KORRIGAN : un formalisme et une méthode pour la spécification formelle et structurée de systèmes mixtes

Pascal Poizat

UNIVERSITÉ DE NANTES
UFR SCIENCES ET TECHNIQUES

ÉCOLE DOCTORALE

SCIENCES ET TECHNOLOGIES
DE L'INFORMATION ET DES MATÉRIAUX

2001

Thèse de DOCTORAT

Spécialité : INFORMATIQUE

Présentée et soutenue publiquement par

Pascal POIZAT

le 20 décembre 2000

à l'UFR Sciences et Techniques, Université de Nantes

KORRIGAN : un formalisme et une méthode pour la spécification formelle et structurée de systèmes mixtes

Jury

Président	:	Professeur Frédéric BENHAMOU	Université de Nantes
Rapporteurs	:	Maritta HEISEL, Professeur	Université de Magdebourg
		Gianna REGGIO, Professeur	Université de Gênes
Examinateurs	:	Christine CHOPPY, Professeur	Université de Paris 13
		Michel LEMOINE, Ingénieur de Recherche	ONERA Centre de Toulouse
		Jean-Claude ROYER, Maître de Conférences	Université de Nantes

Directeur de thèse : Professeur Christine CHOPPY (LIPN, Université de Paris 13)
Laboratoire : INSTITUT DE RECHERCHE EN INFORMATIQUE DE NANTES.
2, rue de la Houssinière, F-44322 NANTES CEDEX 3

N° ED 0366-010

KORRIGAN : UN FORMALISME ET UNE MÉTHODE POUR LA SPÉCIFICATION FORMELLE ET STRUCTURÉE DE SYSTÈMES MIXTES

KORRIGAN*: a Formalism and a Method for the Structured Formal Specification of Mixed Systems*

Pascal POIZAT

favet neptunus eunti

Université de Nantes

Pascal POIZAT

KORRIGAN ***: un formalisme et une méthode pour la spécification formelle et structurée de systèmes mixtes***

xviii+236 p.

Ce document a été préparé avec LaTeX2$_\varepsilon$ et la classe `these-IRIN` version 0.92 de l'association de jeunes chercheurs en informatique LOGIN, Université de Nantes. La classe `these-IRIN` est disponible à l'adresse :

`http://www.sciences.univ-nantes.fr/info/Login/`

Impression : these.tex – 19/2/2001 – 15:53

Révision pour la classe : $Id: these-IRIN.cls,v 1.3 2000/11/19 18:30:42 fred Exp

La légende raconte qu'un jeune homme, qui occupé avec sa fiancée tardait à rentrer chez lui, décida de passer par la lande pour gagner du temps. Il se vit brusquement entouré par de nombreux lutins chantant :

Lundi, mardi, mercredi,
Et jeudi, et vendredi, et samedi, ...

Ce à quoi, le jeune homme ajouta :

Et samedi, et dimanche,
Voila terminée la semaine.

Très contents de sa prouesse artistique, les korrigans lui firent don d'un sac de feuilles qui se changèrent en autant de pièces d'or une fois qu'il fut rentré chez lui.

Une autre version raconte que le jeune homme était bossu et qu'en cadeau les korrigans lui retirèrent sa bosse.

Une autre, enfin, que mécontents du fait qu'il ait ajouté le dimanche (jour lié à la religion), les korrigans le forcèrent à danser toute la nuit jusqu'à épuisement.

Dilun, dimeurzh, dimerc'her, Ayota, ayota, ayota, ayota, Dilun, dimeurzh, dimerc'her, Ha diriaou, ha digwener. Ha disadorn, ha disul, Ayota, ayota, ayota, ayota Ha disadorn, ha disul, Setu echu ar sizhun.

Lundi, mardi, mercredi, Ayota, ayota, ayota, ayota, Lundi, mardi, mercredi, Et jeudi, et vendredi. Et samedi, et dimanche, Ayota, ayota, ayota, ayota, Et samedi, et dimanche, Voila terminée la semaine.

— folklore traditionnel breton.

Résumé

L'emploi des spécifications formelles est d'une grande importance, tout particulièrement pour le développement de systèmes dits sécuritaires ou critiques. Les spécifications mixtes ont pour but de permettre l'expression des différents aspects que peuvent présenter ces systèmes : statique (types de données), dynamiques (comportements) et composition (architecture, concurrence et communication). La complexité des systèmes de taille réelle rend indispensable la définition de moyens de structuration des spécifications mixtes. Pour cela, nous proposons un modèle basé sur une hiérarchie de structures que nous appelons *vues*, ainsi que KORRIGAN, le langage formel associé. Les vues intègrent des *systèmes de transition symboliques*, des spécifications algébriques et une forme de logique temporelle. Elles permettent la spécification des différents aspects de façon unifiée. Elles sont expressives, lisibles et favorisent la définition de composants à un haut niveau d'abstraction. Notre modèle offre trois moyens de structuration des spécifications. La *structuration interne* permet de définir les aspects de base des composants (dynamique et statique). La *structuration externe* permet de définir de façon unifiée différents types de compositions : tant l'intégration d'aspects que la composition concurrente de composants communicants. Une forme simple de *structuration d'héritage* permet de réutiliser les composants. Nous pensons qu'il est important, pour que les méthodes formelles soient réellement utilisées, qu'elles disposent d'une méthode associée. Dans cette optique, nous proposons une méthode pour la spécification mixte et structurée, applicable aux spécifications KORRIGAN ainsi qu'à d'autres formalismes. Enfin, nous proposons un atelier, ASK, dédié à la spécification mixte en KORRIGAN. Il intègre des mécanismes de vérification des spécifications KORRIGAN par traduction et de génération de code orienté objet.

Mots-clés : spécifications formelles, spécifications mixtes, structuration, sémantique, vues, systèmes de transitions symboliques, méthode, atelier, génération de code orienté objet

Abstract

The use of formal specifications is quite knowledgeable, in particular when developing safety critical systems. The aim of mixed specifications is to allow one to express the different aspects present in these systems, i.e. static (datatypes), dynamic (behaviour), and composition (architecture, concurrency and communication). The complexity of real size applications requires that structuring means for mixed specifications should be defined. This is why we present a model based on a hierarchy of structures that we call *views*, together with KORRIGAN, the associated formal language. Views integrate *symbolic transition systems*, algebraic specifications, and a form of temporal logic. They allow one to specify the different aspects in a unified way. They are expressive, readable, and promote the component definition at a high level of abstraction. Our model comprises three different means for structuring specifications. The basic aspects (static and dynamic) of the components are defined within the *internal structuring*. The different kinds of composition (integration of aspects, concurrent composition of communicating components) are defined in a unified way within the *external structuring*. Components may be reused through a simple form of *inheritance structuring*. To put formal methods into practice, it is important that they should be equipped with an appropriate method. To this end we propose a method for writing mixed and structured specifications that may be used for KORRIGAN but also for other mixed specification formalisms. Finally the ASK toolbox that is dedicated to mixed specification in KORRIGAN is described. ASK comprises verification means for KORRIGAN specifications through translation and object-oriented code generation.

Keywords: formal specifications, mixed specifications, structuring, semantics, views, symbolic transition systems, method, toolbox, object-oriented code generation

remerciements à : Frédéric Benhamou pour avoir accepté de présider mon jury de thèse et pour s'être réellement intéressé à mon travail et avoir suivi sa finalisation • Maritta Heisel pour avoir accepté de rapporter sur ma thèse, pour m'avoir fait découvrir les agendas et pour toutes les discussions à Dagstuhl ou par courrier électronique concernant le contenu de ce mémoire • Gianna Reggio pour avoir accepté de rapporter sur ma thèse et pour ses nombreuses remarques et commentaires • Michel Lemoine pour avoir accepté d'être membre de mon jury de thèse, pour ses remarques et ses encouragements lors des dernières phases de ma thèse, et pour m'avoir montré qu'il est possible de faire cohabiter formel et informel, B et UML • Christine Choppy pour avoir il y a plusieurs années accepté de diriger ma thèse, pour toutes les discussions fructueuses sur mon travail et pour m'avoir appris à gravir la montagne recherche par la face pédagogique • Jean-Claude Royer pour m'avoir supporté comme étudiant de projet de maîtrise et pour avoir persisté en m'initiant à la recherche en DEA. Il était alors naturel que ça se poursuive en tant que co-directeur scientifique de thèse. Merci pour m'avoir appris que les objets aussi peuvent être formels • Michel Allemand et Christian Attiogbé pour leurs conseils avisés et leurs relectures attentives de mon manuscrit • les membres du projet SHE'S pour m'avoir montré qu'il est possible d'animer un projet scientifique dans la bonne humeur • Henri Habrias pour avoir créé une équipe "méthodes et spécifications formelles" à Nantes, ainsi que pour son approche philosophique, parallèle et décalée de la recherche en informatique, et qui induit une touche d'esprit critique là où parfois il en manque cruellement • ma famille qui me supporte depuis longtemps et me supportera encore longtemps je l'espère • Frédéric Goualard et Leslie Lamport sans qui ce manuscrit n'aurait pas eu la même forme • les "petites mains" de l'IRIN, qui sont si peu souvent remerciées, mais sans qui rien ne tournerait : Christine, Jacky et Yann • ceux parmi les chercheurs de l'IRIN qui ont bien voulu, un jour, partager un avis ou discuter sur tel ou tel point concernant la recherche ou la couleur du temps • tous ceux qui m'ont accompagné dans mon apprentissage de l'enseignement, car la thèse c'est aussi cela ; les équipes pédagogiques ; mes tuteurs du monitorat d'enseignement supérieur : Christophe et Michel ; la directrice et les directeurs des départements où j'ai enseigné et qui ont su me faire confiance • le Professeur Ken Turner pour sa disponibilité et ses informations sur SDL et LOTOS • Radu Mateescu pour sa gentillesse et pour le chapitre 1 de sa thèse, la meilleure introduction à la logique temporelle que je n'aie jamais lue • Juan Manuel Murillo pour son travail, sa disponibilité, et pour m'avoir permis de me remettre à l'espagnol • les anciens du DUT : Yann, Jacky, Binôme, José, Richard, Olivier et David, ainsi que tous les "crêpophiles" et partisants de la crêpe "andouille-chèvre-whisky" pour toutes les soirées passées à faire flamber galettes et crêpes ou à vider des pintes au Graslin • le Graslin et ses tenanciers • Zahra et Jean-Claude, ainsi qu'Honorine et Antoine, pour les soirées passées chez eux • Pascal, mon prédécesseur en tant que thésard de Jean-Claude, et donc conseiller d'une certaine façon • Djenabou, avec qui j'ai si longtemps partagé le bureau, pour toutes les discussions sur nos thèses respectives • Élise et Gwen parce que ce sont Elise et Gwen ; courage Gwen, c'est bientôt toi qui sera en train d'écrire des remerciements de thèse • les membres - spécial dédicace à mes estimés collègues fondateurs - de LOGIN : Association des jeunes chercheurs en informatique de Nantes ; c'est quoi l'informatique de Nantes ? • Nantes, le tram', les marbellas en général et Dipsy en particulier • David et la trip-hop • toutes les technologies Poquémone • mon brave PC sans qui rien n'aurait été possible • Wallace et Gromit • la découverte ou l'ignorance •

Sommaire

Introduction

L'emploi de méthodes de spécification formelle est d'une grande importance pour le développement de systèmes logiciels et ceci tout particulièrement pour la conception et la vérification de systèmes dits sécuritaires ou critiques (par exemple ceux impliquant la vie humaine). Le fait que les spécifications formelles permettent une description non ambiguë, précise, complète et indépendante de toute implémentation conduit à une compréhension approfondie du système à développer, et beaucoup d'erreurs sont ainsi évitées. La possibilité de leur appliquer des vérifications et des validations est l'un des points qui concourent à leur utilisation de plus en plus fréquente. Les méthodes formelles permettent aussi de dériver automatiquement des prototypes à partir des spécifications. Ainsi, dans plusieurs secteurs industriels, l'utilisation des spécifications formelles s'étend : SDL dans les télécommunications, les langages synchrones dans l'avionique ou le nucléaire, la méthode B dans le transport pour citer quelques exemples.

Les systèmes présentant un haut niveau de complexité possèdent généralement plusieurs aspects : statique (types de données), dynamiques (communication, comportements), et composition (assemblage de composants[1], patrons, concurrence et synchronisation inter composants). Il est important de prendre en compte l'ensemble de ces aspects lors de la spécification des systèmes, c'est le champ d'application des spécifications mixtes.

Spécifications mixtes

Nous considérons qu'il existe deux grandes approches de spécification mixte : l'approche hétérogène et l'approche homogène.

L'approche hétérogène fait intervenir différents formalismes pour exprimer les différents aspects. Il peut s'agir d'algèbres de processus, de systèmes de transition ou de réseaux de Petri pour les aspects dynamiques, et de formalismes basés modèles (Z, B) ou propriétés (spécifications algébriques, logiques) pour les aspects statiques. Plusieurs combinaisons ont été étudiées. Il s'agit d'une démarche qui dépasse d'ailleurs le cadre de la spécification mixte. De façon générale, les spécifications hétérogènes sont très utiles dans un contexte de réutilisation de spécifications existantes ou d'environnements logiciels dédiés. D'autre part, elles permettent une bonne expressivité puisqu'il est possible de choisir les formalismes les mieux adaptés à tel ou tel aspect. Ce type de spécification fait actuellement l'objet de nombreuses recherches, essentiellement autour du problème majeur consistant à leur intégration et à la définition de critères de cohérence entre formalismes différents. Ces spécifications sont cependant celles qui remportent le plus de succès dans le milieu industriel.

L'approche homogène se base sur le fait de pouvoir exprimer les différents aspects à l'aide d'un unique formalisme. Ceci se fait aux dépens de l'expressivité, un unique formalisme étant en général adapté à un aspect, et pas à tous. Il s'agit souvent de l'extension syntaxique ou méthodologique d'un formalisme adapté à l'un des aspects pour couvrir les autres aspects. S'il est possible d'être très expressif, cela se fait alors aux dépens de la lisibilité des spécifications en ayant recours à des formalismes de très haut niveau (logiques temporelles ou d'ordre supérieur). L'avantage majeur de cette approche reste cependant que les problèmes de cohérence entre aspects ne se posent pas. En effet, on reste dans le

[1] Un composant correspondant à l'unité de base de spécification et de réutilisation.

contexte du formalisme de base où la définition d'une sémantique globale et la vérification se font plus aisément. Ces différents points en font un bon formalisme de vérification (par traduction d'un formalisme hétérogène) plutôt que de spécification. C'est une approche qui reste d'un usage plus académique.

Contexte de la thèse

Dans ce contexte des spécifications mixtes, nous prenons le parti de tenter de combiner un certain nombre de besoins concernant les formalismes de spécification. Notons de prime abord que nous nous plaçons dans le cadre hétérogène puisque nous pensons qu'il est plus adapté (expressivité et lisibilité) à la spécification mixte. Nous tentons cependant aussi de définir un formalisme permettant de spécifier de façon unifiée les différents aspects, ce qui facilitera leur intégration et la définition d'une sémantique globale. De ce point de vue, nous visons aussi à bénéficier de certains des avantages de l'approche homogène.

Nous cherchons à définir un formalisme expressif (tout en le destinant aux systèmes répartis tels que les protocoles et services de communication et en restreignant donc volontairement ainsi son champ d'application) mais en restant le plus lisible possible, et ce en vue de son utilisation hors du contexte universitaire. Pour cela, nous cherchons à recourir à des concepts connus (composants, objets identifiés, encapsulation) ou encore à des formalismes graphiques tels que les systèmes à base d'états et de transitions.

En vue d'éviter l'explosion combinatoire résultant généralement du manque de possibilité d'abstraction proposée par des formalismes tels que les automates et systèmes de transitions habituels, nous proposons d'avoir recours à une forme symbolique de systèmes de transitions.

Le haut niveau de lisibilité désiré passe aussi par la définition de moyens de structuration des spécifications, tels que la définition séparée des aspects (très utile dans un contexte de réutilisation), leur intégration, l'héritage et la composition parallèle de composants.

Nous pensons aussi que formalisme et méthode d'utilisation sont intimement liés. C'est pourquoi il nous paraît important de définir une méthode pour notre formalisme, et plus généralement pour les spécifications mixtes. Enfin, il est important que tous ces aspects soient intégrés dans un environnement de spécification, permettant la vérification ou encore la génération de prototypes orientés objets (on ne peut plus douter de la qualité de ce code).

Pour répondre à cette problématique, nous avons proposé un modèle, et un langage formel associé, permettant la spécification mixte et structurée de systèmes mixtes. Ce modèle se base sur une structure unificatrice appelée *vue*. Les vues sont une représentation à la fois algébrique et à base de systèmes de transitions très abstraits, les *systèmes de transitions symboliques*. Ces deux parties des vues sont très liées et permettent la spécification des différents aspects des systèmes. Nous proposons une hiérarchie de structures de vue pour cela. Un premier niveau de structuration concerne la structuration que nous qualifions d'interne et permettant de spécifier les aspects statiques et dynamiques des composants. Nous définissons aussi une forme d'héritage pour ces structures. Un second niveau de structuration concerne la définition de composants globaux en ce qu'à la fois leurs aspects statiques et dynamiques sont définis. Il peut s'agir de l'intégration de ces deux aspects au sein d'un tout, ou de la composition d'un nombre indéterminé de composants globaux communicants qui auront une exécution concurrente. Nous proposons un mécanisme de colle de composants à base à la fois d'axiomes et de logique temporelle, et permettant la définition de différentes sémantiques de concurrence et de communication. Notre modèle dispose d'une sémantique opérationnelle qui passe par le calcul d'une structure de vue globale pour les

compositions.

Nous proposons aussi une méthode semi-formelle de conception de systèmes mixtes. Cette méthode a été appliquée sur de nombreux cas d'étude (hôpital, nœud de transit dans un réseau de communication, ascenseur, système de messagerie téléphonique, gestion des mots de passe sous Unix). Elle est particulièrement adaptée à notre formalisme à base de systèmes de transitions abstraits puisqu'elle en permet le calcul semi-automatique à partir des besoins, ainsi que l'obtention d'une partie importante des axiomes des parties algébriques des vues. Elle a cependant aussi été appliquée dans le cadre d'autres formalismes de spécification tels que LOTOS ou SDL, en ayant recours à des patrons de traduction.

Enfin, nous avons défini un atelier pour la spécification mixte de systèmes. Cet atelier est implanté dans un cadre orienté objet. Il est ouvert et les moyens d'intégration de nouveaux langages et outils ont été définis. En ce qui concerne notre modèle, l'atelier prend en compte l'ensemble des mécanismes formels utilisés pour la définition d'une sémantique opérationnelle (gestion des formules de logique temporelle, moteur de réécriture conditionnelle, calcul des systèmes de transitions globaux). L'atelier permet aussi la génération d'un prototype du système dans un langage concurrent orienté objet, Active Java.

Plan du document

Le chapitre 1 contient un état de l'art concernant les spécifications mixtes. Deux types d'approches sont étudiées par le biais de leurs représentants principaux. Il s'agit des approches hétérogènes combinant plusieurs formalismes pour exprimer les aspects, et des approches homogènes qui les expriment dans un cadre unique. Des critères concernant les spécifications mixtes sont présentés puis les approches existantes sont décrites suivant ces critères. Cette étude nous a permis de nous situer et de faire des choix qui ont conduit à la définition d'un modèle et d'un formalisme, d'une méthode et d'un environnement dont la présentation fait l'objet des chapitres suivants.

Le chapitre 2 présente notre modèle ainsi que le langage formel correspondant, KORRIGAN [70]. Il s'agit d'un modèle basé sur une hiérarchie de vues permettant de spécifier les systèmes mixtes de façon unifiée, intégrée et structurée. Nous présentons tout d'abord la structure générale de vue, et les systèmes de transitions symboliques sur lesquels elle repose. Une forme simple d'héritage pour les vues est étudiée. Nous donnons ensuite la définition des structures de vue permettant l'expression des différents aspects (statiques et dynamiques) des systèmes. La définition des structures permettant différents types de composition (intégration des aspects et composition parallèle) est ensuite donnée. Enfin, nous donnons une sémantique opérationnelle à notre modèle [69].

Le chapitre 3 propose une méthode de développement de spécifications mixtes [223, 71], applicable entre autres à notre modèle. Cette méthode est appliquée dans le chapitre à une étude de cas de taille réelle, celle d'un nœud de transit dans un réseau de communication. Nous présentons tout d'abord un état de l'art des méthodes et styles de spécification concernant les systèmes répartis et les protocoles et services de communication. Notre méthode, présentant les intérêts des différents styles étudiés, est ensuite présentée en utilisant le concept d'agenda, une notation semi-formelle permettant la description des méthodes associées aux formalismes. Les apports principaux de notre méthode concernent l'obtention semi automatique des automates abstraits des comportements à partir de l'analyse des besoins, l'utilisation à la fois de notations graphiques basées sur UML et de notations textuelles formelles, ainsi que la génération de différentes spécifications par le biais de patrons de traduction. Cette méthode s'insère dans le développement entre une analyse semi-formelle, en UML par exemple, et la génération de code orienté objet (que nous étudions dans le chapitre 4).

Le chapitre 4 présente un environnement dédié à la spécification mixte centré autour de notre mo-

dèle et du langage KORRIGAN. Il s'agit d'un environnement logiciel ouvert permettant l'intégration de langages et d'outils dédiés de spécification mixte dans un cadre orienté objet [72]. Dans ce cadre, nous proposons une librairie de spécifications mixtes permettant la traduction entre spécifications. Nous l'utilisons dans un but de validation et de vérification de nos spécifications, en les traduisant dans divers formalismes cibles. Une sous-librairie plus particulièrement dédiée aux systèmes de transitions est aussi présentée. Elle nous permet de gérer toute la partie de notre formalisme axée sur les systèmes de transitions symboliques. Les règles concernant le calcul de systèmes de transitions globaux et leur sémantique opérationnelle, présentées dans le chapitre 2 y sont implémentées. Enfin, nous présentons les moyens de génération de code orienté objet que nous avons définis pour nos spécifications [224].

L'annexe A présente la syntaxe textuelle et graphique du langage KORRIGAN. Les annexes suivantes présentent les études de cas les plus significatives (taille, réalisme et complexité) que nous avons étudiées et spécifiées à l'aide de notre formalisme. **L'annexe B** présente une étude de cas de gestionnaire de mots de passe. **L'annexe C** présente une étude de cas de messagerie téléphonique. **L'annexe D** présente une étude de cas de nœud de transit dans un réseau de communication.

CHAPITRE 1

Spécifications formelles mixtes

Most important properties of specifications methods are not only the underlying theoretical concepts but more pragmatics issues such as readability, tractability, support for structuring, possibilities of visual aids and machine support.

— M. Broy, [54].

Nous présentons dans ce chapitre le contexte de cette thèse. La spécification de systèmes informatiques présentant un certain niveau de complexité requiert d'une part la possibilité de les spécifier sous leurs différents aspects (spécification mixte), et d'autre part des moyens de structurer les spécifications obtenues. Nous présentons ici les différents aspects et besoins liés à la spécification mixte de systèmes informatiques, les différentes approches existantes et nous montrons en quoi nous pensons qu'elles présentent ou non une réponse à ces besoins.

1.1 Définitions préliminaires

Nous appelons *spécification mixte* une spécification prenant en compte différents aspects des systèmes. Ces aspects sont nombreux. Le modèle orienté objet (OMT [238], UML [249]) distingue généralement trois aspects : statique, fonctionnel et dynamique. Les méthodes structurées (SSADM [245], SA/DT [190]) utilisent aussi trois plans correspondants : données, processus et comportement. Lorsqu'on s'intéresse aux systèmes répartis, plusieurs autres aspects peuvent intervenir : composition concurrente et communication, distribution, mobilité. De façon générale, et ce même si le nombre des aspects peut être important, les formalismes de spécification mixte ne prennent pas en compte l'ensemble de ces aspects. Ceci est lié à la fois au problème de la cohérence entre les différents aspects des systèmes (plus le système a d'aspects, plus il est difficile de vérifier la cohérence de l'ensemble) ainsi qu'à la difficulté de définir une sémantique globale pour le système. Dans notre contexte, nous nous intéressons aux aspects que nous nommons aspects statiques et aspects dynamiques, et que nous définissons ci-après.

Nous appelons *aspect statique* (mais aussi *aspect données*) la spécification (ou la partie de la spécification dans le cadre de spécifications homogènes) correspondant aux types de données (au sens d'un ensemble de types et de propriétés d'opérations sur ces types) d'un composant. Nous appelons *aspect dynamique* (mais aussi *aspect comportemental*) la spécification (ou la partie de la spécification) correspondant aux communications entre composants, au comportement individuel des composants (séquences possibles de communications) et à la concurrence. Enfin, nous appelons *aspect composition* (ou plus simplement *composition*) la définition d'un composite à partir de composants.

Nous appelons *spécification hétérogène* (ou encore spécification multi-formalisme) une spécification faisant intervenir différents formalismes de spécification. Une spécification hétérogène est l'un des moyens de réaliser des spécifications mixtes. À chaque aspect est associé une spécification dans un formalisme différent. Il s'agit en général d'un formalisme bien approprié à cet aspect, et ce hors de tout contexte de combinaison d'aspects. Le problème de la cohérence entre aspects se pose alors. Les spécifications hétérogènes fournissent un cadre très général de spécification mais présentent une grande complexité à analyser et font actuellement l'objet de nombreuses études [102, 100, 115].

Nous appelons *spécification mixte homogène* (par abus, spécification homogène) une spécification mixte dans laquelle chacun des aspects est décrit dans un seul et unique formalisme. Il s'agit souvent de solutions ad hoc dans lesquelles un formalisme bien approprié à un aspect donné est étendu pour pouvoir prendre d'autres aspects en compte. Il s'agit en général là de leur limitation, l'extension étant en général peu expressive (du point du vue de l'aspect rajouté) ou de faible lisibilité. D'autre part, la prise en compte de nouveaux aspects est relativement complexe.

Dans ce qui suit, nous présentons les principaux aspects à retenir pour la spécification mixte des systèmes, ainsi que les besoins associés à ces systèmes. Nous faisons ensuite un panorama des méthodes existantes. Par abus de langage, nous utilisons le terme "spécification" tant pour le produit résultant du processus de spécification que pour les méthodes et formalismes associés.

Les besoins associés aux spécifications mixtes sont de deux types : ceux associés aux spécifications en général, et ceux plus particuliers liés au caractère mixte de ces spécifications.

1.2 Besoins associés aux spécifications

Un certain nombre de besoins peuvent être associés aux spécifications. Un formalisme de spécification est toujours un compromis fait entre ces différents besoins en fonction d'un domaine de systèmes à spécifier. Dans notre cas, il s'agit particulièrement des systèmes répartis, embarqués, ou des protocoles et services de communication.

1.2.1 Formalité

Il s'agit peut être avant tout du principal besoin puisque nous traitons de spécifications formelles. La *formalité* est le fait qu'une sémantique formelle soit clairement et proprement définie pour le formalisme. Il s'agit de différents types de sémantiques [267]. Elle pourra être opérationnelle (comme les systèmes à base d'états et de transitions), dénotationnelle ou axiomatique. Elle peut d'autre part être exécutable [120, 143] (comme les systèmes à base de réécriture [127, 269, 201]) et ainsi autoriser une animation des spécifications.

Les propriétés de non ambiguïté et de précision sont inhérentes à l'aspect formel de la spécification. Elles permettent aussi concrètement de pouvoir effectuer (de façon cohérente) des tests, validations et vérifications des spécifications (c'est-à-dire que la spécification vérifie une ou plusieurs propriétés). Certaines notations, pourtant couramment utilisées (tel qu'UML [249]) ne présentent qu'un très faible degré de formalisation. Les possibilités de validation de ces spécifications sont limitées, voire inexistantes. Plus grave, un spécifieur n'est absolument pas assuré que celui qui lira sa spécification en comprendra le sens voulu sans avoir à lui fournir un glossaire détaillé des termes utilisés.

1.2.2 Expressivité

L'*expressivité* est associée aux notations (à la fois textuelles et graphiques) des spécifications qui doivent être suffisantes pour spécifier un ensemble spécifique de systèmes. Différents aspects pourront être pris en compte par différentes notations. Il n'existe pas de formalisme universel, et augmenter l'expressivité d'un formalisme risque de réduire sa lisibilité et par là même son utilisation.

1.2.3 Abstraction

L'*abstraction* est une propriété importante puisque des formalismes présentant un plus grand degré d'abstraction conduiront à des spécifications plus concises et plus aisées à valider. Un faible niveau d'abstraction peut ainsi conduire par exemple à une explosion du nombre d'états d'une spécification à base de système états/transitions et rendre impossible sa validation. Nous reviendrons sur ce point (différents types d'explosion d'états) lors de la présentation de notre modèle. L'abstraction aide le spécifieur à ne s'attacher qu'aux points importants par rapport à un problème ou à une propriété donnée. Cependant, l'implémentation (par exemple pour faire un prototype) d'une spécification écrite à l'aide d'un formalisme très abstrait peut présenter beaucoup plus de difficulté.

Dans un cadre dynamique, la sémantique d'un formalisme est dite *pleinement abstraite* quand elle ne contient pas de détails (d'implémentation) non nécessaires. Les composants pleinement abstraits sont décrits en termes de comportements externes (leur interface). La notion de pleine abstraction nécessite la définition de relations d'équivalence ou de congruence spécifiques (par exemple l'équivalence observationnelle [205] pour les processus). La compatibilité de composants et une forme de sous-typage peuvent ensuite être définis en se basant sur la pleine abstraction [246]. Des composants présentant des comportements externes équivalents (ils se comportent de façon identique dans n'importe quel contexte) seront considérés comme équivalents. Un langage est dit *modulaire* si deux composants ayant une même interface se comportent de façon identique dans chaque contexte possible [159].

1.2.4 Lisibilité

La *lisibilité* induit la façon dont les spécifieurs pourront utiliser un formalisme avec ou sans un grand degré d'expertise. Cet aspect est souvent lié à la proximité des concepts de spécification et des concepts maîtrisés par le spécifieur (comme les concepts ensemblistes sous-jacents à Z [251] ou B [4] par exemple) ou à l'existence de notations graphiques formellement bien définies. L'expressivité et la lisibilité sont en quelque sorte opposées : les formalismes très expressifs comme les spécifications algébriques [269], les logiques d'ordre supérieur [130, 214, 30], les algèbres de processus [205, 53] et les calculs [203] peuvent être considérés comme peu lisibles et par là même réservés à des spécialistes.

Les spécifications ne peuvent pas être totalement graphiques, et ce pour des raisons d'expressivité, mais tout formalisme devrait utiliser des notations graphiques communément acceptées (et formalisées) lorsque cela est possible. Un système dual avec une notation graphique (avec quelques composants textuels, utilisés par exemple dans les compositions) et une contrepartie textuelle associée semble être la meilleure approche. Les formalismes totalement graphiques (UML [249] sans OCL) ou totalement textuels (CCS [205], CSP [53], π-calculus [203]) ne sont pas pratiques pour différentes raisons : soit au niveau formel (une jolie notation graphique sans sémantique formelle et donc sans possibilité de vérification), soit au niveau de la lisibilité lorsqu'il s'agit de spécifications de taille importante. Certains systèmes utilisent les deux types de notations (SDL [105], Raise [260], [233, 82], Argos [159], CPN [157]), d'autres autorisent la dérivation d'une représentation graphique à partir de la totalité ou d'une partie d'une représentation textuelle (Estelle [153], LOTOS [154]).

1.2.5 Moyens de structuration

La spécification de systèmes présentant un certain niveau de complexité rend nécessaire la définition de moyens de structuration des spécifications. Ce concept est aussi appelé modularité dans certains contextes [269]. Les différents moyens de structuration sont :

- l'enrichissement : ajout d'opérations, d'axiomes, de règles, etc. La cohérence doit être vérifiée : dans quelles limites ces ajouts ne contredisent-ils pas la spécification initiale ?
- le renommage (d'opérations, de variables, etc.),
- le masquage (pour rendre impossible l'invocation d'une opération hors d'un certain contexte),
- la combinaison de spécifications, qui construit une hiérarchie d'utilisation (un arbre),
- différentes formes d'héritage et de raffinements, qui peuvent dans certains cas correspondre à une ou plusieurs des structurations précédentes.

Ces moyens rendent possible la définition de "grosses" spécifications à partir de "petites". Elles rendent aussi possible la réutilisation de composants déjà spécifiés.

Un formalisme a la propriété de *compositionalité* quand la sémantique des composites peut être dérivée de celle des composants à partir desquels il est construit. La compositionalité autorise la spécification (et la vérification) modulaire de composites, elle augmente la lisibilité des compositions. Elle conduit à la définition de composants en termes d'interfaces. Les détails des composants derrière leur interface sont masqués, la compositionalité est donc fortement reliée à l'abstraction. Des exemples de formalismes avec une sémantique compositionelle sont LOTOS [154] et Argos [189].

1.2.6 Validation et vérification

La *validation* permet de vérifier que la spécification d'un système correspond aux besoins exprimés pour ce système. La *vérification* permet de s'assurer que les modèles de la spécification, à différents niveaux d'abstraction[1] correspondent ou ont des propriétés voulues. Pour citer [45], la validation correspond à la question "am I building the right system?" (suis-je en train de construire le bon système ?), tandis que la vérification correspond à la question "am I building the system right?" (suis-je en train de construire correctement le système ?). La validation et la vérification peuvent être faites par des preuves de théorèmes, de la vérification de modèle et des tests d'équivalences. Il est aussi possible de pratiquer des tests mais qui ne présentent pas une garantie d'exhaustivité.

Avec la *vérification de modèle* [75], les propriétés sont exprimées dans une logique (généralement temporelle) et sont vérifiées par rapport aux modèles de la spécification définis (ou transformés en) systèmes finis d'états et de transitions [17]. La spécification doit donc pouvoir être traduite en un système d'états et de transitions. L'intérêt de cette approche est son aspect automatique. Même si les propriétés peuvent être plus complexes à exprimer, la vérification de modèle se termine par une réponse positive ou par la production d'un contre-exemple. L'inconvénient majeur est l'impossibilité de recourir à cette technique sur des systèmes non finis.

En contraste avec la vérification de modèle, la *preuve de théorème* nécessite que l'utilisateur construise (au minimum une partie de) la preuve, même s'il existe des systèmes de génération de scripts de preuves pour certains formalismes (voir par exemple TLP [106] en ce qui concerne TLA). Les spécifications peuvent être données en termes de logiques [177, 201] ou de spécifications algébriques [99] et les preuves sont faites sur les modèles de la spécification (les spécifications algébriques ayant une sémantique initiale sont par exemple transformées en règles de réécriture) et vérifiées par des prouveurs [127, 30, 130]. L'avantage de la preuve de théorème par rapport à la vérification de modèle est la possi-

[1] Y compris l'implémentation, niveau le moins abstrait.

bilité de prendre en compte des données complexes qui conduiraient à une explosion du nombre d'états rendant toute vérification de modèle trop coûteuse en temps, voire impossible. L'inconvénient majeur est la relative complexité des preuves manuelles. Il existe des systèmes qui combinent preuve et vérification de modèle, tels que PVS [214, 215].

Les *outils de test d'équivalence* tels qu'Aldébaran [109] utilisent des relations d'équivalence particulières pour vérifier que deux modèles (systèmes d'états et de transitions) partagent une propriété donnée. Il s'agit souvent de prouver une propriété relativement abstraite, avec peu de données. Comme pour la vérification de modèle, il est nécessaire que les deux systèmes soient finis, même s'il existe des techniques "à la volée" [152, 181] où la vérification se fait en même temps que la construction des systèmes.

L'*animation* de spécifications permet de valider une séquence (dans le cadre de systèmes dynamiques) ou une construction (dans le cadre de spécifications algébriques). Elle permet aussi d'avoir une sorte de prototype de la spécification ou du système spécifié. Il existe de nombreux outils dédiés à l'animation de spécifications (par exemple Design/CPN [73] pour les réseaux de Petri colorés [157] et Statemate [142] pour les Statecharts). L'animation nécessite cependant que la spécification ait une sémantique opérationnelle ou qu'elle soit exécutable, ce qui n'est pas le cas de spécifications présentant un haut niveau d'abstraction par exemple.

Des *tests* peuvent être effectués sur une spécification, soit à un niveau abstrait si le formalisme a une sémantique opérationnelle (interaction avec des outils d'animation), soit au niveau d'une implémentation (avec un prototype généré, comme pour LOTOS avec CADP [110]). Tests et preuves sont souvent opposés mais sont en fait complémentaires. Au contraire des preuves, le test n'est pas exhaustif. Des outils sont utilisés pour trouver des suites de test intéressantes [108, 139]. D'autre part, la possibilité d'avoir un développement formel entièrement et formellement prouvé restera toujours un mythe, au moins tant qu'il existera des parties non prouvées telles que les compilateurs ou les systèmes d'exploitation.

En fait, même sans preuves (ou tests), les spécifications formelles sont toujours utiles puisqu'un bon formalisme et une méthodologie associée permettent souvent de se poser (et d'y répondre) les bonnes questions sur le système [7]. Cela permet aussi au spécifieur de détecter plus tôt les problèmes (incohérences).

1.2.7 Méthode

Nous pensons, comme [18] que la définition de méthodes associées aux formalismes de spécification est nécessaire à leur plus grande utilisation. Associées à des spécifications formelles graphiques, ou disposant d'une représentation graphique, ces méthodes permettent l'utilisation des spécifications formelles par des non spécialistes. C'est un point très important qui fera l'objet du chapitre 3, dans lequel nous étudions les méthodes existantes associées aux spécifications mixtes et nous faisons une proposition destinée à améliorer ces méthodes.

1.2.8 Outillage

Il s'agit d'outils permettant l'automatisation (ou la semi-automatisation) de traitements associés aux spécifications. L'aspect le plus abouti des outils est l'atelier de spécification qui présente, dans un cadre unique, un ensemble complet d'outils. C'est souvent d'ailleurs un préalable à l'utilisation industrielle d'une méthode et d'un formalisme de spécification. Citons pour exemple CADP [110] associé aux spécifications LOTOS, l'Atelier B pour les spécifications B, ou bien encore ObjectGéode pour les spécifications SDL. Il est à noter qu'il peut s'agir d'intégration d'outils provenant de différentes plates-formes.

1.3 Besoins associés aux spécifications mixtes

Pour les spécifications mixtes, les mêmes besoins que pour les spécifications en général sont nécessaires. Cependant, du fait même de la mixité des composants spécifiés, certains nouveaux besoins apparaissent.

1.3.1 Expression des aspects

Comme nous l'avons dit plus haut, les composants peuvent avoir plusieurs aspects différents. Le formalisme doit être à même de les exprimer. Dans le cadre de cette thèse, nous nous intéressons aux systèmes intégrant à la fois des aspects statiques et dynamiques. Il existe d'autres aspects que nous ne prendrons pas en compte ici. Dans ce contexte, seuls les aspects suivants sont pertinents :

– l'aspect statique,
– l'aspect dynamique,
– la composition.

Ce sont des aspects que l'on peut retrouver dans de nombreux formalismes, tant dans le domaine orienté objet (UML [249]), que dans celui des communications (LOTOS [47], SDL [105]) par exemple.

Aspects statiques et formalismes associés

Nous appelons *aspect statique* (mais aussi *aspect données*) la spécification (ou la partie de la spécification dans le cadre de spécifications homogènes) correspondant aux types de données (au sens d'un ensemble de types et de propriétés d'opérations sur ces types) du système à spécifier. Ceci correspond à la fois aux plans statiques et fonctionnels des modèles à objets [238, 249]. Il peut s'agir des données échangées entre composants, ou bien d'une représentation de l'état d'un ou de l'ensemble des composants. Les spécifications de l'aspect statique peuvent être de deux types : orientées modèles ou orientées propriétés.

Spécifications statiques orientées modèle. Ces spécifications (VDM [158], Z [184, 251], B [4]) sont basées sur des concepts ensemblistes et ont un bon niveau de lisibilité (car elle recourent aux notions ensemblistes) mais un moindre niveau d'abstraction (car se basant sur une représentation ensembliste des données) que les spécifications orientées propriétés.

Dans ces spécifications, la notion de base est celle d'un modèle. Une représentation en termes ensemblistes en est donnée (ensembles de base, constructions et contraintes ensemblistes), puis des opérations sont définies sur cette représentation. Ces opérations peuvent être données sous forme d'algorithme abstrait (forme explicite de VDM), de substitutions généralisées (B) ou sous forme axiomatique (relations entre les états du modèle avant et après l'opération, en Z et forme implicite en VDM). Ces trois formes utilisent largement les prédicats.

La figure 1.1 donne un exemple de spécification d'un buffer borné (limité à `maxElements` éléments) en Z.

Cet exemple illustre bien le fait que la spécification se base sur une implémentation ensembliste donnée (ici un entier et une séquence) et pas sur les propriétés (abstraites) du modèle. Par contre, la définition des opérations en termes de préconditions et de postconditions est naturelle, expressive (entrées et sorties, généricité) et lisible (petit nombre d'opérateurs mathématiques bien connus utilisés). Il faut

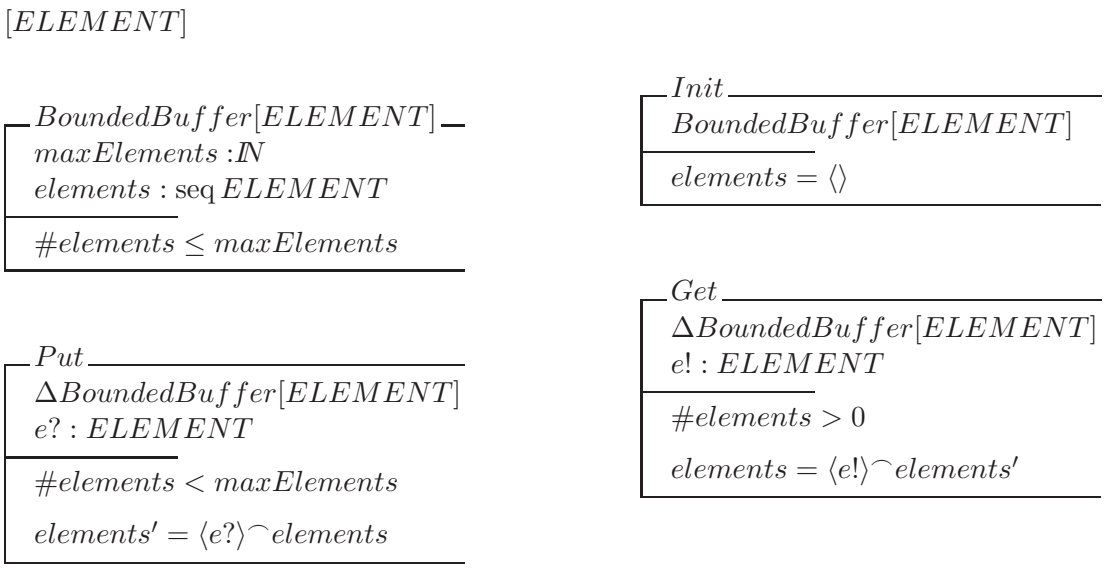


Figure 1.1 : Buffer borné – Z

noter que l'effet des opérations lorsque les préconditions ne sont pas respectées n'est pas spécifié, en cela la spécification obtenue est "lâche".

Il est cependant possible de considérer Z non seulement comme un formalisme, mais aussi comme une méthode de conception des spécifications écrites dans ce formalisme [251]. Dans cet esprit, qui est aussi celui que nous développons dans le chapitre 3, chaque spécification d'opération (avec des pré et postconditions) est combinée avec une spécification des erreurs associées (avec une précondition équivalente à la négation de celle de l'opération) dans le but d'obtenir une spécification globalement robuste.

Spécifications statiques orientées propriétés. Les principaux représentants de cette catégorie sont les langages de spécification algébriques [99, 269]. L'idée est qu'il est possible de modéliser un type de données comme un ensemble d'ensemble de valeurs (un ensemble de valeurs par type de donnée) et d'opérations (typées par des signatures) sur ces ensembles. Les propriétés de ces opérations sont données sous forme d'axiomes. Les modèles de la spécification sont des algèbres. C'est une représentation très abstraite des données et des fonctions car l'on s'attache à leurs propriétés et pas à la façon (algorithme) de les obtenir.

Deux approches existent. La première (ACT ONE [99]) fait l'hypothèse d'une algèbre particulière correspondant à la spécification : l'algèbre initiale. Celle-ci est équivalente, à un isomorphisme près, à l'ensemble des algèbres vérifiant la spécification. C'est ce qu'on appelle l'hypothèse de la *sémantique initiale*. Son avantage majeur est la possibilité de transformer les axiomes en un ensemble de règles (orientées) de réécriture et d'obtenir ainsi une sémantique exécutable pour les spécifications. Dans certains cas, cette hypothèse est trop stricte (fonctions partielles, généricité) et la sémantique des spécifications est *lâche* (ASL [268], Pluss [128], la famille OBJ [121, 129], LSL [127], CASL [79, 208]), c'est-à-dire que les algèbres vérifiant la spécification ne sont pas forcément isomorphes.

De façon générale, la famille des spécifications algébriques est grande. CoFI (Common Framework Initiative for algebraic specification and development [78]) est une proposition récente pour tenter de proposer un cadre général à l'ensemble des spécifications algébriques. Dans cette optique, le langage CASL

[79, 208] (Common Algebraic Specification Language) a été développé. Cette proposition ne se limite d'ailleurs pas aux spécifications statiques mais propose aussi des extensions pour prendre en compte des aspects dynamiques (COFI-Reactive [80], CASL-LTL [232]).

La figure 1.2 donne un exemple de spécification d'une liste bornée en CASL.

```
spec
  Elem =
  sort Elem
end

spec
  List [Elem] =
  free type
    List[Elem] ::=
      nil |
      cons(first:?Elem;
           rest:?List[Elem])
  op
    __ ++ __ :
    List[Elem] X List[Elem]
      -> List[Elem],
    assoc, unit nil

  vars
    e:Elem; l,l':List[Elem]

  axiom
    cons(e,l) ++ l'
      = cons(e,l ++ l')

  op
    reverse : List[Elem]
      -> List[Elem]
  axioms
    reverse(nil)
      = nil;
    reverse(cons(e,l))
      = reverse(l) ++ cons(e,nil)
end
```

```
spec
  Bounded_List [Elem]
    [ op bound : Nat]
    given Nat =
  List [Elem] and Ordered_Nat
then

  op
    length : List[Elem] -> nat

  vars
    e:Elem; l:List[Elem]

  axioms
    length(nil)
      = zero;
    length(cons(e,l))
      = succ length(l)
  sort
    Bounded_List[Elem]
      = { l:List[Elem] ·
          length(l) < bound }
  type
    Bounded_List[Elem] ::=
      nil |
      cons(
        first:?Elem;
        rest:?Bounded_List[Elem])?
end
```

Figure 1.2 : Liste bornée – CASL

Nous en donnons ici juste une partie, pour montrer l'expressivité et les moyens de structuration du langage CASL. La spécification complète peut être consultée à [77].

Une autre famille de cette catégorie est la famille des langages associés aux systèmes de preuve en logique du premier ordre comme LP [127] ou, plus performants, d'ordre supérieur comme PVS [215, 214], HOL [130], Isabelle [213] ou Coq [30]. Ce sont des formalismes très puissants mais plus adaptés aux preuves qu'aux spécifications (très expressives mais peu lisibles). Elles sont d'ailleurs souvent utilisées dans ce contexte de preuve uniquement (comme pour CASL, Z et TLA dont il existe une théorie en Isa-

belle [207, 168, 162], CSP et CCS en HOL [63, 211], algèbres de processus en PVS [31] ou en Coq). Elles peuvent aussi servir de base à l'intégration de différents formalismes [49, 171].

La table 1.1 résume les approches statiques par rapport aux besoins.

critère	formalismes		
	orienté modèle	algébrique	logique
expressivité	+	++	+++
abstraction	-	+	+
lisibilité	+	-	- -
structuration	-	++	++
base	schéma, machine	module	théories
moyens	+(B,ObjectZ)/-(Z)	++	++
	tous types (B)	tous types	tous types
outils	peu	oui	oui
vérification	typage, vérif. de modèle, test	réécriture, preuve, test	preuve

Table 1.1 : Tableau récapitulatif – Formalismes statiques

Nous pensons que l'approche algébrique est celle qui présente le meilleur compromis. En effet, les outils disponibles sont plus nombreux, les moyens de structuration meilleurs que pour l'approche orientée modèle. Les possibilités d'abstraction sont très bonnes, et ce tout en conservant un meilleur niveau de lisibilité que les logiques d'ordre supérieur.

Caractéristiques des aspects dynamiques

Nous appelons *aspect dynamique* (mais aussi *aspect comportemental*) la spécification (ou la partie de la spécification) correspondant aux communications entre composants, au temps et à la concurrence, et au comportement individuel des composants (séquences possibles de communications).

Communication. Il existe deux possibilités de communication : la communication synchrone et la communication asynchrone.

Dans le cadre de la communication *synchrone* (LOTOS [47], CCS [205], CSP [53], Réseaux de Petri [235]), deux (ou plus dans le cadre d'une synchronisation multi-sens) composants se synchronisent lors des communications. Si l'un des composants n'est pas prêt, il y a blocage. Il peut y avoir émission et réception de valeurs par l'intermédiaire de messages, partage de valeurs correspondant aux offres de la synchronisation, ou encore utilisation de variables partagées. Les structures d'interprétation associées à la communication synchrone [239] sont relativement simples : traces d'événements (ou d'états des composants), arbres, structures de Kripke et automates étiquetés.

Dans le cadre de la communication *asynchrone* (SDL [105], Estelle [153]), il y a obligatoirement notion d'émetteur et de receveur. Le receveur se bloque en attente de message mais ce n'est pas le cas de l'émetteur. Une queue où sont stockés les messages est généralement associée à chaque composant. Différents cas sont possibles lorsque le receveur y trouve un message auquel il ne s'attend pas : il peut se bloquer, le supprimer ou bien le remettre à plus tard. La communication asynchrone est plus proche du concept d'envoi de message dans les systèmes à objets. Toutefois, des formalismes comme UML [249] (dans les diagrammes MSC) prennent en compte les deux possibilités de communication. Il est

possible de simuler la communication asynchrone dans un cadre synchrone à l'aide de buffers non bornés et d'événements "triviaux" (ϵ) associés à tous les processus. L'utilisation de ces buffers peut néanmoins conduire à l'explosion du nombre d'états de la spécification. Les structures d'interprétation associées à la communication asynchrone [239] sont plus complexes que pour la communication synchrone (structures d'événements). Une classification de ces structures d'interprétation et les relations qu'elles partagent est donnée dans [239, 212].

Il ne faut pas confondre communication synchrone (respectivement asynchrone) qui est un critère sur la communication entre composants, et langage synchrone (respectivement asynchrone) qui est une caractérisation des langages relativement à la sémantique de la concurrence.

Un point important au niveau de la communication est la prise en compte de la possibilité de spécifier des *systèmes ouverts* (interagissant avec leur environnement). Ceci permet une meilleure structuration (par composition et réutilisation) des composants. Certains formalismes le permettent (LOTOS, SDL), d'autres imposent la spécification de l'environnement du système au sein de la spécification de plus haut niveau (Estelle).

Temps et concurrence. Il existe deux autres critères importants de différenciation des formalismes au niveau de la spécification des aspects dynamiques : le type de temps [156] et la modélisation de la concurrence.

Le *temps réel* correspond à un temps dense (l'ensemble des réels) où, entre deux événements, il est toujours possible qu'il s'en produise un troisième. Il est possible de raisonner sur le temps qui s'est déroulé entre deux instants. Cela permet de modéliser des problèmes complexes (contraintes temporelles des systèmes critiques) mais rend aussi complexe la vérification des spécifications résultantes (la vérification des systèmes hybrides se fait par exemple en travaillant sur des polyèdres ayant autant de dimensions que d'horloges dans le système).

Par opposition, le *temps logique* est un temps discret qui peut être vu comme une suite d'événements (projetés sur l'ensemble des entiers naturels). Seuls les instants où un événement s'est produit sont retenus et il n'est pas possible de raisonner directement sur l'intervalle réel entre deux instants. En contrepartie, c'est un modèle de temps qui fournit des structures d'interprétation beaucoup plus simples (séquences - ou traces, arbres) et sur lequel il est beaucoup plus simple de procéder à des vérifications. Il faut noter qu'il existe des possibilités de simuler une forme de temps réel dans un contexte de temps logique par la définition de composants représentant l'écoulement du temps (à une échelle donnée toutefois) par l'envoi de messages particulier aux autres composants. Ceci peut être fait explicitement (méthodologie de spécification [6, 167]) ou implicitement comme en SDL [105] qui dispose de signaux particuliers associés à l'échéance de timers.

Un second critère de différenciation est la sémantique de concurrence. La concurrence peut être *réelle*, c'est-à-dire que deux événements (non synchronisés) peuvent éventuellement se produire dans un même temps. C'est le domaine des langages synchrones (Lustre [138], Esterel [39], Signal [135]) qui font par ailleurs l'hypothèse de temps zéro. En effet, dans ces langages, il est possible de faire des calculs en temps nul, et une entrée et le résultat d'un calcul sur cette entrée peuvent "arriver" en temps. L'approche synchrone est particulièrement adaptée aux applications soumises à des contraintes temporelles fortes et la composition des composants se fait très bien. Il faut noter qu'il existe des compilateurs performants pour les langages synchrones, le compilateur pour Electre a même été prouvé [16]. Cependant, il s'agit plus de langages de programmation abstraits que de vrais langages de spécification (abstraits).

Les langages asynchrones (LOTOS, SDL) interdisent le fait que deux événements non synchronisés interviennent simultanément. Leur sémantique de concurrence procède par *entrelacement* des événe-

ments non synchronisés. Ceci peut conduire à une explosion combinatoire. Une classification détaillée des différents modèles de concurrence et les relations qu'ils partagent est donnée dans [239, 212].

Comportement des composants. Il existe différentes possibilités pour exprimer le comportement de composants. Par comportement, nous entendons la spécification de l'ensemble des différents enchaînements possibles d'opérations ou de communications du composant. Nous présentons ces possibilités du moins au plus abstrait.

Une première possibilité concerne la définition explicite des séquences d'événements possibles. Il s'agit des formalismes directement (Signal, Esterel, Lustre, MEC, SDL, Estelle, Automates, Réseaux de Petri) ou indirectement[2] (algèbres de processus comme CCS et CSP, LOTOS) à base d'états et de transitions. Ces formalismes peuvent être plus ou moins abstraits. Les transitions peuvent contenir plus ou moins d'informations. Il est aussi possible de coder le système de transitions dans un cadre purement algébrique [23].

Une seconde possibilité est la définition d'une relation reliant un état avant l'opération à un état après l'opération. C'est le cas des formalismes se basant sur les pré et postconditions à la Hoare, ou des formalismes basés modèles comme Z, B ou VDM. Ici, une abstraction naturelle peut être construite à partir de ces pré et postconditions.

Enfin, il est possible d'utiliser une forme de logique temporelle (LTL [174], CTL [76], TLA [177], Unity [67], Trio [206]) pour exprimer le comportement. L'avantage en sera que le système et les propriétés attendues (à vérifier) pourront être exprimés dans le même formalisme.

Dans tous les cas, une représentation temporelle des séquences ou des possibilités de communications est donnée.

Composition

Nous appelons *aspect composition* (ou plus simplement *composition*) la définition d'un composite à partir de composants.

Il est important de différencier deux types de compositions : la composition d'un ensemble de spécifications d'aspects au sein d'un seul et unique composant (intégration d'aspects), et la composition parallèle de composants communicants.

D'autre part, s'il existe de nombreux types de compositions[3] (dans le cadre ensembliste ou orienté objet), nous nous restreignons à la composition forte des systèmes critiques et applications réparties (composition avec dépendance, exclusivité et prédominance).

Aspects dynamiques et formalismes associés

Comme pour les aspects statiques, il existe différentes grandes familles pour spécifier les aspects dynamiques, et qui présentent différentes instanciations des possibilités vues plus haut.

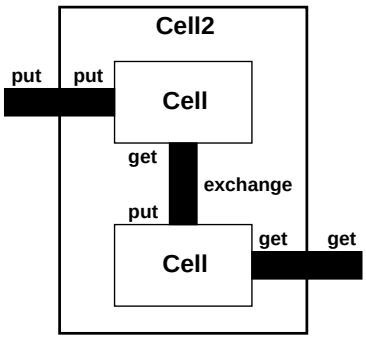
La première approche est celle basée sur les algèbres de processus, formalismes très expressifs (pour la dynamique) et se basant en général sur un petit nombre d'opérateurs de composition d'atomes de base (actions, événements, offres) et de comportements construits (processus, agents, comportements). Comme exemples de cette famille, on peut citer les précurseurs CCS [205] et CSP [53], Basic LOTOS

[2] C'est-à-dire qu'il est possible de dériver un système de transitions à partir de ces spécifications ou bien encore que le système de transitions est le modèle ou la structure d'interprétation de cette spécification.

[3] Voir par exemple la classification de [187] qui se base sur les critères suivants : cardinalité (aussi appelée multiplicité) et cardinalité inverse, dépendance, exclusivité et prédominance.

[47], ACP [38]. Les opérateurs expriment en général la séquence, la composition parallèle (différentes combinaisons de sémantiques du temps et de la concurrence), l'instanciation de processus, le choix, les comportements gardés et le masquage. Si cette approche est peu lisible, elle permet de nombreuses possibilités d'extensions (et de gain en expressivité), comme les versions prenant en compte du temps [116, 252, 240, 85] ou bien encore des probabilités [9, 10]. Ce sont des approches formellement bien définies (sémantiques de traces, d'échecs, ...).

La figure 1.3 donne un exemple de spécification d'une cellule de taille 2 (sans données) qui utilise la communication parallèle de deux cellules de taille 1 en (Basic-)LOTOS. La partie de gauche est une représentation graphique personnelle de cette composition et ne fait pas partie de la syntaxe LOTOS.




```
specification Cell2 [put, get]
: noexit
behaviour
   hide exchange in
      Cell [put, exchange]
   |[exchange]|
      Cell [exchange, get]
where
   process Cell [put, get]
   : noexit :=
      put; get; Cell [put, get]
   endproc
endspec
```

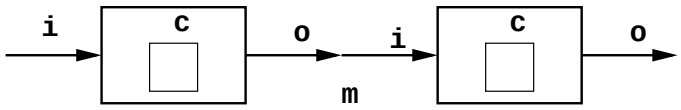
Figure 1.3 : Cellule de taille 2 – LOTOS

Une seconde approche est celle des logiques dans lesquelles des propriétés temporelles des composants sont exprimées. Les aspects de structuration et de composition de ces logiques sont en général nettement insuffisantes (ou procèdent par solution ad hoc). Ces logiques peuvent être linéaires (les modèles sont les différentes séquences - potentiellement infinies - d'actions des composants) comme LTL [174], la partie purement temporelle de TLA [177], Unity [67] ou arborescentes (les modèles sont des arbres représentant à chaque instant les différents choix d'actions possibles pour un composant) comme CTL [76]. Ces dernières sont plus expressives [191]. La forme la plus expressive de ces logiques est le μ-calcul [170] (qui utilise des opérateurs généralisés de points fixes). En règle générale, plus la logique est expressive, moins il existe de procédures de vérifications qui leur sont associées (comme la vérification de modèle [75] qui est particulièrement bien appropriée à la vérification de propriétés écrites dans ces logiques). Ces logiques permettent aussi d'utiliser le même formalisme pour la spécification que celui couramment utilisé pour la vérification.

La figure 1.4 donne un exemple de spécification de cellule de taille 2 par composition de deux cellules de taille 1 en TLA. Cette spécification est inspirée des files de [196].

On voit bien que TLA est très expressif. Ici la communication est synchrone ($o' = o$ dans CellPut, et $i' = i$ dans CellGet), le mode de concurrence par entrelacement ($\Box[i' = i \vee o' = o]_{i,o}$), mais il est possible de faire d'autres choix [196]. Cet exemple simple montre toutefois le manque de lisibilité des spécifications TLA.

La dernière approche est basée sur les formalismes à base de systèmes de transitions comme les automates [17], divers systèmes de transitions, les réseaux de Petri [235], les parties dynamiques d'Estelle



$$
\begin{array}{lcl}
CellPut & \triangleq & c = \langle\rangle \ \wedge \ c' = c \circ \langle i' \rangle \\
 & & \wedge \ o' = o \\
CellGet & \triangleq & c \neq \langle\rangle \ \wedge \ o' = \mathrm{head}(c) \\
 & & \wedge \ c' = \mathrm{tail}(c) \ \wedge \ i' = i \\
CellNext & \triangleq & CellPut \vee CellGet \\
CellInit & \triangleq & c = \langle\rangle \ \wedge \ i = o \\
CellSpec & \triangleq & CellInit \ \wedge \ \Box[CellNext]_{c,o} \\
Cell & \triangleq & \exists c : CellSpec
\end{array}
\qquad
\begin{array}{lcl}
Cell2 & \triangleq & Cell[m/o] \ \wedge \ Cell[m/i] \\
 & & \wedge \ \Box[i' = i \vee o' = o]_{i,o}
\end{array}
$$

Figure 1.4 : Cellule de taille 2 – TLA

[153] ou de SDL [105], ou les langages synchrones comme Lustre [138], Esterel [39] et Signal [135]. Ces approches présentent le double intérêt d'être très lisibles et de bien se prêter à la composition parallèle. Ainsi par exemple le concept de produit synchronisé d'automates [17] ou les différentes façons de coller les réseaux de Petri pour modéliser la communication entre composants (synchrone ou asynchrone d'ailleurs) [112] sont bien définis et compréhensibles. Les modèles à base d'états et de transition servent de structure d'interprétation pour les formules de logique temporelle dans le cadre de la vérification de modèle. Spécifier directement dans ce formalisme présente donc un intérêt.

La figure 1.5 donne un exemple de spécification de cellule de taille 2 utilisant deux réseaux de Petri avec synchronisation sur une transition.

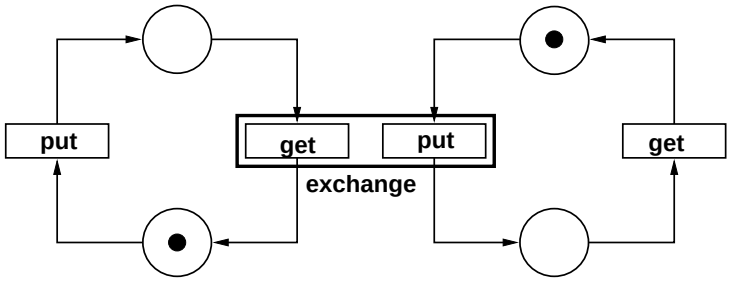


Figure 1.5 : Cellule de taille 2 – Réseaux de Petri

Il est ici facile de se rendre compte de la lisibilité de ce formalisme (comparer par exemple avec la spécification en LOTOS ou en TLA).

La table 1.2 résume les approches dynamiques par rapport à une partie besoins (ceux qui sont communs aux éléments de chaque approche). Les spécificités de tel ou tel formalisme par rapport aux autres besoins seront vues quand nous présenterons différentes approches mixtes.

Nous pensons que les approches à base d'algèbres de processus et celles à base de systèmes états/transitions sont les deux meilleures approches si l'on considère expressivité et lisibilité. Les deux approches permettent la spécification à un niveau d'abstraction suffisant (niveaux de description des algèbres de processus, possibilités d'abstraction des réseaux de Petri ou de certains systèmes de transitions dont nous reparlerons dans le chapitre 2, les systèmes de transitions symboliques). Les logiques sont très

critère	formalismes		
	algèbre de proc.	logique	système états/trans.
expressivité	+	++	+/-
abstraction	+/-	+	+/-
lisibilité	-	- -	+/++
structuration			
base	processus, agent	formule	système états/trans.
moyens	+/++	-	+/++
	compo. parallèles	conjonction de formules	produit synchronisé
	séquence		colle de places
	masquage		colle de transitions
	autres opérateurs		
outils	nombreux	peu, ou dédiés	nombreux
vérification	tout (preuve en HOL)	vérif. de modèle, preuve	tout sauf preuve

Table 1.2 : Tableau récapitulatif – Formalismes dynamiques

puissantes (expressivité, abstraction) mais présentent une lisibilité insuffisante dans le contexte qui nous intéresse.

1.3.2 Cohérence entre aspects

La spécification des différents aspects peut passer par un formalisme homogène très expressif ou par l'utilisation de différents formalismes dédiés. Le problème de la cohérence se pose alors.

Ce point est lié à la sémantique formelle d'une spécification mixte. Cette sémantique peut être partielle et dans ce cas seulement certains aspects auront une sémantique formelle. Tous les avantages des spécifications formelles sont alors perdus puisque l'ensemble des aspects n'est pas formalisé. Même si l'ensemble des différents formalismes associés aux aspects a une sémantique formelle, il est indispensable, à partir du moment où ces parties sont amenées à interagir à l'intérieur d'un tout (le composant global), de définir formellement comment cette interaction procède. Il peut s'agir d'une approche syntaxique ou sémantique[4]. Il s'agira en général de définir une "colle" exprimant les relations entre les parties au sein d'un tout. Cette colle, qui doit être formelle, peut être *implicite* (comportement "par défaut") ou *explicite* (pour une meilleure expressivité). La colle servant à l'intégration des composants peut ou non correspondre à celle liant les composants dans une composition parallèle. Il est évident que s'il s'agit du même type de colle, la lisibilité du formalisme est grandement améliorée. La formalisation passe par la définition d'une sémantique globale pour le composite.

1.3.3 Structuration et réutilisation

La structuration des spécifications doit être adaptée à l'aspect mixte des spécifications, c'est-à-dire prendre en compte les aspects statique, dynamique ainsi que leur composition. D'autre part, il doit être possible de définir séparément les différents aspects mais aussi de les combiner et de les réutiliser séparément. Un critère important est donc que la dynamique et la statique soient des aspects entiers des

[4]Ces termes proviennent de l'étude des approches combinant Z et les algèbres de processus, voir la section 1.4.1.1 pour les définitions.

composants (et pas uniquement une partie trivialement obtenue à partir d'un autre aspect comme par exemple les interprétations dynamiques des opérations dans le cadre de certaines intégrations Z/CSP). La structuration par aspects permet d'éviter le problème bien connu de l'anomalie d'héritage (voir la section 2.2) qui survient lorsque les aspects statiques et dynamiques sont mélangés.

1.3.4 Orientation objet

Le prototypage est un moyen de vérifier les systèmes. Les prototypes sont aussi un moyen de montrer au client comment le système final fonctionnera. La réutilisabilité est un mot-clé tant pour la spécification que la programmation, et la génération de code à partir de spécifications mixtes semble tout naturellement devoir s'orienter vers les languages concurrents à objets [8, 65, 136, 52]. Pour réduire l'espace entre les niveaux concernant les spécifications abstraites et l'implémentation, le formalisme et le langage visé par la génération de code devraient partager autant de concepts que possible [13, 82, 88, 227], et les notions de *classe/instance*, *identifiants d'objets/processus*, *spécialisation/généralisation*[5] devraient être présentes dans le formalisme. Ceci assure aussi une plus grande proximité avec les méthodes semi-formelles d'analyse et de conception à objets, et permet donc (en partie) d'utiliser ces méthodes en association avec le formalisme (les spécifications ayant des propriétés objet utilisées en conception, comblent le vide entre analyse informelle à objet et programmation à objet). OMTroll [88], les Statecharts [142] ou les réseaux de Petri orientés objets [40, 34, 134] sont des exemples de formalismes qui incluent certains concepts objets dans une représentation graphique. Des critères (principalement orientés objet ou basé objet) pour la classification et la comparaison et un panorama de quelques formalismes orientés objets peuvent être trouvés dans [134, 11].

1.3.5 Conclusion sur les besoins

Nous avons donné un certain nombre de besoins associés aux spécifications formelles mixtes qu'il est ici possible de résumer :

- niveau de formalité, expressivité, abstraction et lisibilité concernant les aspects, ceci prenant en compte au niveau de l'aspect dynamique :
  - type de communication (synchrone et/ou asynchrone, prise en compte ou non des systèmes ouverts)
  - type de concurrence (vraie ou entrelacement)
  - prise en compte du temps (temps logique, possibilité de temps réel)
- moyens de structuration (niveau de formalité, expressivité, abstraction et lisibilité concernant les composants globaux) des spécifications :
  - intégration d'aspects : vraie séparation des aspects, existence d'une sémantique globale, type d'intégration (syntaxique ou sémantique), type de colle (implicite ou explicite)
  - composition parallèle : type de colle (implicite ou explicite), colle commune ou non avec l'intégration
  - réutilisation de composants
  - orientation objet
- validation et vérification (types de moyens et types de preuves)
- outillage

[5] Au niveau des langages de programmation, seule la spécialisation est prise en compte. Cela s'appelle alors sous-typage, ou héritage dans le cas des langages de programmation orientés objet.

L'étude des méthodes associées aux formalismes fera l'objet du chapitre 3.

Ces besoins nous serviront de base dans la suite pour étudier les particularités d'un ensemble de formalismes représentatifs (précurseurs, formalismes très utilisés ou développements récents présentant un intérêt important par rapport à ces besoins).

1.4 Panorama des spécifications formelles mixtes

Nous présentons ici un panorama général des spécifications mixtes. Une première classification est faite entre approches hétérogènes et approches homogènes. Des grandes familles sont ensuite distinguées, basées sur les différentes possibilités de combinaisons pour les approches hétérogènes et sur les différents formalismes utilisés pour les approches homogènes. La table 1.3 donne un aperçu des principaux formalismes mixtes. Les formalismes en soulignés seront détaillés par un tableau récapitulatif.

type	dynamique	statique	noms
Hétérogène	AP	Basé modèle	ObjectZ-CSP, CSP-OZ, ZCCS, ZCSP, Raise
	AP	Algébrique	LOTOS, PSF, Raise
	Système E/T	Basé modèle	$\mu\mathcal{SZ}$, MaC, Event Calculus
	Système E/T	Algébrique	SDL, CASL-Chart, TAG
	Système E/T	– spécial –	Estelle, UML, Argos, BDL
	Réseau de Petri	Algébrique	OBJSA, Clown, CO-OPN/2
	Réseau de Petri	– spécial –	CO, OPN
Homogène	Algébrique		LTL, Rewriting Logic, ASM (famille)
	Logique		TLA, Unity, TRIO, OSL (famille)
	AP		CCS+value, CSP, π-calcul

Table 1.3 : Formalismes utilisés pour les aspects

Nous avons volontairement omis l'ensemble des formalismes n'étant pas adaptés à la spécification mixte, comme par exemple les langages synchrones (performants au niveau du contrôle mais qui ne présentent qu'une très faible possibilité de définition de données, et plus proches de langages de programmation que de formalismes de spécification). De même nous ne prenons pas en compte les systèmes hybrides, plus expressifs (temps réel - continu) mais très complexes à analyser - résolution de systèmes d'équations par polyèdres).

1.4.1 Approches hétérogènes

Les approches hétérogènes combinent plusieurs formalismes différents pour exprimer les différents aspects des systèmes. Il s'agit en général de choisir un formalisme approprié par aspect puis de définir un cadre à la combinaison (cohérence, sémantique globale). Nous présentons ces approches en nous aidant d'une première classification sur les formalismes associés à la partie dynamique (algèbres de processus, systèmes états/transitions et/ou logiques temporelles, réseaux de Petri) puis une seconde sur le formalisme concernant la partie statique (basé modèle, algébrique, logique). Ce choix se justifie en ce que l'ensemble des formalismes mixtes données-contrôle hétérogènes procède par extension d'un formalisme dynamique avec de la statique, ou au minimum, encapsulation d'une partie statique dans une partie dynamique.

1.4.1.1 Algèbres de processus et formalismes basés modèle

Combinaisons avec Z. Il existe de nombreuses propositions pour étendre les approches statiques orientées modèle avec des aspects dynamiques ou plus généralement intégrer une partie statique spécifiée en Z ou en B (description des données et des opérations) avec une partie dynamique spécifiée dans un formalisme approprié. Nous donnons dans la suite un aperçu de ces différentes approches.

La majeure partie des développements récents considère la combinaison de Z (ou Object-Z [246, 96]) avec des algèbres de processus (CSP [237, 146, 114, 247], CCS [123, 122, 258], LOTOS [91, 92, 46]).

L'intégration d'une spécification statique en Z et dynamique par l'intermédiaire d'une algèbre de processus correspondante peut se faire, selon une classification proposée par [115], soit à un niveau syntaxique, soit à un niveau sémantique.

L'*approche syntaxique* définit un nouveau langage (plus expressif) comme la combinaison de deux langages (Z/CCS ou Z/CSP) existants. Cela nécessite la définition d'une sémantique globale (dans laquelle on fait correspondre les concepts des 2 langages de base) et d'outils dédiés. Les travaux présentés dans [123, 122, 258] suivent cette approche. Leur modèle se base sur la définition d'un système de transitions avec des états construits à partir des variables des schémas Z et des processus, et avec des transitions construites à partir des opérateurs de l'algèbre de processus mais pouvant faire référence à une opération décrite par un schéma Z. Cela correspond à utiliser Z comme calcul des valeurs dans CSP ou CCS à passage de valeurs. Cette approche est plus intégrée (utilisation de Z comme calcul pour les valeurs dans une algèbre de processus avec passage de valeurs) et il est possible d'accéder aux variables Z (et donc de modifier l'état des données) dans la partie données. Ceci est impossible avec l'approche sémantique en raison de l'absence de variables globales. Une approche équivalente utilisant des spécifications algébriques est LOTOS.

Le tableau 1.4 résume un formalisme représentatif de cette famille, ZCCS, par rapport aux besoins exprimés plus haut.

critère	commentaires		
	statique	dynamique	global
formalité	oui (Z)	oui (CCS)	oui, syntaxique
expressivité	+	+	+
abstraction	-	-	-
lisibilité	+	-	-
communication	synchrone, systèmes ouverts		
concurrence	entrelacement		
temps	logique		
séparation aspects	limité (système de transitions dépendant des variables Z)		
intégration	oui (opérationnelle)		
compo. //	opérateurs CCS		
réutilisation	non		
OO	non		
outils	?		
vérification	?		

Table 1.4 : Tableau récapitulatif – ZCCS

L'*approche sémantique* préserve la cohérence des formalismes initiaux (chaque partie peut donc être analysée séparément avec les outils existants) mais des liens sont faits entre les deux, comme prendre

une partie de la spécification Z et l'identifier à un processus (et lui donner par exemple une sémantique échecs/divergences comme en CSP). Ceci est surtout fait dans les approches basées sur Object-Z (grâce à l'analogie classe/processus) comme CSP-OZ [114] ou ObjectZ-CSP [247] mais aussi dans des approches à base de Z non objet comme [113, 237]. Ces approches ne permettent pas la définition explicite d'un comportement : les opérations sont possibles ou pas en fonction de leurs préconditions mais sinon c'est de l'indéterminisme. Cela représente une approche assez passive des composants (contrairement à l'approche syntaxique). Il n'y a pas séparation réelle des aspects.

Le tableau 1.5 résume un formalisme représentatif de cette famille, ObjectZ-CSP, par rapport aux besoins exprimés plus haut.

critère	commentaires		
	statique	dynamique	global
formalité	oui (Object-Z)	oui (CSP)	oui, sémantique
expressivité	+ (ensembliste)	+ (AP)	+/-
abstraction	- (Z)	- (expl. états)	-
lisibilité	+ (ensembliste)	- (AP)	-
communication	synchrone, systèmes ouverts		
concurrence	entrelacement		
temps	logique		
séparation aspects	non		
intégration	oui (échecs/divergences)		
compo. //	opérateurs CSP		
réutilisation	non		
OO	oui (statique en Object-Z)		
outils	?		
vérification	?		

Table 1.5 : Tableau récapitulatif – ObjectZ-CSP

Une autre classification correspond à la façon dont opérations (de Z) et événements (de l'algèbre de processus) se correspondent [115], soit correspondance entre opérations et événements (utile pour les hauts niveaux d'abstraction), soit utilisation de plusieurs événements et séparation entre réceptions et émissions (plus réaliste).

Il faut noter que récemment, l'application de ces moyens de combinaisons ont été appliqué à B [261, 62] (mais aussi à d'autres formalismes basés modèles comme MOSCA [163] pour VDM ou Raise [260]).

Raise. Il est impossible de terminer sur les approches hétérogènes avec formalisme basé modèle pour la statique et algèbre de processus pour la dynamique sans évoquer Raise. Raise [260] (Rigorous Approach to Industrial Software Engineering) est constitué d'une méthode de développement (Raise) et d'un langage (RSL). RSL [226] est basé sur VDM et ACT ONE pour les données, ainsi que sur CSP pour la dynamique. Cette foison de formalismes bien connus à conduit à son utilisation dans l'industrie. Raise rencontre cependant les problèmes de cohérence d'aspects qui se posent lorsque l'on combine de multiples formalismes. Il existe des outils pour Raise, mais ils ne prennent pas en compte les aspects

vérification (même s'il est possible d'engendrer du code impératif à partir des spécifications).

1.4.1.2 Algèbres de processus et formalismes algébriques

Il existe trois formalismes normalisés (ISO ou CCITT) pour la spécification des protocoles de communication et des systèmes ouverts distribués en général : LOTOS [154], SDL [66] et Estelle [153]. LOTOS est plus utilisé dans le monde universitaire. Estelle mais surtout SDL sont utilisés dans le monde industriel. Nous détaillons plus LOTOS et SDL, qui nous ont fortement inspirés.

LOTOS. L'idée de départ de LOTOS est que les systèmes peuvent être spécifiés en définissant les relations temporelles entre les interactions qui constituent le comportement externe observable d'un système [47].

La partie de LOTOS qui traite des comportements des processus et de leurs interactions (basic-LOTOS) est basée sur les algèbres de processus et s'inspire en cela fortement de CCS [205] et de CSP [53]. La seconde partie de LOTOS traite des données internes aux processus et des traitements qui s'y rattachent et s'appuie sur ACT ONE [99].

La partie statique concerne la définition et la manipulation des données qui sont attachées à la notion de portes de communication [220]. Le formalisme est celui des types abstraits algébriques. La sémantique est donnée par réécriture de termes [182]. Pour [124], LOTOS évite ainsi les problèmes rencontrés en Estelle [153] où les données sont spécifiées au moyen des types Pascal.

LOTOS permet de définir des types directement mais aussi par importation ou combinaison d'autres types. La généricité est permise par l'emploi de types paramétrés par des sortes, des opérateurs et des équations formels. Ces types formels peuvent alors être instanciés (actualisés). L'instanciation peut avoir lieu lors de l'instanciation des processus (voir partie dynamique). Cela est d'ailleurs préalable à toute utilisation dans les expressions comportementales. Notons enfin qu'il est possible de renommer les types (sortes et opérations du type), ce qui permet une forme de réutilisation. LOTOS est fortement typé [124].

La partie dynamique concerne les structures de contrôle. Comme en CCS [204] et CSP [151], le contrôle en LOTOS est décrit syntaxiquement par les termes d'une algèbre ([125]) appelés expressions comportementales. La synchronisation et la communication s'effectuent exclusivement par rendez-vous, sans partage de mémoire et via des portes. Sémantiquement parlant tout comportement représente un automate d'états finis ou infinis [125].

[185] précise que la synchronisation en LOTOS est :

1. *multi-directionnelle* : n processus peuvent participer au rendez-vous[6].
2. *symétrique* : tous les processus ont la même importance lors d'un rendez-vous. Il n'y a pas un initiateur et des répondeurs.
3. *anonyme* : la synchronisation est proposée à l'environnement. Il est théoriquement impossible de s'adresser à un processus particulier. Il est possible d'y remédier en donnant des identifiants aux processus et en communiquant ces valeurs lors du rendez-vous. L'utilisation de l'opérateur de masquage permet aussi un forme de communication privée.
4. *non déterministe* : quand plus d'une synchronisation est possible, l'une d'elles est choisie de façon non déterministe.

[6] Or, dans les articles sur LOTOS, tous les opérateurs sont toujours décrits par une sémantique binaire.

Les portes sont des canaux de communication permettant le rendez-vous entre n processus (ou tâches) se déroulant en parallèle. Il existe deux portes spéciales `: i` (opération interne au processus) et δ (sert à la définition sémantique de LOTOS dans le cas de la terminaison des processus ; elle ne doit pas être utilisée dans une spécification).

Les événements sur ces canaux sont atomiques et constituent les éléments de base de la synchronisation. On parle d'actions observables. L'ensemble des processus avec lequel un processus communique constitue son environnement. Un processus particulier, l'observateur humain, peut faire partie de cet environnement [47] (système ouvert).

Les portes peuvent être accompagnées (full-LOTOS) d'offres (émission (`!Valeur`) ou réception (`?Variable:Sorte`) de valeur, choix aléatoire d'une valeur, correspondance de valeurs). Des gardes permettent de restreindre les valeurs possibles dans le cas du choix aléatoire, ou de contrôler la valeur reçue dans le cas d'une réception. Par exemple, dans l'expression suivante : `échange !Put ?Get:NAT [Get>0]`, la valeur `Put` est émise lors du rendez-vous pendant qu'une valeur de type NAT (qui doit être supérieure à zéro) est reçue dans la variable `Get`.

Le tableau 1.6 est extrait de [124].

offre 1	offre 2	condition	action
$!V_1$	$!V_2$	$V_1 = V_2$	
$!V_1$	$?X_2 : S_2$	$V_1 \in domain(S_2)$	$X_2 := V_1$
$?X_1 : S_1$	$!V_2$	$V_2 \in domain(S_1)$	$X_1 := V_2$
$?X_1 : S_1$	$?X_2 : S_2$	$S_1 = S_2$	$X_1, X_2 := VtqV \in domain(S_1)$

Table 1.6 : Correspondance entre offres – LOTOS

Dans le cas de rendez-vous n-aires, les offres ne peuvent s'unifier que si l'intersection des ensembles de valeurs permis par les offres est non vide.

Dans le cas d'offres n-aires (ex : `p !`e_1 `?x:S !`e_2), il faut en plus que les attributs soient en nombre égal et compatibles. Une offre est toujours atomique, même quand elle comporte plusieurs attributs [182].

Des opérateurs permettent de composer les expressions comportementales pour en obtenir de nouvelles (les expressions ont une définition récursive). [47, 220] utilisent les transitions étiquetées pour donner la sémantique (dénotationnelle) de LOTOS.

Cette sémantique donne un moyen de dériver automatiquement les expressions comportementales. Il peut y avoir non déterminisme dans la réduction des transitions étiquetées, cependant dans de nombreux cas cet indéterminisme est réduit par l'interaction du processus avec son environnement.

Les opérateurs de composition (dynamique) de LOTOS sont :

- `stop` : comportement inactif (blocage)
- `exit` : dénote un processus qui se termine normalement (contrairement à `stop`). `exit` a comme sémantique le franchissement d'une porte δ : `exit` $\equiv$ δ`;stop`.
- `; :p;E` rendez-vous sur la porte p en préalable à l'expression comportementale E. On dit aussi qu'on a une transition étiquetée par la porte p. Si p est `i`, l'action n'est pas observable ; pour toute autre porte elle l'est.
- `[]` : choix non déterministe. L'un des comportements se déclenchera en fonction de l'état des communications sur leurs portes associées.
- `|[liste de portes]|` : composition parallèle. Les processus se synchronisent sur certaines

portes. Les transitions étiquetées par des portes n'appartenant pas à la liste se font de manière asynchrone. Deux opérateurs complémentaires existent : `||` (la liste des portes contient toutes les portes ; les processus sont entièrement synchronisés) et `|||` (la liste des portes ne contient que δ ; il y a entrelacement (interleaving) ; ils ne se synchronisent que sur leur terminaison).

- $[>$: permet de spécifier l'interruption d'un comportement par un autre.
- `hide` : cet opérateur permet de cacher certaines portes d'un comportement et ainsi de gagner en abstraction. En général, cela sert quand on a un processus "boîte noire" composé d'un certain nombre de sous-processus communiquant par des portes qui ne doivent pas être connues/vues de l'extérieur.
- $\rightarrow$: cet opérateur permet de conditionner un comportement par une garde. Dans le cas de gardes non exclusives, il peut y avoir non déterminisme. Quand aucune garde n'est évaluée à vrai le processus est équivalent à `stop`.
- des *prédicats de sélection* permettent d'imposer des restrictions sur les valeurs échangées lors d'une communication.
- `let` : bloc lexical habituel pour assigner des valeurs aux variables (locales).

Au niveau de la nécessité ou non, pour deux processus, de se synchroniser sur une porte, voir le tableau 1.7, extrait de [124].

portes	$\|\|\|$	$\| [liste] \|$	$\|\|$
δ	oui	oui	oui
$p \in liste$	non	oui	oui
$p \notin liste$	non	non	oui
i	non	non	non

Table 1.7 : Composition parallèle et synchronisation – LOTOS

Il faut aussi, pour être exhaustif, noter l'existence de deux opérateurs permettant d'écrire l'un le choix non déterministe sous forme concise (`choice`), et l'autre adapté à la composition parallèle (`par`). On peut se reporter à [124] par exemple pour la syntaxe et la sémantique exacte.

Les processus, lorsqu'ils se terminent ont la possibilité de transmettre des résultats (fonctionnalité). Cela correspond à l'ajout d'offres sur la porte δ. Il doit donc y avoir correspondance entre les types, voire les valeurs de deux processus composés en parallèle (voir [220]).

La fonctionnalité `noexit` dénote la fonctionnalité d'un processus ne terminant pas (`stop`). Le fait qu'un processus ait la fonctionnalité `exit` n'assure pas sa terminaison. C'est une condition nécessaire mais non suffisante. Le comportement résultant de la composition parallèle de deux sous-comportements n'est donc correct que si l'un des deux a la fonctionnalité `noexit` (ne termine pas), ou si tous les deux terminent par `exit` en offrant sur δ des offres compatibles. On dit que les fonctionnalités des comportements composés en parallèle doivent être compatibles.

Il est possible de terminer avec une offre qualifiée "`any`". Le `any` au niveau de la fonctionnalité d'un processus (porte δ) a le rôle du `?` (dans `p?x:T` par exemple) au niveau des offres sur les portes (autres que δ). `any S` permet d'indiquer qu'un processus peut retourner n'importe quelle valeur du domaine de S. La valeur effective provient donc soit de l'indéterminisme (valeur prise au hasard dans le domaine de S), soit d'un processus composé en parallèle qui retourne une valeur du type correspondant (voir le tableau 1.6).

L'opérateur » permet à un processus de se déclencher après la terminaison réussie d'un autre. Il est possible de passer des valeurs en résultat de l'un à l'autre. La fonctionnalité du premier doit être

compatible avec l'attente déclarée du second. Cette communication se fait via la porte δ et est cachée aux autres processus.

En LOTOS, les comportements peuvent être paramétrés par des portes et/ou des variables formelles (limitation au premier ordre). Des processus peuvent ensuite en appeler d'autres en instanciant les paramètres formels. La récursivité est autorisée. Notons que les sortes des paramètres doivent correspondre.

Il n'y a ni héritage, ni sous-typage autorisé entre les sortes. Ni au niveau de l'opérateur » ni à celui de l'instanciation.

[125] cite plusieurs approches pour vérifier les spécifications LOTOS : tests d'équivalences [47] (bissimulation, équivalence observationnelle, ...), transformation de programmes par application de lois algébriques, preuve de propriétés à l'aide de démonstrateurs de théorèmes, traduction de LOTOS vers les réseaux de Petri et/ou les automates d'états finis sur lesquels on peut ensuite évaluer les propriétés voulues.

Notons pour terminer que comme le fait remarquer [182] les principes d'équivalence ne sont valables qu'en l'absence de la partie statique (ACT ONE).

La spécification asynchrone impose des contraintes supplémentaires d'ordonnancement des processus destinées à éviter l'interblocage ([124]).

Les contraintes temporelles ne peuvent pas s'exprimer simplement en LOTOS. LOTOS ne traite que l'ordre relatif des événements. Il n'y a pas de temps universel.

Le tableau 1.8 résume LOTOS par rapport aux besoins exprimés plus haut.

critère	commentaires		
	statique	dynamique	global
formalité	oui (ACT ONE)	oui (Basic LOTOS (CCS/CSP))	oui, syntaxique
expressivité	++ (algébrique)	++ (amélioration CCS/CSP)	++
abstraction	+ (algébrique)	- (expl. états)	-
lisibilité	- (algébrique)	- (AP)	-
communication	synchrone+ (communications multi-directionnelles), systèmes ouverts		
concurrence	entrelacement		
temps	logique		
séparation aspects	limité (partie dynamique utilisant la statique)		
intégration	oui (opérationnelle)		
compo. //	CCS+CSP+améliorations		
réutilisation	pas directement (méthodologie)		
OO	pas directement (méthodologie)		
outils	oui ++		
vérification	vérification de modèle (sureté, vivacité), équivalences, test, animation		

Table 1.8 : Tableau récapitulatif – LOTOS

E-LOTOS [161] est en cours de normalisation par l'ISO et étend LOTOS (E-LOTOS signifiant "Extended LOTOS"). Sa sémantique est basée sur une algèbre de processus qui généralise celle de LOTOS et une partie statique fonctionnelle exécutable et plus lisible. Les extensions de E-LOTOS sont entre autres :

– modularité : définition de modules et d'interfaces, importation/exportation, visibilité,

- types de donnée avancés,
- gestion du temps dans une optique temps réel avec événements atomiques et instantanés,
- unification des fontions et des processus au niveau sémantique (les fonctions sont des processus particuliers),
- opérateur parallèle plus général (n-aire, synchronisation à n parmi m possibles - impossible en LOTOS),
- opérateur suspend/resume qui généralise $[>$, gestion des exceptions,
- typage fort des portes (y compris dans les paramètres des processus).

Il n'existe pas encore d'outillage dédié à E-LOTOS. Un compilateur d'un sous-ensemble stable de E-LOTOS est disponible, TRAIAN [244]. Il ne permet pour l'instant que des fonctionnalités de base comme l'analyse syntaxique ou la vérification du typage.

Dans cette famille d'approches, citons PSF [195] qui combine ACP [38] au lieu de CCS et CSP pour les comportements et ASF [37] au lieu d'ACT ONE pour les données. PSF est plus expressif que LOTOS (il peut en être vu comme une extension [195]) car il permet plus de liberté au niveau des synchronisations entre autres. Nous avons préféré détailler LOTOS car il est normalisé ISO.

1.4.1.3 Systèmes états transitions et formalismes basés modèle

La plupart des travaux concernant des approches basés modèle pour la statique combinent Z et des algèbres de processus comme nous avons pu le voir plus haut. Il existe cependant aussi des travaux combinant des systèmes états/transitions avec Z comme $\mu\mathcal{SZ}$, les machines à configurations ou l'Event Calculus.

$\mu\mathcal{SZ}$. $\mu\mathcal{SZ}$ [60, 59] est un formalisme développé dans le cadre du projet ESPRESS (industrie et instituts de recherche allemands). Il combine Z [184] pour les données (types et opérations) ainsi qu'une forme de Statecharts [140] pour le comportement dynamique et une logique temporelle (logique discrète temporelle d'intervalles) pour exprimer les propriétés de sûreté.

Ce formalisme met en avant les propriétés suivantes :

- modularité et réutilisation : composants communicants qui encapsulent leurs données et présentant une interface bien définie. La structure du système est donnée en utilisant les activity-charts [141, 142] ;
- communication et concurrence : communication synchrone à n processus par événements (échange de valeurs par partage de variables) ou par diffusion (par définition de types d'événements et de variables de ces types dans la partie statique), vrai parallélisme et support temporel (événements "tick") ;
- modèle sémantique sous-jacent permettant de gérer les sorties en fonctions des entrées mais aussi de l'état interne des données (donc plus expressif que les formalismes transformationnels).

La partie Z est étendue par utilisation de rôles : DATA pour l'espace des données (encapsulées) des composants, PORT pour les données partagées constituant l'interface des composants (lecture/écriture), PROPERTY pour un invariant entre variables et INIT pour leur initialisation. Le formalisme prévoit la possibilité d'extension de ces rôles dans le cadre de la méthodologie du projet ESPRESS.

Ce Z est étendu à l'aide d'une logique temporelle pour prendre en compte des propriétés de sûreté [61] qui restreint les modèles d'une spécification Z étendu (schémas dynamiques) travaillant au niveau de traces pour les modèles du schéma Z de base. C'est une logique expressive (propriétés d'états, contraintes sur la longueur des traces, contraintes de durées - mais utilisation dans $\mu\mathcal{SZ}$ d'un temps modélisé par événements -, quantification temporelle).

Les étiquettes utilisées dans les Statecharts de ce formalisme sont de la forme

$$[garde]\ evenement\ /\ action$$

mais la garde et les actions correspondent à des schémas Z. Il est possible de référer des états de la partie dynamique dans la partie statique (ce qui nuit à la séparation des aspects).

L'intégration (opérationnelle) entre aspects se fait par le calcul d'un ensemble maximal de transitions hors conflit (prédicats Z et garde Statechart vrais, états source des Statecharts disjoints). Ces transitions sont effectués en parallèle, et les changements ne sont visibles que dans l'état suivant (atomicité). Le problème qui se pose lorsqu'une même variable peut être modifiée par deux transitions est modélisée par de l'indéterminisme (il faut noter que c'est la valeur dans l'état de départ qui est utilisée). La gestion du temps se fait par événement spécial ("tick") qui n'est autorisé que si aucun n'autre événement n'est possible.

Il est possible de définir des enrichissements dans lesquels deux états du Statechart enrichi et de l'enrichissement de même nom sont identifiés et où deux opérations de la partie statique de même nom sont conjointes. Ce principe permet de définir des comportements plus ou moins abstraits puis de les raffiner. Cet enrichissement n'est toutefois que syntaxique et n'assure par forcement une équivalence (comportementale) quelconque (des conditions pour satisfaire cela ont été définies dans [101]).

Ce formalisme dispose de moyens d'animation par codage des schémas Z et utilisation de l'outil Statemate [141]. Un outil de vérification de type est aussi disponible, mais en raison de l'expressivité de ce formalisme, aucune procédure de vérification n'est disponible. Une méthodologie, basée sur les agendas a été définie [133].

Le tableau 1.9 résume $\mu\mathcal{SZ}$, par rapport aux besoins exprimés plus haut.

critère	commentaires		
	statique	dynamique	global
formalité	oui (Z)	+/- (Statecharts)	+/-, syntaxique
expressivité	+ (ensembliste)	+/- (états/transitions)	+/-
abstraction	- (Z)	+ (Statecharts)	+/-
lisibilité	+ (ensembliste)	+ (Statecharts)	+
communication	synchrone++ (à n ou par diffusion), systèmes ouverts		
concurrence	vrai parallèlisme		
temps	logique		
séparation aspects	non		
intégration	oui (opérationnelle)		
compo. //	Statecharts		
réutilisation	non		
OO	oui ? (Statecharts)		
outils	?		
vérification	?		

Table 1.9 : Tableau récapitulatif – $\mu\mathcal{SZ}$

Machines à configuration. Les machines à configuration [28, 26] sont un formalisme permettant de modéliser à la fois les aspects statiques (données), fonctionnels (opérations) et dynamiques (événements). Cette approche propose un formalisme unifié (un seul concept, les machines) mais ne permet pas vraiment la séparation des aspects puisque les aspects dynamiques et statiques ne sont pas décrits au sein de plusieurs machines puis combinés.

Ce formalisme combine une spécification des données basée modèle (à la Z) et une spécification de la dynamique à base de système de transitions étiquetés [17]. Les deux sont intégrés au sein d'une machine. Il existe une sémantique opérationnelle (construite à l'aide de la sémantique de déduction naturelle) pour les composants au sens global (données + contrôle). L'approche se situe à mi-chemin entre approches syntaxiques et sémantiques (voir les combinaisons de Z et des algèbres de processus). C'est un formalisme expressif, tout en restant cependant très intuitif (car à base de notions simples). Il a les défauts et les avantages des formalismes basés modèles au niveau des données (très implémentatoire).

Une machine est spécifiée par un modèle des données, une configuration initiale, un alphabet d'événements (émis ou reçus) et une série de transactions décrivant l'évolution de la machine. Une transaction correspond à :

- une description d'une configuration de départ servant à la fois d'état source et de garde à la transaction. Une configuration est une conjonction de prédicats (à la Z) décrivant un ensemble d'états concrets,
- un nom d'action qui sera effectuée sur la configuration,
- une description d'une configuration d'arrivée. Cette configuration décrit la postcondition de l'action,
- un éventuel événement reçu, il est intégré à la précondition de l'action (dans la configuration de départ),
- un ou plusieurs événements émis, intégrés à la postcondition.

Il est possible de spécifier des transactions purement dynamiques (sans effet sur les données, pour exprimer des invariants par exemple) ainsi que des transactions purement statiques (événement interne $\bowtie$).

Une spécificité de ce formalisme est de traiter les événements dans les pré et postconditions (donc mélangés aux données) et non pas à part. Cela augmente le niveau d'intégration du formalisme (des contraintes sur les données émises/reçues peuvent ainsi s'exprimer très simplement par un ajout de prédicat dans la configuration correspondant à l'événement) mais limite les possibilités de réutilisation séparée de parties données ou comportement. Le système de transitions correspondant à la sémantique d'une machine (sémantique opérationnelle) est particulier en ce qu'il intègre la prise en compte de la non atomicité des traitements sur les données (une série de transitions entre états) tout en la faisant correspondre à une transition unique au niveau plus abstrait de la dynamique (on parle de transaction entre configurations). Il est ainsi possible d'avoir soit un point de vue très abstrait (configurations représentant des abstractions d'états stables, et transitions correspondant à une action entre configurations) soit plus détaillé (au niveau des éléments atomiques de modification de l'espace d'état se trouvant dans les postconditions) qui sert essentiellement lors de vérifications.

La communication se fait par synchronisation, comme dans les algèbres de processus dont l'aspect dynamique s'inspire. Les événements peuvent contenir des paramètres pour l'échange de valeur entre deux machines. Le modèle de concurrence est par entrelacement. Les machines à configuration permettent de spécifier des systèmes ouverts non déterministes (mais aucun critère d'équité ne peut être donné explicitement, ou ne se trouve implicitement dans le formalisme). Le fait que la dynamique globale du système soit totalement formalisée dans un cadre opérationnel et qu'il existe un système d'historiques

basés sur plusieurs ensembles correspondant aux événement émis, reçus,... devrait permettre par exemple l'extension à un mode de communication asynchrone. Les moyens de structuration parallèle procèdent par association d'événements correspondants (émis et reçus - à la CCS) et partage de variables au niveau des espaces données globaux des compositions. Il est possible (cela ne semble pas encore avoir été fait) de modifier la formalisation de la sémantique opérationnelle pour prendre en compte différentes extensions : priorités, synchronisation à plusieurs, ... Les autres moyens de structuration sont le séquencement de machines et le raffinement des parties données (qui induit un changement de l'atomicité de l'ensemble des transitions construisant une transaction entre deux configurations).

La structure d'interprétation est un ensemble de traces potentiellement infinies. La gestion du temps se fait par hypothèse d'une horloge globale et d'une fonction d'accès à sa valeur.

Il n'existe pas encore d'outils propres aux machines, cependant des expériences sont menées à l'aide d'outils existants. La vérification des spécifications se fait actuellement par traduction [27] en Promela puis vérification avec SPIN [152]. Seule une partie toutefois est automatisée. D'autre part, en raison des faibles possibilités d'expressivité (relativement à Z) au niveau données de Promela, seule une partie des données peut potentiellement être traduite. L'effet des opérations sur les données doit aussi être décrit manuellement dans Promela. Enfin, SPIN ne permettant par la vérification de systèmes ouverts, il est nécessaire de spécifier intégralement un environnement au système, ce qui peut réduire l'intérêt de certaines preuves. Il faut enfin noter que les hypothèses d'équité ne sont pas précisées ce qui peut poser des problèmes pour la vérification des propriétés de vivacité (ceci est dû à SPIN).

Le tableau 1.10 résume les machines à configurations par rapport aux besoins exprimés plus haut.

critère	commentaires		
	statique	dynamique	global
formalité	oui (Z)	oui (transitions)	oui, syntaxique
expressivité	+ (ensembliste)	++ (transitions et transactions)	+
abstraction	- (Z)	+ (système de transitions abstraites)	+/-
lisibilité	+ (ensembliste)	+ (transitions)	+
communication	synchronisation		
concurrence	entrelacement		
temps	logique, potentiellement réel		
	(hypothèse d'horloge globale à valeur accessible)		
séparation aspects	non- (événements dans pré/post conditions)		
intégration	oui (sémantique de traces)		
compo. //	-, implicite, correspondance évènements à la CCS		
réutilisation	non		
OO	non		
outils	non, expérimentation réduite avec PROMELA/SPIN		
vérification	vérification de modèle (réduite)		

Table 1.10 : Tableau récapitulatif – Machines à configuration

Event Calculus. L'Event Calculus[7] [253, 256, 254] est un formalisme qui combine des machines à base d'états communicantes pour la partie dynamique avec du Z pour la partie statique. Les machines ont une représentation graphique mais sont décrites sémantiquement à l'aide de Z.

La communication entre machines se fait par événements partagés (synchronisation), échanges de valeurs et changement simultané d'état. Les machines changent potentiellement d'état lorsqu'un événement survient. La sémantique de la concurrence est l'entrelacement des événements. Les préconditions des opérations dans les schémas sont utilisées en tant que gardes autorisant ou pas une transition (et l'événement associé). Les transitions regroupent donc un événement et une opération de schéma. Le comportement global des systèmes est calculé à partir de ceux de ses composants, tout cela dans un cadre Z.

La spécification d'un système se fait par inclusion des schémas correspondant à la sémantique de base (définissant les notions de machines à états, d'états et d'événements). Les données sont ensuite spécifiées comme habituellement en Z. Enfin, les différentes machines ainsi que leur composition et la liaison entre dynamique et opérations statique est faite. Il s'agit d'un formalisme très intuitif et lisible mais ne permettant pas de donner des propriétés très abstraites des données.

Ce formalisme prend à la base la possibilité de modéliser des contraintes temporelles (toujours dans le même cadre, en utilisant un schéma Z et une modélisation avec des entiers et des ticks). Il existe des extensions pour modéliser les systèmes hybrides (temps discret et continu) à base de machines évoluant dans un temps continu [255]. Ces extensions se basent toutefois sur l'hypothèse d'une extension de Z avec des réels, pour laquelle des techniques existent.

Le tableau 1.11 résume l'Event Calculus par rapport aux besoins exprimés plus haut.

critère	commentaires		
	statique	dynamique	global
formalité	oui (Z)	oui (transitions)	oui, sémantique
expressivité	+ (ensembliste)	+ (transitions)	+
abstraction	- (Z)	- (expl. états)	-
lisibilité	+ (ensembliste)	+ (transitions)	+
communication	synchrone, systèmes ouverts		
concurrence	entrelacement		
temps	logique, réel possible		
séparation aspects	non (liaison forme opérations/dynamique)		
intégration	oui (opérationnelle)		
compo. //	implicite évènements correspondants		
réutilisation	non		
OO	non		
outils	?		
vérification	?		

Table 1.11 : Tableau récapitulatif – Event Calculus

[7] Ce formalisme n'a pas grand chose à voir - à part son nom - avec le formalisme logique utilisé entre autres en intelligence artificielle pour représenter les actions et leurs effets (au départ dans un cadre bases de données).

1.4.1.4 Systèmes états-transitions et formalismes algébriques

Plusieurs approches ont cherché à combiner la lisibilité des systèmes états-transitions et l'expressivité et l'abstraction des formalismes algébriques, parmi ces approches, citons SDL (et les Message Sequence Charts), CASL-Chart et les Types Abstraits Graphiques.

SDL. SDL [66, 105, 222] dispose de constructions pour la structuration du système, la définition de comportements de processus, des interfaces et des liens de communication. Il dispose aussi de moyens d'abstraction, d'encapsulation de module et de raffinement. L'ensemble de ces constructions, conçues pour correspondre à un vaste éventail de systèmes de télécommunications font que SDL est couramment utilisé aujourd'hui dans l'industrie. De nombreux outils sont disponibles.

SDL (Specification and Description Language) est un langage de spécification muni d'une sémantique formelle développée et standardisée par ITU-T[8]. Cette sémantique s'appuie sur un modèle de machines à états finis étendues (pour la description du comportement du système) avec des éléments de types abstraits de données. Depuis sa première standardisation en 1976, SDL a un peu évolué vers une vraie technique de description formelle, SDL-88, qui est définie selon la recommandation ITU-T Z.100 [66]. Des développements récents comme le non déterminisme et l'orienté objet ont conduit à la définition de SDL-92 [105]. SDL est muni de deux notations équivalentes, GR (Graphical Representation) et PR (Phrase Representation). Comme la représentation GR est plus facile à comprendre, la représentation PR est essentiellement utilisée comme format d'interface. L'équivalence entre les représentations GR et PR est traitée dans [66, 105]. SDL est principalement utilisé pour la spécification de protocoles et de services de télécommunication, mais il peut être utilisé de façon plus générale pour les systèmes réactifs. Plusieurs outils sont disponibles pour SDL, tels que ObjectGEODE de Verilog et Tau de Telelogic.

Une spécification SDL décrit un *système* à l'aide de plusieurs machines à états finis (étendues) communicantes. Ces machines échangent des messages appelés *signals* qui peuvent véhiculer des valeurs de données typées (et donc fournir un moyen simple d'échange de données entre machines). Les machines gèrent le concept de *processus*. Un processus est une description commune d'un comportement partagé par ses *instances*. Chaque instance de processus a un identificateur de processus, unique et différent (PId). Ces PIds sont particulièrement utiles pour échanger des messages entre instances de processus. Les processus possèdent des valeurs de données (d'où le terme "étendues" pour les machines). L'échange de données entre processus est possible par partage de signaux ou de variables. Le système peut aussi communiquer avec l'environnement extérieur qui est supposé se comporter comme une instance de processus SDL (par exemple, il est censé avoir un PId différent de ceux des autres composants du système).

SDL fournit des types de base (appelés *sorts* en SDL) utilisables dans la description des comportements de processus. Ils comportent les types usuels comme les entiers ou les booléens mais aussi le temps (pour modéliser les minuteurs), la durée et les PId (pour les instances de processus). Les sortes définies par l'utilisateur peuvent être définies au moyen de *types abstraits de données* : constantes (appelées *literals*), signatures typées des opérations (appelées *operators*) disponibles pour la sorte et les équations (appelées *properties*) pour leur sémantique (modèle identique à celui de la partie ACT ONE de LOTOS).

Les blocs de SDL fournissent un moyen simple de structurer les spécifications (figure 1.6).

Le système est composé de plusieurs blocs (*blocks*) (locaux ou différés - références) connectés. Il y a des sous-structures de blocs (*block substructures*) et des diagrammes de blocs (*block diagrams*). Ils diffèrent en ce que les diagrammes de blocs apparaissent à la fin du processus de décomposition. Tandis

[8] International Telecommunication Union, qui a remplacé le International Telephone and Telegraph Consultative Committee (CCITT).

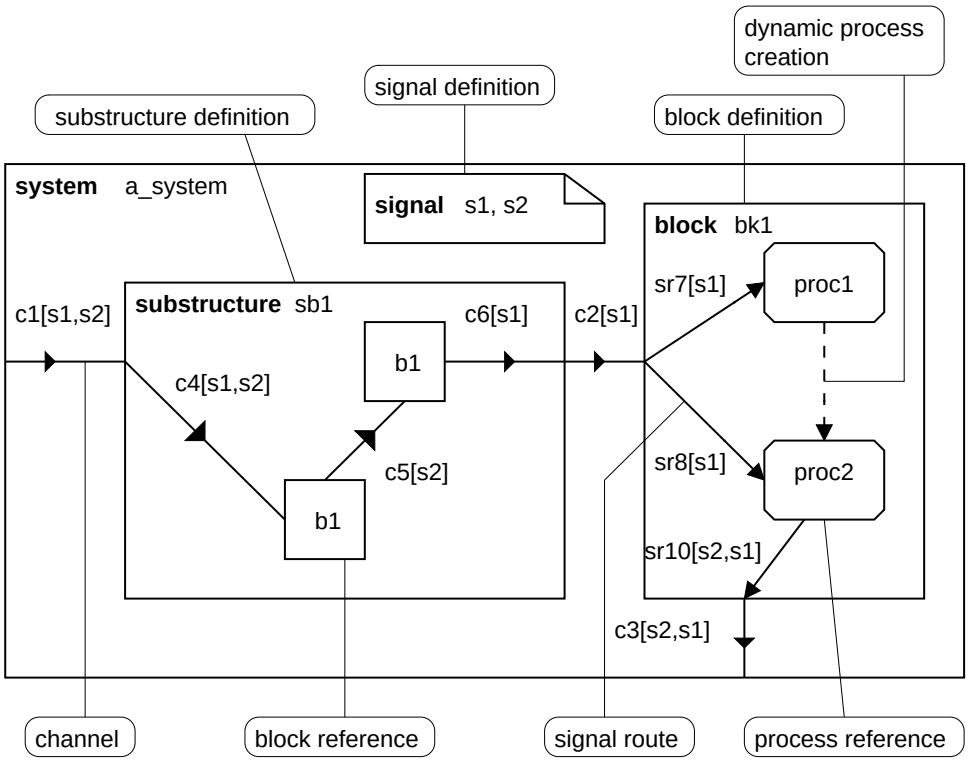


Figure 1.6 : Concepts de structuration – SDL

que les sous-structures de blocs peuvent contenir d'autres sous-structures de blocs ou des diagrammes de blocs, un diagramme de blocs peut seulement contenir des processus.

Les connexions sont modélisées par des canaux (*channels*). Les canaux ne peuvent pas relier plus de deux blocs. Ils peuvent transporter des signaux définis dans leur liste (*signal list*). Les canaux sont uni- ou bi-directionnels. Une liste de signaux est associée à chaque direction. Les canaux entre processus sont appelés des routes (*signal routes*). Il est possible de raffiner les canaux en blocs et canaux.

Des points de connexion (*connection points*) relient les canaux de blocs avec leur superstructure englobante. De même que les blocs, les canaux peuvent être décomposés en sous-composants (blocs et canaux). Ceci peut être utilisé par exemple dans la modélisation de media peu fiables.

De même que pour les processus, les blocs (incluant le système) et les canaux fournissent une définition commune partagée par leurs instances. SDL a une règle de portée de nom simple : toute définition est accessible dans le bloc courant et tous ses sous-blocs. Il faut noter que, dans les blocs, il est possible d'utiliser les définitions ou références de spécifications distantes. Les références sont habituellement des indicateurs de places. Ils fournissent aussi une structuration plus lisible des systèmes et des blocs en séparant les niveaux d'abstraction.

SDL permet de donner un nombre initial et un nombre maximal d'instances dans les références de processus.

Les comportements de processus sont décrits à l'aide de graphes de processus. Une instance de processus peut être dans plusieurs états (*states*) où elle peut recevoir ou envoyer différents ensembles de signaux. Les instances de processus sont créées (à l'initialisation du système ou dynamiquement par une

autre instance de processus du même bloc) dans un état spécial appelé l'état initial (*initial state*) - les états sans nom dans le graphe de processus. La sémantique du comportement du système est donnée en termes des comportements de ses instances de processus.

La communication des instances de processus est *asynchrone* et utilise des signaux (figure 1.7). Les émetteurs ne se bloquent pas à l'envoi de signaux. Les récepteurs ont un buffer illimité où les signaux valides (i.e. les signaux que le processus peut traiter dans un état) sont conservés jusqu'à consommation ou élimination. Si un message peut être traité dans un état courant, l'instance de processus initialise une transition, consomme le signal, effectue une activité optionnelle et va dans l'état cible. Si le message ne peut être traité dans l'état courant, l'instance l'élimine (il y a aussi moyen de sauvegarder le message pour plus tard). Les buffers, aussi appelés *input ports*, ont un comportement "first-in/first-out" (sauf pour les signaux sauvegardés).

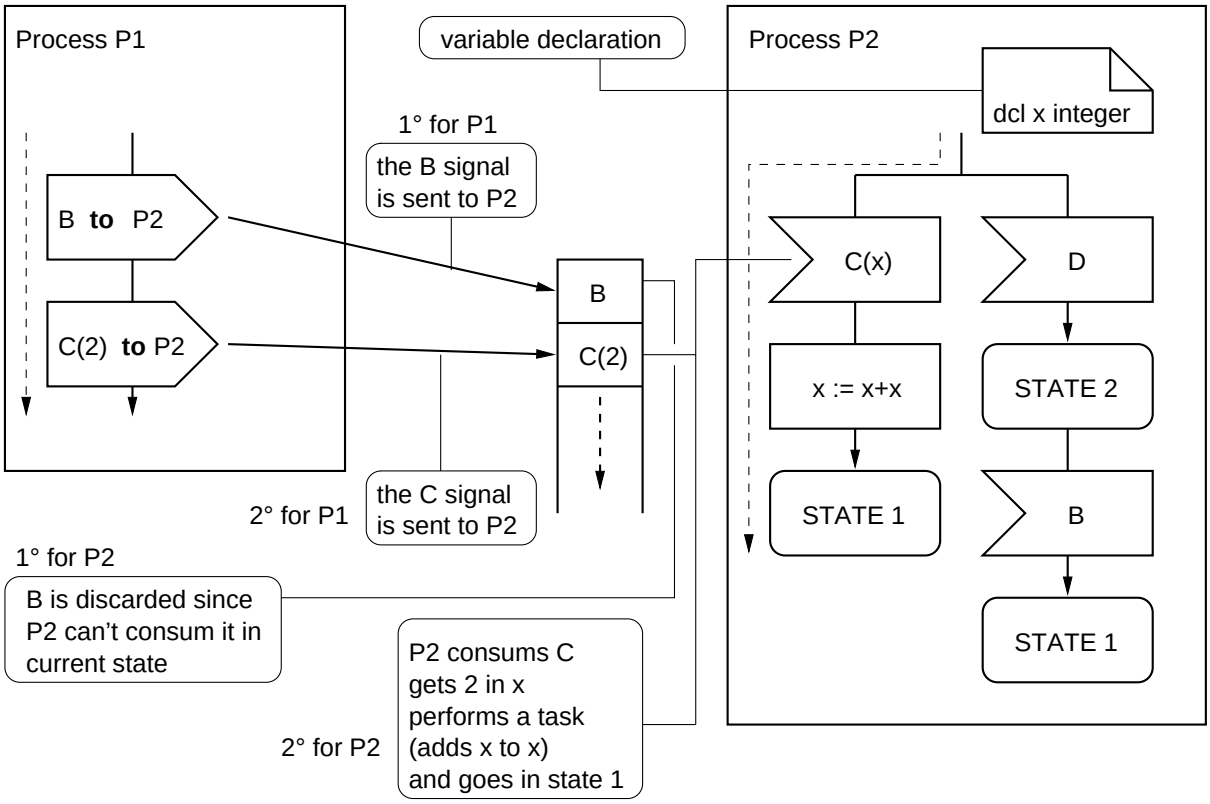


Figure 1.7 : Sémantique des signaux – SDL

Les signaux sont un moyen d'échanger des valeurs. Une instance de processus p peut envoyer des signaux (i) de manière implicite à l'unique instance de processus qui lui est reliée via la structure du système, (ii) à toutes les instances de processus reliées à un certain chemin de signal (via les routes et les canaux de signaux), ou (iii) à une instance spécifique de processus en utilisant les identificateurs de processus (puisqu'il y a unicité d'identificateur pour un processus donné). Les mots-clés de destination peuvent être : *self* (l'instance même de processus), *sender* (l'instance de processus qui a envoyé le dernier message), *offspring* (la dernière instance de processus créée par p), ou *parent* (l'instance de processus qui a créé p).

Il est possible de partager des valeurs soit de manière restreinte au bloc (view/reveal), soit en guise

de raccourci (avec export/import, raccourci modélisé à l'aide de signaux).

Il y a des transitions continues (quand une garde est vraie). Il y a des minuteurs (avec des sortes Time et Duration avec les propriétés de la sorte Real - nombres à décimale flottante) et des moyens pour lire l'heure, positionner ou mettre à jour des minuteurs, réagir à l'expiration de minuteurs et examiner leur statut.

SDL permet de définir des ensembles via le générateur *powerset*. Les opérations usuelles sur les ensembles (`incl` pour ajouter un élément, `del` pour enlever un élément, `in` pour tester la présence d'un élément dans un enesemble, et `empty` pour l'ensemble vide) sont disponibles (entre autres).

Souvent, les types de base doivent être étendus pour définir des opérateurs utilisés dans le processus qui ne sont pas définis dans les types de base. Ceci peut être fait en utilisant le concept d'héritage (*inheritance*) de SDL. Tout ce qui est défini dans le type parent est hérité, et on peut ajouter des définitions avec le mot-clé `adding`. L'héritage partiel (ou renommage) peut être spécifié sur les constantes et/ou les opérateurs.

On peut utiliser le concept SDL *syntype* qui permet de définir (renommer) une sorte avec un ensemble restreint (ou ici égal) de valeurs par rapport au type sur lequel c'est basé.

Dans SDL, on peut utiliser un terme spécial `error!` (inclus implicitement dans chaque sorte). Lorsque un terme `error!` est retourné, une erreur dynamique est levée. Ce sera utilisé lorsque un ordre n'est pas trouvé dans l'ensemble des ordres.

Les opérateurs `*!` sont utilisés dans les définitions de structures de type. L'opérateur *Make!(...)* construit une structure à partir de ses champs, un opérateur tel que *IDExtract!(record)* est utilisé pour extraire la valeur d'un champ *ID* de la structure, et un opérateur tel que *IDModify!(record,value)* est utilisé pour remplacer la valeur d'un champ *ID*.

Le tableau 1.12 résume SDL par rapport aux besoins exprimés plus haut.

critère	commentaires		
	statique	dynamique	global
formalité	oui (type ACT ONE)	oui? (transitions)	non?
expressivité	+ (algébrique)	++ (transitions, création/mort, PID)	++
abstraction	+ (algébrique)	+ (systèmes de transitions symboliques)	+
lisibilité	- (algébrique)	++ (états/transitions)	++
communication	asynchrone, systèmes ouverts		
concurrence	vrai parallélisme		
temps	réels (timers)		
séparation aspects	non		
intégration	oui? (opérationnel)		
compo. //	correspondance implicite, structure de bloc, + PID		
réutilisation	oui (blocs, processus)		
OO	oui partiellement en SDL92		
outils	oui, réduits ?		
vérification	animation, test, vérification de modèle ?		

Table 1.12 : Tableau récapitulatif – SDL

SDL 92 définit de nouvelles orientations. Une forme d'héritage existe pour les sortes. Les signaux, procédures et sortes sont maintenant des types. Il est possible de spécifier des composants génériques instanciables. La prise en compte de l'indéterminisme est améliorée, par la possibilité d'utiliser des

transitions spontanées, ainsi que des “undecided values” (`any(sort)`). Il est dorénavant possible de donner la sémantique des opérations sous forme de procédures (sans toutefois d’effets de bord). Enfin, la liaison avec d’autres formalismes pour la spécification des données est possible par la notion d’externalité du corps des procédures.

MSC. Les Message Sequence Charts [103, 155] (MSC) sont un formalisme qui par de nombreux aspects est complémentaire à SDL. De façon analogue, les MSC ont une forme textuelle (MSC/PR) et une forme graphique (MSC/GR), la plus connue. Les MSC comprennent des concepts d’instance, message, environnement, action, timer, création et terminaison de processus ainsi que des mécanismes conditionnels et structurels. Dans leur version la plus récente (MSC’96 [104]), il prennent en compte des concepts de composition (dont la structuration hiérarchique) et des concepts orientés objet. Enfin, il faut noter qu’il existe une sémantique algébrique des MSC [194].

SDL donne une réprésentation complète du comportement des processus, mais une représentation indirecte de la communication entre processus (blocs et canaux). À l’inverse, les MSC s’intéressent à la communication entre composants et leur environnement par le biais d’échanges de messages. Les MSC ne donnent en général qu’une trace et ne couvrent donc qu’une partie du comportement des processus. Les MSC ont essentiellement un but de spécification de propriétés particulières des systèmes (ils sont d’ailleurs une représentation intuitive de formules simples de logique temporelle) et peuvent ainsi servir à définir des séquences de tests pour des spécifications écrites dans d’autres langages (SDL ou bien encore UML).

CASL-Chart. Les Statecharts [140] sont un formalisme très lisible à base d’états et de transitions permettant la composition parallèle et la structuration (hiérarchisation) des automates. Une forme de Statecharts est utilisée dans UML. Leur inconvénient majeur reste leur nombre important de sémantiques. Toutefois, il faut reconnaître qu’en général c’est [142] qui est cité (mais les deux sémantiques qui y sont données ne sont pas définies formellement). Les autres articles scientifiques concernent en général une extension ou une tentative de formalisation des Statecharts. Les mécanismes de synchronisation sont basiques (synchronisation par diffusion) et peu naturelles.

Ici nous ne discutons que de leur combinaison récente avec des spécifications algébriques, CASL-Chart [230]. Ce travail s’inscrit dans la possible extension de CASL aux systèmes réactifs, COFI/CASL-Reactive. La sémantique globale procède par combinaison (selon une approche générale proposée par [21]) plutôt que traduction dans un seul et même formalisme. La sémantique (ensemble d’exécutions, fonction partielle entre un flot d’entrées et un ensemble de flots de sortie, prise en compte de micro transitions entre deux états stables) est donnée en précisant celles de [142] (qui n’étaient pas formelles) et en y incluant l’aspect algébrique (par évaluation des expressions dans ce cadre). Le grand avantage de cette approche est son pragmatisme et permet d’utiliser très simplement un outillage existant (STATEMATE) en remplaçant son évaluateur par un évaluateur algébrique (un moteur de réécriture concrètement, mais cela impose l’emploi de spécifications à sémantique initiale - l’auteur cite les spécifications conditionnelles - dans la partie CASL). Cette approche se place aussi dans l’optique de formalisation d’UML en remplaçant par exemple OCL (basé sur des modèles à la Z) par CASL. Ses inconvénients sont qu’elle ne sépare pas vraiment données et comportement (mais leur spécification oui) et qu’elle se base sur les Statecharts qui ne sont pas un formalisme très simple (par rapport aux automates) ce qui peut limiter les possibilités de preuves et d’animation.

Le tableau 1.13 résume les CASL-Chart par rapport aux besoins exprimés plus haut.

critère	commentaires		
	statique	dynamique	global
formalité	oui (CASL)	oui (une sémantique des Statecharts)	oui, syntaxique
expressivité	+ (algébrique)	+ (Statecharts)	+
abstraction	+ (algébrique)	+ (Statecharts)	+
lisibilité	- (CASL)	++ (Statecharts)	+
communication	Statecharts, systèmes ouverts		
concurrence	vrai parallélisme		
temps	logique		
séparation aspects	non		
intégration	oui (opérationnelle)		
compo. //	Statecharts		
réutilisation	non		
OO	oui ? (Statecharts)		
outils	possible utilisation Statemate par remplacement évaluateur		
vérification	animation ?		

Table 1.13 : Tableau récapitulatif – CASL-Chart

TAG. L'approche des Types Abstraits Graphiques (TAG) [15] consiste à associer un système de transitions à la spécification algébrique d'un type de données. Il existe deux types de composants TAG : les composants séquentiels et les composants concurrents.

Pour chaque composant, deux vues sont considérées : une vue dynamique et une vue fonctionnelle (statique). La vue dynamique est un système de transitions abstrait, c'est-à-dire un ensemble fini d'états et de transitions abstraits. Les états sont abstraits dans le sens où ils représentent une abstraction d'un ensemble potentiellement infini d'états concrets. Les transitions sont abstraites dans le sens où elles sont étiquetées par des termes non clos. Ces termes correspondent à des appels d'opérations avec des variables libres et des gardes. Ces systèmes de transitions permettent d'échapper à l'explosion du nombre d'états et de transitions et améliorent ainsi la lisibilité des comportements dynamiques. La vue statique est une spécification algébrique d'un type de données abstrait partiel.

Un point très important est que cette approche se sert de la vue dynamique pour dériver une partie des axiomes de la vue statique. La vue dynamique représente une vue graphique d'une relation d'équivalence partielle sur le type de données de la vue statique. L'inconvénient est que les vues statiques et dynamiques sont liées et qu'il n'y a pas de séparation des aspects possible. En échange, ce principe assure la cohérence des deux vues au sein d'un tout, et permet d'avoir des spécifications dans un style opérationnel où les axiomes peuvent être orientés et transformés en règles de réécriture.

En ce qui concerne les composants concurrents et communicants, le produit synchronisé [17] est étendu aux systèmes de transitions abstraits et utilisé avant la génération des axiomes de la partie statique. Il permet l'expression des règles de synchronisation et d'échange de valeurs entre composants.

Le tableau 1.14 résume les TAG par rapport aux besoins exprimés plus haut.

1.4.1.5 Autres systèmes états transitions

D'autres approches utilisent des systèmes états transitions, avec un formalisme ad hoc (impératif ou objet) pour les données.

critère	commentaires		
	statique	dynamique	global
formalité	oui (type abstrait partiel)	oui (système de transitions)	oui, opérationnel
expressivité	+ (algébrique)	+ (système de transitions)	+
abstraction	+ (algébrique)	+ (système de transitions abstraites)	+
lisibilité	- (algébrique)	+ (système de transitions)	+
communication	synchrone, systèmes ouverts		
concurrence	entrelacement		
temps	logique		
séparation aspects	non (liaison forte système de transitions / spécification algébrique)		
intégration	oui (opérationnelle)		
compo. //	produit synchronisé		
réutilisation	non		
OO	partiellement		
outils	atelier [12] (édition, génération de code, traduction vers LP)		
vérification	par traduction vers LP, possibilités d'animation		

Table 1.14 : Tableau récapitulatif – TAG

Estelle. Le langage Estelle [153] est basé sur le concept bien accepté d'automate communicant non déterministe pour la partie dynamique. Ces automates sont étendus dans le sens où les transitions sont étiquetées non seulement par des consommations d'entrées mais aussi des productions de sorties et des actions sur les données. D'autre part, ces automates sont symboliques. Le modèle sémantique prend en compte un état de l'automate et un état des données. La communication est asynchrone et se fait par échange de messages sur des canaux de communication bidirectionnels attachés aux ports de communication des composants (appelés modules). Il existe un buffer associé à chaque point d'interaction. Il est possible de connecter plusieurs canaux sur un même point d'interaction (queue commune, au plus une par module). La description des données en Estelle se fait par l'intermédiaire d'un langage dérivé du Pascal, étendu pour prendre en compte l'envoi de messages, la connexion, l'attachement, la création, la destruction de fils, et des constructions de type itératives pour gérer les types ordinaux et les ensembles de modules et de connexions.

La structuration en Estelle se fait par construction d'une hiérarchie de modules. Il est possible d'avoir soit une vraie concurrence (entre les fils d'un module déclaré *process*) soit un mode par entrelacement (fils d'un module déclaré *activity*). Une activity ne peut qu'être structurée en activity. Un process peut être structuré en process ou activity.

Un module (parent) peut créer et détruire d'autres modules (fils). Parents et fils peuvent communiquer par canaux ou partage de variables (entre un fils et un père uniquement, pas entre deux fils).

Estelle permet la description des systèmes à différents niveaux d'abstraction, et il existe des outils pour procéder au raffinement.

Il existe des transitions spontanées en Estelle, et il est possible de leur associer des délais. Il est possible d'associer des priorités aux transitions. Estelle se base sur des automates potentiellement non déterministes. C'est à l'implémentation de résoudre cet indéterminisme. Les gardes peuvent prendre en compte : entrée courante en tête de la queue, état courant, prédicats sur les variables du contexte et les paramètres du message courant, état des timers et priorité de la transition.

Le modèle d'Estelle est donc un modèle très attractif mais relativement complexe en raison des queues et de la gestion du temps et des priorités. Estelle est plus proche d'un langage (de programmation) que d'un formalisme de spécification abstrait. D'autre part, l'utilisation du Pascal pour les données conduit aussi soit à avoir des données très simples, soit à avoir une implémentation (et pas une abstraction) des données. D'autre part, la gestion des connexions est complexe : différence entre attachement (connexion d'un point d'interaction du père à un point du fils) et connexion (par le père, entre deux points de deux fils), pas abstraite et trop proche d'une implémentation.

Le tableau 1.15 résume Estelle par rapport aux besoins exprimés plus haut.

critère	commentaires		
	statique	dynamique	global
formalité	non (Pascal)	oui (machines étendues)	non, syntaxique
expressivité	+/- (Pascal étendu)	++ (machines Estelle)	+/-
abstraction	- (Pascal)	+ (machines)	+/-
lisibilité	++ (Pascal)	+ (machines Estelle)	+
communication	asynchrone canaux bidirectionnels		
concurrence	vrai parallélisme ?		
temps	logique, réel avec timers		
séparation aspects	non		
intégration	oui (opérationnel)		
compo. //	hiérarchie modules, + création/destruction		
réutilisation	non		
OO	oui (statique en Object-Z)		
outils	peu		
vérification	animation, prototypage		

Table 1.15 : Tableau récapitulatif – Estelle

UML. UML [249] est une notation pour l'analyse et la conception à objets. Une spécification UML peut utiliser jusqu'à 11 différents types de diagrammes décrivant différents aspects des systèmes. Dans le cadre qui nous intéresse, nous citerons principalement :

- pour les aspects statiques : diagrammes de classes (attributs),
- pour les aspects dynamiques : diagrammes de cas d'utilisation, diagrammes de séquence (pour la spécification ou l'expression de propriétés), diagrammes états-transitions (du type Statecharts), diagrammes d'activité,
- pour la composition : diagrammes de classes (relations entre classes), diagrammes de collaboration.

UML est très expressif mais n'est pas défini formellement ce qui pose de nombreux problèmes liés entre autres au sens exact des diagrammes (problèmes d'incohérences entre les interprétations du spécifieur et d'une personne chargée d'implémenter les classes par exemple). [231] a ainsi référencé 31 problèmes différents rien qu'au niveau de la sémantique dynamique d'UML. Le problème de la cohérence de la combinaison des diagrammes [19] est aussi important. Toute forme de vérification des spécifications UML est réduite ou impossible. Enfin, la génération de code à partir de diagrammes UML est très réduite. Elle ne concerne que les diagrammes de classe, et ne permet d'engendrer que les interfaces et signatures de méthodes si le spécifieur ne désire pas documenter l'ensemble (ou du moins une grande

partie) des diagrammes avec du code. La seule exception notable est Rhapsody[9] qui, en permettant de donner la sémantique des opérations dans les diagrammes d'activités, est capable d'engendrer un code complet. UML est donc principalement pour l'instant un "environnement syntaxique" dans laquelle seule la syntaxe graphique est vraiment standard [19].

Il existe actuellement de nombreuses tentatives de formalisation d'UML [229, 119], en général par restriction à une partie de la notation. Nous pensons que l'un des intérêts principaux d'UML dans le cadre qui nous intéresse est d'être utilisé en amont d'une méthode formelle mixte. Cela consiste à choisir une sous-partie d'UML et de lui associer les concepts de la méthode formelle. De cette façon, la lisibilité du formalisme mixte est amélioré.

Le tableau 1.16 résume UML par rapport aux besoins exprimés plus haut.

critère	commentaires		
	statique	dynamique	global
formalité	non	+/- (Statecharts, activity charts, MSC)	non, syntaxique
expressivité	+/- (pas corps méthodes)	++ (différents diagrammes)	+
abstraction	- (implémentation objet)	+ (Statecharts)	+/-
lisibilité	++ (objet)	++ (UML)	++
communication	synchrone, asynchrone, systèmes ouverts, diverses extensions		
concurrence	vrai parallélisme		
temps	logique		
séparation aspects	oui		
intégration	non		
compo. //	composition lié à la statique ?, Statecharts		
réutilisation	oui +		
OO	oui ++		
outils	très limités (syntaxique, génération code réduite))		
vérification	partiellement (recherche)		

Table 1.16 : Tableau récapitulatif – UML (version 1.3)

Approches réactives basées Statecharts. Une approche proche des Statecharts (sans transition entre niveau et avec compositionnalité), mais plus formelle, est celle d'Argos [188, 159]. Argos est modulaire, synchrone (voir plus haut la définition) et les comportements (réactifs) sont définis par des machines états/transitions dans lesquelles les sorties sont liées au transitions. Les machines doivent être déterministes. La communication entre composants est basée sur un synchronisme par diffusion non bloquant (comme les Statecharts). Argos gère le temps selon l'approche des langages synchrones Lustre, Esterel et Signal (plusieurs horloges, pas de temps global). Un environnement pour la spécification en Argos, Argonaute existe. Une extension d'Argos aux systèmes hybrides (temps continu) existe. La vérification se fait par compilation dans un système de transitions étiquetées et la connexion à d'autres outils comme ceux d'Esterel, ou encore Aldébaran [109] pour les équivalences ou Kronos [270] pour la prise en compte du temps. Argos est très lisible mais constitue essentiellement une interface (sa combinaison avec Lustre et Esterel a été étudiée) aux langages synchrones : peu de données (pas évoluées) et mécanismes de

[9] Une version de démonstration de Rhapsody est disponible : `http://www.ilogix.com/modeler.htm`.

synchronisation simples (correspondance d'événements). D'autres approches du même type (systèmes réactifs, vraie concurrence, temps réel) existent. De ce point de vue, citons BDL [259] qui a un ensemble de compositions particulièrement expressif et s'intègre bien dans un développement orienté objet (UML) comme les Statecharts (qui en font partie) ou SDL (qui est utilisé conjointement avec UML).

1.4.1.6 Réseaux de Petri de haut niveau

Les réseaux de Petri (RdP) sont un des formalismes les plus anciens et expressifs. Ils sont aussi très lisibles. Ils permettent l'expression de différents modes de communication (synchrone, asynchrone, causalité entre événements). Des propositions récentes ont été faites pour permettre la structuration (composition parallèle synchrone par fusion de transitions ou de places, asynchrone par réalisation d'un protocole) de RdP, la modularité et la prise en compte d'aspects orientés objets (identité d'objet, création dynamique d'objets et héritage). [33] compare les deux approches consistant à avoir des RdP pour décrire le comportement des objets (ce qui permet la structuration des RdP suivant des concepts objets, et l'expression de concurrence intra-objet) et à avoir des objets dans les RdP (un type de coloration, ce qui est pratique pour la création/destruction dynamique d'instances et la définition de la structure du système) et propose une approche unifiée.

Les extensions les plus intéressantes des RdP concernent les RdP de haut niveau [157, 35, 34, 58, 32, 172, 42, 40, 41] (ou colorés, ou algébriques quand il s'agit de termes dont la sémantique est donnée algébriquement) dans lesquels il est possible de prendre en compte la coloration dans des gardes associées aux transitions et d'effectuer des transformations (par exemple algébriques) sur les jetons colorés. La sémantique de tels systèmes est opérationnelle (et nécessite une sémantique initiale pour la partie statique, par exemple OBJ3 pour [35]). Certaines approches permettent l'expression de la vraie concurrence par la définition d'une sémantique de pas. L'approche de [216] est une tentative de fournir un cadre général (théorie des catégories) aux RdP de haut niveau.

Il est possible d'effectuer des vérifications sur les RdP toutefois cela s'avère assez complexe. Cela nécessite soit un travail sur les invariants de marquage des places, soit le calcul d'un automate de couverture (pouvant exploser) de façon à utiliser la vérification de modèle. En conclusion, on peut être tenté de dire que d'un point de vue utilisation les RdP restent proches d'UML et sont plutôt adaptés à l'analyse et aux premières phases de la conception. C'est particulièrement le cas de formalismes intégrant les RdP avec d'autres approches ou utilisant les RdP pour combiner plusieurs approches ([132] les utilise par exemple pour combiner automates, programmes parallèles et SDL).

Les avantages des RdP par rapport à des formalismes graphiques tels que des systèmes de transitions plus simples et disposant de bons moyens de structuration (produit synchronisé [17] par exemple) concernent essentiellement leur lisibilité et intuitivité. Ils sont outillés [73] et beaucoup utilisés en industrie.

1.4.1.7 Logique temporelle et algébrique

Il existe des approches qui étendent les spécifications algébriques par de la logique temporelle mais le but essentiel est un gain d'expressivité destiné à réduire les modèles de la spécification [183]. Il existe cependant aussi des approches où la combinaison est destinée à faire de la spécification mixte. À notre connaissance, ces approches sont peu nombreuses. Ceci peut s'expliquer par la prédominance de Unity et TLA (approches homogènes) et le fait que la logique soit plus pratique pour exprimer des propriétés que la spécification.

1.4.1.8 Conclusion

L'approche hétérogène fait intervenir différents formalismes pour exprimer les différents aspects. Il peut s'agir d'algèbres de processus, de systèmes de transition ou réseaux de Petri pour les aspects dynamiques, et de formalismes basés modèles (Z, B) ou propriétés (spécifications algébriques, logiques) pour les aspects statiques. Plusieurs combinaisons ont été présentées. L'approche hétérogène permet une bonne expressivité puisqu'il est possible de choisir les formalismes les mieux adaptés à tel ou tel aspect. Son problème principal concerne l'intégration et la définition de critères de cohérence entre formalismes différents. Cette approche permet la réutilisation d'outils existants, soit sur les aspects pris séparément, soit sur l'ensemble lorsqu'une sémantique globale est définie. Ces spécifications sont celles qui remportent le plus de succès (parmi les spécifications mixtes) dans le milieu industriel (UML, SDL, et dans une moindre mesure $\mu\mathcal{SZ}$).

1.4.2 Approches homogènes

Les approches homogènes utilisent un seul formalisme pour exprimer l'ensemble des aspects des systèmes. Il s'agit des formalismes algébriques, logiques et des algèbres de processus.

1.4.2.1 Formalismes algébriques

[83] propose une classification dans laquelle les approches algébriques pour la spécification mixte ([83] parle de processus, systèmes concurrent et/ou objets) peuvent être rangées dans trois groupes :

- la partie statique est spécifiée en utilisant une approche algébrique, et est utilisée par une partie dynamique spécifiée dans une approche différente (algèbres de processus [47, 195], logiques temporelles [183], Statecharts [230]). Une partie représentative de ces approches est décrite ailleurs (formalismes non homogènes).
- la partie dynamique est spécifiée comme la statique dans le cadre algébrique (axiomatiquement), il y a donc formalisme homogène [23, 200, 167]. Le problème principal reste la modularité de telles spécifications (oui pour [23], difficilement pour [200], aucune - du point de vue de composition de composants - pour [167] même s'il est possible d'avoir une modularité de base associée à des groupes de propriétés).
- les types de données de la partie statique peuvent eux-mêmes évoluer dynamiquement (types de donnée "dynamiques") [100, 272, 164].

Nous présentons des exemples de ces deux dernières catégories dans la suite. Une taxinomie plus détaillée complémentaire à [83] est donnée dans [22] et [20] qui prend aussi en compte dans ce cadre les approches dans lesquelles le formalisme algébrique est utilisé à un méta-niveau (définition ou sémantique). Nous pensons que ces approches (CCS, CSP, ACP) ne sont pas mixtes en soi et ne les traitons pas. Des extensions de certains de ces formalismes sont par contre présentés plus bas dans le cadre des formalismes homogènes basés sur les calculs et les algèbres de processus étendus.

LTL. Cette approche est représentative des approches exprimant la dynamique dans un cadre algébrique, ce qui permet lorsque les données le sont aussi, d'avoir un cadre homogène de spécification de systèmes mixtes. Cette approche est proche de [200] en ce qui concerne la classification de [83].

LTL (Labelled Transition Logic[10]) [23, 83] est composée d'une partie théorique (le formalisme LTL) ainsi que d'une partie méthodologique (méthode et outils permettant la spécification et la composition de composants concurrents dans un cadre algébrique habituellement référée sous le nom SMoLCS [24]).

LTL différencie les types de données de base (structures utilisées par le système) des composants du système. Ces types sont spécifiés en utilisant une spécification algébrique usuelle (avec sémantique initiale). En ce qui concerne les composants, une spécification algébrique est créée et c'est seulement ensuite qu'elle est rendue dynamique. Ceci ne permet pas vraiment la séparation entre aspects statiques et dynamiques. Les états d'un composant dynamique sont donnés par un ensemble de constructeurs (un par état). Tout composant dynamique inclut une opération correspondant au prédicat de transition (état source, étiquette, état arrivée). C'est donc un prédicat qui prend en argument deux termes du type du composant (les états) et un terme correspondant à l'étiquette (communication).

L'interface des composants est donnée par l'ensemble des constructeurs et opérations d'une partie interaction associée au composant. Ceci étant fait algébriquement, le typage est fort. Ces interfaces correspondent aussi aux étiquettes des transitions.

La partie dynamique d'un composant est tout simplement l'assemblage de son interface, de sa partie déclaration des états, ainsi que les axiomes correspondant au prédicat de transition.

La structuration du système se fait par :
- inclusion des spécifications des composants
- définition d'un constructeur pour les états, avec un argument par composant du composé (donc un terme du type du composant)
- définition de l'interface du composant (correspondant à l'union des interfaces des composants)
- définition de l'opérateur correspondant au prédicat de transition (comme pour les composants individuels) :
  - donnée des états sources (états structurés)
  - garde éventuelle concernant les états sources
  - garde éventuelle concernant les transitions faites par les composants et leurs valeurs
  - donnée d'une transition effectuée par le composite dans ce cas
  - donnée des états arrivée (états structurés) en fonction des états source ainsi que des termes correspondant aux différentes étiquettes

  C'est un mode de synchronisation très expressif permettant par exemple la synchronisation à n. Il n'y a pas directement de prise en compte de la communication synchrone. Cela passe par l'utilisation de composants supplémentaires (buffers).
- il est possible de définir les opérations (purement passives sur des types de bases) auxiliaires nécessaire à la composition

Il est possible de spécifier des systèmes ayant un nombre non fixe de composants (ce qui présente un réel intérêt dans un cadre objet) et communiquant de différentes façons. La sémantique globale et les preuves sont définies en utilisant les concepts algébriques puisque LTL est définie de façon homogène dans ce cadre. De même, il est possible de traiter (dans un cadre d'ordre supérieur) les processus comme des données puisqu'ils sont en fait définis comme un ensemble de fonctions.

Pour répondre aux problèmes liés au manque d'expressivité des modèles basés sur un ensemble de traces par rapport aux modèles basés sur des arbres (choix à un moment donné du temps), une version arborescente de LTL existe [83] (mais avec des problèmes d'incomplétude liés à l'expressivité du formalisme obtenu). LTL dispose d'une notation graphique pour la structuration du système en composants

[10] À ne pas confondre avec la logique temporelle linéaire.

[233]. LTL permet la séparation des aspects.

Rewriting Logic. La rewriting logic [200, 201] est un formalisme permettant la spécification de systèmes concurrents basé sur la réécriture équationnelle. Les mécanismes sous-jacents sont basés sur quatre règles simples de déduction : réflexivité, congruence, transitivité et remplacement. Les comportements dynamiques sont construits à partir de systèmes de transitions construits dans un cadre algébrique (avec sémantique initiale).

Les transitions sont données sous une forme basée sur les règles de réécriture conditionnelles (une transition correspond à un pas de réécriture). Les états sont les supports de l'algèbre correspondant à la spécification. Il est possible de composer des composants en utilisant des multi-ensembles d'états et de transitions. Cette composition peut être synchrone ou asynchrone.

La rewriting logic peut être vue sous deux aspects : changements dans un système ou déductions dans un système logique. Pour ce qui nous intéresse (changements) l'analogie est :

état	$\leftrightarrow$	terme
transition	$\leftrightarrow$	réécriture
composition de composants	$\leftrightarrow$	structure algébrique (tuple, ...)

La rewriting logic ne permet pas, contrairement à LTL, les transitions étiquetées (donc pas d'interaction avec l'environnement ce qui est pénalisant dans le cadre de systèmes ouvert ou dans celui de la structuration du système en composants communicants). Ceci est dû au fait que les règles expriment en rewriting logic des transitions effectuées et non pas possibles (la rewriting logic est une "logique du changement" [201]). Cependant, la rewriting logic permet, contrairement à LTL l'expression de transitions non atomiques.

Ce formalisme a été employé dans un cadre objet [199]. La rewriting logic est plus un environnement sémantique pour la concurrence (très peu lisible mais il est possible d'exprimer les réseaux de Petri, LOTOS, les algèbres de processus, Unity, etc., dans la rewriting logic [201]) qu'un formalisme de spécification. Suivant cette approche, le langage Maude [198, 197] est basé sur la rewriting logic pour la partie concurrente (un module de Maude est une théorie en rewriting logic) et permet la spécification de systèmes orientés objet concurrents mais peut aussi servir (par les capacités de réflexivité qu'il hérite de la rewriting logic) de méta-langage. Maude est plus structuré que la rewriting logic : modules (basés sur OBJ3 [129] pour les modules fonctionnels), héritage, etc. Des possibilités de prise en compte du temps réel et de systèmes hybrides existent et ont été répercutées dans Timed-Maude [169]. CafeOBJ [94] est un autre langage basé sur la rewriting logic.

Algebraic State Machines. Les machines à états permettent facilement la description des comportements mais peuvent conduire à une explosion d'états. Pour cela, des formalismes comme les machines à états abstraits (Abstract State Machines anciennement Evolving Algebras [137]) ont été développés. Dans ce modèle, les algèbres représentant les modèles de la partie statique peuvent évoluer. Si c'est un formalisme qui a été utilisé avec succès dans le cadre de systèmes mixtes, son problème principal (mais dans le même temps, cela en fait un formalisme très lisible) reste qu'il est très proche d'un langage de programmation (il est basé sur des affectations). Il existe toutefois des outils de vérification (basés sur son exécutabilité comme des outils d'animation) associés à ces machines. Enfin, il s'agit d'un formalisme bien adapté à la définition de sémantiques opérationnelles pour les langages de programmation (cela a été fait pour Prolog [50] et plus récemment Java [51]).

Une première proposition permettant de spécifier les transitions entre états a été faite dans un cadre algébrique [84] et a été reprise dans un cadre objet [164]. Cette idée de l'application de telles machines

dans un cadre objet a aussi donné lieu à la définition de nouvelles structures mathématiques très générales (paramétrées par le support à la spécification des états-algèbres) pour les machines [25] ou l'extension au cadre objet des méthodes existantes [272]. Ces formalismes sont en effet bien appropriés à un cadre objet puisqu'il est possible d'associer non seulement des valeurs (variables d'instance) aux objets mais aussi des fonctions (méthodes) puisque les objets sont des algèbres. [100] propose, mais informellement, un cadre relativement général pour ces approches. La preuve sur ces approches est beaucoup plus complexe que sur des formalismes moins expressifs (algébrique simple, processus, etc).

Cette famille d'approches a conduit très récemment à la notion de machines à états algébriques [56]. Cette approche n'est pas citée dans [83] mais nous pensons qu'elle est intéressante car elle présente un bon compromis entre expressivité, abstraction, modèle de communication, niveau de formalisation et sa possibilité d'intégrer des concepts objets.

Une spécification de ce type est formée de quatre niveaux :

1. un niveau de base (spécification algébrique) décrivant les types de base utilisées par les composants,
2. un niveau pour l'interface des composants (signatures) vue comme un ensemble de canaux d'entrée et de sortie,
3. un niveau pour l'espace des données (spécification algébrique) des composants,
4. un niveau pour le comportement des composants construit avec un ensemble d'axiomes décrivant l'état initial (restriction de l'ensemble des algèbres vérifiant le niveau précédent), et un ensemble d'axiomes représentant les transitions (basés sur un schéma précondition-entrées-sorties-postconditions). Ces transitions permettent de relier les entrées aux sorties, ainsi qu'un état de départ (une algèbre) à un état d'arrivée (une algèbre).

Le langage utilisé pour les spécifications est SPECTRUM [55] plutôt que CASL car il offre des sortes polymorphiques et des fonctions.

Il est possible de composer les spécifications. Les composants sont vus comme des boîtes noires (entrées et sorties) communiquant de façon asynchrone (composition parallèle avec feedback) dans un temps discret et une concurrence par entrelacement. Les structures d'interprétation sont des ensembles de traces infinies (flots "temporisés", i.e. application des naturels vers les messages). La sémantique de composition est donnée par :

– union des espaces d'état (signatures et axiomes),
– restriction des flots admis par le composé en utilisant une fonction de traitement de flot.

L'inconvénient principal de cette approche reste qu'il ne sépare pas vraiment comportement et données puisque les changements induits sur les données par le comportement se trouvent dans les postconditions. Toutefois, ces fonctions sont définies dans un autre niveau et il est donc possible d'avoir une forme réduite de séparation d'aspects. Son avantage est son expressivité (sans pour cela réduire vraiment sa lisibilité), il est ainsi possible par exemple de déclarer une fonction comme attribut d'un composant. Un modèle objet a été défini à partir de ces spécifications, il est basique (d'un point de vue objet) mais peut être étendu et utilisé pour donner une sémantique à des notations comme UML par exemple.

1.4.2.2 Formalismes logiques

Unity. Unity [67] est un framework logique (un formalisme pour les programmes, une logique pour les propriétés et des règles de déduction et des axiomes pour cette logique) très populaire permettant de raisonner sur la correction des programmes, dont les programmes parallèles. Il s'agit de l'un des premiers

systèmes formels permettant de raisonner sur des programmes et spécifications de systèmes concurrents. Il est possible de spécifier ces programmes tant à un niveau opérationnel (instructions) qu'à un niveau déclaratif (propriétés).

Un programme Unity est composé de quatre parties : la déclaration de variables d'état (typées), la définition d'un prédicat exprimant l'ensemble des états initiaux, une partie définissant la valeur de certaines variables en fonction d'autres (partie quantifiée universellement sur le temps) et un ensemble d'instructions d'affectation de valeurs aux variables d'état. Ces instruction peuvent avoir plusieurs formes. La forme conditionnelle est généralisée sur des tuples de valeurs à assigner :

$$\begin{array}{rcl} v_1, ..., v_n & := & e_1^1, ..., e_n^1 \ \ if \ \ g_1 \\ & & e_1^2, ..., e_n^2 \ \ if \ \ g_2 \\ & & ... \\ & & e_1^m, ..., e_n^m \ \ if \ \ g_m \end{array}$$

La forme énumérée (sucre syntaxique) permet de couper de longues assignations : $x := a \ || \ y, z := b, c$ est équivalent à $x, y, z := a, b, c$. La forme quantifiée permet de généraliser par exemple les assignations (parallèles ou exclusives) : $<|| \ i : 1 \leq i \leq N :: A[i] := B[i] >$ permet d'effectuer l'assignation parallèle des valeurs d'un tableau dans un autre.

Le modèle d'exécution est le suivant : partant d'un état qui satisfait le prédicat des états initiaux, une instruction d'affectation est sélectionnée et exécutée. Quand plusieurs conditions d'instructions sont vraies, les assignations de valeurs doivent correspondre (ce qui est différent des commandes gardées de Dijkstra mais se retrouve dans les ASM [137] par exemple). Cette phase est donc déterministe mais cela nécessite des obligations de preuve de cohérence des affectations. Cependant, en raison de sa sélection arbitraire des instructions à exécuter, les programmes Unity sont (en général) indéterministes. Chaque instruction peut être exécutée infiniment souvent, cela modélise une condition d'équité (faible). Un programme termine s'il atteint un point fixe (état stable par rapport à l'ensemble des variables d'état). À ce modèle correspond un point de vue opérationnel [264] dans lequel la sémantique d'un programme est l'ensemble de toutes les exécutions possibles, une exécution étant dans ce cas une séquence d'états (ce qui revient à considérer un modèle pour interpréter une forme de logique temporelle linéaire).

Les propriétés (ou les programmes abstraits) peuvent être exprimés grâce à la logique d'Unity (langage de spécification associé au langage plus opérationnel pour les programmes). Cette logique est une logique temporelle linéaire permettant d'exprimer des propriétés de sûreté (opérateurs *unless*, *stable* et *invariant*) et de vivacité (opérateurs *leads-to* et *until*) sur un ensemble de modèles d'exécution. L'équité (faible) est présupposée.

La représentation de la communication se fait par exemple en affectant en parallèle (dans le cas de communication synchrone) des valeurs à des variables représentant les composants. C'est une approche relativement peu naturelle mais l'objectif est de rester le plus possible dans un contexte logique de base. Il existe aussi une forme syntaxique dans laquelle plusieurs programmes peuvent être "composés" en parallèle mais le résultat est équivalent : l'union des programmes. Pour un système composé d'un certain nombre d'utilisateurs et d'un contrôle, on écrirait par exemple [225] :

$Systeme =< []u :: Usager(u)[]Controle >$

La partie statique est basée sur des types de base (naturels, booléens) et des types de haut niveau (énumérations, enregistrements, ...). Les propriétés de ces types peuvent en partie être exprimées dans

les programmes mais c'est une approche, qui comme les approches basées modèles reste très peu abstraite (très implémentatoire).

Unity ne dispose pas encore d'outils de preuve dédiés et automatiques, mais il est possible d'utiliser des assistants de preuves (Dada [225], LP [68]). Il existe une théorie d'Unity en HOL.

TLA. TLA (Temporal Logic of Actions) [177, 176] est une logique prenant en compte dans un cadre homogène les aspects statiques et dynamique, permettant de spécifier et de raisonner sur les systèmes concurrents.

De fait, TLA combine deux logiques : une logique d'actions pour manipuler les données et une logique temporelle linéaire pour les comportements et la concurrence.

La partie statique est spécifiée à l'aide de la logique des actions. Les données (partie statique) en TLA sont très limitées (variables, "nombres", chaînes de caractères et booléens). La sémantique des formules est définie en termes d'états, dans le cas présent d'affectations de valeurs à des variables.

Une fonction d'état est une expression (non booléenne) construite à l'aide de variables et de symboles de constantes. La sémantique de ces fonctions est une application des états vers l'ensemble des valeurs. Cette sémantique se base sur le fait que les parties droites des affectations aux variables soient calculables et rien de plus n'est dit. TLA suppose un formalisme logique sous-jacent et dans TLA même il n'est pas possible de spécifier les propriétés des opérations (et plus largement des types de données). TLA+ [179] est un langage de spécification complet (mais non typé) basé sur TLA, la logique du premier ordre et la théorie des ensembles (ZF modifiée pour ne pas prendre certains ensembles en compte [179]). Il inclut une syntaxe concrète pour les définitions, une gestion de la statique améliorée (extension utilisant la théorie des ensembles), et une notion de module. Un prédicat d'état est une expression booléenne construite comme une fonction d'état et sa sémantique est une application des états vers les booléens.

Les actions sont des expressions évaluées sur les booléens et qui expriment une relation entre deux états (un "ancien" et le "nouveau"). Ces actions sont formées de constantes et de variables (des primes dénotant les "nouvelles" valeurs de ces variables). Les opérations (atomiques) des systèmes concurrents sont spécifiées en utilisant des actions. La sémantique d'une action est une relation entre états permettant de définir des pas (transitions) entre deux états. Les prédicats sont une forme particulière d'actions (sans variables primées).

La logique des actions différencie les variables utilisées pour modéliser les états (variables flexibles) de celle, quantifiées existentiellement représentant des paramètres de la spécification (variables rigides). Concrètement, la sémantique des fonctions d'état et des actions n'est donc pas uniquement donnée en termes de variables mais de formules logiques du premier ordre utilisant les variables flexibles (et les constantes). La quantification des variables flexibles est interdite dans les fonctions et les actions.

La partie dynamique est spécifiée à l'aide d'une logique temporelle linéaire. Cette logique exprime des propriétés concernant des séquences infinies (comportements, traces) d'états successifs. C'est une logique bien adaptée aux aspects dynamiques, tout en étant moins expressive que les logiques arborescentes (CTL). La relation entre deux états est donnée par les formules d'actions. Il existe en TLA deux formes de quantification des formules. Celle habituelle sur les variables libres (ici les variables rigides), et celle sur les variables représentant l'état interne (variables flexibles). TLA permet d'exprimer à la fois des propriétés de sûreté et de vivacité. De plus, ce qui fait son originalité, TLA permet, toujours dans le même formalisme, d'exprimer des propriétés d'équité (forte et faible).

La forme générale d'une spécification TLA est :
$Init \wedge \Box[N]_x \wedge L$
où $Init$ est un prédicat d'état décrivant les états initiaux, N est un prédicat décrivant les transitions des composants (et donc quantifié universellement sur le temps en utilisant $\Box$ et L est une formule de vivacité exprimée à l'aide de formules d'équité). $[N]_x$ signifie que N est "exécutée" ou que l'ensemble x des variables flexibles de la spécification est inchangé.

L'un des intérêts de l'approche TLA est que l'expression des propriétés se fait dans le même formalisme (formules de logique implicitement quantifiées universellement sur les variables et le temps) que les spécifications ce qui permet de les vérifier par implication logique. De même, un raffinement peut être prouvé correct en vérifiant qu'il implique la spécification de départ. Une propriété des spécifications TLA est qu'elles sont invariantes par rapport aux *stuttering steps* (qui laissent les variables flexibles inchangées). Cette propriété est importante au niveau des preuves de raffinement (elle permet la modification de l'atomicité). C'est elle aussi qui rend possible la prise en considération de comportements infinis [176].

Un autre avantage de TLA est qu'il est possible de composer des systèmes par simple utilisation de l'opérateur de conjonction logique (mais il faut effectuer un renommage des variables pour éviter les conflits de nom). TLA se base sur un ensemble infini de variables et pas sur des ensembles distincts de variables correspondant aux composants. Cette forme de composition permet une méthodologie de preuve compositionnelle [3]. Le type de communication n'est pas fixé à priori et il est possible d'exprimer à la fois de la synchronisation et des comportements asynchrones. Les mécanismes de synchronisation nécessitent cependant un encodage "précautionneux". TLA a une sémantique de la concurrence par entrelacement mais il est possible d'exprimer une forme de vraie concurrence [196, 3]. TLA peut être utilisé pour les systèmes ouverts. Bref, TLA est une logique très expressive tout en se basant sur un petit nombre de concepts et bien connus des spécialistes. TLA peut être utilisée pour raisonner sur du temps physique (réel), et ce même si cette logique se base sur des événements discrets. TLA+ a été utilisé pour la spécification de systèmes hybrides [175].

L'utilisation des variables (flexibles) pour spécifier les états conduit par contre à être trop proche d'une implémentation et insuffisamment abstrait. Ce sont ces variables (libres) qui décrivent l'interface d'un composant. Le masquage est possible par une forme de quantification (qui suppose non pas l'existence d'une mais d'une séquence infinie de valeurs) des variables flexibles. Les variables en TLA ne sont pas typées ce qui présente des inconvénients en termes d'interface et donc de modularité.

La vérification théoriquement simple en TLA [177] (les règles et axiomes tiennent sur une page) se révèle en pratique relativement complexe en présence de données (et encore plus avec TLA+). Cette complexité peut être en partie levée par l'emploi des "predicate-action diagrams" [178], un formalisme graphique (formellement défini en termes de formules TLA) pouvant être utilisé soit pour représenter un aspect donné (niveau d'abstraction) de la spécification, soit pour illustrer une preuve.

Une première approche consistant à générer un script de preuve en LP existe [106]. Actuellement, la preuve se fait plutôt par utilisation de logiques d'ordre supérieur, et une théorie de TLA existe en Isabelle [162]. Il n'existe pas encore de procédure de vérification de modèle pour TLA (ou TLA+) complet. TLC [271] est une première proposition (implémenté en Java) pour vérifier les modèles d'un sous-ensemble très réduit de TLA+.

Notons que si TLA fournit un cadre formellement bien défini, son principal inconvénient reste avant tout sa faible lisibilité. D'autre part, les spécifications TLA ne sont pas exécutables. TLA est donc plutôt un formalisme cible pour des preuves associées à un langage plus lisible ou un formalisme sous-jacent à ce langage. C'est le cas d'OO-LTL [64], qui est une approche logique à la formalisation de systèmes

orientés objet, ou encore de Disco [160], une méthode de spécification de systèmes réactifs disposant d'un outillage important [2] (compilateur, animateur, édition et vérification de scénarios, preuve avec PVS, Kronos et Upaal [181, 180]).

Par bien des aspects, TLA ressemble donc à Unity (mais aussi à TRIO qui prend en compte des aspects temps réel par explicitement du temps dans ses formules) : c'est un framework à base d'une logique expressive mais peu lisible et donnant lieu à des preuves complexes. Unity n'a pas par contre la possibilité de TLA pour prouver qu'une spécification (dans ce cas, un programme) en implémente une autre (ceci est lié à l'absence de la propriété d'invariance sous stuttering). TLA est plus unifié (spécifications - programmes et propriétés utilisent le même formalisme). Unity ne permet pas non plus d'exprimer explicitement les conditions d'équité. TLA cherche enfin à être plus lisible [177] (sic) en recourant moins que Unity aux opérateurs temporels (et utilise plutôt sa notation primée). Ceci nécessite toutefois le recours à différents types de variables et un ensemble d'opérateurs y référant (liés au stuttering ou au masquage - ce dont Unity ne dispose pas par exemple et qui peut être pénalisant). Certains de ces opérateurs rendent TLA plus expressive (mais donc aussi TLA plus complexe et moins lisible). Enfin, Unity ne permet pas la spécification de systèmes ouverts.

OSL. Cette logique [242] destinée à la spécification de communautés d'objets [98] est issue de travaux autour du formalisme de spécification objet OBLOG [241]. Elle est aussi à la base des langages qui en ont été dérivés : TROLL [88] (sémantique réelle de la concurrence, communication synchrone ou - théoriquement uniquement - asynchrone [89], définition de types en utilisant des constructeurs : ensembles, listes, ...) et GNOME [227] (interfaces bien définies, entrelacement, types de base mais impossible d'en définir de nouveaux, peu lisible). Ces langages sont particulièrement bien adaptés au contexte objet. Les spécifications décrivent des classes d'objets et des instances de ces classes sont utilisées dans les compositions. Il est possible de créer et détruire des instances. L'héritage est défini par utilisation de morphismes entre structures d'interprétation. OMTroll définit un cadre méthodologique [266] qui rend TROLL utilisable dans un cadre analyse et conception objet. L'expression des synchronisations se fait par l'emploi d'axiomes dans la logique. Ces axiomes peuvent utiliser des opérateurs temporels.

La structure d'interprétation est celle des traces d'événements (basées sur les structures d'événements et permettant donc un vrai parallélisme [239]). Différentes autres versions de cette logique (DTL [87], TOSL [90]) ont servi de base d'étude à la distribution ou aux transactions (ces transactions pouvant résulter d'un changement d'atomicité lié à un raffinement posent des problèmes lors du raffinement de propriétés prenant en compte le temps [111]), ainsi qu'à la résolution du manque de compositionalité d'OSL [81]. Toutes ces logiques sont linéaires. Une discussion générale sur les logiques pour les systèmes concurrents (avec facilités orientées objet) sort de notre contexte et peut être trouvée dans [97].

Ces approches présentent un intérêt en tant que fondement théorique permettant l'expression de la vraie concurrence et différents modes de communication dans un cadre objet. Cependant, leur définition dans le cadre de la théorie des catégories les rend difficiles à utiliser. La multitude de concepts pris en compte rend aussi difficile les preuves de propriétés. Ainsi, la différence entre attributs et opérations ou la séparation entre types de données et objets contribue à rendre complexe le modèle. Cependant, des outils sont en développement. Un autre inconvénient est que les langages (GNOME, TROLL) sont relativement proches de langages de programmation : données fixes ou générateurs ensemblistes basiques (pas de spécification par propriété des aspects statiques), non séparation de la dynamique et de la statique.

1.4.2.3 Calculs et Algèbres de processus étendus

Les algèbres de processus sont des formalismes conçus pour l'expression des aspects dynamiques et de la composition parallèle des composants dynamiques. Dans certains cas, des tentatives de prise en compte d'aspects statiques dans ce cadre ont été faites.

Extensions de CCS et CSP. CCS [204, 205] est un formalisme pour la spécification de la dynamique. Il existe toutefois quelques extensions pour prendre en compte des aspects statiques : CCS à passage de valeurs [205] et la proposition de Bruns [57]. Les approches combinant CCS et la spécification de données orientée modèle sont présentées plus haut.

La première proposition est basée sur la traduction du CCS à passage de valeurs dans le CCS de base. L'idée générale de la traduction est la suivante. Soit une cellule :

C $\stackrel{def}{=}$ **in**(x).C'(x)

C'(x) $\stackrel{def}{=}$ $\overline{out}$(x).C

La traduction consiste à transformer l'indéterminisme implicite des réceptions de valeurs en indéterminisme explicite par l'indexation des agents et des actions ainsi que la sommation de l'ensemble sur les éléments du type (types non formellement définis) des valeurs reçues.

$C_v \stackrel{def}{=} \sum_{v \in V}$ **in**$_v$.C'$_v$

C'$_v \stackrel{def}{=} \overline{out_v}$.C

Les inconvénients majeurs de cette proposition sont qu'elle assume un ensemble de valeurs mais sans pouvoir le spécifier (syntaxiquement et sémantiquement). Dans le cas où cet ensemble est infini, la somme aussi est infinie ce qui peut poser des problèmes lors de la vérification. D'autre part, les agents avec passage de valeur sont faiblement typés, ce qui présente un inconvénient dans un contexte où ce typage peut justement servir de signature au composant. Enfin, toutes les actions doivent être paramétrées.

La seconde proposition étend celle de Milner [205] pour prendre en compte les actions simples, paramétrées ou indexées, ainsi que l'ensemble des combinaisons possibles. D'autre part, des types de plus haut niveau (listes, tuples et ensembles) sont définis (et leurs opérations aussi), mais il reste impossible de définir des types avancés (syntaxiquement et sémantiquement) sans viser une implémentation donnée en termes des types de base disponibles. La spécification des données se fait dans une approche orientée modèle. Les données sont fortement typées. La sémantique est donnée par traduction dans CCS de base [204] et gestion d'un environnement pour les données. Le problème de la somme infinie reste valable.

CSP [151, 53] est un formalisme basé sur une sémantique échecs/divergences sur traces (finies [53] ou infinies [236]) contrairement à la bissimulation de traces infinies pour CCS. Leur optique et leurs moyens de structuration restent cependant très proches. En termes d'expressivité au niveau des données, CSP est comparable à CCS à passage de valeurs.

π-calcul. Le π-calcul [203] est une évolution de CCS avec valeurs dans laquelle non seulement les agents peuvent être liés et communiquer, mais où une communication entre agents peut conduire à la modification de ces liaisons. Ceci passe par la communication de valeurs correspondant aux liens. Le π-calcul est constitué comme le λ-calcul et CCS d'un très petit nombre de construction et est très expressif. Toutefois, son illisibilité le réduit au point de vue pratique au rôle de modèle pour langage basé sur des aspects functionnels (λ-calcul) pour les données et concurrents (CCS). Il presente aussi un intéret dans

le cadre de la programmation distribuée. Le π-calcul a ainsi par exemple servi de base (pour la partie sémantique opérationnelle) au langage Pict [218] qui sert de banc d'essai pour langages concurrents typés statiquement utilisant des objets ainsi qu'au langage Oz [248]. Divers calculs étendent le π-calcul comme le join-calcul [117] (variante asynchrone) et sa version permettant la représentation d'agents mobiles [118].

1.4.2.4 Conclusion

L'approche homogène se base sur le fait de pouvoir exprimer les différents aspects à l'aide d'un unique formalisme. Ceci se fait aux dépens de l'expressivité, un unique formalisme étant en général adapté à un aspect particulier (ou plusieurs, mais pas à tous). Il s'agit souvent de l'extension syntaxique ou méthodologique d'un formalisme adapté à l'un des aspects pour couvrir les autres aspects. Certaines approches homogènes sont très expressives, cela se fait alors aux dépens de la lisibilité des spécifications en ayant recours à des formalismes de très haut niveau (logiques temporelles ou d'ordre supérieur). L'avantage majeur de cette approche reste cependant que les problèmes de cohérence entre aspects des approches hétérogènes ne se posent pas. En effet, on reste dans le contexte du formalisme de base où la définition d'une sémantique globale et la vérification se font plus aisément. Ces différents points en font un bon formalisme de vérification (par traduction d'un formalisme hétérogène) plutôt que de spécification. C'est une approche qui reste d'un usage académique.

1.5 Conclusions

Dans ce chapitre nous avons présenté les principaux aspects à retenir concernant les formalismes de spécification mixte des systèmes, ainsi que les besoins associés à ces systèmes. Nous avons ensuite étudié les différentes approches existantes pour répondre à cette problématique, et ce en considérant deux approches principales : hétérogènes et homogènes. Cette étude nous a permis de tirer plusieurs conclusions.

Dans ce contexte des spécifications mixtes, nous prenons le parti de tenter de combiner un certain nombre de besoins concernant les formalismes de spécification. Nous pensons que l'approche hétérogène est plus adaptée (expressivité et lisibilité) à la spécification mixte. Nous pensons cependant aussi que définir un formalisme permettant de spécifier de façon unifiée les différents aspects facilitera leur intégration et la définition d'une sémantique globale. De ce point de vue, nous visons aussi à bénéficier de certains des avantages de l'approche homogène.

Il est important d'établir un équilibre entre l'expressivité du formalisme (tout en gardant à l'esprit que nous le destinons principalement aux systèmes répartis tels que les protocoles et services de communication, et en restreignons donc volontairement ainsi son champ d'application) et sa lisibilité en vue de son utilisation hors du contexte universitaire. Pour cela, il est nécessaire de recourir à des concepts connus (composants, objets identifiés, encapsulation) ou encore à des formalismes graphiques tels que les systèmes à base d'états et de transitions. Cependant, en vue d'éviter l'explosion combinatoire résultant généralement du manque de possibilité d'abstraction proposée par de tels formalismes (les réseaux de Petri mis à part) il est nécessaire d'avoir recours à une forme symbolique de systèmes de transitions.

Le haut niveau de lisibilité désiré passe aussi par la définition de moyens de structuration des spécifications, tels que la définition séparée des aspects (très utile dans un contexte de réutilisation), leur intégration, l'héritage et la composition parallèle de composants.

CHAPITRE 2

KORRIGAN et le modèle des vues

KORRIGAN [korigãn] *n. Myth. ; du breton* korrig *(lutin), et* gan *(chanter).* ♦ *Lutin chantant des légendes bretonnes, tantôt bienveillant, tantôt malveillant.* ⇒ ***lutin, nain***.

Dans ce chapitre, nous présentons un modèle basé sur une hiérarchie de vues pour spécifier les systèmes mixtes de façon unifiée, intégrée et structurée. Nous donnons les règles permettant de définir une sémantique opérationnelle pour ces spécifications.

2.1 Le modèle des vues

Nous nous intéressons à la spécification de systèmes mixtes qui comportent plusieurs aspects (statique, dynamique,...) et présentent un niveau de complexité qui rend nécessaire la définition de mécanismes de structuration des spécifications. Nous définissons trois mécanismes de structuration : interne (figure 2.9), externe (figure 2.13) et héritage. Ces mécanismes de structuration s'articulent autour de notre modèle des vues (figure 2.1). Nous définissons aussi un langage de spécification formelle associé à ce modèle : KORRIGAN.

Nous présentons notre modèle dans ce chapitre en s'appuyant sur des études de cas dont les besoins et les spécifications complètes pourront être trouvées en annexe.

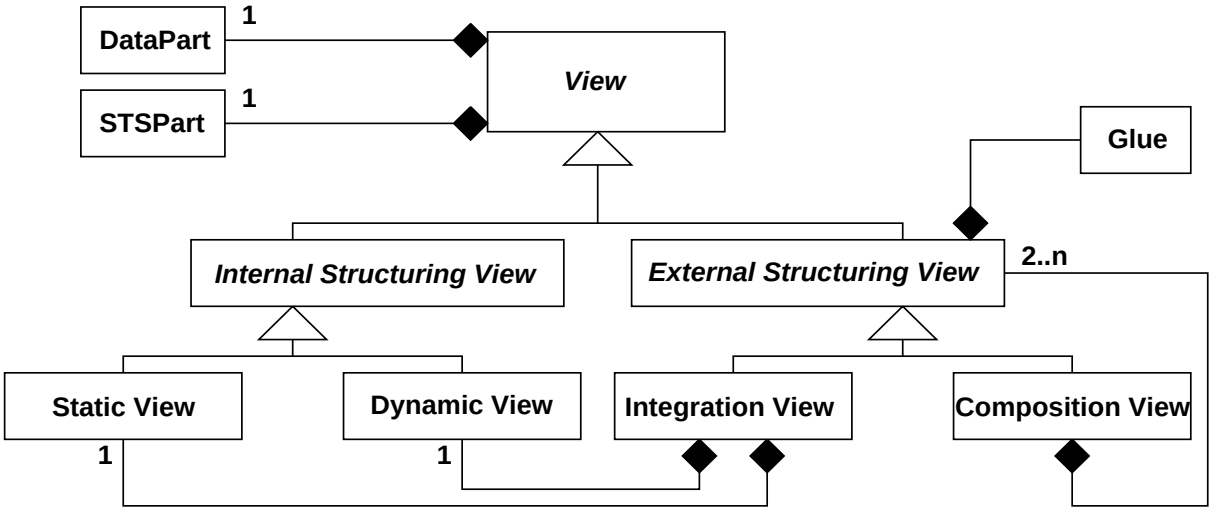


Figure 2.1 : Diagramme de classes – Modèle des vues

Le concept central de notre modèle est le concept abstrait de *vue*. Une vue est utilisée, dans une approche unificatrice, pour spécifier les différents aspects des composants, tant la structuration interne, qu'externe, ou que l'héritage. De façon plus générale, ce concept fournit un principe permettant de spécifier et d'intégrer les différents aspects des composants.

De façon à obtenir un bon niveau d'abstraction et de lisibilité, nous utilisons des *systèmes de transition symboliques* (STS) pour représenter les aspects dynamiques des vues. Les STS sont basés sur les graphes de transition symboliques (Symbolic Transition Graph, STG) [148, 228], qui sont utilisés comme un moyen fini de représentation de systèmes potentiellement infinis. Ces systèmes ont récemment été utilisés pour tenter de donner une sémantique globale à full-LOTOS [243, 166]. D'autres approches nous ont aussi inspiré, telles que celle des Types Abstraits Graphiques [11] (TAG) qui utilise les STS dans le but de donner une interprétation en termes de système fini état-transition aux spécifications algébriques, en conservant toutefois un aspect purement statique et en ne les structurant pas comme dans notre modèle des vues.

Les STG [148] sont une forme d'automate gardé où des variables libres apparaissent dans les gardes et les transitions. Ils sont au départ utilisés pour représenter de façon finie des processus CSP.

Définition 2.1 (Symbolic Transition Graph (STG)). *Un STG est un graphe dirigé (S, S_0, T), où S est l'ensemble des états (appelés aussi nœuds), S_0 est l'ensemble des états initiaux, T est l'ensemble des transitions (appelées aussi branches), et dans lequel :*

- *chaque nœud n est étiqueté par un ensemble de variables libres $fv(n)$*[1]
- *chaque branche est étiquetée par une action gardée (une garde booléenne b et une action α qui peut lier des variables*[2]*)*
- *pour toute une branche étiquetée par (b, α) qui part d'un nœud m et arrive à un nœud n on a :*
  - $fv(b) \cup fv(\alpha) \subseteq fv(m)$
  - $fv(n) \subseteq fv(m) \cup bv(\alpha)$

Dans cette définition, les ensembles des variables libres et liées des actions sont définis de la façon habituelle : $fv(c!e) = fv(e)$, $bv(c?x) = \{x\}$ et sinon, $fv(\alpha)$ et $bv(\alpha)$ à la fois sont vides.

Nos STS (voir l'annexe A pour la grammaire exacte) étendent les STG sur plusieurs points.

Un premier point est que les états de nos STS ne sont pas vus en tant qu'ensemble de variables mais comme une classe d'équivalence de termes (algébriques). Il est aussi possible de les voir comme des tuples de conditions construisant le niveau d'abstraction du composant décrit. Les STS permettent de cette façon à la fois de réduire l'explosion d'états en exprimant un bon (haut) niveau d'abstraction et d'être liés à la partie algébrique de nos vues (définition 2.2 plus bas) puisque les axiomes de cette partie peuvent être partiellement dérivés (préconditions et parties gauches) des STS (voir à ce sujet la section 3.3.4 du chapitre 3, qui présente cette dérivation dans le cadre, plus large, d'une méthode de construction de spécifications mixtes).

Un second point est que, dans nos STS, la sémantique des opérations qui étiquettent les transitions est obtenue par correspondance avec les axiomes d'une spécification algébrique (structure de vue, définition 2.2), ou par collage avec une transition d'un autre STS (colle dans les composition de vues, définition 2.11). Elle est donc explicite. Dans les STG, la seule sémantique est celle (implicite) associée au nom des variables : la valeur d'une variable émise au niveau d'une transition correspondant à celle d'une variable

[1] fv pour "free variables".

[2] Ces variables liées sont alors dénotées par bv, pour "bound variables".

de même nom reçue lors d'une transition précédente. Il est aussi important de noter que dans nos STS, pour chaque état, la valeur du type dit d'*intérêt* (c'est-à-dire du type algébrique associé au STS) dans cet état est dénotée par `self`.

Un dernier point est que nous avons deux gardes au niveau des transitions de nos STS. La première est utilisée pour sélectionner un sous-ensemble de la classe d'équivalence correspondant à l'état source. La seconde est de façon identique utilisée pour sélectionner un sous-ensemble de la classe d'équivalence correspondant à l'état cible, et un sous-ensemble des valeurs potentiellement recevables au niveau d'une transition. Ceci est à rapprocher de l'existence des concepts de garde et de précondition dans des algèbres de processus comme LOTOS par exemple. Les STG ne différentient pas ces deux types de gardes.

Prenons un buffer non borné contenant des entiers, et disposant des opérations `put` et `get`, cette dernière opération n'étant possible que quand le buffer n'est pas vide. Avec l'approche des systèmes de transition habituels, on a une explosion du nombre d'états puisque les variables correspondant aux réceptions et aux paramètres des processus doivent être instanciées (figure 2.2) ; cette explosion est à la fois verticale (liée aux réceptions, sachant qu'il existe une infinité d'entiers pouvant être reçus) et horizontale (liée aux paramètres du processus, sachant qu'il existe une infinité d'états possibles pour le contenu du buffer, et ce à la fois en ce qui concerne sa taille et les valeurs contenues).

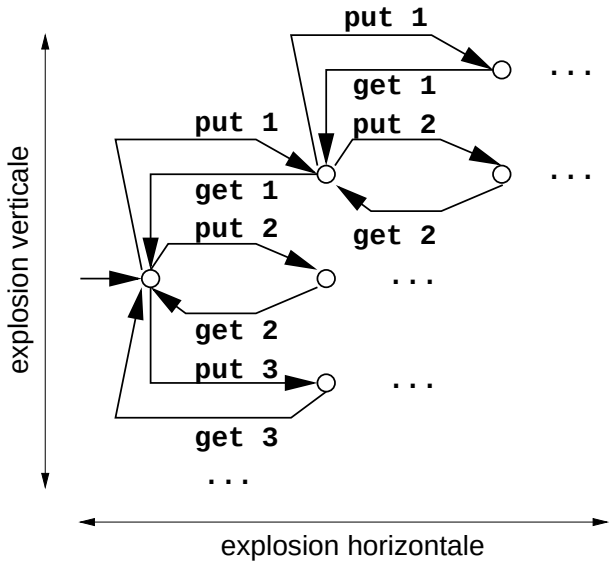


Figure 2.2 : Buffer non borné – Système de transitions habituel

Avec les STS, il est possible de s'abstraire du contenu du buffer et des valeurs reçues, et de s'intéresser uniquement au fait que ce buffer soit vide ou non (figure 2.3).

Pour citer [243] il est possible de dire que les STS sont des systèmes de transition qui séparent les données du comportement dynamique des processus. Pour cette raison, dans la figure 2.3, la sémantique des opérations, ainsi que les relations entre les deux opérations ne sont pas définies. Cela se fera explicitement (voir plus haut la remarque sur la façon implicite dont cela est fait dans les STG) en combinant une spécification algébrique avec le STS, dans le cadre de nos vues.

D'autre part, les STS constituent un formalisme offrant un très bon niveau de lisibilité et permettent de faciliter le travail sur nos vues (en les définissant en utilisant des opérations définies en termes d'états et de transitions).

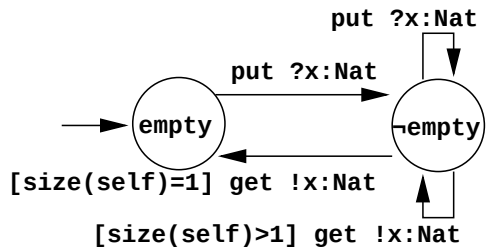


Figure 2.3 : Buffer non borné – Système de transitions symboliques

Cependant, alors que les automates et les systèmes de transition habituels (c'est-à-dire clos) sont une notation communément admise et équipée d'outils de conception ou de vérification - voir [109, 192] par exemple -, les STS ne sont pas encore équipés d'outils (la théorie a été très récemment développée) tels que ceux, par exemple, en rapport avec la vérification de modèle par rapport à une logique temporelle [228, 191, 166] ou avec la bissimulation [148, 243].

Il faut noter qu'il existe d'autres causes pouvant conduire à l'explosion d'états. Celles-ci ne sont pas résolues par l'emploi de systèmes de transition symboliques :

– explosion due à la communication asynchrone (gestion de buffers),
– explosion due à la composition parallèle d'un nombre non borné de processus (paramétrisation),
– explosion due aux contraintes temps réel.

Définition 2.2 (Vue). *Une vue V_T d'un composant T est un tuple (A_T, α, D_T) où A_T est la partie données, α est une abstraction, et D_T la partie comportementale définie sur A_T.*

La partie données concerne la partie fonctionnelle du composant. L'abstraction définit un point de vue (ou une interprétation abstraite) pour le composant. Pour un même aspect d'un composant, il peut exister différents points de vue (plus ou moins abstraits). Ces points de vue sont basés sur des conditions.

Reprenons l'exemple du buffer non borné en le modifiant légèrement pour qu'il le devienne. Le buffer dispose toujours de l'opération `get`, permettant de retirer un élément quand le buffer n'est pas vide, mais maintenant l'opération `put`, permettant d'ajouter un élément, n'est possible que quand le buffer n'est pas plein.

Il existe quatre points de vue sur le buffer : celui construit sans conditions, celui avec une condition correspondant au fait que le buffer soit vide, celui construit avec une condition correspondant au fait que le buffer soit plein, et celui construit avec ces deux conditions.

La partie données et la partie abstraction sont liées par le fait qu'il est possible de dériver les pré-conditions et parties gauches des axiomes de la partie donnée en fonction d'une abstraction donnée (voir le chapitre 3). La partie comportementale s'intéresse aux séquences d'opérations (communications) possibles du composant. Les parties comportementales et abstraction sont aussi très fortement liées. En effet, la sémantique des opérations dans la partie comportementale est donnée en fonction de l'abstraction.

2.1.1 Partie données

La partie données concerne la partie fonctionnelle du composant. Elle définit la sémantique des opérations, dans une approche algébrique. C'est une approche qui présente le mérite d'être à la fois bien connue et pourvue d'outils [269, 99].

Définition 2.3 (Partie données d'une vue). *La partie données A_T (d'une structure (A_T, α, D_T) de vue) est une spécification algébrique comportant les composants $(A', G, V, \Sigma_T, Ax_T, \overline{A})$, où :*

- *A' est un ensemble de spécifications algébriques de base importées (types de données sans vue) ;*
- *G est un ensemble de spécifications algébriques paramètres formels ;*
- *V est un ensemble de variables paramètres formels ;*
- *Σ_T, Ax_T sont les signatures et les axiomes (formules de logique du premier ordre) des opérations de T ;*
- *$\overline{A}$ est l'ensemble des opérations cachées (ou non exportées) de T.*

Il s'agit d'une spécification algébrique classique, voir [269, 99] pour les sémantiques habituellement associées à de tels systèmes. Le minimum requis est de disposer d'une sémantique initiale (réécriture) pour ces spécifications. Dans la suite, nous appellerons *amorphe* un type de donnée "de base", c'est-à-dire qui est décrit directement dans un formalisme de spécification algébrique, sans vue particulière.

2.1.2 Partie comportementale

La partie comportementale s'intéresse aux séquences d'opérations (communications) possibles du composant. Il peut paraître étonnant d'associer une partie comportementale avec communication à tous les aspects des composants. C'est toutefois l'un des aspects qui nous permet d'avoir un formalisme unifié pour intégrer les différents aspects de ces composants. D'autre part, nous verrons plus loin qu'en quelque sorte les aspects statiques des composants "communiquent" avec les aspects dynamiques de ces mêmes composants.

Choix concernant la communication, le temps et la concurrence. La partie comportementale des vues concerne les communications du composant. De ce point de vue, il est important de faire d'abord des choix concernant l'arité des communications et les moyens d'expression des échanges (table 2.1). Ces choix proviennent de l'étude des langages de spécification utilisés en télécommunications [263] (LOTOS [154, 47], SDL [66, 105, 222] et Estelle [153]).

arité	
	1 → 1 (langages orientés objet, SDL) 1 → n (en parallèle, ou par séquence atomique) n ↔ n (synchronisation multipoint de LOTOS)
communication	
implicite	mots-clés particuliers (comme `sender` en SDL) par structuration et masquage d'opérations (SDL, LOTOS)
explicite	mot-clé `from` et identifiants de processus mot-clé `to` et identifiants de processus (SDL)

Table 2.1 : Choix de communication

Une première motivation est le souhait de disposer de spécifications expressives et lisibles. Pour cela, nous autorisons à la fois la communication explicite (à l'aide d'identifiants de composants) et la communication implicite par l'intermédiaire du mot-clé `sender`, qui dénote l'émetteur du dernier message reçu. La sémantique de ce mot-clé est définissable algébriquement de façon unique à partir du moment où il est possible de faire une correspondance entre opérations statiques algébriques (basées sur des signatures algébriques) et opérations dynamiques (opération internes, émissions et réceptions basées sur

ce que nous appelons *signatures dynamiques*, définition 2.4). Nous verrons plus loin qu'il est possible de faire cette correspondance et nous donnerons alors la sémantique de `sender` (figure 2.4).

La communication implicite par structuration est aussi autorisée. En effet, bien qu'elle contribue à diminuer la lisibilité des spécifications, elle offre un plus quant à la réutilisabilité des composants ou des blocs (patterns) de composants.

Une seconde motivation est l'orientation objet que nous désirons donner à nos spécifications. Pour cela, nous faisons le choix d'un mode de communication de type $1 \rightarrow 1$ (mais aussi celle de type $1 \rightarrow n$ atomique que l'on peut retrouver en UML par exemple). Pour des raisons de lisibilité, nous choisissons d'ignorer la communication multipoint (de type communication par "bus") à la LOTOS qui est expressive mais nous apparaît toutefois comme moins facilement compréhensible. De plus, ce type de communication est peu utilisé dans la pratique tant en spécification des télécommunications (elle existe en LOTOS, mais pas directement en Estelle ou en SDL), qu'en analyse et conception à objets. Elle n'existe pas en UML par exemple, même si cela tend à changer, avec une proposition récente d'implémentation de communication multipoint dans le méta modèle d'UML [149].

Il est nécessaire de préciser le modèle de communication (asynchrone ou synchrone) ainsi que les modèles temporels et de concurrence. Pour l'instant nous nous restreignons au modèle de communication synchrone. Ceci peut sembler restreignant si l'on se place par exemple dans un cadre de communication orientée objet où l'envoi de message est intrinsèquement asynchrone. Il est toutefois possible d'utiliser des buffers en mode synchrone pour obtenir une communication asynchrone. En ce qui concerne le choix d'un modèle temporel, nous choisissons le modèle du temps logique (discret) qui permet d'utiliser des procédures de vérification plus simples et plus d'outils existants. En ce qui concerne la sémantique de la concurrence, nous ne faisons pas de choix à priori. Nous verrons lorsque nous nous intéresserons à la sémantique opérationnelle des compositions (section 2.5) que nous autorisons les deux modes, entrelacement et vraie concurrence.

Identifiants de processus. Nos choix concernant les communications nécessitent la définition d'identifiants de processus. Il s'agit simplement de termes de sortes `PId` (comme en SDL). Cependant, et contrairement à SDL, chaque classe de processus à une sorte d'identifiant associée. Ceci permet un contrôle de type supplémentaire au niveau de la spécification des échanges de messages. Une relation de sous-sorte au niveau des identifiants de processus correspond à des classes de composants reliés par héritage (voir la partie concernant la structuration d'héritage).

Il existe deux importantes propriétés concernant les identifiants de processus : l'égalité d'objet et la structure d'objet. L'égalité d'objet exprime le fait que deux objets qui ont le même identifiant dénotent en fait le même objet :

$$\forall o, o' \in Obj \; . \; Id(o) = Id(o') \Leftrightarrow o = o'$$

La structure d'objet est exprimée par le fait que, quand un objet O est composé de plusieurs sous-objets, alors l'identifiant de cet objet est un préfixe pour tous les identifiants des sous-objets :

$$\forall o \in Subobjects(O) \; . \; \exists i : PId \; . \; Id(o) = Id(O).i$$

De cette façon, les identifiants définissent une structure d'arbre indexée par les identifiants par rapport à la composition d'objets. L'indexation par identifiants est très largement utilisée lors de la spécification de composants construits à partir d'autres (sous-)composants. Nos identifiants sont en quelque sorte un

pendant typé et formel aux URL fréquemment utilisées comme identifiants sur les réseaux.

Signatures dynamiques. Ces choix de communication conduisent à la définition de signatures particulières correspondant aux opérations dynamiques (communications) : les signatures dynamiques. Ces signatures s'inspirent des offres en LOTOS [47] et des communications en SDL [105]. Ensemble, les concepts ayant pour origine ces deux langages nous assurent une expressivité plus grande au niveau des communications.

Définition 2.4 (Signatures dynamiques). *Les signatures dynamiques sont données sous une forme où apparaissent les éléments (offres, paramètres) de la communication, enrichie par des arguments concernant le mode de communication explicite.*

La syntaxe exacte de ces signatures pourra être consultée dans l'annexe A, consacrée à nos notations.

Il existe une correspondance simple entre signatures statiques (algébriques, habituelles) et dynamiques :

- `comm?m:Msg from u:PId`
  (réception d'un message `m` de `u` sur le canal de communication `comm`)
  constructeur `comm : T, Msg, PId → T`
- `comm!m:Msg to u:Pid`
  (émission d'un message `m` vers `u` sur le canal de communication `comm`)
  constructeur `comm_c : T → T`
  observateur `comm_o_m : T → Msg`
  observateur `comm_o_id : T → PId`

Cette correspondance faite, il est possible de donner la sémantique algébrique de `sender` (figure 2.4), que nous avions annoncé plus tôt.

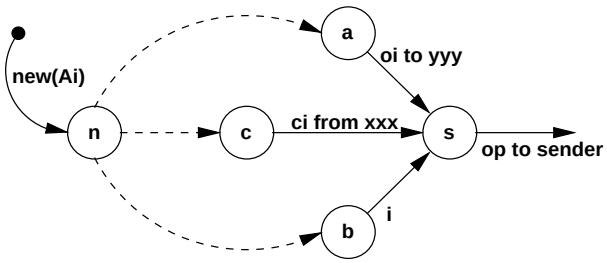


Figure 2.4 : Sémantique de `sender`

Supposons que l'on se trouve dans un état s du STS (figure 2.4), à partir duquel il existe une transitiion dont l'étiquette utilise `sender`. On sait qu'un type de donnée (le type d'intérêt) est associé au STS au sein de la structure de vue (voir la définition 2.2). D'autre part, la valeur courante de cet type d'intérêt par rapport à l'état (courant lui aussi) du STS est dénotée par `self`.

Pour être en s, il existe quatre possibilités :

- on a franchi une transition correspondant à une émission (transition entre a et s), et dans ce cas (il n'y a pas eu réception de message) la valeur de `sender` en s est donc la même qu'en a ;
- on a franchi une transition correspondant à une action interne (transition entre b et s), et dans ce cas (il n'y a pas eu réception de message non plus, une action interne ne faisant pas intervenir

l'environnement du STS) la valeur de `sender` en s est donc la même qu'en b ;
- on a franchi une transition correspondant à une réception (transition entre c et s), et dans ce cas la valeur de `sender` en s est l'un des arguments du constructeur associé à la réception ;
- on est dans l'état initial, et la valeur de `sender` est indéfinie.

Ces possibilités peuvent se résumer, en se basant sur la correspondance entre signatures statiques et dynamiques, à l'aide des quatre axiomes suivants :

```
sender(oi(self,...)) == sender(self);
sender(i(self,...)) == sender(self);
sender(ci(self,...,xxx)) == xxx;
sender(new(Ai)) indefini
```

Définition de la partie comportementale. La partie comportementale décrit les séquences possibles de communications que le composant peut avoir avec son environnment. Il nous paraît important de définir le comportement des composants en fonction d'un certain niveau d'abstraction, pour échapper à l'explosion combinatoire liée à la description de la dynamique en terme de systèmes de transition.

L'idée principale est de définir les opérations en termes de leur effet sur des conditions[3] (C) qui construisent l'abstraction du composant. De cette façon, à chaque opération, il est possible de faire correspondre des préconditions et postconditions définies à partir de ces conditions (voir aussi le chapitre 3, concernant la méthodologie associée à notre formalisme).

Définition 2.5 (Partie comportementale d'une vue). *La partie comportementale D_T (d'une structure (A_T, α, D_T) de vue) est un tuple $(Id, O_T, \overline{O_T})$, où :*
- *Id est un identifiant de sorte PId_T ;*
- *O_T est un ensemble de besoins d'opération, construits à partir des signatures dynamiques $\Sigma_T(A_T)$ (*i.e. *données dans la partie A_T), de préconditions et de postconditions ; c'est-à-dire que $O_T \subseteq Term_{\Sigma_T(A_T)} \times P \times Q$, où P est l'ensemble des préconditions et Q celui des postconditions ;*
- *$\overline{O_T}$, construit à partir de $\overline{A}(A_T)$, est l'ensemble des opérations cachées de O_T.*

Les préconditions sont des formules de logique du premier ordre qui expriment les besoins des opérations en termes des valeurs (P_C) des conditions de C (données dans la partie α) et des arguments des termes $Term_{\Sigma_T(A_T)}$ construits à partir de la signature dynamique (dénotés par P_{io}).

Ainsi par exemple, si l'on considère une opération $put(self, x)$ d'un buffer (défini par rapport à une abstraction plein/non plein et vide/non vide à l'aide des conditions $plein$ et $vide$), qui est uniquement possible si l'argument x est un nombre pair et si le buffer n'est pas plein, alors la précondition de cette opération est $\neg plein \wedge pair(x)$, où $plein$ appartient à C (et donc à P_C) et $pair(x)$ appartient à P_{io}.

Les postconditions sont un ensemble $Q_{C'}$ de formules de logique du premier ordre qui expriment pour chacune des opérations les valeurs C' des conditions de C après que l'opération ait été appliquée, et ce en fonction des valeurs des conditions de C (Q_C), des valeurs de nouvelles[4] conditions Cl (Q_{Cl}), et des arguments des termes construits à partir de la signature dynamique (Q_{io}) après que l'opération ait été appliquée.

Si l'on prend le même exemple que ci-dessus (opération put), on obtient la postcondition $\{Q_{vide'}, Q_{plein'}\}$ dans laquelle $Q_{vide'} = faux$ et $Q_{plein'} = presquePlein$ (c'est-à-dire que le buffer est plein si, avant que l'opération put ait eu lieu, il était presque plein). Il est possible d'observer qu'ici, il a été

[3] Voir le Chapitre 3 pour la définition d'une méthodologie permettant le recensement de ces conditions.

[4] L'expérience montre que ces conditions, appelées conditions limites, sont souvent nécessaires pour exprimer la valeur des conditions de C dans les postconditions.

nécessaire d'introduire une nouvelle condition ($presquePlein$) pour exprimer la postcondition associée à la condition $plein$.

2.1.3 Abstraction

La partie abstraction des vues utilise un ensemble (C) de conditions pour définir un point de vue (basé sur un ensemble de classes d'équivalences) pour un composant donné.

Définition 2.6 (Partie abstraction d'une vue). *La partie abstraction α (d'une structure (A_T, α, D_T) de vue) est un tuple (C, Cl, Φ, Φ_0), où :*

- *$C \subseteq \Sigma_T(A_T)$ est un ensemble fini de prédicats ;*
- *$Cl \subseteq \Sigma_T(A_T)$ est un ensemble fini de conditions limite (prédicats distincts de C) utilisé dans les postconditions des opérations ;*
- *$\Phi \subseteq Ax_T(A_T)$ est un ensemble de formules sur les éléments de C (combinaisons valides) ;*
- *$\Phi_0 \subseteq Ax_T(A_T)$ est une formule sur les éléments de C (valeurs initiales).*

L'ensemble C des conditions étant fini, il existe un nombre fini de classes d'équivalence. Si le cardinal de l'ensemble des conditions est $x =\mid C \mid$, alors il existe au maximum (puisque certaines combinaisons peuvent être rendues invalides par Φ) 2^x classes d'équivalences pour le composant. Voir l'exemple du buffer plus haut.

2.1.4 KORRIGAN

La syntaxe pour les vues en KORRIGAN est donnée dans la figure 2.5. La partie `SPECIFICATION` correspond à A_T, la partie `ABSTRACTION` correspond à α, et la partie `OPERATIONS` correspond à D_T. Il n'est pas nécessaire d'exprimer explicitement $\overline{O_T}$ car cet ensemble peut être obtenu à partir de $\overline{A}$. L'identifiant n'apparaît pas ici car les identifiants sont uniquement utilisés dans le cadre d'une composition (structuration externe).

VIEW T	
SPECIFICATION	**ABSTRACTION**
imports A'	**conditions** C
generic on G	**limit conditions** Cl
variables V	**with** Φ
hides $\overline{A}$	**initially** Φ_0
ops Σ	**OPERATIONS**
axioms Ax	O_i
	pre: P
	post: Q

Figure 2.5 : Syntaxe KORRIGAN– Vues

Un STS (S, S_0, T) peut être obtenu de façon unique à partir des parties `ABSTRACTION` et `OPERA-`

TIONS (voir aussi le chapitre 3).

$$S = \{s \in 2^{|C|} \mid \forall \phi \in \Phi \,.\, s \models \phi\}$$
$$S_0 = \{s_0 \in S \mid s_0 \models \Phi_0\}$$
$$\forall o \in O - \overline{O} \,.\, \forall s \in S \mid s \models o.P_C \,.\, \forall s' \in S \mid s' \models o.Q_C \,.\, t : s \xrightarrow{[o.Q_{Cl}] \;\; \{o\} \;\; [o.P_{io} \wedge o.Q_{io}]} s' \in T$$
$$\forall o \in \overline{O} \,.\, \forall s \in S \mid s \models o.P_C \,.\, \forall s' \in S \mid s' \models o.Q_C \,.\, t : s \xrightarrow{[o.Q_{Cl}] \;\; i_{\{o\}} \;\; [o.P_{io} \wedge o.Q_{io}]} s' \in T$$

Les états sont donnés par les différentes combinaisons de valeur des conditions qui satisfont les relations de Φ. Les états initiaux quant à eux sont les états qui satisfont la formule Φ_0. Enfin, les transitions sont obtenues automatiquement en utilisant les opérations, ainsi que leurs préconditions et postconditions.

De façon inverse, des parties abstraction et opérations triviales peuvent être dérivées à partir d'un STS. L'idée est de donner une condition par état, de considérer ces conditions comme étant exclusives (en utilisant Φ), puis de définir les préconditions et postconditions des opérations en les utilisant.

La syntaxe de nos STS est donnée en annexe A.

Les vues peuvent donc être définies alternativement en utilisant soit un STS, soit en définissant une abstraction et en donnant les préconditions et postconditions des opérations en fonction de cette abstraction. La syntaxe alternative (utilisant les STS) en KORRIGAN est donnée dans la figure 2.6. Il faut noter que les parties STS dans les vues peuvent être données en utilisant soit des termes construits à partir de signatures statiques[5], soit des offres construites à partir de signatures dynamiques (voir la définition 2.4). Ces parties STS sont une représentation textuelle du STS qui ont aussi une forme graphique (*i.e.* plus lisible), comme dans la figure 2.12 par exemple.

VIEW T	
SPECIFICATION	**STS**
imports A'	• → initialState
generic on G	sourceState − transition → targetState
variables V	...
hides $\overline{A}$	
ops Σ	
axioms Ax	

Figure 2.6 : Syntaxe KORRIGAN alternative – Vues

Une propriété intéressante est que, comme dit plus tôt, les vues utilisent une abstraction (α) pour décrire des classes d'équivalence pour les composants. Les STS obtenus à partir des vues ont exactement un état par classe d'équivalence. Le nombre de ces classes étant fini, les STS sont donc aussi finis (pas d'explosion du nombre d'états ou de transitions).

Une fois la structure de vue (V) définie, l'idée est que pour rajouter un concept de structuration $\mathcal{C}$ il suffit de définir (figure 2.7) :

– une structure $\mathcal{S}_\mathcal{C}$ associée à ce concept ;

[5] Le premier argument des termes, `self`, relatif au type d'intérêt est parfois omis. Il est alors implicite.

– une fonction de transformation f de cette structure vers la structure de vue :

$$\mathcal{C} \rightarrow (\mathcal{S}_\mathcal{C}, f : \mathcal{S}_\mathcal{C} \rightarrow V)$$

Il est possible de rajouter des contraintes $\mathcal{K}$ au concept ainsi défini (asymétrique, transitif, contraintes de typage, etc).

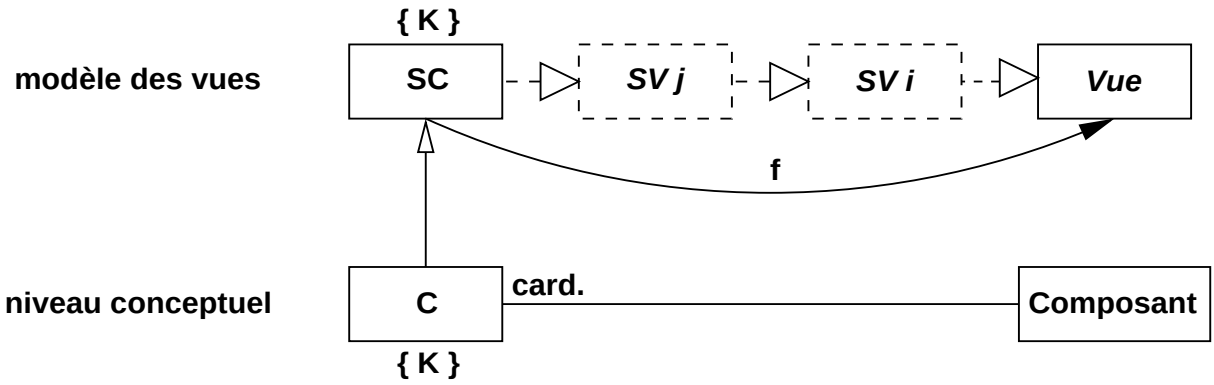


Figure 2.7 : Principe de structuration par vue

Ce principe permet d'étendre notre modèle. Un exemple est décrit dans la figure 2.8. Une *séquence* est un concept qui relie deux composants (*i.e.* deux vues : $Vue1$ et $Vue2$). Une structure correspondante possible est $(Vue1, Vue2)$, et celle-ci peut être convertie dans notre structure de vue en composant les STS des deux vues de façon à ce qu'aux états finaux de $Vue1$ correspondent les états initiaux de $Vue2$.

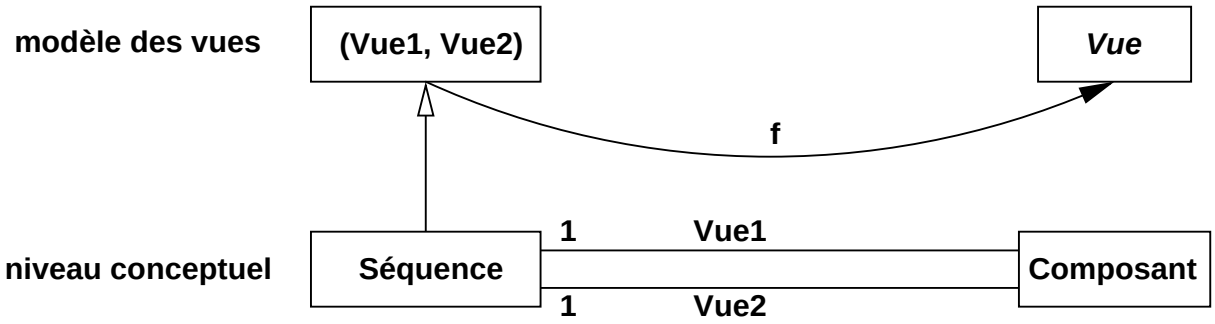


Figure 2.8 : Principe de structuration par vue (Séquence)

2.2 Structuration interne

Nous définissons tout d'abord une structuration interne (*ISV, Internal Structuring View*, figure 2.9). Elle exprime le fait que pour concevoir un objet, il est utile de pouvoir le définir sous chacun de ses aspects (aspects statiques et dynamiques, sans exclure d'autres aspects qui pourraient être identifiés plus tard). Spécifier les composants en tant qu'assemblage de différents aspects est aussi un moyen reconnu permettant de réutiliser ces aspects tout en échappant à l'anomalie d'héritage [193] qui survient lorsque l'expression de la synchronisation est mélangée avec le corps des méthodes (effet sur les données).

Les aspects sont donc spécifiés séparément (ils sont orthogonaux) et une colle fait la liaison.

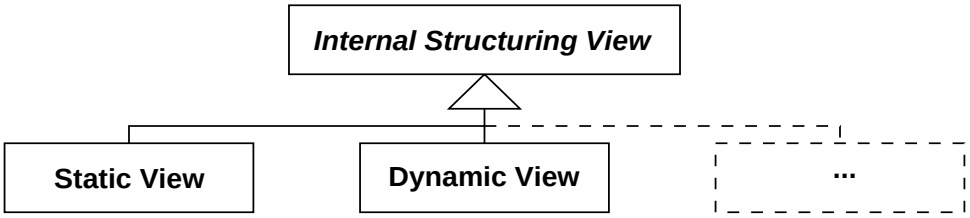


Figure 2.9 : Diagramme de classes – Structuration interne

La description des différents aspects se fait en leur donnant une sémantique basée sur la structure abstraite de vue (et donc en définissant un cas particulier et concret de vue pour chaque aspect).

Dans la suite, nous décrivons les vues statique (*SV, Static View*) et dynamique (*DV, Dynamic View*), *i.e.* des cas particuliers d'ISVs. Pour présenter ces concepts, nous utilisons l'étude de cas de gestion des mots de passe sous Unix (annexe B).

Dans notre approche unificatrice, les vues statique et dynamique sont donc identiques à deux différences près :

- une vue dynamique ne comprend pas de spécification algébrique (partie A_T) complète (elle est purement une abstraction comportementale dans laquelle seules les signatures ont leur importance)
- les mots-clé de communication (`from`, `to`, `sender`), et les sortes PId n'ont pleinement de sens que dans le cadre d'une vue dynamique

Tout ceci est lié au fait que seules les vues dynamiques s'intéressent à l'abstraction (aspect) communication des composants.

S'il est naturel de définir les aspects dynamiques à l'aide d'un système de transitions, il peut par contre paraître étrange de faire de même pour les aspects statiques. L'idée est que nous cherchons à unifier l'intégration des aspects "de base" (statique et dynamique) et la composition parallèle en un seul est unique concept de composition, et ce afin de limiter le nombre de concepts utilisés pour spécifier. Une possibilité pourrait être d'utiliser simplement une vue par composant, avec utilisation de la partie données pour son aspect statique et de la partie STS pour son aspect dynamique. Cette approche présente toutefois deux inconvénients :

- elle ne permet pas la définition de nouveaux aspects "de base" au sein de la structure de vue ;
- elle nuit considérablement au principe de séparation des aspects permettant de réutiliser séparément les deux aspects.

2.2.1 Aspects statiques

Ces aspects recouvrent tout ce qui concerne la gestion des données internes aux composants. Les vues correspondant aux aspects statiques sont les vues statiques.

Définition 2.7 (Vue statique). *Une vue statique SV_T d'un composant T est une vue dans laquelle :*

$$\mathcal{S}_{\mathcal{SV}} = V$$
$$f = Id$$
$$\mathcal{K} = \{\}$$

Si l'on s'intéresse aux aspects statiques du gestionnaire de mots de passe, on utilise une vue statique (figure 2.10) pour modéliser un ensemble de couples $(UserId, Password)$. La représentation graphique

de la partie statique est donnée dans la figure 2.11. Ici l'argument `self` des opérations n'est pas indiqué au niveau des transitions, il est implicite.

STATIC VIEW StaticPasswordManager (SPM)

SPECIFICATION

imports Boolean, UserId, Password

ops

empty : → SPM
add : SPM, UserId, Password → SPM
modify : SPM, UserId, Password → SPM
declared : SPM, UserId → Boolean
correct : SPM, UserId, Password → Boolean

axioms

modify(empty,u,p) == add(empty,u,p);
(u = u2) ⇒ modify(add(spm,u2,p2),u,p) == add(spm,u,p);
not(u = u2) ⇒ modify(add(spm,u2,p2),u,p) == add(modify(spm,u,p),u2,p2);
declared(empty,u) == false;
(u = u2) ⇒ declared(add(spm,u2,p2),u) == true;
not(u = u2) ⇒ declared(add(spm,u2,p2),u) == declared(spm,u);
correct(empty,u,p) == false;
(u=u2) ⇒ correct(add(spm,u2,p2),u,p) == (p = p2);
not(u=u2) ⇒ correct(add(spm,u2,p2),u,p) == correct(spm,u,p);

STS

• → X
X − add ?u : UserId ?p : Password → X
X − modify ?u : UserId ?p : Password → X
X − declared ?u : UserId !b : Boolean → X
X − correct ?u : UserId ?p : Password !b : Boolean → X

Figure 2.10 : Gestionnaire de mots de passe – Vue Statique

2.2.2 Aspects dynamiques

Ces aspects recouvrent le comportement des composants ainsi que les communications entre composants. Les composants sont ici vus en tant que processus (c'est-à-dire des objets actifs). Seuls les protocoles de communication sont décrits ici, la correspondance entre entrées et sorties, en termes de valeurs, est définie dans une vue statique qui sera liée. Les vues correspondant aux aspects dynamiques sont les vues dynamiques.

Définition 2.8 (Vue dynamique). *Une vue dynamique DV_T d'un composant T est une vue dans laquelle :*

$$\mathcal{S}_{\mathcal{DV}} = V$$
$$f = Id$$
$$\mathcal{K} = \{Ax_T = \Phi \cup \Phi_0 \wedge \forall \sigma \in \Sigma_T . dynamique(\sigma)\}$$

Si l'on s'intéresse aux aspects dynamiques du gestionnaire de mots de passe, on utilise une vue

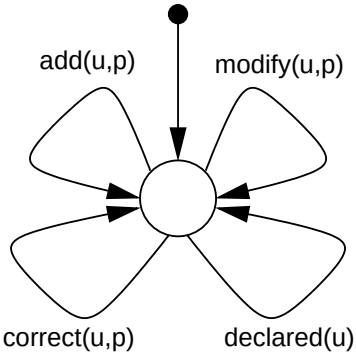


Figure 2.11 : Gestionnaire de mots de passe – STS statique

dynamique. La représentation graphique de la partie dynamique[6] est donnée dans la figure 2.12.

2.2.3 Autres aspects

Le schéma consistant à définir un cas particulier de vue pour un aspect donné peut être réutilisé pour d'autres aspects. Il suffit pour cela que sa sémantique puisse être donnée en termes des trois composants qui constituent une vue. Pour donner quelques exemples, citons les aspects temporels, ou bien les aspects concernant la création ou la destruction des objets qui sont un préalable à la prise en compte d'autres catégories de compositions [187].

2.2.4 Intégration des aspects

Un composant à une *sémantique globale* si chacun de ses aspects est défini. D'autre part, il faut décrire la façon dont les aspects sont composés. En effet, les aspects d'un composant sont liés entre eux et interagissent. La description de l'intégration des aspects est faite à l'aide de la structuration externe qui fournit donc un cadre unique pour la composition (concurrente) et l'intégration d'aspects.

2.3 Structuration externe

Un second moyen de structuration est la structuration externe (figure 2.13), qui exprime le fait qu'un objet peut être composé de plusieurs autres objets. D'autre part, ces objets peuvent être soit simples (intégration des différents aspects de leur structuration interne), soit des objets composites (composition). Dans tous les cas, ils ont une sémantique globale (chacun de leurs aspects est bien défini).

L'intégration des différents aspects d'un objet suit le principe de masquage de l'information[7] (figure 2.14) : les différents aspects (statique pour l'instant) ne peuvent être accédés que via la partie dynamique. Tous les aspects ne sont donc pas au "même" niveau. C'est la partie dynamique des composants qui fournit le protocole de communication entre composants, en se basant entre autre sur les identifiants de ces composants.

Définition 2.9 (External Structuring View). *Une* external structuring view ESV_T *d'un composant* T *est un tuple* (A_T, Θ, K_T) *où* A_T *est une partie données optionnelle,* Θ *est la colle de l'ESV, et* K_T *la partie composition.*

[6] Le super-état VAL correspond à un raccourci syntaxique (voir cette notation dans l'annexe A, figure A.9).

[7] Le masquage d'information inclut, entre autres, le principe d'encapsulation.

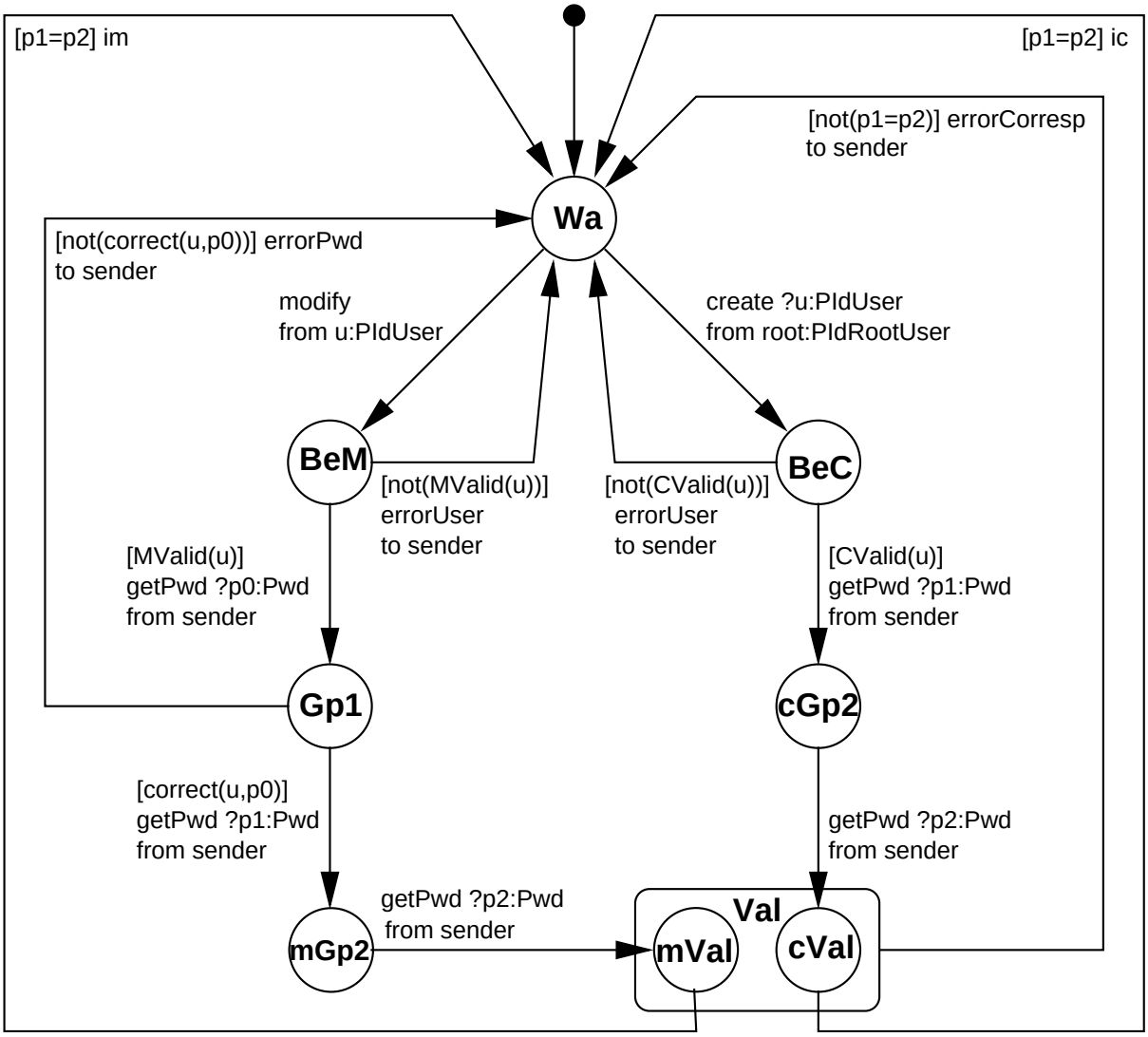


Figure 2.12 : Gestionnaire de mots de passe – STS dynamique

$\mathcal{S}_{\mathcal{ESV}} = (A_T, \Theta, K_T)$
f =voir la section 2.5 sur la sémantique
$\mathcal{K} = \{\}$

La partie donnée (optionnelle) permet de définir de nouveaux paramètres et/ou types de données amorphes qui peuvent être nécessaire dans la composition.

Définition 2.10 (Partie données d'une ESV). *La* partie données A_T *(dans une structure d'ESV* (A_T, Θ, K_T)*) est un tuple* (A', G, V) *où:*

- A' *est une ensemble de spécifications algébriques de base importées (types amorphes) ;*
- G *est un ensemble de spécifications algébriques paramètres formels ;*
- V *est un ensemble de variables paramètres formels .*

Θ décrit la façon dont les composants d'un composite interagissent. Cette colle sert à définir la liaison

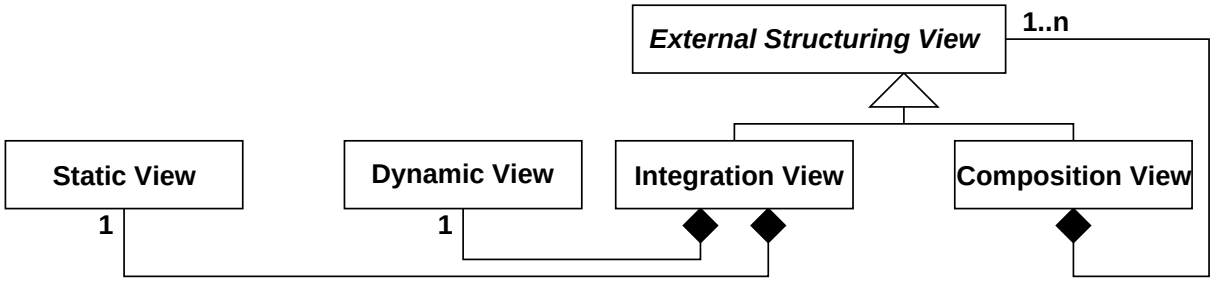


Figure 2.13 : Diagramme de classes – Structuration externe

Figure 2.14 : Principe de masquage d'information dans les intégrations de vues

entre composants réutilisables indépendamment les uns des autres.

Définition 2.11 (Partie colle d'une ESV). *La* partie colle Θ *(dans une structure d'ESV* (A_T, Θ, K_T)*) est un tuple* $(Ax_\Theta, \Phi, \Psi, \Phi_0, \delta)$ *où:*

- Ax_Θ *sont des axiomes qui relient gardes et opérations de différentes vues (voir l'intégrations des aspects statiques et dynamiques du gestionnaire de mots de passe dans la figure 2.23) ;*
- Φ *est un ensemble de formules (d'état) construites à partir de l'*id*-indexation des membres* C *des composants du composite*[8] *;*
- Ψ *est un ensemble de couples de formules (de transition)* id*-indexées qui relient (collent) les transitions (des STS) des différents composants du composite. C'est une forme généralisée du vecteur de synchronisation du produit d'automates [17] ;*
- Φ_0 *est une formule (d'état) qui définit les états initiaux du composite (cette formule doit être cohérente avec les* Φ_0*s des composants du composite, c'est-à-dire* $\vdash \Phi_0 \Rightarrow \bigwedge_{id} id.\Phi_0(Obj_{id})$*) ;*
- δ *définit la façon dont les transitions non liées (par la colle des transitions) sont gérées (voir la Section 2.5.1.1).*

Les formules Φ et Ψ sont implicitement quantifiées par rapport au temps (c'est-à-dire que l'on a[9] $AG\Phi$ et $AG\Psi$); Ψ, qui est donné sous la forme de couples $\Psi = \{\psi_i = (\psi_{i_s}, \psi_{i_d})\}$, signifie $AG \bigwedge_i \psi_{i_s} \Leftrightarrow \psi_{i_d}$. De même, pour Φ, on a $AG \bigwedge_i \phi_i$. Θ utilise les notions d'environnement, de formule d'état et de formule de transition qui sont définies dans la suite.

2.3.1 Environnements

Un environnement Γ lie des variables à des états ou à des transitions.

$$\Gamma : Var \rightarrow (STATE \cup TRANS)$$

Les formules d'état et de transition sont simples : elles ne comportent pas d'opérateurs de point fixe (particuliers comme AG et EF ou généralisés) comme ceux de logiques temporelles plus expressives

[8] Ceci est utilisé par exemple pour exprimer l'exclusion (mutuelle) entre deux conditions/états de deux différents composants d'un composite.

[9] Avec AG signifiant "toujours globalement".

comme CTL [76], le μ-calcul [170] ou XTL [191, 192]. Le point important ici est qu'elles sont suffisamment expressives pour exprimer la colle entre composants. Des logiques plus complexes et expressives pourront être définies plus tard pour vérifier les propriétés des modèles, en suivant par exemple l'approche de [191]. Utiliser de telles logiques pour la colle pose de nombreux problèmes liés essentiellement à :

- le calcul de la sémantique de la composition en termes de structure de vue ;
- la propagation de contraintes engendrées par la liaison de transitions "distantes" dans le temps.

2.3.2 Formules d'état

Le domaine d'interprétation des formules d'état (Φ) est l'ensemble des états du STS. Les règles de déduction (positives) concernant les formules d'état sont données dans la figure 2.15.

$$\Phi ::= c \mid x \mid \exists x.\Phi_1 \mid true \mid false \mid \neg\Phi_1 \mid \Phi_1 \wedge \Phi_2 \mid -\Psi_1 \mid i.\Phi_1$$

$\frac{c(s)}{\Gamma,s\models c}$	$PROP_s$		
$\frac{\{x\mapsto s'\}\in\Gamma, s\sim s'}{\Gamma,s\models x}$	$IDENT_s$	$\frac{\Gamma\cup\{x\mapsto s\},s\models\Phi_1}{\Gamma,s\models\exists x.\Phi_1}$	$BIND_s$
$\overline{\Gamma, s \models true}$	$TRUE_s$	$\overline{\Gamma, s \not\models false}$	$FALSE_s$
$\frac{\Gamma,s\not\models\Phi_1}{\Gamma,s\models\neg\Phi_1}$	NOT_s	$\frac{\Gamma,s\models\Phi_1\wedge\Gamma,s\models\Phi_2}{\Gamma,s\models\Phi_1\wedge\Phi_2}$	AND_s
$\frac{\exists t=s\xrightarrow{l}s'\in T,\Gamma,t\models\Psi_1}{\Gamma,s\models-\Psi_1}$	$TRANS$	$\frac{\exists i.s'\in s\ ,\ \Gamma,s'\models\phi_1}{\Gamma,s\models i.\phi_1}$	$INDEX_s$

Figure 2.15 : Règles de déduction – Formules d'état

Moins formellement, la signification des formules d'état est :

- $PROP_s$: un état s vérifie une condition c (incluse dans l'ensemble C des conditions de la structure de vue correspondante) ssi c est vrai en s (ce qui est dénoté par $c(s)$) ;
- $IDENT_s$: un état s vérifie x (x est une variable) ssi x est liée à un état s' dans l'environnement Γ, et si s et s' sont similaires (ce qui est noté $s \sim s'$). Deux états s et s' sont *similaires* ssi $\forall c \in C\ .\ c(s) \Leftrightarrow c(s')$;
- $BIND_s$: cette formule est utilisée pour lier des variables à des états. Un état s vérifie $\exists x.\Phi_1$ ssi il vérifie Φ_1 dans un environnement où x a été liée à s ;
- $TRUE_s$: tous les états vérifient $true$;
- $FALSE_s$: aucun état ne vérifie $false$;
- NOT_s : un état s vérifie $\neg\Phi_1$ ssi il ne vérifie pas Φ_1 ;
- AND_s : un état s vérifie $\Phi_1 \wedge \Phi_2$ ssi il vérifie à la fois Φ_1 et Φ_2 ;
- $TRANS$: cette formule d'état fait le lien entre formules d'état et formules de transition. Un état s vérifie $-\Psi_1$ ssi il existe une transition t ayant s comme origine et qui vérifie Ψ_1 ;
- $INDEX_s$: un état s vérifie $i.\Phi_1$ ssi il a un sous-composant i qui vérifie Φ_1. Cette règle permet de gérer la structuration (indexation) des composites.

2.3.3 Formules de transition

Le domaine d'interprétation des formules de transition (Ψ) est l'ensemble des transitions du STS. Les règles de déduction (positives) concernant les formules d'état sont données dans la figure 2.16.

$$\Psi ::= [g]l[g'] \mid x \mid \exists x.\Psi_1 \mid true \mid false \mid \neg\Psi_1 \mid \Psi_1 \wedge \Psi_2 \mid > \Phi_1 \mid i.\Psi_1$$

$$\frac{t=s \xrightarrow{[g_t]\ \ l_t\ \ [g'_t]} s', l \sim l_t, g_t \Rightarrow g, g'_t \Rightarrow g'}{\Gamma, t \models [g]l[g']} \quad PROP_t$$

$$\frac{\{x \mapsto t'\} \in \Gamma, t \sim t'}{\Gamma, t \models x} \quad IDENT_t \qquad \frac{\Gamma \cup \{x \mapsto t\}, t \models \Psi_1}{\Gamma, t \models \exists x.\Psi_1} \quad BIND_t$$

$$\frac{}{\Gamma, t \models true} \quad TRUE_t \qquad \frac{}{\Gamma, t \not\models false} \quad FALSE_t$$

$$\frac{\Gamma, t \not\models \Psi_1}{\Gamma, t \models \neg\Psi_1} \quad NOT_t \qquad \frac{\Gamma, t \models \Psi_1 \wedge \Gamma, t \models \Psi_2}{\Gamma, t \models \Psi_1 \wedge \Psi_2} \quad AND_t$$

$$\frac{t=s \xrightarrow{l} s', \Gamma, s' \models \Phi_1}{\Gamma, t \models > \Phi_1} \quad TARGET \qquad \frac{\exists i.t' \in t\ ,\ \Gamma, t' \models \psi_1}{\Gamma, t \models i.\psi_1} \quad INDEX_t$$

Figure 2.16 : Règles de déduction – Formules de transition

Moins formellement, la signification des formules de transition est :

- $PROP_t$: une transition t vérifie une formule $s \xrightarrow{[g_t]\ \ l_t\ \ [g'_t]} s'$ ssi elle a des offres similaires et si les gardes de t impliquent celles de cette formule. Deux offres $l = po_i$ et $l_t = p_t o_{t_j}$ sont *similaires* (ce qui est noté $l \sim l_t$) si les portes de communications ont le même nom ($p = p_t$), si le nombre des paramètres de l'offre est identique ($i = j$) et si ces paramètres correspondent ($\forall i.o_i = o_{t_i}$) ;
- $IDENT_t$: une transition t vérifie x (x est une variable) ssi x est liée à une transition t' dans l'environnement Γ et que que t et t' sont similaires dans l'environnement Γ ($t \sim_\Gamma t'$), c'est-à-dire si $\Gamma, t \models t'$ et $\Gamma, t' \models t$;
- $BIND_t$: cette formule est utilisée pour lier des variables à des transitions. Une transition t vérifie $\exists x.\Psi_1$ ssi elle vérifie Ψ_1 dans un environnement où x a été liée à t ;
- $TRUE_t$: tout transition vérifie $true$;
- $FALSE_t$: aucune transition ne vérifie $false$;
- NOT_t : une transition t vérifie $\neg\Psi_1$ ssi elle ne vérifie pas Ψ_1 ;
- AND_t : une transition t vérifie $\Psi_1 \wedge \Psi_2$ ssi elle vérifie à la fois Ψ_1 et Ψ_2 ;
- $TARGET$: cette formule de transition fait le lien entre formules de transition et formules d'état. Une transition t vérifie $> \Phi_1$ ssi elle part d'un état s et arrive dans un état s' tel que s' vérifie Φ_1 ;
- $INDEX_t$: une transition t vérifie $i.\Psi_1$ ssi elle a un sous-composant i qui vérifie Ψ_1. Cette règle permet de gérer la structuration (indexation) des composites.

Voici un exemple qui explicite le concept de liaison/identification des formules d'état et de transition, ainsi que le lien entre ces deux types de formules. On désire exprimer la propriété de pouvoir partir d'un état (quelconque) puis de faire deux fois une même transition (quelconque) et revenir dans l'état de départ. Informellement, ceci correspond au schéma de la figure 2.17.

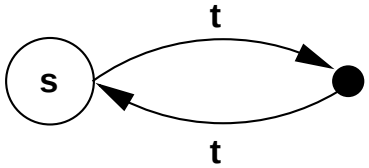


Figure 2.17 : Formules temporelles – Exemple

Pour exprimer cette propriété, on utilise donc la formule suivante :

$$\exists s. - \exists t. > - t \wedge \ > s$$

Comme habituellement, $\models$ peut être défini pour les formules d'état comme $s \models \Phi$ ssi $\{\}, s \models \Phi$, et pour les formules de transitions comme $t \models \Psi$ ssi $\{\}, t \models \Psi$. De plus, par abus quand Φ désigne un ensemble de formules, on écrit parfois $s \models \Phi$ pour $\forall \phi_i \in \Phi \ . \ s \models \phi_i$.

Les environnements et les formules étant définis, il est possible de donner la définition des parties composition des ESV.

Définition 2.12 (Partie composition d'une ESV). *La* partie composition K_T *(dans une structure d'ESV* (A_T, Θ, K_T)*) est un tuple* (Id, Obj_{id}, I_{id}) *où :*

- Id *est un identifiant de sorte* PId_T *;*
- Obj_{id} *est une famille* id*-indexée de composants (integration views ou composition views). Les identifiants respectent les principes d'égalité et de structure d'objet (voir plus haut) ;*
- I_{id} *sont des ensembles* id*-indexés optionnels de termes instanciant les paramètres des composants.*

La syntaxe des ESV en KORRIGAN est donnée dans la figure 2.18. L'identifiant de l'ESV (Id) n'apparaît que quand le composite est lui-même composant dans un autre composite.

EXTERNAL STRUCTURING VIEW T	
SPECIFICATION	**COMPOSITION** δ
imports A'	**is**
generic on G	$id_i : Obj_i[I_i]$
variables V	**axioms** Ax_Θ
	with Φ, Ψ **initially** Φ_0

Figure 2.18 : Syntaxe KORRIGAN– ESV

Il existe un raccourci syntaxique qui peut être utilisé dans les compositions de composants (ESV) : l'opérateur *range*.

$$i : [x..y].f(i)$$

Cet opérateur est un quantificateur universel sur un ensemble d'identifiants. Il peut servir à la fois dans les formules et dans les énumérations de composants (Obj_i).

Prenons l'exemple d'une voiture, composée de quatre roues (figure 2.19).

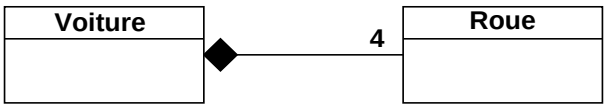


Figure 2.19 : Diagramme de classes – Voitures

Dans le cadre d'une ESV (ce qui est un abus, les ESV étant abstraites c'est avec une composition view que les voitures seraient décrites), cette composition serait décrite par le schéma KORRIGAN de la figure 2.20. L'opérateur range est aussi ici utilisé dans la partie `with` pour exprimer que toutes les roues sont en même temps immobiles ou en mouvement (en supposant que la spécification des roues utilise une condition `moving` pour cela). L'ensemble des roues est donc similaire à cette abstraction conditionnelle près. Il faut noter, que via l'utilisation des paramètres des ESV, il aurait aussi été possible de remplacer

le chiffre quatre par un paramètre formel de type variable, et ainsi de pouvoir parler des véhicules avec un nombre quelconque de roues (figure 2.21). D'autre part, il faudrait aussi spécifier que toutes les roues bougent en même temps (opérateur range utilisé dans la partie Ψ). Bien sur, il est aussi possible de ne pas spécifier la synchronisation entre les roues au niveau de leur état (Φ) mais plutôt de la prouver à partir de la synchronisation au niveau des communications qui leurs sont adressées (Ψ).

EXTERNAL STRUCTURING VIEW VOITURE
COMPOSITION
is i:[1..4].roue.i : ROUE **with** i:[1..4].j:[1..4].(roue.i.moving ⇔ roue.j.moving), ...

Figure 2.20 : KORRIGAN– Voitures à 4 roues

EXTERNAL STRUCTURING VIEW VOITURE2	
SPECIFICATION	**COMPOSITION**
variables nbRoues:NATURAL	**is** i:[1..nbRoues].roue.i : ROUE **with** i:[1..nbRoues].j:[1..nbRoues]. (roue.i.moving ⇔ roue.j.moving), ...

Figure 2.21 : KORRIGAN– Voitures à `nbRoues` roues

2.3.4 Integration Views

Une Integration View (IV) est un cas particulier d'ESV utilisé pour décrire un composant ayant une sémantique globale (voir section 2.5) à partir de l'intégration de tous les aspects de ce composant. Les IV ont la même structure que les ESV avec la contrainte qu'elles sont composées d'exactement une vue par aspect du composants décrit (c'est-à-dire une vue de chaque type des sous-classes d'ISV). Pour le moment, cela correspond à une vue statique et une vue dynamique, identifiées respectivement par s et d.

$\mathcal{S}_{\mathcal{IV}} = \mathcal{S}_{\mathcal{ESV}}$
f =voir la section 2.5 sur la sémantique
$\mathcal{K} = \{\#Obj_{id} = \#subclasses(ISV) \wedge \forall s \in subclasses(ISV)\ .\ \exists x : s \in Obj_{id}\}$

La syntaxe KORRIGAN des IV est donnée dans la figure 2.22.

INTEGRATION VIEW T	
SPECIFICATION	**COMPOSITION** δ
imports A' **generic on** G **variables** V	**is** STATIC s: $Obj_s[I_s]$ DYNAMIC d: $Obj_d[I_d]$ **axioms** Ax_Θ **with** Φ, Ψ **initially** Φ_0

Figure 2.22 : Syntaxe KORRIGAN– IV

Reprenons l'exemple du gestionnaire de mots de passe. Une IV est utilisée pour le définir globalement, c'est-à-dire à partir de son aspect statique (voir le STS, figure 2.11) et de son aspect dynamique (voir le STS, figure 2.12). Cette IV est donnée dans la figure 2.23.

INTEGRATION VIEW Password Manager

COMPOSITION ALONE

is
STATIC VIEW s: SPM
DYNAMIC VIEW d: DPM
axioms
d.MValid(s,u) == s.declared(s,u)
d.CValid(s,u) == not(s.declared(s,u))
d.correct(s,u,p) == s.correct(s,u,p)
with true, {
(d.([true] im), s.(modify(d.u,d.p1))),
(d.([true] ic), s.(add(d.u,d.p1)))
}
initially true

Figure 2.23 : Gestionnaire de mots de passe – Intégration des aspects

2.3.5 Composition Views

Une composition view (CV) est une ESV qui peut contenir un ou plusieurs composants ayant une sémantique globale (*i.e.* integration views or composition views). La composition correspond informellement à l'union de sous-composants au sein d'un composite. La structure des CV ainsi que leur syntaxe est la même que pour les ESV.

$\mathcal{S}_{\mathcal{CV}} = \mathcal{S}_{\mathcal{ESV}}$
f =voir la section 2.5 sur la sémantique
$\mathcal{K} = \{\#Obj_{id} \geq 2 \wedge \forall x \in Obj_{id} \,.\, \exists s \in subclasses(ESV) \,.\, x : s\}$

Un exemple de CV est la voiture vue plus haut.

2.4 Structuration d'héritage

L'héritage [202] consiste généralement à ajouter des méthodes à une classe. Le cas consistant à ajouter des attributs peut être assimilé au fait d'ajouter les observateurs correspondant à chacun des attributs. Mais l'héritage peut aussi servir à ajouter des contraintes. Enfin il peut permettre de surcharger ou masquer des méthodes. Pour l'instant nous en avons une forme simple, basée sur notre structure de vue.

La structuration d'héritage s'applique aux vues de base (ISV). Une contrainte importante est que les vues reliées par l'héritage soient (exactement) du même type. Ainsi, une DV ne peut pas hériter d'une SV.

Par *héritage* nous entendons une forme simple d'héritage qui consiste à ajouter dans une sous-classe de nouvelles conditions et de nouvelles opérations qui peuvent utiliser ces nouvelles conditions. L'hé-

ritage est possible au niveau des vues statiques et dynamiques. Notre forme d'héritage ne prend pas en compte la surcharge.

De plus, la sous-classe comprend toujours les opérations de sa super-classe ainsi que ses conditions. Il n'y a donc pas de masquage. Pour ces raisons, de nouveaux états, correspondant à chaque combinaison possible des nouvelles conditions, vont apparaître dans la sous-classe. Ces états sont hiérarchiques dans le sens où chacun d'entre eux inclut implicitement le STS complet de la super-classe. De ce fait, au niveau de la réprésentation de la sous-classe en termes d'un STS, uniquement les nouveaux états et les nouvelles transitions sont explicitement décrites.

Cette forme d'héritage est relativement simple mais est d'un usage pratique et est bien définie.

Comme exemple nous prenons un service de messagerie téléphonique décrit en annexe C. Nous avons des utilisateurs simples qui peuvent uniquement être appelés. Il existe d'autre part des utilisateurs abonnés qui peuvent aussi appeler d'autres utilisateurs (simples ou abonnés).

Nous commençons par décrire le comportement d'utilisateurs de base (*i.e.* utilisateurs simples sans possibilité d'abonnement).

La figure 2.24 donne la vue dynamique, en KORRIGAN, de tels utilisateurs.

DYNAMIC VIEW D-BASIC-USER	
SPECIFICATION	**STS**
imports PId, MSG **generic on** server:PIdServer **ops** called ?p:PIdUser **from** server ok **to** server nok **to** server comm ?m:Msg **from** PIdUser comm !m:Msg **to** PIdUser eot **to** server eot **from** server	• → IDLE IDLE − called ?p:PIdUser **from** server → ASKED ASKED − ok **to** server → COMM ASKED − nok **to** server → IDLE COMM − comm ?m:Msg **from** p → COMM COMM − comm !m:Msg **to** p → COMM COMM − eot **to** server → IDLE COMM − eot **from** server → IDLE

Figure 2.24 : Utilisateurs de base – Vue dynamique

La figure 2.25 donne une représentation graphique du STS de cette vue dynamique.

La prise en compte des possibilités d'abonnement et de résiliation, ainsi que des fonctionnalités des utilisateurs abonnés peut se faire en spécifiant un comportement d'utilisateur complet, étendant celui des utilisateurs de base. Concrètement, ceci peut être décrit en utilisant l'héritage et en ajoutant les nouvelles conditions et les nouvelles opérations correspondant à ces nouvelles fonctionalités.

La figure 2.26 donne la vue dynamique, en KORRIGAN, des utilisateurs complets. Cette vue hérite de celle des utilisateurs de base. Elle peut donc uniquement ajouter de nouvelles conditions et opérations utilisant ces conditions. Ici nous avons une nouvelle condition, `subscribed`, qui est vraie lorsque l'utilisateur a souscrit un abonnement, et faux sinon.

Le STS correspondant est défini en utilisant des états hiérarchiques pour chaque valeur possible de la nouvelle condition. Ici il n'y a qu'une nouvelle condition, c'est pourquoi le STS contient deux états hiérarchiques (`subscribed` et `not subscribed`). Chacun des deux états hiérarchiques du STS de

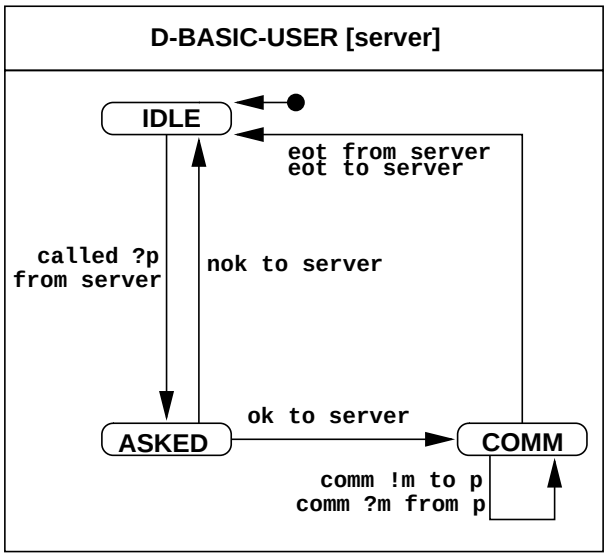


Figure 2.25 : Utilisateurs de base – STS dynamique

la sous-classe contient le STS complet de la vue (dynamique) dont il hérite. C'est pourquoi les états contenus dans ces états hiérarchiques peuvent être référés en utilisant notre notation préfixée habituelle (ex. `subscribed.IDLE` pour l'état `IDLE` dans l'état hiérarchique `subscribed`).

De plus, il est possible d'utiliser les noms d'états hérités en lieu et place des formules de conditions correspondantes dans les préconditions et postconditions des opérations. Ainsi, par exemple, les préconditions de l'opération `subscribe` précisent que la condition `subscribed` doit être fausse et le statut `IDLE`. Lorsque la valeur des conditions n'est pas modifiée par l'opération (c'est-à-dire lorsqu'il existe une postcondition `c':c` pour une certaine condition `c`), nous l'omettons (elle est alors implicite).

La figure 2.27 donne une réprésentation graphique du STS de cette vue dynamique.

2.5 Sémantique

Les External Structuring Views (ESV) ont une sémantique globale dans le sens où elles décrivent des objets dont l'ensemble des aspects (ceux liés à la structuration interne) sont définis. Pour cela, il nous faut premièrement expliquer comment il est possible d'obtenir une structure de vue (définition 2.2) pour une ESV. Une telle vue est qualifiée de globale pour les mêmes raisons que la sémantique de cette ESV. Dans un second temps, nous aurons à donner la sémantique d'une vue globale. Une autre façon de faire aurait été de définir directement une sémantique globale à partir d'une structure d'ESV.

2.5.1 Obtention d'une structure de vue pour les ESV

Les structures de vues associées aux ESV sont un type particulier de vue où les états (respectivement les transitions) sont construits comme des compositions indexées d'états (respectivement de transitions). Cette indexation nous permet de conserver la structuration interne du système tout en "collant" les composants entre eux. Ce collage est utilisé pour permettre de spécifier comment les états et les transitions des différents aspects (statiques et dynamiques) sont reliés.

DYNAMIC VIEW D-FULL-USER EXTENDS D-BASIC-USER

SPECIFICATION

ops
subscribe **to** server
unsubscribe **to** server
ok **from** server
nok **from** server
addc !v:PIdUser **to** server
subc !v:PIdUser **to** server
call !p:PIdUser **to** server

ABSTRACTION

conditions subscribed
initially ¬ subscribed

OPERATIONS

subscribe
pre: IDLE ∧ ¬ subscribed
post: {subscribed':true}
unsubscribe
pre: IDLE ∧ subscribed
post: {subscribed':false}
ok
pre: ASKING ∧ subscribed
post: {COMM':true}
nok
pre: ASKING ∧ subscribed
post: {IDLE':true}
addc !v:PIdUser, subc !v:PIdUser
pre: IDLE ∧ subscribed
post: {}
call !p:PIdUser
pre: IDLE ∧ subscribed
post: {ASKING':true}

Figure 2.26 : Utilisateurs complets - Vue dynamique

Nous désirons donner des structures de vues aux ESV (*i.e.* définir les trois composants de la définition 2.2 à partir des éléments constituant une ESV, définition 2.9) de façon à pouvoir les composer à leur tour dans d'autres ESVs. Il est possible de travailler à l'obtention d'une structure de vue soit dans sa forme (A_T, α, D_T), soit dans sa forme alternative (A_T, STS) (avec $STS = (S, S_0, T)$). Ces deux façons de procéder sont décrites dans la figure 2.28.

La première façon de faire consiste à utiliser les structures de vues des composants de l'ESV et à obtenir (flèche A dans la figure 2.28) une structure de vue pour l'ESV. La seconde façon de faire consiste à utiliser les structures de vues alternatives (utilisant des STS) des composants de l'ESV et à obtenir (flèche B dans la figure 2.28) une structure de vue alternative pour l'ESV. Étant données les fonctions permettant de faire correspondre structures de vues et structures de vues alternatives (ici f et g), le diagramme doit commuter. Nous travaillerons donc sur l'obtention d'une structure alternative (A_T, STS) à partir de la structure d'ESV (A_T, Θ, K_T). En effet, travailler directement sur les STS est plus aisé. Les règles que nous utilisons pour la partie STS peuvent être comparées au produit synchronisé [17]. Son vecteur de synchronisation est ici remplacé par la partie Θ des ESV, qui est plus générale. De plus, nous construisons uniquement les transitions structurées qui correspondent à la colle, alors que [17] calcule un produit libre puis supprime les transitions illégales et les états non atteignables.

2.5.1.1 Obtention de la partie STS

Θ (dans les structures d'ESV, définition 2.11) définit les règles pour coller états et transitions. Il reste à définir ce qu'il convient de faire des transitions qui n'ont pas à être collées (*transitions solitaires*). C'est le rôle du composant δ de Θ. Nous proposons trois différentes possibilités :

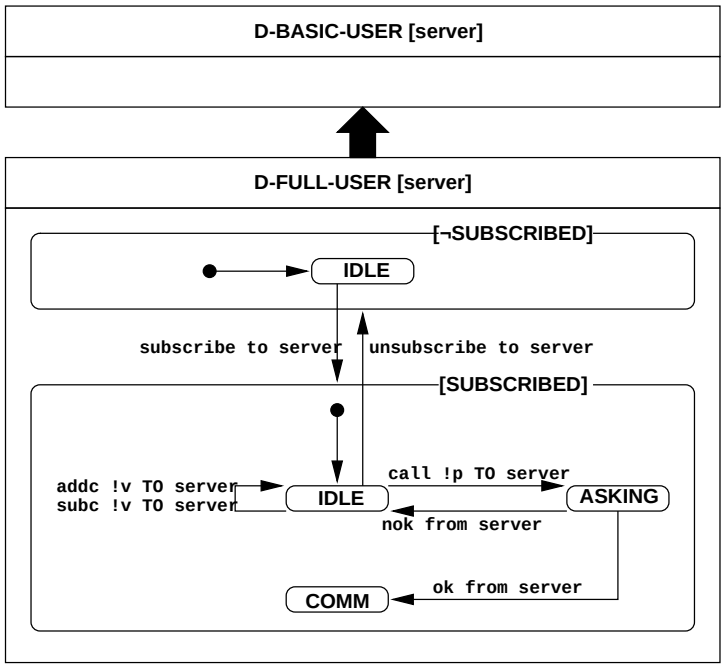


Figure 2.27 : Utilisateurs complets – STS dynamique

– KEEP : chaque transition solitaire peut survenir seule (collée à une transition "triviale" dénotée par ϵ), ou collée avec n'importe quelle autre transition solitaire ;
– ALONE : comme en LOTOS, chaque transition solitaire peut survenir, mais seule ;
– LOOSE : les transitions solitaires sont éliminées.

Prenons un cas simple pour montrer la différence entre ces trois façons de coller.

Supposons que l'on ait deux composants, c_1 et c_2 dont les STS sont donnés dans la figure 2.29.

Si l'on a $\Psi = \{(c_1.a, c_2.a)\}$ et que l'on fait le choix LOOSE, on obtiendra le STS de la figure 2.30 dans lequel seules les transitions ayant été collées restent.

Si l'on fait le choix ALONE, on obtiendra le STS de la figure 2.31 où les transitions non collées surviennent seules (collées avec des transitions ϵ).

Enfin, en faisant le choix KEEP, on obtient le STS de la figure 2.32, qui correspond à celui obtenu avec le choix ALONE, mais en autorisant aussi le collage entre transitions solitaires.

Les règles d'obtention (figures 2.33, 2.34, 2.35 et 2.36) ont été implémentées dans la partie CLAP de l'atelier décrit dans le chapitre 4.

Les états initiaux du STS global sont construits (règle $INIT$) à partir de couples d'états initiaux indexés des vues statique et dynamique (dénoté par $\langle\ s.s_{0_s}/d.s_{0_d}\ \rangle$) qui vérifient les formules (d'état) des composants Φ et Φ_0 de la structure d'ESV. L'ensemble des états est constitué des états initiaux (règle $STATE_0$) et des états cibles des transitions (règle $STATE_1$).

Un couple de transitions (une transition de la vue statique, une transition de la vue dynamique) qui vérifie un des couples des formules de transitions du composant Ψ de la structure d'ESV est collé en un transition composée (dénotée par $\langle\ s.s_s/d.s_d\ \rangle \xrightarrow{[\langle\ s.g_s/d.g_d\ \rangle]\ \langle\ s.l_s/d.l_d\ \rangle\ [\langle\ s.gio_s/d.gio_d\ \rangle]} \langle\ s.s'_s/d.s'_d\ \rangle$) si :

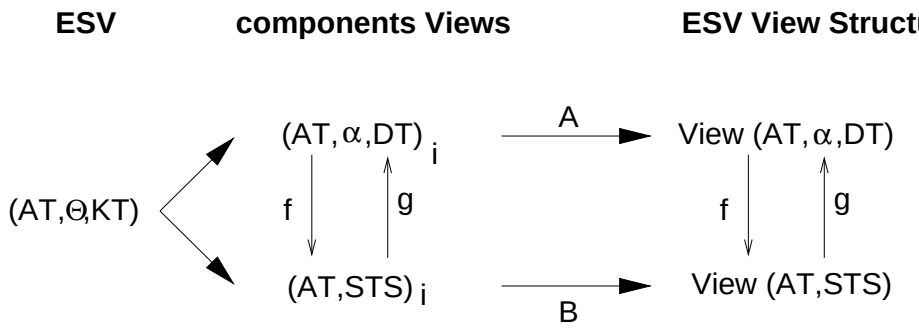


Figure 2.28 : Obtention d'une structure de vue pour les ESV

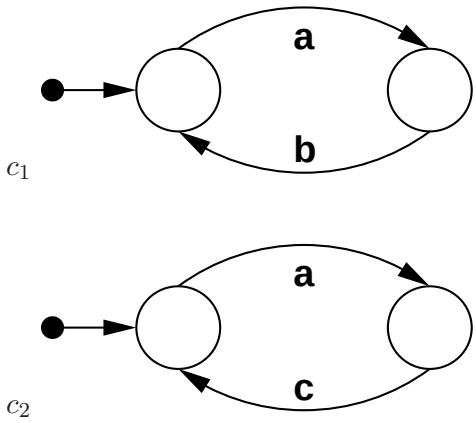


Figure 2.29 : LOOSE, ALONE et KEEP – Composants de l'exemple

– l'état source composé ($\langle\ s.s_s/d.s_d\ \rangle$) correspondant (composé à partir des deux états source) est un état valide, c'est-à-dire un état global initial ou état global destination d'une transition déjà construite ;
– l'état destination composé ($\langle\ s.s'_s/d.s'_d\ \rangle$) corresponsant (composé à partir des deux états destination) vérifie la formule (d'état) du composant Φ de la structure d'ESV.

La composition d'états et de transitions utilise l'indexation pour conserver la structure du système.

Deux autres règles vont permettre de prendre en compte les sémantiques correspondant à KEEP (figure 2.35) et ALONE (figure 2.36).

Un de ces règles peut être ajoutée à la règle LOOSE pour prendre en compte la sémantique correspondante (voir la table 2.2).

semantique	règles
LOOSE	INIT, STATE_0, STATE_1, $\text{TRANSACTION}_{LOOSE}$
KEEP	INIT, STATE_0, STATE_1, $\text{TRANSACTION}_{LOOSE}$, $\text{TRANSACTION}_{KEEP}$
ALONE	INIT, STATE_0, STATE_1, $\text{TRANSACTION}_{LOOSE}$, $\text{TRANSACTION}_{ALONE}$

Table 2.2 : Relations entre sémantiques et règles

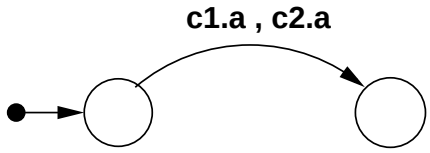


Figure 2.30 : LOOSE, ALONE et KEEP – LOOSE

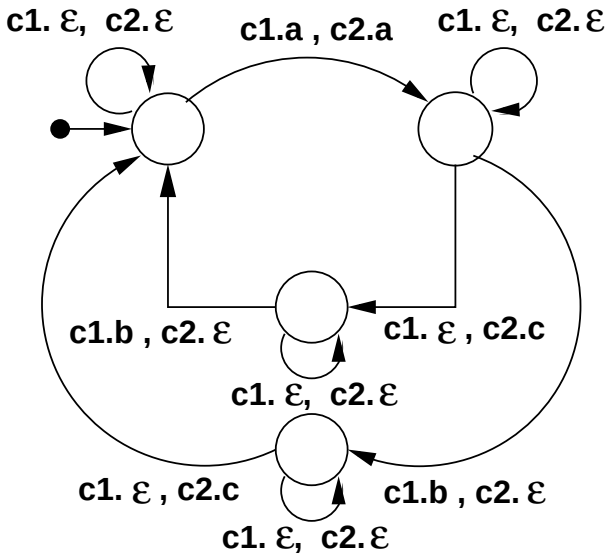


Figure 2.31 : LOOSE, ALONE et KEEP – ALONE

En ce qui concerne le gestionnaire de mots de passe, à partir des STS de base des figures 2.11 et 2.12, et à partir de l'intégration décrite dans la figure 2.23, on obtient le STS global de la figure 2.37.

2.5.1.2 Obtention de la partie A_T

Les règles utilisées pour la partie données ($A_T = (A', G, V, \Sigma_T, Ax_T, \overline{A})$) utilisent l'indexation pour construire un type produit (statique) pour A_T:

- $A' = A'(ESV) \cup I_{G_i} \bigcup_i A'(Obj_i(ESV))$, les I_{G_i} étant les instanciations des I_i de $G(Obj_i(ESV))$
- $G = \bigcup_i i.G(Obj_i(ESV))$, en supprimant les éléments de $G(Obj_i(ESV))$ instanciés
- $V = \bigcup_i i.V(Obj_i(ESV))$, en supprimant les éléments de $V(Obj_i(ESV))$ instanciés
- $\Sigma_T = \bigcup_i i.\Sigma(Obj_i(ESV))$
- $Ax_T = Ax_\Theta \bigcup_i i.Ax(Obj_i(ESV))$. Un travail de renommage et d'indexation doit être fait sur les axiomes
- $\overline{A} = \overline{A} \bigcup_i \overline{A}(Obj_i(ESV))$

Le renommage des axiomes consiste à indexer chaque opération où le type d'intérêt (le type du composant dont on indexe les axiomes) est en première position ("receveur").

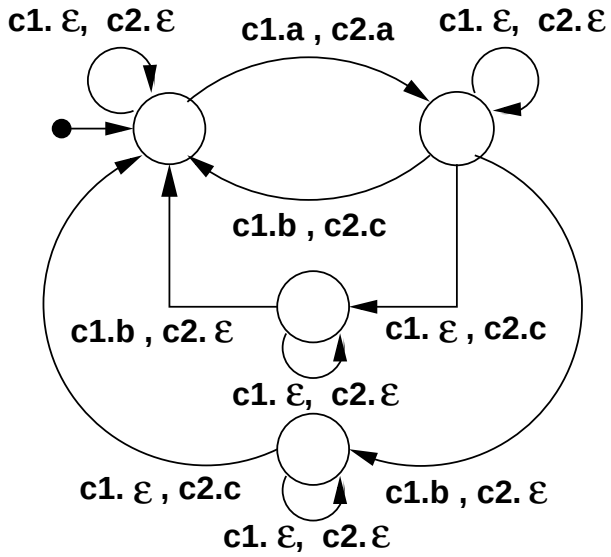


Figure 2.32 : LOOSE, ALONE et KEEP – KEEP

$$\frac{s_{0_s} \in S_0(Obj_s), s_{0_d} \in S_0(Obj_d), \langle\ s.s_{0_s}/d.s_{0_d}\ \rangle \models \Phi \wedge \Phi_0}{\langle\ s.s_{0_s}/d.s_{0_d}\ \rangle \in S_0} \quad INIT$$

$$\frac{s \in S_0}{s \in S}\ STATE_0 \qquad \frac{s \in S,\ s \xrightarrow{l} s' \in T}{s' \in S}\ STATE_1$$

Figure 2.33 : Règles d'obtention des états

2.5.2 Sémantique des ESV

La sémantique est donnée de façon opérationnelle et selon une approche basée sur les systèmes de transition et la réécriture. Le profil de la fonction sémantique est : $Term \times \Gamma \times STATE \times EVENT \rightarrow Term \times \Gamma \times STATE$, où $EVENT$ est un évènement de communication (offre "à la LOTOS"). Cette fonction est écrite : $\boxed{t, \Gamma, s} \xrightarrow{e} \boxed{t', \Gamma', s'}$. Ici les environnements (Γ) lient des termes aux variables.

Les composants sont initialement instanciés. Ceci est dénoté par l'application d'un générateur $*$ à un vecteur $\vec{t}$ de termes. Ici $\overset{Ax,\Gamma}{\downarrow}(t)$ dénote la forme normale du terme t, en utilisant les axiomes Ax dans l'environnement Γ.

$$\textbf{if } \langle\ s_s/s_d\ \rangle \in S_0 \textbf{ then } \xrightarrow{*(\vec{t})} \boxed{\overset{Ax_T}{\downarrow}(*(\vec{t})), \{\}, \langle\ s_s/s_d\ \rangle}$$

Ensuite, des transitions peuvent subvenir. $\triangleleft$ représente l'union disjointe d'environnement, et $unify$ est une fonction qui unifie deux termes et retourne un environnement dans lequel ces deux termes sont équivalents (à une substitution près). Enfin, $\overline{c_d}$ dénote l'évènement qui correspond (à travers la communication) à c_d. Nous utilisons des suffixes pour dénoter les entrées (i) et les sorties (o), et des indices pour dénoter les parties statiques ($._s$) et dynamiques ($._d$).

$$\frac{\begin{array}{c} t_s = s_s \xrightarrow{[g_s]\ \ l_s\ \ [gio_s]} s'_s \in T(Obj_s), \\ t_d = s_d \xrightarrow{[g_d]\ \ l_d\ \ [gio_d]} s'_d \in T(Obj_d), \\ \exists(\psi_{i_s}, \psi_{i_d}) \in \psi \ \mid\ ts \models \psi_{i_s}\ ,\ td \models \psi_{i_d}, \\ s = \langle\ s.s_s/d.s_d\ \rangle, s' = \langle\ s.s'_s/d.s'_d\ \rangle, \\ g = \langle\ s.g_s/d.g_d\ \rangle, gio = \langle\ s.gio_s/d.gio_d\ \rangle, l = \langle\ s.l_s/d.l_d\ \rangle, \\ s \in S\ ,\ s' \models \phi \end{array}}{s \xrightarrow{[g]\ \ l\ \ [gio]} s' \in T} \quad \begin{array}{r} TRANSACTION \\ (\texttt{LOOSE}) \end{array}$$

Figure 2.34 : Règle LOOSE d'obtention des transitions

$$\frac{\begin{array}{c} t_s = s_s \xrightarrow{[g_s]\ \ l_s\ \ [gio_s]} s'_s \in T(Obj_s) \cup \{\epsilon\}, \\ t_d = s_d \xrightarrow{[g_d]\ \ l_d\ \ [gio_d]} s'_d \in T(Obj_d) \cup \{\epsilon\}, \\ \nexists(\psi_{i_s}, \psi_{i_d}) \in \psi \ \mid\ t_s \models \psi_{i_s}, \\ \nexists(\psi_{i_s}, \psi_{i_d}) \in \psi \ \mid\ t_d \models \psi_{i_d}, \\ s = \langle\ s.s_s/d.s_d\ \rangle, s' = \langle\ s.s'_s/d.s'_d\ \rangle, \\ g = \langle\ s.g_s/d.g_d\ \rangle, gio = \langle\ s.gio_s/d.gio_d\ \rangle, l = \langle\ s.l_s/d.l_d\ \rangle, \\ s \in S\ ,\ s' \models \phi \end{array}}{s \xrightarrow{[g]\ \ l\ \ [gio]} s' \in T} \quad \begin{array}{r} TRANSACTION \\ (\texttt{KEEP}) \end{array}$$

Figure 2.35 : Règle KEEP d'obtention des transitions

$$\textbf{if}$$

$$\langle\ s_s/s_d\ \rangle \xrightarrow{[\langle\ g_s/g_d\ \rangle]\ \ \langle\ c_s\ \vec{i_s}\ \vec{o_s}/c_d\ \vec{i_d}\ \vec{o_d}\ \rangle\ \ [\langle\ gio_s/gio_d\ \rangle]} \langle\ s'_s/s'_d\ \rangle \in T$$

$$\sigma_{event} = unify(c_d\ \ \vec{i_d}\ \vec{o_d}\ ,\ \overline{c_d}\ \ \vec{ai}\ \vec{ao})\ \wedge\ \sigma_{event} \neq \emptyset$$

$$\sigma_{s/d} = unify(c_s\ \ \vec{i_s}\ \vec{o_s}\ ,\ c_d\ \ \vec{i_d}\ \vec{o_d})\ \wedge\ \sigma_{s/d} \neq \emptyset$$

$$\Gamma' = \Gamma \triangleleft \sigma_{event} \triangleleft \sigma_{s/d}$$

$$\overset{Ax_T,\Gamma'}{\downarrow}\ (g_s) \wedge \overset{Ax_T,\Gamma'}{\downarrow}\ (g_d) \wedge \overset{A',\Gamma'}{\downarrow}\ (gio_s) \wedge \overset{A',\Gamma'}{\downarrow}\ (gio_d)$$

$$\textbf{then}$$

$$\boxed{t\ ,\ \Gamma\ ,\ \langle\ s_s/s_d\ \rangle} \xrightarrow{\overline{c_d}\ \vec{ai}\ \vec{ao}} \boxed{\overset{Ax_T,\Gamma'}{\downarrow}\ (c_s(t, \vec{i_s}, \vec{o_s}))\ ,\ \Gamma'\ ,\ \langle\ s'_s/s'_d\ \rangle}$$

Cette règle exprime qu'à chaque fois qu'une transition ayant l'état courant comme source peut être appliquée, alors elle l'est. Cette application conduit à l'obtention d'un terme et d'un environnement nouveaux (courants). Une transition peut être appliquée si, (i) il existe un évènement qui peut s'unifier avec la transition dynamique et si, (ii) après l'unification des transitions statiques et dynamique toutes les gardes sont vraies.

$$\frac{\begin{array}{c} t_s = s_s \xrightarrow{[g_s]\ \ l_s\ \ [gio_s]} s'_s \in T(Obj_s) \cup \{\epsilon\}, \\ t_d = s_d \xrightarrow{[g_d]\ \ l_d\ \ [gio_d]} s'_d \in T(Obj_d) \cup \{\epsilon\}, \\ \nexists(\psi_{i_s}, \psi_{i_d}) \in \psi \ \mid\ t_s \models \psi_{i_s}, \\ \nexists(\psi_{i_s}, \psi_{i_d}) \in \psi \ \mid\ t_d \models \psi_{i_d}, \\ s = \langle\ s.s_s/d.s_d\ \rangle, s' = \langle\ s.s'_s/d.s'_d\ \rangle, \\ g = \langle\ s.g_s/d.g_d\ \rangle, gio = \langle\ s.gio_s/d.gio_d\ \rangle, l = \langle\ s.l_s/d.l_d\ \rangle, \\ s \in S\ ,\ s' \models \phi \\ t_s = \epsilon \vee t_d = \epsilon \end{array}}{s \xrightarrow{[g]\ \ l\ \ [gio]} s' \in T} \quad \begin{array}{r} TRANSACTION \\ (\texttt{ALONE}) \end{array}$$

Figure 2.36 : Règle ALONE d'obtention des transitions

2.6 Conclusions

Dans ce chapitre, nous avons présenté notre modèle hiérarchique à base de vues pour la spécification de systèmes mixtes [70]. Le langage formel associé, KORRIGAN, a aussi été introduit sur plusieurs exemples.

Notre modèle permet de spécifier les différents aspects des systèmes (statique, dynamique et composition) en suivant une approche hétérogène basée sur des systèmes de transitions abstraits, les *systèmes de transitions symboliques*, ainsi que sur des spécifications algébriques. Ces deux parties, qui composent la structure unificatrice que nous appelons *vue*, sont fortement liées, et assurent à la fois une bonne expressivité (puisque conservant les avantages des langages dédiés à chacuns des aspects) et une bonne lisibilité (formalisme graphique) de la spécification des aspects. L'emploi des systèmes de transitions symboliques assure aussi un bon niveau d'abstraction des spécifications et évite l'explosion d'états liée aux systèmes de transitions habituels.

Lorsqu'il s'agit de spécifier des systèmes de taille réelle, il est important de disposer de moyens de structuration des spécifications.

Nous avons pour cela défini un premier niveau de structuration, que nous appelons structuration interne, permettant la spécification des différents aspects de base des systèmes (statique, dynamique). Un second niveau, appelé structuration externe, permet de façon unifiée de spécifier l'intégration des aspects au sein de composants globaux (ensemble des aspects définis) ainsi que la composition de composants globaux concurrents et communicants. La structuration externe utilise un mécanisme de *colle* à la fois à base d'axiomes et de logique temporelle, ce qui permet une grande expressivité dans l'expression de la façon dont plusieurs vues interagissent. La lisibilité est aussi préservée puisque la logique temporelle est simple et sans opérateurs de point fixe. Enfin, nous avons défini un dernier niveau de structuration, permettant une forme simple d'héritage des structures de vue.

Nous avons défini une sémantique opérationnelle pour notre modèle [69]. Cette sémantique est basée sur plusieurs ensembles de règles permettant d'exprimer, dans un cadre de temps logique, les différentes sémantiques de concurrence : par entrelacement ou réelle.

La spécification à l'aide de notre modèle se fait donc de façon unifiée, intégrée, et structurée. Le modèle à base de vues permet à la fois un bon niveau d'expressivité et de lisibilité, et favorise la définition de composants à un haut niveau d'abstraction.

Nous avons dans ce chapitre fait une proposition destinée à répondre à la première partie de la problématique développée en introduction et dans le chapitre 1. Comme nous avons pu déjà le préciser, la

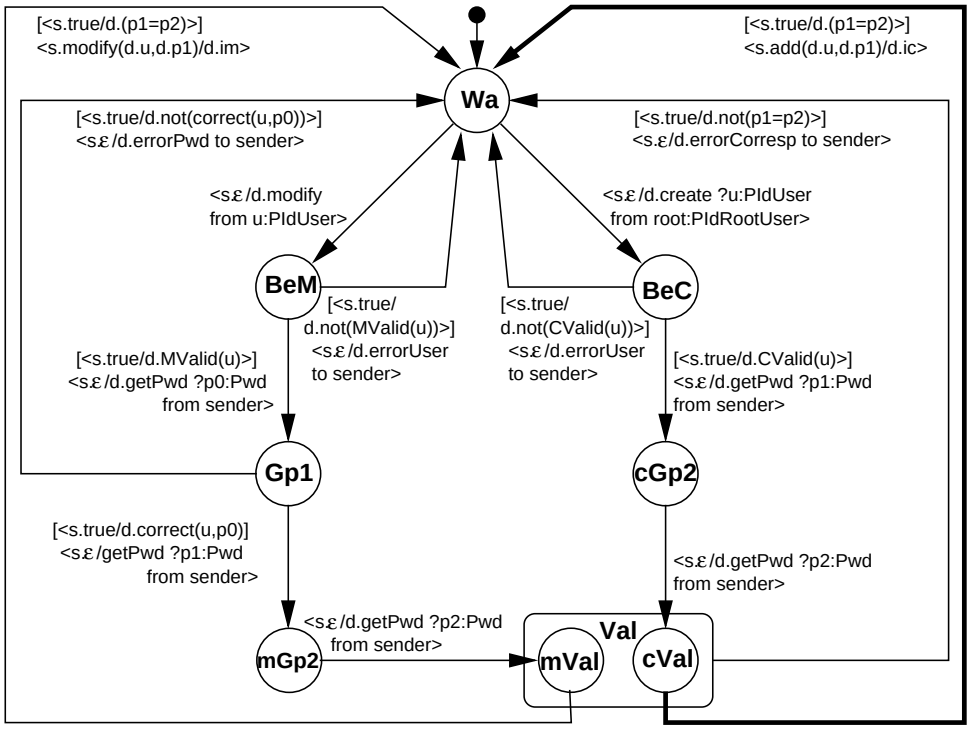


Figure 2.37 : Gestionnaire de mots de passe – STS global

définition d'un formalisme n'est pour nous qu'une étape, aussi nécessaire que la définition d'une méthode associée et d'un environnement de spécification dédié. Ces deux points, basés sur nos propositions de modèle et de formalisme, font l'objet des deux chapitres suivants.

CHAPITRE 3

Méthode de spécification pour systèmes mixtes

Though this be madness,
yet there is method in 't.

— W. Shakespeare, Hamlet.

Dans ce chapitre, nous présentons les intérêts et les grands principes des méthodes de spécification de systèmes mixtes puis nous proposons une nouvelle méthode qui améliore les méthodes existantes sur plusieurs points. Cette méthode est basée sur l'utilisation de systèmes de transitions symboliques communicants qui sont au cœur de notre modèle à base de vues présenté dans le précédent chapitre.

3.1 Motivations

Si l'importance de l'utilisation des spécifications formelles lors du développement de systèmes logiciels est largement acceptées, elles ne sont pas faciles à utiliser par un ingénieur "standard".

Nous pensons, comme [18] que la définition de méthodes associées aux formalismes de spécification est nécessaire à leur plus grande utilisation. Pour citer les auteurs, *"un formalisme ne fournit pas une méthode en soi"*. Associées à des spécifications formelles graphiques, ou disposant d'une représentation graphique, ces méthodes sont un support permettant l'utilisation des spécifications formelles par des non spécialistes. Ceci est d'autant plus nécessaire quand il s'agit de formalismes faisant intervenir la spécification de plusieurs aspects des systèmes, car les spécifications résultantes auront de fortes chances d'être plus complexes.

Comme le faisait déjà remarquer M. Broy dans [54], *"Most important properties of specifications methods are not only the underlying theoretical concepts but more pragmatics issues such as readability, tractability, support for structuring, possibilities of visual aids and machine support"*. Le support méthodologique aux spécifications doit donc se baser, entre autres, sur les points suivants : aide à l'écriture et à la réutilisation des spécifications, connexion de formalismes avec des méthodes existantes (formelles ou semi formelles comme UML), définition de moyens de génération de code (dans un formalisme cible pour faire de la vérification ou réutiliser des outils, ou dans un langage de programmation comme les langages orientés objets ou faire du prototypage), définition de moyens de validation et vérification, possibilité de preuves.

3.2 État de l'art

Pour présenter les méthodes actuelles de développement de systèmes mixtes, nous nous basons essentiellement sur les méthodes existant en LOTOS et SDL. En effet, ces systèmes sont particulièrement

représentatifs au niveau méthodologique de par la maturité des méthodes associées et leur utilisation (dans l'industrie pour SDL, dans un cadre universitaire pour LOTOS). Il existe des développements récents qui se basent sur l'emploi de formalismes mixtes pour la conception à objet, il s'agit d'un cadre plus particulier que celui qui nous intéresse et le lecteur pourra se référer à [11, 266] par exemple, ou encore à [95] qui présente une interprétation formelle des méthodes orientées objet d'analyse en LOTOS.

LOTOS et SDL sont suffisamment expressifs pour permettre un nombre important de styles de spécification [124, 125] et il est possible de spécifier un problème de plusieurs façons. Des indications sont nécessaire à leur utilisation, mais peu d'articles scientifiques en proposent pour s'intéresser essentiellement aux aspects les plus théoriques de ces formalismes.

[220] donne des contextes d'utilisation de chaque opérateur de LOTOS, ce qui peut être étendu à l'ensemble des algèbres de processus disposant de ces opérateurs généraux. Des liens entre opérateurs et concepts de programmation concurrente sont faits.

Plus généralement, [263] explique que du point de vue conception, les trois styles les plus pertinents sont le style orienté contraintes, le style orienté états (appelé style orienté automate) et le style orienté ressources. Pour les auteurs, les spécifications sont écrites, dans la majorité des cas, en utilisant un style mixte et le point important consiste à trouver un bon équilibre entre les styles possibles de façon à obtenir une spécification bien structurée.

Plusieurs indications sont données :

– utilisation du style orienté contraintes pour représenter les systèmes comme des *boîtes noires* sans considérer leur structure interne. Ce style est recommandé pour spécifier à un haut niveau d'abstraction.
– utilisation du style orienté états comme modèle d'implémentation pour dériver des implémentations efficaces (rapides).
– utilisation du style orienté ressources pour spécifier les systèmes où les sous-parties sont bien définies et où la communication entre ces sous-parties est cachée (style appelé *boîte blanche* dans [263]).

En ce qui concerne la partie algébrique, les auteurs proposent une discipline par constructeurs. Pour le reste, la méthode est articulée autour de trois phases : conception des spécifications, vérification puis implémentation et prototypage.

L'ISO a défini des orientations pour l'utilisation de LOTOS et [265] précise tout d'abord que trois principes doivent être respectés lors de la conception de systèmes distribués :

– orthogonalité : les besoins indépendants doivent être spécifiés indépendamment
– généralité : les définitions génériques et paramétriques doivent être préférées aux définitions répondant à un problème particulier
– flexibilité : les conceptions doivent pouvoir être mises à jour, c'est-à-dire étendues et modifiées.

Il semble accepté que les spécifications mixtes conduisent à respecter ces propriétés. Nous pensons par ailleurs que notre formalisme répond bien à ces trois problématiques et particulièrement l'orthogonalité (par la séparation des aspects statiques et dynamiques) et la flexibilité (par la construction de points de vue abstraits basés sur des conditions et permettant héritage et réutilisation).

D'un point de vue méthodologique, [265] préconise de penser d'abord le système de façon abstraite et en fonction des besoins (description *extensionnelle*). Deux styles de spécification correspondent à cette attente :

– monolithique : seules les différentes suites d'actions possibles sont étudiées. Ce style est très in-

dépendant de toute implémentation mais il manque de structure et est peu compréhensible.
– orienté contraintes : les actions observables (et elles seules) sont présentées selon un ordonnancement temporel défini comme la conjonction de contraintes. Ce style répond bien aux trois principes cités plus haut.

Dans un second temps, le spécifieur pourra s'attacher à la façon dont le système répond aux besoins, c'est-à-dire sa structure (interne) en termes de sous-parties qui interagissent ou en termes d'états abstraits. Les deux styles suivants permettent cette description *intensionnelle* :
– orienté états : le système est vu comme une ressource à état interne explicite variant avec les interactions. Ce style est utile en phase finale de conception, quand la ressource peut être vue comme un objet séquentiel.
– orienté ressources : les interactions internes et observables sont présentées. Le comportement est vu comme une communication cachée entre plusieurs sous-systèmes composés. Contrairement à l'approche orientée contraintes, la décomposition se fait ici en fonction de la ressource (type) et pas des contraintes entre interactions. Les sous-systèmes sont spécifiés dans n'importe lequel des styles existants.

On peut donc bien se rendre compte que, parmi les méthodes que l'on retrouve mentionnées, il existe trois approches principales : l'approche orientées contrainte, l'approche orientée ressources, et l'approche orientée états. Il est important de remarquer que l'ensemble de ces méthodes mettent l'accent essentiellement sur la partie dynamique des systèmes.

3.2.1 Parties dynamiques

3.2.1.1 Approche orientée contraintes

L'*approche orientée contrainte* [47] est une méthode de type diviser pour régner. Un certain nombre de processus sont composés en parallèle, le résultat de la composition respectant un ensemble de contraintes équivalent à l'ensemble des contraintes des sous-processus, ensemble auquel il faut rajouter les contraintes de synchronisation entre ces sous-processus. Cette approche permet [185] de structurer la spécification en termes de décomposition d'un problème en plusieurs sous-processus et de spécifier (ou réutiliser) séparément les sous-processus.

Les approches orientées contraintes sont utilisées essentiellement pour la spécification de protocoles et services de communication, privilégiant souvent l'aspect dynamique par rapport à l'aspect statique (souvent les processus n'ont d'ailleurs pas de partie donnée correspondante). Le point fort de cette approche est que la décomposition d'un processus en sous-processus est assez aisée.

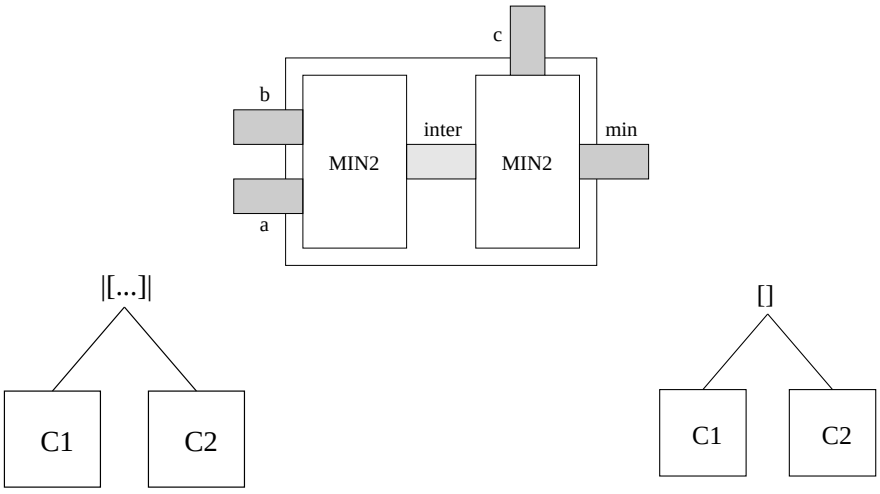
La figure 3.1 donne un exemple spécifié en LOTOS en utilisant cette approche. Il s'agit de définir un processus calculant le minimum de trois entiers.

L'approche suivie se base sur le fait qu'il est possible, pour calculer le minimum de trois nombres, de combiner deux processus calculant le minimum de deux nombres. Soient a, b et c les trois nombres, et min le minimum désiré. Soient deux processus $MIN2_1$ et $MIN2_2$ calculant dans une sortie out la minumum de leurs entrées $in1$ et $in2$. On a donc:

$$out_{MIN2_1} = min(in1_{MIN2_1}, in2_{MIN2_1})$$
$$out_{MIN2_2} = min(in1_{MIN2_2}, in2_{MIN2_2})$$

Si l'on compose ces deux processus de façon à ce que a corresponde à $in1_{MIN2_1}$, b à $in2_{MIN2_1}$, c à $in1_{MIN2_2}$ et que out_{MIN2_1} corresponde à $in2_{MIN2_2}$, alors on obtient:

$$out_{MIN2_1} = min(in1_{MIN2_1}, in2_{MIN2_1})[a/in1_{MIN2_1}, b/in2_{MIN2_1}, inter/out_{MIN2_1}]$$
$$out_{MIN2_2} = min(in1_{MIN2_2}, in2_{MIN2_2})[c/in1_{MIN2_2}, inter/in2_{MIN2_2}, min/out_{MIN2_2}]$$




```
process MIN3[a,b,c,min] : noexit :=
   hide inter in
      ( MIN2[a,b,inter]
   |[inter]|
      MIN2[c,inter,min] )
endproc
```

```
process MIN2 [in1,in2,out]: noexit:=
      in1?x1:NAT; in2?x2:NAT;
        out!min(x1,x2); stop
   []
      in2?x2:NAT; in1?x1:NAT;
        out!min(x1,x2); stop
endproc
```

Figure 3.1 : Approche orientée contraintes

Soit:

$$inter = min(a, b)$$
$$min = min(c, inter)$$

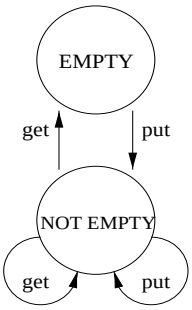
Et donc bien $min = min(c, min(a, c))$ ce qui est désiré. Il faut noter que le processus $MIN2$ peut à son tour être spécifié en utilisant une approche orientée contraintes mais qui procède par disjonction (différents cas possibles) et non pas par conjonction (union des contraintes).

3.2.1.2 Approche orientée états

Il s'agit [182] d'une approche hybride consistant à définir un objet sous forme d'un processus gérant un type de donnée. Pour chaque état, différents ensembles de communications sont autorisés en fonction de l'état interne des données du processus. Les transitions peuvent représenter des communications entre processus ou des actions internes (cachées), mais elles ont généralement un effet sur les types de données.

La figure 3.2 donne un exemple en LOTOS de cette approche sur un buffer non borné. Ici, nous avons utilisé un processus par état. L'autre approche est d'utiliser des gardes au niveau des transitions (lorsque le language formel le permet) correspondant aux états source de ces transitions.

[74] donne une méthode mixte de conception de systèmes embarqués très proche de l'approche orientée états. La spécification est basée sur la notion d'objets concurrents communicants. Tout objet est un processus ayant un paramètre indiquant son état interne (caché). Cet objet communique avec d'autres objets vus comme son environnement. La méthode s'appuie aussi sur la composition parallèle de processus dont le comportement correspond à un ensemble de contraintes et sur l'utilisation de *cycles* pour




```
process BUFFER[put,get](b:Buffer):noexit:=
    EMPTY[put,get](b)
endproc
process EMPTY[put,get](b:Buffer):noexit:=
    put?x:T;NON-EMPTY[put,get](push(b,x))
endproc
process NON-EMPTY[put,get](b:Buffer):noexit:=
    [nb(b)=1]→get!top(b);
        EMPTY[put,get](pop(b))
[] [nb(b)>1]→get!top(b);
        NON-EMPTY[put,get](pop(b))
[] put?x:T;NON-EMPTY[put,get](push(b,x))
endproc
```

Figure 3.2 : Approche orientée états

la définition du comportement des sous-processus les plus simples. Un cycle correspond à un comportement qui peut rendre un service particulier (une ou plusieurs communications séquentielles) puis qui se rappelle récursivement.

L'avantage majeur de l'approche orientée états est de permettre une bonne compréhension des mécanismes internes d'un processus (par l'emploi possible d'un formalisme très lisible, celui des systèmes états/transitions). [263] précise que cette approche permet de produire des implémentations plus efficaces des processus dans des langages séquentiels. L'inconvénient de cette approche est qu'elle peut conduire à des spécifications non structurées [126].

3.2.1.3 Approche orientée ressources

Il s'agit d'une approche intermédiaire entre approche orientée contraintes et approche orientée états. Cette approche permet de décomposer un système en sous-systèmes en se basant sur un ensemble de ressources disponibles et aux fonctionnalités associées (en termes de communications). C'est une approche qui remplace l'approche orientée contraintes dans les premières étapes de décomposition d'un problème lorsque les données des processus sont significatives (c'est-à-dire qu'il est possible de les considérer comme des ressources et pas uniquement comme des données que nous pourrions qualifier d"'amorphes" et qui sont juste échangées entre processus). Un exemple de cette approche est donné dans la décomposition de l'étude de cas du nœud de transit d'un réseau que nous utilisons dans la suite pour présenter notre méthode.

3.2.2 Parties statiques

Les auteurs de [126], en se basant sur leur expérience, remarquent qu'il est plus sage de traiter la spécification de la partie données après celle de la partie comportement. Nous remarquons qu'en effet cela permet d'avoir un ensemble minimal et complet (en termes de réponse aux besoins exprimés dans la description informelle) d'opérations dans la partie données. De plus cette approche permet de faire correspondre une partie statique à une partie dynamique et favorise la séparation des aspects. L'approche consistant à se baser en premier lieu sur des données peut conduire à une perte d'abstraction de la partie statique.

Pour [126] il s'agit tout d'abord de recenser les sortes et opérations apparues au niveau de la dynamique des processus. Puis les équations correspondant aux opérations non constructrices sont créées (les auteurs parlent de "discipline par constructeurs" ou axiomatisation constructive au niveau de la construction des axiomes). Cette approche constructive est aussi vantée par [263].

3.3 Une proposition

Nous donnons ici notre méthode pour la spécification de systèmes mixtes. Elle est inspirée de [126].

Les éléments apportés par notre méthode sont :
- la description informelle est orientée en fonction des services du processus.
- la description des communications est fortement typée et se fait lors de la description des fonctionnalités externes des processus (et non pas lors du travail sur les composants séquentiels et leur activité interne). Il nous apparaît en effet que le type des données émises et reçues via une porte font partie de la description externe d'un processus et est utile lors de la composition parallèle.
- la composition des processus utilise éventuellement des schémas.
- l'automate du comportement est obtenu de façon semi-automatique.
- il existe plusieurs possibilités de génération de spécifications (KORRIGAN, LOTOS, SDL).
- l'utilisation à la fois de notations graphiques basées sur UML et de notations textuelles formelles.
- une automatisation partielle et des guides facilitent la production de la partie statique.

Nous utilisons pour la présentation de notre méthode le concept d'agendas, introduits dans [144, 147, 133]. Un agenda est une liste d'activités à effectuer pour réaliser une tâche dans le cadre d'un développement logiciel. Il est possible d'associer à chacune des étapes des agendas des patrons de spécification. Notre méthode est générale aux systèmes mixtes et nous choisissons des patrons correspondant à notre modèle de vues. Alors que de nombreuses méthodes sont expliquées de façon purement informelle, les agendas sont un moyen de décrire une méthode de façon semi-formelle, point par point et à différents niveaux de détail. Appliqués dans le cadre des spécifications formelles, les agendas guident le spécifieur dans le développement de la spécification, et facilitent ainsi l'emploi des méthodes formelles. Les agendas prévoient également les critères de validation qui permettent de vérifier la cohérence et la complétude de la spécification [144].

Nous présentons notre méthode par l'intermédiaire d'une étude de cas concernant un nœud de transit servant au routage de données dans un réseau de communication. Il s'agit d'une étude de cas dont la description informelle a été donnée dans le cadre du projet SPECS et a ensuite été modifiée pendant le projet VTT [43, 209]. La description informelle de l'étude de cas, ainsi que les détails de l'application de la méthode à cette étude de cas, pourront être trouvés en annexe D.

Vision globale

Pour l'établissement de la spécification, notre méthode se compose, à un premier niveau, de quatre étapes (cf. figure 3.3) :

1. **description informelle** du système à spécifier ;
2. description de l'**activité concurrente** ;
3. description des **composants séquentiels** sous forme d'automate ;

4. spécification des **types de données** correspondants.

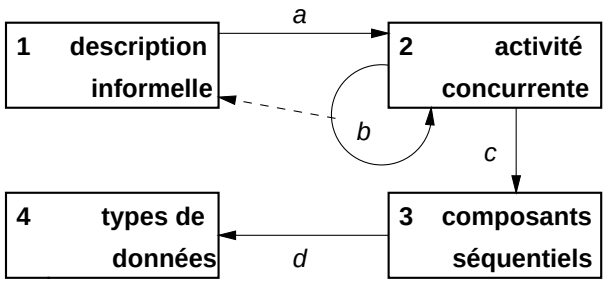


Figure 3.3 : Dépendances entre étapes au niveau global.

Il est possible d'associer des étapes de validation à la méthode. Elles sont discutées dans le chapitre concernant l'environnement de spécification mixte (chapitre 4).

Nous avons légèrement modifié le formalisme des agendas de façon à indiquer au niveau des dépendances les éléments utilisés entre étapes (cf. figure 3.3). Ces éléments sont :

(a) données, fonctionnalités et contraintes décrites dans la description informelle ;
(b) types de données, contraintes et communications de processus à sous-processus ;
(c) conditions de communication et de comportement, préconditions et postconditions ;
(d) automate et typage des communications.

Chacune des étapes sera décrite dans un agenda différent.

3.3.1 Etape 1 : Description informelle

étape	expression / schéma	conditions de validation
1.1 : fonctionnalités du système	F_i : texte	○ pas de redondance
1.2 : contraintes du système	C_i : texte	○ cohérence
1.3 : données du système	D_i : sorte	

Figure 3.4 : Agenda de la description informelle.

Cette première étape permet de dégager les caractéristiques du système. Elle peut de fait être appliquée avant chaque itération de l'étape concernant l'activité concurrente.

3.3.1.1 Etape 1.1 : description des fonctionnalités externes du système

Il s'agit ici de recenser les opérations possibles. On leur donne un nom (F_i dans la figure 3.4) et on les décrit. Il ne doit pas y avoir de redondance : si deux opérations de nom différent font la même chose c'est que l'on a oublié quelque chose (dans la description et/ou éventuellement une contrainte associée).

Dans notre exemple, les opérations sont données par les clauses 6 et 7 :

– au niveau du port de contrôle en entrée : réception d'un message de commande (`inCmde`)
– au niveau des ports de données en entrée : réception d'un message de données (`inData`)

Les clauses 7 et 9 induisent de plus à envisager les opération suivantes :

– au niveau du port de contrôle en sortie : émission d'un message erroné (`outCmde`)

– au niveau des ports de données en sortie : émission d'un message de données (`ouData`)

Enfin, la clause 12 précise qu'une source temporelle est disponible dans l'environnement. Nous supposerons que le nœud de transit peut récupérer cette valeur du temps.

D'autres opérations apparaissent dans les clauses, mais elles sont internes au nœud de transit. Elles seront envisagées lors de la décomposition du système.

Notons que la clause 9 ne précise pas ce que le port de contrôle en sortie fait des messages d'erreurs reçus. Par analogie avec les ports de données en sortie, nous supposerons qu'ils sont émis un par un.

3.3.1.2 Etape 1.2 : description des contraintes du système

On exprime ici les contraintes sur le système : ordres, ordonnancement des opérations, limites de taille, ... Les contraintes doivent être cohérentes, c'est-à-dire ne pas être contradictoires.

Les contraintes de notre exemple seront reprises et détaillées au fur et à mesure de leur utilisation.

3.3.1.3 Etape 1.3 : description des données du système

Les données sur lesquelles va travailler le système sont ici nommées, c'est donc un point de vue très abstrait pour l'instant (un nom et une sorte). À ce niveau, on ne présume pas du choix final fait pour ces données (vue statique, type de donnée amorphe ou identifiant dans une composition).

Le nœud de transit dispose d'une liste de numéros de ports actifs (clause 6a), d'une liste de routes (clauses 6b et 7a) et d'une collection de messages erronés (clauses 4, 6c, 7a et 9).

3.3.2 Etape 2 : Activité concurrente

Il s'agit de trouver les composants exécutés en parallèle. Chacun est modélisé par un processus. La spécification peut être vue comme le processus de plus haut niveau. Les processus peuvent terminer ou non et cette information (appelée par exemple en LOTOS fonctionnalité du processus) est précisée.

La décomposition de processus en sous-processus concurrents s'inspire des styles de spécification orienté contraintes et orienté ressources [47, 263, 265]. D'autre part, dans le cas de spécifications permettant la réutilisation (comme le modèle des vues), il est possible de combiner une analyse descendante (décomposition) des composants avec une analyse ascendante (combinaison de composants déjà spécifiées).

Comme nous l'avons indiqué dans le chapitre 1 (Section 1.3.1), il existe différents type de compositions et nous nous restreignons à la composition forte des systèmes critiques et des applications réparties. Ceci peut être comparé à la composition[1] d'UML (diamant noir). C'est pourquoi nous utiliserons ce symbole pour la représenter. Toutefois, notre approche est plus orientée "composants" qu'UML puisque nous nous intéressons explicitement au typage des communications dans les interfaces des composants. D'autre part, la concurrence et les communications entre composants sont décrites dans nos diagrammes de communication ce qui permet la définition de patrons de communication. En UML ces informations sont réparties dans plusieurs diagrammes (collaboration, séquence) mais aussi enfouies dans les Statecharts.

Nous décomposons cette étape de la manière suivante :

[1] La composition forte est appelée simplement "composition" en UML, la composition "faible" étant l'agrégation.

– 2.1 interfaces de communication,
– 2.2 décomposition et répartition, et
– 2.3 (re)composition parallèle.

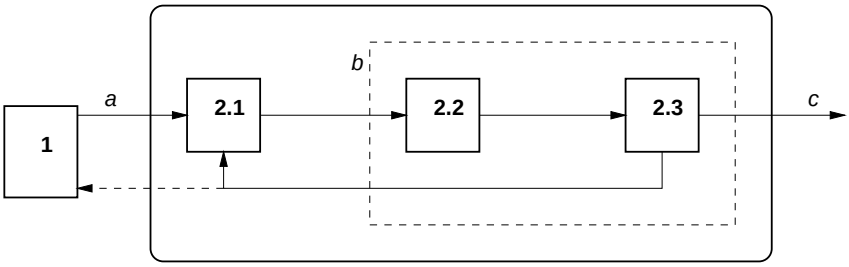


Figure 3.5 : Dépendances entre étapes : activité concurrente.

Comme le montre la figure 3.5, les sous-étapes sont itérées. Nous associons à chaque étape différents types de diagrammes inspirés par UML. Il est aussi possible de leur associer des spécifications formelles dans un langage cible (KORRIGAN, LOTOS, SDL). Cette dualité entre notation graphique et langage formel est un point qui permet d'améliorer l'utilisation des méthodes formelles et fait actuellement l'objet d'un intérêt croissant.

Le tableau 3.1 donne par exemple la correspondance, au niveau de l'étape d'activité concurrente, entre sous-étapes, diagrammes et spécification KORRIGAN. Il est possible de réaliser des tableaux équivalents pour les différents langages de spécification mixte. Nous présenterons nos notations au fur et à mesure de leur utilisation sur l'étude de cas.

sous-étape	**diagramme**	**KORRIGAN**
interfaces de communication	interface (ex. figure 3.8)	Σ dans les Internal Structuring Views partie `SPECIFICATION`
decomposition	composition (ex. figure 3.10)	External Structuring View (partielle: clauses `imports` et `is`)
recomposition	communication (ex. figure 3.18)	External Structuring View (complète)

Table 3.1 : Sous-étapes, diagrammes et KORRIGAN

3.3.2.1 Etape 2.1 : interfaces de communication

étape 2.1.1 : ports de communication et données. Les interactions entre composants sont modélisées par des ports de communication formels qui représentent les services (interface) du composant. Les ports de communication externes sont fournis par la description informelle (fonctionnalités, sous-étape 1.1). Il en va de même pour les données (D_i, sous-étape 1.3).

étape 2.1.2 : typage des communications. On détermine les communications entre les composants. Les données émises ou reçues sur les ports sont typées. L'émission d'une donnée de type T est notée `!x:T`. La réception d'une donnée de type T est notée `?x:T`. À ce niveau, le typage des processus communicants avec le composant sur ces ports peut aussi être indiqué. Ce typage des communications

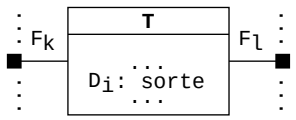
étape	expression / schéma	conditions de validation
2.1.1 : ports de communication et données		○ pas d'oubli : prise en compte des F_i et D_i de 1.1 et 1.3
2.1.2 : typage des communications	F_i : ?x_j:s_j !x_k:s_k	○ pas d'oubli : prise en compte des F_i de 1.1 ○ sortes émises "disponibles"

Figure 3.6 : Agenda des communications.

sert non seulement pour définir l'interface du processus mais aussi ultérieurement pour travailler sur la partie statique de la spécification.

Nous utilisons pour l'étape des interfaces de communication une notation particulière à base de *symboles d'interface* (figure 3.7). Ces symboles sont complétés (ex. figure 3.8) par les informations concernant le typage des communications. Nous avons ici une approche "systèmes ouverts", donc beaucoup plus proche de LOTOS que d'UML par exemple.

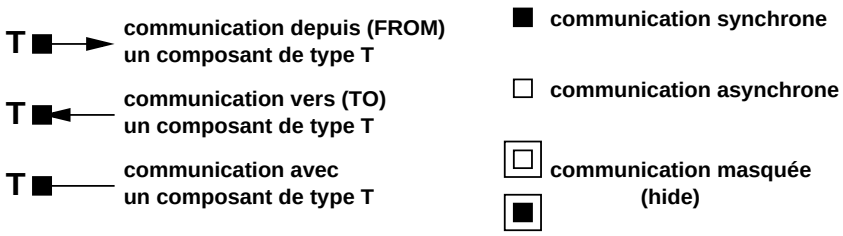


Figure 3.7 : Notation pour les interfaces de communication

Tous les types de données définis dans cette phase devront avoir une spécification algébrique associée. Il est possible d'avoir des types génériques mais ils devront alors être instanciés (étape 3.3.4). Un critère de validation consiste à s'assurer que les sortes émises par un processus sont "disponibles". Une sorte `s` est *disponible* pour un processus si :

- soit elle est *directement disponible*, c'est-à-dire prédéfinie et importée, ou bien définie par le processus,
- soit il existe une opération importée `f:` $d^* \rightarrow s$ telle que toutes les sortes de d^* sont disponibles,
- soit la sorte est reçue par le processus.

La spécification des données ne se faisant qu'à l'étape 3.3.4, cette validation est incomplète à ce niveau.

Nous pouvons maintenant donner la description du nœud de transit au niveau le plus abstrait (figure 3.8). Nous utilisons un type `Msg` pour représenter les différents messages. Le nœud de transit a quatre fonctionalités :

- réception d'un message de commande : `inCmde :?m:Msg`
- réception d'un message de données : `inData :?m:Msg`
- sortie d'un message erroné : `outCmde :!m:Msg`
- sortie d'un message de données : `outData :!m:Msg`

Ses données sont composées de trois listes et il est paramétré par `N`, le nombre maximum de ports de données (en entrée et en sortie) qu'il peut contenir.

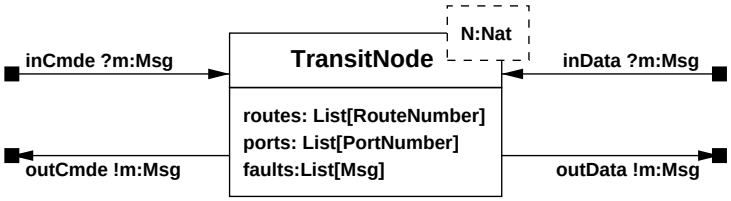


Figure 3.8 : `TransitNode` - Interface de communication

3.3.2.2 Etape 2.2 : décomposition et répartition

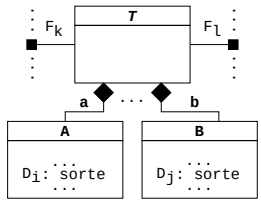
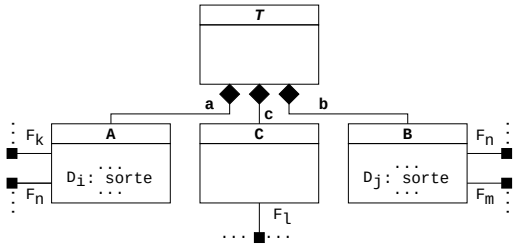
étape	expression / schéma	conditions de validation
2.2.1 : répartition des données		○ toutes les données réparties au niveau des sous-processus (cf. 2.1.1)
2.2.2 : répartition des fonctionnalités		○ fonctionnalités et données associées ○ toutes les fonctionnalités réparties au niveau des sous-processus (cf. 2.1.1)

Figure 3.9 : Agenda de la décomposition et de la répartition.

La décomposition d'un processus en sous-processus se fait en général à partir des informations (contraintes) de la description informelle. Lorsque cette dernière ne fournit pas d'information, il est possible d'effectuer le découpage à partir des données et/ou des fonctionnalités. Enfin, la décomposition peut être faite en cherchant à réutiliser des composants déjà spécifiés. Dans tous les cas, après la décomposition il est nécessaire de répartir les données et les fonctionnalités entre les sous-processus.

Décomposer un processus à partir de ses données permet de suivre un style de spécification orienté ressources. Les fonctionnalités sont ensuite regroupées avec les données qui leurs sont liées.

Décomposer un processus à partir de ses fonctionnalités se rapproche plus d'un style de spécification orienté contraintes. Dans ce cas, on considère des familles de fonctionnalités liées. Les données sont ensuite associées aux fonctionnalités qui les utilisent.

La clause 1 nous conduit naturellement à voir que 4 catégories de composants se trouvent dans le nœud de transit : le port de contrôle en entrée (`InputControlPort`), les ports de données en entrée

(`InputDataPort`), le port de contrôle en sortie (`OutputControlPort`) et les ports de données en sortie (`OutputDataPort`). Un composant supplémentaire (`FaultyCollection`), gèrera la collection des messages d'erreurs (clause 9). Plusieurs étapes de décomposition conduisent à l'obtention de ces composants.

Décomposition du nœud de transit.

Le système peut dans un premier temps être décomposé (figure 3.10) en partie contrôle (`Control-Ports`) et partie données (`DataPorts`). Les routes, ainsi que les numéros de ports et les messages d'erreurs sont associés aux ports de contrôle.

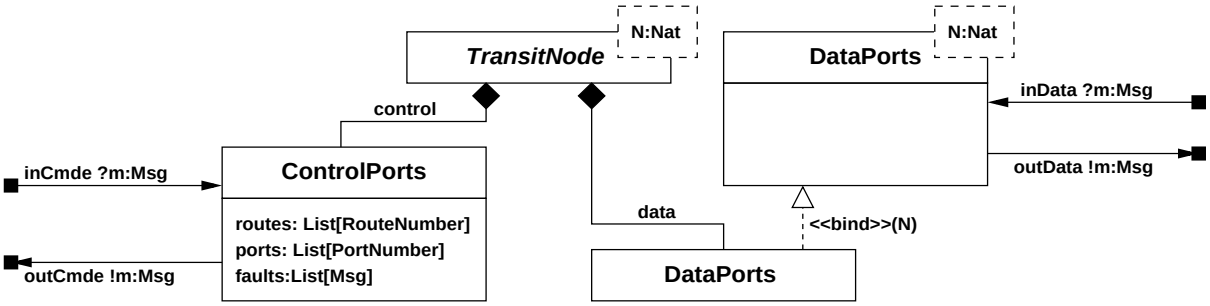


Figure 3.10 : `TransitNode` – Diagramme de composition

Il s'agit bien ici de la composition forte d'UML. Nos décompositions, puis leur composition comme dans les étapes suivantes, correspondent en fait à des diagrammes intégrant les diagrammes de classe, de séquence et de collaboration d'UML. Toutefois, nous plaçons l'ensemble des informations dans un unique diagramme ce qui les lie fortement (contrairement à UML où la cohérence n'est pas du tout assurée).

A ce stade, il est possible d'obtenir une spécification partielle, par exemple en KORRIGAN (figure 3.11), pour le nœud de transit en utilisant sa représentation graphique. Une telle spécification est partielle puisque les classes des sous-composants utilisés (clause `imports`) n'ont pas encore été définies. Les communications entre sous-composants (ce qui les "colle") seront spécifiées dans une étape suivante (d'où l'utilisation du mot-clé `undefined` dans la partie `COMPOSITION`). Ceci n'induit cependant pas forcément que la spécification sera partielle puisqu'il est possible que deux sous-composants ne communiquent pas du tout.

COMPOSITION VIEW TransitNode

SPECIFICATION

imports ControlPorts, DataPort, Nat

variables N : Nat

COMPOSITION `undefined`

is control : ControlPorts, data : DataPorts<N>

Figure 3.11 : `TransitNode` in KORRIGAN

Étude des contraintes induites par la décomposition du nœud de transit (nouvelles données).

De nouvelles données apparaissent suite à la décomposition. Les ports de données en sortie sont sérialisés (clause 7b) et disposent donc d'un buffer de messages. Il en va de même pour les ports de contrôle en sortie. Les ports de données dans leur ensemble ont besoin d'un identifiant de façon à les reconnaître lors des phases d'activation (par le port de contrôle en entrée, voir la remarque concernant la création de nouveaux ports par l'emploi d'une communication interne de nom `enable` page 100) ou de routage (d'un port de données en entrée vers un port de données en sortie). Ceci est pris en compte par l'utilisation des identifiants dans les diagrammes de composition.

Décomposition de la partie contrôle.

La partie contrôle peut être décomposée en un port d'entrée (`InputControlPort`), un port de sortie (`OutputControlPort`) et une `FaultyCollection` (figure 3.12). Les routes et les numéros de ports seront placés dans l'`InputControlPort` puisqu'il modifie ces listes (clause 6). La liste des messages d'erreurs est associée à la `FaultyCollection`. Elle contient les messages d'erreurs jusqu'à réception d'un message `sendfault`. L'`OutputControlPort` émet les messages d'erreurs vers l'extérieur du nœud de transit un par un. Il dispose pour cela d'un buffer de sortie.

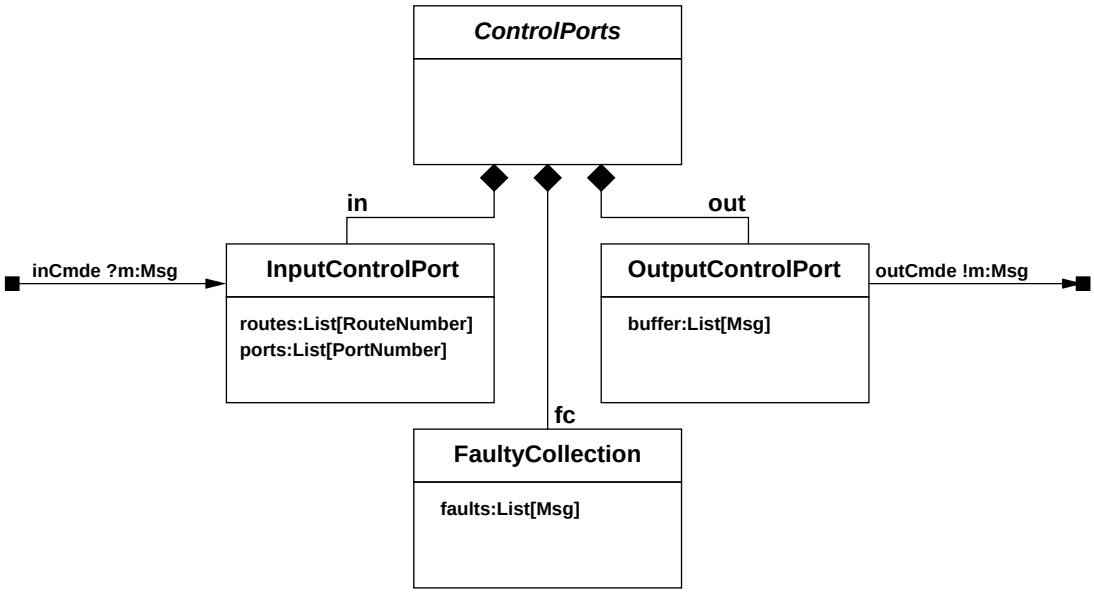


Figure 3.12 : `ControlPorts` – Diagramme de composition

Décomposition de la partie données.

De son côté, la partie données est décomposée (figure 3.13) en `N` ports d'entrée (`InputDataPort`) et `N` ports de sortie (`OutputDataPort`). Les ports de données disposent d'un identifiant de façon à pouvoir les activer. Cette identification utilise l'opérateur range de KORRIGAN et étend la notation empruntée à UML. Le port de sortie dispose aussi d'un buffer pour les sorties de messages.

Au regard des informations recueillies lors des étapes précédentes, il est possible de représenter le système à l'aide d'un diagramme de composition représentant l'ensemble des classe obtenues. Conjointe-

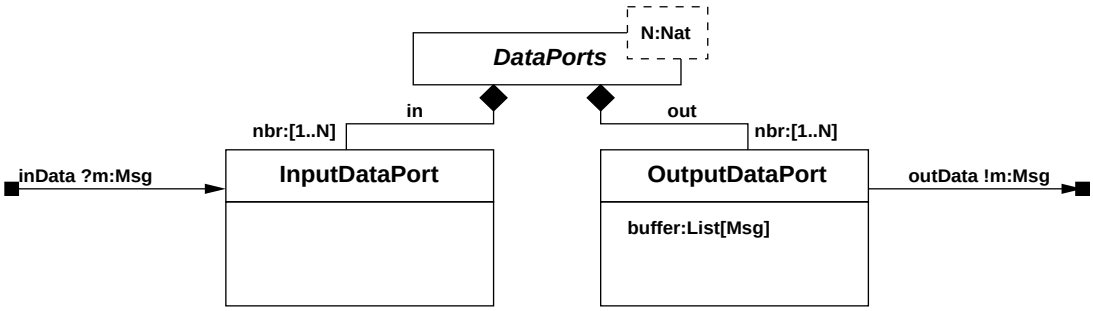


Figure 3.13 : `DataPorts` – Diagramme de composition

ment à la notation inspirée d'UML, nous autorisons l'emploi d'une notation inspirée par les diagrammes d'architecture de SDL (figure 3.14) et donnant une représentation plus "à plat" du système. Ceux-ci conviennent particulièrement lorsque chaque décomposition est suivie de la composition parallèle des sous-composants. En effet, cette notation prend directement en compte la communication entre sous-composants[2]. Ceci peut conduire à une représentation faiblement structurée, ce qui n'est pas le cas de notre notation inspirée par UML. D'autre part, la représentation inspirée de SDL est beaucoup moins expressive en ce qui concerne l'identification des composants.

3.3.2.3 Etape 2.3 : composition parallèle

Les compositions de processus (figure 3.15) entrent souvent dans des cadres déjà connus. Nous proposons d'utiliser une bibliothèque de "schémas de composition" (ou patrons). Si la composition ne se trouve pas dans cette bibliothèque, il est bien sûr possible de l'étendre. Quelques schémas (appelés schémas de synchronisation ou styles d'architectures) peuvent être trouvés dans [182, 125, 173, 145]. L'instanciation de schémas doit être simulée (voir [262] par exemple) lorsque le langage de spécification ne permet pas de la faire directement.

Si nécessaire, certaines particularités de la composition parallèle du langage cible peuvent ici être mises en œuvre. Ainsi, il est possible de masquer les communications non pertinentes à un certain niveau d'abstraction grâce à l'opérateur $hide$. Dans le cadre des vues, c'est ici que les expressions de colle sont données.

Il faut noter que la composition parallèle de processus est un moyen d'exprimer certaines contraintes entre processus. Ainsi, la clause 2b crée une contrainte sur la composition parallèle des différents ports. Pour la respecter, les différents ports de données en entrée (respectivement en sortie) seront plutôt composés par entrelacement que par synchronisation[3].

Il est possible de vérifier si une composition de sous-processus respecte les contraintes désirées pour le processus englobant. Pour cela, il suffit de faire l'hypothèse sur les contraintes respectées par chaque sous-processus ou bien de réutiliser un composant pour lequel certaines propriétés ont déjà été prouvées. Il suffira alors que les contraintes du processus soient incluses dans la composition de celles des sous-processus.

La spécification de la composition parallèle des sous-processus peut être dérivée de l'ensemble des

[2] Les commentaires concernant les communications internes au nœud de transit sont donnés dans la section concernant la composition parallèle.

[3] La clause 2b précise que ces processus sont "concurrents" ce qui signifie ici qu'il ne communiquent pas entre eux et ne se synchronisent pas non plus.

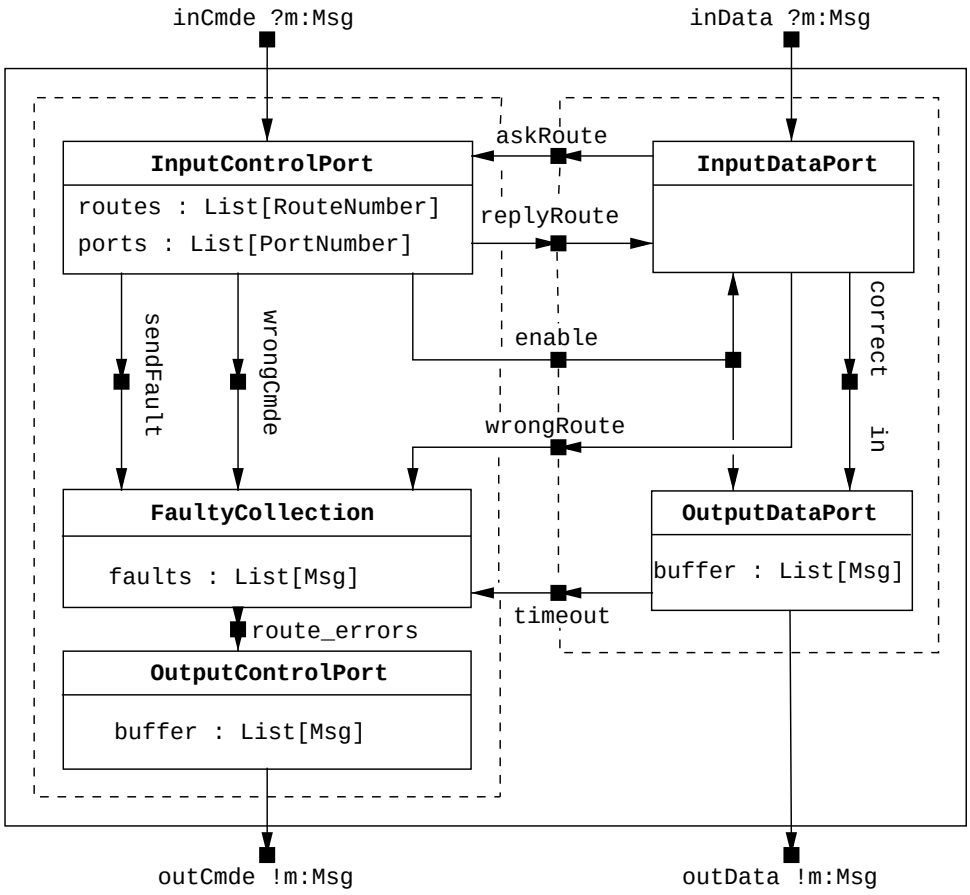


Figure 3.14 : Vue interne du nœud de transit.

schémas de composition. Les différents composants sont reliés entre eux par la colle des vues, les opérateurs de composition parallèle de LOTOS ou encore la structure de blocs et les canaux de SDL. Ceci se fait en respectant les contraintes de synchronisation (clause 2b par exemple).

Étude des contraintes induites par la décomposition du nœud de transit (communications internes).

Messages erronés. Ils sont sauvés dans la collection de la `FaultyCollection` (clause 9). Il s'agit (clause 10) des messages de commande incorrects (`wrongCmde`, origine `InputControlPort`), des messages de données avec une route incorrecte (`wrongRoute`, origine `InputDataPort`) et des messages obsolètes (`timeout`, origine `OuputDataPort`).

Emission des messages d'erreur. La commande `sendFault`, reçue par l'`InputControlPort` déclenche le routage des messages erronés (en proportion non spécifiée) de la `FaultyCollection` vers l'`OutputControlPort` (clause 9). Cette phase utilise une communication de nom `sendFault` entre `InputControlPort` et `FaultyCollection` et une communication de nom `routeErrors` entre `FaultyCollection` et `OutputControlPort`. Nous supposerons que les messages reçus par l'`OutputControlPort` sont stockés et émis un par un (la spécification ne parle pas de ce point, mis à part la clause 8).

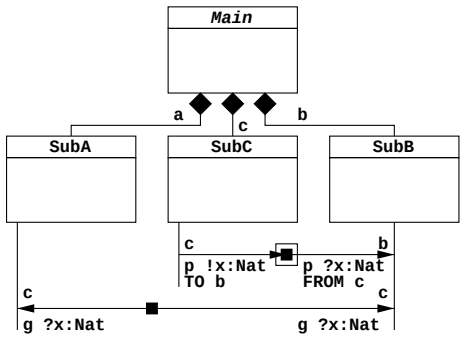
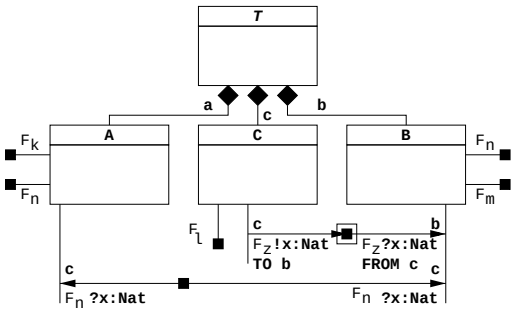
étape	expression / schéma	conditions de validation
2.3.1 : choix d'un schéma de composition		
2.3.2 : application du schéma (cf. étapes 3.3.2.2 et 2.3.1)		○ correspondance entre les contraintes (souhaitées) du processus et celles obtenues par la composition des sous-processus

Figure 3.15 : Agenda de la composition parallèle.

Informations sur les routes. L'`InputDataPort` a besoin d'informations sur les routes (existence ou non, liste des ports de sortie associés). Ces informations font partie des données de l'`InputControlPort`. L'échange d'information se fera par communication de type question/réponse entre ces deux composants (`askRoute` et `replyRoute`).

Routage d'un message. Lorsque le message indique une route correcte, le message est routé (communication `correct`) par l'`InputDataPort` sur un des `OutputDataPorts` correspondant à la route (clause 7a).

Nouveaux ports. Lorsque l'`InputControlPort` reçoit le message `addDataPortIn&Out` il crée les ports correspondants (clause 6). Le formalisme des vues ne permet pas la création dynamique de composants. De plus, ce que l'`InputControlPort` fait lorsque le port existe déjà n'est pas indiqué. Pour ne pas sur-spécifier, nous avons opté pour une solution qui est de créer au départ tous les processus des `InputDataPorts` et `OutputDataPorts` (`N` de chaque, clause 1) ; pour respecter la clause 5, ces processus ne sont activés que lorsque l'`InputControlPort` reçoit le message demandant de les créer (communication `enable`). Ce qu'il faut faire lorsqu'un port de données est activé plusieurs fois n'est pas indiqué et donc pas spécifié.

Composition parallèle des sous-composants du nœud de transit.

La notation que nous utilisons (diagrammes de communication) consiste à compléter les diagrammes de composition avec une représentation graphique de la "colle" de communication entre sous-composants. Il s'agit en fait d'une représentation graphique de la partie `COMPOSITION` des External Structuring Views de KORRIGAN (chapitre 2). En effet, cette partie des vues est suffisamment expressive pour permettre d'exprimer les différents types de communication de langages de spécification mixte comme LOTOS, SDL, ou CCS. Nous discutons donc ici des spécifications KORRIGAN. Des patrons de traduction peuvent ensuite être utilisés pour engendrer d'autres spécifications (LOTOS et SDL par exemple, voir le chapitre 4).

La partie axiomatique de la colle (clause `axioms`), les formules temporelles d'états (Φ et Φ_0) et le mode de concurrence (δ) sont placées dans le composant agrégeant (ex. `DataPorts` pour `InputDataPort` et `OutputDataPort`) comme indiqué dans la figure 3.16.

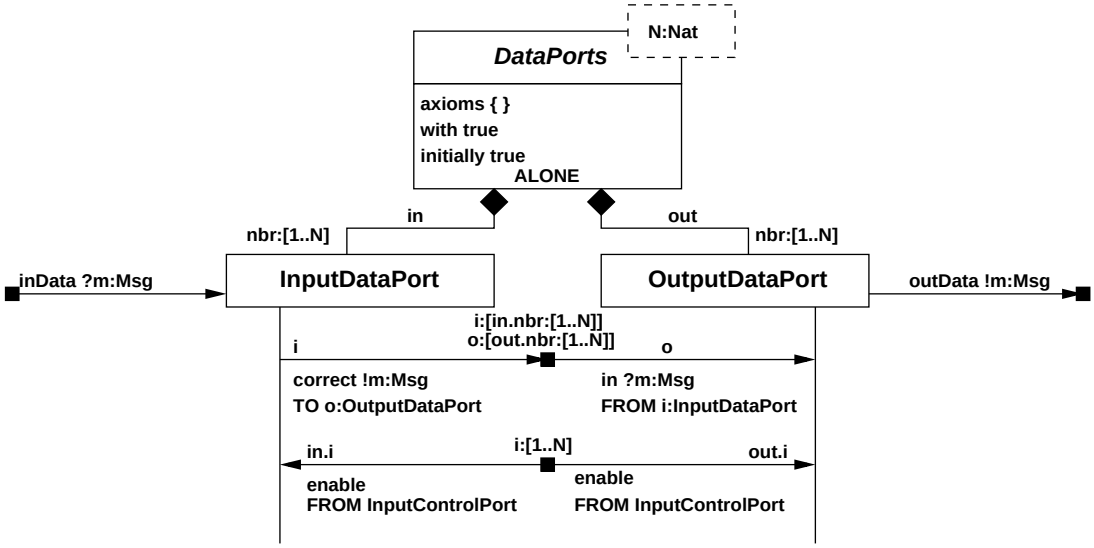


Figure 3.16 : `DataPorts` – Diagramme de communication (partiel)

Chaque couple de Ψ (communications) est traité en liant les composants impliqués par un nœud (carrés dans la figure 3.7). Les parties du couple relatives à chacun des composants sont placées au niveau des liens : identification des composants au-dessus, formule de transition au-dessous. Lorsque certaines communications sont cachées (hide), le nœud correspondant est entouré par un carré (agenda 3.15). Ici encore l'opérateur range de KORRIGAN peut être utilisé en tant que raccourci syntaxique (un seul lien en lieu et place des $\mathtt{N}\times\mathtt{N}$ qui auraient été utilisés sans le range) :

$$\forall nbr_i \in [1..N],\ \forall nbr_o \in [1..N],\ \forall m : Msg\ .\ i = in.nbr_i,\ o = out.nbr_o\ \mid$$
$$\text{i.correct !m : Msg TO o} \longrightarrow\!\blacksquare\!\longrightarrow \text{o.in ?m : Msg FROM i}$$

L'obtention d'une spécification KORRIGAN à partir des diagrammes de communication, comme par exemple pour le composant `DataPorts` est directe (figure 3.17).

Il faut noter que puisque KORRIGAN traite de façon unifiée composition d'intégration (Integration Views) et composition concurrente (Composition Views), alors la méthode et les notations présentées sur la composition et la communication concurrente s'appliquent aussi en présence d'intégrations.

COMPOSITION VIEW DataPorts

SPECIFICATION	COMPOSITION ALONE
imports InputDataPort, OutputDataPort **variables** N : Natural	**is** in.nbr[1..N] : InputDataPort out.nbr[1..N] : OutputDataPort **with** true,{ i:[1..N].(in.i.enable **from** s:Server, out.i.enable **from** s:Server), (i:[1..N].o:[1..N].(i.correct !m **to** o:OutputDataPort, o.correct ?m **from** i:InputDataPort), ...}

Figure 3.17 : `DataPorts` en KORRIGAN (partiel)

Lorsque des composants à différents niveaux (*i.e.* pas directement fils d'un même composant) sont impliqués, nous adoptons un schéma de communication structuré. On n'ajoute pas de lien supplémentaire au niveau des nœuds de communication. En contrepartie, on ajoute l'information concernant cette communication au niveau de l'ancêtre commun (d'un point de vue décomposition) des sous-composants concernés.

Ceci est par exemple le cas pour traiter la communication entre un port de contrôle en entrée autorisant[4] un certain port de donnée en entrée et le port de donnée en sortie lui correspondant. L'ancêtre commun est le nœud de transit (figure 3.18). Cette approche correspond d'ailleurs à l'approche à base de contrôleurs utilisée dans notre génération de code orienté objet (chapitre 4).

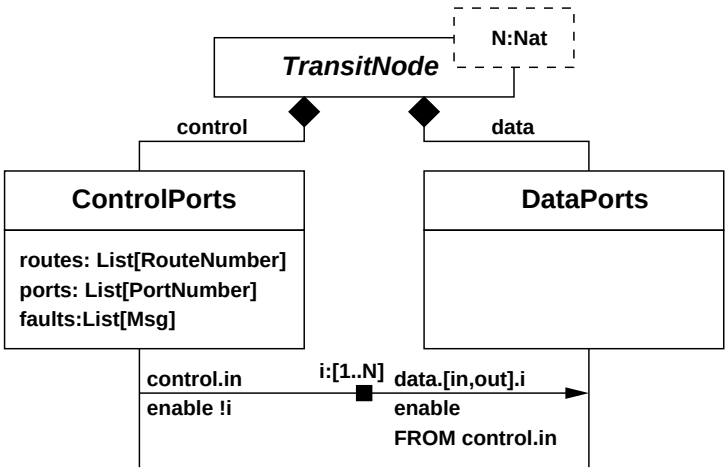


Figure 3.18 : `TransitNode` – Diagramme de communication (partiel)

Pour terminer sur la communication, remarquons que notre notation est suffisamment expressive pour

[4]Cette procédure d'autorisation est utilisée en KORRIGAN pour implémenter la création dynamique d'objets puisqu'il n'y a pas de support direct pour cela.

décrire différents types de communications et patrons de communications. Par exemple, il est possible d'exprimer la communication point à point (`ptp`) et la diffusion (`broadcast`) dans un patron de type client-serveur (figure 3.19). De tels patrons peuvent ensuite être utilisés dans la méthode.

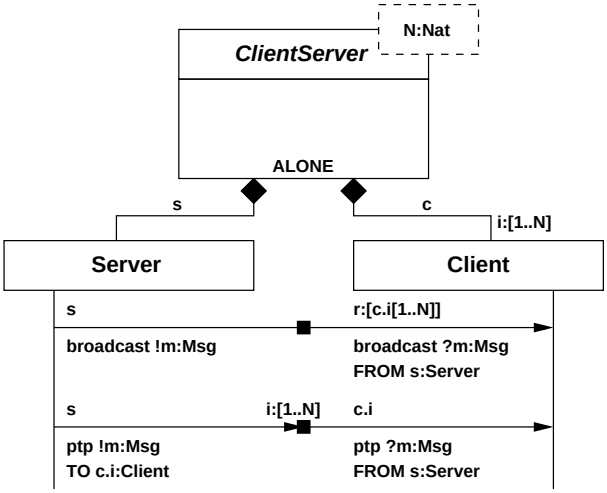


Figure 3.19 : Patron Client-Serveur

L'étape 3.3.2 a été répétée plusieurs fois jusqu'à obtention du schéma de la figure 3.14 ainsi que des différents diagrammes de composition et de communications présentés.

En ce qui concerne le nœud de transit, nous présentons l'application de la méthode au composant `InputDataPort`. Le typage des communications de l'`InputDataPort` est le suivant :

- `inData : ?m:Msg`
- `askRoute : !r:RouteNumber`
- `replyRoute : !r:RouteNumber?l:List[PortNumber]`
- `enable : !ident:PortNumber`
- `wrongRoute : !m:Msg`
- `correct : !ident:PortNumber!m:Msg`

3.3.3 Etape 3 : Composants séquentiels

Nous avons ici légèrement modifié la notation des agendas : là où seul $\vdash$ était utilisé pour indiquer une validation automatique, nous différencierons $\vdash_\mu$ (qui indiquera une vérification de modèle) de $\vdash_\lambda$ (qui indiquera une preuve algébrique).

Chaque composant séquentiel (figures 3.20 et 3.21) est décrit par un automate d'états finis. L'utilisation d'un automate se justifie pleinement en ce que, comme le fait remarquer [263], "la représentation sous forme d'arbre (comme souvent en LOTOS par exemple) du comportement entier d'un système n'est pratique que lorsque le nombre d'états est limité". Plus exactement, nous travaillons à l'aide de systèmes de transitions symboliques (STS), présentés dans le chapitre 2 et qui ont de bonnes propriétés d'abstraction.

Puisque nous sommes aussi intéressés par une notation inspirée d'UML, il nous parait important de comparer ici ces STS aux Statecharts (ceux de [140], ou ceux d'UML). Les différences principales sont :

- les STS sont plus simples que les Statecharts ;

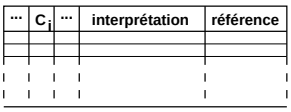
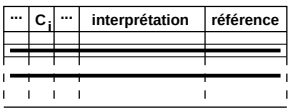
étape	expression / schéma	conditions de validation
3.1 : détermination des ports de O, OC, C, CC	`O, OC, C, CC`	∘ ensembles disjoints
3.2 : conditions associées aux communications sur les ports de OC ou CC catégorie : précondition ou comportement	$\mathtt{F}_i$: $\mathtt{C}_j$(catégorie)	∘ 1.2 (voir aussi 3.3.2.3)
3.3 : relations entre conditions	$\vdash \phi_i(\mathtt{C}_j)$	$\vdash_\lambda$ cohérence : $\vdash \wedge_i \phi_i(\mathrm{C}_j)$
3.4 : simplification des conditions		$\vdash_\lambda$ simplifications
3.5 : création de la table des conditions		
3.6 : élimination des cas impossibles		$\vdash_\lambda \phi_i(\mathrm{C}_j)$ (3.3)
3.7 : états	$\mathcal{E}_i =< ..., v(\mathtt{C}_j), ... >$ $v(\mathtt{C}_j) \in \{V, F\}$	

Figure 3.20 : Agenda des composants séquentiels – Partie 1

- les STS décrivent des composants séquentiels (la concurrence est réalisée par l'intermédiaire des External Structuring Views et du calcul d'un STS structuré et global à partir des STS des sous-composants, chapitre 2) ;
- il est possible de construire les STS à partir de conditions, comme cela est indiqué dans la suite, ainsi que de les dériver semi automatiquement à partir de l'analyse des besoins ;
- les STS peuvent être vus comme la représentation graphique d'une interprétation abstraite d'un type de donnée algébrique.

3.3.3.1 Etape 3.1 : détermination des ports de O, OC, C, CC

Nous cherchons à représenter les composants séquentiels par des automates dont les états correspondent à différents états abstraits dans lesquels peut se trouver le composant. Dans le domaine des spécifications algébriques, le terme de constructeur s'applique à une opération dont le codomaine correspond au type de données courant (ce qui est généralement assimilé à une modification du type de donnée). De même, la notion d'observateur correspond aux autres opérations. Nous cherchons à caractériser les opérations faisant passer le composant d'un état abstrait à un autre. Par analogie avec les spécifications algébriques, nous nommerons *constructeurs* toute opération faisant changer d'état abstrait et *observateurs* les autres opérations.

Dans nos automates les états sont caractérisés par des conditions (étapes 3.5 à 3.7). Les constructeurs sont donc les opérations modifiant les conditions et les observateurs celles qui ne modifient pas les conditions.

étape	expression / schéma	conditions de validation
3.8 : préconditions des opérations	$\mathcal{P}_k =< ..., v(\mathtt{C}_j), ... >$ $v(\mathtt{C}_j) \in \{V, F, \forall\}$	$\vdash_\lambda$ cohérence des préconditions par rapport à $\phi_i(\mathrm{C}_j)$ $\vdash$ correction par rap. à 3.2
3.9 : postcondition des opérations	$\mathcal{Q}_k =< ..., \Phi_i(\mathtt{C'}_j), ... >$	C' : C + nouvelles conditions
3.10 : relations entre conditions	$\vdash \phi_i(\mathtt{C'}_j)$	$\vdash_\lambda$ cohérence : $\vdash \wedge_i \phi_i(\mathrm{C'}_j)$ $\vdash_\lambda$ cohérence des postconditions par rapport à $\phi_i(\mathrm{C'}_j)$ $\vdash$ correction par rap. à 3.2
3.11 : calcul des transitions	$\mathcal{T} = f(\mathcal{E}, \mathcal{P}, \mathcal{Q})$	tous états atteignables (sinon retour possible en 3.3)
3.12 : choix d'un état initial en donnant la liste des opérations possibles (O_i) et celles impossibles ($\overline{O_j}$)	$\mathcal{E}_{init}$	$\vdash_\lambda$ cohérence de $\wedge_i \mathcal{P}_{O_i} \wedge_j \neg \mathcal{P}_{\overline{O_j}}$ $\vdash_\lambda$ un seul état initial
3.13 : simplifications de l'automate		
3.14 : traduction de l'automate dans le langage cible		$\vdash$ automate / spécification
3.15 : simplifications de la spécification		$\vdash$ simplifications correctes

Figure 3.21 : Agenda des composants séquentiels – Partie 2

Pour chaque processus, nous déterminons les quatre ensembles suivants :
- C, *constructeurs non conditionnés* : la communication sur ces ports modifie la valeur d'au moins une condition et n'est pas soumise à condition ;
- CC, *constructeurs conditionnés* : la communication sur ces ports modifie la valeur d'au moins une condition et est soumise à certaines conditions ;
- O, *observateurs non conditionnés* : la communication sur ces ports ne modifie la valeur d'aucune condition et n'est pas soumise à condition ;
- OC, *observateurs conditionnés* : la communication sur ces ports ne modifie la valeur d'aucune condition et est soumise à certaines conditions.

Ces ensembles sont récapitulés dans le tableau 3.2.

Le but de cette distinction est de faciliter la création de l'automate du processus. De plus, ces ensembles correspondent aussi à des ensembles d'opérations du type sous-jacent au processus.

$\sigma = C + CC + O + OC$ dénote l'ensemble des ports du processus. Il faut noter qu'un port reliant un processus P à un processus Q peut appartenir à deux ensembles différents selon que l'on s'occupe de P ou de Q.

3.3.3.2 Etape 3.2 : conditions associées aux communications

Par "condition" nous entendons aussi bien ce qui autorise une communication (précondition) que ce qui peut modifier l'effet d'une communication (condition de comportement). Dans le cas de OC et de

	non conditionnés	conditionnés
modification état interne	C	CC
conservation état interne	O	OC

Table 3.2 : Ensembles C, CC, O et OC.

CC nous donnons et nommons les conditions.

Il faut ici s'assurer que l'on a bien pris en compte toutes les conditions énumérées lors de l'étape 1.2. Toutefois, certaines d'entres elles sont prises en compte lors de la composition parallèle des sous-processus (étape 3.3.2.3).

L'application des étapes 3.1 et 3.2 à l'`InputDataPort` a permis de recenser les conditions suivantes : autorisé (le port est actif), reçu (un message reçu et non traité), demandé (demande d'information de routage envoyée), répondu (réponse à la demande de routage reçue) et routeErr (erreur de routage). La condition routeErr porte sur la valeur de la variable l de `replyRoute` après communication (voir le typage de la communication à la fin de l'étape 2).

Les opérations ont été rangées dans les différentes catégories et sont conditionnées comme indiqué ci-dessous :

- O : $\emptyset$,
- C : {`enable`},
- OC : $\emptyset$,
- CC : {`inData` : autorisé $\wedge$ $\neg$reçu, `askRoute` : autorisé $\wedge$ reçu $\wedge$ $\neg$répondu, `replyRoute` : autorisé $\wedge$ reçu $\wedge$ demandé $\wedge$ $\neg$répondu, `correct` : autorisé $\wedge$ reçu $\wedge$ demandé $\wedge$ répondu $\wedge$ routeErr), `wrongRoute` : autorisé $\wedge$ reçu $\wedge$ demandé $\wedge$ répondu $\wedge$ $\neg$routeErr}

3.3.3.3 Etape 3.3 : relations entre conditions

Des formules logiques ($\vdash \phi_i(\mathrm{C}_j)$) expriment des propriétés reliant les conditions. Dans cette étape, un prouveur de théorèmes peut être employé afin de vérifier l'absence de contradictions entre ces formules.

Pour l'`InputDataPort` : reçu $\Rightarrow$ autorisé, demandé $\Rightarrow$ reçu, répondu $\Rightarrow$ demandé, routeErr$\Rightarrow$ répondu. Il n'y a aucune contradiction.

3.3.3.4 Etape 3.4 : simplification des conditions

Les formules logiques données précédemment par le spécifieur vont éventuellement permettre de simplifier certaines conditions, voire d'en restreindre le nombre. Cette étape peut être automatisée par l'utilisation d'un prouveur de théorèmes basé sur la réécriture (par exemple LP [127]).

L'application à l'`InputDataPort` donne les résultats suivants (modifications uniquement) :

- CC : {`inData` : autorisé $\wedge$ $\neg$reçu, `askRoute` : reçu $\wedge$ $\neg$répondu, `replyRoute` : demandé $\wedge$ $\neg$répondu, `correct` : répondu $\wedge$ routeErr, `wrongRoute` : répondu $\wedge$ $\neg$routeErr}

autorisé	reçu	demandé	répondu	routeErr	état
V	V	V	V	V	RI (Route Incorrecte)
V	V	V	V	F	RC (Route Correcte)
V	V	V	F	F	AR (Attente Réponse)
V	V	F	F	F	PAD (Pret À Demander)
V	F	F	F	F	PAR (Pret À Recevoir)
F	F	F	F	F	NA (Non Autorisé)

Table 3.3 : Table des conditions du Data Port In.

3.3.3.5 Etapes 3.5 à 3.7 : obtention des états

L'idée est qu'un processus peut se trouver dans différents états abstraits dans lesquels il est en mesure ou non de rendre un service (c'est-à-dire d'autoriser une communication sur un port). Les états sont donc obtenus par composition des conditions de communication (un recensement des conditions a été fait lors du travail sur les ports - étapes 3.2 à 3.4 - dans une table de vérité - 3.5).

Des formules logiques ($\vdash_\lambda \phi_i(\mathrm{C}_j)$) expriment des propriétés reliant ces conditions et permettent de supprimer les états incohérents (3.6).

Un automate est construit avec un état (3.7) pour chaque cas différent dans la table de composition des états (3.5). Les n-uplets correspondant à la table de vérité y sont notés.

La table de l'`InputDataPort` (après suppression des états incohérents) est donnée en table 3.3.

3.3.3.6 Etapes 3.8 à 3.11 : obtention des transitions

Pour chacun des ports, on donne les préconditions (3.8) et les postconditions (3.9) en fonction des conditions trouvées auparavant. $\mathcal{P}$ désigne les valeur des conditions avant communication et $\mathcal{Q}$ leur valeur après communication. $\forall$ signifie que la valeur de la condition n'importe pas et = qu'elle n'est pas modifiée par l'opération.

Plusieurs vérifications sont possibles. Tout d'abord, les préconditions doivent être cohérentes par rapport aux relations existant entre les conditions. Ainsi, une précondition ne peut réclamer que deux conditions c_1 et c_2 soient toutes les deux vraies si d'un autre côté, on sait (étape 3.3) que $\vdash c_1 \Rightarrow \neg c_2$. Il est d'autre part possible de restreindre certaines préconditions automatiquement. En conservant la relation ci-dessus, si la précondition est que c_1 soit vraie et $\forall$ pour c_2, alors on la restreint en signifiant que c_2 doit être fausse. Une seconde vérification peut être faite en ce qui concerne le classement des ports dans les différents ensembles (3.1). Les opérations dont la précondition est $\forall$ pour toutes les conditions doivent être dans O ou C, les autres dans OC ou CC. À ce niveau, il est donc possible de devoir revenir à l'étape 3.1 et de devoir redérouler l'agenda à partir de ce point.

En ce qui concerne les postconditions, les vérifications sont de même ordre. Après avoir donné d'éventuelles relations faisant intervenir les nouvelles conditions apparues dans les postconditions (c'est l'étape 3.10, identique à l'étape 3.3), il faut vérifier la cohérence des postconditions. Au niveau du classement dans les ensembles, les opérations ne modifiant la valeur d'aucune condition ($\forall j \bullet \mathcal{Q}_k =< \ldots, \Phi(\mathrm{C'}_j), \ldots >, \Phi(\mathrm{C'}) =\mathrm{C'}$) doivent être dans O ou OC, et les autres dans C ou CC. Comme pour les préconditions, il est donc possible de devoir revenir à l'étape 3.1.

Les préconditions et postconditions des opérations de l'`InputDataPort` se trouvent ci-dessous avec les notations suivantes : a pour autorisé, r pour reçu, d pour demandé, rep pour répondu et rerr pour routeErr.

`enable`	a	r	d	rep	rerr
$\mathcal{P}$	$\forall$	$\forall$	$\forall$	$\forall$	$\forall$
$\mathcal{Q}$	V	=	=	=	=

`inData`	a	r	d	rep	rerr
$\mathcal{P}$	V	F	$\forall$	$\forall$	$\forall$
$\mathcal{Q}$	=	V	=	=	=

`askRoute`	a	r	d	rep	rerr
$\mathcal{P}$	V	V	$\forall$	F	$\forall$
$\mathcal{Q}$	=	=	V	=	=

`replyRoute`	a	r	d	rep	rerr
$\mathcal{P}$	V	V	V	F	$\forall$
$\mathcal{Q}$	=	=	=	V	l=[]

`correct`	a	r	d	rep	rerr
$\mathcal{P}$	V	V	V	V	F
$\mathcal{Q}$	=	F	F	F	=

`wrongRoute`	a	r	d	rep	rerr
$\mathcal{P}$	V	V	V	V	V
$\mathcal{Q}$	=	F	F	F	=

L'idée générale pour l'obtention de l'automate est qu'un état abstrait est identifié par un ensemble de services que peut rendre le processus. Les transitions correspondent à des opérations qui sont possibles sous certaines conditions (et donc à partir de certains états ; nous avons ici les points de départ des transitions) et conduisent ou non à modifier l'état abstrait (c'est le cas des constructeurs qui font changer d'état abstrait ; nous avons ici les points d'arrivée des transitions).

Les préconditions décrivent l'ensemble des états dans lesquels une communication sur un port est possible ou non. Les éléments de C ou O ne sont pas préconditionnés et ceux de CC ou OC ne sont préconditionnés que par leur condition d'application définie plus tôt par le spécifieur.

Les postconditions décrivent l'effet des opérations sur la valeur des conditions et donc le changement d'état abstrait.

Il existe en général des *cas critiques*. Il s'agit des cas pour lesquels le type de donnée lié au processus représenté à l'aide du système de transitions est dans un état limite dans lequel l'appel de l'opération fait changer la valeur des conditions. Il faut se rappeler que les états de nos systèmes de transitions symboliques représentent une classe d'équivalence (algébrique), et donc plusieurs états concrets. L'effet des opérations sur les conditions d'état ne peut donc pas être exprimé uniquement en fonction des conditions d'état. Il est nécessaire d'utiliser de nouvelles conditions pour tester si le type de données correspondant au processus est ou non dans un état critique. Méthodologiquement, ces nouvelles conditions vont apparaître lorsque l'on cherche à exprimer les postconditions (3.9).

En ce qui concerne les transitions (3.11), les automates peuvent être construits séparément (un par opération), puis intégrés, en prenant dans l'ordre :

– états initiaux possibles du processus ainsi que l'ensemble des observateurs sans conditions
– chacun des observateurs avec conditions
– chacun des constructeurs sans conditions
– chacun des constructeurs avec conditions.

Les états initiaux indiquent dans quel(s) état(s) sont instanciés les processus.

Les observateurs sans conditions correspondent à des transitions d'un état vers lui-même (un observateur ne modifiant pas l'état abstrait) et sont possibles dans tous les états.

Les observateurs conditionnels ne sont autorisés qu'à partir des états ayant la bonne valeur de vérité pour la condition correspondante. Ils correspondent à des transitions de chacun de ces états vers eux-mêmes.

L'idée générale pour obtenir les transitions correspondant aux constructeurs est la suivante : à l'aide

d'un n-uplet représentant les valeurs des conditions avant une opération (répondant donc à sa précondition) et des postconditions, il est possible d'obtenir le n-uplet correspondant à la valeur des conditions après cette opération (postcondition).

Les transitions correspondant aux constructeurs non conditionnés permettent de passer d'un état à un autre mais n'ont pas de précondition associée.

Pour les trouver on procède comme suit :
- prendre un état e (n-uplet de booléens)
- en supposant que ce n-uplet corresponde aux valeurs de précondition de l'opération, trouver le n-uplet correspondant aux valeurs de postcondition et l'état f correspondant
- on a une transition entre e et f
- recommencer avec un état non encore choisi

Chaque n-uplet correspondant à un et un seul état, on a donc une relation entre états représentant les transitions de l'automate.

Les constructeurs conditionnés permettent de passer d'un état à un autre lorsque certaines préconditions sont vérifiées. Pour les trouver on procède comme pour les constructeurs non conditionnés mais les nouvelles conditions apparaissant au niveau des postconditions sont reportées en tant que préconditions sur les transitions. Une notation particulière est utilisée au niveau des transitions. Une transition d'étiquette `[c] e [c']` représente une communication de nom `e` conditionnée par deux préconditions : `c` concernant l'état du processus (données internes) avant la communication , et `c'` concernant des contraintes sur les données reçues lors de la communication. C'est à rapprocher de la différence entre garde et précondition en LOTOS.

Dans certains cas, les conditions inhérentes aux états impliquent celles des préconditions et postconditions et simplifient ainsi les tests à faire.

Cette démarche automatique permet de repérer des cas qui peuvent ne pas être détectés sans suivre la méthode. Ces cas correspondent aux cas critiques.

3.3.3.7 Etape 3.12 : état initial

Pour déterminer l'*état initial*, on examine les services (qui sont associés à des ports) que le processus doit rendre ou non (contraintes). A partir des préconditions des ports et de la table de création des états on trouve les états susceptibles d'être initiaux. S'il n'y en a pas, c'est que le spécifieur a donné des contraintes non cohérentes pour son état initial. En ce qui concerne certains types de spécification (LOTOS, SDL par exemple), il est nécessaire de n'avoir qu'un seul état initial (état dans lequel "démarre" le processus lors de son instanciation). Dans le cas où il y a plusieurs états initiaux possibles, un choix peut être fait en ajoutant une contrainte sur les services ou en faisant un choix arbitraire.

La spécification de l'`InputDataPort` en KORRIGAN est donnée dans la figure 3.22.

Le STS correspondant est donné dans la figure 3.23. On peut remarquer que, comme signalé plus haut, les conditions de `replyRoute` concernent les données reçues lors de cette communication et sont donc représentées après le nom de l'opération dans les transitions.

DYNAMIC VIEW `InputDataPort`

SPECIFICATION

imports
Msg, RouteNumber, PortNumber
variables fc: FaultyCollection
ops
enable
FROM InputControlPort
inData ?m:Msg
FROM InputControlPort
askRoute !r:RouteNumber
TO InputControlPort
replyRoute ?l:List[PortNumber]
FROM InputControlPort
wrongRoute !m:Msg
TO FaultyCollection
correct !m:Msg
TO OutputDataPort
axioms ...

ABSTRACTION

conditions
enable, received, asked, replied,
routeErr
with
replied ⇒ enabled
asked ⇒ received
replied ⇒ asked
routeErr ⇒ replied
initially ¬ enabled

OPERATIONS

enable
pre: true
post: enable' : true
inData
pre: enable ∧ ¬ received
post: received' : true
...

Figure 3.22 : `InputDataPort` en KORRIGAN (partiel)

3.3.3.8 Etapes 3.13 à 3.15 : simplifications et traduction

Il est possible de traduire l'automate vers divers langages de spécification en lui appliquant des schémas de traduction. Cette technique a été appliquée à LOTOS et à SDL et est détaillée dans le chapitre 4.

En amont, l'automate lui-même peut parfois être simplifié selon des techniques identiques ou proches de celles employées dans la traduction. [11] propose ainsi trois techniques de simplifications :

- regroupement des transitions similaires, c'est-à-dire ayant même opération, même condition et des destinations identiques (les transitions en boucle sont un cas particulier de transitions similaires où l'on considère que la destination identique est "-" au sens de SDL), en agrégeant les états d'origine.
- composition d'états : les états fortement connexes sont regroupés et des boucles conditionnées remplacent les transitions initialement existantes entre les états de départ. Un état composé est en fait un sous-automate.
- composition de transitions : des notations particulières pour des transitions dites "séquentielles" (deux transitions telles que l'état e de destination de l'une soit l'état d'origine de l'autre et que le degré de e soit 2) et des transitions dites "parallèles" (même origine et même destination) simplifient l'automate.

3.3.4 Etape 4 : Types de données

La dernière étape est de construire la spécification des types de données associés à chaque processus, ainsi que celle des types de données manipulés par les processus. En ce qui concerne le type de données associé à chaque processus (partie statique des structures de vues dans le cas du langage KORRIGAN),

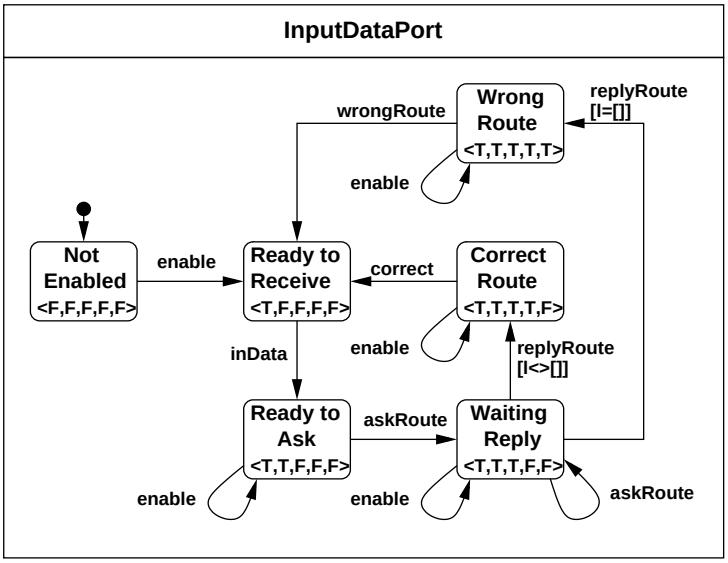


Figure 3.23 : `InputDataPort` – Diagramme de comportement (STS)

le travail effectué au cours des étapes précédentes fournit l'essentiel de la signature. En effet, les noms des opérations et leurs profils sont obtenus automatiquement à partir du typage des communications (voir la correspondance entre signatures statiques et signatures dynamiques), et à partir des différentes conditions identifiées en construisant les automates. Quelques opérations supplémentaires peuvent être nécessaires pour exprimer les axiomes. La plupart des axiomes[5] sont écrits dans un style "constructif" ce qui nécessite d'avoir identifié les générateurs.

Cette discipline par constructeurs permet entre autres, comme le fait remarquer [126], de dériver des implémentations concrètes de la spécification grâce à des outils comme Caesar, ou bien encore de leur appliquer les principes utilisés dans la génération de code, chapitre 4.

[14] développe une méthode pour obtenir le type abstrait associé à un automate et pour calculer un ensemble minimal d'opérations nécessaire pour atteindre tous les états. Dans notre exemple, cet ensemble est `init, enable, inData, askRoute, replyRoute`. [14] utilise l'Ω-dérivation [44] pour écrire les axiomes (qui sont des équations conditionnelles). Pour extraire les axiomes décrivant les propriétés d'une opération `op(dpi:InputDataPort)`, on recherche les états où cette opération peut être effectuée ainsi que les générateurs permettant d'atteindre ces états, ce qui permet d'obtenir les prémisses et les membres gauches des axiomes.

Nous montrons ici une partie de ce processus automatique pour les axiomes de l'opération `correct_c`. Cette opération est associée à la transition `correct` qui est possible dans l'état RC, qui est atteignable via les compositions de générateurs qui apparaissent en partie gauche des axiomes ci-dessous. Les prémisses expriment les conditions sur les états de départ et sur la valeur de la variable `l`.

```
RC(dpi) => correct_c(enable(dpi))
           = correct_c(dpi)
AR(dpi) /\ not(l=[]) => correct_c(replyRoute(askRoute(dpi),l))
                        = correct_c(dpi)
AR(dpi) /\ not(l=[]) => correct_c(replyRoute(enable(dpi),l))
```

[5] À l'exception des formules exprimant les relations entre conditions.

```
                          = correct_c(dpi)
PAD(dpi) /\ not(l=[]) => correct_c(replyRoute(askRoute(enable(dpi)),l))
                          = correct_c(dpi)
PAR(dpi) /\ not(l=[]) => correct_c(replyRoute(askRoute(inData(dpi,m)),l))
                          = dpi
```

La spécification algébrique peut ensuite être utilisée pour effectuer des preuves de propriétés à des fins de vérification et de validation de la spécification.

Remarques

Le fait de traiter la partie données après la partie comportement permet, comme nous l'avons fait remarquer plus haut, d'obtenir un ensemble *minimal et complet* d'opérations dans la partie données. Ceci est entendu par rapport aux besoins préalablement exprimés dans la description informelle du processus.

La minimalité est assurée par le fait que seules des opérations servant dans la spécification de la partie dynamique sont spécifiées dans la partie algébrique.

La "complétude" résulte du couplage qui a été fait entre un processus (resp. partie comportement des vues) et le type de données (reps. partie données des vues) qu'il gère. Un processus peut être vu comme l'interface (au sens des services) d'un type de données. Les opérations constructrices du type de donnée résultent ainsi directement des communications sur les portes (de C ou CC) du processus associé.

3.4 Conclusions

Comme l'a fait remarquer [18], formalisme et méthode sont intimement liés. La définition de méthodes associées aux formalismes de spécification permet leur utilisation hors d'un contexte universitaire. Nous pensons que ce support doit se baser sur l'utilisation de notations graphiques formellement bien définies, sur l'aide à l'écriture des spécifications, et sur la connexion avec des méthodes existantes (formelles ou semi-formelles). Le premier point a été abordé dans le chapitre 3 par l'emploi des systèmes de transition symboliques. Le dernier point a été abordé ici sous l'axe consistant à réutiliser partiellement une partie des notations UML dans le cadre de notre méthode. Il sera aussi abordé dans le chapitre 4, dans le cadre des possibilités de traduction de nos spécifications vers les formalismes LOTOS et SDL. Dans ce chapitre nous nous sommes principalement intéressés à l'aide à l'écriture des spécifications.

Après avoir présenté les principales approches utilisées dans la spécification des systèmes mixtes, et principalement des protocoles et services de communications et des systèmes répartis en LOTOS et SDL, nous avons présenté une méthode [223] qui améliore les méthodes existantes sur plusieurs points :

- la définition semi-formelle de la méthode par l'utilisation des agendas ;
- l'utilisation à la fois de notations graphiques basées sur UML et de notations textuelles formelles [71] ;
- l'utilisation de formalismes abstraits et généraux (schémas de composition et systèmes de transitions symboliques) basés sur le formalisme KORRIGAN et permettant l'adéquation de la méthode à plusieurs formalismes mixtes (KORRIGAN, LOTOS, SDL) ;
- la construction semi-automatique des systèmes de transitions représentant les comportements à partir de l'analyse des besoins ;
- une automatisation partielle et des guides facilitant la production des parties axiomatiques.

Nous avons montré dans ce chapitre comment cette méthode pouvait être employée sur une étude de cas de taille réelle. Nous l'avons aussi appliquée sur plusieurs autres exemples : ascenseur, files d'attente d'un hôpital, système de messagerie téléphonique, gestionnaire de mots de passe sous Unix.

CHAPITRE 4

ASK : un Atelier pour Spécifier avec KORRIGAN

Dans ce chapitre nous présentons un environnement logiciel dédié à la spécification mixte et aux modèles, langages et méthodes que nous avons présentés dans les précédents chapitres. Nous l'avons conçu dans un cadre orienté objet de façon à ce qu'il soit ouvert et permette l'intégration de nouveaux langages et outils, existants ou non. Nous présentons d'abord les grandes lignes de cet environnement, les grands principes qui le régissent puis les outils qui y sont implémentés. Nous insisterons plus particulièrement sur deux d'entre eux, liés aux mécanismes de traduction de KORRIGAN vers d'autres formalismes dans un but de vérification et validation, et à ceux de la génération de code orienté objet pour nos spécifications mixtes.

4.1 L'environnement ASK

L'architecture de l'environnement ASK (Atelier pour Spécifier en KORRIGAN) est présentée dans la figure 4.1.

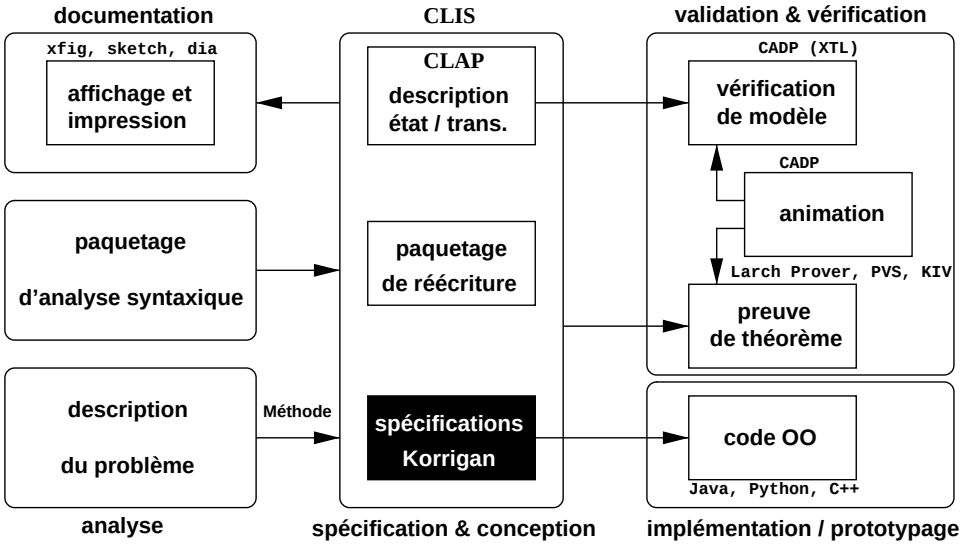


Figure 4.1 : ASK– Architecture

ASK est articulé autour d'une librairie de spécifications, CLIS (Class Library of Interfaces to Specification languages). Cette librairie est constituée d'interfaces pour langages et outils de spécification. Ces interfaces sont la description en termes de classes (réification dans le modèle orienté objet) des structures liées aux langages de spécification. Ces interfaces nous permettent de lire et d'écrire les spécifications dans ces langages. Il s'agit par exemple de viser des langages (de spécification ou correspondant à des outils) particuliers dans le but de valider et vérifier les spécifications écrites en KORRIGAN.

Dans cette optique, la librairie prend en compte des interfaces pour le Larch Prover [127], ou encore LOTOS et SDL. La prise en compte de PVS [214] est prévue. Bien sûr, les concepts liés à notre modèle ainsi qu'au langage KORRIGAN sont implémentés dans CLIS. Ces interfaces prennent aussi en compte des langages permettant de documenter les spécifications. Actuellement, l'outil XFig [1] est pris en compte, d'autres formats (UML, HTML, XML) sont potentiellement aussi candidats.

Nous avons aussi développé notre propre moteur de réécriture conditionnelle dans ASK. Il s'agit d'un moteur développé dans le contexte d'un travail antérieur consistant à donner une sémantique par réécriture aux modèles à objets [221]. Dans le futur, si le besoin d'un moteur plus efficace se faisait sentir, il serait possible de rajouter des interfaces vers des langages de systèmes de réécriture externes, comme ELAN [48] par exemple. Pour l'instant nous préférons une facilité d'utilisation (moteur écrit dans le même environnement que les autres outils) à la rapidité de traitement des réécritures. Ce moteur est basé sur une réification dans le contexte objet des concepts algébriques, et des classes ont été définies pour permettre la description de divers termes algébriques (termes, équations, offres, gardes, axiomes conditionnels pour n'en citer que quelques-uns).

Un paquetage est dédié à l'analyse syntaxique. Il est constitué d'un ensemble d'analyseurs lexicaux, d'analyseurs syntaxiques et d'interpréteurs, servant à lire les différents types de spécifications à partir de fichiers et générer les instances des classes CLIS correspondantes.

Enfin, une partie importante de CLIS est dédiée à la description des systèmes à base d'états et de transitions. La librairie CLAP (Class Library for Automata in PYTHON) contient des interfaces pour différents types de ces systèmes, dont bien sûr les systèmes de transitions symboliques utilisés dans notre modèle. Différents outils génériques aux systèmes de transitions y ont été implémentés.

4.1.1 Principes de conception

La conception de l'environnement ASK suit plusieurs principes.

Le premier principe est la possibilité d'interfacer ASK avec des outils et environnements existants, par exemple des outils de vérification de modèle (comme XTL [191] dans CADP [110]), des prouveurs de théorèmes (comme le Larch Prover [127], KIV [234], ELAN [48], PVS [214] ou encore HOL-CASL [207]), et des langages de programmation (comme Java [131] ou C++ [257]). Puisque ces langages et outils sont nombreux et évoluent, l'environnement doit être ouvert et extensible.

Un second principe est de fournir des outils suffisamment généraux et qui puissent être utiles à d'autres environnements ou d'autres formalismes. Par exemple, dans cette optique, la librairie CLAP peut être utilisée pour calculer les compositions (synchrone ou asynchrone, et avec différentes sémantiques de concurrence) de n'importe quel type de diagramme à base d'états et de transitions (automates, réseaux de Petri, systèmes de transition symboliques selon l'approche KORRIGAN).

Pour satisfaire à ces deux principes, nous utilisons les concepts orientés objet à la fois dans la conception et l'implémentation de notre environnement. Nous avons choisi le cadre orienté objet puisqu'il as-

sure de bonnes propriétés [202] et que nous avons une bonne expérience de ce modèle dans un contexte formel [13, 221].

4.1.2 Le langage PYTHON

L'implémentation de l'environnement est faite en PYTHON [186]. Il s'agit d'un langage interprété orienté objet, et de ce fait il est très pratique pour produire, rapidement, à la fois des scripts et des prototypes d'environnements complexes.

Une particularité importante est que PYTHON est un logiciel libre, "open source". Les programmes écrits en PYTHON sont portables sur plusieurs plateformes (Unix, Linux, Windows, MacOS). PYTHON est à la fois proche de LISP, PERL et C++, mais est beaucoup plus lisible. Il s'agit d'un langage typé dynamiquement. Il dispose à la fois de fonctionnalité orientées objet et fonctionnelles (lambda expressions).

PYTHON dispose d'un MOP (Meta Object Protocol [165]) simple qui en a fait le langage d'implémentation de méta-langages concurrents à objets comme ATOM [217]. PYTHON dispose aussi d'un mécanisme de traitement des exceptions, de types de données de base puissants et d'une librairie de modules complète (génération d'analyseurs syntaxiques, programmation CORBA, analyse syntaxique XML).

4.1.3 Processus d'intégration

En nous basant sur ce que nous attendions de notre environnement orienté objet, nous avons défini un processus général (figure 4.2) pour intégrer de nouveaux langages et de nouveaux outils dans ASK.

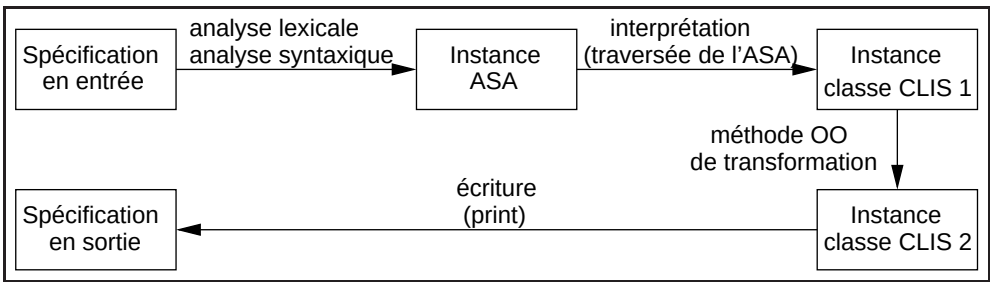


Figure 4.2 : ASK– Processus d'intégration

À partir de la description en termes de syntaxe abstraite d'une spécification donnée, nous obtenons une instance d'un arbre de syntaxe abstraite (ASA) en utilisant un mécanisme général d'analyse syntaxique (section 4.4.1). A partir de cette instance, un interpréteur construit une instance d'une des classes de la hiérarchie CLIS. Des transformations (coercitions, transformations d'instance à instance) peuvent ensuite être appliquées à cette instance pour obtenir une instance d'une autre classe de la librairie CLIS.

Prenons un exemple. Supposons que nous voulions transformer la spécification algébrique (partie données) contenue dans une structure de vue en une spécification CASL ou LP. Ceci peut être fait en définissant une méthode de la classe source (parties données des vues) vers la classe cible (par exemple spécifications LP). Enfin, la méthode d'impression contenue dans chacune des classes de CLIS (et donc dans la classe réifiant les spécifications LP) serait utilisée pour imprimer la spécification finale.

Une fois que l'analyse syntaxique et l'impression pour un langage donné ont été implémentés, la tache principale du concepteur est donc d'implémenter des méthodes de conversion entre plusieurs

classes CLIS. C'est ce processus général qui a par exemple été utilisé pour la génération de documentation XFig pour les systèmes de transitions décrit par les instances des classes CLAP, ou bien encore pour la génération de LOTOS ou de SDL à partir de KORRIGAN.

Il faut remarquer que ce processus, peut être effectué dans un autre langage (C par exemple) puis interfacé avec PYTHON et notre atelier en utilisant SWIG (Simplified Wrapper and Interface Generator, [36]).

4.2 La librairie CLIS

CLIS est une hiérarchie de classes (figure 4.3) extensible correspondant à une classification de spécifications. Par spécification nous entendons aussi bien langage de spécification formel que langage de programmation ou de documentation. CLIS fournit des classes correspondant aux éléments de notre modèle à base de vues (figure 2.1), ainsi que pour d'autres formalismes.

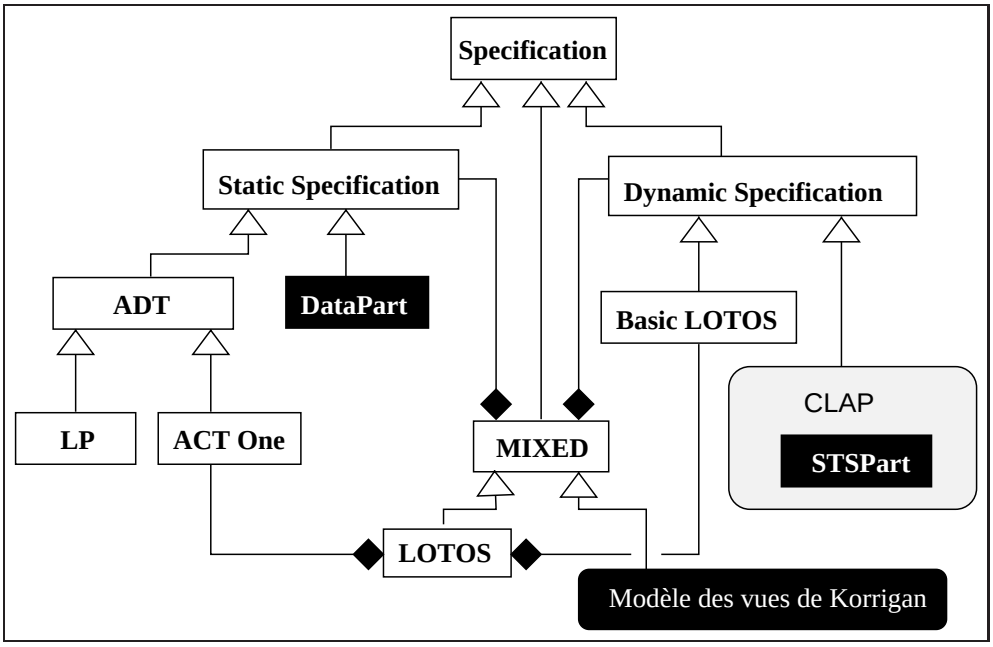


Figure 4.3 : Diagramme de classes – CLIS (partie)

Nous utilisons une hiérarchie générale pour les spécifications, avec des sous-classes correspondant aux langages dédiés à la spécification des types de données, ou aux comportements dynamiques. Nous essayons de classifier les différentes approches (voir aussi le chapitre 1 sur ces aspects), mais il s'agit d'une tâche difficile. En cela, notre hiérarchie est plus pragmatique que théorique et n'est proposée que dans un but conceptuel (conception de l'environnement ASK).

Comme exemple de langage dédié aux aspects statiques nous avons le langage de l'outil LP, et comme exemples de langages dédiés aux comportements dynamiques nous avons les membres de la hiérarchie CLAP (section 4.3). Le langage KORRIGAN est réifié par plusieurs sous-classes de la classe `MIXED` représentant les langages de spécification mixte. LOTOS et SDL sont d'autres exemples de sous-classes de cette classe. Dans le futur, nous envisageons d'intégrer dans cette hiérarchie d'autres langages de spécification, comme CASL ou CASL-LTL.

4.3 La librairie CLAP

Les systèmes à base d'états et de transitions sont un important formalisme de spécification des aspects dynamiques des systèmes. Ils sont aussi au cœur du langage KORRIGAN, c'est pourquoi une partie de la librairie CLIS leur est dédiée.

CLAP permet la définition de différents types de systèmes à base d'états et de transitions en fournissant une librairie de classes extensible leur correspondant. Dans l'état actuel de la librairie, il y a ainsi par exemple des classes pour les systèmes avec paramètres d'état (initialité des états, coloration) ou de transition (étiquettes, émissions et réceptions, gardes), ainsi que pour les réseaux de Petri.

Il est facile d'ajouter une nouvelle classe correspondant à un nouveau type de diagramme à base d'états et de transitions en sous-classant une ou plusieurs classes existantes.

Les diagrammes sont enregistrés dans des fichiers suivant un format interne générique. Un analyseur syntaxique générique pour ce format est disponible. Les diagrammes peuvent être automatiquement transformés en formats d'entrée pour outils graphiques (au choix, XFig ou DOT). Cette fonctionnalité a été réalisée en suivant le principe décrit dans la figure 4.2.

La partie originale de cette librairie concerne en fait les systèmes de transition symboliques. Il n'existe en effet actuellement aucun paquetage dédié à ce type de systèmes. Nous n'avons pas pour le moment une implémentation très efficace des STS, mais nous ne pensons pas que l'efficacité soit actuellement le problème le plus important lié à ces systèmes de transition. Les STS sont abstraits et ont donc généralement une taille réduite qui n'impose pas les mêmes besoins en termes d'efficacité que les automates par exemple. Les algorithmes de stockage de graphes habituels sont donc suffisamment efficaces pour l'instant.

Une partie de la hiérarchie actuelle de CLAP est présentée dans la figure 4.4.

Différents types de systèmes de transitions symboliques sont représentés par les classes `GATDiagram` et `STSKorriganDiagram`. La classe `GATDiagram` décrit les STS généraux avec termes construits à partir de signatures statiques et sans offres, ni émissions ou réceptions. Elle peut être utilisée pour décrire les STS des vues statiques. La classe `STSKorriganDiagram` est celle utilisée dans le langage KORRIGAN pour les STS des vues dynamiques, elle utilise des termes construits à partir de signatures dynamiques (chapitre 2).

4.4 Outils de l'atelier ASK

Nous avons implémenté plusieurs outils dans l'environnement ASK. Il s'agit soit d'outils généraux (analyse syntaxique, génération de formats de documentation), soit d'outils dédiés aux formalisme KORRIGAN et à notre modèle à base de vues (outils liés à sa sémantique opérationnelle et à la construction d'un STS global, traductions)

Le point est plutôt mis sur l'aspect général de l'ensemble des outils et l'obtention rapide de prototypes. L'efficacité n'est pas un problème clé pour l'instant. Il faut noter que l'ensemble des mécanismes sont implémentés par des classes et sont donc faciles à étendre. D'autres outils ont été expérimentés dans d'autres contextes (Smalltalk, CLOS) et seront intégrés prochainement.

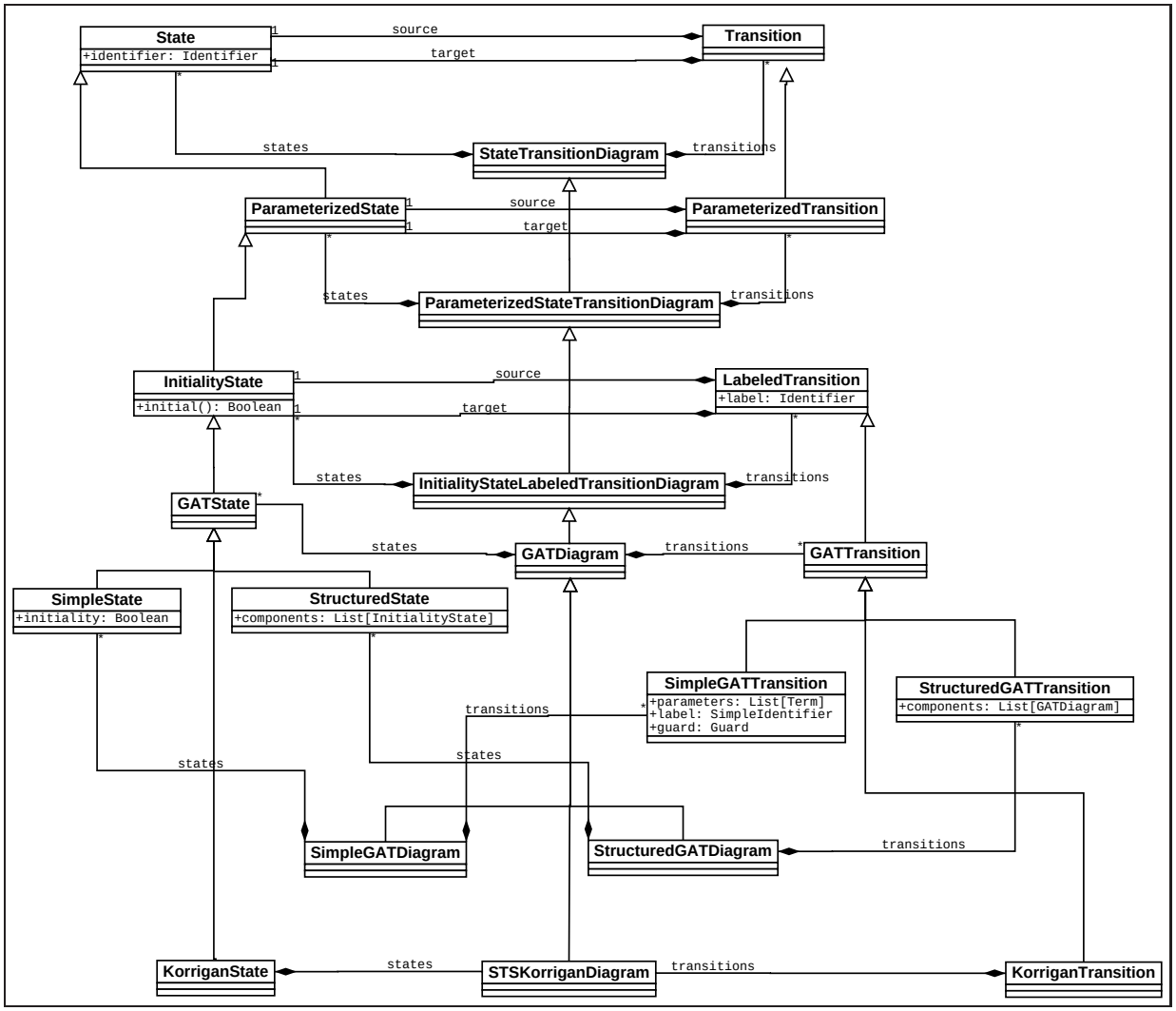


Figure 4.4 : Diagramme de classes - CLAP (partie)

4.4.1 Analyse syntaxique

Nous choisissons de réutiliser une approche simple et générale (figure 4.5) basée sur SPARK [29].

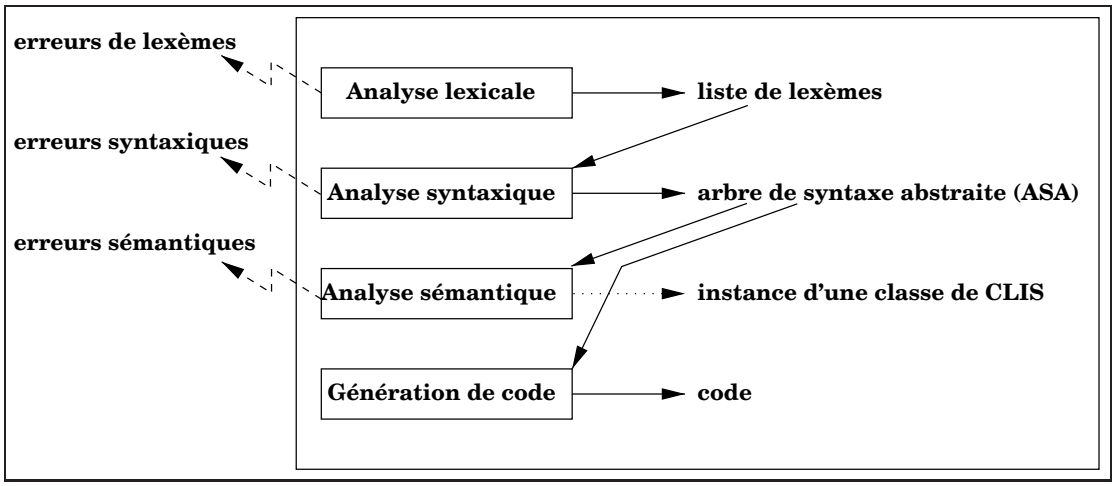


Figure 4.5 : Principes d'analyse syntaxique basés sur SPARK [29]

SPARK définit un paquetage avec quatre niveaux dédiés respectivement à l'analyse lexicale, à l'analyse syntaxique, à l'analyse sémantique et à la génération de code. Chacun de ces niveaux fournit une classe générale ainsi que quelques méthodes.

Pour définir un système particulier (*i.e.* dédié à un langage donné), il faut sous-classer ces classes générales et rédéfinir certaines méthodes. Par exemple, dans le but de produire un analyseur lexical pour un nouveau langage de spécification, nous sous-classons la classe générique `GenericScanner`. Nous redéfinissons aussi les méthodes qui servent à déclarer une expression régulière et les actions associées. Plus la hiérarchie de classes s'étoffe et plus les parties à sous-classer sont réduites. Le travail consiste parfois juste à redéfinir un patron ou un mot-clé.

L'avantage principal de SPARK est d'utiliser l'approche orientée objet qui assure l'extensibilité et la réutilisabilité des composants.

La figure 4.6 présente une partie de notre hiérarchie actuelle dédiée à l'analyse syntaxique. Il s'agit de la partie correspondant à notre modèle à base de vues.

Il existe en fait trois hiérarchies parallèles dans ce paquetage dédié à l'analyse syntaxique : une pour l'analyse lexicale, une pour l'analyse syntaxique proprement dite et la dernière pour les interpréteurs. Les interpréteurs sont utilisés pour transformer une instance d'ASA créée par un analyseur syntaxique en une instance d'une classe de CLIS. Ces instances servent ensuite en tant que format interne aux spécifications dans ASK.

4.4.2 Opérations pour systèmes de transitions

Les systèmes de transition sont au cœur de notre formalisme et de nombreux autres (chapitre 1). De plus, nous utilisons des STS et puisque la définition de la sémantique des vues utilise une forme de produit synchronisé sur ces STS, alors nous avons décidé d'implémenter une forme très générale de ce produit dans ASK. Elle est applicable à de nombreux types de systèmes à base d'états et de transitions.

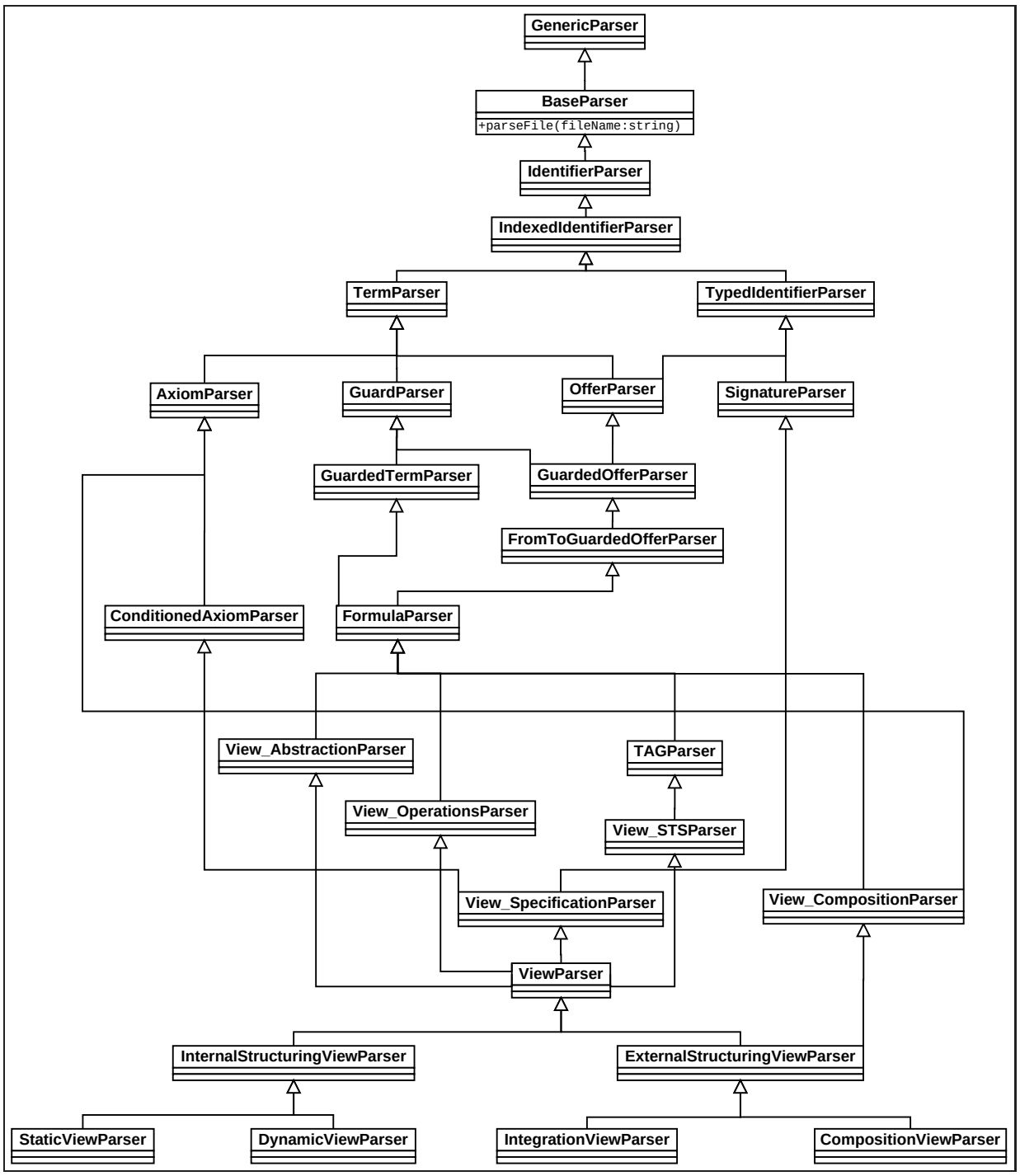


Figure 4.6 : Diagramme de classes - Analyse syntaxique dans ASK (partie)

Il faut noter que le produit de deux STS simples (non structuré, non indexé) n'est pas un STS simple, comme nous avons pu le voir par exemple dans le cas du STS global du gestionnaire de mots de passe (figure 2.37). C'est la raison pour laquelle, une grande partie de la hiérarchie CLAP (figure 4.4) sert à gérer la structuration (indexation, identifieurs de type "range") de nos STS. Ceci passe par la définition d'identifieurs, d'états et de transitions structurés.

Un certain nombre de fonctionnalités sont associées à cette hiérarchie. CLAP permet de définir des formules de logique temporelle pour calculer les états initiaux, les états et transitions atteignables depuis les états initiaux. Ces formules sont aussi utilisées comme paramètres au produit synchronisé. C'est ce produit qui constitue la fonctionnalité la plus intéressante de CLAP. Elle est suffisamment générale pour permettre de calculer le produit de n'importe quel nombre de systèmes de transition, en choisissant les règles (ensembles de formules temporelles) et le mode (`LOOSE`, `ALONE`, `KEEP`) de synchronisation. Pour cela le produit se base sur les règles de KORRIGAN. Ce produit permet par exemple de simuler à la fois la synchronisation LOTOS ou CCS en jouant sur les différents paramètres.

Un autre outil est associé à la librairie CLAP : l'outil de G-dérivation. Cet outil suit les principes définis dans l'étape 3.3.4 du chapitre 3. Il adapte l'idée consistant à écrire les axiomes d'une spécification algébrique de façon constructive à partir d'un ensemble donné (constructeurs) du type d'intérêt dans le contexte des STS. Comme nous l'avons dit plus haut, cet outil est très utile puisqu'il fournit une aide non négligeable à un utilisateur non spécialiste et assure que la spécification algébrique obtenue sera compatible avec le STS.

Les points principaux implémentés sont :
- extraction automatique de la signature ;
- choix des constructeurs par parcours du STS ;
- construction des parties gauches et conditions des axiomes ;
- saisie interactive des parties droites.

Pour l'instant cet outil est implémenté en Smalltalk [12], il est prévu de le traduire en PYTHON et de l'étendre aux STS structurés.

4.4.3 Traduction, validation et vérification des spécifications

Il est possible de générer des fichiers servant d'entrée à des outils de validation et vérification à partir des spécification KORRIGAN (parmi d'autres). Ce processus utilise le principe de traduction décrit dans la figure 4.2.

Il est par exemple possible de générer des spécification LOTOS à partir de spécifications KORRIGAN. Ces spécifications peuvent ensuite être vérifiées à l'aide de l'environnement CADP [110]. Ce processus utilise des patrons de traduction pour les parties dynamiques. La traduction des parties données est relativement directe puisque les formalismes algébriques utilisés en KORRIGAN et en LOTOS sont très proches. Le principe général appliqué à LOTOS est décrit dans la figure 4.7. Les détails de cette approche, qui a été appliquée avec succès à LOTOS et SDL font respectivement l'objet des sections 4.5.1 et 4.5.2.

On part d'un fichier contenant une vue KORRIGAN. Le mécanisme d'analyse syntaxique construit une instance de la classe réifiant les vues. Cette instance est constituée d'une instance de partie statique (une spécification algébrique) et d'une instance de partie dynamique (un STS). La spécification algébrique est transformée en instance de la classe réifiant ACT ONE, de façon triviale (mais toujours en suivant le mécanisme de la figure 4.2). Le STS est transformé en instance de la classe réifiant Basic LOTOS (section 4.5.1). Ces deux instances construisent une instance de la classe réifiant LOTOS, instance

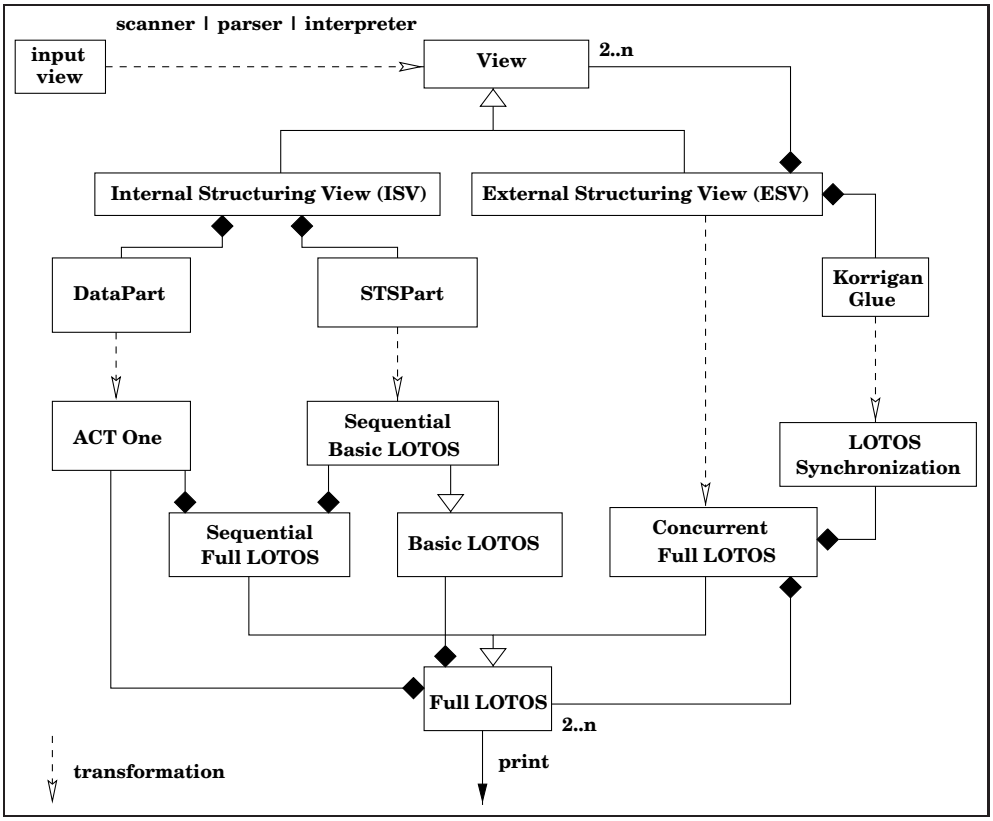


Figure 4.7 : ASK– Processus de traduction KORRIGAN en LOTOS

qu'il est possible d'imprimer pour obtenir une spécification LOTOS.

De façon générale, ces processus de traduction peuvent nécessiter une interaction et ne sont pas totalement automatiques. Ces interactions sont pour l'instant réduites au minimum puisque ASK n'a pas encore d'interface utilisateur graphique.

Une autre idée (non encore implémentée) serait de traduire les spécification KORRIGAN dans un formalisme où statique et dynamique sont exprimés dans un cadre algébrique (approche homogène comme par exemple CASL-LTL [232] ou PVS [6]) puis de vérifier la spécification résultante à l'aide d'un prouveur de théorème.

Il est prévu de définir et implémenter dans ASK des moyens spécifiques de vérification des spécifications KORRIGAN. Les spécifications mixtes à base de STS requièrent de moyens de vérification symboliques spécifiques [228, 166]. Les outils de vérification de modèle actuels ne sont pas bien adaptés à nos STS. C'est pourquoi, il est important de définir de nouveaux outils ou d'en adapter des anciens afin de vérifier ces STS.

4.4.4 Génération de code concurrent orienté objet

Des mécanismes de génération de code orienté objet à partir de spécification KORRIGAN ont été étudiés et implémentés dans ASK. Ces mécanismes sont détaillés dans la section 4.6. Il faut noter que ces mécanismes sont pour l'instant adaptés à une forme restreinte de spécifications KORRIGAN où les expressions de synchronisation (la "colle" des vues) sont données en termes de synchronisations "à la LOTOS" (tout leur pouvoir d'expressivité n'est donc pas utilisé). L'étude des mécanismes de génération de code sur KORRIGAN complet passe par la définition de schémas de contrôleurs (voir la section 4.6) adaptés à l'ensemble nos règles de logique temporelle.

4.5 Traduction des spécifications en LOTOS et SDL

Comme nous l'avons expliqué plus haut, il est possible de traduire les spécifications KORRIGAN vers divers langages de spécification en leur appliquant des schémas de traduction. Cette technique a été appliquée à LOTOS et à SDL et nous prévoyons de l'utiliser pour générer des spécifications en PVS suivant la méthodologie proposée par [6]. Le but de cette démarche est de pouvoir effectuer des vérifications et validations sur nos spécifications (et ce en l'absence de mécanismes dédiés de preuve). Cela permet aussi d'utiliser des outils dédiés à ces langages.

En préalable à l'explication des patrons de traductions utilisés pour LOTOS et SDL, il est nécessaire de parler des différences les plus importantes entre LOTOS et SDL, c'est-à-dire celles qui concernent la communication. En ce qui concerne les types de données, les deux approches sont connexes et notre démarche consistant à dériver une spécification constructive à partir des systèmes de transitions est utilisée.

En LOTOS, les communications sont en théorie proposées à tout l'environnement du processus émetteur (proche du `via all` de SDL), mais il est possible d'utiliser l'émission de valeurs particulières pour remédier à cette impossibilité (4^{e} ligne de la table 4.1). En SDL, il est possible de communiquer directement entre processus particuliers de façon naturelle. Pour cela, chaque processus dispose à sa création d'un identifiant particulier (type de données `PId`). Un processus peut ensuite s'adresser à un processus donné en utilisant des mots-clé comme `sender` (le dernier processus à lui avoir envoyé un signal), `parent` (le processus qui l'a créé), ...

En SDL, contrairement à LOTOS, les processus ne se synchronisent pas. L'émission est non bloquante. La réception passe par un buffer où sont placés les messages (signaux) dans l'ordre de leur arrivée (figure 1.7). Lorsqu'un processus est en attente, le premier signal du buffer est dépilé. S'il ne correspond à aucun des signaux qu'attendait le receveur, alors ce signal est éliminé (sauf cas particulier où le receveur peut explicitement demander à ce qu'il soit conservé). Si aucun signal attendu n'est dans le buffer, le receveur se bloque.

Nous pensons toutefois que les communications en LOTOS sont plus générales, c'est pourquoi nous utilisons une notation qui lui est empruntée (signature dynamique), mais qui tient aussi de SDL en ce qui concerne les informations sur émetteur et receveurs. Une fois les différences prises en compte, il est possible d'appliquer une série de schémas de traduction aux automates construits à l'aide de notre méthode (chapitre 3).

Dans la suite, nous parlons des spécifications des composants de base (parties comportement des vues). Les parties données sont créées en suivant l'approche constructive décrite dans le chapitre 3. La composition des composants de base (External Structuring Views) a été discutée brièvement dans ce même chapitre. Il s'agit de composer les spécifications de bases en utilisant respectivement la structure de bloc de SDL ou les schémas de composition de LOTOS en lieu et place des règles de colle des ESV.

4.5.1 Génération des spécifications LOTOS

Chacune des parties dynamiques est modélisée par un processus LOTOS. Nous indiquons aussi la fonctionnalité des processus. Il s'agit ici de processus ne terminant pas ; ils ont donc la fonctionnalité $noexit$ et sont modélisés par des processus récursifs. Chaque processus dispose de données internes (partie type de données des vues) et d'une partie contrôle (partie comportement). Un type est créé pour chaque type de processus et un objet de ce type constitue le paramètre formel du processus (noté `p:tp`). Ce type devra disposer d'une opération d'initialisation utilisée à l'instanciation du processus. Nous l'appelerons `init`. Lors de l'instanciation des processus, un appel à l'opération `init` est substitué au paramètre formel des processus.
Enfin, les interactions entre composants sont modélisées par des portes formelles. On peut voir cela comme les services (interface) du composant.

Nous donnons maintenant notre méthode semi-automatique d'obtention des spécifications LOTOS à partir de l'automate. Il est possible de procéder de deux façons (voir les schémas de transformation des figures 4.8 et 4.9) : un processus est associé à chaque automate et (i) pour chaque état une branche conditionnelle est créée puis des simplifications sont effectuées afin d'obtenir une spécification plus lisible, ou (ii) un sous-processus est créé pour chaque état de l'automate (cette dernière approche nécessitant/autorisant beaucoup moins de simplifications).

Signatures des opérations.

Il est possible ici d'extraire automatiquement les signatures des opérations correspondant aux communications sur les portes en fonction du typage de ces portes :

– pour toute porte de O ou OC,
`nomPorte` :$!X_{s_1}$:T_{s_1} ... $!X_{s_n}$:T_{s_n} $?X_{r_1}$:T_{r_1} ... $?X_{r_m}$:T_{r_m}
devient une famille d'opérations

`nomOper`$_i$: $T \times T_{r_1} \times ... \times T_{r_m} \rightarrow T_{s_i}, i \in \mathbb{N}, 1 \leq i \leq n$
où T est le nom du type associé au processus.
– pour toute porte de C ou CC,
`nomPorte` : $!X_{s_1}{:}T_{s_1} ... !X_{s_n}{:}T_{s_n}\ ?X_{r_1}{:}T_{r_1} ... ?X_{r_m}{:}T_{r_m}$
devient une famille d'opérations
`nomOper`$_i$: $T \times T_{r_1} \times ... \times T_{r_m} \rightarrow T_{s_i}, i \in \mathbb{N}, 1 \leq i \leq n$
et une opération
`nomOper`$_\omega$: $T \times T_{r_1} \times ... \times T_{r_m} \rightarrow T$

Dans le cas d'opérations ayant plus d'un type résultat, différentes nouvelles opérations sont créées de façon à ce que chacune ne retourne qu'une seule valeur.

Nous faisons la distinction entre les *opérations internes* (le type résultat est le type d'intérêt) et les *opérations externes* (le type résultat n'est pas le type d'intérêt). Les opérations internes pour lesquelles le type d'intérêt n'apparaît pas dans les arguments sont qualifiées d'*opérations internes de base*. Les opérations sont *totales* lorsqu'elles s'appliquent dans tous les états et *partielles* dans le cas contraire.

Nous appelerons β l'application faisant correspondre à une porte P la (ou les) opération(s) (signature) associée(s). De plus, pour chaque état de l'automate, ainsi que pour chacune des conditions (conditions d'état et conditions de transitions), une opération externe "observateur d'état" est créée avec le profil : `oper` : $T \rightarrow Bool$.

Création par simplification des expressions logiques.

Les éléments de O sont codés selon le schéma suivant :

`o?x`$_{r_j}$`:T`$_{r_j}$ `!x`$_{s_i}$ `; PROCESSNAME [portes] (paramètres)`

À priori, la valeur des x_{s_i} est fonction des données du processus et des valeurs reçues : $\beta_{o_i}(o)(p, \{x_{r_j}\})$. Les listes de portes et de paramètres sont celles de l'appel.

Les éléments de OC sont codés selon le schéma suivant :

`[c-cond(p)]` $\rightarrow$ `c?x`$_{r_j}$`:T`$_{r_j}$ `!x`$_{s_i}$ `; PROCESSNAME [portes] (paramètres)`

`c-cond` correspond à la condition correspondant à l'élément en question. À priori, la valeur des x_{s_i} est fonction des données du processus et des valeurs reçues : $\beta_{o_i}(c)(p, \{x_{r_j}\})$. Les listes de portes et de paramètres sont celles de l'appel.

On crée une branche conditionnelle correspondant à chaque état. La précondition associée est la condition d'état du processus. Ces conditions peuvent être trouvées dans la table des cas ayant servi à trouver les états.

En ce qui concerne les éléments de C et CC, des branches conditionnelles sont utilisées pour chacune des transitions partant des états. Le principe est décrit dans la figure 4.8. g(x) représente une communication sur la porte g avec une éventuelle réception de valeurs (x_{r_j}) et une éventuelle émission de valeurs ($\beta_{o_i}(g)(p, x_{r_j})$). f(p,x) (avec f= $\beta_\omega(g)$) est fonction des données internes du processus et des données éventuellement reçues via la porte g). σ désigne l'ensemble des portes du processus.

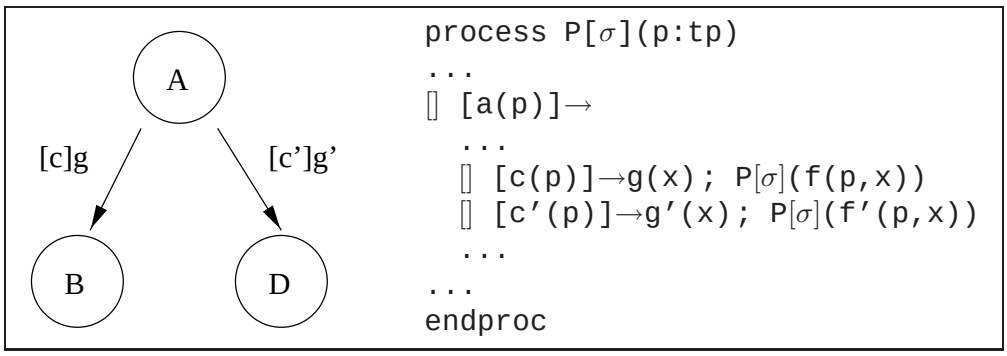


Figure 4.8 : Traduction LOTOS – Spécification par simplification pour les éléments de C et CC

Enfin, il y a appel récursif du processus dans le cas des états qui ne sont pas des puits (un puits correspondant à la terminaison avec blocage – `stop` – du processus : aucune transition ne peut être faite).

Il est possible de regrouper les conditions des transitions étiquetées par une même porte.

La spécification obtenue est “orientée état” : elle est construite autour de tests de l'état abstrait (courant) du processus, la liste des communications autorisées dépendant de cet état.

Ici, un regroupement plus poussé peut être fait sur les portes. On aurait pu obtenir une spécification plus concise en travaillant directement sur des automates séparés, un par condition, et pas sur l'automate les intégrant toutes.

Il est possible de simplifier encore la spécification obtenue en travaillant sur les conditions : l'emploi par exemple d'un outil de simplification (et de preuve) d'expressions logiques permet d'obtenir une expression plus concise des fonctions booléennes de la spécification.

Cette simplification conduit en fait à une spécification équivalente à celle qui aurait été obtenue en travaillant directement sur les conditions d'application des portes et en traitant - au niveau de la création de la spécification - les éléments de CC comme des éléments de OC.

Le fait de passer par un automate permet toutefois de mieux appréhender le comportement interne des processus.

Création par sous-processus.

Une autre possibilité pour obtenir les spécifications est d'ajouter un paramètre au processus représentant son état actuel. C'est cet état qui sert de précondition aux branches conditionnelles. Il est possible de représenter cet état par un tuple de conditions, ce qui fournit une sorte de codage binaire des états. Dans le cas d'une transition d'un état e vers f, nous avons une branche testant e et un appel récursif remplaçant e par f.

En allant plus loin, il est possible de créer un sous-processus par état. [263] explique que ce style permet une implémentation rapide, efficace et précise quand la spécification est traduite dans un langage séquentiel et donne des règles de transformation automatique de l'automate en LOTOS. [126] fait cependant remarquer que cela conduit à obtenir des spécifications non structurées.

Nous utilisons les règles suivantes :

– le processus principal appelle le processus correspondant à l'état initial
– pour toute transition [c] p allant d'un état A à un état B, nous créons une branche conditionnelle correspondante dans le processus A (voir figure 4.9). Ceci est à comparer avec l'approche de la figure 4.8.

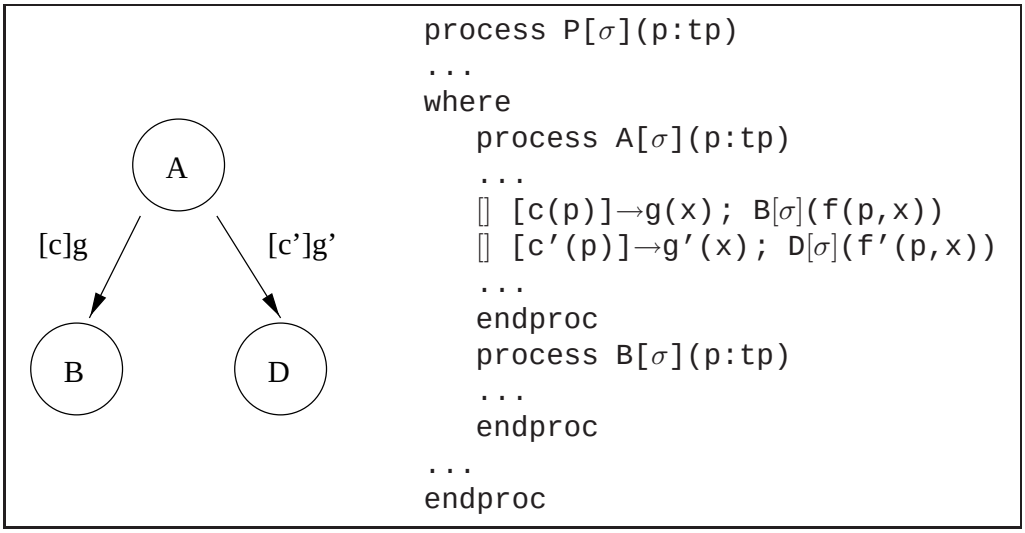


Figure 4.9 : Traduction LOTOS – Spécification en sous-processus

Le résultat est un processus éclaté en autant de sous-processus que l'automate contient d'états.

4.5.2 Génération des spécifications SDL

Tout d'abord, à un automate correspond un processus SDL. Le cadre du processus est construit (figure 4.10) et comprend : la déclaration du processus (`Process P`), la déclaration du paramètre formel correspondant à son type propre (`fpar paramP tP`) et enfin la déclaration de son type propre (`dcl a_tP tP`). Le type `tP` devra disposer d'une opération `init : S`$_i$ `-> tP`.

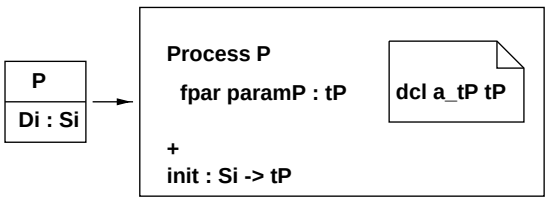
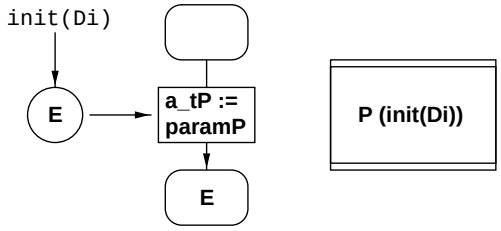


Figure 4.10 : Traduction SDL – Cadre des processus

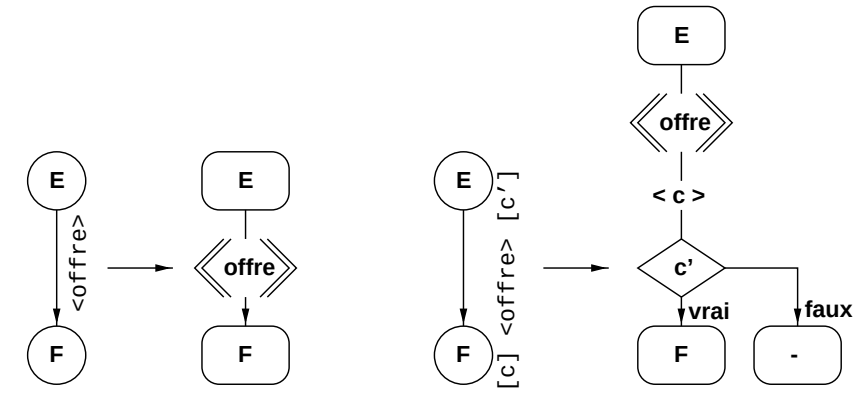
La transition initiale est construite en utilisant le schéma de la figure 4.11a. L'instanciation se fait comme donné dans la figure 4.11b, c'est-à-dire en instanciant le processus à l'aide de l'opération `init`.

La traduction des transitions non conditionnées se fait selon le schéma de la figure 4.12a. La prise en compte des conditions au niveau de la traduction se fait en utilisant des conditions d'acceptation et des symboles de décision (figure 4.12b). Les offres correspondant aux transitions sont traduites en utilisant les schémas des figures suivantes.



(a) transition initiale *(b) instance du processus*

Figure 4.11 : Traduction SDL – Transition initiale



(a) transition non conditionnée *(b) transition conditionnée*

Figure 4.12 : Traduction SDL – Traduction des transitions

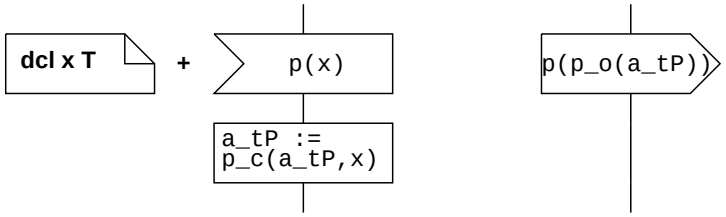
Quatre type d'offres existent (comme en LOTOS) et sont indiquées dans la table 4.1. Les offres multiples sont découpées en offres simples.

processus 1	processus 2	interprétation par rapport au processus 1
p ?x:T	p !x:T	réception de valeur
p !x:T	p ?x:T	émission de valeur
p ?x:T	p ?x:T	partage d'une valeur prise "au hasard" dans T (indéterminisme)
p !x:T	p !x:T	synchronisation sur valeur particulière (permet en LOTOS de s'adresser à un processus particulier)

Table 4.1 : Traduction SDL – Types d'offres

La dernière catégorie correspond à un cas particulier en LOTOS et se traduit en SDL par l'emploi du mot-clé `to`.

Le traduction de la première catégorie d'offre est donnée dans la figure 4.13a, et celle de la seconde catégorie (symétrique) dans la figure 4.13b. Il faut noter que dans le cas d'une réception il est nécessaire de déclarer une variable du bon type. Ce n'est pas le cas des émissions car les données émises sont en fait des fonctions sur le type paramètre du processus (`a_tP`).



(a) réception (p ?x:T) *(b) émission (p !x:T)*

Figure 4.13 : Traduction SDL – Traduction des offres d'émission/réception

Pour pouvoir traduire la troisième catégorie d'offre, il est nécessaire de lui attribuer un "sens", c'est-à-dire de choisir un processus qui calculera la valeur `x` à l'aide du mot-clé `any` et qui sera chargé de l'émettre au second processus (figure 4.14).

La traduction peut ne pas être optimale et c'est pourquoi des simplifications doivent éventuellement lui être appliquées.

En ce qui concerne SDL, l'utilisation des listes d'états ou de l'étoile dans les états de début des transitions ainsi que du tiret dans les états de fin des transitions permet de structurer la description des processus par rapport aux transitions (signaux reçus) et non pas des états. Cette technique peut par exemple être appliquée avec succès aux membres de O (figure 4.15) : ces communications étant possibles depuis tous les états, le regroupement par signal permet de regrouper toutes les transitions `o` en une seule transition SDL.

Lors de la traduction des offres, nous avons découpé les offres multiples en offres simples. Du fait du typage des signaux en SDL, il faut créer autant de signaux que d'offres simples. Il est en fait possible de regrouper des séries d'émissions (resp. réceptions) les unes à la suite des autres et de ne modifier le type paramètre du processus qu'à la fin des réceptions. Cette modification peut même être omise si

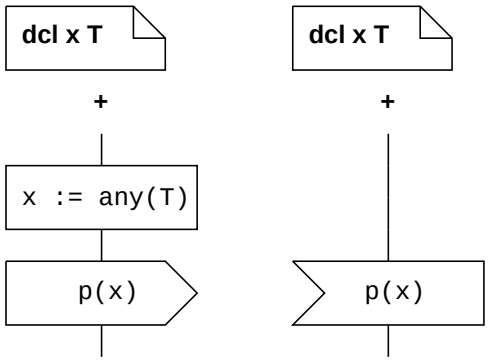


Figure 4.14 : Traduction SDL – Traduction des offres ?x:T / ?x:T.

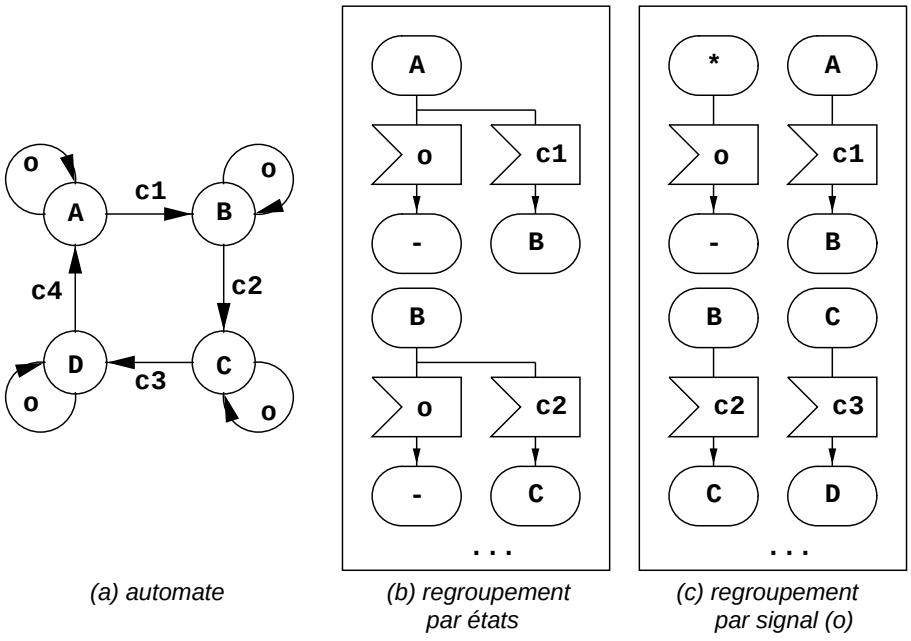


Figure 4.15 : Traduction SDL – Comparaison entre regroupements

elle n'est pas pertinente (comme par exemple un processus qui récupererait deux nombres et émettrait le minimum, figure 4.16).

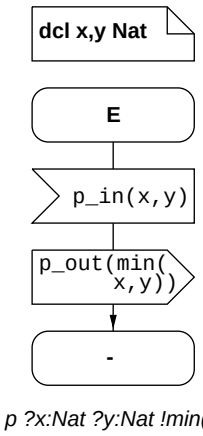


Figure 4.16 : Traduction SDL – Calcul de minimum

L'application de ce type de simplification à l'offre `p?x:T?y:T!z:U` (où `z` dépendrait des valeurs de `x`, `y` et du type paramètre) est donné dans la figure 4.17a. À des fins de comparaison, ce que nous aurions obtenu sans simplifier est donné dans la figure 4.17b.

Il faut remarquer qu'il est aussi possible de simplifier l'automate avant traduction (chapitre 3).

La figure 4.18 donne une partie du processus SDL correspondant au DPI.

4.6 Génération de code concurrent orienté objet en ActiveJAVA

La spécification formelle obtenue n'est pas nécessairement exécutable. La partie dynamique est souvent exécutable car elle est basée sur des modèles opérationnels (diagrammes états-transitions). Mais la partie statique (types abstraits algébriques) n'est pas toujours exécutable.

Nous allons montrer notre génération de code pour le langage Java, mais cette méthode convient pour d'autres langages orientés objet.

La méthode générale est présentée dans la figure 4.19 et elle comporte deux parties : la partie dynamique (à droite sur la figure 4.19) et la partie statique (à gauche sur la figure 4.19). Il faut noter que nous visons ActiveJAVA comme langage cible pour la partie dynamique. En effet, coder les mécanismes de communication (sockets, ...) serait faisable en Java, mais pour des raisons de transparence nous préférons passer par ActiveJAVA qui gère cela pour nous.

4.6.1 Génération de la partie statique

Des classes Java sont engendrées pour chaque type abstrait de la spécification, et ce processus est réalisé par quatre étapes intermédiaires (cf. figure 4.19). La traduction est partiellement automatique, car une interaction avec un spécifieur (ou un programmeur) peut permettre d'obtenir un résultat plus simple ou plus efficace. La première étape consiste à obtenir une spécification exécutable (éventuellement via des raffinements). La génération de code consiste ensuite en (i) le choix d'une hiérarchie pour représenter les générateurs de la spécification, (ii) la traduction en classes formelles [13] (i.e. des abstractions de

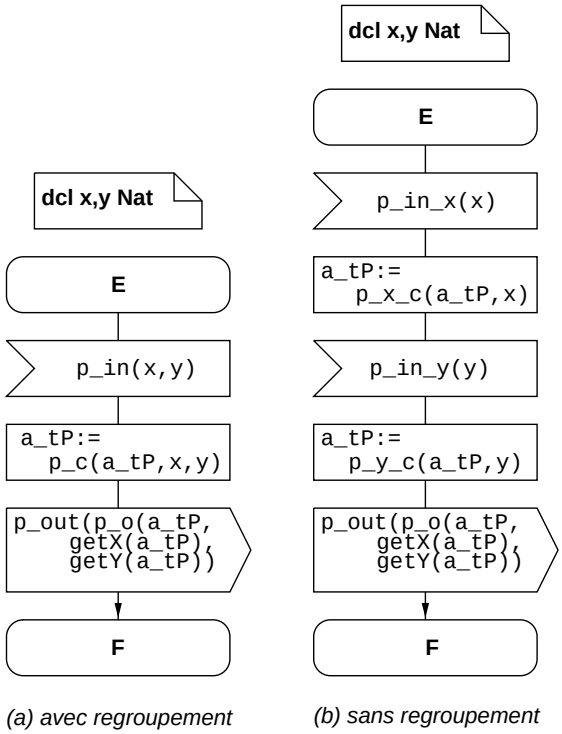


Figure 4.17 : Traduction SDL – Regroupement d'offres simples

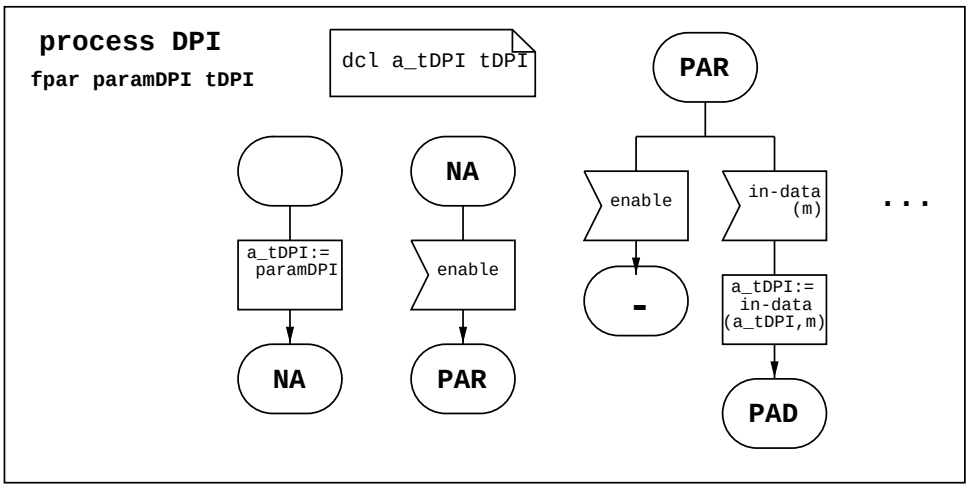


Figure 4.18 : Traduction SDL – Processus DPI (partie)

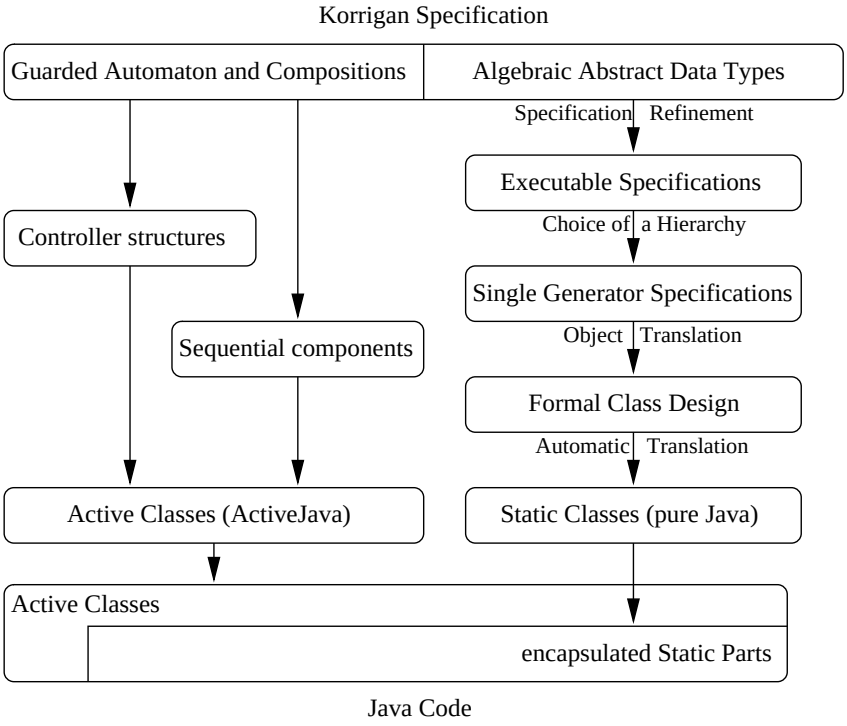


Figure 4.19 : Schéma de génération de code orienté objet

classes en langages orientés objet), à partir desquelles (iii) la génération dans un langage choisi (par exemple Java) peut être effectuée.

4.6.1.1 Spécifications exécutables

Le style “constructif” adopté pour les spécifications associées aux automates conduit généralement à des spécifications exécutables (par exemple par réécriture, et des outils tels que [127] peuvent être utilisés pour vérifier la convergence). Toutefois, d’autres modules de spécification peuvent être introduits (par exemple, pour les données gérées par les processus) avec d’autres styles de spécification (par exemple en style observationnel). Un processus de raffinement (implémentation abstraite [107]) est alors nécessaire pour ajouter des éléments d’exécutabilité tels que des choix algorithmiques, etc.

4.6.1.2 Spécifications à générateur unique

Dans les langages orientés objet, les classes ont une opération unique de génération appelée par exemple “new” (ou le nom de la classe), alors que les spécifications algébriques admettent plusieurs générateurs. Le problème ici est comment représenter ces générateurs au sein des classes, ou, plus précisément, comment transformer (par exemple par implémentation abstraite) les spécifications algébriques de départ en spécifications à générateur unique à partir desquelles les classes peuvent être dérivées. Nous proposons plusieurs solutions à cette question.

Une première solution est d’associer aux différents générateurs de la spécification algébrique un générateur unique muni d’un “aiguillage” (vers chacun des générateurs de départ) ; nous appelerons cette solution “l’organisation plate”. Une autre solution est d’utiliser une classe associée comme interface vers

des sous-classes, et chaque sous-classe est associée à un générateur de la spécification de départ ; nous appelerons cette solution “organisation hiérarchique à deux niveaux”. Bien sûr il est également possible de composer ces deux solutions.

Plusieurs paradigmes ont été développés et sont disponibles pour l'implémentation abstraite [107], une première approche est de suivre la représentation abstraite de Hoare [150]. Elle consiste à définir une fonction d'abstraction, à montrer que c'est une fonction surjective et à démontrer l'implémentation des opérations. Nous ne détaillons pas ces aspects ici.

Dans ce qui suit, nous présentons les organisations alternatives pour les spécifications à générateur unique. Lorsque le type abstrait a un seul générateur nous appliquons directement la représentation simple décrite ci-dessous pour obtenir une classe.

Organisation plate. Dans cette organisation, un module de spécification avec plusieurs générateurs est transformé en module de spécification à générateur unique avec un aiguillage vers chaque générateur de départ. Par exemple, dans le module de spécification `DPI`, les générateurs sont `init, enable, in_data, ask_route, reply_route`.

Nous définissons `SwitchDPI = {init, enable, in_data, ask_route, reply_route}` et le générateur unique `newSDPI` (SDPI signifie Switch DPI) avec pour profil `newSDPI : Switch PortNumber Msg RouteNumber List SDPI -> SDPI` (notons que ce profil peut être aisément calculé à partir des profils des générateurs de DPI). La fonction d'abstraction `Abs` est définie comme habituellement :

```
Abs(newSDPI(reply_route, Bport, Bmsg, Broute, Z, T)) == reply_route_c(T, Z)...
```

Les valeurs des termes qui commencent par `B` ne sont pas pertinentes. Nous introduisons aussi des sélecteurs associés aux arguments pertinents qui apparaissent dans le générateur unique :

```
switch(newSDPI(S, X, Y, R, Z, T)) = S
(S = reply_route ∧ WA(T)) => selRoutes(newSDPI(S, X, Y, R, Z, T)) = Z...
```

Les axiomes sont ensuite transformés dans ce cadre afin de compléter la spécification.

Organisation en hiérarchie à deux niveaux. Dans cette approche, plusieurs modules de spécification sont associés au module de spécification de départ : un module qui est uniquement une interface vers les autres modules qui introduisent chacun un des générateurs de départ avec la sous-sorte appropriée. Clairement, cette approche peut impliquer des questions sémantiques (selon le cadre adopté), et peut ne pas être aussi pratique et aussi directe que la précédente. Toutefois, dans certains cas le style de spécification peut être plus lisible.

Organisation Mixte. Bien sûr, entre les deux solutions extrêmes précédentes, il existe plusieurs autres manières, selon le choix de hiérarchie. [11] montre comment trouver une bonne hiérarchie et un processus général pour l'obtenir. Toutefois, certains problèmes importants demeurent : celui de trouver une métrique pour évaluer la meilleure hiérarchie et des problèmes liés à l'héritage des propriétés.

Dans le cas de types abstraits avec moins de cinq générateurs, l'organisation plate est acceptable, mais pas avec des types plus complexes.

Une autre manière de résoudre ce problème est d'introduire une sorte d'héritage ou de sous-sorte ("à la OBJ") dans cette méthode. Il est connu que ce problème est difficile et complexe en présence de systèmes concurrents.

4.6.1.3 Conception en classes formelles : le modèle

Ce modèle [13] définit la notion de *classe formelle* comme une abstraction de classe concrète de langages tels que C++, Eiffel, Java ou Smalltalk. Une classe formelle est une spécification algébrique avec une orientation objet. Ce modèle général est fonctionnel et unifie les concepts majeurs de la programmation orientée objet. Il peut être utilisé à la fois pour construire des spécifications formelles et pour concevoir un système. Une sémantique opérationnelle abstraite [13] est donnée à ce modèle en utilisant la réécriture conditionnelle [93].

La figure 4.20 montre un exemple de classe formelle associée au module de spécification du SDPI et obtenue avec l'organisation plate.

FCDPI

field selectors

```
switch     : FCDPI ⟶ Switch
ident      : FCDPI ⟶ PortNumber
   requires: switch(Self) = init
selRoutes  : FCDPI ⟶ List
   requires: switch(Self) = reply_route ∧ WA(previous(Self))
   ...
```

methods

```
correct_c : FCDPI ⟶ FCDPI
;; correct_c : internal operation associated to a correct route
      (switch(Self) = enable ∧ CR(previous(Self))) =>
      correct_c(Self) = correct_c(previous(Self))
      (switch(Self) = reply_route ∧ switch(previous(Self)) = ask_route
       ∧ WA(previous(Self)) ∧ is_empty(list(Self))) =>
       correct_c(Self) = correct_c(new(reply_route, Bport, Bmsg, Broute,
                         selRoutes(Self), previous(previous(Self))))
      ...
```

Figure 4.20 : Formal Class FCDPI

La traduction en code Java (purement fonctionnel) est directe. Une classe formelle est traduite en une interface (correspondant à la signature) et en une classe d'implémentation. Nous utilisons des méthodes et des classes abstraites quand cela est nécessaire (selon l'organisation choisie). La structure est représentée par des variables d'instance privées. Les sélecteurs de champs sont codés par un accesseur public à la variable d'instance correspondante avec une condition correspondant à la précondition. Un outil pour engendrer les classes Java est prévu, et des outils expérimentaux pour engendrer des classes en Eiffel et en Smalltalk ont été réalisés.

4.6.1.4 Conception en classes formelles : une représentation simple

La représentation simple permet de traduire un type à générateur unique en une classe formelle, notée FCADT. Ce générateur sera la fonction d'instantiation newFCADT du modèle objet. Nous devons identifier les sélecteurs, c'est-à-dire les opérations sel_i telles que $sel_i(new(X_1, ..., X_n)) =$

X_i. Ces sélecteurs de champs correspondent aux variables d'instance de la classe. Nous supposons que la spécification (axiomes et préconditions) n'a pas de variable appelée `self`. Un terme est dit être en position receveur s'il apparaît en première position dans les arguments d'une opération qui n'est pas un générateur. Si une variable apparaît en position receveur dans le terme gauche de la conclusion alors elle sera remplacée par `self`. Dans notre modèle cette variable dénote l'objet receveur. Une règle importante de traduction est de remplacer `newSADT(`e_1`, ..., `e_n`)` par `V` avec `V : FCADT`. Cela conduit à un ensemble d'équations: sel_i`(V) =` e_i.

1. Cette règle est appliquée sur chaque occurrence de `newSADT` en position receveur dans le terme gauche de la conclusion, avec `V` renommé `self`. Si e_i est une variable alors elle est remplacée par sel_i`(Self)` dans les axiomes. Si e_i n'est ni une variable ni un terme neutre, l'équation sel_i`(Self) =` e_i est ajoutée aux conditions de l'axiome.
2. Cette règle est appliquée aux autres occurences du terme gauche de la conclusion pour toutes les variables qui ne sont pas `self`.

Cette représentation est effectuée sur la spécification à générateur unique SDPI et le résultat est la classe formelle de la figure 4.20.

4.6.2 Génération de la partie dynamique

Cette partie concerne la génération de code, relative à la partie dynamique, dans un langage orienté objet. Le langage que nous ciblons est ActiveJava [8], un dialecte de Java (transformé par compilation en Java pur) basé sur le modèle ATOM [217].

Le modèle utilisé par ActiveJava définit des états abstraits, des prédicats d'états, des conditions d'activation associées aux méthodes et des notifications d'état. Ses avantages principaux sont : (i) sa syntaxe est définie comme une extension de Java, (ii) il permet de modéliser à la fois de la concurrence inter et intra objet, et (iii) il supporte à la fois la communication synchrone et l'envoi (asynchrone) de messages. ActiveJava présente une bonne adéquation entre la réutilisabilité des composants (par le biais de la séparation des aspects) et l'expressivité du langage.

La génération de code pour la partie dynamique est basés sur (i) le codage de la structuration en sous-composants et une sémantique de communication "à la LOTOS", (ii) le codage des automates séquentiels, et (iii) l'intégration des parties dynamiques et statiques.

4.6.2.1 Structuration et sémantique de communication

La sémantique de communication "à la LOTOS" est plus stricte que celle (orientée objet) d'ActiveJava. Pour cette raison, et pour proposer une meilleure structuration des systèmes, pour choisissons de modéliser chaque structuration en sous-systèmes avec un *contrôleur*. Cette approche est proche de celle des Coordinated Roles [210] et vise à avoir les mêmes propriétés : une meilleure structuration et une meilleure réutilisation des patrons de composition.

La structuration du système est vue comme un arbre avec des composants séquentiels au niveau des feuilles et des contrôleurs au niveau des nœuds, dans lesquels les mécanismes de structuration de LOTOS sont codés. Les sous-arbres au niveau d'un nœud sont appélés ses *fils*. La structuration concernant le nœud de transit est par exemple présentée dans la figure 4.21, où les contrôleurs sont représentés par des rectangles et les composant séquentiels par des ellipses. Il s'agit d'une autre reprrésentation de la figure 3.14, mais qui correspond aux schémas de compositions présentés dans les figures 3.16 et 3.18 par exemple.

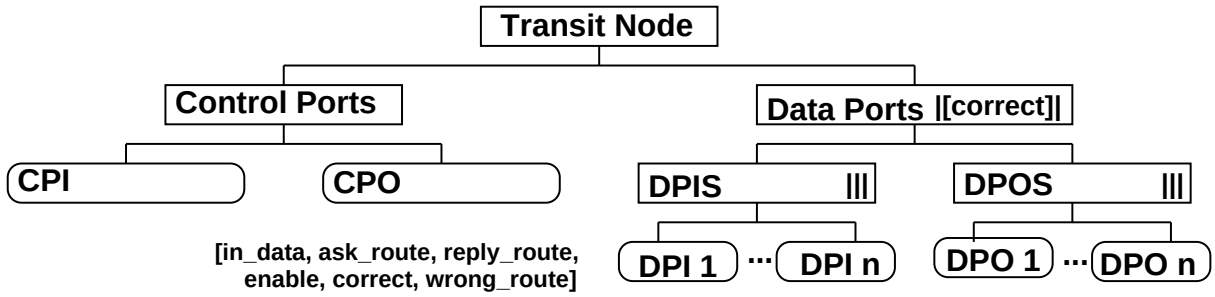


Figure 4.21 : Structuration et contrôleurs pour le nœud de transit

élément 1	élément 2	intersection commune
(“m”,_:T)	(“m”,valeur:T)	(“m”,valeur:T)
(“m”,valeur:T)	(“m”,_:T)	(“m”,valeur:T)
(“m”,valeur:T)	(“m”,valeur:T)	(“m”,valeur:T) – mêmes valeurs
(“m”,_:T)	(“m”,_:T)	(“m”,_:T)

Table 4.2 : Correspondance entre élément et intersections résultantes

Il existe trois principaux mécanismes de communication pour la structuration : l’entrelacement, la synchronisation et la synchronisation cachée.

Mécanismes communs. La communication se fait en trois phases comme indiqué dans la figure 4.22.

Dans la *phase d’exécution* les contrôleurs lancent des appels aux méthodes `run` de leurs fils qui ne sont pas en attente. Ainsi, dans la figure 4.22-1, C envoie un appel à la méthode `run` de P mais pas à celle de E. Lorsque ces appels arrivent à un composant qui n’est pas un contrôleur (*i.e.* les feuilles de l’arbre, comme P dans la figure 4.22-1), ou à un contrôleur dont tous les fils sont bloqués en attente, alors la seconde phase, la *phase de retour* peut commencer.

Dans cette phase de retour, les fils retournent à leur contrôleur l’information des communications qu’ils sont prêts à faire dans une *liste d’exécution* : une liste de tuples contenant le nom de la communication et ses arguments, avec des valeurs pour les arguments en émission et un indicateur particulier (_) pour les arguments en réception. Ici (figure 4.22-2) P retourne `[(“m”,_,4),(“n”,3)]` pour indiquer qu’il est prêt pour une communication sur `m` ou sur `n`. Le contrôleur calcule alors un *intersection commune* des listes d’exécution de ses fils (*i.e.* une liste d’exécution) et la renvoie plus haut, à son propre contrôleur. Ici `n` de P ne correspond à aucun communication avec E, tandis que deux éléments `m` correspondent pour faire une intersection que C peut envoyer à son contrôleur. Puisque E et P ont des éléments de leurs listes d’exécution qui sont dans l’intersection commune, E et P sont appelés *participants*. Les éléments avec un nom de communication commun doivent se correspondrent au sens où deux offres en LOTOS se correspondent. Les correspondances possibles sont données dans la table 4.2. Toutes les autres concernent des éléments ne se correspondant pas.

La seconde phase se termine lorsqu’il n’y a pas d’intersection possible (ceci correspondant à un état de blocage) ou au niveau de la racine où une *intersection finale* est calculée. Le contrôleur où la seconde phase se termine est appelé *racine temporaire*.

Dans la troisième phase, la *phase d’application* (figure 4.22-3), la racine temporaire envoie vers le bas, à ses fils (s’il s’agit de contrôleurs, il répercuteront ce message à leur tour à leurs fils), le message

correspondant à l'intersection finale qu'il a préalablement calculée. Ce message doit être unique, et le non déterminisme (soit qu'une valeur reçue n'a pas été liée, ou qu'il y ait un non déterminisme entre deux communications possibles) doit être résolu par le contrôleur racine temporaire. Les contrôleurs n'envoient le message qu'à leur fils participants (*i.e.* à la fois P et E pour C) puis détruisent l'entrée correspondant à ces fils dans leurs tables. Les fils qui ne sont pas participants sont laissés en attente. Pour finir, le contrôleur racine temporaire relance la première phase en faisant de nouveau appel à la méthode `run` de ses fils qui ne sont pas en attente.

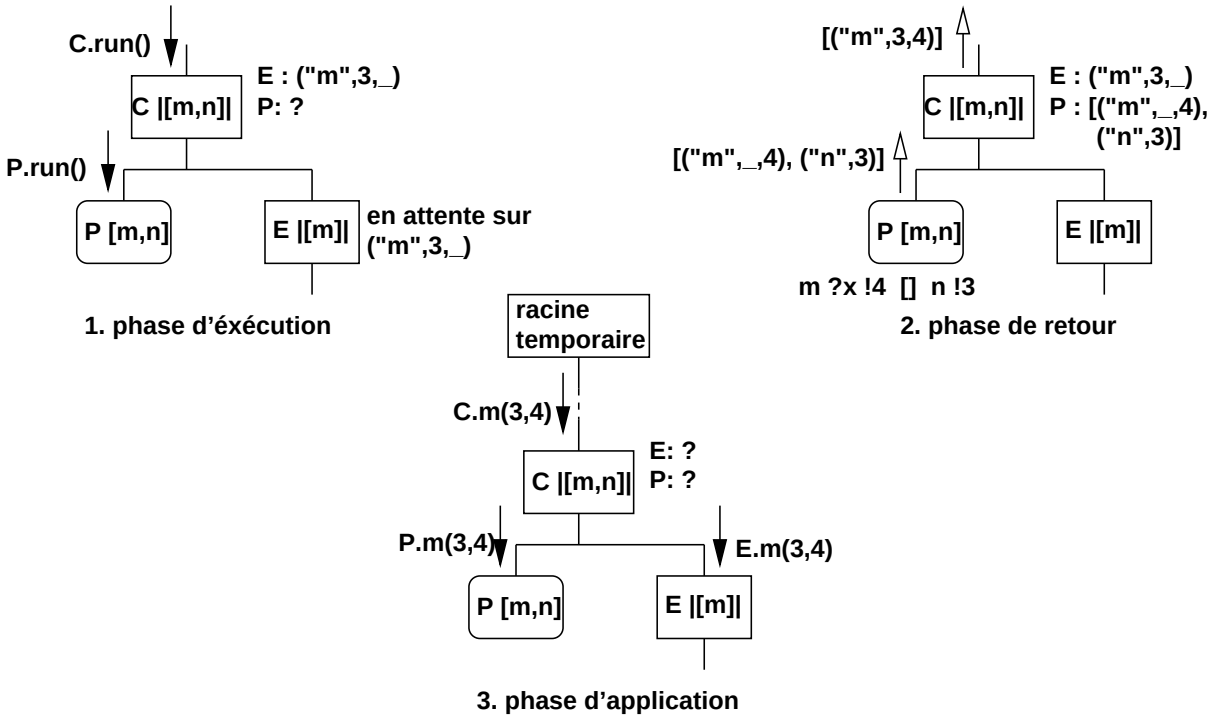


Figure 4.22 : Exemple de communication en 3 phases

Entrelacement. Aussitôt que le contrôleur reçoit un élément non synchronisé dans une liste d'exécution, il le transmet vers le haut.

Synchronisation. Lorsque deux fils sont composés de façon à se synchroniser sur une méthode particulière, leur parent contrôleur transmettra l'élément correspondant présent dans ses listes d'exécution uniquement s'il a reçu cet élément dans les listes des deux fils.

Synchronisation cachée. Lors de la phase de retour, et lorsque les listes d'exécution atteignent un nœud, les éléments correspondant à des messages cachés ne sont pas transmis vers le haut, mais sont conservés à ce niveau de nœud. Lorsque uniquement des messages cachés atteignent un contrôleur qui doit les bloquer, ce contrôleur agit comme contrôleur racine temporaire. S'il y a aussi des messages non cachés, le contrôleur choisit soit de les transmettre vers le haut, soit d'agir comme contrôleur racine temporaire (ce choix sert à simuler de l'indéterminisme).

4.6.2.2 Codage de l'automate en ActiveJava

Les tables de préconditions et de postconditions sont utilisées pour coder l'automate. Toutefois, ceci doit être légèrement modifié pour prendre en compte les réceptions de messages `run`. Le schéma de la figure 4.23 est appliqué à chaque état.

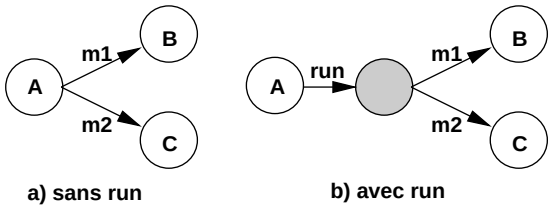


Figure 4.23 : Ajout de `run` dans l'automate

Les préconditions des opérations sont définies en tant que "conditions d'activation" en ActiveJava. Optionnellement, les variables conditionnelles peuvent être valuées dans les méthodes de type "post actions" (effectuées après les actions) d'ActiveJava. Dans le constructeur de la classe, des valeurs initiales sont données aux conditions en respectant l'état initial de l'automate. La figure 4.24 illustre ce principe sur une partie de l'exemple du DPI.

4.6.2.3 Intégration des parties statiques et dynamiques

L'intégration des parties statiques et dynamiques est effectuée en encaspulant une instance de la classe statique dans la classe dynamique, et en effectuant des appels aux méthodes statiques dans les corps des méthodes dynamiques (figure 4.25). Les observateurs sont appelés dans la méthode `run` de façon à calculer certains des arguments des éléments des listes d'exécution. Les méthodes statiques sont appelées dans chaque méthode dynamique leur correspondant.

4.7 Conclusions

Dans ce chapitre nous avons proposé un environnement [72] pour supporter l'utilisation de notre modèle et du langage KORRIGAN pour la spécification mixte.

Cet environnement suit deux principes : ouverture et extensibilité. En accord avec ces principes, il fournit des outils de traduction pour interfacer notre modèle avec d'autres formalismes, comme LOTOS, SDL, LP ou XFig.

Nous disposons d'une librairie support pour le travail avec les spécification mixtes, CLIS, ainsi qu'un outil dédié à la description des systèmes à base d'états et des transitions, CLAP. Cette librairie permet de calculer différentes compositions de ces systèmes. Elle implémente ainsi les mécanismes théoriques étudiés dans le chapitre 2 au niveau de la sémantique de notre modèle à base de vues. CLAP permet aussi de construire automatiquement la documentation (XFig) associée aux systèmes de transition.

L'atelier implémente des mécanismes de génération de code orienté objet [224] dans un dialecte Java concurrent à partir de nos spécifications mixtes. Ces mécanismes doivent être étendus pour prendre en

```
active class DPI(CPI cpi, FC fc, DPO dpo) {
    boolean v_a, v_r, v_d, v_rep, v_rerr;
    PortNumberList received_l; ...

abstract state definitions {a as is_a(); ...}

activations conditions {
    reply_route(RouteNumber r, PortNumberList l)
        with reply_route_precondition(); ...}

synchronization actions {
    reply_route(RouteNumber r, PortNumberList l) with
        post_actions reply_route_postcondition(); ...}

implement synchronization {
    boolean is_a() {return v_a;}
    ...
    boolean reply_route_precondition() {
        return v_a==TRUE && v_r==TRUE && v_d==TRUE &&
v_rep==FALSE;}
    ...
    void reply_route_postcondition() {
        v_rep=TRUE; v_rerr=received_l.isEmpty();}
    ...
}}
```

Figure 4.24 : Classe active pour le DPI (partie)

compte toute l'expressivité de notre colle à base de logique temporelle. Pour l'instant, ils sont efficaces quand la synchronisation des composants est donnée sous une forme proche de LOTOS.

L'environnement ASK est basé sur une classification de spécification et un mécanisme général de parsing. De nouveaux formalismes peuvent être intégrés, et des mécanismes de traduction définis pour eux. Il suffit pour cela de suivre un mécanisme général d'intégration que nous avons défini. La prise en compte de nouvelle possibilités de vérification des spécifications KORRIGAN, par traduction en PVS et en suivant l'approche de [6] sera aussi faite en suivant ce mécanisme général.

D'autres outils ont été expérimentés en Smalltalk ou en CLOS, il seront intégrés à l'environnement actuel.

Nous ne disposons pas encore d'une interface utilisateur. Sa conception est en cours. Elle intégrera la prise en compte de la méthode de spécification mixte que nous avons présentée dans le chapitre 3, ainsi que l'ensemble des outils expérimentés.

```
import StaticClass;

active class DynamicClass {
    StaticClass nested;
    < ActiveClass part >
    public methods {
        public DynamicClass( < arguments > ) {
            nested = new StaticClass( < arguments > );
            < dynamic initialization (initial state) >
        }
        public RunnableList run() {
        // uses nested.observers return values in run-
nable list
        }
        public void otherMethod( < arguments > ) {
            nested.otherMethod( < arguments > );
        }}}
```

Figure 4.25 : Intégration des parties dynamiques et statiques

Conclusion

Bilan

L'intérêt des méthodes de spécification formelle dans le cadre de la conception et du développement de systèmes dits critiques ou sécuritaires, est aujourd'hui indiscutable. Ces systèmes, et plus particulièrement les systèmes répartis tels que les protocoles et services de communication, présentent souvent plusieurs aspects à prendre en compte, comme les aspects statiques, dynamiques, et de composition. Le cadre de spécification correspondant à la prise en compte de l'ensemble des aspects de tels systèmes est celui des spécifications mixtes. La complexité de ces systèmes impose aussi la définition de moyens de structuration des spécifications, tant dans un but de lisibilité que de réutilisation.

De nombreuses propositions de formalismes pour la spécification mixte existent, centrées autour de deux approches : hétérogène et homogène. La première est bien adaptée à l'expression des différents aspects ainsi qu'à la réutilisation d'outils existants, mais pose des problèmes de cohérence entre aspects et de définition d'une sémantique globale complète. La seconde résout ces problèmes par l'emploi d'un unique formalisme, ce qui nuit à son expressivité et/ou à sa lisibilité, généralement au niveau de l'expression de la dynamique des systèmes et de leur communication. D'autre part, certains de ces formalismes (les plus formels) ne sont pas équipés de méthode associée, ce qui restreint leur utilisation dans le contexte industriel, au profit d'approches semi-formelles telles UML ou plus généralement celles posant des problèmes de cohérence entre aspects.

Pour répondre à cette problématique, nous avons proposé une approche qui, tout en gardant les avantages des langages dédiés à chacun des aspects (spécifications algébriques pour les aspects statiques, systèmes de transitions pour les aspects dynamiques) et en étant donc intrinsèquement hétérogène, fournit un cadre unificateur (et donc homogène) disposant d'un modèle sémantique approprié. Ce cadre est basé sur notre notion de *vue*. Cette notion utilise des spécifications algébriques, des systèmes de transitions symboliques et une logique temporelle simple. Les vues permettent d'assurer aux spécifications obtenues de bonnes propriétés d'abstraction (pas d'explosion du nombre d'états) et d'expressivité, tout en conservant un bon niveau de lisibilité (notations graphiques intuitives).

Pour répondre à la nécessité de structuration des spécifications de systèmes de taille réelle, nous avons proposé trois moyens de structuration : structuration interne, structuration externe et héritage. La structuration interne permet de définir les différents aspects des composants (dynamique et statique). La structuration externe permet de définir des composants globaux, c'est-à-dire dont l'ensemble des aspects internes est bien défini. Elle permet aussi de définir des compositions d'objets en se basant sur la communication entre ces objets. Enfin, une forme simple d'héritage permet de réutiliser les composants. Chacun des moyens de structuration est lui-même basé à son tour sur notre notion de vue. Nous avons défini un langage associé à ces spécifications mixtes et à leur structuration : KORRIGAN [70]. La sémantique des vues ainsi que des différents types de composition de vues a été étudiée [69].

Nous pensons aussi qu'il est important, pour que les méthodes formelles soient réellement utilisées, et particulièrement hors du contexte universitaire, qu'elles disposent d'une méthode associée. Dans cette optique, nous avons proposé une méthode semi-formelle pour la spécification mixte et structurée [223], applicable aux spécifications KORRIGAN, mais aussi à des langages comme LOTOS ou SDL. Cette

méthode dispose d'une notation inspirée par UML qui permet de définir le système de façon partiellement graphique puis de générer les spécifications résultantes dans tel ou tel langage de spécification [71]. Cette méthode a été appliquée sur plusieurs exemples.

Enfin, nous avons défini un atelier [72] pour la spécification mixte de systèmes. Cet atelier est implanté dans un cadre orienté objet. Il est ouvert et les moyens d'intégration de nouveaux langages et outils ont été définis. En ce qui concerne notre modèle, l'atelier prend en compte l'ensemble des mécanismes formels utilisé pour la sémantique opérationnelle (gestion des formules de logique temporelle, moteur de réécriture conditionnelle, calcul des systèmes de transitions globaux). L'atelier permet aussi la génération d'un prototype du système dans un langage concurrent orienté objet [224], Active Java.

Perspectives

Un premier point concerne la définition de moyens directs de preuve pour le langage KORRIGAN, entre autres tirant partie de l'aspect systèmes de transitions symboliques du formalisme. En effet, la validation et la vérification se font actuellement par traduction (via l'atelier ASK) vers des formalismes dédiés, comme LOTOS ou PVS. À la complexité inhérente à l'utilisation des systèmes de transitions symboliques [166, 243], se rajoute celle liée à l'existence des contraintes dans une partie des gardes de nos transitions, contraintes pouvant porter à la fois sur les données reçues ou des correspondances entre valeurs de variables provenant de plusieurs vues dans une composition. Nous pensons nous orienter sur une approche faisant collaborer vérification de modèle et solveur de contraintes, en se basant sur le fait que les spécifications algébriques utilisées sont constructives et fournissent par là même un mécanisme d'induction. Il semble pour cela possible de se baser sur les travaux concernant la vérification de modèle dans un cadre de solveur de contraintes [86, 219].

Un second point concerne la résolution de l'explosion d'états liée à la composition d'un nombre non fixé de composants au sein d'une composition. Actuellement, la solution adoptée passe par le calcul d'un automate global. Nous pensons qu'il est possible d'observer des correspondances entre composants communicants, ou bien encore d'abstraire en fonction de sous-groupes communicants. Ceci impose de modifier les gardes (nouveau type de contraintes concernant des ensembles de composants communicants) ainsi que les règles de colle, et donc le calcul du STS global et sa sémantique. Une piste semble être celle des systèmes de transitions symétriques [5].

Concernant la méthode et l'environnement, les perspectives consistent essentiellement à implanter la méthode dans des environnements pouvant fournir un cadre associé, comme Proplane [250], et définir des moyens d'intégrer ce type d'outil à notre environnement. Un autre point méthodologiquement intéressant concerne l'aide à la réutilisation, tant de spécifications de base (vues simples), que de colle (patrons de colle) ou plus généralement de compositions (patrons). Ceci passe entre autres par le recensement de patrons et la définition de compositions abstraites, ainsi que la définition de distances entre interfaces dans un but de recherche dans une librairie de spécifications. Ceci pourrait être basé sur une bissimulation modulo une distance en profitant du fait que nos composants sont définis en termes de systèmes de transitions. Un dernier point, que nous avons uniquement esquissé dans ce manuscrit concerne les liens entre notre formalisme, notre méthode et UML. Nous ne nous plaçons pas dans une approche consistant à tenter de formaliser UML, mais plutôt celle consistant à utiliser, lorsque cela est possible la notation UML dans le cadre des concepts présents dans les deux formalismes.

Enfin, plus particulièrement sur la génération de code, il est nécessaire de pouvoir étendre les mécanismes décrits pour prendre en compte toute l'expressivité de KORRIGAN. Il serait aussi intéressant d'étudier la transformation vers d'autres langages concurrents orientés objets, prenant par exemple en compte des normes de communication comme CORBA ou les travaux en cours autour des composants objets distribués (comme les Enterprise Java Beans par exemple).

Bibliographie

[1] Xfig version 3.2 user manual. http://www-epb.lbl.gov/xfig/.

[2] T. AALTONNEN, M. KATARA et R. PITKÄNEN. DisCo Toolset - The New Generation. In G. SCHELLHORN et W. REIF, réds., *The 4th Workshop on Tools for System Design and Verification, FMTOOLS'2000*, pages 11–16, 2000.

[3] M. ABADI et L. LAMPORT. Conjoining Specifications. *ACM Toplas*, 17(3):507–534, 1995.

[4] J.-R. ABRIAL. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, août 1996.

[5] K. AJAMI, S. HADDAD et J. M. ILIÉ. Exploiting Symmetry in Linear Time Temporal Logic Model Checking: One Step Beyond. In B. STEFFEN, réd., *Tools and Algorithms for Construction and Analysis of Systems, TACAS'98*, volume 1384 of *Lecture Notes in Computer Science*, pages 52–67. Springer-Verlag, 1998.

[6] M. ALLEMAND. Verification of properties involving logical and physical timing features. In *Génie Logiciel & Ingénierie de Systèmes & leurs Applications, ICSSEA'2000*, 2000.

[7] M. ALLEMAND, C. ATTIOGBÉ et H. HABRIAS, réds. *International Workshop on Comparing Systems Specification Techniques - What questions are prompted by ones particular method of specification?*, 1998. ISBN 2-906082-29-5.

[8] L. A. ÁLVAREZ, J. M. MURILLO, F. SÁNCHEZ et J. HERNÁNDEZ. ActiveJava, un modelo de programación concurrente orientado a objeto. In *III Jornadas de Ingenería del Software, Murcia, Spain*, 1998.

[9] S. ANDOVA. Process algebra with probabilistic choice. In J.-P. KATOEN, réd., *ARTS'99*, volume 1601 of *Lecture Notes in Computer Science*, pages 111–129. Springer-Verlag, 1999.

[10] S. ANDOVA. Time and Probability in Process Algebra. In T. RUS, réd., *International Conference on Algebraic Methodology And Software Technology (AMAST'2000)*, volume 1816 of *Lecture Notes in Computer Science*, pages 323–338. Springer-Verlag, 2000.

[11] P. ANDRÉ. *Méthodes formelles et à objets pour le développement du logiciel : Etudes et propositions*. PhD Thesis, Université de Rennes I (Institut de Recherche en Informatique de Nantes), Juillet 1995.

[12] P. ANDRÉ. Spécification de l'atelier ASFO. Rapport technique 185, Institut de Recherche en Informatique de Nantes, 1999.

[13] P. ANDRÉ, D. CHIOREAN et J.-C. ROYER. The formal class model. In *Joint Modular Languages Conference*, pages 59–78, Ulm, Germany, 1994.

[14] P. ANDRÉ et J.-C. ROYER. How To Easily Extract an Abstract Data Type From a Dynamic Description. Research Report 159, Institut de Recherche en Informatique de Nantes, septembre 1997.

[15] P. ANDRÉ et J.-C. ROYER. An Algebraic Approach to Heterogeneous Software Systems. Rapport technique 00.7, Institut de Recherche en Informatique de Nantes, 2000.

[16] P. ARGON. *Etude sur l'application de méthodes formelles à la compilation et à la validation de programmes Electre*. Thèse de doctorat, Université de Nantes et Ecole Centrale de Nantes, 1998.

[17] A. ARNOLD. *Systèmes de transitions finis et sémantique des processus communicants*. Etudes et recherches en informatique. Masson, 1992.

[18] E. ASTESIANO et G. REGGIO. Formalism and method. In M. BIDOIT et M. DAUCHET, réds., *TAPSOFT'97*, volume 1214 of *Lecture Notes in Computer Science*, pages 93–114. Springer-Verlag, 1997.

[19] E. ASTESIANO. UML as heterogeneous multiview notation. strategies for a formal foundation. In L. ANDRADE, A. MOREIRA, A. DESHPANDE et S. KENT, réds., *Proceedings of the OOPSLA'98 Workshop on Formalizing UML. Why? How?*, 1998.

[20] E. ASTESIANO, M. BROY et G. REGGIO. *Algebraic Foundations of System Specification*, chapter Algebraic Specification of Concurrent Systems. 1999.

[21] E. ASTESIANO, M. CERIOLI et G. REGGIO. Pluggin Data Constructs into Paradigm-Specific Languages: Towards an Application to UML. In T. RUS, réd., *International Conference on Algebraic Methodology And Software Technology (AMAST'2000)*, volume 1816 of *Lecture Notes in Computer Science*, pages 273–292. Springer-Verlag, 2000.

[22] E. ASTESIANO et G. REGGIO. Algebraic Specification of Concurrency. In M. BIDOIT et C. CHOPPY, réds., *Recent Trends in Data Type Specification*, volume 655 of *Lecture Notes in Computer Science*, pages 1–39. Springer-Verlag, 1993. Invited Lecture.

[23] E. ASTESIANO et G. REGGIO. Labelled Transition Logic: An Outline. Rapport technique TR-96-20, DISI, 1996.

[24] E. ASTESIANO, G. REGGIO et F. MORANDO. The SMoLCS ToolSet. In P. D. MOSSES, M. NIELSEN et M. I. SCHWARTZBACH, réds., *TAPSOFT'95: Theory and Practice of Software Development*, volume 915 of *Lecture Notes in Computer Science*, pages 801–802. Springer-Verlag, 1995.

[25] E. ASTESIANO et E. ZUCCA. D-oids: a model for dynamic data-types. *Mathematical Structures in Computer Science*, 5(2):257–282, 1995.

[26] J. C. ATTIOGBÉ. Configuration Machines: A Simple Formalism for Specifying Multifaceted Systems. Rapport technique 181, Institut de Recherche en Informatique de Nantes, 1998.

[27] J. C. ATTIOGBÉ. Génération de code Promela pour la simulation et la vérification. In *International Conference on Software and Systems Engineering and their Applications, ICSSEA*, 1999.

[28] J. C. ATTIOGBÉ. Système de controle d'accès : une spécification à base des machines à configurations. In *Approches Formelles dans l'Assistance au Développement de Logiciels, AFADL*, 2000.

[29] J. AYCOCK. Compiling Little Languages in Python. In *7th International Python Conference*, Houston, Texas, November 1998. `http://www.foretec.com/python/workshops/1998-11/proceedings.html`.

[30] B. BARRAS, S. BOUTIN, C. CORNES, J. COURANT, Y. COSCOY, D. DELAHAYE, D. de RAUGLAUDRE, J.-C. FILLIÂTRE, E. GIMÉNEZ, H. HERBELIN, G. HUET, H. LAULHÈRE, C. MUÑOZ, C. MURTHY, C. PARENT-VIGOUROUX, P. LOISELEUR, C. PAULIN-MOHRING, A. SAÏBI et B. WERNER. The Coq Proof Assistant Reference Manual: Version 6.2.4. Rapport technique, INRIA, 1999. Online version available at http://pauillac.inria.fr/coq/doc/main.html.

[31] T. BASTEN et J. HOOMAN. Process Algebra in PVS. In R. CLEAVELAND, réd., *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 1579 of *Lecture Notes in Computer Science*, pages 270–284. Springer-Verlag, 1999.

[32] R. BASTIDE et P. PALANQUE. Cooperative Objects : a Concurrent Petri Net Based Object-Oriented Language. In *IEEE / System Man and Cybernetics 93 "Systems Engineering in the Service of Humans"*, 1993.

[33] R. BASTIDE. Approaches in unifying Petri nets and the Object-Oriented Approach. In *1st Workshop on Object-Oriented Programming and Models of Concurrency*, Turin, Italy, 1995. Available at http://wrcm.dsi.unimi.it/PetriLab/ws95/abstract/proc9502.html.

[34] E. BATTISTON, A. CHIZZONI et F. D. CINDIO. Inheritance and Concurrency in CLOWN. In *1st Workshop on Object-Oriented Programming and Models of Concurrency*, Turin, Italy, 1995.

[35] E. BATTISTON, F. D. CINDIO et G. MAURI. Modular Algebraic Nets to Specify Concurrent Systems. *IEEE Transactions on Software Engineering*, 22(10):689–705, 1996.

[36] D. M. BEAZLEY. SWIG : An Easy to Use Tool For Integrating Scripting Languages with C and C++. In *4th Annual USENIX Tcl/Tk Workshop*, pages 287–294, Monterey, California, 1996.

[37] J. A. BERGSTRA, J. HEERING et P. KLINT, réds. *Algebraic Specification*. ACM Press Frontier. The ACM Press in co-operation with Addison-Wesley, 1989.

[38] J. A. BERGSTRA et J. W. KLOP. *Math. & Comp. Sci. II*, volume 4 of *CWI Monograph*, chapter Process algebra: Specification and verification in bisimulation semantics. North Holland, 1986.

[39] G. BERRY et G. GONTHIER. The ESTEREL synchronous programming language : design, semantics, implementation. Technical Report RR-0842, Inria, Institut National de Recherche en Informatique et en Automatique, May 1988.

[40] O. BIBERSTEIN et D. BUCHS. Structured Algebraic Nets with Object-Orientation. In G. AGHA et F. DE CINDIO, réds., *Workshop on Object-Oriented Programming and Models of Concurrency'95*, pages 131–145, 1995.

[41] O. BIBERSTEIN, D. BUCHS et N. GUELF. Object-Oriented Nets with Algebraic Specifications: The CO-OPN/2 Formalism. In G. AGHA et F. DE CINDIO, réds., *Advances in Petri Nets on Object-Orientation*, Lecture Notes in Computer Science. to appear.

[42] O. BIBERSTEIN, D. BUCHS et N. GUELFI. CO-OPN/2: A Concurrent Object-Oriented Formalism. In C. . HALL, réd., *Second IFIP Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 57–72, 1997.

[43] M. BIDOIT, C. CHOPPY et F. VOISIN. Validation d'une spécification algébrique du "Transit-node" par prototypage et démonstration de théorèmes. Chapitre du Rapport final de l'Opération VTT, Validation et vérification de propriétés Temporelles et de Types de données, LaBRI, Bordeaux, 1994.

[44] M. BIDOIT. Types abstraits algébriques : spécifications structurées et présentations gracieuses. In *Colloque AFCET, Les mathématiques de l'informatique*, pages 347–357, Mars 1982.

[45] B. W. BOEHM. Verifying and Validating Software Requirements and Design Specifications. *IEEE Software*, 1(1):75–88, 1984.

[46] E. BOITEN, H. BOWMAN, J. DERRICK et M. STEEN. Viewpoint consistency in Z and LOTOS: A case study. In *FME'97, Formal Methods: Their Industrial Application and Strengthened Foundations*, Lecture Notes in Computer Science. Springer-Verlag, septembre 1997. To appear.

[47] T. BOLOGNESI et E. BRINKSMA. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–29, January 1988.

[48] P. BOROVANSKY, C. KIRCHNER, H. KIRCHNER, P. MOREAU et C. RINGEISSEN. An Overview of ELAN. In I. C. et H. KIRCHNER, réds., *2nd International Workshop on Rewriting Logic and its Applications, WRLA'98*, volume 15. Elsevier Science B. V., 1998.

[49] R. BOULTON, A. GORGON, M. GORDON, J. HEBERT et J. van TASSEL. Experience with embedding hardware description language in hol. In *International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 129–156, Nijmegen, North-Holland, June 1992. IFIP TC10/WG 10.2.

[50] E. BÖRGER et D. ROSENZWEIG. A Mathematical Definition of Full Prolog. *Science of Computer Programming*, 24(3):249–286, 1995.

[51] E. BÖRGER et W. SCHULTE. A Programmer Friendly Modular Definition of the Semantics of Java. In J. ALVES-FOSS, réd., *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 353–404. Springer-Verlag, 1999.

[52] J.-P. BRIOT et R. GUERRAOUI. Objets pour la programmation parallèle et répartie : intérêts, évolutions et tendances. *Technique et science informatiques*, 15(6):765–800, 1996.

[53] S. D. BROOKES, C. A. R. HOARE et A. W. ROSCOE. A Theory of Communicating Sequential Processes. *Journal of the ACM*, 31(3):560–599, July 1984.

[54] M. BROY. Specification and top down design of distributed systems. In H. EHRIG, C. FLOYD, M. NIVAT et J. THATCHER, réds., *TAPSOFT'85*, volume 185 of *Lecture Notes in Computer Science*, pages 4–28. Springer-Verlag, 1985.

[55] M. BROY, C. FACCHI, R. GROSU, R. HETTLER, H. HUSSMANN, D. NAZARETH, R. REGENSBURGER et K. STOLEN. The Requirement and Design Specification Language SPECTRUM. Rapport technique TUM-I9140, Technische Universität München, Institut für Informatik, 1991.

[56] M. BROY et M. WIRSING. Algebraic State Machines. In T. RUS, réd., *International Conference on Algebraic Methodology And Software Technology (AMAST'2000)*, volume 1816 of *Lecture Notes in Computer Science*, pages 89–118. Springer-Verlag, 2000.

[57] G. BRUNS. *Distributed Systems Analysis with CCS*. International Series in Computer Science. Prentice-Hall, 1997.

[58] N. BUSI, L. CAPRA et A. CHIZZONI. Representing Dynamic Aspects of a Concurrent OO Formalism by Mobile Nets. In *11th European Conference on Object-Oriented Programming (ECOOP'97)*, 1997.

[59] R. BÜSSOW, R. GEISLER, W. GRIESKAMP et M. KLAR. The μSZ Notation Version 1.0. Technical Report 97–26, TU Berlin, FB 13, 1997.

[60] R. BÜSSOW, R. GEISLER, W. GRIESKAMP et M. KLAR. Integrating Z with Dynamic Modeling Techniques for the Specification of Reactive Systems. 1998.

[61] R. BÜSSOW, W. GRIESKAMP, F. LATTEMANN et E. LEHMANN. The Definition of Dynamic Z. ESPRESS Projektbericht. URL: http://uebb.cs.tu-berlin.de/~wg/papers/dynz.ps, May 1997.

[62] M. BUTLER. csp2B: A practical approach to combining CSP and B. In J. M. WING, J. WOODCOCK et J. DAVIES, réds., *FM'99—Formal Methods, Volume I*, volume 1708 of *Lecture Notes in Computer Science*, pages 490–508. Springer, 1999.

[63] A. J. CAMILLERI, P. INVERARDI et M. NESI. Combining interaction and automation in process algebra verification. In S. ABRAMSKY et T. S. E. MAIBAUM, réds., *International Joint Conference on Theory and Practice of Software Development (TAPSOFT), Volume 2: Advances in Distributed Computing (ADC) and Colloquium on Combining Paradigms for Software Developmemnt (CCPSD)*, volume 494 of *Lecture Notes in Computer Science*, pages 283–296. Springer-Verlag, 1991.

[64] E. CANVER et F. W. von HENKE. Formal Specification and Verification of Object-Based Systems in a Temporal Logic Setting. Rapport technique, DeVa, 1997.

[65] D. CAROMEL et J. VAYSSIÈRE. A Java Framework for Seamless Sequential, Multi-threaded, and Distributed Programming. In *ACM Workshop "Java for High-Performance Network Computing"*, pages 141–150, Stanford University, Palo Alto, California, 1998.

[66] CCITT. *Recommendation Z.100: Specification and Description Language SDL*, blue book, volume x.1 édition, 1992.

[67] K. M. CHANDY et J. MISRA. *Parallel Program Design: A Fundation*. Addison Wesley, 1988.

[68] B. CHETALI. Formal verification of concurrent program using the Larch Prover. In U. H. ENGBERG, K. G. LARSEN et A. SKOU, réds., *Proceedings of the Workshop on Tools and Algorithms for The Construction and Analysis of Systems, TACAS* (Aarhus, Denmark, 19–20 May, 1995), number NS-95-2 in Notes Series, pages 174–186, Department of Computer Science, University of Aarhus, mai 1995. BRICS.

[69] C. CHOPPY, P. POIZAT et J.-C. ROYER. A Global Semantics for Views. In T. RUS, réd., *International Conference on Algebraic Methodology And Software Technology (AMAST'2000)*, volume 1816 of *Lecture Notes in Computer Science*, pages 165–180. Springer-Verlag, 2000.

[70] C. CHOPPY, P. POIZAT et J.-C. ROYER. Integration and Composition of Static and Dynamic "Views": Unifying Approach to Complex System Specification. In H. EHRIG, M. GROSSE-RHODE et F. OREJAS, réds., *INT'2000, Workshop on Integration of Specification Techniques with Applications in Engineering*, pages 12–20, Technische Universitat Berlin, Germany, 2000. Bericht-Nr. 2000/04, ISSN 1436-9915.

[71] C. CHOPPY, P. POIZAT et J.-C. ROYER. Specification of Mixed Systems in KORRIGAN with the Support of an UML-Inspired Graphical Notation. In *Fundamental Approaches to Software Engineering (FASE'2001)*, Lecture Notes in Computer Science. Springer-Verlag, 2001. à paraître.

[72] C. CHOPPY, P. POIZAT et J.-C. ROYER. The KORRIGAN Environment. *Journal of Universal Computer Science*, 2001. Special issue: Tools for System Design and Verification, ISSN : 0948-6968. à paraître.

[73] S. CHRISTENSEN, J. B. JØRGENSEN et L. M. KRISTENSEN. Design/CPN - A Computer Tool for Coloured Petri Nets. In E. BRINKSMA, réd., *Tools and Algorithms for Construction and Analysis of Systems, TACAS'97*, volume 1217 of *Lecture Notes in Computer Science*, pages 209–223. Springer-Verlag, 1997.

[74] R. CLARK. Using LOTOS in the Object-Based Development of Embedded Systems. In C. RATTRAY et R. CLARK, réds., *The Unified Computation Laboratory*, pages 307–319, Department of Computing Science and Mathematics, University of Stirling, Scotland, 1992. Oxford University Press.

[75] E. CLARKE, O. GRUMBERG et D. LONG. Verification tools for finite-state concurrent systems. In *A Decade of concurrency – Reflections and Perspectives*, volume 803 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[76] E. M. CLARKE, E. A. EMERSON et A. P. SISTLA. Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[77] COFI. CASL Examples. accessible from [79].

[78] COFI. The Common Framework Initiative for Algebraic specification and development, electronic archives. Notes and Documents accessible by WWW[1] and FTP[2], 1999.

[79] COFI LANGUAGE DESIGN TASK GROUP. CASL – The CoFI Algebraic Specification Language – Summary (version 1.0). Documents/CASL/Summary, in [78], 1999.

[80] COFI REACTIVE TASK GROUP. The CoFI-Reactive Systems Group Home Page . ReactiveSystemsTaskGroup.html, in [78], 1999.

[81] S. CONRAD. Compositional Object Specification and Verification. In I. ROZMAN et M. PIVKA, réds., *Proc. of the Int. Conf. on Software Quality (ICSQ'95), Maribor, Slovenia*, pages 55–64, 1995. Available at http://wwwiti.cs.uni-magdeburg.de/iti_db/veroeffentlichungen/95/Con95icsq.html.

[82] E. COSCIA et G. REGGIO. JTN: A Java-Targeted Graphic Formal Notation for Reactive and Concurrent Systems. In J.-P. FINANCE, réd., *Fundamental Approaches to Software Engineering (FASE'99)*, volume 1577 of *Lecture Notes in Computer Science*, pages 77–97. Springer-Verlag, 1999.

[83] G. COSTA et G. REGGIO. Specification of abstract dynamic data types: A temporal logic approach. *Theoretical Computer Science*, 73(2), 1997.

[84] P. DAUCHY et M.-C. GAUDEL. Algebraic Specifications with Implicit State. In *Working papers of the international Workshop on Information System Correctness and Reusability, IS-CORE'93*, 1993.

[85] J. DAVIES et S. SCHNEIDER. A brief history of Timed CSP. *Theoretical Computer Science*, 138(2):243–271, 1995.

[86] G. DELZANNO et A. PODELSKI. Model checking in clp. In R. CLEAVELAND, réd., *TACAS'99, International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 223–239. Springer-Verlag, 1999.

[87] G. DENKER et H.-D. EHRICH. Specifying Distributed Information Systems: Fundamentals of an Object-Oriented Approach Using Distributed Temporal Logic. In H. BOWMAN et J. DERRICK, réds., *2nd International IFIP Workshop on Formal Methods for Open Object-Based Distributed Systems*. Chapman & Hall, 1997.

[88] G. DENKER et P. HARTEL. TROLL – An Object Oriented Formal Method for Distributed Information System Design: Syntax and Pragmatics. Informatik-Bericht 97–03, Technische Universität Braunschweig, 1997. Available at http://www.cs.tu-bs.de/idb/html_e/home/denker/pub_97.html.

[89] G. DENKER et J. KÜSTER-FILIPE. Towards a Model for Asynchronously Communicating Objects. In H.-M. HAAV et B. THALHEIM, réds., *2nd Int. Baltic Workshop on Databases and Information Systems*, pages 182–193, 1996.

[90] G. DENKER, J. RAMOS, C. CALEIRO et A. SERNADAS. A Linear Temporal Logic Approach to Objects with Transactions. In M. JOHNSON, réd., *Sixth Int. Conf. on Algebraic Methodology and Software Technology (AMAST'97)*, volume 1349 of *Lecture Notes in*

[1] http://www.brics.dk/Projects/CoFI
[2] ftp://ftp.brics.dk/Projects/CoFI

Computer Science, pages 170–184. Springer-Verlag, 1997. Available at http://www.cs.tu-bs.de/idb/html_e/home/denker/pub_97.html.

[91] J. DERRICK, E. BOITEN, H. BOWMAN et M. STEEN. Weak refinement in Z. *Lecture Notes in Computer Science*, 1212:369–??, 1997.

[92] J. DERRICK, H. BOWMAN, E. A. BOITEN et M. STEEN. Comparing LOTOS and Z refinement relations. In *FORTE/PSTV'96*, pages 501–516, Kaiserslautern, Germany, octobre 1996. Chapman & Hall.

[93] N. DERSHOWITZ et J.-P. JOUANNAUD. *Rewrite Systems*, volume B of *Handbook of Theoretical Computer Science*, chapter 6, pages 243–320. Elsevier, 1990. Jan Van Leeuwen, Editor.

[94] R. DIACONESCU et K. FUTATSUGI. *CafeOBJ Report – The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, volume 6 of *AMAST Series in Computing*. World Scientific, 1998.

[95] A. M. DINIS MOREIRA. *Rigorous Object-Oriented Analysis*. Thèse de Doctorat, Department of Computing Science and Mathematics, University of Stirling, 1994.

[96] R. DUKE, G. ROSE et G. SMITH. Object-Z: A specification language advocated for the description of standards. *Computer Standards and Interfaces*, 17:511–533, 1995.

[97] H.-D. EHRICH, C. CALEIRO, A. SERNADAS et G. DENKER. *Logics for Databases and Information Systems*, chapter Logics for Specifying Concurrent Information Systems, pages 167–198. Kluwer Academic Publishers, 1998. Available at http://www.cs.tu-bs.de/idb/html_e/home/denker/pub_98.html.

[98] H.-D. EHRICH, G. DENKER et A. SERNADAS. Constructing Systems as Object Communities. In M.-C. GAUDEL et J.-P. JOUANNAUD, réds., *Theory and Practice of Software Development (TAPSOFT'93)*, volume 668 of *Lecture Notes in Computer Science*, pages 453–467. Springer-Verlag, 1993.

[99] H. EHRIG et B. MAHR. *Fundamentals of Algebraic Specification*, volume 1. Springer-Verlag, Berlin, 1985.

[100] H. EHRIG et F. OREJAS. "integration paradigm for data type and process specification techniques". *Bulletin of EATCS - Formal Specification Column*, 1998.

[101] H. EHRIG, R. GEISLER, M. KLAR et J. PADBERG. Horizontal and vertical structuring techniques for statecharts. In *CONCUR'97 : Concurrency Theory*, volume 1243 of *Lecture Notes in Computer Science*, pages 181–195, Warsaw, Poland, July 1997.

[102] H. EHRIG et F. OREJAS. Integration and Classification of Data Type and Process Specification Techniques. Rapport technique 98–10, TU Berlin, July 1998.

[103] R. EKKART, P. GRAUBMANN et J. GRABOWSKI. Tutorial on Message Sequence Charts. *Computer Networks and ISDN Systems*, 28(12):1629–1641, 1996.

[104] R. EKKART, P. GRAUBMANN et J. GRABOWSKI. Tutorial on Message Sequence Charts (MSC'96). In *Tutorials of the First Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification (FORTE/PSTV'96),*, 1996.

[105] J. ELLSBERGER, D. HOGREFE et A. SARMA. *SDL : Formal Object-oriented Language for Communicating Systems*. Prentice-Hall, 1997.

[106] U. ENGBERG, P. GRØNNING et L. LAMPORT. Mechanical Verification of Concurrent Systems with TLA. In *Fourth International Workshop on Computer-Aided Verification, CAV'92*, 1992.

[107] M. N. F. OREJAS et A. SANCHEZ. Implementation and behavioural equivalence: a survey. In M. BIDOIT et C. C. (EDS.), réds., *Recent Trends in data Type Specification*, volume 655 of *Lecture Notes in Computer Science*, pages 93–125. Springer-Verlag, août 1993.

[108] J.-C. FERNANDEZ, C. JARD, T. JÉRON et C. VIHO. Using on-the-fly verification techniques for the generation of test suites. In A. ALUR et T. HENZINGER, réds., *Conference on Computer-Aided Verification (CAV '96), New Brunswick, New Jersey, USA*, volume 1102 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[109] J.-C. FERNANDEZ. *ALDEBARAN : un système de vérification par réduction de processus communicants*. Thèse de Doctorat, Université Joseph Fourier, Grenoble, Mai 1988.

[110] J.-C. FERNANDEZ, H. GARAVEL, A. KERBRAT, R. MATEESCU, L. MOUNIER et M. SIGHIREANU. CADP: A Protocol Validation and Verification Toolbox. In *8th Conference on Computer-Aided Verification*, pages 437–440, New Brunswick, New Jersey, USA, 1996.

[111] J. L. FIADEIRO et T. MAIBAUM. Sometimes "Tomorrow" is "Sometime". Action Refinement in a Temporal Logic of Objects. In D. GABBAY et H. OHLBACH, réds., *Temporal Logic*, volume 827 of *Lecture Notes in Artificial Intelligence*, pages 48–66. Springer-Verlag, 1994.

[112] A. FINKEL et L. PETRUCCI. Propriétés de la composition/décomposition de réseaux de Petri et de leurs graphes de couverture. *RAIRO – Informatique Théorique et Applications*, 28(2):73–124, 1994.

[113] C. FISCHER. Combining csp and z. Rapport technique TRCF-97-1, Université d'Oldenburg, 1997.

[114] C. FISCHER. CSP-OZ: a combination of Object-Z and CSP. In H. BOWMAN et J. DERRICK, réds., *Proc. 2nd IFIP Workshop on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 423–438, Canterbury, UK, 1997. Chapman & Hall.

[115] C. FISCHER. How to Combine Z with a Process Algebra. In J. P. BOWEN, A. FETT et M. G. HINCHEY, réds., *ZUM'98: the Z Formal Notation*, volume 1493 of *Lecture Notes in Computer Science*, pages 5–23. Springer-Verlag, 1998.

[116] F.MOLLER et C. TOFTE. A temporal calculus of communicating systems (tccs). In *CONCUR-90: Theories of Concurrency*, volume 458 of *Lecture Notes in Computer Science*, pages 401–415. Springer-Verlag, 1990.

[117] C. FOURNET et G. GONTHIER. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385, St. Petersburg Beach, Florida, janvier 21-24 1996. ACM.

[118] C. FOURNET, G. GONTHIER, J.-J. LÉVY, L. MARANGET et D. RÉMY. A calculus of mobile agents. In *7th International Conference on Concurrency Theory (CONCUR'96)*, pages 406–421, Pisa, Italy, août 26-29 1996. Springer-Verlag. LNCS 1119.

[119] R. FRANCE et B. RUMPE, réds. *UML'99 – The Unified Modelling Language*, volume 1723 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[120] N. E. FUCHS. Specifications are (Preferably) Executable. *Software Engineering Journal*, 7(5):323–334, september 1992.

[121] K. FUTATSUGI, J. A. GOGUEN, J.-P. JOUANNAUD et J. MESEGUER. Principles of OBJ2. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 52–66, New Orleans, janvier 1985. SIGACT, SIGPLAN.

[122] A. J. GALLOWAY et W. STODDART. An operationnal semantics for zccs. In M. G. HINCHEY et S. LIU, réds., *International Conference of Formal Engineering Methods (IFCEM)*. IEEE Computer Society Press, 1997.

[123] A. GALLOWAY. *Integrated Formal Methods with Richer Methodological Profiles for the Development of Multi-Perspective Systems*. Thèse de Doctorat, University of Tesside, School of Computing and Mathematics, August 1996.

[124] H. GARAVEL. *Compilation et vérification de programmes LOTOS*. Thèse de doctorat (PhD Thesis), Université Joseph Fourier, Grenoble, Novembre 1989.

[125] H. GARAVEL. Introduction au langage LOTOS. Rapport technique, VERILOG, Centre d'Etudes Rhône-Alpes, Forum du Pré Milliet, Montbonnot, 38330 Saint-Ismier, 1990.

[126] H. GARAVEL et C. RODRIGUEZ. An example of LOTOS Specification : The Matrix Switch Problem. Rapport SPECTRE C22, Laboratoire de Génie Informatique — Institut IMAG, Grenoble, June 1990. Available at http://www.inrialpes.fr/vasy/Publications/Garavel-Rodriguez-90.html.

[127] S. GARLAND et J. GUTTAG. An overview of LP, the Larch Prover. In *Proc. of the Third International Conference on Rewriting Techniques and Applications*, volume 355 of *Lecture Notes in Computer Science*, pages 137–151. Springer-Verlag, 1989.

[128] M.-C. GAUDEL. A first introduction to PLUSS. Technical report, LRI, Université de Paris-Sud, Orsay, 1984.

[129] J. GOGUEN, C. KIRCHNER, H. KIRCHNER, A. MEGRELIS, J. MESEGUER et T. WINKLER. An introduction to OBJ-3. In J.-P. JOUANNAUD et S. KAPLAN, réds., *Proceedings 1st International Workshop on Conditional Term Rewriting Systems, Orsay (France)*, volume 308 of *Lecture Notes in Computer Science*, pages 258–263. Springer-Verlag, juillet 1987. Also as internal report CRIN: 88-R-001.

[130] M. J. C. GORDON et T. F. MELHAM, réds. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[131] GOSLING, JOY et STEELE. *The Java Language Specification*. Addison Wesley, 1996.

[132] B. GRAHLMANN. Combining Finite Automata, Parallel Programs and SDL Using Petri Nets. In B. STEFFEN, réd., *Proceedings of TACAS'98 (Tools and Algorithms for the Construction and Analysis of Systems)*, volume 1384 of *Lecture Notes in Computer Science*, pages 102–117. Springer-Verlag, mars 1998.

[133] W. GRIESKAMP, M. HEISEL et H. DÖRR. Specifying Embedded Systems with Statecharts and Z: An Agenda for Cyclic Software Components. In E. ASTESIANO, réd., *FASE'98*, volume 1382 of *Lecture Notes in Computer Science*, pages 88–106. Springer-Verlag, 1998.

[134] N. GUELFI, O. BIBERSTEIN, D. BUCHS, E. CANVER, M.-C. GAUDEL, F. von HENKE et D. SCHWIER. Comparison of object-oriented formal methods. Technical report of the esprit long term research project 20072 "design for validation", University of Newcastle Upon Tyne, Department of Computing Science, 1997. ftp://lglftp.epfl.ch/pub/Papers/guelfi_coofm.ps.gz.

[135] P. GUERNIC, T. GAUTIER, M. BORGNE et C. MAIRE. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1305–1320, septembre 1991.

[136] R. GUERRAOUI. Les langages concurrents à objets. *Technique et science informatiques*, 14(8):945–971, Octobre 1995.

[137] Y. GUREVICH. May 1997 draft of the asm guide. Rapport technique CSE-TR-336-97, University of Michigan EECS Department, 1997.

[138] N. HALBWACHS, F. LAGNIER et C. RATEL. Programming and verifying critical systems by means of the synchronous data-flow programming language Lustre. *IEEE Transactions on Software Engineering*, 18(9), septembre 1992. Special Issue on the Specification and Analysis of Real-Time Systems.

[139] N. HALBWACHS, X. NICOLLIN, P. RAYMOND et D. WEBER. Test Automatique de Systèmes Réactifs. In F. CASSEZ, C. JARD, O. ROUX et B. ROZOY, réds., *MOVEP'98 – MOdélisation et VErification des processus Parallèles*, pages 1–7, Nantes, France, juillet 1998.

[140] D. HAREL. On Visual Formalisms. *Communications of the ACM*, 31(5):514–530, 1988.

[141] D. HAREL, H. LACHOVER, A. NAAMAD, A. PNUELI, M. POLITI, R. SHERMAN, A. SHTULL-TRAURING et M. TRAKHTENBROT. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *IEEE Transactions on Software Engineering*, 16(3):403–414, 1990.

[142] D. HAREL et A. NAAMAD. The statemate semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, October 1996.

[143] I. J. HAYES et C. B. JONES. Specifications are not Necessary Executable. *Software Engineering Journal*, 4(6):330–338, November 1989.

[144] M. HEISEL. Agendas – A Concept to Guide Software Development Activities. In R. N. HORSPOOL, réd., *Systems Implementation 2000*, pages 19–32. Chapman & Hall, 1998.

[145] M. HEISEL et N. LÉVY. Using LOTOS Patterns to Characterize Architectural Styles. In M. BIDOIT et M. DAUCHET, réds., *TAPSOFT'97*, volume 1214 of *Lecture Notes in Computer Science*, pages 818–832, 1997.

[146] M. HEISEL et C. SUHL. Formal specification of safety-critical software with Z and real-time CSP. In E. SCHOITSCH, réd., *SAFECOMP'96: 15th International Conference on Computer Safety, Reliability and Security*, page 31, Vienna, Austria, 1996. Springer.

[147] M. HEISEL et C. SUHL. Methodological Support for Formally Specifying Safety-Critical Software. In P. DANIEL, réd., *16th International Conference on Computer Safety, Reliability and Security, SAFECOMP*, pages 295–308. Springer-Verlag, 1997.

[148] M. HENNESSY et H. LIN. Symbolic Bisimulations. *Theoretical Computer Science*, 138(2):353–389, 1995.

[149] B. HNATKOWSKA et H. ZBIGNIEW. Extending the UML with a Multicast Synchronisation. In T. CLARK, réd., *Proceedings of the third Rigorous Object-Oriented Methods Workshop (ROOM)*, BCS eWics, ISBN: 1-902505-38-7, 2000.

[150] C. HOARE. Proof of Correctness of Data Representations. *Acta Informatica*, 1:271–281, 1972.

[151] C. HOARE. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[152] G. J. HOLZMANN. The Spin Model Checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[153] ISO/IEC. ESTELLE: A Formal Description Technique based on an Extended State Transition Model. ISO/IEC 9074, International Organization for Standardization, 1989.

[154] ISO/IEC. LOTOS: A Formal Description Technique based on the Temporal Ordering of Observational Behaviour. ISO/IEC 8807, International Organization for Standardization, 1989.

[155] ITU-T. *Recommendation Z.120: Message Sequence Chart (MSC)*, 1994.

[156] C. S. JENSEN, J. CLIFFORD, R. ELMASRI, S. K. GADIA, P. HAYES, S. JAJODIA, C. DYRESON, F. GRANDI, W. KAFER, N. KLINE, N. LORENTZOS, Y. MITSOPOULOS, A. MONTANARI, D. NONEN, E. PERESSI, B. PERNICI, J. F. RODDICK, N. L. SARDA, M. R. SCALAS, A. SEGEV, R. T. SNODGRASS, M. D. SOO, A. TANSEL, P. TIBERIO et G. WIEDERHOLD. A Consensus Glossary of Temporal Database Concepts. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 23(1):52–64, 1994. subsumed by version at http://www.cs.auc.dk/ csj/Glossary/.

[157] K. JENSEN. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use.* Monographs in Theoretical Computer Science. Springer-Verlag, 1997. 3 volumes.

[158] C. B. JONES. *Systematic Software Development Using VDM*. Prentice-Hall International, Englewood Cliffs, New Jersey, second édition, 1990. ISBN 0-13-880733-7.

[159] M. JOURDAN et F. MARANINCHI. A modular state/transition approach for programming real-time systems. In *ACM Sigplan Workshop on Language, compiler and tool support for real-time systems*, Orlando, FL, juin 1994.

[160] H.-M. JÄRVINEN et R. KURKI-SUONIO. DisCo specification language: marriage of actions and objects. In *11th International Conference on Distributed Computing Systems*, pages 142–151. IEEE Computer Society Press, 1991.

[161] I. JTC1/SC7/WG14. E-LOTOS Final Draft International Standard (in balloting). ftp://ftp.inrialpes.fr/pub/vasy/publications/elotos/elotos-fdis/, 2000.

[162] S. KALVALA. A Formulation of TLA in Isabelle. In E. T. SCHUBERT, P. J. WINDLEY et J. ALVES-FOSS, réds., *Higher Order Logic Theorem Proving and Its Applications, TPHOLs'95*, volume 971 of *Lecture Notes in Computer Science*, pages 214–228. Springer-Verlag, 1995.

[163] J. v. KATWIJK et H. TOETENEL. Loose specification of real time systems. *Informatica; an international journal of computing and informatics*, 19(1):25–42, 1995.

[164] C. KHOURY. *Définition d'une approche orientée-objet de la spécification algébrique de systémes informatiques*. Thèse de Doctorat, Université de Paris-Sud, 1999.

[165] G. KICZALES, J. des RIVIÉRES et D. G. BOBROW. *The art of meta-object protocol*. The MIT Press, 1992.

[166] C. KIRKWOOD et M. THOMAS. Towards a Symbolic Modal Logic for LOTOS. In *Northern Formal Methods Workshop NFM'96*, eWIC, 1997.

[167] T. KNAPIK. Concurrency and real–time specification with many–sorted logic and abstract data types: an example. In J. TANKOANO, réd., *2nd African Conference on Research in Computer Science*, pages 401–418, 1994.

[168] S. KOLYANG et T. WOLFF. A structure preserving encoding of z in isabelle/hol. In *International Conference on Theorem Proving in Higher Order Logic*, volume 1125 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[169] P. KOSIUCZENKO et M. WIRSING. Timed rewriting logic with an application to object-based specification. *Science of Computer Programming*, 28(2–3):225–246, 1997.

[170] D. KOZEN. Results on the Propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[171] B. KRIEG-BRÜCKNER, J. PELESKA, E.-R. OLDEROG et A. BAER. The UniForM Workbench, a Universal Development Environment for Formal Methods. In *Proceedings of the International FM'99*, pages 1187–1205, Toulouse, France, septembre 1999. Springer-Verlag LNCS 1708.

[172] C. LAKOS. The Consistent Use of Names and Polymorphism in the Definition of Object Petri Nets. 380-399 . In W. R. JONATHAN BILLINGTON, réd., *Application and Theory of Petri Nets 1996, 17th International Conference*, volume 1091 of *Lecture Notes in Computer Science*, pages 380–399. Springer-Verlag, 1996.

[173] T. LAMBOLAIS, N. LÉVY et J. SOUQUIÈRES. Assistance au développement de spécifications de protocoles de communication. In *AFADL'97 Approches Formelles dans l'Assistance au Développement de Logiciel*, pages 73–84, 1997.

[174] L. LAMPORT. "Sometime" is Sometimes "Not Never". On the Temporal Logic of Programs. In *7th Annual ACM Symposium on Principles of Programming Languages, POPL'80*, pages 163–173, 1980.

[175] L. LAMPORT. Hybrid Systems in TLA+. In *Hybrid Systems*, Lecture Notes in Computer Science. Springer-Verlag, 1993.

[176] L. LAMPORT. Introduction to TLA. SRC Technical Note 1994-001, DIGITAL, 1994. Available at http://gatekeeper.dec.com/pub/DEC/SRC/technical-notes/abstracts/src-tn-1994-001.html.

[177] L. LAMPORT. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[178] L. LAMPORT. TLA in Pictures. *IEEE Transactions on Software Engineering*, 21(9):768–775, 1995. Available at http://www.research.digital.com/SRC/personal/Leslie_Lamport/tla/papers.html.

[179] L. LAMPORT. The Operators of TLA+. SRC Technical Note 1997-006a, DIGITAL, 1997. Available at http://gatekeeper.dec.com/pub/DEC/SRC/technical-notes/abstracts/src-tn-1997-006.html.

[180] K. G. LARSEN, P. PETTERSSON et W. YI. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, 1997.

[181] K. G. LARSEN, P. PETTERSSON et W. YI. UPPAAL: Status and developments. Number 1254 in Lecture Notes in Computer Science, pages 456–459. Springer–Verlag, juin 1997.

[182] G. J. LEDUC. LOTOS, un outil utile ou un autre langage académique ? In *Actes des Neuvièmes Journées Francophones sur l'Informatique — Les réseaux de communication — Nouveaux outils et tendances actuelles (Liège)*, Janvier 1987.

[183] F. LESSKE. Constructive Specifications of Abstract Data Types Using Temporal Logic. In A. NERODE et M. A. TAITSLIN, réds., *Logical Foundations of Computer Science, LFCS'92*, volume 620 of *Lecture Notes in Computer Science*, pages 269–280. Springer-Verlag, 1992.

[184] D. LIGHTFOOT. *Formal Specification using Z*. Macmillan, 1991.

[185] L. LOGRIPPO, M. FACI et M. HAJ-HUSSEIN. An Introduction to LOTOS: Learning by Examples. *Computer Networks and ISDN Systems*, 23:325–342, 1992. improved version available by ftp at lotos.csi.uottawa.ca.

[186] M. LUTZ. *Programming Python*. O'Reilly & Associates, 1996.

[187] M. MAGNAN et C. OUSSALAH. *Objets et composition*, chapter 2, pages 61–92. InterEditions, 1997.

[188] F. MARANINCHI. The Argos language: Graphical Representation of Automata and Description of Reactive Systems. In *IEEE Workshop on Visual Languages*, 1991.

[189] F. MARANINCHI. Operational and compositional semantics of synchronous automaton compositions. In *CONCUR*, volume 630 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[190] D. A. MARCA et M. C. L.. *SADT – Structured Analysis and Design Technique*. McGraw-Hill Book Company, 1988.

[191] R. MATEESCU. *Vérification des propriétés temporelles des programmes parallèles*. Thèse de Doctorat, Institut National Polytechnique de Grenoble, 1998.

[192] R. MATEESCU et H. GARAVEL. XTL: A Meta-Language and Tool for Temporal Logic Model-Checking. In *International Workshop on Software Tools for Technology Transfer STTT'98*, 1998.

[193] S. MATSUOKA et A. YONEZAWA. *Research Directions in Concurrent Object-Oriented Programming*, chapter 4: Analysis of inheritance anomaly in object-oriented concurrent programming languages, pages 107–150. The MIT Press, 1993.

[194] S. MAUW et M. A. RENIERS. An Algebraic Semantics of Basic Message Sequence Charts. *The Computer Journal*, 37(4):269–277, 1994.

[195] S. MAUW et G. J. VELTINK. An introduction to psf_d. In J. DIAZ et F. OREJAS, réds., *TAPSOFT'89*, volume 352 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.

[196] S. MERZ. A User's Guide to TLA. In F. CASSEZ, C. JARD, O. ROUX et B. ROZOY, réds., *MOVEP'98 – MOdélisation et VErification des processus Parallèles*, pages 29–44, Nantes, France, juillet 1998.

[197] J. MESEGUER. A logical theory of concurrent objects and its realization in the maude language. In G. AGHA, P. WEGNER et A. YONEZAWA, réds., *Research Directions in Concurrent Object-Oriented Programming*, pages 314–390. The MIT Press, 1993.

[198] J. MESEGUER et T. WINKLER. Parallel programming in Maude. In J.-P. BANATRE et D. L. METAYER, réds., *Research Directions in High-Level Parallel Programming Languages*, volume 574 of *Lecture Notes in Computer Science*, pages 253–293. Springer-Verlag, 1992.

[199] J. MESEGUER. A Logical Theory of Concurrent Objects. In *ECOOP/OOPSLA*, pages 101–115, 1990.

[200] J. MESEGUER. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[201] J. MESEGUER. Rewriting logic as a semantic framework for concurrency: a progress report. In *CONCUR'96 : Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, pages 331–372, Pisa, Italy, 1996.

[202] B. MEYER. *Object-oriented Software Construction*. International Series in Computer Science. Prentice Hall, 1988.

[203] R. MILNER, J. PARROW et D. WALKER. A calculus of mobile processes. *Information and Computation*, (100):1–77, 1992.

[204] R. MILNER. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1980.

[205] R. MILNER. *Communication and Concurrency*. International Series in Computer Science. Prentice-Hall, 1989.

[206] A. MORZENTI, D. MANDRIOLI et C. GHEZZI. A Model-Parametric Real-Time Logic. *ACM TOPLAS-Transactions on Programming Languages and Systems*, 14(4):521–573, 1992.

[207] T. MOSSAKOWSKI et B. KRIEG-BRÜCKNER. Static semantic analysis and theorem proving for CASL. *Lecture Notes in Computer Science*, 1376:333–348, 1998.

[208] P. D. MOSSES. CASL: A Guided Tour of Its Design. In J. FIADEIRO, réd., *Recent Trends in Algebraic Development Techniques, Selected Papers of the 13th International Workshop on Algebraic Development Techniques (WADT'98)*, volume 1589 of *Lecture Notes in Computer Science*, pages 216–240, Lisbon, Portugal, 1999. Springer-Verlag.

[209] L. MOUNIER. A LOTOS Specification of a "Transit-Node". Rapport technique 94-8, SPECTRE, VERIMAG, 1994.

[210] J. M. MURILLO, J. HERNÁNDEZ, F. SÁNCHEZ et L. A. ÁLVAREZ. Coordinated Roles: Promoting Re-usability of Coordinated Active Objects Using Event Notification Protocols. In P. CIANCARINI et A. L. WOLF, réds., *Third International Conference, COORDINATION'99*, volume 1594 of *Lecture Notes in Computer Science*, Amsterdam, The Nederlands, April 1999. Springer-Verlag.

[211] M. NESI. Value-passing ccs in hol. In J. J. JOYCE et C.-J. H. SEGER, réds., *6th Workshop on Higher Order Logic Theorem Proving and Applications*, volume 780 of *Lecture Notes in Computer Science*, pages 352–365. Springer-Verlag, 1993.

[212] M. NIELSEN, V. SASSONE et G. WINSKEL. Relationships between Models of Concurrency. In *REX'93 : A Decade of Concurrency*, volume 803 of *Lecture Notes in Computer Science*, pages 425–475. Springer-Verlag, 1994.

[213] T. NIPKOW et L. C. PAULSON. Isabelle-91. In D. KAPUR, réd., *11th International Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence, pages 673–676. Springer-Verlag, 1992.

[214] S. OWRE, S. RAJAN, J. RUSHBY, N. SHANKAR et M. SRIVAS. PVS: Combining specification, proof checking, and model checking. In R. ALUR et T. A. HENZINGER, réds., *Computer-Aided Verification, CAV '96*, number 1102 in Lecture Notes in Computer Science, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[215] S. OWRE, J. M. RUSHBY et N. SHANKAR. PVS: A prototype verification system. In D. KAPUR, réd., *Proceedings of the 11th International Conference on Automated Deduction (CADE-11)*, volume 607 of *LNAI*, pages 748–752, Saratoga Springs, NY, juin 1992. Springer.

[216] J. PADBERG. Abstract Petri Nets as a Uniform Approach to High/Level Petri Nets. In J. FIADEIRO, réd., *Recent Trends in Algebraic Development Techniques, Selected Papers of the 13th International Workshop on Algebraic Development Techniques (WADT'98)*, volume 1589 of *Lecture Notes in Computer Science*, pages 241–260, Lisbon, Portugal, 1999. Springer-Verlag.

[217] M. PAPATHOMAS, J. HERNÀNDEZ, J. M. MURILLO et F. SÀNCHEZ. Inheritance and expressive power in concurrent object-oriented programming. In *LMO'97 Langages et Modèles à Objets*, pages 45–60, 1997.

[218] B. C. PIERCE et D. N. TURNER. Concurrent objects in a process calculus. In T. ITO et A. YONEZAWA, réds., *Theory and Practice of Parallel Programming (TPPP)*, volume 907 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[219] A. PODELSKI. Model checking as constraint solving. In J. PALSBERG, réd., *SAS'2000: Static Analysis Symposium*, volume 1824 of *Lecture Notes in Computer Science*, pages 221–237. Springer-Verlag, 2000.

[220] T. POIDEVIN. LOTOS. Rapport de stage de DESS, DESS Informatique, Université d'Orsay, 1991.

[221] P. POIZAT. Applications de la réécriture de termes aux modèles à objet. Master's thesis, DEA Informatique, Université de Nantes, Juin 1995.

[222] P. POIZAT. *Software Specification Methods: An Overview Using a Case Study*, chapter SDL: a Specification and Description Language based on an Extended Finite State Machine Model with Abstract Data Types. Formal Approaches to Computing and Information Technology (FACIT). Springer-Verlag, 2000.

[223] P. POIZAT, C. CHOPPY et J.-C. ROYER. Concurrency and Data Types: A Specification Method. An Example with LOTOS. In J. FIADEIRO, réd., *Recent Trends in Algebraic Development Techniques, Selected Papers of the 13th International Workshop on Algebraic Development Techniques (WADT'98)*, volume 1589 of *Lecture Notes in Computer Science*, pages 276–291, Lisbon, Portugal, 1999. Springer-Verlag.

[224] P. POIZAT, C. CHOPPY et J.-C. ROYER. From Informal Requirements to COOP: a Concurrent Automata Approach. In J. WING, J. WOODCOCK et J. DAVIES, réds., *FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems*, volume 1709 of *Lecture Notes in Computer Science*, pages 939–962, Toulouse, France, 1999. Springer-Verlag.

[225] P. QUEINNEC, M. FILALI, G. PADIOU, P. MAURAN et P. PAPAIX. Système de contrôle d'accès. Développement d'une spécification formelle en Unity. Rapport technique 97-33-R, Institut de Recherche en Informatique de Toulouse, 1997.

[226] RAISE LANGUAGE GROUP. *The RAISE Specification Language*. BCS Practitioner Series. Prentice Hall International, 1992.

[227] J. RAMOS et A. SERNADAS. A Brief Introduction to GNOME. Research report, Section of Computer Science, Department of Mathematics, Instituto Superior Técnico, 1096 Lisboa, Portugal, 1995. Available at ftp://ftp.cs.math.ist.utl.pt/pub/SernadasA/95-RS-GnomeInt.ps.

[228] J. RATHKE et M. HENNESSY. Local Model Checking for Value-Passing Processes (Extended Abstract). In M. ABADI et T. ITO, réds., *Third International Symposium on Theoretical Aspects of Computer Software TACS'97*, volume 1281 of *Lecture Notes in Computer Science*, pages 250–266. Springer-Verlag, 1997.

[229] G. REGGIO, M. CERIOLI et E. ASTESIANO. An Algebraic Semantics of UML Supporting its Multiview Approach. In D. HEYLEN, A. NIJHOLT et G. SCOLLO, réds., *Twente Workshop on Language Technology, AMiLP 2000*, 2000.

[230] G. REGGIO et L. REPETTO. CASL-Chart : A Combination of Statecharts and of the Algebraic Specification Language CASL. In T. RUS, réd., *International Conference on Algebraic Methodology And Software Technology (AMAST'2000)*, volume 1816 of *Lecture Notes in Computer Science*, pages 243–257. Springer-Verlag, 2000.

[231] G. REGGIO et R. WIERINGA. Thirty one Problems in the Semantics of UML 1.3 Dynamics. In *OOPSLA'99 workshop "Rigorous Modelling and Analysis of the UML: Challenges and Limitations"*, 1999.

[232] G. REGGIO, E. ASTESIANO et C. CHOPPY.. Casl-ltl: A casl extension for dynamic reactive systems - summary. Rapport technique DISI-TR-99-34, DISI - Università di Genova, 1999. Revised February 2000.

[233] G. REGGIO et M. LAROSA. A graphic notation for formal specifications of dynamic systems. In J. FITZGERALD, C. B. JONES et P. LUCAS, réds., *Formal Methods Europe (FME'97)*, volume 1313 of *Lecture Notes in Computer Science*, pages 40–61. Springer-Verlag, 1997.

[234] W. REIF. The KIV-approach to Software Verification. In M. BROY et S. JAHNICHEN, réds., *KORSO: Methods, Languages, and Tools for the Construction of Correct Software - Final Report*, volume 1009 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[235] W. REISIG. *Petri Nets: an Introduction*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1985.

[236] A. W. ROSCOE et G. BARETT. Unbounded Nondeterminism in CSP. In M. MAIN, A. MELTON, M. MISLOVE et D. SCHMIDT, réds., *9th International Conference in Mathematical Foundation of Programming Semantics*, volume 442 of *Lecture Notes in Computer Science*, pages 160–193. Springer-Verlag, 1989.

[237] A. W. ROSCOE, J. C. P. WOODCOCK et L. WULF. Non-Interference Through Determinism. In D. GOLLMANN, réd., *Third European Symposium on Research in Computer Security, ESORICS'94*, volume 875 of *Lecture Notes in Computer Science*, pages 3–18. Springer-Verlag, 1994.

[238] J. RUMBAUGH, M. BLAHA, W. LORENSEN, F. EDDY et W. PREMERLANI. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[239] V. SASSONE, M. NIELSEN et G. WINSKEL. A Classification of Models for Concurrency. In *Proceedings of Fourth International Conference on Concurrency Theory, CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 82–96. Springer-Verlag, 1993.

[240] S. SCHNEIDER. An operational semantics for timed CSP. *Information and Computation*, 116(2):193–213, 1995.

[241] A. SERNADAS, C. SERNADAS et H.-D. EHRICH. Object-Oriented Specification of Databases: An Algebraic Approach. In P. M. STOECKER et W. KENT, réds., *1"th International Conference on Very Large Databases VLDB'87*, pages 107–116. VLDB Endowment Press, 1987.

[242] A. SERNADAS, C. SERNADAS et J. F. COSTA. Object Specification Logic. *Journal of Logic and Computation*, 5(5):603–630, 1995.

[243] C. SHANKLAND, M. THOMAS et E. BRINKSMA. Symbolic Bisimulation for Full LOTOS. In *Algebraic Methodology and Software Technology (AMAST'97)*, volume 1349 of *Lecture Notes in Computer Science*, pages 479–493. Springer-Verlag, 1997.

[244] M. SIGHIREANU. LOTOS NT User's Manual. http://www.inrialpes.fr/vasy/traian/, 2000.

[245] S. SKIDMORE, G. MILLS et L. FORMEN. *SSADM Models & Methods, Version 4 - Revised edition*. NCC Publications, 1994.

[246] G. SMITH. A Fully-Abstract Semantics of Classes for Object-Z. *Formal Aspects of Computing*, 7(E):30–65, 1995.

[247] G. SMITH. A Semantic Integration of Object-Z and CSP for the Specification of Concurrent Systems. In J. FITZGERALD, C. B. JONES et P. LUCAS, réds., *Formal Methods Europe (FME'97)*, volume 1313 of *Lecture Notes in Computer Science*, pages 62–81. Springer-Verlag, 1997.

[248] G. SMOLKA. The Oz programming model. In J. van LEEUWEN, réd., *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 324–343. Springer-Verlag, 1995.

[249] R. SOFTWARE. Unified Modeling Language, Version 1.3. Rapport technique, Rational Software Corp., http://www.rational.com/uml, juin 1999.

[250] J. SOUQUIÈRES et N. LÉVY. PROPLANE: A Specification Development Environment. In M. WIRSING et M. NIVAT, réds., *International Conference on Algebraic Methodology and Software Technology (AMAST'96)*, volume 1101 of *Lecture Notes in Computer Science*, pages 612–615. Springer-Verlag, 1996.

[251] M. SPIVEY. *The Z Notation: a Reference Manual*. Series in Computer Science. Prentice-Hall, 1992.

[252] C. STIRLING. An introduction to modal and temporal logics for ccs. In *1989 Joint UK/Japan workshop on Concurrency*, volume 491 of *Lecture Notes in Computer Science*, pages 2–20. Springer-Verlag, 1991.

[253] W. J. STODDART. The Event Calculus, Specification and Modelling of Real Time Systems using Diagrams and Z. Rapport technique, University of Teeside, 1994.

[254] W. J. STODDART. An Introduction to the Event Calculus. In J. P. BOWEN, M. G. HINCHEY et D. TILL, réds., *ZUM'97: The Z Formal Specification Notation*, volume 1212 of *Lecture Notes in Computer Science*, pages 10–34. Springer-Verlag, 1997.

[255] W. J. STODDART. The Event Calculus Extensions for Modelling Hybrid Systems. Rapport technique, University of Teeside, 1997.

[256] W. J. STODDART. The Event Calculus vsn 2. Rapport technique, University of Teeside, 1997.

[257] B. STROUSTRUP. *The C++ Programming Language*. Addison Wesley, 1987.

[258] K. TAGUCHI et K. ARAKI. The state-based CCS semantics for concurrent Z specification. In M. G. HINCHEY et S. LIU, réds., *Formal Engineering Methods: Proc. 1st International Conference on Formal Engineering Methods (ICFEM'97)*, pages 283–292, Hiroshima, Japan, 12–14 novembre 1997. IEEE Computer Society Press.

[259] J.-P. TALPIN, A. BENVENISTE, B. CAILLAUD, C. JARD, Z. BOUZIANE et H. CANON. BDL, a language of distributed reactive objects. In *ISORC'98, The 1st IEEE International Symposium on Object-oriented Real-time Distributed Computing*, 1998.

[260] The Raise Method GROUP. *The RAISE Development Method*. The Practitioner Series. Prentice-Hall, 1995.

[261] H. TREHARNE et S. SCHNEIDER. Using a process algebra to control B OPERATIONS. In *IFM'99 1st International Conference on Integrated Formal Methods*, pages 437–457, York, juin 1999. Springer-Verlag.

[262] K. TURNER. Relating architecture and specification. *Computer Networks and ISDN Systems*, 29(4):437–456, 1997.

[263] K. J. TURNER, réd. *Using Formal Description Techniques, An introduction to Estelle, LOTOS and SDL*. Wiley, 1993.

[264] R. UDINK et J. KOK. On the relation between Unity properties and sequences of states. In J. de BAKKER, W.-P. de ROEVER et G. ROZENBERG, réds., *Semantics: Foundations and Applications*, volume 666 of *Lecture Notes in Computer Science*, pages 594–608. Springer-Verlag, 1993.

[265] C. A. VISSERS, G. SCOLLO, M. VAN SINDEREN et E. BRINKSMA. Specification Styles in Distributed Systems Design and Verification. *Theoretical Computer Science*, 89(1):179–206, 1991.

[266] R. WIERINGA, R. JUNGCLAUS, P. HARTEL, G. SAAKE, et T. HARTMANN. OMTROLL – Object Modeling in Troll. In U. W. LIPECK et G. KOSCHORREK, réds., *International Worskshop on Information Systems – Correctness and Reusability (IS-CORE'93)*, pages 267–283, 1993.

[267] G. WINSKEL. *The Formal Semantics of Programming Languages, An Introduction*. The MIT Press, 1993.

[268] M. WIRSING. Structured algebraic specifications: A kernel language. Rapport technique, Passau, 1985.

[269] M. WIRSING. *Algebraic Specification*, volume B of *Handbook of Theoretical Computer Science*, chapter 13, pages 675–788. Elsevier, 1990. Jan Van Leeuwen, Editor.

[270] S. YOVINE. Kronos: A verification tool for real-time systems. *Springer International Journal of Software Tools for Technology Transfer*, 1(1–2), 1997.

[271] Y. YU, P. MANOLIOS et L. LAMPORT. Model Checking TLA+ Specifications. In L. PIERRE et T. KROPF, réds., *Correct Hardware Design and Verification Methods (CHARME'99)*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66. Springer-Verlag, 1999.

[272] A. V. ZAMULIN. Object-Oriented Abstract State Machines. In *International Workshop on Abstract State Machines*, 1998.

Liste des tableaux

Table des figures

Table des matières

Annexes

ANNEXE A

Notations

A.1 Notation textuelle

A.1.1 Systèmes de transition symboliques

Nous nous intéressons ici à leur représentation textuelle, ce qui présente essentiellement un intérêt au niveau de la description des étiquettes qu'il est possible d'attribuer aux états et aux transitions. Les STS ont aussi une représentation graphique qui est décrite plus loin.

La grammaire utilisée pour la représentation textuelle des STS est la suivante :

```
STS                       ::= STS INITIAL-TRANSITION TRANSITION*
INITIAL-TRANSITION        ::= • - TRANSITION-LABEL -> STATE
TRANSITION                ::= STATE - TRANSITION-LABEL -> STATE
STATE                     ::= STATE-LABEL
STATE-LABEL               ::= SIMPLE-STATE-IDENTIFIER
                            | TUPLE-STATE-IDENTIFIER
TUPLE-STATE-IDENTIFIER    ::= ( CONDITION-VALUE ( , CONDITION-VALUE )* )
CONDITION-VALUE           ::= CONDITION-IDENTIFIER
                            | ¬ CONDITION-IDENTIFIER
TRANSITION-LABEL          ::= STATIC-TRANSITION-LABEL
                            | DYNAMIC-TRANSITION-LABEL
STATIC-TRANSITION-LABEL   ::= [ [ GUARD ] ] STATIC-LABEL [ [ GUARD ] ]
DYNAMIC-TRANSITION-LABEL  ::= [ [ GUARD ] ] DYNAMIC-LABEL [ [ GUARD ] ]
STATIC-LABEL              ::= TERM
                            | ϵ
DYNAMIC-LABEL             ::= OFFER-TERM
                            | ϵ
OFFER-TERM                ::= INPUT* OUTPUT*
INPUT                     ::= GATE-IDENTIFIER INPUT-OFFER* [ FROM PID ]
OUTPUT                    ::= GATE-IDENTIFIER OUTPUT-OFFER* [ TO PID ]
INPUT-OFFER               ::= ? TYPED-VARIABLE-IDENTIFIER
OUTPUT-OFFER              ::= ! TERM : TYPE-IDENTIFIER
PID                       ::= sender
                            | PID-IDENTIFIER [ : PID-TYPE-IDENTIFIER ]
TYPED-VARIABLE-IDENTIFIER ::= VARIABLE-IDENTIFIER : TYPE-IDENTIFIER
```

```
GUARD                  ::= boolean-term
TERM                   ::= algebraic-term
SIMPLE-STATE-IDENTIFIER ::= identifier
CONDITION-IDENTIFIER   ::= identifier
GATE-IDENTIFIER        ::= identifier
VARIABLE-IDENTIFIER    ::= identifier
TYPE-IDENTIFIER        ::= identifier
PID-IDENTIFIER         ::= identifier
PID-TYPE-IDENTIFIER    ::= identifier
```

Un `identifier` est un identifiant construit à partir de lettres et de chiffres. `self` est un identifiant particulier qui sert à dénoter la valeur du type d'intérêt (type algébrique associé à une vue). Un `algebraic-term` correspond à un terme dans une spécification algébrique. Un `boolean-term` correspond à un terme qui s'évaluerait dans les booléens. Il peut s'agir d'une proposition ou d'un prédicat, ainsi que de termes construits avec des opérateurs infixes de conjonction ($\wedge$) ou de disjonction ($\vee$).

A.1.2 Formules de logique temporelle

Il existe des formules d'état et des formules de transition.

Pour rappel, la définition des formules d'état est :

$$\Phi ::= c \mid x \mid \exists x.\Phi_1 \mid true \mid false \mid \neg\Phi_1 \mid \Phi_1 \wedge \Phi_2 \mid -\Psi_1 \mid i.\Phi_1$$

La définition des formules de transition est :

$$\Psi ::= [g]l[g'] \mid x \mid \exists x.\Psi_1 \mid true \mid false \mid \neg\Psi_1 \mid \Psi_1 \wedge \Psi_2 \mid > \Phi_1 \mid i.\Psi_1$$

Leur syntaxe abstraite des formules d'état et celle des formules de transition sont les suivantes :

```
STATE-FORMULA      ::= INDEXED-IDENTIFIER
                     | ∃ identifier . STATE-FORMULA
                     | true
                     | false
                     | ¬ STATE-FORMULA
                     | STATE-FORMULA ∧ STATE-FORMULA
                     | ( STATE-FORMULA )
                     | - TRANSITION-FORMULA
                     | INDEXED-IDENTIFIER . STATE-FORMULA
TRANSITION-FORMULA ::= TRANSITION-LABEL
                     | ∃ identifier . TRANSITION-FORMULA
                     | true
                     | false
                     | ¬ TRANSITION-FORMULA
                     | TRANSITION-FORMULA ∧ TRANSITION-FORMULA
                     | ( TRANSITION-FORMULA )
                     | > STATE-FORMULA
                     | INDEXED-IDENTIFIER . TRANSITION-FORMULA
```

Le non-terminal `INDEXED-IDENTIFIER` correspond aux identifiants indexés, leur syntaxe est la suivante :

```
INDEXED-IDENTIFIER ::= identifier
                     | identifier . INDEXED-IDENTIFIER
                     | RANGE-IDENTIFIER
                     | RANGE-IDENTIFIER . INDEXED-IDENTIFIER
RANGE-IDENTIFIER   ::= identifier : [ RANGE-LIMIT .. RANGE-LIMIT ]
RANGE-LIMIT        ::= identifier
```

A.1.3 Vues

La représentation textuelle d'une vue est donnée dans la figure A.1.

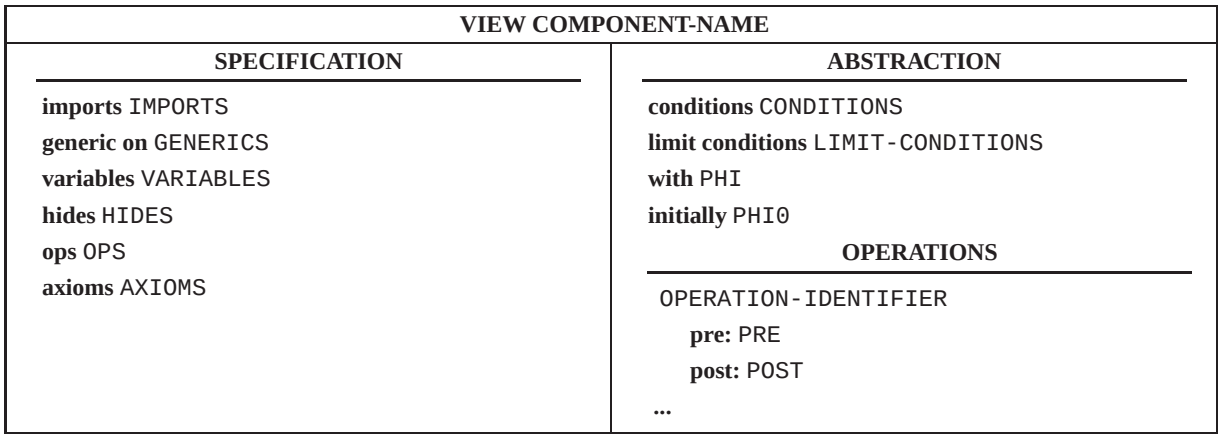


Figure A.1 : Notation textuelle KORRIGAN– Vue

Il existe aussi une syntaxe alternative, utilisant un STS au lieu des parties ABSTRACTION et OPERATIONS, donnée dans la figure A.2.

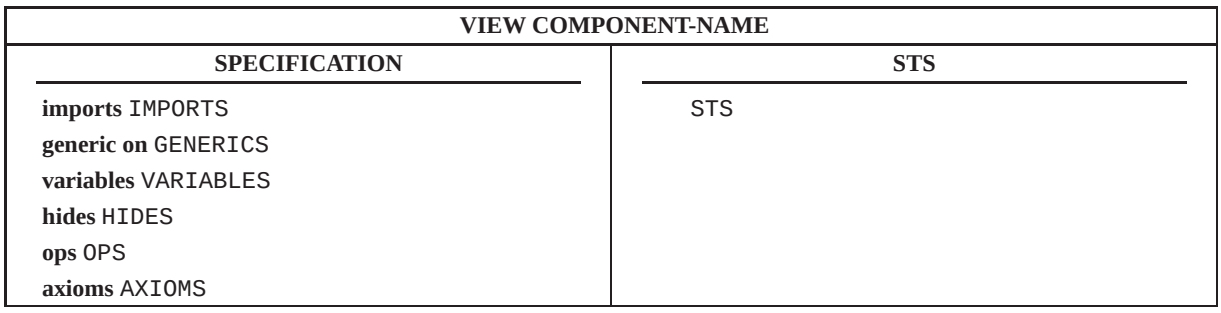


Figure A.2 : Notation textuelle KORRIGAN– Vue (syntaxe alternative)

La grammaire abstraite correspondante est donnée dans la suite. La syntaxe abstraite des STS a été vue plus haut.

```
VIEW                   ::= VIEW COMPONENT-NAME VIEW-SYNTAX
                         | VIEW COMPONENT-NAME ALTERNATIVE-VIEW-SYNTAX
VIEW-SYNTAX            ::= SPECIFICATION-WITH-OP ABSTRACTION OPERATIONS
ALTERNATIVE-VIEW-SYNTAX ::= SPECIFICATION-WITH-OP STS
COMPONENT-NAME         ::= identifier
```

La syntaxe de la partie SPECIFICATION est :

```
SPECIFICATION-WITH-OP ::= SPECIFICATION [ HIDES-DECL ] [ OPS-DECL ] [ AXIOMS-DECL
                          ]
SPECIFICATION         ::= SPECIFICATION [ IMPORTS-DECL ] [ GENERICS-DECL ] [
                          VARIABLES-DECL ]
IMPORTS-DECL          ::= imports IMPORTS
GENERICS-DECL         ::= generic on GENERICS
VARIABLES-DECL        ::= variables VARIABLES
HIDES-DECL            ::= hides HIDES
OPS-DECL              ::= ops OPS
AXIOMS-DECL           ::= axioms AXIOMS
IMPORTS               ::= IMPORT ( , IMPORT )*
GENERICS              ::= GENERIC-IDENTIFIER ( , GENERIC-IDENTIFIER )*
VARIABLES             ::= TYPED-VARIABLE-IDENTIFIER ( , TYPED-VARIABLE-IDENTIFIER
                          )*
HIDES                 ::= HIDE-IDENTIFIER ( , HIDE-IDENTIFIER )*
OPS                   ::= SIGNATURE ( , SIGNATURE )*
SIGNATURE             ::= STATIC-SIGNATURE
                        | DYNAMIC-SIGNATURE
STATIC-SIGNATURE      ::= OPERATION-IDENTIFIER [ SIGNATURE-INS ] -> TYPE-
                          IDENTIFIER
SIGNATURE-INS         ::= TYPE-IDENTIFIER ( , TYPE-IDENTIFIER )*
DYNAMIC-SIGNATURE     ::= OFFER-TERM
AXIOMS                ::= AXIOM (; AXIOM )*
IMPORT                ::= from identifier import IMPORT-ELEMENTS
IMPORT-ELEMENTS       ::= ALL
                        | IMPORT-IDENTIFIER (, IMPORT-IDENTIFIER )*
GENERIC-IDENTIFIER    ::= identifier
HIDE-IDENTIFIER       ::= identifier
IMPORT-IDENTIFIER     ::= identifier
OPERATION-IDENTIFIER  ::= identifier
AXIOM                 ::= conditional-axiom
```

La syntaxe de la partie ABSTRACTION est :

```
ABSTRACTION           ::= ABSTRACTION [ CONDITIONS-DECL ] [ LIMIT-CONDITIONS-DECL
                          ] [ WITH-DECL ] [ INITIALLY-DECL ]
CONDITIONS-DECL       ::= conditions CONDITIONS
LIMIT-CONDITIONS-DECL ::= limit conditions LIMIT-CONDITIONS
WITH-DECL             ::= with PHI
INITIALLY-DECL        ::= initially PHI0
CONDITIONS            ::= CONDITION-NAME ( , CONDITION-NAME )*
LIMIT-CONDITIONS      ::= LIMIT-CONDITION ( , LIMIT-CONDITION )*
PHI                   ::= STATE-FORMULA ( , STATE-FORMULA )*
PHI0                  ::= STATE-FORMULA
LIMIT-CONDITION       ::= boolean-term
```

Enfin, la syntaxe de la partie OPERATIONS est :

```
OPERATIONS   ::= OPERATIONS OPERATION*
OPERATION    ::= SIGNATURE pre PRE post POST
PRE          ::= PRE-ELEMENT*
POST         ::= POST-ELEMENT*
PRE-ELEMENT  ::= CONDITION-NAME : boolean-term
POST-ELEMENT ::= CONDITION-NAME : boolean-term
```

A.1.4 Vues statiques

La représentation textuelle des vues statiques est donnée dans la figure A.3.

STATIC VIEW COMPONENT-NAME	
SPECIFICATION	**ABSTRACTION**
imports IMPORTS	**conditions** CONDITIONS
generic on GENERICS	**limit conditions** LIMIT-CONDITIONS
variables VARIABLES	**with** PHI
hides HIDES	**initially** PHI0
ops OPS	**OPERATIONS**
axioms AXIOMS	OPERATION-IDENTIFIER **pre:** PRE **post:** POST **...**

Figure A.3 : Notation textuelle KORRIGAN– Vue statique

La syntaxe abstraite correspondante est :

```
STATIC-VIEW ::= STATIC VIEW
```

Elle réutilise celle des vues à une exception près. La règle :

```
SIGNATURE ::= STATIC-SIGNATURE
            | DYNAMIC-SIGNATURE
```

devient :

```
SIGNATURE ::= STATIC-SIGNATURE
```

A.1.5 Vues dynamiques

La représentation textuelle des vues dynamiques est donnée dans la figure A.4.
La syntaxe abstraite correspondante est :

```
DYNAMIC-VIEW ::= DYNAMIC VIEW
```

DYNAMIC VIEW COMPONENT-NAME	
SPECIFICATION	**ABSTRACTION**
imports IMPORTS	**conditions** CONDITIONS
generic on GENERICS	**limit conditions** LIMIT-CONDITIONS
variables VARIABLES	**with** PHI
hides HIDES	**initially** PHI0
ops OPS	**OPERATIONS**
axioms AXIOMS	OPERATION-IDENTIFIER **pre:** PRE **post:** POST **...**

Figure A.4 : Notation textuelle KORRIGAN– Vue dynamique

Elle réutilise celle des vues à une exception près. La règle :

```
SIGNATURE ::= STATIC-SIGNATURE
            | DYNAMIC-SIGNATURE
```

devient :

```
SIGNATURE ::= DYNAMIC-SIGNATURE
```

A.1.6 External Structuring Views

La représentation textuelle d'une ESV est donnée dans la figure A.5.

EXTERNAL STRUCTURING VIEW COMPONENT-NAME	
SPECIFICATION	**COMPOSITION COMPOSITION-TYPE**
imports IMPORTS	**is**
generic on GENERICS	COMPOSITION-ELEMENT
variables VARIABLES	COMPOSITION-ELEMENT
	...
	axioms AXIOMS
	with PHI, PSI
	initially PHI0

Figure A.5 : Notation textuelle KORRIGAN– External Structuring View

La grammaire abstraite correspondante est :

```
ES-VIEW                    ::= EXTERNAL STRUCTURING ES-VIEW0
ES-VIEW0                   ::= VIEW COMPONENT-NAME SPECIFICATION COMPOSITION
COMPOSITON                 ::= COMPOSITION COMPOSITION-TYPE COMPOSITION-ELEMENTS
                               COMPOSITION-GLUES
COMPOSITION-TYPE           ::= LOOSE
                             | ALONE
                             | KEEP
COMPOSITION-ELEMENTS       ::= is COMPOSITION-ELEMENT COMPOSITION-ELEMENT+
COMPOSITION-ELEMENT        ::= COMPOSITION-ELEMENT-NAME : COMPOSITION-ELEMENT-TYPE [
                               COMPOSITION-ELEMENT-PARAMS ]
COMPOSITION-ELEMENT-NAME   ::= INDEXED-IDENTIFIER
COMPOSITION-ELEMENT-PARAMS ::= [ COMPOSITION-ELEMENT-PARAM ( , COMPOSITION-ELEMENT-
                               PARAM )* ]
COMPOSITION-ELEMENT-PARAM  ::= VARIABLE-IDENTIFIER = TERM
                             | GENERIC-IDENTIFIER = TERM
COMPOSITION-GLUES          ::= [ AXIOM-GLUE ] [ TEMPORAL-GLUE ]
AXIOM-GLUE                 ::= axioms AXIOMS
TEMPORAL-GLUE              ::= [ GLUE-WITH ] [ GLUE-INITIALLY ]
GLUE-WITH                  ::= with PHI , PSI
GLUE-INITIALLY             ::= initially PHI0
PSI                        ::= { PSI-COUPLES }
PSI-COUPLES                ::= PSI-COUPLE ( , PSI-COUPLE )*
PSI-COUPLE                 ::= [ INDEXED-IDENTIFIER . ] ( TRANSITION-FORMULA ,
                               TRANSITION-FORMULA ) [ HIDDEN ]
COMPOSITION-ELEMENT-TYPE   ::= identifier
```

A.1.7 Integration Views

La représentation textuelle des Integration Views est donnée dans la figure A.6.

INTEGRATION VIEW COMPONENT-NAME	
SPECIFICATION	**COMPOSITION COMPOSITION-TYPE**
imports IMPORTS	**is**
generic on GENERICS	STATIC COMPOSITION-ELEMENT
variables VARIABLES	DYNAMIC COMPOSITION-ELEMENT
	axioms AXIOMS
	with PHI, PSI
	initially PHI0

Figure A.6 : Notation textuelle KORRIGAN– Integration View

La syntaxe abstraite correspondante est :

```
INTEGRATION-VIEW ::= INTEGRATION ES-VIEW0
```

Elle réutilise celle des ESV à une exception près. La règle :

```
COMPOSITION-ELEMENTS ::= is COMPOSITION-ELEMENT COMPOSITION-ELEMENT+
```

devient :

```
COMPOSITION-ELEMENTS ::= is STATIC COMPOSITION-ELEMENT DYNAMIC COMPOSITION-
                         ELEMENT
```

A.1.8 Composition Views

La représentation textuelle des Composition Views est donnée dans la figure A.7.

COMPOSITION VIEW COMPONENT-NAME	
SPECIFICATION	**COMPOSITION COMPOSITION-TYPE**
imports IMPORTS **generic on** GENERICS **variables** VARIABLES	**is** COMPOSITION-ELEMENT COMPOSITION-ELEMENT ... **axioms** AXIOMS **with** PHI, PSI **initially** PHI0

Figure A.7 : Notation textuelle KORRIGAN– Composition View

La syntaxe abstraite correspondante est :

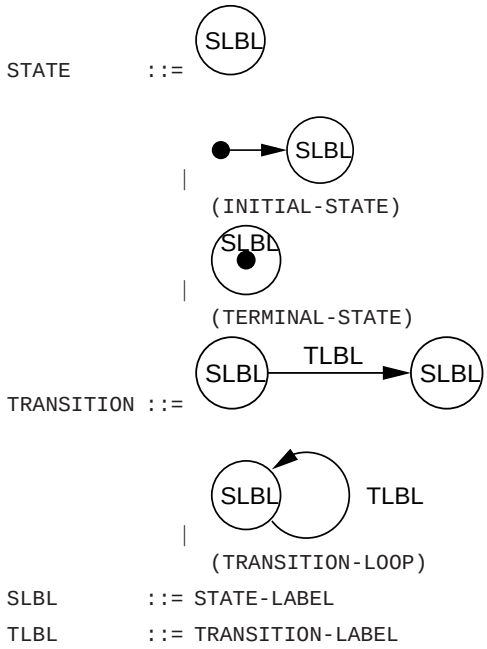
```
COMPOSITION-VIEW ::= COMPOSITION ES-VIEW0
```

Elle réutilise celle des ESV.

A.2 Notation graphique

A.2.1 Systèmes de transition symboliques

Les notations graphiques utilisées pour les STS sont les suivantes :



Les étiquettes sont décrites dans la section concernant les notations textuelles.

Il est possible de simplifier la représentation graphique d'un STS en utilisant les trois simplifications décrites dans les figures A.8, A.9 et A.10. Il ne s'agit que de simplifications syntaxiques qui ne changent en rien la sémantique des STS.

La première simplification (figure A.8) consiste à réunir les états qui sont accessibles à partir d'un même état source via deux transitions partageant la même étiquette.

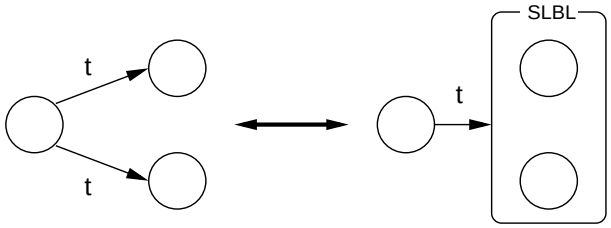


Figure A.8 : Notation graphique – simplification des STS (1)

La seconde simplification (figure A.9) procède à l'identique dans le cas de deux états sources arrivant, via deux transitions d'étiquettes identiques, dans un même état cible. Cette simplification a par exemple été utilisée dans les figures 2.12 et 2.37.

Enfin, la troisième simplification (figure A.10) consiste à réunir deux transitions ayant même source et même cible en une seule transition.

Une autre forme de simplification consiste à utiliser la notion d'activité (figure A.11). Il s'agit d'une notion syntaxique permettant de combiner plusieurs STS. Cette combinaison ne se fait pas de façon concurrente, par opposition à la composition parallèle de STS résultant d'une External Structuring View, mais de façon séquentielle.

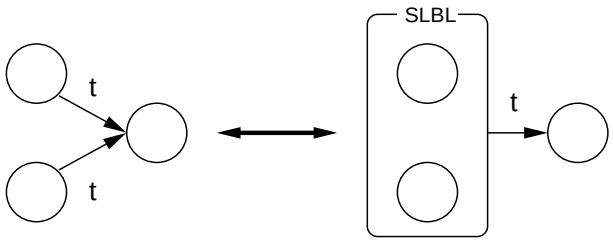


Figure A.9 : Notation graphique – simplification des STS (2)

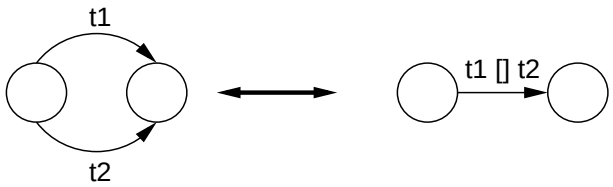


Figure A.10 : Notation graphique – simplification des STS (3)

A.2.2 Diagrammes d'interface et vues

Les diagrammes d'interface définissent l'interface des composants, et donc de l'ensemble des vues. La notation utilisée est présentée dans la figure A.12.

En ce qui concerne les vues, elles sont simplement représentées par une boîte portant le nom du composant correspondant et équipée de ses interfaces. Le cas échéant, ses aspects génériques (clauses `generic on` et `variables`) sont représentés dans un cadre non continu. La figure A.13 donne la notation correspondante.

La partie inférieure de la boîte peut être utilisée pour indiquer les axiomes du composant ainsi que les parties `ABSTRACTION` et `OPERATIONS` (figure A.14), ou le STS sous sa forme graphique (figure A.15).

Les vues dynamiques sont simplement représentées comme des vues. De façon à pouvoir les différencier des vues dynamiques, les vues statiques sont représentées avec un en-tête grisé (figure A.16).

Il est possible d'instancier les paramètres (`VARIABLES` et `GENERICS`) des vues. Pour cela on utilise la notation de la figure A.17. Ceci est fait au niveau des ESV (voir plus bas), qui définissent l'instanciation et utilisent le composant instancié.

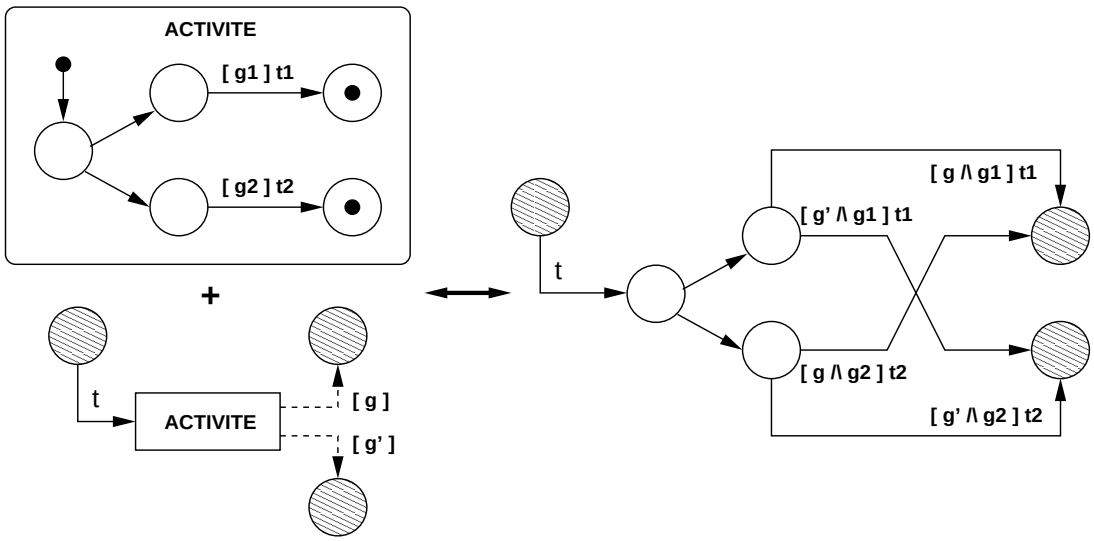


Figure A.11 : Notation graphique – STS et activités

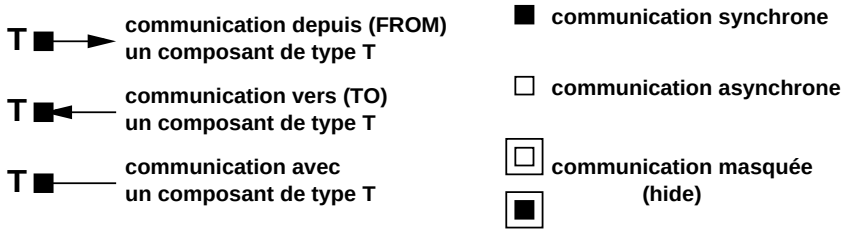


Figure A.12 : Notation graphique – Diagrames d'interfaces

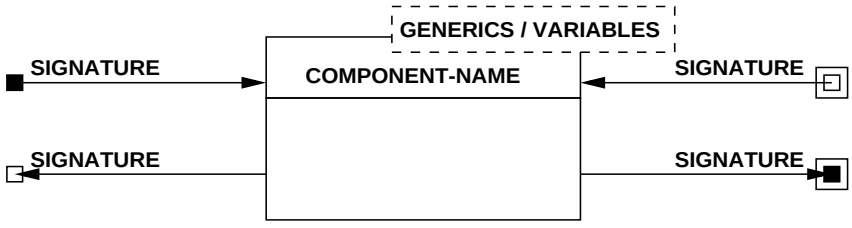


Figure A.13 : Notation graphique – Vues

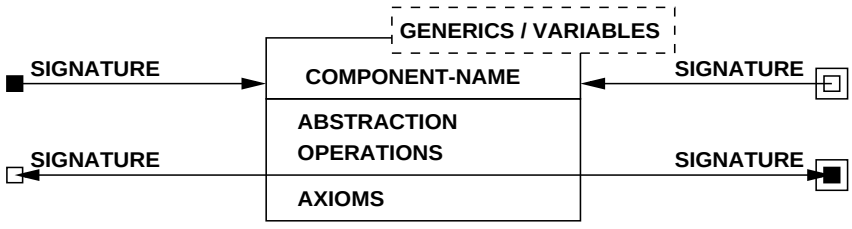


Figure A.14 : Notation graphique – Vues (complète)

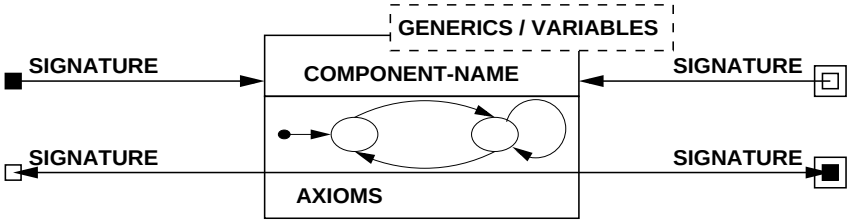


Figure A.15 : Notation graphique – Vues (complète, alternative)

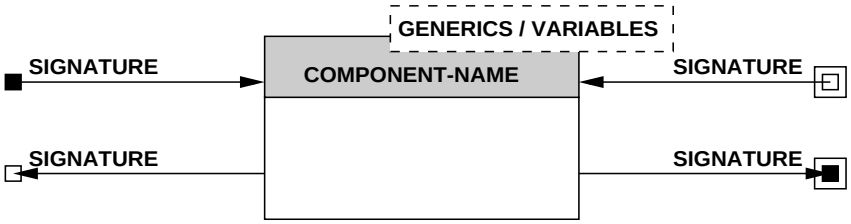


Figure A.16 : Notation graphique – Vues statiques

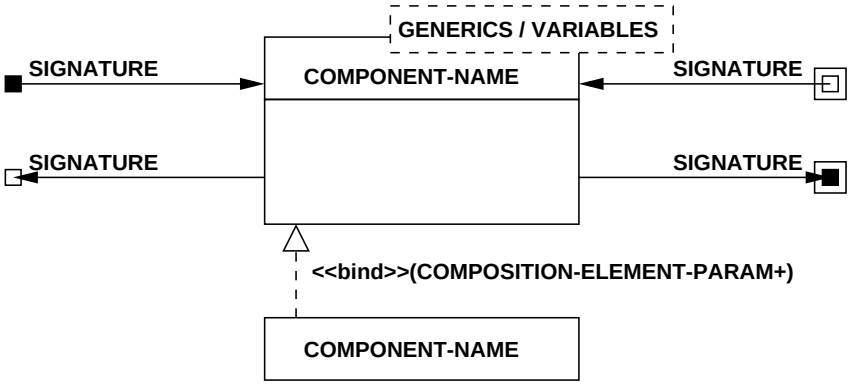


Figure A.17 : Notation graphique – Instanciation

A.2.3 Diagrammes de composition, de communication et ESV

Les diagrammes de composition servent à décrire la décomposition d'un composant en termes de sous-composants. On utilise pour cela une notation inspirée d'UML (figure A.18) où le super-composant est relié par des liens de composition à la représentation graphique des ses sous-composants. On ajoute au niveau de ces liens les informations concernant l'identification des sous-composants. L'instantiation se fait de la même façon que pour les vues simples.

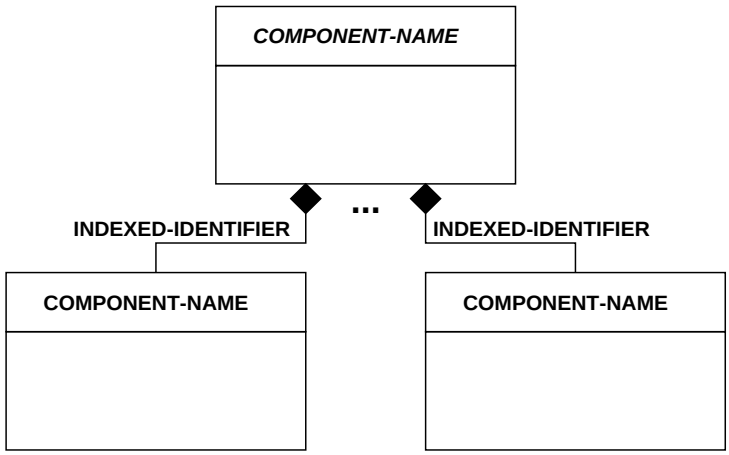


Figure A.18 : Notation graphique – Diagrames de composition

Une ESV est une structure abstraite (il n'existe pas concrètement d'ESV, uniquement des CV ou des IV). Concrètement, en ce qui concerne les diagrammes, il peut s'agir des cas suivants :

- le super-composant est une IV, et alors
  - il existe exactement deux sous-composants ;
  - l'un des sous-composant est une vue statique ;
  - l'autre est une vue dynamique.
- le super-composant est une CV, et alors
  - il exite plusieurs sous-composants ;
  - certains sont des IV ;
  - les autres sont des CV.

KORRIGAN unifie la composition (CV) et l'intégration d'aspects (IV) par la définition de l'abstraction ESV. Concrètement encore une fois, il est possible d'utiliser une notation spéciale dans le cas des IV (figure A.19) qui est équivalente à celle vue de la figure A.20, mais qui rend mieux compte du principe de masquage de l'information.

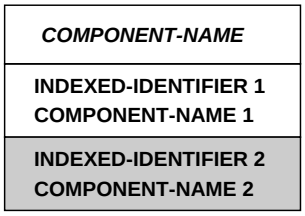


Figure A.19 : Notation graphique – Diagrames de composition (IV, 1)

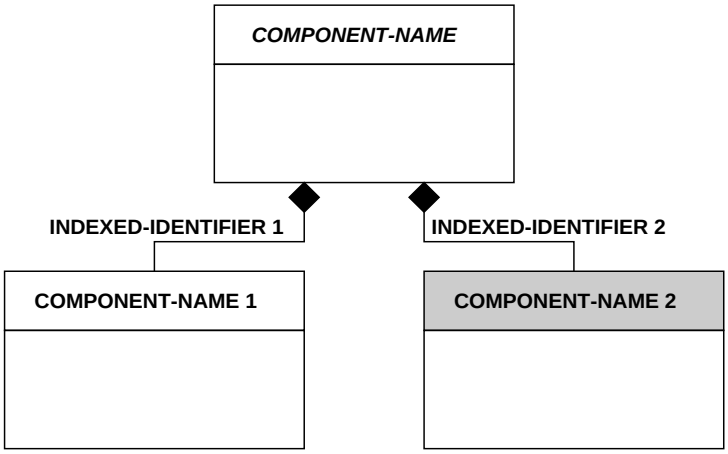


Figure A.20 : Notation graphique – Diagrames de composition (IV, 2)

L'utilisation des diagrammes de composition intervient dans le cadre de la méthode définie dans le chapitre 3. Il a été vu que ces diagrammes ne correspondent qu'à une partie (clauses `imports` et `is`) des ESV. Une étape complémentaire consiste à ajouter l'information concernant la communication entre les sous-composants au niveau d'un diagramme de composition. Le diagramme devient alors un diagramme de communication. Comme le montre la figure A.21, une partie de la colle de communication est placée dans l'ESV du super-composant. Chacun des couples de `PSI` (*i.e.* Ψ dans la colle des vues) est représenté graphiquement comme un lien entre les sous-composants concernés.

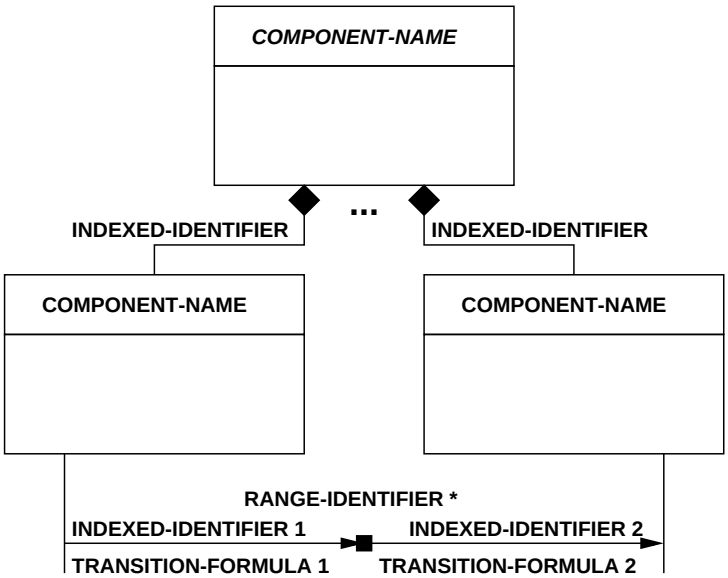


Figure A.21 : Notation graphique – Diagrames de communication

L'élément de Ψ correspondant à la figure A.21 est (modulo l'opérateur de range qui n'est que syntaxique) :

```
RANGE-IDENTIFIER*.(
```

```
INDEXED-INDENTIFIER 1.TRANSITION-FORMULA 1,
INDEXED-INDENTIFIER 2.TRANSITION-FORMULA 2).
```

Ψ contient des couples. Toutefois, au niveau de la notation graphique, il est possible de réunir plus de deux interfaces : si on a $(i_1.a, i_2.b)$ et $(i_1.a, i_3.c)$ (modulo le fait que la présence de (x, y) dans Ψ est équivalent à celle de (y, x)), il est possible de créer un lien unique correspondant à $(i_1.a, i_2.b, i_3.c)$.

Il faut remarquer que, dans les première étapes de décomposition de la méthode (*i.e.* avant la décomposition finale des composants en aspect statique et aspect dynamique), les ESV sont représentées de façon temporaire avec une forme textuelle de leur partie statique (ESV et partie dynamique des composants l'ESV sont alors confondues). La figure A.22 montre cette notation (un exemple en est la figure 3.13 du chapitre 3). Ici la partie dynamique est implicitement assimilée à l'ESV.

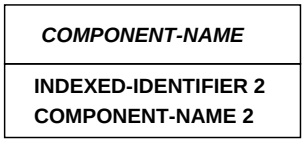


Figure A.22 : Notation graphique – Diagrames de composition (IV, représentation temporaire)

En fait, la notation finale (après toutes les étapes de décomposition, y compris celle correspondant à la séparation des aspects), est celle de la figure A.18 ou bien celle de la figure A.20.

ANNEXE B

Étude de cas : gestionnaire de mots de passe

B.1 Description de l'étude de cas

Le système est composé de plusieurs *utilisateurs de base*, d'un *utilisateur privilégié* (root) et du gestionnaire de mots de passe. Les utilisateurs de base peuvent modifier leur mot de passe. L'utilisateur privilégié peut agir comme un utilisateur de base (et changer son mot de passe), mais peut aussi créer des comptes utilisateurs (*i.e.* ajouter un identifiant d'utilisateur et un mot de passe lui correspondant).

La création d'un compte utilisateur se passe comme suit :

1. l'identifiant de l'utilisateur est demandé puis
   (a) soit il existe déjà, et une erreur est signalée ;
   (b) soit il n'existe pas, et le mot de passe est demandé.
2. si aucune erreur n'est survenue, le mot de passe est demandé à nouveau puis
   (a) soit il est différent du mot de passe précédemment entré, et une erreur est signalée ;
   (b) soit il est identique, et le compte est crée (*i.e.* le couple formé par l'identifiant d'utilisateur et le mot de passe est enregistré).

La modification d'un mot de passe par un utilisateur se passe comme pour la création, mis à part les deux différences suivantes :

– l'identifiant de l'utilisateur doit déjà exister ;
– l'ancien mot de passe est démandé (et vérifié) avant la partie concernant la saisie du nouveau mot de passe.

Il est possible de modéliser des systèmes plus complexes, avec par exemple des restrictions concernant le mot de passe. Nous verrons plus bas, comment les prendre en compte.

B.2 Spécification du gestionnaire de mots de passe

Nous donnons ici la spécification complète, en KORRIGAN, du gestionnaire de mots de passe.

B.2.1 Vue statique du gestionnaire de mots de passe

La vue statique du gestionnaire de mots de passe en KORRIGAN est donnée dans la figure B.1. Nous avons fait ici le choix de la définir "from scratch". Un autre approche aurait été de la définir comme une

instanciation d'une vue statique représentant les ensembles (voir l'annexe C pour un exemple de cette approche).

STATIC VIEW StaticPasswordManager (SPM)

SPECIFICATION

imports Boolean, UserId, Pwd

ops

empty : → SPM
add : SPM, UserId, Pwd → SPM
modify : SPM, UserId, Pwd → SPM
declared : SPM, UserId → Boolean
correct : SPM, UserId, Pwd → Boolean

axioms

modify(empty,u,p) == add(empty,u,p);
(u = u2) ⇒ modify(add(spm,u2,p2),u,p) == add(spm,u,p);
not(u = u2) ⇒ modify(add(spm,u2,p2),u,p) == add(modify(spm,u,p),u2,p2);
declared(empty,u) == false;
(u = u2) ⇒ declared(add(spm,u2,p2),u) == true;
not(u = u2) ⇒ declared(add(spm,u2,p2),u) == declared(spm,u);
correct(empty,u,p) == false;
(u=u2) ⇒ correct(add(spm,u2,p2),u,p) == (p = p2);
not(u=u2) ⇒ correct(add(spm,u2,p2),u,p) == correct(spm,u,p);

STS

• → X
X − add ?u : UserId ?p : Pwd → X
X − modify ?u : UserId ?p : Pwd → X
X − declared ?u : UserId !b : Boolean → X
X − correct ?u : UserId ?p : Pwd !b : Boolean → X

Figure B.1 : Gestionnaire de mots de passe – Vue Statique

Ici le STS n'a qu'un état puisque les opérations sont toujours possibles et qu'il n'y a donc pas de conditions (partie `ABSTRACTION` des vues). Nous donnons toutefois quand même une représentation en terme de STS de ce type de données car c'est cela qui va nous permettre, dans la suite, d'intégrer les deux aspects du gestionnaire de mots de passe (par collage de transitions entre autres).

La figure B.2 donne la représentation graphique du STS de la vue statique du gestionnaire de mots de passe.

B.2.2 Vue dynamique du gestionnaire de mots de passe

Le protocole de communication du gestionnaire de mots de passe, *i.e.* sa vue dynamique, peut être vu comme constitué de deux activités distinctes (voir l'annexe A pour la définition et la notation relatives aux activités) : la création et la modification de mots de passe.

La figure B.3 donne une représentation graphique du STS de la vue dynamique du gestionnaire de mots de passe.

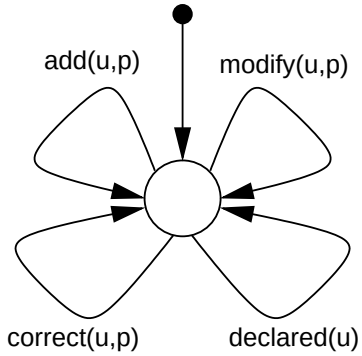


Figure B.2 : Gestionnaire de mots de passe – STS statique

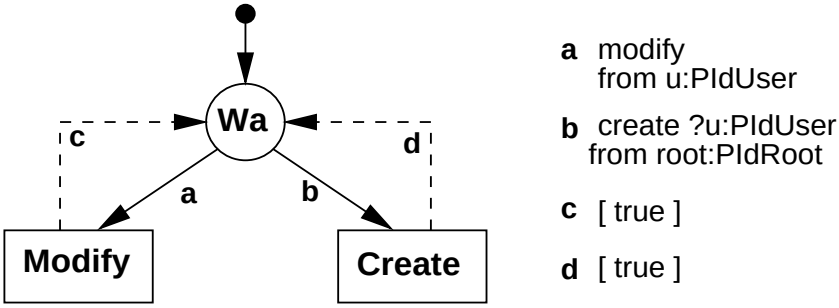


Figure B.3 : Gestionnaire de mots de passe – STS dynamique (activités)

Activité de modification

La figure B.4 donne la représentation graphique de l'activité de modification de mot de passe.

Activité de création

La figure B.5 donne la représentation graphique de l'activité de création de mot de passe.

Intégration des deux activités

A partir de la définition de la notation des activités (annexe A), il est possible de calculer le STS représentant le protocole de communication du gestionnaire de mots de passe. Ce STS intègre les STS correspondant aux deux activités et est donné dans la figure B.6.

Ici, il est possible de prendre en compte des améliorations de l'étude de cas, comme par exemple d'imposer des restrictions sur le mot de passe (6 caractères minimum, etc.). Pour cela il suffit de :

1. définir dans le type de données amorphe (*i.e.* n'étant pas défini par une vue) `Pwd` une opération `correct` prenant en compte les nouvelles restrictions ;
2. modifier les transitions de la partie dynamique du gestionnaire (dans les deux activités) et rajouter une garde à chaque réception de mot de passe pour imposer le respect des restrictions : la transition `[MValid(u)] getPwd?p0:Pwd from sender` entre les états `BeM` et `Gp1` devenant ainsi par exemple `[MValid(u)] getPwd?p0:Pwd from sender [correct(p0)]`.

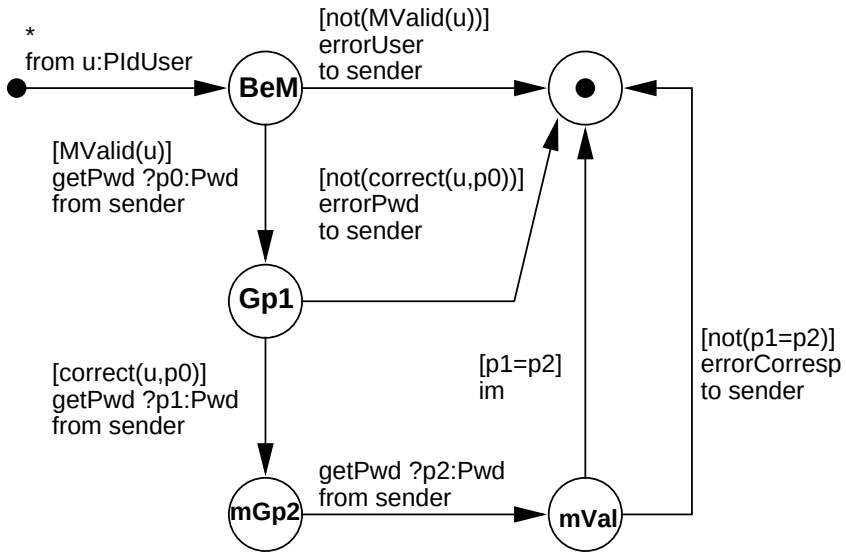


Figure B.4 : Gestionnaire de mots de passe – Activité de modification de mot de passe

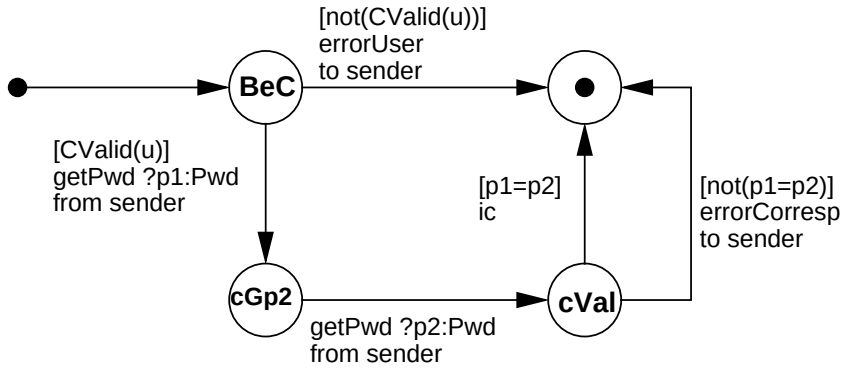


Figure B.5 : Gestionnaire de mots de passe – Activité de création de mot de passe

B.2.3 Vue d'intégration des aspects du gestionnaire de mots de passe

Dans un dernier temps, il faut décrire comment les aspects statique et dynamique du gestionnaire sont reliés, *i.e.* donner la vue d'intégration correspondante. Cette vue d'intégration (en KORRIGAN) est donnée dans la figure B.7.

La colle entre les deux vues (`SPM` et `DPM`) constituant le gestionnaire de mots de passe est composée de deux parties : une première permettant de faire correspondre les gardes abstraites (`MValid(u)`, `CValid(u)` et `correct(u,p)` qui n'ont pas été définies par des axiomes) de la vue dynamique et certaines opérations de la vue statique (axiomes), et une seconde décrivant comment sont collées les transitions des STS des deux vues (formules de logique temporelle).

La première partie de la colle est définie dans la clause `axioms`. Le premier axiome correspond à la condition de validité d'un identifiant d'utilisateur lors d'une modification de mot de passe. Cet identifiant, réceptionné à travers la partie dynamique doit être déclaré dans la partie statique. On fait donc correspondre la garde `MValid(u)` de la partie dynamique (on ajoute pour cela dans les axiomes

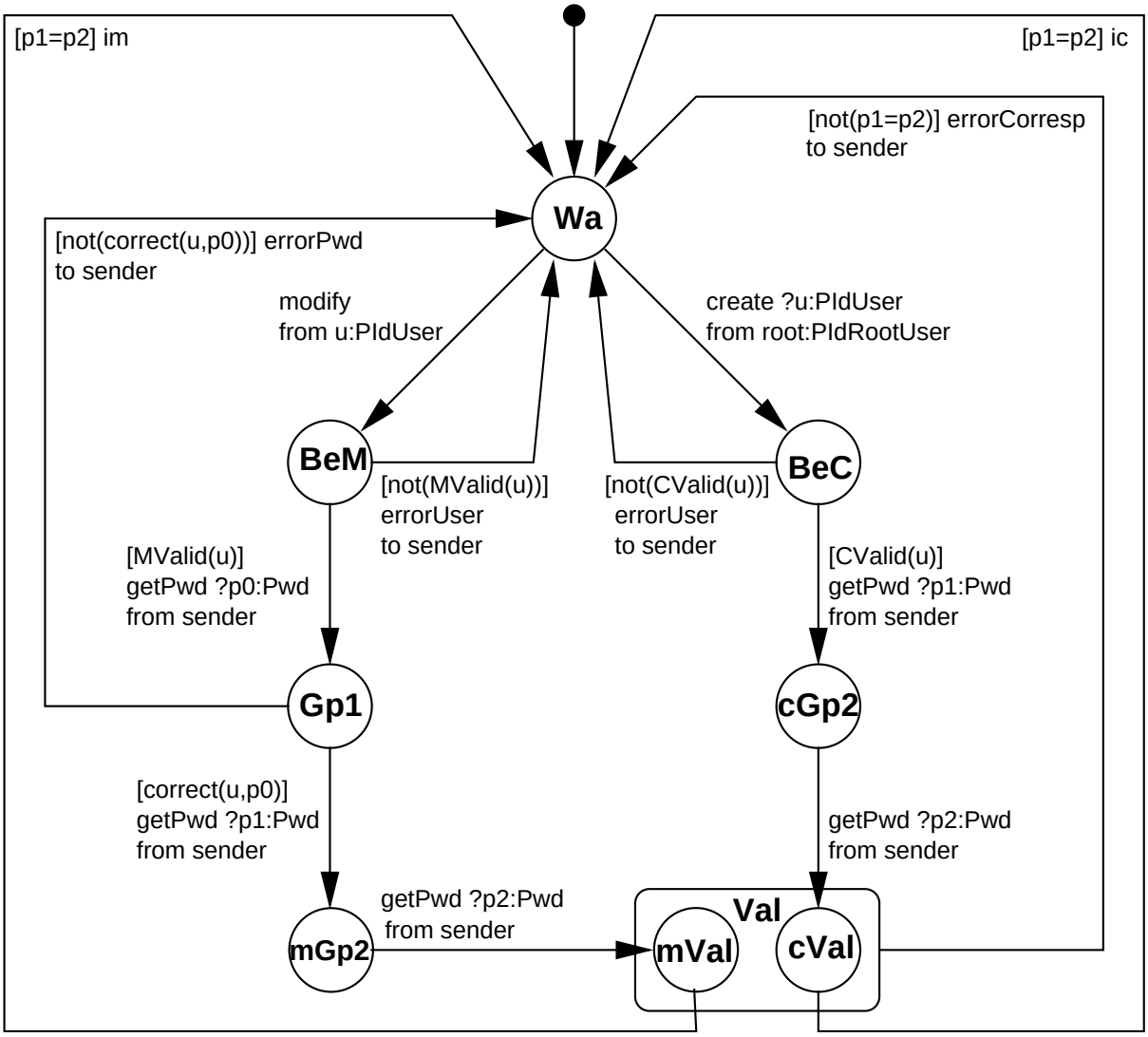


Figure B.6 : Gestionnaire de mots de passe – STS dynamique (intégration des activités)

les paramètres `self` qui étaient implicites) avec le résultat de l'opération `declared(self ,u)`.

Le second axiome fait de même en ce qui concerne la création d'un mot de passe. L'identifiant réceptionné de doit pas être déclaré (`not declared(self, u)`) dans la partie statique.

Le dernier axiome spécifie que la condition de correction (`correct`, entre un identifiant d'utilisateur et un mot de passe) utilisée dans la vue dynamique correspond à celle utilisée (`correct`) dans la vue statique.

La seconde partie de la colle est définie dans la clause `with`. Son rôle est de faire correspondre les transitions (cachés) `im` et `ic` de la vue dynamique avec (respectivement) les transitions de la vue statique correspondant à la modification (`modify`) et l'ajout (`add`) d'un mot de passe. La correspondance entre les paramètres est obtenue à travers la colle.

INTEGRATION VIEW Password Manager

COMPOSITION ALONE

is

STATIC s: SPM

DYNAMIC d: RDPM

axioms

d.MValid(s,u) == s.declared(s,u)

d.CValid(s,u) == not(s.declared(s,u))

d.correct(s,u,p) == s.correct(s,u,p)

with true, {

(d.([true] im), s.(modify(d.u,d.p1))),

(d.([true] ic), s.(add(d.u,d.p1)))

}

initially true

Figure B.7 : Gestionnaire de mots de passe – Vue d'intégration

Le STS global correspondant à cette intégration est donné dans la figure 2.37 du chapitre 2.

ANNEXE C

Étude de cas : messagerie

C.1 Description de l'étude de cas

La société EasyPhone désire développer un service de messagerie téléphonique. Ce système doit permettre à ses clients de communiquer via un serveur central. Les exigences sont les suivantes.

C.1.1 Clients

Un client ou *utilisateur* est identifié par un numéro personnel à dix chiffres. Un client peut avoir deux statuts : il peut être simple ou abonné.

Lorsqu'une communication se met en place entre deux clients, l'*appelant* est celui qui a demandé la communication et l'*appelé* est l'autre client. Un *appelé potentiel* est un client pouvant être appelé par un appelant abonné.

Les activités des clients dépendent du fait qu'ils soient simples ou abonnés.

Client simple.

1. Il peut, lorsqu'il n'est pas en communication, demander au serveur de devenir abonné.
2. Il peut être appelé par le serveur lorsqu'un appelant abonné en fait la demande.
3. Il peut refuser de prendre une communication.
4. Il peut accepter une communication en provenance d'un appelant abonné.
5. Il peut interrompre une communication en cours.

Client abonné.

Un client abonné a les mêmes droits qu'un client simple, mais aussi :

1. Il peut, lorsqu'il n'est pas en cours ou en demande de communication, demander au serveur d'annuler son abonnement.
2. Il peut demander au serveur d'enregistrer un nouveau client comme l'un de ses appelés potentiels.
3. Il peut demander au serveur de supprimer un client de la liste de ses appelés potentiels.
4. Il peut demander au serveur une communication avec un autre client.
5. Il peut interrompre une communication en cours.

C.1.2 Serveur central

Le serveur central est composé de quatre unités : une unité de connexion, une unité de gestion, une unité de déconnexion et une base de données.

Unité de connexion.

1. Elle s'occupe des demandes de connexion.
2. Elle envoie une requête à l'appelé pour lui signaler qu'une demande de connexion a été faite.
3. Elle établit la connexion entre l'appelant et l'appelé.

Unité de gestion.

1. Elle gère toutes les demandes de changement de statut (client simple / client abonné).
2. Elle gère les demandes de mise à jour de la liste des appelés potentiels des clients abonnés.

Unité de déconnexion.

1. Elle s'occupe des demandes de déconnexion.
2. Elle peut décider d'elle-même d'interrompre une connexion.

Unité de base de données.

1. La base de données contient l'ensemble des numéros d'abonnés, et pour chaque abonné l'ensemble des numéros de ses appelés potentiels.
2. Elle prend en compte les changements de statut des clients, et toute modification éventuelle des listes d'appelés potentiels.
3. Elle dispose d'informations sur les communications en cours, l'appelant et l'appelé.

Une communication entre un appelant i et un appelé j est possible quand :

1. L'appelant est abonné (mais ce n'est pas nécessaire pour l'appelé).
2. $i \neq j$.
3. j doit être enregistré en tant qu'appelé potientiel de i.
4. Une communication entre i et j n'est pas déjà en cours.
5. i a demandé une communication vers j.

C.2 Spécification de la messagerie

Nous donnons dans cette annexe la spécification KORRIGAN complète de la messagerie. La messagerie est spécifiée en suivant une méthode légèrement différente de celle que nous avons développée pour les spécifications mixtes (chapitre 3). Ici nous définissons d'abord quelques spécifications de base (usuelles). Ensuite, nous spécifions les utilisateurs de la messagerie : d'abord le comportement de clients de base (*i.e.* clients simples sans possibilité d'abonnement), puis le comportement de clients complets (simples, abonnés) en utilisant le principe d'héritage de KORRIGAN. Enfin, nous spécifions chacune des unités du serveur central, leur composition (le serveur), et l'ensemble du système de messagerie (figure

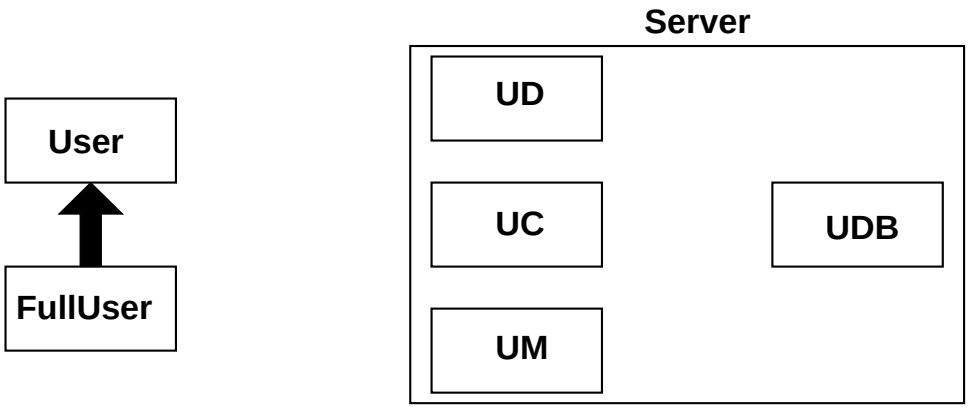


Figure C.1 : Messagerie – Composants

STATIC VIEW S-EMPTY	
SPECIFICATION	**STS**
	• → IDLE

Figure C.2 : `EMPTY` – Vue statique

C.1) c'est-à-dire la composition du serveur et de ses utilisateurs. Les communications entre ces composants seront étudiées dans les figures C.26 et C.27.

C.2.1 Spécifications de base

Nous donnons d'abord les spécifications de quelques éléments de base qui sont d'un usage courant : la vue (statique) vide, une vue statique (générique) pour les ensembles, et la spécification algébrique des couples. Aucune de ces spécifications n'est spécifique à cette étude de cas, c'est pourquoi nous les regroupons sous le terme "spécifications de base".

Vue statique `EMPTY`.

Cette vue (figure C.2) est utilisée lorsque les composants n'ont pas de partie statique à proprement parler. Elle est construite à partir d'un unique état qui est initial, `IDLE`. Elle n'a aucune transition.

Vue statique des ensembles.

La figure C.3 donne la vue statique des ensembles en KORRIGAN. Les signatures des opérations et les axiomes des ensembles y sont définis. Les ensembles sont abstraits comme étant vides ou pas (en utilisant une condition `empty`), puis les préconditions et les postconditions des opérations sont définies en exprimant leur effet sur la condition `empty`. Les noms de conditions simples (sans primes) utilisés dans les préconditions et les postconditions correspondent aux valeurs des conditions avant l'application des opérations. Les noms de conditions primés correspondent aux valeurs des conditions après l'application des opérations. Les axiomes peuvent être obtenus *à partir* du STS de la figure C.4 en utilisant les principes de dérivation présentés dans le chapitre 3.

STATIC VIEW SET

SPECIFICATION

imports NAT
generic on T
ops
new : → SET
add : SET × T → SET
member : SET × T → BOOL
size : SET → NAT
nextEmpty : SET → BOOL
remove : SET × T → SET
empty : SET × BOOL
axioms
nextEmpty(s) == size(s)=1;
member(new,e) == false;
(e1=e) ⇒ member(add(s,e1),e)
== true;
not(e1=e) ⇒ member(add(s,e1),e)
== member(s,e);
size(new) == 0;
size(add(s,e)) == size(s)+1;
remove(new,e) == new;
(e1=e) ⇒ remove(add(s,e1))
== remove(s,e);
not(e1=e) ⇒ remove(add(s,e1))
== add(remove(s,e),e1);

ABSTRACTION

conditions empty
limit conditions nextEmpty
initially empty

OPERATIONS

member, size
pre: true
post: {empty':empty}
add
pre: true
post: {empty':false}
remove
pre: ¬ empty ∧ member(e)
post: {empty':nextEmpty}

Figure C.3 : SET – Vue statique

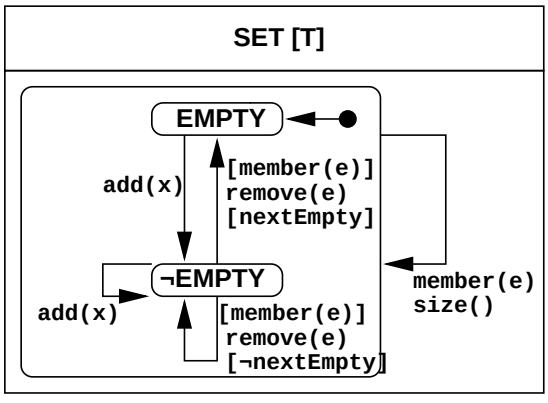


Figure C.4 : SET – STS statique

Couple	
generic on T	
imports BOOL	
ops	
(_ , _):T × T → Couple	; couple constructor
_ = _: Couple → BOOL	; equality of couples
p1 : Couple → T	; first element selector
p2 : Couple → T	; second element selector
axioms	
(a,b) = (c,d) == (a=c) ∧ (b=d);	
p1((a,b)) == a;	
p2((a,b)) == b;	

Figure C.5 : `Couple` – Spécification algébrique

La figure C.4 donne une représentation graphique de la partie STS (calculable à partir des parties ABSTRACTION et OPERATIONS) de la vue statique des ensembles. Les arguments des opérations correspondant au composant (*type d'intérêt*) sont implicites, et sont omis. Puisque certaines opérations (`member` et `size`) sont définies sur chaque état et ne modifient pas la valeur des conditions (de telles opérations sont appelées *observateurs non conditionnés*), nous utilisons un super-état de façon à simplifier la représentation graphique du STS.

Spécification algébrique des couples.

La figure C.5 donne la spécification algébrique des couples (génériques). Cette spécification est par exemple utilisée pour spécifier les couples d'utilisateurs en communication. Les opérations `p1` et `p2` correspondent (respectivement) à la première et à la seconde projection des couples.

C.2.2 Utilisateurs de base

Vue dynamique des utilisateurs de base.

La figure C.6 donne la vue dynamique des utilisateurs de base en KORRIGAN.

La figure C.7 donne une représentation graphique de la partie STS de la vue dynamique des utilisateurs de base.

Vue d'intégration des utilisateurs de base.

Les utilisateurs de base n'ont pas de vue statique (vue statique vide). La figure C.8 donne la vue d'intégration des utilisateurs de base en KORRIGAN.

C.2.3 Utilisateurs complets

Vue dynamique des utilisateurs complets.

La figure C.9 donne la vue dynamique des utilisateurs complets en KORRIGAN. Cette vue hérite de la vue dynamique des utilisateurs de base. Elle peut donc uniquement ajouter des nouvelles conditions,

DYNAMIC VIEW D-BASIC-USER

SPECIFICATION

imports PId, MSG
generic on
server:PIdServer
ops
called ?p:PIdUser **from** server
ok **to** server
nok **to** server
comm ?m:Msg **from** PIdUser
comm !m:Msg **to** PIdUser
eot **to** server
eot **from** server

STS

• → IDLE
IDLE – called ?p:PIdUser **from** server
→ASKED
ASKED – ok **to** server→COMM
ASKED – nok **to** server→IDLE
COMM – comm ?m:Msg **from** p→COMM
COMM – comm !m:Msg **to** p→COMM
COMM – eot **to** server→IDLE
COMM – eot **from** server→IDLE

Figure C.6 : Utilisateurs de base – Vue dynamique

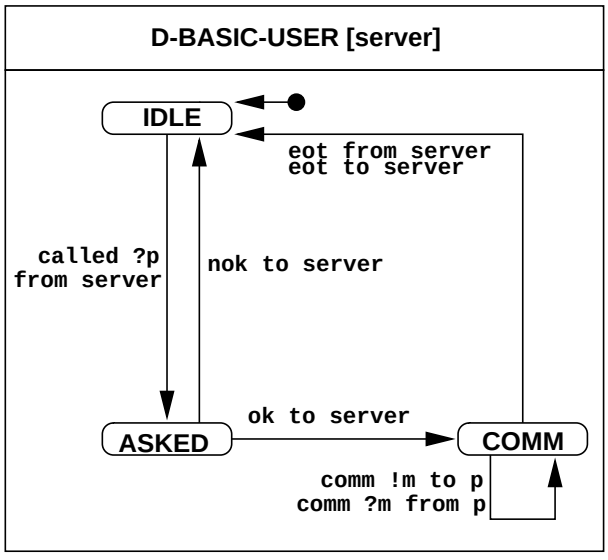


Figure C.7 : Utilisateurs de base – STS dynamique

INTEGRATION VIEW BASIC-USER

SPECIFICATION

generic on
server:PIdServer

COMPOSITION

is
s : S-EMPTY
d : D-BASIC-USER [server]
with true, true
initially true

Figure C.8 : Utilisateurs de base – Vue d'intégration

et des nouvelles opérations définies en utilisant ces nouvelles conditions.

Ici nous avons une nouvelle condition, `subscribed` qui est vraie lorsque l'utilisateur a souscrit un abonnement, et faux dans le cas contraire.

Le STS est défini en utilisant des états hiérarchiques pour chacune des valeurs possibles des nouvelles conditions. Ici nous avons une seule condition, donc le STS ne contiendra que deux états hiérarchiques (`subscribed` and `not subscribed`). Chacun des états hiérarchiques contient implicitement le STS complet de la vue dynamique dont sa vue hérite. C'est pourquoi les états contenus dans ces états hiérarchiques peuvent être référés en utilisant notre notation préfixée habituelle (par exemple `subscribed.IDLE` pour l'état `IDLE` de l'état hiérarchique `subscribed`). De plus, il est possible d'utiliser les noms des états du STS hérité à la place des formules correspondantes (construites à partir des conditions héritées). Ces noms peuvent être utilisés pour simplifier l'écriture des préconditions et des postconditions. Les préconditions de l'opération `subscribe` expriment ainsi par exemple que la condition `subscribed` doit être fausse et que l'état (hérité des utilisateurs de base) doit être `IDLE`. Lorsque la valeur des conditions n'est pas modifiée par une opération (*i.e.* lorsqu'il y a `c':c` dans une postcondition pour une condition `c` donnée) il est possible de l'omettre (elle est implicite).

La figure C.10 donne une représentation graphique de la partie STS de la vue dynamique des utilisateurs complets.

Vue d'intégration des utilisateurs complets.

La figure C.11 donne la vue d'intégration des utilisateurs complets en KORRIGAN. La partie statique des utilisateurs complets est un ensemble d'identifiants d'utilisateurs (`PIdUser`). Ceci correspond à l'ensemble des utilisateurs connus de l'abonné. Ces identifiants peuvent correspondre à des utilisateurs de base ou à des utilisateurs complets, tout `PIdFullUser` étant aussi un `PIdUser` puisqu'il existe une relation de sous-sorte entre les sortes `PIdFullUser` et `PIdUser`. La partie colle précise que l'utilisateur complet peut uniquement ajouter (ou supprimer) à sa liste d'appelés potentiels, et communiquer avec des éléments qui sont dans sa liste d'utilisateurs connus.

DYNAMIC VIEW D-FULL-USER EXTENDS D-BASIC-USER

SPECIFICATION

ops
subscribe **to** server
unsubscribe **to** server
ok **from** server
nok **from** server
addc !v:PIdUser **to** server
subc !v:PIdUser **to** server
call !p:PIdUser **to** server

ABSTRACTION

conditions subscribed
initially ¬ subscribed

OPERATIONS

subscribe
pre: IDLE ∧ ¬ subscribed
post: {subscribed':true}
unsubscribe
pre: IDLE ∧ subscribed
post: {subscribed':false}
ok
pre: ASKING ∧ subscribed
post: {COMM':true}
nok
pre: ASKING ∧ subscribed
post: {IDLE':true}
addc !v:PIdUser, subc !v:PIdUser
pre: IDLE ∧ subscribed
post: {}
call !p:PIdUser
pre: IDLE ∧ subscribed
post: {ASKING':true}

Figure C.9 : Utilisateurs complets – Vue dynamique

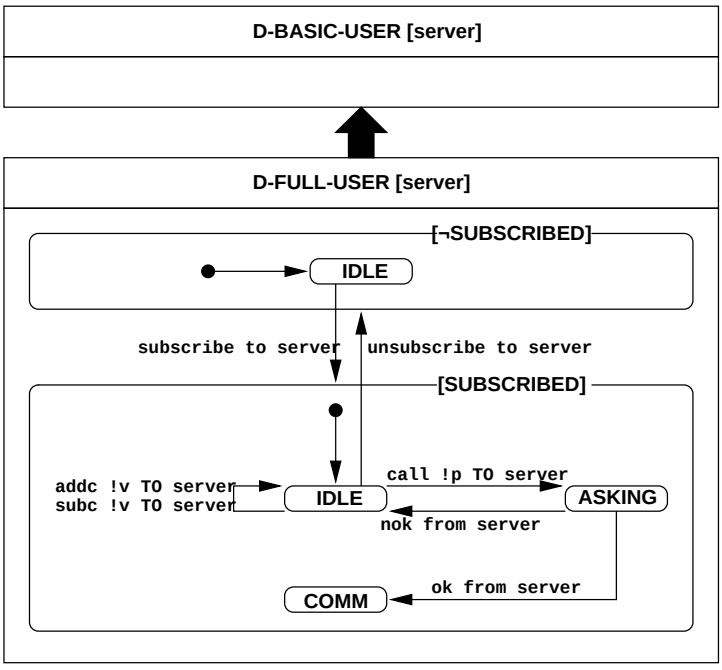


Figure C.10 : Utilisateurs complets – STS dynamique

INTEGRATION VIEW FULL-USER

SPECIFICATION

imports PId
generic on
server:PIdServer

COMPOSITION

is
s : SET [PIdUser]
d : D-FULL-USER [server]
with true, {
(true,d.addc !v to server ⇒ s.member(v)),
(true,d.subc !v to server ⇒ s.member(v)),
(true,d.call !p to server ⇒ s.member(p))}
initially true

Figure C.11 : Utilisateurs complets – Vue d'intégration

C.2.4 Unité de connexion

Vue dynamique de l'unité de connexion.

La figure C.12 donne la vue dynamique de l'unité de connexion en KORRIGAN.

La figure C.13 donne une représentation graphique de la partie STS de la vue dynamique de l'unité de connexion.

Vue d'intégration de l'unité de connexion (UC).

L'unité de connexion n'a pas de vue statique (vue statique vide). La figure C.14 donne la vue d'intégration de l'unité de connexion en KORRIGAN.

C.2.5 Unité de déconnexion

Vue dynamique de l'unité de déconnexion.

La figure C.15 donne la vue dynamique de l'unité de déconnexion en KORRIGAN.

La figure C.16 donne une représentation graphique de la partie STS de la vue dynamique de l'unité de déconnexion.

Vue d'intégration de l'unité de déconnexion (UD).

L'unité de déconnexion n'a pas de vue statique (vue statique vide). La figure C.17 donne la vue d'intégration de l'unité de déconnexion en KORRIGAN.

DYNAMIC VIEW D-CONNECTION-UNIT

SPECIFICATION

imports PId
generic on udb : PIdDB
ops
call ?p:PIdUser **from** PIdUser
newCx !p:PIdUser !p2:PIdUser
to udb
ask !p:PIdUser !p2:PIdUser
to udb
ok **from** udb
nok **from** udb
ok **to** PId
nok **to** PId
called !p:PIdUser **to** PId
ok **from** PId
nok **from** PId

STS

$\bullet \rightarrow$ IDLE
IDLE − call ?p:PIdUser **from** user → TESTING
TESTING − ask !p:PIdUser !user:PIdUser
to udb → WAIT1
WAIT1 − ok **from** udb → CALL
WAIT1 − nok **from** udb → CANCEL
CALL − call !user:PIdUser **to** p → WAIT2
WAIT2 − ok **from** p → $\bullet_1$
WAIT2 − nok **from** p → CANCEL
$\bullet_1$ − ok **to** user → $\bullet_2$
$\bullet_2$ − newCx !user:PIdUser !p:PIdUser **to** udb
→ IDLE
CANCEL − nok **to** user → IDLE

Figure C.12 : Unité de connexion – Vue dynamique

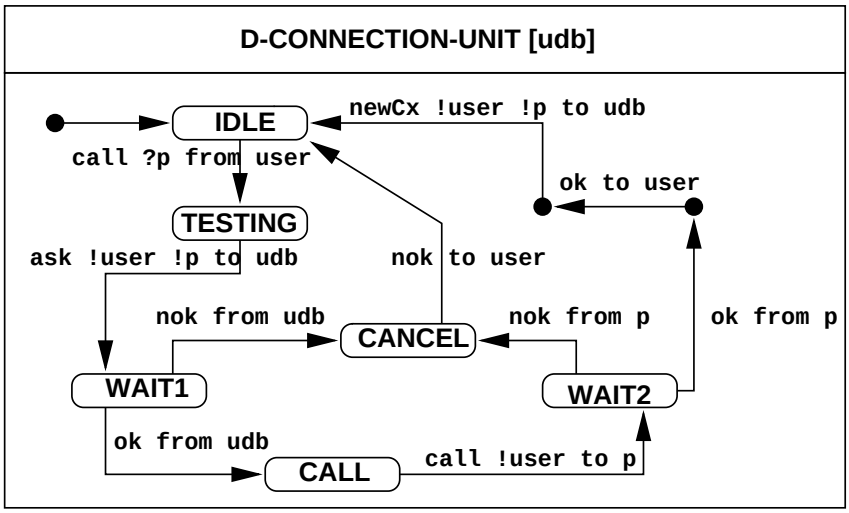


Figure C.13 : Unité de connexion – STS dynamique

INTEGRATION VIEW CONNECTION-UNIT (UC)

SPECIFICATION

generic on
udb:PIdDB

COMPOSITION

is
s : S-EMPTY
d : D-CONNECTION-UNIT [udb]
with true, true
initially true

Figure C.14 : Unité de connexion – Vue d'intégration

DYNAMIC VIEW D-DISCONNECTION-UNIT

SPECIFICATION

imports PId
generic on udb : PIdDB
ops
eot **from** PIdUser
eot **to** PIdUser
askOther !p:PIdUser **to** udb
getOther ?p:PIdUser **from** udb
askSomeIn **to** udb
getSomeIn ?p:PIdUser **from** udb

STS

• → IDLE
IDLE − eot **from** user1 → ASK
IDLE − askSomeIn **to** udb → $\bullet_1$
$\bullet_1$ − getSomeIn ?user1:PIdUser **from** udb → $\bullet_2$
$\bullet_2$ − eot **to** user1 → ASK
ASK − askOther !user1:PIdUser **to** udb → $\bullet_3$
$\bullet_3$ − → getOther ?user2:PIdUser **from** udb → $\bullet_4$
$\bullet_4$ − eot **to** user2 → IDLE

Figure C.15 : Unité de déconnexion – Vue dynamique

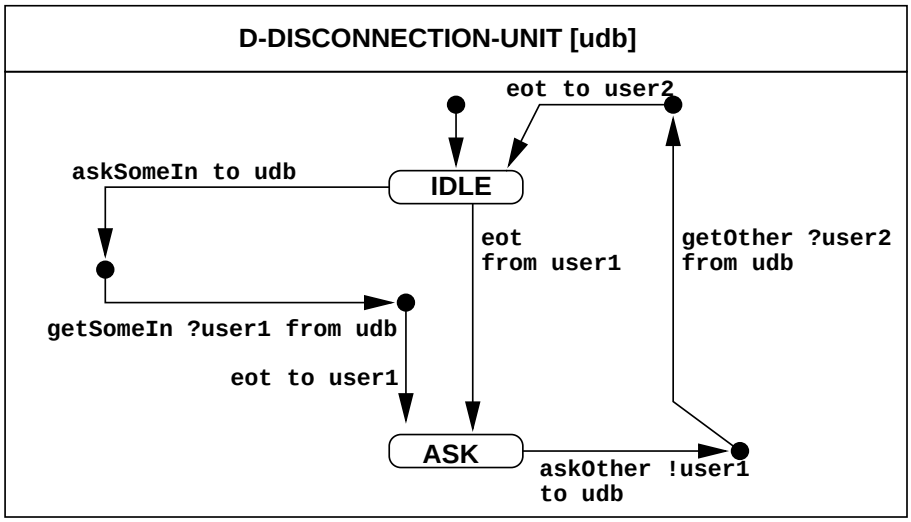


Figure C.16 : Unité de déconnexion – STS dynamique

INTEGRATION VIEW DISCONNECTION-UNIT (UD)

SPECIFICATION

generic on
udb:PIdDB

COMPOSITION

is
s : S-EMPTY
d : D-DISCONNECTION-UNIT [udb]
with true, true
initially true

Figure C.17 : Unité de déconnexion – Vue d'intégration

DYNAMIC VIEW D-MANAGEMENT-UNIT

SPECIFICATION	**STS**
imports PId	• → IDLE
generic on udb : PIdDB	IDLE − addc ?p:PIdUser **from** user → $\bullet_1$
ops	IDLE − subc ?p:PIdUser **from** user → $\bullet_2$
addc ?p:PIdUser **from** PIdUser	$\bullet_1$ − addc !user:PIdUser !p:PIdUser **to** udb
subc ?p:PIdUser **from** PIdUser	→ IDLE
addc !u:PIdUser !p:PIdUser **to** udb	$\bullet_2$ − subc !user:PIdUser !p:PIdUser **to** udb
subc !u:PIdUser !p:PIdUser **to** udb	→ IDLE
sub **from** PIdUser	IDLE − sub **from** user → $\bullet_3$
unsub **from** PIdUser	IDLE − unsub **from** user → $\bullet_4$
sub !u:PIdUser **to** udb	$\bullet_3$ − sub !user:PIdUser **to** udb → IDLE
unsub !u:PIdUser **to** udb	$\bullet_4$ − unsub !user:PIdUser **to** udb → IDLE

Figure C.18 : Unité de gestion – Vue dynamique

C.2.6 Unité de gestion

Vue dynamique de l'unité de gestion.

La figure C.18 donne la vue dynamique de l'unité de gestion en KORRIGAN.

La figure C.19 donne une représentation graphique de la partie STS de la vue dynamique de l'unité de gestion.

Vue d'intégration de l'unité de gestion (UM).

L'unité de gestion n'a pas de vue statique (vue statique vide). La figure C.20 donne la vue d'intégration de l'unité de gestion en KORRIGAN.

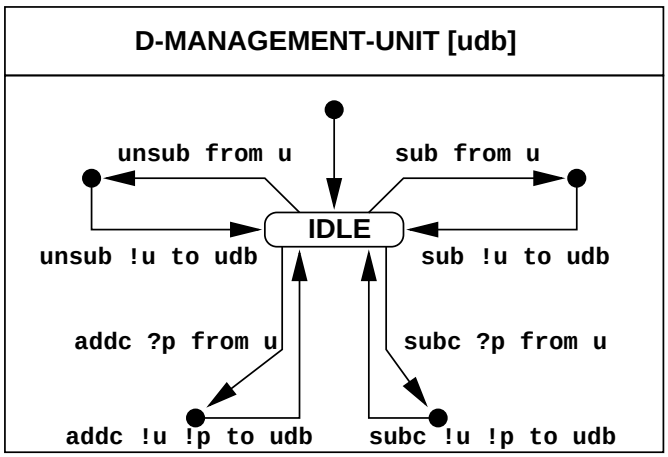


Figure C.19 : Unité de gestion – STS dynamique

INTEGRATION VIEW MANAGEMENT-UNIT (UM)

SPECIFICATION

generic on
udb:PIdDB

COMPOSITION

is
s : S-EMPTY
d : D-MANAGEMENT-UNIT [udb]
with true, true
initially true

Figure C.20 : Unité de gestion – Vue d'intégration

DYNAMIC VIEW D-DATABASE-UNIT

SPECIFICATION	**STS**
imports PId **generic on** uc:PIdUC, ud:PIdUD, um:PIdUM **ops** addc ?u:PIdUser ?u2:PIdUser **from** um subc ?u:PIdUser ?u2:PIdUser **from** um sub ?u:PIdUser **from** um unsub ?u:PIdUser **from** um newCx ?u:PIdUser ?u2:PIdUser **from** uc askSomeIn **from** ud getSomeIn !p:PIdUser **to** ud askOther ?u:PIdUser **from** ud getOther !u2:PIdUser **to** ud ask ?u1:PIdUser ?u2:PIdUser **from** uc ok **to** uc nok **to** uc	• → IDLE IDLE − addc ?u:PIdUser ?u2:PIdUser **from** um → IDLE IDLE − subc ?u:PIdUser ?u2:PIdUser **from** um → IDLE IDLE − sub ?u:PIdUser **from** um → IDLE IDLE − unsub ?u:PIdUser **from** um → IDLE IDLE − newCx ?u:PIdUser ?u2:PIdUser **from** uc → IDLE IDLE − askSomeIn **from** ud → REPL3 REPL3 − getSomeIn !p:PIdUser **to** ud → IDLE IDLE − askOther ?u:PIdUser **from** ud → REPL2 REPL2 − getOther !u2:PIdUser **to** ud → IDLE IDLE − ask ?u1:PIdUser ?u2:PIdUser **from** uc → REPL1 REPL1 − [auth(u1,u2)] ok **to** uc → IDLE REPL1 − [¬ auth(u1,u2)] nok **to** uc → IDLE

Figure C.21 : Unité de base de données – Vue dynamique

C.2.7 Unité de base de données

Vue dynamique de l'unité de base de données.

La figure C.21 donne la vue dynamique de l'unité de base de données en KORRIGAN.

La figure C.22 donne une représentation graphique de la partie STS de la vue dynamique de l'unité de base de données.

Vue d'intégration de l'unité de base de données (UDB).

La figure C.23 donne la vue d'intégration de l'unité de base de données en KORRIGAN. Sa partie statique est constituée de trois vues statiques. `sl` est l'ensemble des `PId` des abonnés. `al` correspond aux autorisations (appelés potentiels). Si un couple `(u1,u2)` est dans `al`, alors `u2` est autorisé à appeler `u1`. `cl` contient les couples d'identifiants d'utilisateurs en cours de communication.

Il s'agit d'une extension des vues d'intégration habituelles, dans lesquelles il y a une seule vue dynamique et une seule vue statique. Ceci est équivalent à la vue statique unique qui résulterait de la composition libre (`ALONE`) des trois vues statiques. Une autre solution aurait été de construire une vue dynamique triviale (*i.e.* avec des opérations dynamiques correspondant aux opérations statiques) pour chacune des trois vues statiques et de les intégrer (*i.e.* trois vues d'intégration), d'intégrer la vue

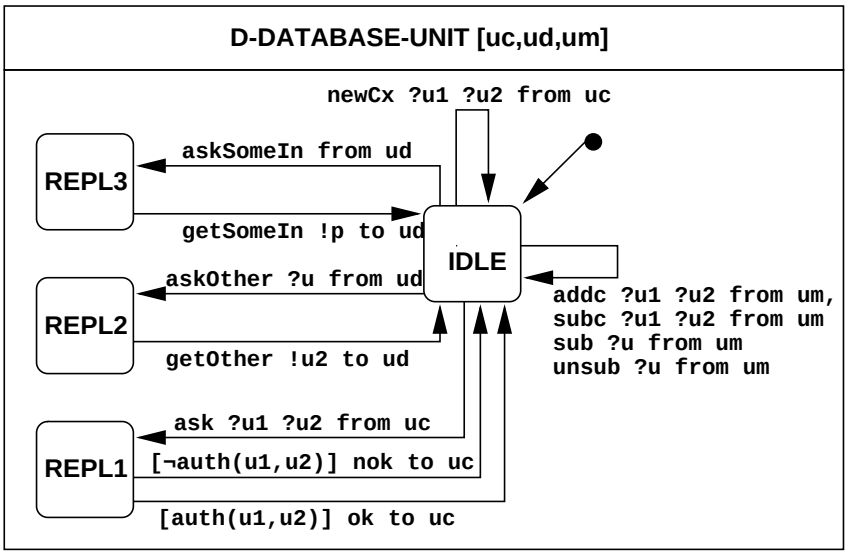


Figure C.22 : Unité de base de données – STS dynamique

dynamique de l'unité de base de données avec une vue statique vide (*i.e.* une vue d'intégration), puis de composer l'ensemble (*i.e.* les quatres vues d'intégration) au sein d'une vue de composition.

C.2.8 Serveur

La figure C.24 donne une représentation graphique de la composition du serveur en termes de composants et de communications entre ces composants.

La figure C.25 donne la vue de composition du serveur en KORRIGAN. Des instances d'UC, d'UM, d'UD et d'UDB sont utilisées. Les paramètres leur correspondant sont instanciés.

C.2.9 Vue de composition de la messagerie

La figure C.26 donne une représentation graphique de la composition du système de messagerie en termes de composants et de communications entre ces composants.

La figure C.27 donne la vue de composition de la messagerie en KORRIGAN. La messagerie est générique sur `n`, le nombre d'utilisateurs de base, et `m`, le nombre d'utilisateurs complets. Les utilisateurs sont crées en utilisant l'opérateur *range*, un raccourci syntaxique pour un ensemble fini de `PId` (construits en utilisant des entiers naturels). Ainsi par exemple, `i:[1..3].user.i : USER` est utilisé comme raccourci pour :

user.PId(1) : USER
user.PId(2) : USER
user.PId(3) : USER

L'opérateur de range peut aussi être utilisé dans les formules de colle.

INTEGRATION VIEW DATABASE-UNIT (UDB)

SPECIFICATION

generic on
uc:PIdUC, ud:PIdUD, um:PIdUM

COMPOSITION

is
sl : SET [PIdFullUser]
al : SET [Couple [PIdUser]]
cl : SET [Couple [PIdUser]]
d : D-DATABASE-UNIT [uc,ud,um]
with true, {
(sl.add(u), d.sub ?u **from** um)
(sl.remove(u), d.unsub ?u **from** um)
(al.add((u1,u2)), d.addc ?u1 ?u2 **from** um)
(al.remove((u1,u2)), d.subc ?u1 ?u2 **from** um)
(al.member((u1,u2)), d.auth(u1,u2) **from** um)
(cl.add((u1,u2)), d.newCx ?u1 ?u2 **from** uc)
(cl.member(u), d.getSomeIn !p1(u) **to** ud $\vee$ d.getSomeIn !p2(u) **to** ud)
(cl.member((u,u2)) $\vee$ cl.member((u2,u)),
d.($-$ askOther ?u **from** ud $>$ $-$ getOther !u2 **to** ud $>$))}
initially true

Figure C.23 : Unité de base de données – Vue d'intégration

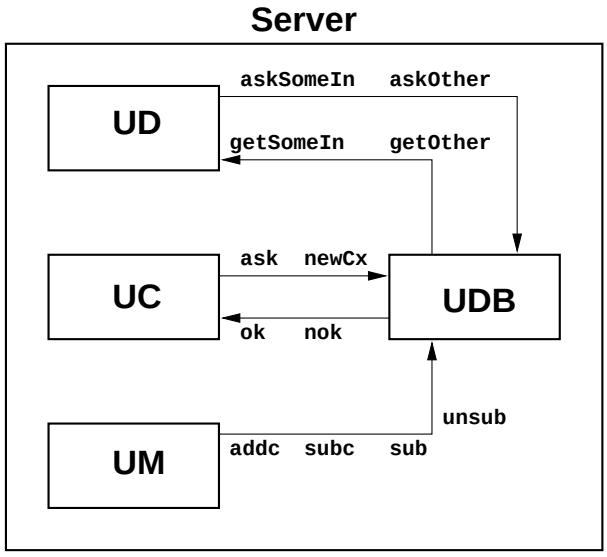


Figure C.24 : Serveur – Composition

COMPOSITION VIEW SERVER

COMPOSITION

is
uc : UC [udb]
um : UM [udb]
ud : UD [udb]
udb : UDB [uc,ud,um]
with true, {
(ud.askSomeIn, udb.askSomeIn)
(ud.getSomeIn ?p:PIdUser, udb.getSomeIn !p:PIdUser)
(ud.askOther !p:PIdUser, udb.askOther ?p:PIdUser)
(ud.getOther ?p:PIdUser, udb.getOther !p:PIdUser)
(uc.ask !u1:PIdUser !u2:PIdUser, udb.ask ?u1:PIdUser ?u2:PIdUser)
(uc.newCx !u1:PIdUser !u2:PIdUser, udb.newCx ?u1:PIdUser ?u2:PIdUser)
(uc.ok **from** udb, udb.ok)
(uc.nok **from** udb, udb.nok)
(um.addc !u1:PIdUser !u2:PIdUser, udb.addc ?u1:PIdUser ?u2:PIdUser)
(um.subc !u1:PIdUser !u2:PIdUser, udb.subc ?u1:PIdUser ?u2:PIdUser)
(um.sub !p:PIdUser, udb.sub ?p:PIdUser)
(um.unsub !p:PIdUser, udb.unsub ?p:PIdUser)}
initially true

Figure C.25 : Serveur – Vue de composition

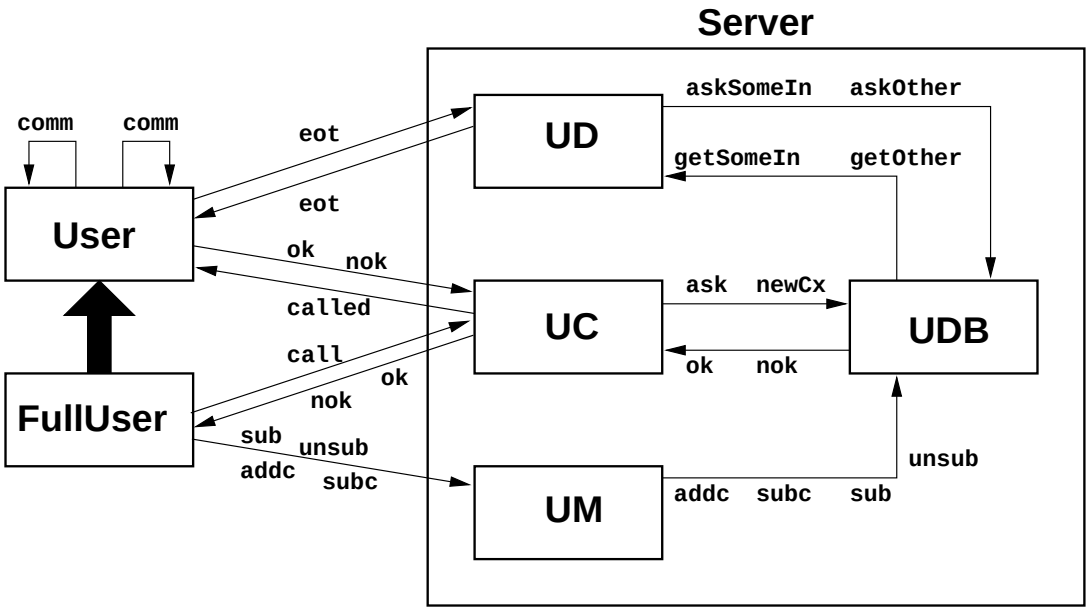


Figure C.26 : Messagerie – Composition

COMPOSITION VIEW PHONE-SERVICE

SPECIFICATION

generic on n,m : NATURAL

COMPOSITION

is
s : SERVER
i:[1..n].user.i : USER [s]
j:[n+1..n+m].user.j : FULL-USER [s]
with true, {
(user.i.comm ?m:Msg **from** user.j, user.j.comm !m:Msg **to** user.i)
(ud.eot **to** user.i, user.i.eot **from** ud)
(ud.eot **from** user.i, user.i.eot **to** ud)
(um.sub **from** user.i, user.i.sub **to** um)
(um.unsub **from** user.i, user.i.unsub **to** um)
(um.addc ?p:PIdUser **from** user.i, user.i.addc !p:PIdUser **to** um)
(um.subc ?p:PIdUser **from** user.i, user.i.subc !p:PIdUser **to** um)
(uc.ok **from** user.i, user.i.ok **to** uc)
(uc.nok **from** user.i, user.i.nok **to** uc)
(uc.ok **to** user.i, user.i.ok **from** uc)
(uc.nok **to** user.i, user.i.nok **from** uc)
(uc.called !p:PIdUser **to** user.i, user.i.called ?p:PIdUser **from** uc)
(uc.call ?p:PIdUser **from** user.i, user.i.call !p:PIdUser **to** uc)}
initially true

Figure C.27 : Messagerie – Vue de composition

ANNEXE D

Étude de cas : nœud de transit

D.1 Description de l'étude de cas

Cette étude de cas a été proposée lors du projet RACE 2039 (SPECS : Specification Environment for Communication Software) puis modifiée lors du projet VTT [43]. Il s'agit d'un nœud de transit (routeur) de réseau dans lequel des messages arrivent, sont routés, et quittent le nœud (figure D.1).

Nous reprenons ici la spécification originale (c'est pourquoi elle est en anglais).

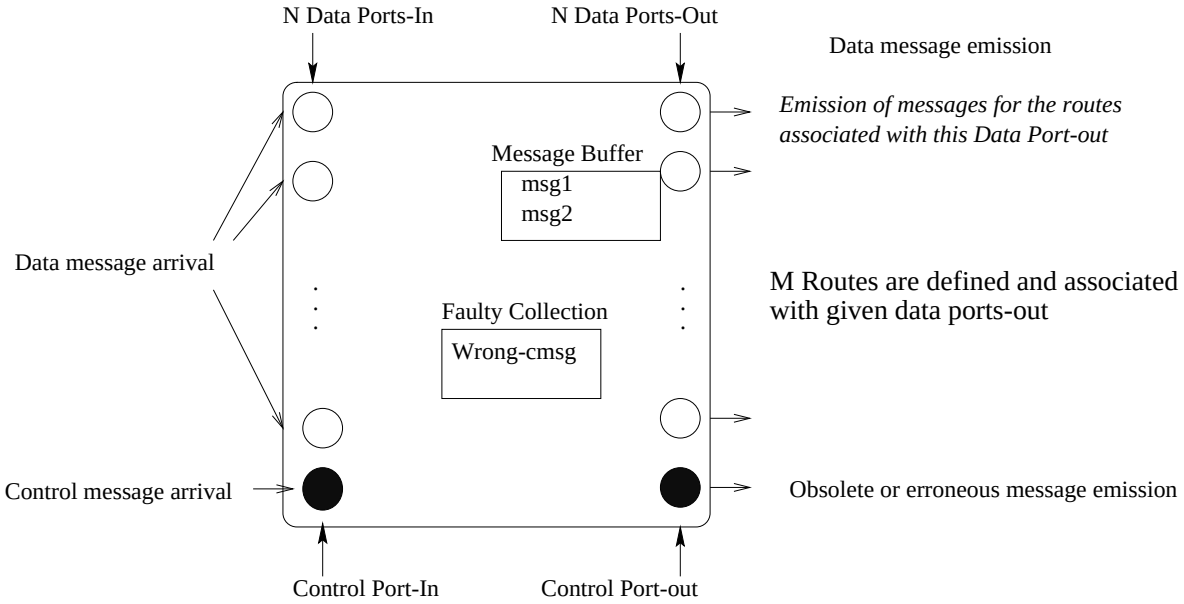


Figure D.1 : Nœud de transit

La spécification informelle est la suivante :

clause 1 The system to be specified consists of a transit node with: one *Control Port-In,* one *Control Port-Out,* N *Data Ports-In,* N *Data Ports-Out,* M *Routes Through*. The limits of *N* and *M* are not specified.

clause 2 **(a)** Each port is serialized. **(b)** All ports are concurrent to all others. The ports should be specified as separate, concurrent entities. **(c)** Messages arrive from the environment only when a *Port-In* is

able to treat them.

clause 3 The node is "fair". All messages are equally likely to be treated, when a selection must be made,

clause 4 and all data messages will eventually transit the node, or become faulty.

clause 5 *Initial State :* one *Control Port-In*, one *Control Port-Out*.

clause 6 The *Control Port-In* accepts and treats the following three messages:

(a) *Add-Data-Port-In-&-Out(n)* gives the node knowledge of a new *Port-In(n)* and a new *Port-Out(n)*. The node commences to accept and treat messages sent to the *Port-In*, as indicated below on *Data Port-In*.

(b) *Add-Route(m,n_i)* , associates route *m* with *Data-Port-Out(n_i)*.

(c) *Send-Faults* routes some messages in the faulty collection, if any, to *Control Port-Out*. The order in which the faulty messages are transmitted is not specified.

clause 7 A *Data Port-In* accepts only messages of the type : *Route(m).Data*.

(a) The *Port-In* routes the message, unchanged, to any one (non determinate) of the open *Data Ports-Out* associated with route *m*. If no such port exists, the message is put in the faulty collection.

(b) (Note that a *Data Port-Out* is serialized – the message has to be buffered until the *Data Port-Out* can process it).

(c) The message becomes a faulty message if its transit time through the node (from initial receipt by a *Data Port-In* to transmission by a *Data Port-Out*) is greater than a constant time *T*.

clause 8 *Data Ports-Out* and *Control Port-Out* accept messages of any type and will transmit the message out of the node. Messages may leave the node in any order.

clause 9 All faulty messages are eventually placed in the faulty collection where they stay until a *Send-Faults* command message causes them to be routed to *Control Port-Out*.

clause 10 Faulty messages are **(a)** messages on the *Control Port-In* that are not one of the three commands listed, **(b)** messages on a *Data Port-In* that indicate an unknown route, or **(c)** messages whose transit time through the node is greater than *T*.

clause 11 **(a)** Messages that exceed the transit time of *T* become faulty as soon as the time *T* is exceeded.
(b) It is permissible for a faulty message not to be routed to *Control Port-Out* by a *Send-Faults* command (because, for example, it has just become faulty, but has not yet been placed in a faulty message collection),
(c) but all faulty messages must eventually be sent to *Control Port-Out* with a succession of *Send-Faults* commands.

clause 12 It may be assumed that a source of time (time-of-day or a signal each time interval) is available in the environment and need not be modeled within the specification.

D.2 Description informelle

D.2.1 Description des fonctionnalités externes du système

Ici, les opérations sont données par les clauses 6 et 7 :
– au niveau du port de contrôle en entrée : réception d'un message de commande (`inCmde`)
– au niveau des ports de données en entrée : réception d'un message de données (`inData`)
Les clauses 7 et 9 induisent de plus à envisager les opération suivantes :
– au niveau du port de contrôle en sortie : émission d'un message erroné (`outCmde`)
– au niveau des ports de données en sortie : émission d'un message de données (`ouData`)
Enfin, la clause 12 précise qu'une source temporelle est disponible dans l'environnement. Nous supposerons que le nœud de transit peut récupérer cette valeur du temps.

D'autres opérations apparaissent dans les clauses, mais elles sont internes au nœud de transit. Elles seront envisagées lors de la décomposition du système.

Notons que la clause 9 ne précise pas ce que le port de contrôle en sortie fait des messages d'erreurs reçus. Par analogie avec les ports de données en sortie, nous supposerons qu'ils sont émis un par un.

D.2.2 Description des données du système

Le nœud de transit dispose d'une liste de numéros de ports actifs (clause 6a), d'une liste de routes (clauses 6b et 7a) et d'une collection de messages erronés (clauses 4, 6c, 7a et 9).

D.3 Étude de l'activité concurrente

D.3.1 Interfaces de communication

Nous donnons ici la description du nœud de transit au niveau le plus abstrait (Fig. D.2). Nous utilisons un type `Msg` pour représenter les différents messages. Le nœud de transit a quatre fonctionalités :
– réception d'un message de commande : `inCmde :?m:Msg`
– réception d'un message de données : `inData :?m:Msg`
– sortie d'un message erroné : `outCmde :!m:Msg`
– sortie d'un message de données : `outData :!m:Msg`
Ses données sont composées de trois listes et il est paramétré par `N`, le nombre maximum de ports de données (en entrée et en sortie) qu'il peut contenir.

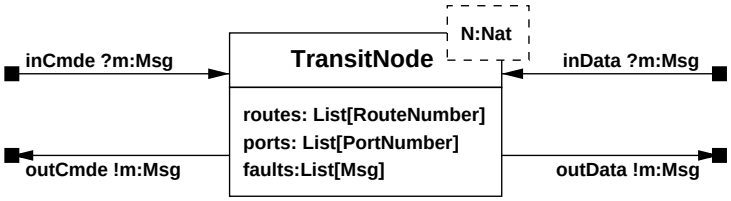


Figure D.2 : `TransitNode` - Interface de communication

D.3.2 Décomposition et répartition

La clause 1 nous conduit naturellement à voir que 4 catégories de composants se trouvent dans le nœud de transit : le port de contrôle en entrée (`InputControlPort`), les ports de données en entrée (`InputDataPort`), le port de contrôle en sortie (`OutputControlPort`) et les ports de données en sortie (`OutputDataPort`). Un composant supplémentaire (`FaultyCollection`), gèrera la collection des messages d'erreurs (clause 9). Plusieurs étapes de décomposition conduisent à l'obtention de ces composants.

Décomposition du nœud de transit.

Le système peut dans un premier temps être décomposé (figure D.3) en partie contrôle (`ControlPorts`) et partie données (`DataPorts`). Les routes, ainsi que les numéros de ports et les messages d'erreurs sont associés aux ports de contrôle.

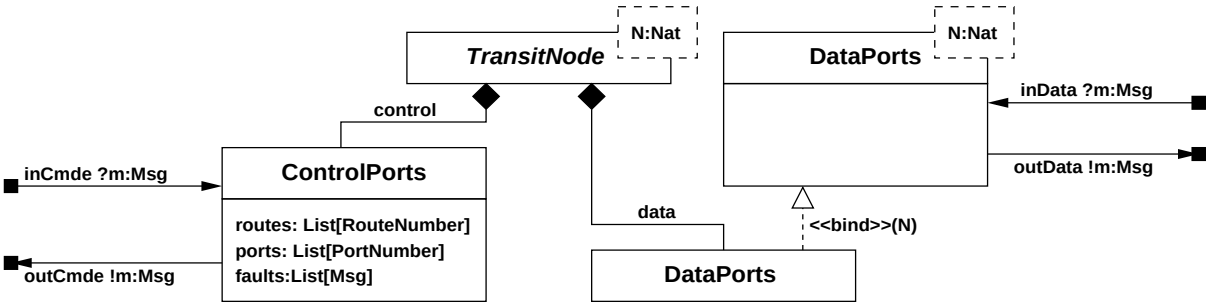


Figure D.3 : `TransitNode` – Diagramme de composition

Le composant `DataPorts` est, comme le `TransitNode`, paramétré par le nombre maximum de ports de données : `N` en entrée et `N` en sortie. Dans la figure, la liaison est faite entre ces deux paramètres par l'utilisation de l'opérateur `bind`, emprunté à UML.

Étude des contraintes induites par la décomposition du nœud de transit (nouvelles données)

De nouvelles données apparaissent suite à la décomposition. Les ports de données en sortie sont sérialisés (clause 7b) et disposent donc d'un buffer de messages. Il en va de même pour les ports de contrôle en sortie. Les ports de données dans leur ensemble ont besoin d'un identifiant de façon à les reconnaître lors des phases d'activation (par le port de contrôle en entrée, voir la remarque concernant la création de nouveaux ports par l'emploi d'une communication interne de nom `enable`, page 225) ou de routage (d'un port de données en entrée vers un port de données en sortie). Ceci est pris en compte par l'utilisation des identifiants dans les diagrammes de composition.

Décomposition de la partie contrôle.

La partie contrôle peut ensuite être décomposée en un port d'entrée (`InputControlPort`), un port de sortie (`OutputControlPort`) et une `FaultyCollection` (Fig. D.4). Les routes et les numéros de ports seront placés dans l'`InputControlPort` puisqu'il modifie ces listes (clause 6). La liste des messages d'erreurs est associée à la `FaultyCollection`. Elle contient les messages

d'erreurs jusqu'à réception d'un message `sendfault`. L'`OutputControlPort` émet les messages d'erreurs vers l'extérieur du nœud de transit un par un. Il dispose pour cela d'un buffer de sortie.

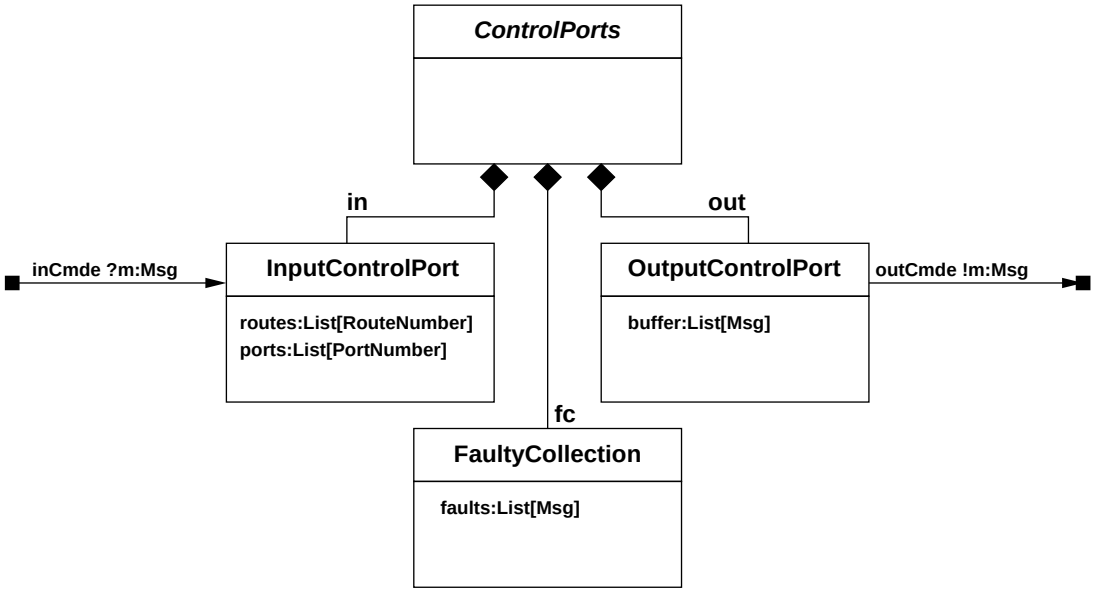


Figure D.4 : `ControlPorts` – Diagramme de composition

Décomposition de la partie données.

De son côté, la partie données est décomposée (figure D.5) en `N` ports d'entrée (`InputDataPort`) et `N` ports de sortie (`OutputDataPort`). Les ports de données disposent d'un identifiant (`nbr`) de façon à pouvoir les activer. Le port de sortie dispose aussi d'un buffer pour les sorties de messages.

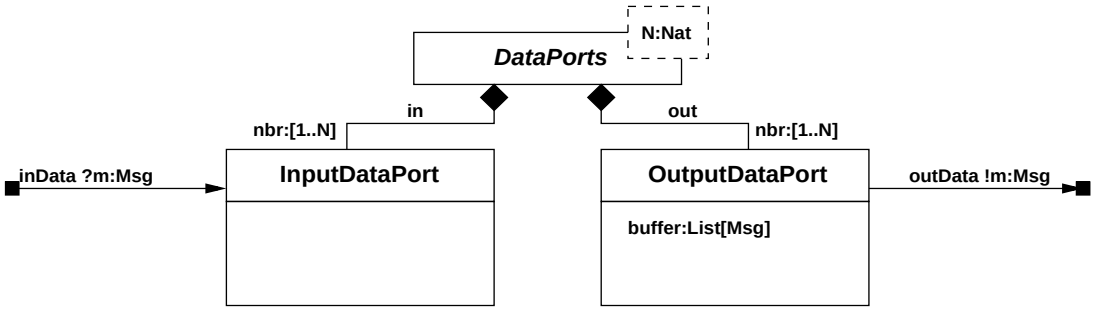


Figure D.5 : `DataPorts` – Diagramme de composition

La figure D.6 donne une réprésentation plus "à plat" du système comprenant une partie des informations de communication[1].

[1] Les commentaires concernant les communications internes au nœud de transit sont donnés dans la section concernant la composition parallèle.

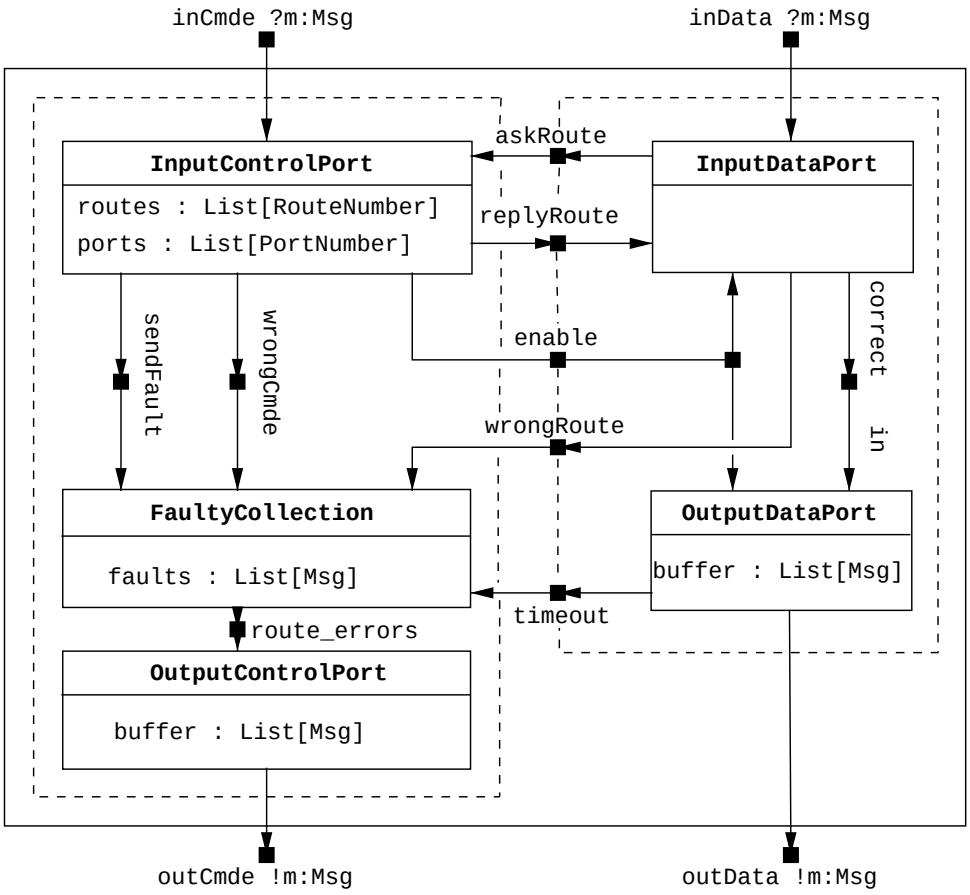


Figure D.6 : Vue interne du nœud de transit.

D.3.3 Composition parallèle

Étude des contraintes induites par la décomposition du nœud de transit (communications internes).

Messages erronés. Ils sont sauvés dans la collection de la `FaultyCollection` (clause 9). Il s'agit (clause 10) des messages de commande incorrects (`wrongCmde`, origine `InputControlPort`), des messages de données avec une route incorrecte (`wrongRoute`, origine `InputDataPort`) et des messages obsolètes (`timeout`, origine `OuputDataPort`).

Emission des messages d'erreur. La commande `sendFault`, reçue par l'`InputControlPort` déclenche le routage des messages erronés (en proportion non spécifiée) de la `FaultyCollection` vers l'`OutputControlPort` (clause 9). Cette phase utilise une communication de nom `sendFault` entre `InputControlPort` et `FaultyCollection` et une communication de nom `routeErrors` entre `FaultyCollection` et `OutputControlPort`. Nous supposerons que les messages reçus par l'`OutputControlPort` sont stockés et émis un par un (la spécification ne parle pas de ce point, mis à part la clause 8).

Informations sur les routes. L'`InputDataPort` a besoin d'informations sur les routes (existence ou non, liste des ports de sortie associés). Ces informations font partie des données de l'`InputControlPort`.

L'échange d'information se fera par communication de type question/réponse entre ces deux composants (`askRoute` et `replyRoute`).

Routage d'un message. Lorsque le message indique une route correcte, le message est routé (communication `correct`) par l'`InputDataPort` sur un des `OutputDataPorts` correspondant à la route (clause 7a).

Nouveaux ports. Lorsque l'`InputControlPort` reçoit le message `addDataPortIn&Out` il crée les ports correspondants (clause 6). Le formalisme des vues ne permet pas la création dynamique de composants. De plus, ce que l'`InputControlPort` fait lorsque le port existe déjà n'est pas indiqué. Pour ne pas sur-spécifier, nous avons opté pour une solution qui est de créer au départ tous les processus des `InputDataPorts` et `OutputDataPorts` (`N` de chaque, clause 1) ; pour respecter la clause 5, ces processus ne sont activés que lorsque l'`InputControlPort` reçoit le message demandant de les créer (communication `enable`). Ce qu'il faut faire lorsqu'un port de données est activé plusieurs fois n'est pas indiqué et donc pas spécifié.

D.4 Étude des composants séquentiels

D.4.1 Étude du Data Port In

Ce composant a été étudié en détail dans le chapitre 3.

D.4.2 Étude du Data Port Out

Le typage des communications du DPO a été étudié et est le suivant :
– `out_data : !m:Msg`
– `correct : !ident:PortNumber?m:Msg`
– `enable : !ident:PortNumber`
– `timeout : !m:Msg`
L'application des étapes 3.1 et 3.2 au DPO donne les résultats suivants :
– O : $\emptyset$,
– C : {`enable`},
– OC : $\emptyset$,
– CC : {`out_data` : autorisé, `correct` : autorisé, `timeout` : autorisé}
La table des états du DPO est donnée en table D.1.

autorisé	état
V	A (autorisé)
F	$\overline{A}$ (non autorisé)

Table D.1 : Table des conditions du Data Port Out.

Les préconditions et postconditions des opérations du DPO se trouvent ci-dessous.

`enable`	autorisé
$\mathcal{P}$	$\forall$
$\mathcal{Q}$	V

`out_data`	autorisé
$\mathcal{P}$	V
$\mathcal{Q}$	=

autorisé	vide	état
V	V	AV
V	F	A$\overline{V}$
F	V	$\overline{A}$V
F	F	$\overline{AV}$

Table D.2 : Table révisée des conditions du Data Port Out.

`correct`	autorisé
$\mathcal{P}$	V
$\mathcal{Q}$	=

`timeout`	autorisé
$\mathcal{P}$	V
$\mathcal{Q}$	=

On peut ici remarquer que les préconditions et postconditions ne correspondent pas au classement des opérations. Ainsi, au regard de ces tables, `out_data`, `correct` et `timeout` devraient se trouver dans OC et non pas dans CC. Pourtant, intuitivement nous les avons bien placées dans CC.

En fait, nous avions omis la condition concernant le fait que le buffer du DPO soit vide ou non. Cette condition influe sur la possibilité d'effectuer ou non des émissions de message (`out_cmde` et `timeout`). D'autre part, les opérations d'émission et de réception de message (`correct`) vont modifier sa valeur.

Nous recommençons donc l'application de l'agenda à partir de l'étape 3.1.

L'application des étapes 3.1 et 3.2 au DPO donne les résultats suivants :

– O : $\emptyset$,
– C : {`enable`},
– OC : $\emptyset$,
– CC : {`out_data` : autorisé $\wedge$ $\neg$vide, `correct` : autorisé, `timeout` : autorisé $\wedge$ $\neg$vide}

La table des états du DPO est donnée en table D.2.

Les préconditions et postconditions des opérations du DPO se trouvent ci-dessous.

`enable`	autorisé	vide
$\mathcal{P}$	$\forall$	$\forall$
$\mathcal{Q}$	V	=

`out_data`	autorisé	vide
$\mathcal{P}$	V	F
$\mathcal{Q}$	=	unSeulMsg

`correct`	autorisé	vide
$\mathcal{P}$	V	$\forall$
$\mathcal{Q}$	=	F

`timeout`	autorisé	vide
$\mathcal{P}$	V	F
$\mathcal{Q}$	=	unSeulMsg

L'automate de DPO est donné dans la figure D.7. L'état $\overline{AV}$ est supprimé car il n'est pas atteignable. À ce niveau, nous trouvons la formule caractéristique de cet état. Elle est obtenue à partir de la ligne de la table des conditions qui lui correspond : $\neg$autorisé $\wedge$ $\neg$vide. Puisque cet état n'est pas atteignable, nous prenons la négation de cette formule, ce qui donne : autorisé $\vee$ vide. Cette formule est ensuite proposée au spécifieur en tant qu'ajout aux formules de l'étape 3.3. Si le spécifieur valide la formule, il est possible de redérouler l'agenda à partir de cette étape (phases de validation essentiellement). Si le spécifieur ne valide pas cette formule (ou si elle est incohérente par rapport aux autres formules), c'est qu'une incohérence dans sa spécification a été détectée.

D.4.3 Étude du Control Port In

Le typage des communications du CPI a été étudié et est le suivant :

– `in_cmde :?m:Msg`

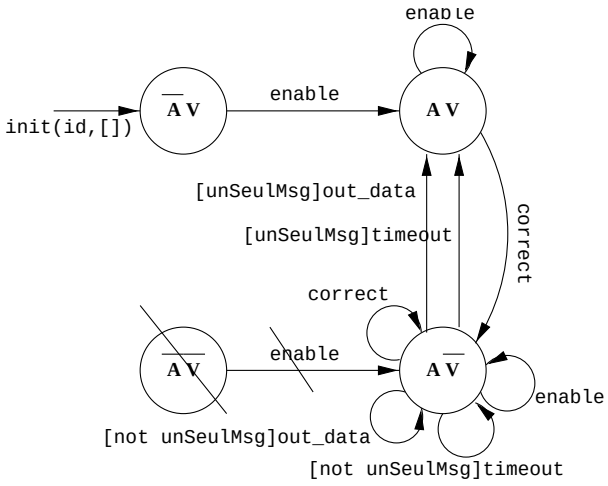


Figure D.7 : Automate du Data Port Out.

– `enable : !n:PortNumber`
– `sendfault`
– `wrong_cmde : !m:Msg`
– `ask_route : ?r:RouteNumber`
– `reply_route : !r:RouteNumber !l:list[PortNumber]`

L'application des étapes 3.1 et 3.2 au CPI donne les résultats suivants :

– O : $\emptyset$,
– C : $\emptyset$,
– OC : $\emptyset$,
– CC : {`in_cmde` : ¬reçu, `ask_route` : ¬demandé, `reply_route` : demandé, `sendfault` : reçu $\wedge$ code=*Send-Faults*, `enable` : reçu $\wedge$ code=*Add-Data-Port-In-&-Out*, `wrong_cmde` : reçu $\wedge$ code$\notin$ {*Send-Faults*, *Add-Data-Port-In-&-Out*, *Add-Route*}}

La condition de `wrong_cmde` fait apparaitre la nécessité d'un évènement interne correspondant à la reception d'un message/transition interne d'ajout de route. Nous ajoutons donc à CC : $\mathtt{i}_{add\_route}$: reçu $\wedge$ code=*Add-Route*.

Nous utiliserons les notations suivantes.

– code=*Send-Faults* : c_{sf} (pour sendfault)
– code=*Add-Data-Port-In-&-Out* : c_{en} (pour enable)
– code=*Add-Route* : c_i
– code$\notin$ {*Send-Faults*, *Add-Data-Port-In-&-Out*, *Add-Route*} : c_{wc} (pour wrong_cmde)

Nous avons la relation suivante : reçu $\Leftrightarrow c_{sf} \oplus c_{en} \oplus c_i \oplus c_{wc}$.

La table des états du CPI est donnée en table D.3.

Ces relations permettent de simplifier les conditions des opérations de CC comme suit :

{`in_cmde` : ¬reçu, `ask_route` : ¬demandé, `reply_route` : demandé, `sendfault` : c_{sf}, `enable` : c_{en}, `wrong_cmde` : c_{wc}, $\mathtt{i}_{add\_route}$: c_i}

Les préconditions et postconditions des opérations du CPI se trouvent ci-dessous (en utilisant r pour reçu et d pour demandé). En ce qui concerne les postconditions de `in_cmde?m:Msg`, nous utilisons des racourcis, `csf` désignant par exemple en fait le fait que le message `m` reçu est de type *Send-Faults* soit : is_sendfault(m).

c_{sf}	c_{en}	c_i	c_{wc}	reçu	demandé	état
F	F	F	V	V	V	WC-DEM
F	F	F	V	V	F	WC-ATT
F	F	V	F	V	V	I-DEM
F	F	V	F	V	F	I-ATT
F	V	F	F	V	V	EN-DEM
F	V	F	F	V	F	EN-ATT
V	F	F	F	V	V	SF-DEM
V	F	F	F	V	F	SF-ATT
F	F	F	F	F	V	ATT-DEM
F	F	F	F	F	F	ATT-ATT

Table D.3 : Table des conditions du Control Port In.

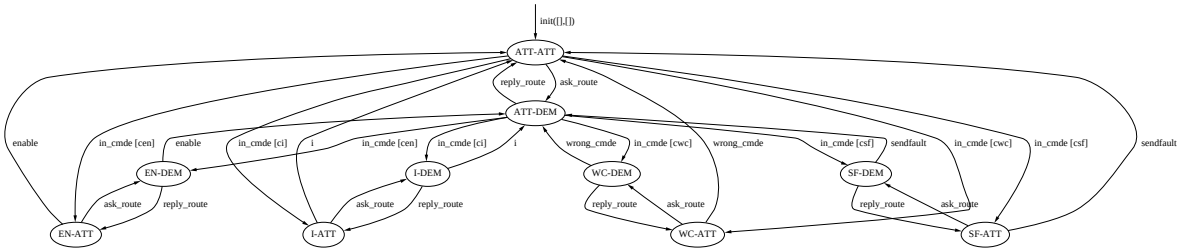


Figure D.8 : Automate du Control Port In.

in_cmde	c_{sf}	c_{en}	c_i	c_{wc}	r	d
$\mathcal{P}$	F	F	F	F	F	$\forall$
$\mathcal{Q}$	csf	cen	ci	cwc	V	=

ask_route	c_{sf}	c_{en}	c_i	c_{wc}	r	d
$\mathcal{P}$	$\forall$	$\forall$	$\forall$	$\forall$	$\forall$	F
$\mathcal{Q}$	=	=	=	=	=	V

reply_route	c_{sf}	c_{en}	c_i	c_{wc}	r	d
$\mathcal{P}$	$\forall$	$\forall$	$\forall$	$\forall$	$\forall$	V
$\mathcal{Q}$	=	=	=	=	=	F

sendfault	c_{sf}	c_{en}	c_i	c_{wc}	r	d
$\mathcal{P}$	V	F	F	F	V	$\forall$
$\mathcal{Q}$	F	F	F	F	F	=

enable	c_{sf}	c_{en}	c_i	c_{wc}	r	d
$\mathcal{P}$	F	V	F	F	V	$\forall$
$\mathcal{Q}$	F	F	F	F	F	=

i_{add_route}	c_{sf}	c_{en}	c_i	c_{wc}	r	d
$\mathcal{P}$	F	F	V	F	V	$\forall$
$\mathcal{Q}$	F	F	F	F	F	=

wrong_cmde	c_{sf}	c_{en}	c_i	c_{wc}	r	d
$\mathcal{P}$	F	F	F	V	V	$\forall$
$\mathcal{Q}$	F	F	F	F	F	=

L'automate de CPI est donné dans la figure D.8.

D.4.4 Étude de la Faulty Collection

Le typage des communications de la FC a été étudié et est le suivant :
– `timeout :?m:Msg`
– `wrong_route :?m:Msg`

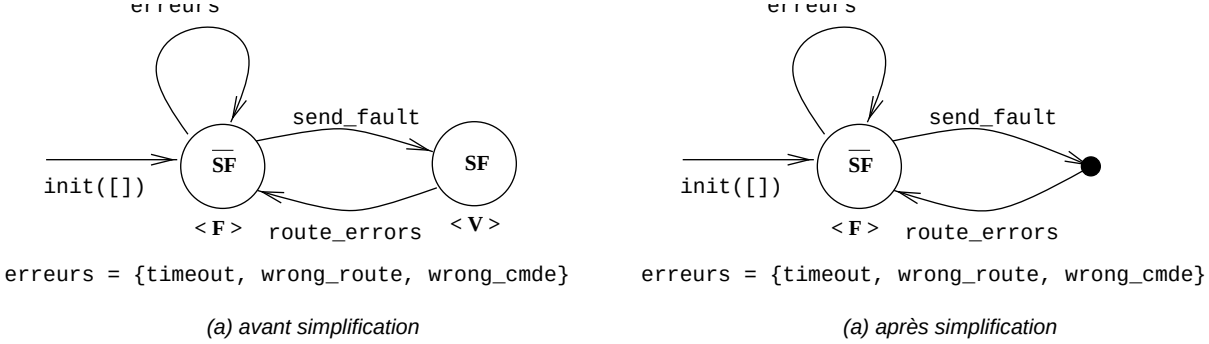


Figure D.9 : Automate de la Faulty Collection.

– `wrong_cmde :?m:Msg`
– `sendfault`
– `route_errors :!l:List[Msg]`

Nous emploirons le terme `erreurs` pour représenter `timeout`, `wroung_route` et `wrong_cmde`.
L'application des étapes 3.1 et 3.2 à la FC donne les résultats suivants :
– O : $\emptyset$,
– C : $\emptyset$,
– OC : {`erreurs` : ¬reçu},
– CC : {`sendfault` : ¬reçu, `route_errors` : reçu}
La table des états de la FC est donnée en table D.4.

reçu	état
V	SF (sendfault reçu)
F	$\overline{SF}$ (sendfault non reçu)

Table D.4 : Table des conditions de la Faulty Collection.

Les préconditions et postconditions des opérations de la FC se trouvent ci-dessous.

`sendfault`	reçu
$\mathcal{P}$	F
$\mathcal{Q}$	V

`erreurs`	reçu
$\mathcal{P}$	F
$\mathcal{Q}$	=

`route_errors`	reçu
$\mathcal{P}$	V
$\mathcal{Q}$	F

L'automate de la FC est donné dans la figure D.9a et sa version simplifiée dans la figure D.9b. La simplification a consisté à réunir les deux transitions qui constituaient un "tunnel", c'est-à-dire qui reliaient un état de dégré 2.

D.4.5 Étude du Control Port Out

Le typage des communications du CPO a été étudié et est le suivant :
– `out_cmde :!m:Msg`
– `route_errors :?l:List[Msg]`

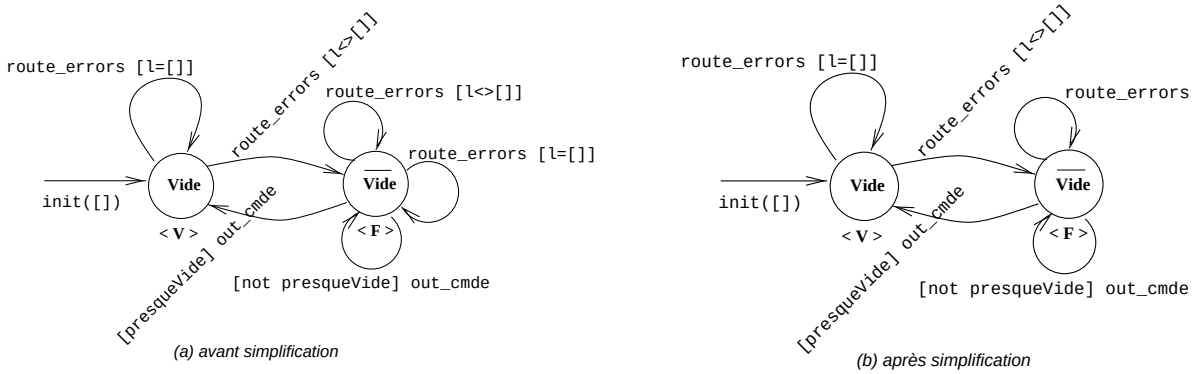


Figure D.10 : Automate du Control Port Out.

L'application des étapes 3.1 et 3.2 au CPO donne les résultats suivants :
– O : $\emptyset$,
– C : {route_errors},
– OC : $\emptyset$,
– CC : {out_cmde : ¬vide}
La table des états du CPO est donnée en table D.5.

vide	état
V	Vide
F	$\overline{Vide}$

Table D.5 : Table des conditions du Control Port Out.

Les préconditions et postconditions des opérations du CPO se trouvent ci-dessous.

route_errors	vide
$\mathcal{P}$	$\forall$
$\mathcal{Q}$	l=[] $\wedge$ vide

out_cmde	vide
$\mathcal{P}$	F
$\mathcal{Q}$	presqueVide

L'automate du CPO est donné dans la figure D.10a et sa version simplifiée dans la figure D.10b. La simplification a consisté à réunir dans une seule transition les transitions de même origine, même destination et même étiquette.

D.4.6 Spécification algébrique des types de données : application au DPI

Résultats de la dérivation:

```
type DPI

% GENERATEURS
init : Nat -> DPI
enable : DPI -> DPI
in_data : DPI x MSG -> DPI
ask_route : DPI -> DPI
reply_route : DPI x LIST[NAT] -> DPI
```

```
% CONDITIONS DE BASE
 enabled : DPI -> BOOL
    recu : DPI -> BOOL
 demande : DPI -> BOOL
 repondu : DPI -> BOOL
routeerr : DPI -> BOOL

% CONDITIONS D'ETAT
DIS : DPI -> BOOL
PAR : DPI -> BOOL
PAD : DPI -> BOOL
 AR : DPI -> BOOL
 CR : DPI -> BOOL
 WR : DPI -> BOOL

% AUTRES
correct_c : DPI -> DPI
wrong_route_c : DPI -> DPI
correct_o : DPI -> MSG
wrong_route_o : DPI -> MSG

% RELATIONS ENTRE CONDITIONS
recu(dpi) => enabled(dpi)
demande(dpi) => recu(dpi)
repondu(dpi) => enabled(dpi)
routeerr(dpi) => repondu(dpi)

% CONDITIONS DE BASE
enabled(init(dpi,id)) = false
enabled(enable(dpi)) = true
enabled(in_data(dpi,m)) = enabled(dpi)
enabled(ask_route(dpi)) = enabled(dpi)
enabled(reply_route(dpi,l)) = enabled(dpi)

recu(init(dpi,id)) = false
recu(enable(dpi)) = recu(dpi)
recu(in_data(dpi,m)) = true
recu(ask_route(dpi)) = recu(dpi)
recu(reply_route(dpi,l)) = recu(dpi)

demande(init(dpi,id)) = false
demande(enable(dpi)) = demande(dpi)
demande(in_data(dpi,m)) = demande(dpi)
demande(ask_route(dpi)) = true
demande(reply_route(dpi,l)) = demande(dpi)
```

```
repondu(init(dpi,id)) = false
repondu(enable(dpi)) = repondu(dpi)
repondu(in_data(dpi,m)) = repondu(dpi)
repondu(ask_route(dpi)) = repondu(dpi)
repondu(reply_route(dpi,l)) = true

routeerr(init(dpi,id)) = false
routeerr(enable(dpi)) = routeerr(dpi)
routeerr(in_data(dpi,m)) = routeerr(dpi)
routeerr(ask_route(dpi)) = routeerr(dpi)
routeerr(reply_route(dpi,l)) = (l=[])

% CONDITIONS D'ETAT
DIS(dpi) = not(enabled(dpi)) /\ not(recu(dpi)) /\ not(demande(dpi))
        /\ not(repondu(dpi)) /\ not(routeerr(dpi))
PAR(dpi) =     enabled(dpi)  /\ not(recu(dpi)) /\ not(demande(dpi))
        /\ not(repondu(dpi)) /\ not(routeerr(dpi))
PAD(dpi) =     enabled(dpi)  /\     recu(dpi)  /\ not(demande(dpi))
        /\ not(repondu(dpi)) /\ not(routeerr(dpi))
 AR(dpi) =     enabled(dpi)  /\     recu(dpi)  /\     demande(dpi)
        /\ not(repondu(dpi)) /\ not(routeerr(dpi))
 CR(dpi) =     enabled(dpi)  /\     recu(dpi)  /\     demande(dpi)
        /\     repondu(dpi)  /\ not(routeerr(dpi))
 WR(dpi) =     enabled(dpi)  /\     recu(dpi)  /\     demande(dpi)
        /\     repondu(dpi)  /\     routeerr(dpi)

% AUTRES
CR(dpi) => correct_c(enable(dpi))
           = correct_c(dpi)
AR(dpi) /\ not(l=[]) => correct_c(reply_route(ask_route(dpi),l))
           = correct_c(dpi)
AR(dpi) /\ not(l=[]) => correct_c(reply_route(enable(dpi),l))
           = correct_c(dpi)
PAD(dpi) /\ not(l=[]) => correct_c(reply_route(
                         ask_route(enable(dpi)),l))
           = correct_c(dpi)
PAR(dpi) /\ not(l=[]) => correct_c(reply_route(
                         ask_route(in_data(dpi,m)),l))
           = dpi

CR(dpi) => correct_o(enable(dpi))
           = correct_o(dpi)
AR(dpi) /\ not(l=[]) => correct_o(reply_route(ask_route(dpi),l))
           = correct_o(dpi)
AR(dpi) /\ not(l=[]) => correct_o(reply_route(enable(dpi),l))
```

```
           = correct_o(dpi)
PAD(dpi) /\ not(l=[]) => correct_o(reply_route(
                         ask_route(enable(dpi)),l))
           = correct_o(dpi)
PAR(dpi) /\ not(l=[]) => correct_o(reply_route(
                         ask_route(in_data(dpi,m)),l))
           = m

CR(dpi) => wrong_route_c(enable(dpi))
           = wrong_route_c(dpi)
AR(dpi) /\ not(l=[]) => wrong_route_c(reply_route(ask_route(dpi),l))
           = wrong_route_c(dpi)
AR(dpi) /\ not(l=[]) => wrong_route_c(reply_route(enable(dpi),l))
           = wrong_route_c(dpi)
PAD(dpi) /\ not(l=[]) => wrong_route_c(reply_route(
                         ask_route(enable(dpi)),l))
           = wrong_route_c(dpi)
PAR(dpi) /\ not(l=[]) => wrong_route_c(reply_route(
                         ask_route(in_data(dpi,m)),l))
           = dpi

CR(dpi) => wrong_route_o(enable(dpi))
           = wrong_route_o(dpi)
AR(dpi) /\ not(l=[]) => wrong_route_o(reply_route(ask_route(dpi),l))
           = wrong_route_o(dpi)
AR(dpi) /\ not(l=[]) => wrong_route_o(reply_route(enable(dpi),l))
           = wrong_route_o(dpi)
PAD(dpi) /\ not(l=[]) => wrong_route_o(reply_route(
                         ask_route(enable(dpi)),l))
           = wrong_route_o(dpi)
PAR(dpi) /\ not(l=[]) => wrong_route_o(reply_route(
                         ask_route(in_data(dpi,m)),l))
           = m
```

KORRIGAN : un formalisme et une méthode pour la spécification formelle et structurée de systèmes mixtes

Pascal POIZAT

Résumé

L'emploi des spécifications formelles est d'une grande importance, tout particulièrement pour le développement de systèmes dits sécuritaires ou critiques. Les spécifications mixtes ont pour but de permettre l'expression des différents aspects que peuvent présenter ces systèmes : statique (types de données), dynamiques (comportements) et composition (architecture, concurrence et communication). La complexité des systèmes de taille réelle rend indispensable la définition de moyens de structuration des spécifications mixtes. Pour cela, nous proposons un modèle basé sur une hiérarchie de structures que nous appelons *vues*, ainsi que KORRIGAN, le langage formel associé. Les vues intègrent des *systèmes de transition symboliques*, des spécifications algébriques et une forme de logique temporelle. Elles permettent la spécification des différents aspects de façon unifiée. Elles sont expressives, lisibles et favorisent la définition de composants à un haut niveau d'abstraction. Notre modèle offre trois moyens de structuration des spécifications. La *structuration interne* permet de définir les aspects de base des composants (dynamique et statique). La *structuration externe* permet de définir de façon unifiée différents types de compositions : tant l'intégration d'aspects que la composition concurrente de composants communicants. Une forme simple de *structuration d'héritage* permet de réutiliser les composants. Nous pensons qu'il est important, pour que les méthodes formelles soient réellement utilisées, qu'elles disposent d'une méthode associée. Dans cette optique, nous proposons une méthode pour la spécification mixte et structurée, applicable aux spécifications KORRIGAN ainsi qu'à d'autres formalismes. Enfin, nous proposons un atelier, ASK, dédié à la spécification mixte en KORRIGAN. Il intègre des mécanismes de vérification des spécifications KORRIGAN par traduction et de génération de code orienté objet.

Mots-clés : spécifications formelles, spécifications mixtes, structuration, sémantique, vues, systèmes de transitions symboliques, méthode, atelier, génération de code orienté objet

Abstract

The use of formal specifications is quite knowledgeable, in particular when developing safety critical systems. The aim of mixed specifications is to allow one to express the different aspects present in these systems, i.e. static (datatypes), dynamic (behaviour), and composition (architecture, concurrency and communication). The complexity of real size applications requires that structuring means for mixed specifications should be defined. This is why we present a model based on a hierarchy of structures that we call *views*, together with KORRIGAN, the associated formal language. Views integrate *symbolic transition systems*, algebraic specifications, and a form of temporal logic. They allow one to specify the different aspects in a unified way. They are expressive, readable, and promote the component definition at a high level of abstraction. Our model comprises three different means for structuring specifications. The basic aspects (static and dynamic) of the components are defined within the *internal structuring*. The different kinds of composition (integration of aspects, concurrent composition of communicating components) are defined in a unified way within the *external structuring*. Components may be reused through a simple form of *inheritance structuring*. To put formal methods into practice, it is important that they should be equipped with an appropriate method. To this end we propose a method for writing mixed and structured specifications that may be used for KORRIGAN but also for other mixed specification formalisms. Finally the ASK toolbox that is dedicated to mixed specification in KORRIGAN is described. ASK comprises verification means for KORRIGAN specifications through translation and object-oriented code generation.

Keywords: formal specifications, mixed specifications, structuring, semantics, views, symbolic transition systems, method, toolbox, object-oriented code generation