
$End_CaseIndex	$VerdictId	$Line [4]GOTO L1 (7)
$End_TestCaseIndex	$Comment	$CRef
$Begin_TestStepIndex	$End_BehaviourLine	$VerdictId
$End_TestStepIndex	$BehaviourLine	$Comment
$Begin_DefaultIndex	$LabelId	$End_BehaviourLine
$End_DefaultIndex	$Line [2]Counter:=0 (3)	$End_BehaviourDescription
$End_SuiteOverviewPart	$CRef	$Comment
$DeclarationsPart	$VerdictId	$End_TestCase
$End_DeclarationsPart $ConstraintsPart	$Comment	$End_TestCases
	$End_BehaviourLine	$End_DynamicPart $End_Suite

Après la génération du code TTCN-MP, les suites de tests peuvent être importées par l'outil Workbench [TTCN 94], éditeur pour les suites de tests TTCN. Workbench nous permet essentiellement de valider le code TTCN-MP généré et de l'imprimer dans le format de tableaux caractéristique de TTCN.

En faisant importer le code TTCN-MP de l'exemple précédent par workbench, nous obtenons la table suivante:

Main Tables Tools Display Structure Block

Test Case Dynamic Behaviour

Test Case Name:	Test_Case1
Group:	Exemple/
Purpose:	
Default:	
Comments:	

Label	Behaviour Description	Constraints Ref	V	Comments
	+ FRUO_PRE (1)			
	!ALERT (2)			
	Counter:=0 (3)			
L1	?REL_COM (4)		PAS	
	?TIMEOUT (5)		FAI	
	+ FRUO_UN (6)			
	GOTO L1 (7)			

Detailed Comments:

Table 7-1: Représentation par Workbench de l'exemple de la figure 7-5

Nous avons attribué des numéros aux instructions TTCN obtenues dans cette table et leur correspondants dans le code TTCN-MP généré. Ces numéros sont à droite de chaque instruction et entre parenthèses. Nous pouvons ainsi constater la correspondance entre ces instructions et les symboles de la figure 7-5.

7.5 Traducteur vers SDL-PR

Dans cette partie, le traducteur génère du SDL-PR à partir de la représentation interne. Seule la partie commune à SDL et à TTCN du comportement dynamique (voir chapitre 5)

est concernée par cette traduction. Le système SDL généré contient un seul bloc qui, lui aussi, contient un seul processus. C'est ce processus qui représente la suite de tests. Les cas de tests sont représentés par des procédures qui peuvent être appelées par ce processus.

De même que pour la traduction vers TTCN-MP, le traducteur génère une instruction SDL-PR pour chaque symbole de cas de test.

En appliquant ce traducteur à la même suite de tests de l'exemple de la section précédente, nous obtenons le code SDL-PR suivant:

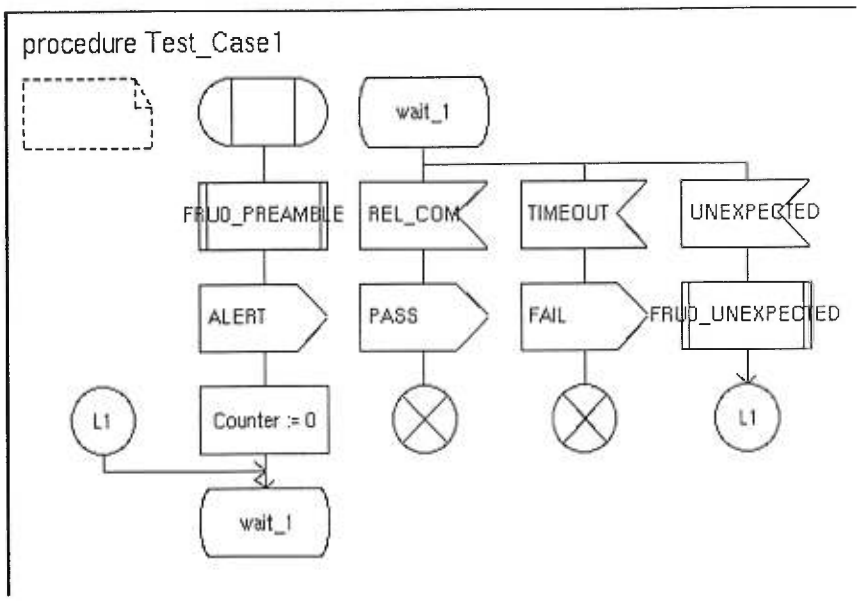
```
system Exemple_system;
signal Pass, Fail, Inconclusive;
block Exemple_block referenced;
endsystem Exemple_system;

block Exemple_block;
signalroute verdict from Exemple_process to env with Pass, Fail, Inconclusive;
process Exemple_process referenced;
endblock Exemple_block;

process Exemple_process;
procedure Test_Case1 referenced;
start;
call Test_Case1;
stop;
endprocess Exemple_process;


Procedure Test_Case1;
start;
call  FRU0_PREAMBLE;
output ALERT;
task Counter:=0 ;
L1 :
nextstate wait_1;
state wait_1;
input REL_COM ;
output PASS ;
return;
input TIMEOUT ;
output FAIL ;
return;
call FRU0_UNEXPECTED;
join L1 ;
return;
endprocedure Test_Case1;
```

Nous remarquons que le premier constructeur de la troisième alternative de cet exemple est un appel de procédure (FRU0_UNEXPECTED). Nous rappelons aussi qu'à la fin de la section 5.2, nous avons mentionné que dans SDL, le premier constructeur d'une alternative ne peut être qu'une entrée (Input). Le code SDL-PR tel qu'il a été généré par le traducteur est alors incorrect. Ceci nous a amené à changer dans ce code. Nous avons ajouté une entrée (Input UNEXPECTED) juste avant l'appel de procédure (call FRU0_UNEXPECTED). Ensuite, nous l'avons importé par l'outil SDT et nous avons obtenu le graphique SDL suivant:



Conclusion

Le test de conformité est une étape très importante dans le cycle de développement d'un protocole. Elle permet de s'assurer de la fiabilité et de la conformité du protocole par rapport à sa spécification de référence.

Les suites de tests de protocoles de communication tendent de plus en plus à être spécifiées en TTCN, langage de description standard de suites de tests. Cependant, ce langage offre un format non convivial et peu désiré de la part de ses utilisateurs.

Le but du présent travail est de développer un environnement convivial pour l'édition de suites de tests. Pour concevoir un tel environnement, nous nous sommes inspirés de la représentation graphique du langage SDL, langage de spécification et de description des systèmes. Nous avons ainsi utilisé quelques symboles de ce langage.

Malgré sa convivialité, le langage SDL ne possède pas certains constructeurs dont nous avons besoin pour réaliser notre projet; ces constructeurs sont spécifiques aux tests. Cela nous a amenés à concevoir d'autres symboles graphiques pour compléter le sous-ensemble inspiré du langage SDL.

L'éditeur GETS ainsi réalisé, utilise ces symboles pour l'édition du comportement dynamique de suites de tests.

GETS comprend un traducteur qui permet la génération du code TTCN-MP à partir des suites de tests déjà éditées. Le code ainsi généré peut être importé et validé à l'aide d'autres outils d'édition de suites de tests tel que WORKBENCH.

De plus, GETS comprend un traducteur qui permet la génération du code SDL-PR à partir de ce qui a été édité. Le code généré peut être importé par des outils supportant SDL tel que SDT. Il pourra ainsi bénéficier des possibilités offertes par ces outils telle que la validation.

L'éditeur GETS permettra même à des utilisateurs qui ne sont pas experts en TTCN d'éditer des suites de tests dans un environnement convivial.

La réalisation de l'outil GETS nous a été très bénéfique. En effet, elle nous a permis d'approfondir nos connaissances des langages SDL et TTCN et d'acquérir une bonne expérience de programmation en Java, langage utilisé pour la réalisation de cet outil.

L'étude que nous avons menée sur les langages TTCN et SDL, nous a montré que le passage du premier au deuxième n'était pas toujours évident à cause des constructeurs spécifiques aux tests qui n'existent pas dans SDL. Pour cela, nous suggérons une extension du langage SDL en lui ajoutant ces constructeurs. Nous obtiendrons ainsi un langage très convivial permettant de palier aux insuffisances du langage TTCN.

Enfin, nous suggérons de compléter l'outil GETS par une édition plus conviviale de la partie déclaration de suites de tests.

Références

[Amou 94] M. C. Amoussou, "Étude de traduction entre les langages de spécification SDL et VHDL", Mémoire de maîtrise, Université de Montréal, 1993.

[Beli 89] F. Belina et D. Hogrefe, "*The CCITT-Specification and Description Language SDL*", Computer Networks and ISDN Systems, Vol.16 (4), 1989, pp.311-341.

[Bell 89] O.B Bellal, "Analyse automatique de résultats de tests appliquée aux protocoles de communication", Mémoire de maîtrise, Université de Montréal, 1989.

[Boch 87] G.V. Bochmann, "Usage of Protocol Development Tools: The Result of a Survey ", Protocol Specification, Testing and Verification, VII, H. Rudin and C.H. West (Editors), North-Holland, 1987, pp. 139-161.

[Boch 89a] G.V. Bochman, "Protocol Specification for OSI", Computer Networks and ISDN Systems, 18, 1989.

[Caou93] C. Caouette, "Évaluation du flux de données couvert par une suite de tests", Mémoire de maîtrise, Université de Montréal, 1993.

[Chau 97] S. Chaumette et A. Miniussi, "Présentation du projet JEM: Java Experimentation environMent", NOTERE'97 Article #14.

[Corn 96] G. Cornell et C. Horstmann, "Core Java", Prentice Hall, 1996.

[Djer93] Y. Djerbib, "Évaluation de la couverture de tests écrits en TTCN et expérience avec le protocole LAPD", Mémoire de maîtrise, Institut national de formation en informatique, Algérie, 1993.

[Dsso 86] R. Dssouli, "Étude des méthod#es de tests de l'implantations de protocoles de Communication Basées sur les Spécifications Formelles", thèse de doctorat, Université de Montréal, 1986.

[Flan 96] D. Flanagan, "Java in a Nutshell", O'Reilly and Associates, 1996.

[Geod 95] Geode, Version 1.1.0, VERILOG 1995.

[Gold 83] A. J. Goldberg et D. Robson, "Smaltalk-80, the Language and its Implementation" Addison Wesley, 1983.

[ISO 9646] ISO-Information Processing Systems - Open Systems Interconnection- OSI Conformance Testing Methodology and Framework, Parts 1, 2, 3, 4, 5, ISO 9646.

[ITU Z.100] ITU-T, "Specification and Description Language (SDL) Recommendation", Z.100, 1993.

[Jard 88] C. Jard, "Valider les protocoles en simulant ", CFIP'88 - Ingénierie des protocoles, R. Castanet et O. Rafiq (Editeurs), Eyrolles, Paris, France, 1988, pp.281-292.

[Java] http://www.iro.umontreal.ca/java/

[Kras 88] G. E. Krasner et S. T. Pope, "A Cookbook for Using the Model-view-Controller User Interface, Paradigm in Smalltalk-80", ParcPlace Systems, 1988.

[Meye 88] B. Meyer, "Object-Oriented Software Construction", Prentice Hall, 1988.

[Rafi 83] O. RAFIQ, "A Protocol Validator and its Applications", Protocol Specification, Testing and Verification, III, H. Rudin and C.H. West (Editors), North-Holland, 1983, pp. 189-197.

[Rafi 91] O. RAFIQ, "Le test de conformité des protocoles", Réseaux et informatique répartie, Vol. 1, N° 1, 1991.

[Rayn 87] D.Rayner, "OSI Conformance Testing", Computer Networks and ISDN Systems, Vol.14, 1987, pp.79-98.

[Ritc 95] T. Ritchey, "Java!", New Riders, 1995.

[Rumb 91] J. Rumbaugh, "Object-Oriented Modeling and Design", Prentice Hall, 1991.

[Salv 93] P. Salvail, "Développement d'un environnement de test et de diagnostique", Mémoire de maîtrise, Université de Montréal, 1993.

[Sara 87] R. Saraco et P.A.J. Tilanus, "CCITT SDL: Overview of the Language and its Applications", Computer Networks and ISDN Systems, 13, pp.65-74, 1987.

[Sari 89] B. Sarikaya, "Conformance Testing Architectures and Tests Sequences", Computer Networks and ISDN Systems, 17 (1989), pp.111-126.

[Sari 92] B. Sarikaya and A.Wiles, "Standard Conformance Test Specification Language TTCN", Computer Standards & Interfaces 14 (1992), pp.117-144.

[SDT 96] SDT, Version 3.1, Telelogic AB, 1996.

[Tann 90] A. Tannenbaum, "Réseaux, Architectures, Protocoles, Applications", Interéditions, Paris, 1990.

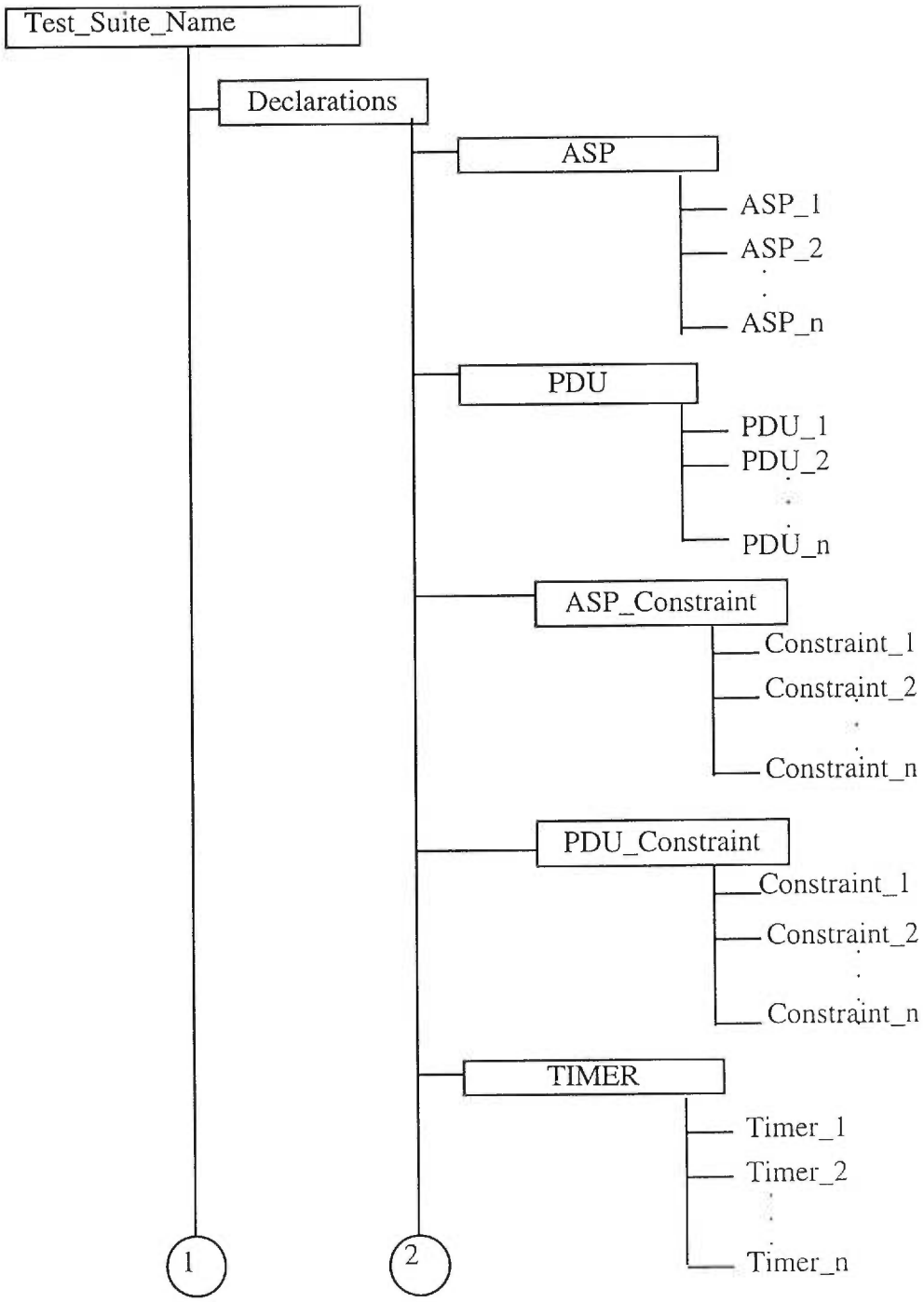
[Thib 94] A. Thiboutôt, "Traduction d'un sous-ensemble de SDL en ESTELLE", Mémoire de maîtrise, Université de Montréal, 1994.

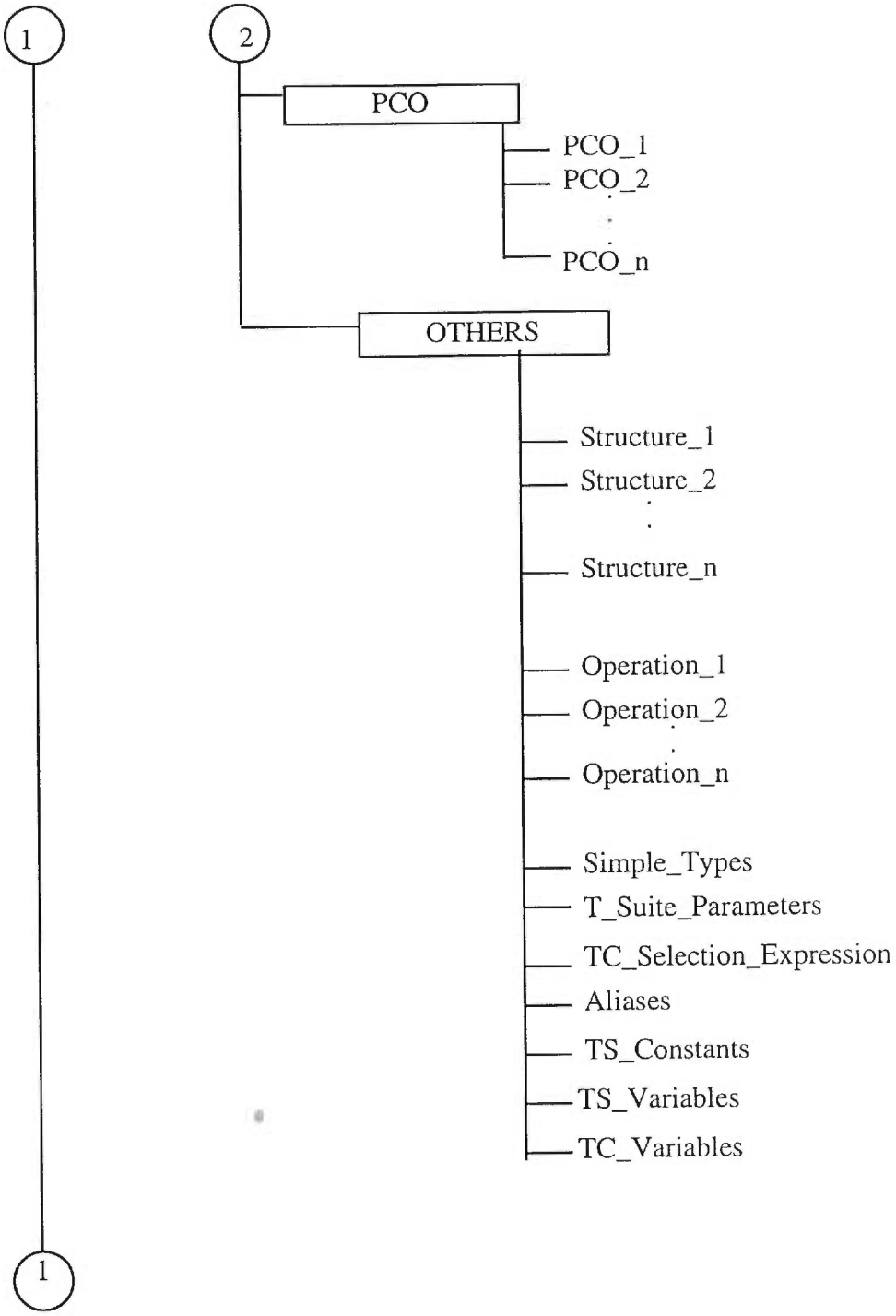
[TTCN 94] TTCN Workbench, Version 3.0, 1994.

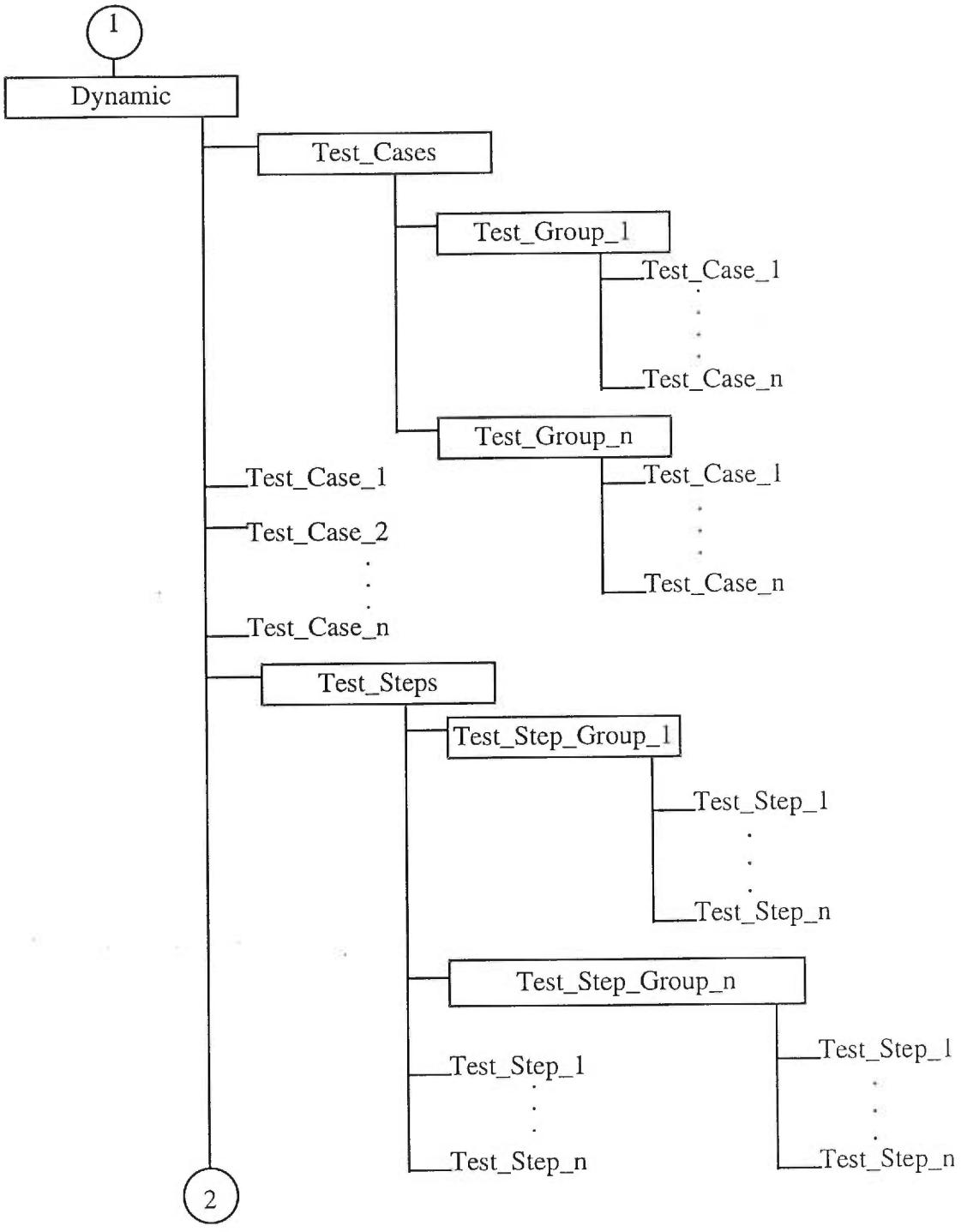
[Zafi 80] P. Zafiropulo, "Towards Analyzing and Synthesizing Protocols ", IEEE Transactions on Communications, Vol.28 (4), 1980, pp.651-660.

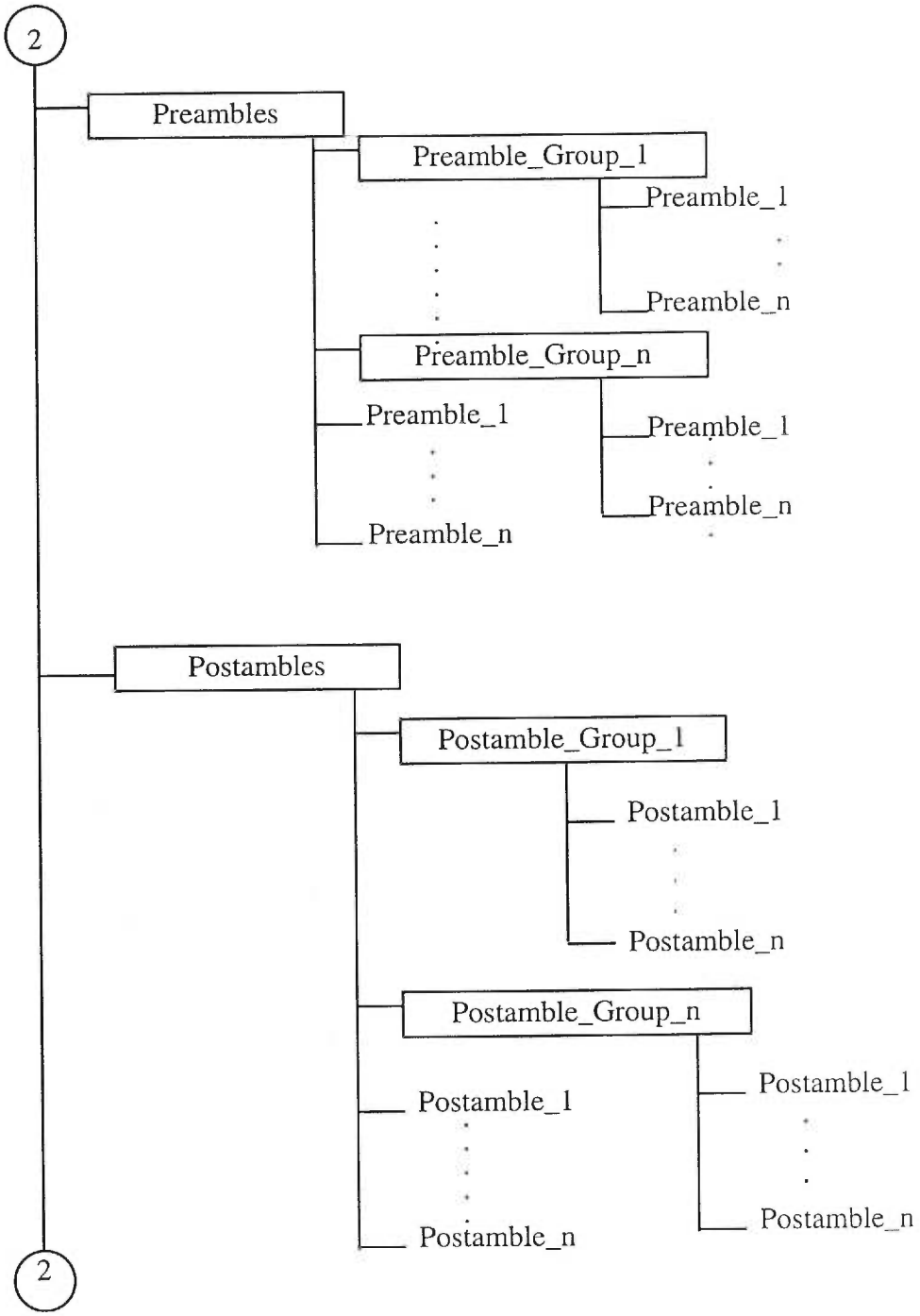
Annexe

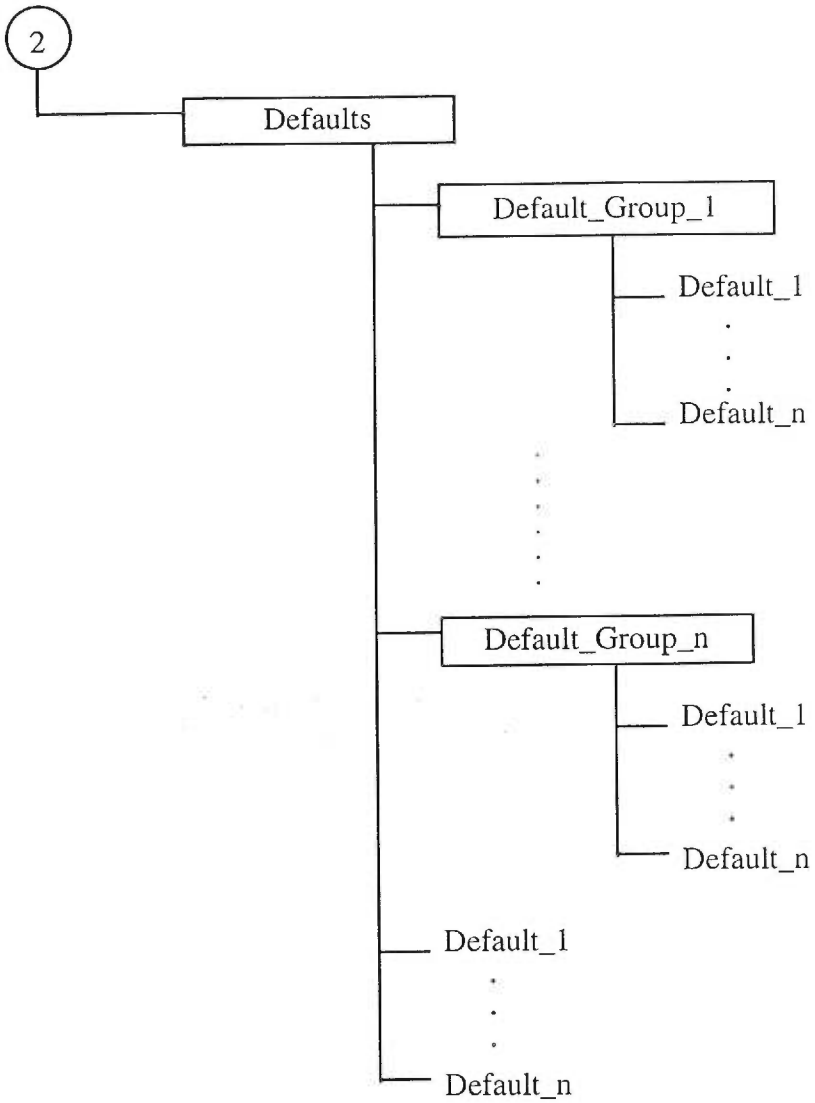
A.1. Organisation des fichiers et répertoires représentant une suite de tests




1
2
PCO
PCO_1
PCO_2
PCO_n
OTHERS
Structure_1
Structure_2
Structure_n
Operation_1
Operation_2
Operation_n
Simple_Types
T_Suite_Parameters
TC_Selection_Expression
Aliases
TS_Constants
TS_Variables
TC_Variables
1


1
Dynamic
Test_Cases
Test_Group_1
Test_Case_1
Test_Case_n
Test_Group_n
Test_Case_1
Test_Case_n
Test_Case_1
Test_Case_2
Test_Case_n
Test_Steps
Test_Step_Group_1
Test_Step_1
Test_Step_n
Test_Step_Group_n
Test_Step_1
Test_Step_n
Test_Step_1
Test_Step_n
2


2
Preambles
Preamble_Group_1
Preamble_1
Preamble_n
Preamble_Group_n
Preamble_1
Preamble_n
Preamble_1
Preamble_n
Postambles
Postamble_Group_1
Postamble_1
Postamble_n
Postamble_Group_n
Postamble_1
Postamble_n
Postamble_1
Postamble_n
2



Legend

: Directory

: File

A.2. Définition des Classes d'objets :

Classe : Test_Suite

Attribut : STRING Name

STRING Free_Text /* Standards Ref, PICS Ref, PIXIT Ref, Test Methods, Comment */

DECLARATIONS Declaration

DYNAMIC_BEHAVIOR Dynamic

Methodes : Make_Directories

Open_Directory

Save

Import

Export

Exit

Classe : Declarations

Attribut : ASP[] ASP_List

PDU[] PDU_List

ASP_Constraint[] ASP_C

PDU_Constraint[] PDU_C

TIMER[] Timer

PCO[] Pco

OTHER[] Other

Methodes : Open_Directory

Create_File(Parameter)

Display_Definition(Parameter)

Display_List(Parameter) /*Parameter = ASP or PDU or ASP_Constraint or PDU_Constraint or TIMER or PCO or OTHER */

Find

Close

Paste_Declaration_Into_Symbol

Classe : ASP extends Declaration_Elt

Attribut : STRING Type='ASP'

Classe : PDU extends Declaration_Elt

Attribut : STRING Type='PDU'

Classe : ASP_Constraint extends Declaration_Elt

Attribut : STRING Type='ASP_Constraint'

Classe : PDU_Constraint extends Declaration_Elt

Attribut : STRING Type='PDU_Constraint'

Classe : TIMER extends Declaration_Elt

Attribut : STRING Type='TIMER'

Classe : PCO extends Declaration_Elt

Attribut : STRING Type='PCO'

Classe : OTHER extends Declaration_Elt

Attribut : STRING Type='OTHER'

Classe : Declaration_Elt

Attribut : STRING Type

STRING Name

STRING Free_Text /* Free_Text = Defenition of the element */

Methodes : Save_Elt (Parameter) /* Save definition of the element */

Close (Parameter)

Classe : Dynamic_Behavior

Attribut : Test_Cases T_Case

Test_Steps T_Step

Preambles Preamble

Postambles Postamble

Defaults Default

Methodes : Open_Directory

Display_Test(Parameter) /* Parameter = Test_Cases or Test_Steps or Preambles or Postambles or Defaults */

Classe : Preambles

Attribut : Preamble_Group[] Preamble_G

Preamble[] preamble

Methodes : Create_Preamble_Group(Parameter)

Open_Preamble_Group(Parameter) /* Parameter = Name of group */

Create_Preamble(Parameter)

Display_Preamble(Parameter) /* Parameter = Name of Preamble */

/* Graphical Display */

Find(Parameter) /* Parameter = Name of Preamble group or Preamble */

Close

Move (Parameter1, Parameter2) /* Parameter1 = Preamble or Preamble group to be moved , Parameter2 = Group destinator */

Classe : Postambles

Attribut : Postamble_Group[] Postamble_G

Postamble[] postamble

Methodes : Create_Postamble_Group(Parameter)

Open_Postamble_Group(Parameter) /* Parameter = Name of group */

Create_Postamble(Parameter)

Display_Postamble(Parameter) /* Parameter = Name of postamble */

/* Graphical Display */

Find(Parameter) /* Parameter = Name of postamble group or postamble */

Close

Move (Parameter1, Parameter2) /* Parameter1=Postamble or Postamble group to be moved , Parameter2 = Group destinator */

Classe : Defaults

Attribut : Default_Group[] Default_G

Default[] default

Methodes : Create_Default_Group(Parameter)

Open_Default_Group(Parameter) /* Parameter = Name of group */

Create_Default(Parameter)

Display_Default(Parameter) /* Parameter = Name of Default */

/* Graphical Display */

Find(Parameter) /* Parameter = Name of Default group or Default */

Close

Move (Parameter1, Parameter2) /* Parameter1 = Default or Default group to be moved ,
Parameter2 = Group destinator */

Classe : Test_Case_Group

Attribut : STRING Name

Test_Case[] T_Case

Test_Case_Group[] T_Case_G

Classe : Test_Step_Group

Attribut : STRING Name

Test_Step[] T_Step

Test_Step_Group[] T_Step_G

Classe : Preamble_Group

Attribut : STRING Name

Preamble[] Preamble_G

Preamble_Group[] Preamble_G

Classe : Postamble_Group

Attribut : STRING Name

Postamble[] Postamble_G

Postamble_Group[] Postamble_G

Classe : Default_Group

Attribut : STRING Name

Default[] Default_G

Default_Group[] Default_G

Classe : Test_Case

Attribut : STRING Name

STRING Header

Graphic_Symbol[] Symbol

Methodes : Add_Symbol

Delete_Symbol

Insert_Symbol

Save

Print

Close

Classe : Test_Step

Attribut : STRING Name

STRING Header

Graphic_Symbol[] Symbol

Methodes : Add_Symbol

Delete_Symbol

Insert_Symbol

Save

Print

Close

Classe : Preamble

Attribut : STRING Name

STRING Header

Graphic_Symbol[] Symbol

Methodes : Add_Symbol

Delete_Symbol

Insert_Symbol

Save

Print

Close

Classe : Postamble

Attribut : STRING Name

STRING Header

Graphic_Symbol[] Symbol

Methodes : Add_Symbol

Delete_Symbol

Insert_Symbol

Save

Print

Close

Classe : Default

Attribut : STRING Name

STRING Header

Graphic_Symbol[] Symbol

Methodes : Add_Symbol

Delete_Symbol

Insert_Symbol

Save

Print

Close

Classe : Graphic_Symbol

Attribut : INTEGER Pos_x, Pos_y

String Texte

Methodes : Draw

Select

Deselect

Set_Text(Texte)

Reposition(New_X, New_y)

Clear

Connect

Disconnect