ÉCOLE NORMALE SUPÉRIEURE DE CACHAN

THÈSE

présentée
pour obtenir

LE GRADE DE DOCTEUR EN SCIENCES
DE L'ÉCOLE NORMALE SUPÉRIEURE DE CACHAN
Spécialité INFORMATIQUE

PAR

Nicola (Nicky) WILLIAMS

Application des spécifications algébriques à la rétro-ingénierie de codes Fortran

Soutenue le 1er février 2000 devant le jury composé de:

Mme.	Véronique DONZEAU-GOUGE	Président et Rapporteur
M.	Michel LEMOINE	Rapporteur
M.	Michel BIDOIT	Directeur de thèse
M.	Jérôme RYCKBOSCH	Examinateur
Mme.	Corinne ANCOURT	Examinateur
M.	Laurent FRIBOURG	Examinateur

Laboratoire Spécification et Vérification - CNRS UMR 8643,
ENS de Cachan, 61 Avenue du Président Wilson, 94235 CACHAN Cedex

Remerciements

Je remercie tout d'abord EDF pour son soutien financier à travers le contrat d'étude R32/1K2924/ER-292 et aussi pour m'avoir permis d'étudier et d'employer dans la validation de mes idées, des fragments de ses vrais codes opérationnels. Jérôme Ryckbosch m'a encadré au sein d'EDF. Sans lui, cette thèse n'aurait peut-être pas eu lieu et je le remercie pour avoir toujours cru à l'intérêt du sujet, su situer mon travail par rapport aux besoins d'EDF et veillé à la dissémination de mes résultats au sein d'EDF. Merci aussi à Francine Bena, Jean-Yves Lucas et Laurent Pierre d'EDF pour toute leur aide et à Karim Chenikhar, Penny Tsamasphyrou et Catherine Dufourd pour leur aide pratique et leur soutien moral !

Je voudrais remercier Michel Bidoit, d'abord pour son ouverture d'esprit en acceptant comme étudiant une femme étrangère nettement plus âgée que le thésard moyen ! Ensuite pour sa relecture consciencieuse des nombreux rapports et spécifications intermédiaires et l'orientation judicieuse de mes recherches. Cela a été un plaisir de partager avec lui un intérêt réel dans l'application des techniques universitaires aux problèmes industriels. Merci aux thésards et autres membres du LSV pour l'ambiance conviviale dans laquelle j'ai travaillé.

Je tiens à remercier aussi Béatrice Creusillet, François Irigoin, Corinne Ancourt, Fabien Coelho et les autres membres de l'équipe PIPS au CRI de l'Ecole des Mines pour m'avoir si gentiment écouté, aidé et accueilli quelque temps. Merci à Corinne pour tous ses commentaires sur les chapitres concernant PIPS.

Finalement, je remercie les rapporteurs et autres membres du jury pour avoir accepté de jouer ce rôle et pour l'avoir si bien rempli. Merci à Michel Lemoine et à Véronique Donzeau-Gouge pour leurs relectures attentives de la thèse et toutes leurs suggestions pour son amélioration, ainsi que pour des discussions intéressantes.

Résumé

Nous proposons une méthode de rétro-ingénierie de vieux codes Fortran77 à l'aide de spécifications algébriques : une spécification *concrète* décrit les structures de données spécialisées et les algorithmes efficaces du code et une spécification *abstraite* décrit la finalité du calcul dans les termes du domaine d'application. Les deux sont liées par des spécifications intermédiaires intégrant un par un les choix successifs d'implémentation qui expliquent le passage à la spécification concrète, depuis la spécification abstraite.

La construction de la spécification concrète est rendue difficile par de nombreuses caractéristiques de Fortran77, largement exploitées par les programmeurs de l'époque. Nous proposons diverses analyses et transformations du code et notamment une méthode d'inférence automatique des *profils* des *opérations* spécifiant les sous-programmes Fortran. La spécification abstraite est une description de la compréhension intuitive du code acquise par le rétro-ingénieur pendant la partie de l'analyse qui doit être effectuée à la main.

La spécification concrète doit être suffisamment proche du code pour assurer qu'elle en est une description exacte et la spécification abstraite doit être justifiée par une comparaison avec la spécification concrète, à travers les spécifications intermédiaires. Pour des raisons pratiques, l'élaboration des spécifications, ainsi que la justification de la spécification abstraite, sont effectuées sur un fragment à la fois du code entier. Les abstractions que nous désirons exprimer ne satisfont pas toujours l'inverse de la relation d'*implémentation correcte* entre les deux spécifications du fragment de code. C'est pourquoi nous proposons une nouvelle relation, appelée *abstraction correcte*, entre spécifications algébriques.

Nous avons testé notre méthode sur de nombreux exemples extraits de plusieurs codes réels fournis par EDF et nous présentons les spécifications concrète et abstraite, ainsi que les sept spécifications intermédiaires, du fragment de code qui constitue notre exemple principal.

mots clés : rétro-ingénierie, méthodes formelles, spécifications algébriques, raffinement, Fortran, analyse statique, régions de tableaux

Abstract

We propose a method for the description of legacy Fortran77 programs using algebraic specifications: a *concrete* specification describing the specialised data-structures and efficient algorithms of the code and an *abstract* specification describing its purpose in the context of the application domain. The two are linked by intermediate specifications describing the successive choices of design or implementation which explain how the concrete specification is obtained from the abstract specification.

Several characteristics of the Fortran77 language make the concrete specification difficult to obtain, but were often used by the programmers of the time. We propose different analyses and transformations, both automatic and manual, of the code and, in particular, a way to automatically infer the *profiles* of the *operations* specifying the Fortran sub-programs. The abstract specification is a description of the intuitive understanding of the code that the reverse engineer acquires during the part of the analysis which must be carried out manually.

The concrete specification must be sufficiently close to the code to inspire confidence in the accuracy of its description and the abstract specification is then checked by comparing it to the concrete specification, via the comparison of each successive pair of intermediate specifications. For practical reasons, the construction of the specifications and the justification of the abstract specification must be carried out on one fragment at a time of the whole program. The abstractions which we want to express are not always the inverse of a *correct implementation* of one specification of the program fragment by another. For this reason, we propose a new relation between algebraic specifications, called *correct abstraction*.

We have tested our method on numerous examples taken from real programs supplied by EDF and we present the concrete and abstract specifications, as well as the seven intermediate specifications, of the program fragment which we chose as our main example.

key words: reverse engineering, formal methods, algebraic specifications, refinement, Fortran, static analysis, array regions

Table des matières

Introduction

EDF, comme bien d'autres entreprises, s'est constitué un patrimoine important de logiciels écrits il y a 15 ans, 20 ans ou plus. Ces logiciels ont été créés à une époque où la priorité était à l'économie de la place mémoire et du temps d'exécution et non pas à l'aide à la maintenance et à l'évolution future du code. Ces "vieux" logiciels sont peu documentés et ont subi de nombreuses modifications au cours des années et la connaissance que les ingénieurs en avaient lors de leur mise en œuvre a été diluée par des changements successifs de personnels. Cependant, ils tournent sans problème depuis des années et sont nécessaires à la bonne marche de l'exploitation.

C'est pourquoi il est intéressant de retrouver la maîtrise complète de ces logiciels, afin de mieux les maintenir (à moindre coût, avec moins de délais), les faire évoluer, voire de pouvoir en extraire le savoir faire et le retour d'expérience pour en bénéficier lors de l'élaboration de nouvelles applications. Car l'effort humain et financier de développement de ces "vielles" applications se chiffre en dizaines, voire en centaines d'ingénieurs/an et il peut ne pas être judicieux d'oublier complètement le savoir faire accumulé, et de recommencer *ex nihilo* le développement extrêmement coûteux de leur remplaçant.

La maîtrise de ces logiciels passe par la construction de la bonne description. Il faut une description très précise, qui mette en évidence les conventions de programmation, les hypothèses implicites, le modèle sous-jacent (du réseau électrique, par exemple), ... Or, non seulement la documentation existante est souvent lacunaire et n'a pas toujours été mise à jour après chaque modification du code, mais aussi elle décrit le logiciel tel qu'il est fait, sans expliciter les choix de conception ou d'implémentation qu'il intègre. Or ces choix déterminent les caractéristiques fondamentales du code que l'on a besoin de connaître pour évaluer l'impact des modifications souhaitées aux spécifications, aux conditions d'utilisation, à la modélisation, aux algorithmes, etc.

Le but de cette étude est d'apporter une réponse à ce problème, adaptée en particulier aux codes Fortran77 employés par le service Etudes de Réseaux de la Division Recherche et Développement d'EDF.

Nous proposons de décrire le code à l'aide des *spécifications formelles* et de construire non pas une spécification, mais aux moins deux :

– la spécification *abstraite* est la description minimale nécessaire du calcul effectué par le code, en des termes les plus classiques et bien connus possibles pour une lisibilité maximale. Elle décrit **ce qui est fait** par le code, sans décrire **comment il est fait**. Elle devrait donc être compréhensible par un ingénieur électricien non informaticien.

– La spécification *concrète* est une description fidèle du code : les structures de données spécialisées et algorithmes efficaces qui sont employés dans la réalisation de ce calcul.

Les liens que nous pouvons établir entre les deux spécifications, grâce à leur caractère formel, mettent en évidence les **conditions d'application** du code.

La construction *a posteriori* des spécifications abstraite et concrète d'un logiciel existant, est un travail de rétro-ingénierie. Or les études de rétro-ingénierie déjà entreprises et décrites dans la littérature concernent surtout la restructuration plus ou moins automatique afin de faciliter la compréhension du code. Nous allons nous appuyer sur la notion très puissante et très générale d'*implémentation d'une spécification algébrique* qui permet des étapes d'abstraction beaucoup plus importantes et plus variées que la simple restructuration. Toutefois, nous ne pouvons pas espérer effectuer cette abstraction de façon entièrement automatique. En effet, nous sommes convaincus que l'abstraction exprime la **compréhension** intuitive du code apportée par une partie de l'analyse du code Fortran qui doit être effectuée "à la main", même si des outils automatiques peuvent apporter une aide précieuse dans le reste de l'analyse.

Notre étude se décompose naturellement en deux volets :

1. l'analyse initiale du code Fortran et l'élaboration de la spécification concrète et
2. l'élaboration de la spécification abstraite et la relation formelle entre les deux spécifications.

Chaque volet constitue à lui seul un domaine de recherche assez vaste, alors nous avons décidé de limiter notre étude

- en ce qui concerne le premier volet : à cerner les spécificités de notre cas, avant d'étudier plus en détail certains aspects de l'analyse statique, mais sans réaliser des outils opérationnels,
- en ce qui concerne le deuxième volet : à l'étude très rigoureuse de la justification de l'abstraction d'un exemple de code.

Nous essayons de réaliser une couverture des deux domaines suffisamment complète pour pouvoir fournir dans cette thèse un début de méthode de rétro-ingénierie des codes Fortran employés au service Etudes de Réseaux d'EDF.

Les deux volets de l'étude correspondent aux deux parties principales de cette thèse.

- Avant la première partie, il y a deux chapitres de "mise en scène" :
  - dans le Chapitre 1 nous situons notre approche de rétro-ingénierie dans le contexte des *méthodes formelles* et la comparons à d'autres approches proposées dans la littérature.
  - Dans le Chapitre 2, nous décrivons les codes Fortran fournis par EDF et dont nous tirons les exemples cités tout au long de la thèse, avec une description détaillée du fragment de code qui constitue notre exemple principal.
- La Partie I décrit l'analyse initiale du code et la construction de la spécification concrète. Pour des raisons pratiques et dans l'absence de modules (qui n'existent pas en Fortran77), nous devons décider la décomposition du code entier en fragments afin d'effectuer l'analyse détaillée, la spécification concrète et l'abstraction sur un fragment à la fois.
  - La Partie I commence par le Chapitre 3, qui contient une discussion des problèmes posés par la construction d'une spécification algébrique d'un code Fortran77 et un rappel du rôle de la spécification concrète. Ce chapitre se termine par un résumé des étapes successives d'analyse que nous proposons dans le reste de la première partie.
  - Le Chapitre 4 est une présentation générale des techniques et outils opérationnels d'analyse statique sous la perspective de l'aide à la rétro-ingénierie de codes Fortran77.
  - Le Chapitre 5 enchaîne avec une description plus détaillée des possibilités et des limitations de l'outil *PIPS*. Il termine par notre définition des *vrais paramètres d'entrée et de sortie* des sous-programmes Fortran77 et une discussion de l'adéquation de leur caractérisation par les *régions* calculées par *PIPS*.
  - Le Chapitre 6 propose quelques transformations du code Fortran à effectuer avant les analyses statiques définitives (élimination de quelques entorses à la norme *ANSI*, ainsi que des aspects "non fonctionnels", simplification des structures de contrôle, décomposition des sous-programmes et traitement des variables prenant plusieurs valeurs successives), ainsi que d'autres transformations pour faciliter la compréhension du code.
  - Le Chapitre 7 décrit une analyse à effectuer suite aux transformations préliminaires. Il s'agit de l'inférence, à l'aide de *PIPS*, des *profils* des *prédicats* et *opérations* de la spécification concrète, qui constitue notre résultat principal du premier volet de l'étude.
  - Le Chapitre 8 propose d'autres analyses à effectuer au même moment, afin de préciser les *sortes*, *domaines* des opérations *partielles* et structure en *types abstraits de données* de la spécification concrète.
  - Finalement, le Chapitre 9 décrit l'élaboration de la spécification concrète suite aux transformations et analyses proposées, illustrée par la spécification concrète du fragment de code qui constitue notre exemple principal.

- La Partie II traite l'élaboration et la justification de la spécification abstraite de notre exemple principal.
  - Dans le Chapitre 10, nous décrivons comment nous avons obtenu la spécification la plus abstraite et puis une suite de spécifications à des niveaux d'abstraction intermédiaires entre celle-ci et la spécification concrète.
  - Dans le Chapitre 11, nous rappelons la définition de l'*implémentation correcte* d'une spécification algébrique, et expliquons pourquoi la spécification concrète de notre exemple **n'est pas** une implémentation correcte de la spécification abstraite.
  - Finalement, dans le Chapitre 12 nous présentons notre résultat principal en ce qui concerne le deuxième volet de cette étude : une caractérisation de l'implémentation d'un fragment d'un code entier optimisé et une proposition de définition de l'*abstraction correcte* d'une spécification algébrique d'un fragment de code, dans le cas où celui-ci n'est pas un module bien encapsulé.
- Cette deuxième et dernière partie est suivie par la Conclusion.

Chapitre 1

Notre approche

Notre approche s'inspire du processus de **développement** de logiciels à l'aide des méthodes formelles [Wir90]. Rappelons ce processus.

1.1 Le développement à l'aide des méthodes formelles

Il commence par l'élaboration d'une spécification formelle *abstraite*, c'est-à-dire : déclarative, décrivant le comportement ou les résultats attendus du logiciel et non pas une procédure calculatoire pour les obtenir. La spécification abstraite peut même être une description incomplète du comportement, car seuls les aspects essentiels doivent être précisés à ce niveau.

Ensuite, une spécification *concrète* est construite. Dans un langage formel *"wide spectrum"* (c'est-à-dire : intégrant un langage de programmation avec une sémantique formelle, et un langage non exécutable de spécifications formelles), celle-ci peut être entièrement écrit dans le langage de programmation. Dans un langage *exécutable* de spécifications formelles, la spécification *concrète* est exécutable (voir les *spécifications constructives* de [LEW96]). Dans notre cas de spécifications algébriques exprimées dans un langage non exécutable, il faut que la spécification concrète soit la plus proche possible d'un programme afin d'avoir le plus de confiance possible dans l'étape de codage qui ne peut pas être vérifiée formellement. Il faut alors une spécification concrète *OP-complète* [EKMP82] (intuitivement, ceci veut dire calculatoire, constructive) qui est une *extension conservatrice* de la spécification du langage Fortran [1].

Ensuite, il faut démontrer formellement que la spécification *concrète* satisfait la spécification *abstraite*. Dans le cas de deux spécifications algébriques, il s'agit de démontrer que la première est une *implémentation correcte* de la deuxième, comme nous l'expliquons avec plus de détails dans le Chapitre 11.

En fait, le passage de la spécification abstraite vers la spécification concrète est souvent décomposé, en construisant plusieurs spécifications intermédiaires. Chaque nouvelle spécification intermédiaire intègre alors l'un des *choix d'implémentation* qui permettent de s'approcher d'une spécification concrète. Ces choix peuvent inclure :

- un choix d'une possibilité parmi plusieurs afin d'éliminer le caractère non-déterministe de la spécification abstraite,
- un choix d'algorithme pour une procédure de calcul,
- un choix de représentation des données ou
- la décision de pré-calculer et de mémoriser le résultat d'un calcul, au lieu de le recalculer à plusieurs reprises.

Si le critère de satisfaction d'une spécification par une autre est transitif, alors il suffit de prouver que chaque spécification intermédiaire satisfait la précédente afin de prouver que la spécification la

1. L'*extension conservatrice* sera définie dans le Chapitre 11. Intuitivement, une *extension conservatrice* de la spécification du langage Fortran est une spécification "au-dessus" de la spécification de Fortran qui n'"interfère" pas avec elle de façon à ne pas modifier ses conséquences logiques.

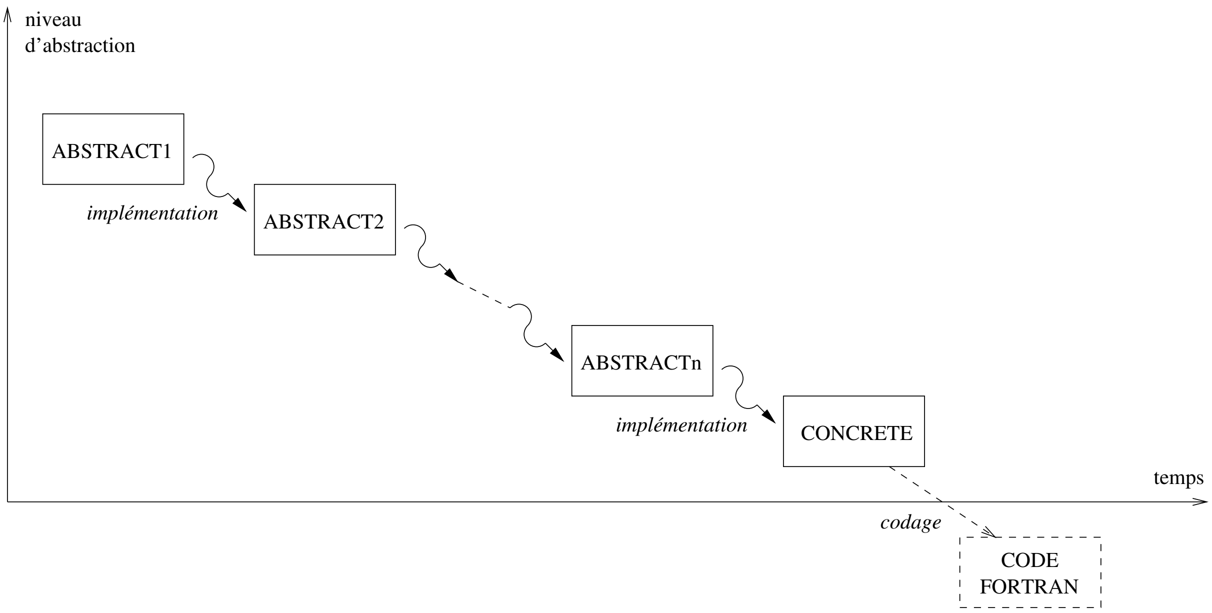


FIG. 1.1: Le développement de logiciels à l'aide des méthodes formelles

plus abstraite satisfait la spécification concrète. C'est la notion du développement d'un logiciel par le *raffinement pas à pas* [2] de sa spécification.

Le processus de développement de logiciels à l'aide des méthodes formelles est illustré dans la Figure 1.1.

Certaines approches des méthodes formelles s'appuient sur des "bibliothèques" de transformations élémentaires dont la "correction" est déjà prouvée. Celles-ci peuvent être appliquées automatiquement : il s'agit de génération automatique ou semi-automatique (par exemple l'*Atelier B* [3] [Abr96]) de codes. Sinon, chaque spécification est élaborée à la main et vérifiée semi-automatiquement, à l'aide d'un démonstrateur de théorèmes.

1.2 Notre approche de rétro-ingénierie à l'aide des méthodes formelles

Notre approche est basée sur l'inverse du développement de logiciels à l'aide des méthodes formelles. Elle est illustrée dans la Figure 1.2.

Nous proposons de commencer par une analyse en partie manuelle et en partie à l'aide d'outils automatiques, du code et de sa documentation, y compris les commentaires. Cette analyse prépare l'élaboration de la spécification concrète, mais permet aussi de **comprendre** le code.

L'analyse terminée, nous élaborons une *spécification concrète* formelle qui est une *extension conservatrice OP-complète* de la spécification de Fortran. Elle est exprimée donc en termes d'opérations de base sur des entiers, tableaux d'entiers ..., et éventuellement en termes d'autres opérations et structures de données qui ne sont pas présentes en Fortran mais qui sont simulées par le code spécifié. Cette spécification doit être suffisamment proche du code pour convaincre qu'elle en est la représentation fidèle, et pour permettre de localiser facilement l'implémentation dans le code de chaque symbole de la spécification concrète. Toutefois, nous voulons une spécification concrète qui sert aussi à documenter le code et à aider à le comprendre et à le maintenir, elle doit être plus facile à comprendre que le code lui-même, grâce

2. Le terme *raffinement* est employé dans un sens plus ou moins flou dans le domaine des méthodes formelles. Il a aussi une définition très précise dans les spécifications algébriques, que nous expliquons dans la deuxième partie de cette thèse.

3. L'*Atelier B* est une marque déposée de STERIA, 12, Paul Dautier, BP 58, 78142 Vélizy, France, voir http://www.atelierb.societe.com/.

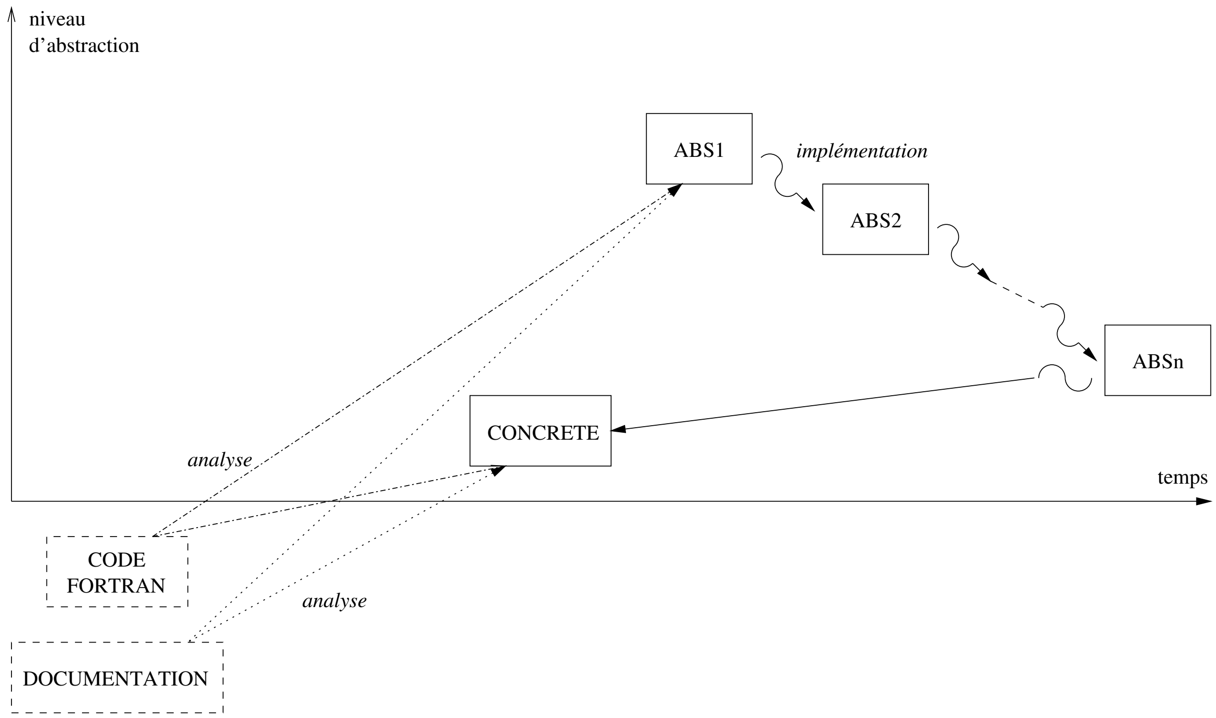


FIG. 1.2: Notre approche de rétro-ingénierie à l'aide des méthodes formelles

à sa structure, aux conventions de codage explicitées et à la "mise à plat" des structures de contrôle compliquées.

Nous décrivons en détail l'analyse du code et l'élaboration de la spécification concrète dans la première partie de ce rapport.

Ensuite, nous élaborons une *spécification abstraite* formelle décrivant notre compréhension de la fonctionnalité du code. Cette spécification a pour vocation d'être compréhensible par un ingénieur EDF expert en réseaux électriques, qui n'est pas informaticien. C'est une description du modèle du réseau et de la finalité des calculs effectués. Il y a donc une différence importante entre le niveau d'abstraction de la spécification concrète et celui de cette spécification la plus abstraite. Évidemment, le rétro-ingénieur non expert en réseaux électriques aura parfois besoin du concours d'un expert à ce stade.

Une fois les deux spécifications construites, nous devons vérifier la satisfaction de la spécification abstraite par la spécification concrète. Nous employons les mêmes techniques que dans le développement de logiciels : la décomposition en plusieurs étapes d'implémentation (qui peut être inspirée par notre compréhension du code sur plusieurs niveaux d'abstraction) et la vérification de la "correction" de l'abstraction. Comme il n'existe pas de définition d'*abstraction correcte*, nous proposons de vérifier la correction de l'implémentation de la spécification abstraite par la spécification concrète. Cependant, à la fin de cette thèse, nous proposons notre propre définition d'*abstraction correcte*. Évidemment, en cas déchec, la vérification de la "correction" de l'abstraction peut indiquer des modifications à apporter à la spécification abstraite.

La construction de la spécification la plus abstraite et des spécifications intermédiaires, ainsi que leur validation, sont illustrées sur un exemple dans la deuxième partie de cette thèse.

Une fois la spécification la plus abstraite et les spécifications intermédiaires finalisées, nous disposons de deux descriptions principales du code : une documentation de l'implémentation en Fortran et une description beaucoup plus abstraite de sa fonctionnalité en termes de calculs sur un modèle du réseau électrique. Ces descriptions étant exprimées dans un langage formel, il est possible de vérifier formellement que le code possède les propriétés qui peuvent nous intéresser. Par ailleurs, la vérification de la spécification abstraite passe par la définition d'une correspondance entre les sortes, prédicats et opérations de la spécification abstraite et ceux de la spécification concrète (en passant par toutes les spécifications intermédiaires). Cette correspondance permet de localiser dans le code l'implémentation de tout aspect qui nous intéresse de la spécification la plus abstraite, ou des autres spécifications.

En conclusion, nous proposons une analyse aidée autant que possible par des outils automatiques ou semi-automatiques, mais avec une partie manuelle incontournable, car nécessaire à la compréhension du code par le rétro-ingénieur. En effet, nous croyons illusoire, dans l'état actuel de l'art,

- l'idée d'un outil automatique capable de construire une spécification formelle de bonne qualité à partir du code Fortran,
- toute comme l'idée de l'abstraction automatique vers le niveau que nous avons envie d'atteindre dans la spécification la plus abstraite.

La compréhension des boucles, en particulier, nécessite en général de trouver l'*invariant* de la boucle. Ce problème est indécidable dans le cas général.

Nous ne proposons pas d'effectuer l'abstraction pas à pas de la spécification concrète, en construisant d'abord la spécification concrète, ensuite une spécification légèrement moins concrète, et ainsi de suite. Une démarche pas à pas "bottom-up" peut être utile dans la simplification des structures de contrôle du code Fortran, mais nous proposons de l'effectuer pendant la phase initiale d'analyse du code, à l'aide d'outils automatiques de restructuration, ou à la main. Une spécification algébrique du code à ce niveau d'abstraction peut être aussi illisible que le code lui-même, alors il nous semble plus facile de travailler avec d'autres représentations, telles qu'organigrammes de petits fragments de code, pour les transformations manuelles ou semi-automatiques. Dans notre approche, le résultat de la somme des transformations de simplification des structures de contrôle est intégré avec notre compréhension intuitive du code dans une spécification formelle beaucoup plus abstraite que la spécification concrète et nous proposons de vérifier à la main (ou à l'aide d'un démonstrateur de théorèmes) la "correction"de cette spécification abstraite.

1.3 Le choix de *CASL*

Il existe de nombreux langages de spécification algébrique, qui sont plus ou moins expressifs et associés à plus ou moins d'outils d'aide à la vérification des spécifications. Le stage de DEA qui a précédé cette thèse, ainsi que plusieurs études menées au sein du service Etudes de Réseaux de la Division Recherche et Développement d'EDF, ont été effectués à l'aide du langage *LSL* ("Larch Specification Language") [GH93], qui est associé à l'environnement *Larch* de vérification.

Cependant, nous avons choisi d'exprimer les spécifications de cette thèse dans le langage *CASL* [oLD99]. *CASL* est un nouveau langage (pour lequel il existe peu d'outils opérationnels pour l'instant) défini par un consortium de chercheurs européens dans le domaine des spécifications algébriques, dont des membres du LSV (le laboratoire d'accueil de cette thèse). *CASL* représente alors l'état de l'art. En particulier, il intègre trois aspects qui ne sont pas présents dans *LSL* et qui ont été très importants dans notre étude. Ce sont :

- le *"hiding"*,
- les opérations *partielles*
- et les *sous-sortes*.

Parmi ces trois aspects, nous pouvons considérer les sous-sortes comme du "sucre syntaxique" défini au-dessus des opérations partielles. Elles n'augmentent pas réellement le pouvoir d'expression du langage, mais permettent de se servir d'un concept bien connu et très utile qui rend les spécifications plus lisibles.

Les deux autres aspects augmentent réellement le pouvoir d'expression du langage et apportent de nouvelles possibilités d'implémentation.

Par exemple : certains sous-programmes Fortran77 n'acceptent en entrée que certaines combinaisons de valeurs des arguments. Cette restriction apporte une information supplémentaire sur le sous-programme qui peut aider à le comprendre. Les opérations partielles permettent de l'expliciter dans la spécification concrète, empêchent une spécification abstraite du sous-programme avec un domaine de définition trop grand et peuvent expliquer alors une implémentation qui ne serait pas correcte sur l'opération totale.

Le mécanisme de "hiding" conduit à une extension de la définition de l'implémentation correcte qui n'est pas possible pour les spécifications écrites en *LSL* mais que nous avons utilisée à de nombreuses reprises et qui nous semble essentielle dans toute comparaison de deux spécifications à des niveaux très différents d'abstraction, d'un code réel.

1.4 Comparaison avec d'autres approches de rétro-ingénierie

1.4.1 D'autres approches basées sur les méthodes formelles

Il y a peu d'autres approches à la rétro-ingénierie basées sur les méthodes formelles et, à notre connaissance, aucune basée sur le concept très général d'*implémentation correcte* employée dans le *raffinement* des spécifications algébriques.

Une autre méthode de rétro-ingénierie des codes Fortran à l'aide des spécifications algébriques

[Che98] aborde exactement le même sujet que nous mais en partant de certaines hypothèses que nous ne partageons pas, et avec des objectifs différents. Ainsi, elle suppose qu'une analyse manuelle préalable du code a permis de détecter le "domaine" de certaines des variables du code Fortran dont celui-ci n'est pas nécessairement le type Fortran déclaré (`ENTIER`, par exemple) mais parfois un type n'existant pas en Fortran77, tel que *pile*. Elle suppose que ces informations "acquises" suffisent à déduire les "domaines" (ou sortes) de toutes les variables du "module" de code analysé, à l'aide d'une simple inférence (sans sous-types) de type.

Le but de la méthode proposée dans [Che98] n'est pas d'élaborer une spécification abstraite, mais uniquement une spécification algébrique extraite du code Fortran et "simplifiée" à la main de façon ad-hoc afin de la rendre plus lisible. Elle ne s'appuie donc pas sur la notion d'implémentation correcte d'une spécification algébrique. Par ailleurs, sa méthode se veut généralisable à d'autres langages et styles de programmation, tandis que nous nous restreignons aux codes Fortran d'EDF.

[Che98] se pose le même problème que nous de la recherche des "vrais" paramètres des sous-programmes et se donne une définition de ceux-ci similaire à la notre, mais qui ne prend en compte ni l'approximation inhérente à toute analyse statique en présence de nombreux chemins d'exécution, ni le cas spécial des tableaux.

Elle propose d'extraire les axiomes de la spécification algébrique à l'aide d'une technique de recherche de blocs de code dont la structure de contrôle correspond à des "schémas" typiques stockés dans une base de connaissances (cette technique appartient à la famille de recherche de *clichés* décrite dans la Section 1.4.2 ci-dessous). Dans une étape similaire à notre décomposition des sous-programmes, elle propose d'employer le *"slicing"* (voir aussi Section 1.4.2) sur chaque paramètre de sortie des blocs de code identifiés. C'est le développement des techniques automatisables de recherche des blocs de code, de génération des axiomes spécifiant chaque bloc, et ensuite des axiomes spécifiant tout le "module" de code, qui constitue la plus grand part de la description de la méthode.

Nous ne proposons pas de technique automatique pour l'élaboration des axiomes de la spécification concrète, alors la méthode de [Che98] est surtout **complémentaire** à la notre.

REDO

L'objectif du projet *ESPRIT REDO* était la définition de techniques et d'outils d'aide à la maintenance. En particulier, certains partenaires ont étudié la transformation semi-automatique de codes programmés en *Cobol* ou Fortran, vers des spécifications formelles. Dans les travaux les plus proches de notre approche [BLB93], un fragment de code est d'abord traduit dans un autre langage "universel" défini dans le cadre du projet. Ce langage a une sémantique plus simple et plus propre que celles de *Cobol* et de Fortran : nombre réduit de structures de contrôle ; paramètres d'entrée et de sortie des sous-programmes explicitées, y compris les variables globales ; *appel par référence* remplacé par *appel par valeur* ; ... Ensuite, une spécification fonctionnelle équationnelle non typée est automatiquement générée et transformée dans une forme normale. Finalement, le rétro-ingénieur transforme cette spécification à la main à l'aide d'heuristiques, afin de la "simplifier" et de comprendre ainsi la fonctionnalité du fragment de code. La représentation abstraite devrait être intégrée ensuite dans une représentation orientée-objet du code, exprimée dans une extension orientée-objet du langage *Z* de spécifications formelles, mais la construction de la représentation orientée-objet semble avoir été étudiée surtout pour le cas des codes écrits en *Cobol*.

Il s'agit donc d'une phase de transformation automatique du code suivie par la construction automatique d'une spécification à un niveau bas d'abstraction. Le niveau de la spécification construite automatiquement semble similaire au niveau de notre spécification concrète, mais la spécification formelle n'est pas typée, encore moins structurée en types abstraits de données, et étant générée automatiquement, ne peut pas employer des noms de variables et de fonctions qui indiquent ce qu'ils représentent dans le domaine modélisé par le code. L'abstraction est effectuée à la main sur cette représentation, comme dans notre cas, mais sans un cadre comme celui de l'implémentation correcte des spécifications algébriques pour aider à sa vérification. Les partenaires de ce projet insistent comme nous sur la nécessité de l'intervention humaine pour atteindre un niveau relativement abstrait dans la représentation.

"Maintainers Assistant"

Le *"Maintainers Assistant"* [Bul90] est un outil semi-automatique d'aide à la rétro-ingénierie, basé sur une "bibliothèque" de transformations de spécifications. Il a été prouvé pour chaque transformation dans la bibliothèque que, appliquée sous les bonnes conditions,

- soit elle ne modifie pas la sémantique du code,
- soit son inverse est un raffinement fonctionnel.

Le code est d'abord traduit dans un langage formel typé *"wide spectrum"*, appelé *WSL*, dans lequel les transformations sont définies (il n'y a pas encore de traducteur pour Fortran). Ensuite, l'outil aide le rétro-ingénieur à sélectionner et à appliquer des transformations.

Comme l'approche précédente (mais qui n'a pas, à notre connaissance, été concrétisée par le développement d'un outil opérationnel), celle-ci consiste à traduire le code automatiquement dans une spécification formelle et puis à effectuer pas à pas des transformations afin de l'abstraire. Les développeurs de l'outil admettent qu'il est le plus utile dans la restructuration (i.e. simplification des structures de contrôle) du code, où l'intervention de l'utilisateur permet des transformations plus intéressantes que celles obtenues par des outils automatiques de restructuration. Par contre, les transformations de la bibliothèque sont limitées en ce qui concerne l'abstraction des données et ne permettent pas d'atteindre un niveau très haut d'abstraction [Ben98].

Génération de spécifications formelles par exécution symbolique

[CdLM95] propose d'effectuer l'exécution symbolique d'un fragment de code *C* afin de générer une spécification formelle axiomatique (non typée) sous forme de formules en logique de premier ordre exprimant ses *pré-* et *postconditions* [Hoa69]. L'intervention de l'utilisateur peut être nécessaire pour l'exécution symbolique des boucles, ce problème étant indécidable en général.

Cette approche concerne uniquement la génération automatique des spécifications formelles à un niveau très bas d'abstraction. Nous étudions la génération de spécifications à l'aide de l'exécution symbolique dans la Section 3.5 du Chapitre 3 et dans le Chapitre 4.

AUTOSPEC et *SPECGEN*

L'outil *AUTOSPEC* [GC97] génère des spécifications formelles axiomatiques de codes *C* d'à peu près la même façon que dans l'approche précédente.

De plus, l'outil *SPECGEN* associé [GC99b] aide à l'abstraction des spécifications de bas niveau générées par *AUTOSPEC*, en effectuant de certaines façons prédéfinies, soit le renforcement de la précondition, soit le relâchement de la postcondition.

Le paradigme des *transformateurs de prédicats* employé par les auteurs de [GC99b] intègre une notion d'état : les pré- et postconditions sont exprimées sur le même ensemble de variables, mais la précondition est vraie pour leurs valeurs **avant** l'application de la fonction spécifiée, et la postcondition est vraie pour leurs valeurs **après** l'application de la fonction. Cet aspect rend difficiles des comparaisons directes avec notre formalisme de raffinement. Cependant, il nous semble que la notion d'abstraction employée dans *SPECGEN* est équivalente à l'implication logique entre deux spécifications sur la même signature. La simulation d'une structure de données par une autre n'est pas modélisée.

Les auteurs proposent dans [GC99a] d'employer d'autres outils plus classiques d'analyse statique pour guider l'utilisation d'*AUTOSPEC* et de *SPECGEN*. Nous proposons aussi d'employer des outils d'analyse statique, mais pas dans le même but.

Abstraction dans *RWSL*

[YLZ98] définit un langage formel *"wide spectrum"* orienté-objet et avec une représentation d'intervalles de temps (*RWSL*) et une "taxonomie" d'abstractions définies formellement.
Contrairement à des approches antérieures, celle de [YLZ98] aborde les problèmes de

- la rétro-ingénierie de programmes temps-réel et
- la définition formelle d'abstractions très générales et donc potentiellement capables de caractériser des grands pas d'abstraction.

Toutefois, il s'agit d'un programme de recherche qui débute.

1.4.2 D'autres approches

Une autre approche intéressante de rétro-ingénierie concerne le développement des outils de recherche de *clones* (instances de duplication du même fragment de code, ou presque, voir [BYM+98]) ou de *clichés* (par exemple, [RS97, QWZ97]). Dans ce dernier cas, il s'agit de définir des structures de données ou algorithmes souvent retrouvés dans des codes, une représentation de ces *clichés* (par exemple, sous forme de graphe) et une méthode de recherche d'instances de ces clichés dans le code (intégrant la possibilité de relâcher des contraintes de la description). Des techniques et représentations issues de l'*intelligence artificielle* sont parfois utilisées.

Il nous semble que la recherche de *clichés* serait très utile à EDF après la rétro-ingénierie d'un certain nombre de fragments de code. En effet, le domaine des codes employés au département Etudes de Réseaux est très restreint et les anciens codes sont souvent écrits dans un "style maison". Par conséquent, nous retrouvons dans différents codes des structures de données ou algorithmes (souvent spécialisés à ce domaine) qui sont sensiblement les mêmes. Une fois la rétro-ingénierie effectuée d'une structure de données ou d'un algorithme susceptible d'être employé dans d'autres codes, on pourrait définir le *cliché* correspondant et en chercher des instances dans d'autres codes. En théorie, ceci permettra de bénéficier d'un effet "boule de neige" : avec la croissance de la quantité de code analysé, la rétro-ingénierie deviendrait de plus en plus automatique. Cependant, nous n'avons pas eu le temps d'étudier cette possibilité dans le cadre de cette thèse.

Toutes les autres approches à la rétro-ingénierie concernent l'analyse du code qui intervient, dans notre approche, avant l'élaboration de la spécification concrète et est décrite dans la première partie de cette thèse, en la comparant à certains des travaux cités ici. Elles peuvent être regroupées dans les deux catégories suivantes :

1. le développement d'outils de *visualisation* des codes (par exemple [MWT94] et [HIBM97]) et de *"slicing"* [4], par exemple [BMW93, NEK94, BBGK97] ;
2. le développement d'outils de restructuration architecturale du code en modules (en employant, par exemple, le *"concept analysis"* [LS97, SR97]), d'extraction de classes d'objets (par exemple, en Fortran, [OT93, AC94, SB96]) ou de types abstraits de données [GKS97].

Remarquons que très peu d'outils de rétro-ingénierie sont développés pour le Fortran : dans tous les exemples ci-dessus (y compris les outils de recherche de *clones* ou de *clichés*) seuls les outils d'extraction de classes d'objets traitent des codes Fortran.

Il y a aussi dans la littérature sur la rétro-ingénierie quelques études de cas assez intéressantes [BM82, Byr91, BM97], ainsi que des études générales sur les problèmes posés par la rétro-ingénierie et la nature de l'abstraction [RSW95, JOR90, War95, HP92]. Finalement, des bibliographies [vdBKV96] [5] et le tutoriel [Mül96] offrent un survol du domaine.

4. Le *"slice" arrière* est l'extrait du code permettant de calculer la valeur d'une variable donnée à un point donné de l'exécution. Le *"slice" avant* étant l'extrait qui dépend de cette valeur [Wei84].

5. Voir aussi http://liinwww.ira.uka.de/bibliography/SE/reengineering.html
et http://www.informatik.uni-stuttgart.de/ifi/ps/reengineering/

Chapitre 2

Nos exemples

Nous avons développé et expérimenté notre approche à la rétro-ingénierie sur plusieurs exemples de codes réels et opérationnels. En effet, EDF a fourni trois codes tirés d'applications de conduite et surveillance de réseaux électriques et choisis parmi les plus représentatifs d'éventuels travaux de réingénierie futurs. Ces codes sont :

1. *COURCIRC*,
2. le code `QIOUVR` de l'application *SILODI*
3. et le code *ESTIME* de l'application *SYSDIC*.

Nous décrivons brièvement ces codes, avant de décrire de façon plus détaillée le fragment du dernier qui constitue l'exemple principal sur lequel nous avons expérimenté notre approche à la construction des spécifications algébriques successives et à leur validation.

Nous avons disposé, comme seule documentation de ces codes, des commentaires ainsi que des spécifications techniques d'*ESTIME*. Ces dernières sont adressées aux spécialistes des réseaux électriques et expliquent le bien fondé des calculs matriciels mis en œuvre par le code, mais ne donnent pas de renseignements sur sa conception.

2.1 *COURCIRC*

Nous avons analysé une partie du logiciel qui exclut l'interface utilisateur et dans laquelle certains appels aux routines de "bibliothèque" extérieure sont éliminés ou remplacés. Ce fragment est composé de 110 sous-programmes et de 24854 lignes de code. Nous avons restreint l'analyse statique automatique aux sous-programmes appartenant à l'arbre des appels de `PCXCAL`, qui contient 57 routines et 10412 lignes de code.

Tous les sous-programmes de *COURCIRC* sont programmés et commentés de façon méthodique et uniforme. Les commentaires sont nombreux et instructifs. Les problèmes posés par l'analyse de *COURCIRC* sont les suivants :

- sa taille, beaucoup plus important que les autres codes fournis (mais pas exceptionnelle pour un code EDF)
- son emploi d'une "bibliothèque" de fonctions Fortran pour la simulation de l'allocation dynamique de la mémoire et qui repose sur une `EQUIVALENCE` entre plusieurs blocs `COMMON` (voir la Section 3.3.2 du Chapitre 3) et
- la structure de contrôle utilisée dans plusieurs sous-programmes qui simule le `break` à l'aide d'un `GOTO` au milieu d'une boucle `DO`. Cette structure de contrôle complique l'analyse statique (voir la Section 5.1.6 du Chapitre 5).

```
QIOUVR
    MESSER
    ODIBAR
        MESSER
        STOPRO
            MESSER
    ODIDAM
        STOPRO
            MESSER
    NOUPOI
        MESSER
    VULCAN
    ODIDAV
        STOPRO
            MESSER
    SUIVAN
        MESSER
```

FIG. 2.1: Graphe des appels de `QIOUVR`

2.2 *QIOUVR*

Le sous-programme `QIOUVR` de l'application *SILODI* a pour rôle de déterminer la protection dans un poste qui s'est ouverte suite à un court-circuit. Son graphe des appels est présenté dans la Figure 2.1. Nous avons analysé le sous-programme `QIOUVR`, ainsi que les autres appelés directement ou indirectement par lui, regroupant 949 lignes de code, au total.

Les structures de contrôle des sous-programmes sont compliquées. `QIOUVR` est composé de deux boucles imbriquées réalisées à l'aide de `GOTO` et contenant de nombreux branchements à l'intérieur et plusieurs sorties. `ODIDAV` contient même une boucle *non réductible* (voir la Section 4.3.1 du Chapitre 4).

La structuration en sous-programmes de l'ensemble est loin d'être "fonctionnelle" : plusieurs sous-programmes regroupent dans leurs résultats

- des données et
- des "drapeaux" déterminant le flot de contrôle au retour dans l'appelant.

Selon le chemin d'exécution suivi, `QIOUVR` modifie des ensembles complètement différents de paramètres de sortie.

La structure de données sous-jacente n'est que partiellement révélée par cet ensemble de sous-programmes. Elle est constituée de nombreux tableaux dont les valeurs de l'un sont employées comme indices d'un autre dans des indirections d'un "profondeur" de 4. Cette structure de données est parcourue par `QIOUVR` à l'aide de variables "drapeau".

Finalement, les sous-programmes étudiés n'ont aucun bloc `COMMON` mais `QIOUVR` (avec 300 lignes de code) a 38 paramètres, dont 22 sont lus uniquement, ainsi que 25 variables locales!

2.3 *ESTIME*

Nous avons analysé le code *ESTIME*, de l'application *SYSDIC*, écrit en 1977. *ESTIME* est le dernier maillon de la chaîne d'acquisition des données pour le contrôle en temps réel[1] du réseau de très haute tension. Une fois les mesures grossièrement erronées éliminées par les autres programmes de la chaîne, *ESTIME* effectue une analyse plus fine, basée sur des méthodes statistiques, afin de détecter et de corriger des mesures fausses des transitions à travers chaque ligne du réseau.

ESTIME est composé de 63 sous-programmes et de 7460 lignes de code. Nous avons limité les analyses statiques aux 26 sous-programmes appartenant à l'arbre des appels de `BESTI1` présenté dans la Figure 2.2 et regroupant 3408 lignes de code.

1. Toutefois, *ESTIME* ne contient pas d'interruptions ou autres spécificités temps réel.

```
BESTI1
    NLIRE
    MOVEK
    PREFIX
        NLIRE
        PRXPLO
            PRCHEN
        NECRIR
    CAMAT
        NLIRE
        CATMFA
            NLIRE
            F4LIR
                NLIRE
        CAISAX
        CACONT
            CASDEG
        MOVEK
        CADSTM
            CAISO
            CAEDI
        CAJAC
            CAPIV
            CATMT
        CAGAIN
            CRETAB
            FLIZER
            ORDO
            FACTO
            MOVEK
            GARE
    ECDISC
        MOVEK
        NECRIR
    NECRIR
```

FIG. 2.2: Graphe des appels d'*ESTIME*

2.4 Notre exemple principal

Notre exemple principal est composé

- d'un fragment du sous-programme `PRXPLO` et du sous-programme `PRCHEN` appelé uniquement par ce fragment, qui servent à construire et à écrire une certaine structure de données (et qui sont présentés dans les Annexes F et G),
- ainsi que
  - un fragment de `CACONT` et le sous-programme `CASDEG` appelé par ce fragment et
  - un fragment de `CAJAC` et le sous-programme `CAPIV` (présenté dans l'Annexe H) appelé par ce fragment

  qui lisent les données.

L'exemple de `PRXPLO` et de `PRCHEN` a été choisi pour nos rapports précédents [WP96, WP97, eNWP97], parce qu'il combine, dans quelques lignes de code peu commentées, une structure de contrôle et une structure de données indéchiffrables à première vue mais qui réalisent une spécialisation d'une fonction très classique.

L'analyse de la structure de contrôle de `PRCHEN` est décrite dans [WP96, WP97, eNWP97]. Nous avons repris cet exemple et l'avons étendu aux sous-programmes lecteurs de la structure de données pour en étudier une abstraction plus poussée que celle réalisée dans les rapports précédents.

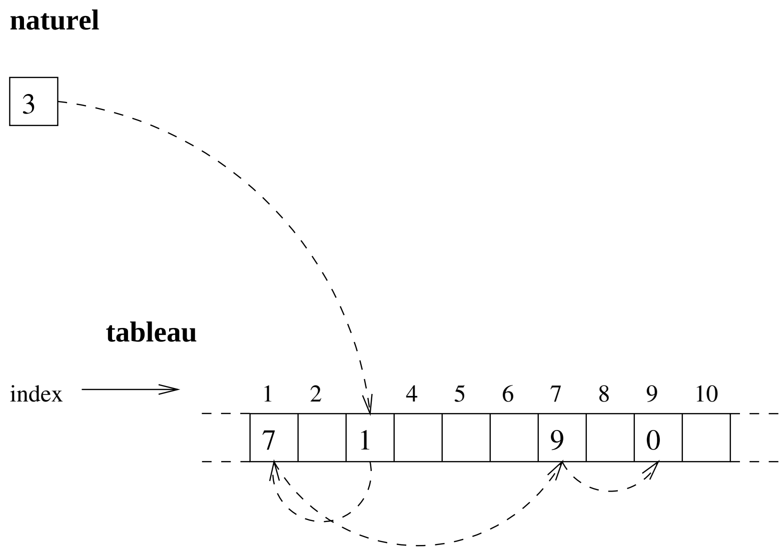


FIG. 2.3: L'implémentation de la liste [3,1,7,9]

2.4.1 Les tables partitionneuses

Rappelons la structure de données concernée. Les tableaux `IPCO` et `ICCO` paramètres formels de `PRCHEN` réalisent une table, indexée par des numéros positifs, de listes de numéros positifs. La table entière peut être stockée dans deux tableaux de façon astucieuse, grâce à la propriété suivante :

> chaque élément a au plus une seule occurrence dans une seule liste de la table.

Nous appelons une table avec cette propriété une *table partitionneuse*.

Chaque liste sans doublons de numéros positifs de la table, est représentée par

- un nombre naturel et
- un tableau stockant des naturels et indexé par les positifs.

Elle est réalisée sous forme de liste chaînée de la façon suivante (illustrée dans le schéma de la Figure 2.3) : si le nombre naturel est zéro, alors la liste est vide car zéro n'est pas un élément permis de la liste et indique en effet la fin de la liste. Sinon, le nombre naturel a une valeur positive qui est

- la valeur du premier élément de la liste
- et aussi l'indice du reste de la liste dans le tableau.

La valeur dans le tableau indiquée par cet indice est :

- soit 0, indiquant la fin de la liste
- soit une valeur strictement positive
  - valeur de l'élément suivant dans la liste
  - et aussi l'indice du reste de la liste.

Et ainsi de suite.

L'économie en espace mémoire réalisée par la double interprétation des valeurs de sorte *Elem*

- comme élément de la liste
- et aussi comme pointeur vers l'élément suivant

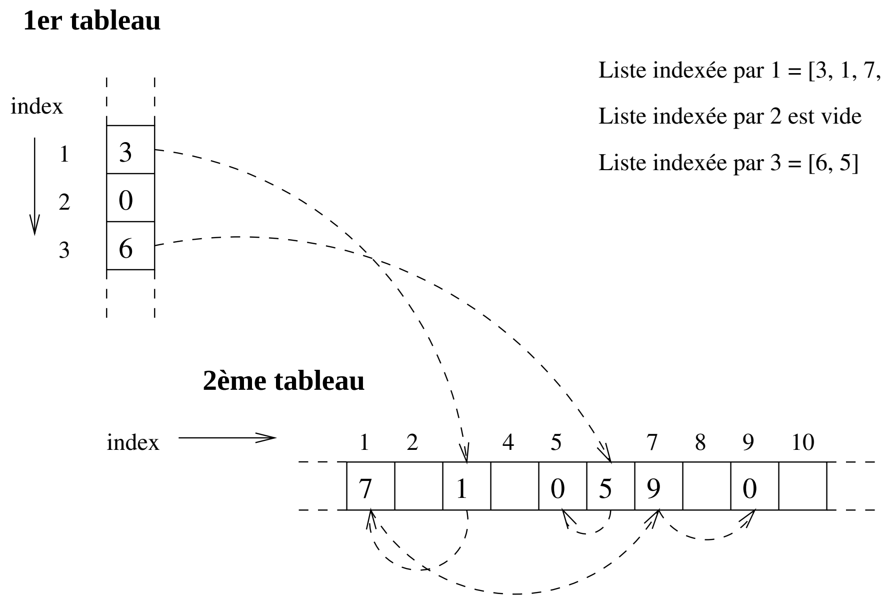


FIG. 2.4: L'implémentation d'une table de listes dans deux tableaux

n'est possible que pour des listes sans doublons.

La table partitionneuse entière est représentée par deux tableaux (les paramètres `IPCO` et `ICCO` de `PRCHEN`) stockant des naturels et indexé par les positifs.

La nature partitionneuse de l'ensemble des listes permet alors de stocker dans le même tableau (`ICCO`) les queues de toutes les listes non vides de la table, et de réaliser ainsi encore une économie d'espace mémoire. Les naturels représentant le début de chaque liste (valeur du premier élément ou zéro en cas de liste vide) sont stockés dans le premier tableau de la paire et indexés par les indices de la table. Cette implémentation est illustrée dans le schéma de la Figure 2.4.

2.4.2 Le calcul du co-cycle

Le fragment de `PRXPLO` initialise et puis remplit (à l'aide d'appels de `PRCHEN`) deux tables partitionneuses. L'étude des valeurs stockées dans les deux tables, ainsi que de leur lecture par `CASDEG` et `CAPIV`, nous a appris que les deux tables sont indexées par les numéros de sommets du réseau et stockent pour chaque sommet la liste des numéros des lignes connectées à ce sommet à l'origine (pour une table) resp. extrémité (pour l'autre). Ce fragment de `PRXPLO` réalise donc, avec `PRCHEN`, le calcul du *co-cycle* (l'ensemble des lignes connectées au sommet) pour chaque sommet du réseau électrique. `CASDEG` et `CAPIV`, qui effectuent des calculs dépendant du co-cycle, relisent les tables, sans les modifier.

2.4.3 L'ordre des éléments des listes

`PRCHEN` réalise l'insertion de chaque numéro dans la table partitionneuse, dans la bonne liste et de façon à respecter un ordre sur les éléments de chaque liste. Il implémente donc une spécialisation de l'insertion d'un élément dans une liste ordonnée, étendue aux tables de listes. L'ordre des éléments est choisi afin de permettre à `CASDEG` d'implémenter un algorithme efficace de comptage des *liaisons* de chaque sommet, comme nous l'expliquons dans le Chapitre 10 (Section 10.4).

Première partie

Élaboration de la spécification concrète

Chapitre 3

Problématique de l'élaboration de la spécification concrète

Résumé

Dans ce chapitre, nous passons en revue le problème général de l'élaboration d'une spécification concrète d'un code Fortran77. Ceci afin de fournir les informations de base pour les chapitres suivants de cette partie, et d'expliquer leurs orientations.

Nous commençons, dans la Section 3.1, par une description des spécificités du Fortran77 qui rendent son analyse difficile.

La Section 3.2 permet d'expliciter une hypothèse de base de notre analyse.

Notre étude concernant plus particulièrement les codes Fortran77 employés par le service Etudes de Réseaux de la Division Recherche et Développement d'EDF, nous décrivons quelques spécificités de ces codes, dans la Section 3.1.

La Section 3.4 est un rappel de ce qu'il faut "extraire" du code Fortran pour en construire une spécification algébriqueet dans la Section 3.5 nous discutons la possibilité de la construction automatique de la spécification concrète, ou du moins des axiomes décrivant les résultats des prédicats et opérations.

Dans la Section 3.6, nous rappelons le rôle de la spécification concrète dans notre approche et les implications quant à son niveau d'abstraction.

La Section 3.7 contient une brève description de l'"abstraction" du code que nous proposons d'effectuer **avant** l'élaboration de la spécification concrète, non pas afin de spécifier cette abstraction dans la spécification concrète, mais pour profiter, dans l'élaboration de la spécification concrète, de la compréhension du code qu'elle nous apporte.

Dans la Section 3.8, nous expliquons pourquoi il est difficile d'effectuer certaines analyses sur un fragment de code à la fois.

Finalement, la Section 3.9 contient l'énumération des étapes successives d'analyse, de transformation et de décomposition du code que nous proposons d'effectuer avant l'élaboration de la spécification concrète.

3.1 Spécificités du Fortran77

Nous sommes convaincus que la tâche d'élaboration d'une spécification concrète dépend fortement du langage de programmation employé. Obtenir une spécification d'un code écrit en *C* ou en *ML* ne pose pas les mêmes problèmes que si le code est écrit en Fortran. La première tâche du rétro-analyste est l'analyse du code, à la main afin de le comprendre, mais aussi à l'aide d'outils automatiques. Ces analyses automatiques et manuelles sont compliquées par les spécificités suivantes du langage Fortran77 [1]. Par contre, nous verrons que l'analyse automatique de Fortran77 est facilitée par l'interdiction de coder des sous-programmes récursifs.

1. Certaines caractéristiques varient de compilateur en compilateur : nous parlons alors des caractéristiques imposées par les compilateurs employés par EDF.

3.1.1 Identificateurs succincts

Les identificateurs sont limités à six caractères, il est souvent difficile pour le rétro-analyste de donner un "sens" à un nom de variable ou de sous-programme et d'avoir une idée de ce qu'il représente dans le domaine des réseaux électriques. De même, les étiquettes (employées dans les `GOTO`) sont limitées aux chiffres.

3.1.2 Typage primitif

Le système de typage de Fortran77 ne nous aide pas beaucoup dans notre tâche. Il y a très peu de types prédéfinis [2] et le programmeur n'a pas la possibilité de définir d'autres types.

La déclaration explicite des types n'est pas nécessaire. En effet, si le type d'une variable n'est pas déclaré, alors il dépend de la première lettre de l'identificateur [3].

Par conséquent, les variables locales (variables référencées dans le code mais pas déclarées comme paramètres, ni dans un bloc `COMMON`) ne sont explicitement déclarées nulle part.

Par ailleurs, un programme qui contient des incohérences dans les types (ou en d'autres termes : contient des coercitions implicites) ou dans les différentes déclarations des dimensions des mêmes tableaux n'est pas rejeté par le compilateur.

> Par exemple, voici la déclaration, extraite d'*ESTIME*, du sous-programme `MOVEK` et du type (tableau de caractères d'une dimension, l'astérisque indiquant que la longueur de cette dimension est inconnue) de ses deux premiers paramètres :

```
SUBROUTINE MOVEK(CHAIN1,CHAIN2,IND1,IND2,LON)
CHARACTER*1 CHAIN1(*),CHAIN2(*)
```

> et voici, extraits du sous-programme `CAGAIN`, les déclarations du tableau `M_GAIN` (dont le type implicite des éléments est entier) et du tableau `LCHBG` (d'une dimension et de trois éléments, tous implicitement de type entier, ce tableau appartenant au bloc `COMMON` nommé `CHBG`), ainsi qu'un appel du sous-programme `MOVEK` :

```
DIMENSION LHAC(*), NCHA(*), EHA(*), IVAC(*), M_GAIN(*)
COMMON/CHBG/LCHBG(3)
CALL MOVEK(M_GAIN(I_LCHBG),LCHBG,1,1,LON)
```

> Le premier paramètre formel de `MOVEK` déclaré comme étant de type tableau de caractères est instancié dans cet appel par un élément, de type entier, d'un autre tableau et le deuxième paramètre formel, du même type déclaré que le premier, est instancié par un tableau d'entiers.

3.1.3 Utilisation très fréquente des tableaux

Les tableaux sont la seule structure des données fournie par Fortran77, hormis les chaînes de caractères. Par conséquent, ils sont employés très souvent, et notamment pour simuler d'autres structures de données telles que listes, `records`, `structures` ...

3.1.4 Codes non structurés

Fortran77 contient la boucle `DO` et le `IF/THEN/ELSE`, mais pas d'instruction `while`, `repeat/until` ou `break`. Il y a le `GOTO` qui permet de simuler ces dernières, mais aussi d'inventer des structures de contrôle très compliquées !

Par ailleurs Fortran77 possède des instructions de contrôle spécifiques qui (tout comme le `GOTO`), servent à économiser quelques instructions ou une variable locale (`IF` *arithmétique*, `GOTO` *calculé* et `GOTO` avec étiquette variable).

---

2. Les types du langage Fortran77 sont `INTEGER`, `REAL`, `DOUBLE PRECISION`, `COMPLEX`, `LOGICAL`, chaîne de caractères d'une certaine longueur, ainsi que les types des tableaux : un pour chaque combinaison possible du nombre de dimensions et du type des valeurs (scalaires) stockées.

3. Par défaut, toute variable dont l'identificateur commence par une lettre comprise entre `I` et `N` est de type `INTEGER` et les autres de type `REAL`.

3.1.5 Différents noms ou configurations pour la même variable

Le programmeur peut donner plusieurs identificateurs aux mêmes variables et, dans le cas des tableaux, plusieurs indexations alternatives. En effet, il peut référencer la même zone de mémoire globale de façon différente dans différents sous-programmes en déclarant de façon différente le même bloc `COMMON`.

Par exemple, voici le bloc `COMMON` nommé `CHBG` déclaré tantôt comme trois variables scalaires entières, tantôt comme un tableau contenant trois éléments de type entier :

```
COMMON/CHBG/NS,NR,LF
COMMON/CHBG/LCHBG(3)
```

De même, une instruction `EQUIVALENCE` entre une variable locale et une variable déclarée dans un bloc `COMMON`, permet de référencer la même zone mémoire de différentes façons au sein du même sous-programme.

Par exemple, dans les déclarations suivantes (extraites du sous-programme `FARPRE` d'*ESTIME*), `*4` et `*2` indique le nombre d'octets employés pour stocker les variables scalaires. `ITAB` est donc déclaré comme un tableau d'une dimension contenant 50 éléments de type entier, chacun de longueur 4. `HBAN` est déclaré comme un tableau d'une dimension dont les éléments sont de type entier aussi mais de longueur 2. L'effet de l'instruction `EQUIVALENCE` est donc de nommer "`ITAB`" la plage du tableau `HBAN` comprise entre le deuxième et le 101ème élément, en prenant ces éléments deux par deux :

```
INTEGER*4 ITAB(50)
INTEGER*2 HBAN
DIMENSION HBAN(131 + 20*NMS + 15*NML + 10*NMG + 4*NMC)
EQUIVALENCE( ITAB(1), HBAN(2) )
```

3.1.6 Appel par référence

Lors d'un appel de sous-programme, c'est l'adresse de chaque paramètre effectif qui est passé au sous-programme, et non pas sa valeur. C'est l'*appel par référence*. Il permet en particulier l'instanciation d'un paramètre formel (muet) déclaré comme tableau, par un paramètre effectif qui n'est pas un tableau mais un élément de tableau (pas forcément le premier élément), voire une variable scalaire de bloc `COMMON`. L'effet est de passer au sous-programme un **fragment** du tableau resp. bloc `COMMON`.

3.1.7 Paramètres des sous-programmes non indiqués

Fortran77 oblige à déclarer les paramètres des sous-programmes et leur type mais n'impose pas, dans le cas des `SUBROUTINE`, de préciser lesquels sont des paramètres d'entrée et lesquels les paramètres de sortie. Par ailleurs une `FUNCTION` peut modifier la valeur des variables déclarées comme paramètres d'entrée. À cause de l'appel par référence, de telles modifications sont effectives dans l'appelant.

Une variable de bloc `COMMON` ne peut être directement référencée dans un sous-programme que si tout le bloc est déclaré, mais Fortran77 n'impose pas de préciser les variables du bloc **effectivement référencées en lecture resp. écriture**.

Une variable d'un bloc `COMMON` non déclaré dans le sous-programme peut être référencée dans un autre sous-programme appelé (directement ou indirectement) par celui-ci. Par conséquent, pour connaître les blocs `COMMON` dont les variables peuvent être directement ou indirectement référencées par un sous-programme donné, il faut étudier les déclarations de `COMMON` non seulement de ce sous-programme mais de tous les autres sous-programmes appelés, directement ou indirectement, par lui.

Un paramètre formel de sous-programme peut être déclaré comme tableau mais en mettant un astérisque à la place des limites de chaque dimension. Les longueurs des dimensions du paramètre formel sont alors inconnues. Dans le cas où ce paramètre est instancié par un élément de tableau ou par une variable scalaire de bloc `COMMON`, l'appel par référence permet au sous-programme de référencer des éléments (resp. d'autres variables du même bloc) au delà de celui passé. Nous ne savons pas alors les limites de la plage du tableau (resp. bloc `COMMON`) effectivement référencée. Appelons ce cas de figure *l'exploitation de l'appel par référence.*

Un exemple de l'instanciation d'un paramètre formel déclaré comme un tableau par un élément de tableau est donné dans la Section 3.1.2 ci-dessus.

Voici un exemple d'instanciation par une variable de bloc `COMMON`. Le sous-programme `NLIRE` copie le contenu du fichier indiqué par son deuxième paramètre dans le tableau correspondant à son premier paramètre. Le nombre de caractères copiés étant donné par son troisième paramètre. Son premier paramètre est déclaré comme un tableau, de longueur inconnue, de caractères :

```
SUBROUTINE NLIRE (CTAB, CNOMF, LON, KER)
CHARACTER*1 CTAB(*),...
```

Le bloc `COMMON` nommé `PAREG` est déclaré dans le sous-programme `PREFIX` comme étant composé de plusieurs variables dont la première est (implicitement) une variable scalaire de type réel nommée `ZMAXI`:

```
COMMON/PAREG/ZMAXI,ZMAXF,PRECA,PRECR,SEUA,\ldots
```

`PREFIX` contient l'appel suivant de `NLIRE`:

```
CALL NLIRE (ZMAXI, 'ASESWZ', 50, KER)
```

qui a pour effet de copier les 50 premiers caractères du fichier `ASESWZ` dans une zone mémoire débutant à l'adresse de `ZMAXI`, c'est-à-dire non seulement dans la variable `ZMAXI` passée comme paramètre, mais dans toutes les autres variables du bloc `COMMON`!

En conclusion, il ne suffit pas d'étudier les déclarations de paramètres et de blocs `COMMON` d'un sous-programme pour connaître toutes les variables non locales dont il peut lire la valeur en entrée ou fixer la valeur en sortie.

3.1.8 Deuxième ordre

Le passage en paramètre d'un autre nom de sous-programme, de point d'entrée de sous-programme ou de fonction incorporée, permet en principe de coder des fonctions de deuxième ordre.

3.1.9 Absence de modules

Il n'y a pas de modules dans Fortran77 : un programme est composé d'un programme principal et de sous-programmes [4].

3.2 Hypothèse de départ

Les codes que nous analysons sont utilisés par EDF depuis au moins dix ans. Nous supposons alors qu'ils ne contiennent pas de bogues, ni de boucles infinies. En fait, de petites fautes de programmation peuvent subsister, comme illustré par l'exemple suivant.

`PRCHEN` commence par une instruction de lecture, avec un indice qui peut valoir 0, d'un tableau indexé en fait par des positifs. L'instruction suivante provoque la sortie du sous-programme si l'indice vaut 0, alors le fait que la valeur lue dans le tableau (et copiée dans une variable locale au sous-programme) n'est pas définie dans ce cas n'a aucune influence sur le résultat du sous-programme.

3.3 Spécificités des codes analysés

Notre approche est sans doute influencée par les spécificités suivantes des codes analysés.

3.3.1 Économie de mémoire

À l'époque où les codes ont été réalisés, l'économie de variables ou de lignes de code était plus prioritaire que les principes tels que respect des types, encapsulation, programmation structurée, ... Par conséquent, les programmeurs ont souvent profité du fait que les compilateurs Fortran77 n'imposent pas le respect des types déclarés. Ils ont exploité aussi l'appel par référence de la façon décrite dans la Section 3.1.7 ci-dessus.

4. (Les sous-programmes sont déclarés à l'aide des instructions `BLOCK DATA`, `FUNCTION` et `SUBROUTINE`.)

3.3.2 Simulation de l'allocation dynamique

Certains blocs `COMMON` sont gérés de façon à simuler, à l'aide de *l'exploitation de l'appel par référence* (voir Section 3.1.7), l'allocation dynamique de mémoire. Typiquement, un grand tableau (d'une dimension) est déclaré dans un bloc `COMMON` et puis les valeurs d'un certain nombre d'indices sont calculées dynamiquement. Ce sont les indices des premiers éléments de différentes plages du tableau. Les différentes plages peuvent être utilisées pour stocker des informations de natures différentes. Dans les appels de certains sous-programmes, dans lesquels le bloc `COMMON` n'est pas déclaré, au lieu de passer comme paramètre tout le tableau, le premier élément de la plage référencée par le sous-programme instancie un paramètre formel déclaré comme tableau et dimensionné par un astérisque. Un autre paramètre ou variable de bloc `COMMON` indique la longueur de la plage. Seule cette plage du grand tableau est effectivement référencée par le sous-programme, comme si elle était un petit tableau. C'est parce que ce petit tableau "virtuel" a une dimension calculée dynamiquement qu'il n'est pas déclaré en tant que tel.

3.3.3 Héritage de FortranIV

Certains des codes étudiés ont d'abord été réalisés en FortranIV avant d'être portés en Fortran77. Or FortranIV n'avait pas la structure de contrôle `IF/THEN/ELSE` et ces codes contiennent donc de nombreuses instances d'utilisation d'`IF` *logique* ou `IF` *arithmétique* avec des `GOTO`, qui "simulent" la structure du `IF/THEN/ELSE` (et qui n'ont pas été remplacées dans le portage). L'emploi des `IF/THEN/ELSE` permet d'éliminer des `GOTO` et donne des codes plus faciles à analyser.

3.3.4 Absence de certaines constructions

Nous ne parlerons pas de l'analyse des instructions suivantes, qui sont absentes des codes analysés :

- `EXTERNAL` ou `INTRINSIC` (signalant la réalisation de fonctions de deuxième ordre) ;
- `ASSIGN` (signalant le `GOTO` avec étiquette variable et sans liste de valeurs possibles) ;
- `GOTO` avec étiquette variable suivie par la liste de ses valeurs possibles ;
- `ENTRY` (point d'entrée nommé et arguments formels alternatifs pour un sous-programme) ;
- `SAVE` (sauvegarde des valeurs de certaines variables locales entre exécutions successives de sous-programme) ;
- `RETURN` paramétré (par la position dans la liste des arguments d'une étiquette du sous-programme appelant)
- et `BLOCK DATA` (pour initialiser un bloc `COMMON`).

3.3.5 Trois catégories de structure de données

Comme nous l'avons expliqué dans la Section 3.1.3, les tableaux sont souvent employés dans les programmes Fortran77 pour réaliser d'autres structures de données. Mis à part l'emploi de grands tableaux de zone `COMMMON` pour simuler l'allocation dynamique de mémoire, décrit dans la Section 3.3.2, nous avons trouvé dans les codes analysés trois façons principales d'employer les tableaux pour simuler d'autres structures de données.

Classe d'objets

Plusieurs tableaux, $T1$, $T2$, ... , d'une dimension et de la même longueur n (mais pas nécessairement du même type), sont employés pour stocker les valeurs des "attributs" de n "instances" de la même "classe" d'"objets". Il y a un tableau par attribut. Chaque tableau stocke la valeur de l'attribut pour chacun des n objets. Les valeurs des attributs sont représentées donc par des nombres ou des chaînes de caractères. Tous les tableaux sont référencés de la même façon par les objets d'une même classe. Les objets sont numérotés et l'état d'un objet est composé des valeurs indexées par son numéro dans tous les tableaux. L'état de l'objet de numéro i est donc $T1[i], T2[i], \ldots$, où $T1[i]$ est la valeur de l'attribut stocké dans $T1$, $T2[i]$ est la valeur de l'attribut stocké dans $T2$, ... Par exemple, les tableaux `ZIMP`, `ATG` et `SUSL` paramètres formels du sous-programme `CAPIV` (présenté dans l'Annexe H) sont indexés par les numéros de ligne électrique du réseau et servent à stocker les valeurs des propriétés électriques d'*impédance*, d'*angle de perte* et de *susceptance*, respectivement, de chaque ligne. En cas d'attributs dont la valeur n'est pas un seul nombre ou chaîne de caractères mais où **chaque valeur** nécessite une structure de données pour son stockage, un tableau adressé de cette façon peut être le "point d'entrée" de l'une des structures de données plus compliquées décrites ci-dessous.

Table partitionneuse de listes de positifs

Cette structure peut être vue comme une extension de la précédente, car les valeurs d'un attribut sont indexées de la même façon par le premier tableau $T0$. Cette fois, les valeurs sont sous forme de liste sans doublons et les queues des listes pour tous les objets sont stockées dans un deuxième tableau T dans la façon décrite dans le Chapitre 2 (Section 2.4.1). Les queues des listes de tous les objets peuvent être stockées dans le même tableau grâce à la nature *partitionneuse* de la table. Par exemple, les tableaux `IPCO` et `ICCO` stockent, pour chaque numéro de sommet, la liste des numéros de ligne sortant du sommet.

Table de listes d'entiers

Dans cette structure, les éléments de chaque liste sont stockés dans une séquence d'éléments contigus du deuxième tableau T et (contrairement à l'exemple précédent) les listes pour tous les objets sont stockées l'une après l'autre dans T. Le tableau $T0$ est indexé de la même façon que dans les deux structures précédentes en ce qui concerne le début de chaque liste. Les listes sont de longueur variable, alors pour un objet de numéro i, le premier élément de la liste est stocké dans $T[T0[i]]$ et la fin dans $T[T0[i+1]]-1$. Contrairement au cas de la classe d'objets, pour stocker les listes de n objets, il faut écrire les $n+1$ premiers éléments de $T0$. Un exemple de cette structure étendue à trois tableaux est donné par les tableaux `LHAC`, `NCHA` et `EHA`. Le premier tableau stocke les indices de début et de fin de deux listes de la même longueur pour chaque objet : la première stockée dans `NCHA` et la deuxième dans `EHA`. Dans ce cas les objets sont des lignes d'une matrice creuse et c'est le numéro de colonne de chaque élément non nul de la ligne qui est stocké dans `NCHA`, et la valeur de l'élément dans `EHA`.

3.4 Le passage du Fortran vers les spécifications algébriques

Nous pouvons voir l'élaboration d'une spécification algébrique d'un code Fortran comme la recherche dans le code Fortran des *opérations*, *prédicats*, *sortes*, *axiomes* et finalement des types abstraits de données de la spécification algébrique. Nous allons discuter de façon plus détaillée les différents aspects de cette recherche dans la suite de cette partie, mais ici nous ébauchons les principes qui constituent son point de départ.

Les opérations

En principe, chaque *opération* de la spécification algébrique spécifie soit une opération primitive prédéfinie en Fortran soit un sous-programme. En fait il faudra ajouter d'autres opérations à la spécification algébrique. Ceci pour deux raisons principales :

1. Le résultat d'une opération est défini par un axiome qui spécifie les instructions du sous-programme, mais souvent la formulation de celui-ci nécessite la définition d'opérations auxiliaires, à cause de la structure de contrôle du sous-programme. Par exemple, le `DO` et le `GOTO` (qui sert à réemployer un fragment de code) permettent d'écrire des fragments de code à l'intérieur d'un sous-programme, dont chacun ne peut être spécifié que par une opération récursive spécifique.

2. La structuration d'un code Fortran77 en sous-programmes ne correspond pas en général aux opérations (ou prédicats) d'une spécification algébrique. Par exemple, les sous-programmes peuvent regrouper des traitements indépendants de données différentes. De plus, les opérations de base d'un type abstrait de données sont parfois implémentées par une ou deux instructions Fortran, alors qu'un programmeur "économiserait" la définition d'un sous-programme dans ce cas.

La détermination des opérations et prédicats de la spécification algébrique est discutée de façon plus détaillée dans le Chapitre 6 (Section 6.3).

Les prédicats

Les prédicats de la spécification algébrique servent en principe à spécifier soit des tests de base prédéfinis tels que les inégalités sur les nombres, soit des rares sous-programmes (ou fragments de code) avec un seul paramètre de sortie, de type `LOGICAL`. Nous y ajouterons des prédicats auxiliaires spécifiant des fragments de code.

Les paramètres des prédicats et opérations

Les paramètres des prédicats et opérations représentent les *vrais paramètres* des sous-programmes Fortran77, qui ne sont que partiellement indiqués par les déclarations (voir Section 3.1). Intuitivement, les vrais paramètres d'entrée d'un sous-programme sont toutes les variables non-locales dont la valeur **lors de l'appel** est utilisée dans le sous-programme et les vrais paramètres de sortie sont toutes les variables non-locales modifiées par le sous-programme et dont la nouvelle valeur est utilisée après le retour de l'appel. Ceci afin d'exclure des vrais paramètres de sortie d'éventuelles variables non-locales employées par le sous-programme pour stocker des valeurs intermédiaires du calcul. Par ailleurs, à des paramètres passés depuis ou vers un autre sous-programme appelant, il faut ajouter les entrées/sorties fichier, ainsi que l'interface avec l'utilisateur à travers les entrées/sorties écran, clavier, imprimante ...

Les tuples

Une `SUBROUTINE` et même une `FUNCTION` peuvent avoir plusieurs *vrais paramètres* de sortie, mais une opération n'en a qu'un seul : alors pour spécifier les sous-programmes avec de nombreux paramètres de sortie, il faut définir le *tuple* des paramètres de sortie, avec sa sorte.

Les opérations de construction du tuple et les projections permettant de récupérer ses composants ne seront pas explicitement implémentées dans le code Fortran dans ce cas. Nous pouvons donc considérer certains paramètres d'entrée d'un sous-programme Fortran aussi comme étant regroupés en tuples implicites.

En effet, nous avons remarqué dans la Section 3.1.3 que Fortran77 n'offre que les tableaux comme façon de structurer les données numériques. D'autres structures de données sont souvent implémentées par des tuples de variables (scalaires ou tableaux) qui sont passées ensemble aux sous-programmes.

Les variables

Les variables d'une spécification algébrique sont employées dans les axiomes comme arguments des prédicats et opérations et certaines représentent donc les *vrais paramètres* formels d'entrée des sous-programmes avec leur valeur à l'entrée du sous-programme. Les éventuelles valeurs ultérieures des paramètres formels d'entrée, les valeurs des vrais paramètres formels de sortie, ainsi que les valeurs des variables locales sont représentées dans la spécification par des termes construits à partir d'opérations (y compris constantes), de prédicats et des variables.

Les constantes

Les constantes de la spécification algébriquespécifient les constantes prédéfinies de Fortran77 (nombres et chaînes de caractères), ainsi que les "constantes nommées" (i.e. identificateurs pour des valeurs constantes) dont la valeur est fixée lors de la compilation du code, à travers les instructions `PARAMETER`, `DATA` ou `BLOCK DATA`.

En ce qui concerne les valeurs constantes de tableaux, nous pourrions supposer qu'un sous-programme ou fragment de code servant à initialiser à zéro un tableau dont l'adresse est passé en paramètre serait spécifié aussi par une constante. C'est le cas, à condition :

- que le tableau soit un (vrai) paramètre de **sortie** du sous-programme, qui ne **lit** aucune valeur,

– que le nombre de cases du tableau écrites soit donc constant (les dix premières cases, par exemple) et

– que toute lecture de ce tableau suite à son initialisation et avant toute modification éventuelle de sa valeur, soit limitée aux éléments initialisés.

En effet une variable Fortran77 à laquelle aucune valeur n'a encore été affectée a soit la valeur par défaut que lui donnent certains compilateurs, soit une valeur qui dépend de l'état de la mémoire au moment de l'exécution du code. La lecture d'un élément de tableau Fortran77 est donc spécifiée par une opération partielle qui n'est pas définie pour les éléments non encore explicitement initialisés. Pour chaque sorte de tableau Fortran, nous spécifions une **constante** représentant le tableau illisible partout et qui sert d'argument à la première opération d'écriture effectuée sur un tableau vierge. Cette constante correspond alors à la valeur, au moment de son écriture, d'un tableau déclaré (peut-être dans un autre sous-programme qui a ensuite passé son adresse à celui qui l'initialise), mais pas encore explicitement initialisé.

Les sortes

Chaque sorte de la spécification algébrique spécifie un ensemble de valeurs de constantes ou de variables Fortran.

Nous expliquons ci-dessous pourquoi nous voulons, même au niveau de la spécification concrète, définir les sortes plus fines que les types Fortran77. C'est-à-dire que chaque sorte de la spécification algébrique regroupe un ensemble de valeurs plus restreint qu'un type Fortran77.

Une variable Fortran n'est qu'un identificateur d'adresse mémoire où de nombreuses valeurs peuvent être successivement stockées. Parfois une variable qui contient une valeur représentant une chose et qui n'est plus utile au calcul, est réutilisée pour stocker une autre valeur avec une signification complètement différente. Dans ce cas, les valeurs successives de la même variable Fortran peuvent ne pas avoir la même **sorte** dans la spécification algébrique.

La détermination des sortes de la spécification algébrique est discutée de façon plus détaillée dans les Chapitres 7 et 8 (Section 8.1) ci-dessous.

L'engendrement fini

Supposons que nous ayons identifié tous les fragments de code où les variables (ou tuples de variables) d'une certaine sorte sont écrites. Si tous ces fragments de code sont spécifiés par des opérations avec un résultat de cette sorte, alors nous pouvons définir *l'engendrement fini* de cette sorte par ces opérations. Ceci a pour effet de restreindre la sorte aux valeurs pouvant être écrites par ces opérations. L'intérêt est démontré dans le Chapitre 9 (Section 9.8).

Le domaine

Le *domaine* de définition des sous-programmes n'est pas explicité dans le code Fortran77. Il faut le déduire à partir

– des conditions de sortie en cas d'erreur, qui indiquent les valeurs des entrées qui ne sont pas acceptées par le sous-programme,

– des conditions de terminaison des boucles, car nous supposons qu'aucune exécution de boucle n'est infinie, ce qui impose des contraintes sur les entrées du sous-programme,

– ainsi que des domaines des opérations de base prédéfinies. Par exemple : le résultat d'un sous-programme effectuant juste la lecture d'un élément d'un tableau pris en entrée, n'est défini que si cet élément du tableau a été écrit. Le sous-programme n'accepte donc en entrée que les tableaux satisfaisant cette condition.

Toutefois, une contrainte sur les valeurs d'un seul (vrai) paramètre du sous-programme peut être spécifiée soit par la sorte du paramètre (dans le profil de l'opération), soit par le domaine de l'opération. Par contre, une contrainte sur les combinaisons de valeurs des différents paramètres d'entrée ne peut être spécifiée qu'à travers le domaine.

Les axiomes

La plupart des axiomes de la spécification algébrique définissent le résultat d'une opération (ou la condition de vérité d'un prédicat) spécifiant un sous-programme ou fragment de sous-programme. Ils spécifient donc les instructions du corps du fragment de code, ainsi que le flot de contrôle. Certaines des instructions sont spécifiées par des applications de prédicats ou d'autres opérations et la structure de contrôle les reliant par

- la composition des opérations (ou prédicats),
- l'implication ou la forme __ *when* __ *else* __ de *CASL* (permettant de spécifier les instructions conditionnelles)
- ou par la récursion (permettant de spécifier une boucle).

L'analyse des instructions Fortran afin d'obtenir les axiomes est discutée dans la Section 3.5 ci-dessous.

La structure

Une spécification algébrique est normalement structurée autour de plusieurs types abstraits de données, chacun avec les opérations (les *constructeurs*) qui servent à stocker les données de cette façon et les opérations ou prédicats (les *observateurs*) qui accèdent aux données stockées. Nous allons essayer d'identifier des types abstraits de données dans le code Fortran77 (comme expliqué dans la Section 8.3 du Chapitre 8 ci-dessous) et de structurer la spécification concrète de cette façon. Toutefois, les codes Fortran que nous analysons n'étaient pas conçus selon ce modèle et par conséquent les constructeurs et observateurs peuvent ne pas être encapsulés sous forme de sous-programme. D'autre part, comme les structures de données sont souvent implémentées sous forme de tuple de variables, ce n'est pas toujours toute la structure qui est le paramètre des observateurs (et même des constructeurs), mais juste les composants du tuple nécessaires à cette opération.

3.5 Analyse automatique

Serait-il possible d'extraire la spécification concrète du code Fortran de façon automatique ?

En principe, nous pourrions adopter une approche basée sur la *sémantique* du langage de programmation. Prenons, par exemple, la sémantique *axiomatique*. Elle associe à un fragment de code sa *précondition* et sa *postcondition* exprimées sous forme d'expression logique close. Celles-ci seraient d'une grande utilité dans la formulation des axiomes définissant les domaines et les résultats des opérations.

Si nous pouvions décomposer chaque fragment de code en instructions élémentaires dont nous connaissions la sémantique, alors la sémantique du fragment serait obtenue en composant les sémantiques des instructions élémentaires. Malheureusement, dans le cas du Fortran77 la sémantique d'une instruction peut dépendre de son contexte. En particulier, la décomposition d'un fragment de code en blocs indépendants est difficile à cause des sauts non structurés réalisés à l'aide des `GOTO`.

L'*exécution symbolique* s'appuie sur la sémantique des instructions qui ne modifient pas le flot de contrôle. Supposons donnée pour chacune de ces instructions, une expression des valeurs des variables suite à l'exécution de l'instruction en fonction des valeurs des variables avant l'exécution. L'exécution symbolique simule alors le flot de contrôle dans le fragment de code, afin de calculer une expression de la valeur des sorties du fragment en fonction de la valeur des entrées. Cette expression est souvent conditionnelle, associant à chaque combinaison des valeurs des variables en sortie, une précondition sur les valeurs des variables en entrée.

Nous décrivons l'exécution symbolique avec plus de détails dans le chapitre suivant. Elle a comme inconvénient de fournir des préconditions exprimées comme des combinaisons de contraintes difficiles à simplifier et dont la satisfiabilité est un problème indécidable. En effet, l'indécidabilité du problème de reconnaissance des chemins d'exécution impossibles est une limitation de toute technique d'*analyse statique*, dont l'exécution symbolique. Mais l'exécution symbolique est indécidable pour une autre raison : une expression close de la postcondition d'une boucle n'est obtenue qu'en trouvant une solution à l'équation récursive (ou système d'équations simultanées) décrivant la boucle. C'est l'*invariant* de la boucle et il ne peut pas toujours être trouvé automatiquement. En fait, nous n'avons pas besoin de la solution de l'équation récursive pour la spécification concrète : une définition récursive de l'opération qui spécifie la boucle nous suffit. Il semblerait théoriquement possible alors de générer automatiquement des axiomes par une technique similaire à celle de l'exécution symbolique. Cependant, ceci serait très difficile à mettre en œuvre et produirait une spécification plus illisible que le code source.

D'autres techniques de l'analyse statique (décrites dans le chapitre suivant) associent à chaque point d'un fragment de code des informations qui sont en règle générale plus partielles que dans le cas de l'exécution symbolique et visent ainsi à fournir de façon entièrement automatique des informations utiles même si partielles et approximatives.

Notre approche consistera donc à employer l'analyse statique (et implicitement, une sémantique partielle de Fortran) pour extraire de façon entièrement automatique certaines informations du code. Ensuite, nous nous appuyons sur ces informations pour terminer à la main la construction de la spécification concrète.

3.6 Niveau de la spécification concrète

Notre but est d'élaborer une spécification concrète qui soit

1. une spécification exacte du code Fortran,
2. suffisamment proche du code Fortran pour permettre de retrouver facilement dans le code les objets de la spécification et
3. plus lisible (mieux structurée, plus explicite,...) que le code Fortran.

Nous pouvons réconcilier ces objectifs en nous permettant de représenter parfois (pour atteindre le troisième objectif) non pas le vrai code mais le résultat de transformations mineures (mineures à cause du deuxième objectif) qui ne changent pas sa sémantique (pour satisfaire au premier objectif) : i.e. le code transformé donne les mêmes sorties pour les mêmes entrées que le code initial.

Afin de maintenir des liens avec le code source d'origine, nous proposons de spécifier chaque sous-programme du code par un prédicat ou une opération du même nom. Cependant, le prédicat ou l'opération spécifiant un sous-programme entier peut être défini à l'aide d'autres prédicats et opérations dont chacun ne correspond pas forcément à un bloc d'instructions du sous-programme, mais peut spécifier des instructions dispersées dans le code Fortran. Ainsi, nous pouvons construire une spécification concrète structurée autour de types abstraits de données, même si les sous-programmes du code Fortran initial ne correspondent pas exactement aux prédicats et opérations des types abstraits de données.

De même, nous proposons d'employer autant que possible les noms des paramètres formels d'un sous-programme comme noms de variables dans les axiomes définissant le sous-programme. La sorte d'un ensemble de variables Fortran devrait avoir un nom qui évoque une ou plusieurs variables Fortran de cette sorte.

Afin d'améliorer la lisibilité de la spécification, nous proposons de spécifier non pas le “vrai” code, mais celui obtenu après une restructuration qui simplifie le plus possible les structures de contrôle. Nous aurions pu commencer par une spécification algébrique de très bas niveau, et probablement moins lisible que le code source, résultat d'une “transcription” du code initial (par exécution symbolique, voir la Section 3.5) et puis valider formellement la restructuration. Nous avons choisi de ne pas porter notre effort au niveau de ce tout premier pas d'abstraction, car nous croyons que la restructuration du code est une abstraction relativement facile à justifier.

Pour la même raison, nous voulons spécifier, à l'aide des paramètres des prédicats et opérations, les données **effectivement passées** entre les sous-programmes, et non pas une représentation fidèle de l'appel par référence avec passage de **pointeurs** vers les données.

De la même façon, nous proposons de définir dans la spécification algébrique un ensemble de sortes qui sont plus fines que les types Fortran77. En effet, à cause de la pauvreté du système de typage de Fortran77, le programmeur est obligé de donner à un paramètre formel de sous-programmes un type tel qu'`INTEGER` même s'il sait que ce paramètre ne sera instancié que par certaines des variables de type `INTEGER`, par exemple : les variables représentant un numéro de ligne électrique mais pas celles représentant un numéro de sommet du réseau. Nous voudrions définir des sortes qui correspondent mieux au modèle conceptuel du programmeur que les types Fortran77.

Nous proposons donc une spécification concrète déjà “un peu abstraite”. Ainsi, nous proposons de ne pas spécifier

- certains aspects “non-fonctionnels” du code, par exemple : le choix de format des sorties, les messages de déboguage et statistiques diverses, ...
- le détail des opérations de bas niveau, telles que la lecture des fichiers.

Dans la spécification algébrique (partielle) de Fortran enrichie par la spécification concrète, nous proposons de

- faire abstraction des limites à la représentation des nombres et des dimensions des tableaux, en spécifiant des tableaux infinis [5],
- et de distinguer, parmi les entiers, les naturels et positifs.

3.7 Abstraction

Nous retrouvons les abstractions du code de trois façons :

1. à l'aide des restructurations du code effectuées par des outils automatiques
2. à l'aide des transformations manuelles ou semi-automatiques de l'organigramme du code, la recherche manuelle des invariants des boucles et l'élimination de certains paramètres de sous-programme (décrits pour l'exemple principal de cette thèse dans [WP96], [WP97] et [eNWP97]),
3. à travers la compréhension du code qui résulte des deux catégories précédentes d'abstractions, ainsi que du reste de la partie manuelle de l'étude du code nécessaire à l'élaboration de la spécification concrète.

Les restructurations automatiques afin de simplifier les structures de contrôle, rendent le code plus clair et sa spécification aussi. Elles sont souvent suffisamment mineures pour justifier l'élaboration de la spécification concrète à partir du code restructuré, comme nous l'avons expliqué dans la section précédente.

Les transformations de la deuxième catégorie ci-dessus sont plus poussées et traduisent donc un pas d'abstraction plus grand. En règle générale, le résultat de ces transformations n'est donc pas spécifié par la spécification concrète, mais elles aident à la compréhension du code.

C'est la vision abstraite du code apportée par la compréhension intuitive qui est spécifiée par la spécification la plus abstraite. Cependant, nous proposons d'effectuer avant l'élaboration de la spécification concrète toutes les restructurations manuelles nécessaires à la compréhension du code sur un plan plus abstrait. Car la spécification concrète est plus claire si elle tient compte de notre vision abstraite du ”sens” du code dans sa présentation, sa structure, les noms des sortes, prédicats et opérations auxiliaires, ...

3.8 Analyse d'un fragment d'un code entier

Les codes que nous analysons ne sont pas composés de routines de “bibliothèque” informatique mais de fragments de code [6] dont les programmeurs connaissaient d'avance les “utilisateurs”, car ils étaient limités à d'autres fragments du même code. Les programmeurs en ont souvent tenu compte : la structuration des codes en sous-programmes et l'absence de tests de la légitimité des valeurs effectives des paramètres des sous-programmes en témoignent.

> Par exemple : la sous-programme `CATMFA` contient un branchement suite au test du paramètre d'entrée `IBT`. La deuxième branche commence par la lecture de variables de bloc `COMMON` qui ne sont

5. Grâce à la spécification de la lecture des tableaux par une opération partielle, l'abstraction de leurs dimensions en mémoire n'empêche pas, dans les exemples que nous avons étudiés, la spécification précise des algorithmes.

6. Évidemment, ces fragments de code peuvent contenir quelques appels à des routines “bibliothèque”, mais elles ne sont pas spécifiques au domaine des réseaux électriques et sont relativement bien connues, alors ce n'est pas leur analyse qui nous intéresse le plus.

initialisées que par la première branche. En fait, le programmeur a compté sur le fait que `CATMFA` est toujours appelée deux fois de suite. La valeur d'`IBT` est modifiée par l'appelant de façon à assurer que lors du premier appel c'est la première branche qui est suivie et lors du deuxième appel, la deuxième branche !

Les programmeurs ont aussi apporté aux codes, des optimisations qui supposent un certain contexte d'appel. Nous ne pouvons donc pas ignorer cet aspect et analyser chaque sous-programme comme s'il était un module parfaitement encapsulé et indépendant. En fait, la prise en compte du contexte d'appel nous permet certaines inférences supplémentaires dans l'analyse du code.

Ainsi, nous allons inférer les *vrais paramètres* de chaque sous-programme par une analyse statique des variables référencés dans le sous-programme, mais aussi à l'extérieur du sous-programme. Par exemple, nous voulons exclure des vrais paramètres de sortie d'éventuelles variables non-locales utilisées par le sous-programmes pour stocker des valeurs intermédiaires du calcul. Ce sont les variables non-locales modifiées par le sous-programme et dont les nouvelles valeurs sont utilisées à l'intérieur du sous-programme mais pas à l'extérieur, au retour de l'appel. Par ailleurs, nous allons inférer les sortes des vrais paramètres des sous-programmes à partir d'une analyse des alias dynamiques entre paramètres formels et effectifs dans tous les sites d'appel de chaque sous-programme. Finalement, nous verrons dans la deuxième partie de cette thèse que nous pouvons avoir besoin de connaître les vrais paramètres du code entier pour justifier l'abstraction d'un fragment de celui-ci.

Cependant, à cause de la taille du code entier, sa spécification concrète et abstraite en une fois n'est pas envisageable. Nous proposons donc dans la suite de cette partie, de spécifier des fragments du code correspondant autant que possible aux types abstraits de données réalisés, mais la reconnaissance de ces fragments permettant la décomposition du code repose sur les résultats de certaines analyses ultérieures du code entier, notamment les *vrais paramètres* des sous-programmes et leurs sortes.

Or certains outils d'analyse automatique peuvent prendre en entrée un code entier, mais normalement les analyses qui ne sont pas automatisées sont réalisées à la main sur un fragment du code à la fois. Il s'agit alors d'une décomposition provisoire du code afin d'effectuer les analyses manuelles. Dans ce cas, nous risquons plusieurs itérations d'analyse, car il faudra ré-effectuer l'analyse de chaque fragment en fonction des résultats des analyses des autres fragments. Si la structure du graphe des appels le permet, nous pouvons limiter les itérations en décomposant provisoirement le code en fragments correspondant à des "sous-arbres" du graphe des appels et regroupant des ensembles **disjoints** de sous-programmes. L'interface d'un tel fragment avec le reste du code est limitée à celle du sous-programme à la racine du sous-arbre.

C'est une solution peu satisfaisante et c'est pourquoi il faut essayer d'automatiser autant que possible les analyses à effectuer sur le code entier, avant la décomposition du code pour l'analyse détaillée, et en partie manuelle, de chaque fragment.

3.9 Démarche d'élaboration de la spécification concrète

Nous avons décrit la problématique de l'élaboration de la spécification concrète d'un code Fortran77. La démarche que nous proposons peut être résumée par les étapes suivantes. Elles sont décrites avec plus de détails dans la suite de cette partie (sauf en ce qui concerne les trois premières étapes et l'étape d'abstraction).

1. **Rassembler** le code complet auquel appartient le fragment étudié, y compris les fonctions de bibliothèque appelées ou, sinon, une description suffisamment détaillée des parties manquantes pour permettre de définir l'interface avec le fragment analysé de la façon évoquée dans la Section 3.8. Cette première étape nécessite d'employer un outil d'analyse statique (parmi ceux décrits dans le chapitre suivant) qui construit le graphe des appels du code et indique d'éventuels sous-programmes manquants (ou jamais appelés). Il faut essayer d'employer pour cette étape, un outil qui accepte le code sans les transformations préliminaires évoquées dans l'étape numéro 3 ci-dessous.

2. **Analyse manuelle préliminaire** : il faut une première analyse manuelle du code étudié afin de décider de la meilleure façon d'extraire les informations contenues dans les commentaires. Souvent ceux-ci sont organisés selon une convention particulière au code. Les noms des sous-programmes et variables peuvent aussi respecter une convention et si nous arrivons à la deviner alors la compréhension du code devient beaucoup plus facile.

3. **Nettoyer** le code : il s'agit de tester le respect de la norme *ANSI* de Fortran77, ou d'autres règles de bonne programmation proposées par des outils d'analyse statique (voir le chapitre suivant). Les codes que nous avons analysés contiennent de nombreux entorses à la norme *ANSI*. Les plus graves indiquent souvent des "astuces" de programmation qui n'ont aucun "sens" dans le modèle du domaine employé par le code et perturbent donc notre appréhension de ce modèle. C'est pourquoi il peut être intéressant de les localiser. Par ailleurs, certains outils d'analyse statique n'acceptent pas en entrée des codes contenant certaines entorses à la norme *ANSI*. Alors il faut transformer le code afin d'éliminer l'anomalie qui empêche son analyse par l'outil, sans modifier sa sémantique. Pour la même raison, il peut être nécessaire d'éliminer les numéros de ligne qui se trouvent à la fin des lignes dans certains codes Fortran77, les lignes plus longues que 70 caractères, ... Une fois cette étape de nettoyage achevée, de nombreuses analyses automatiques ou semi-automatiques peuvent être effectuées et ré-effectuées pendant les étapes suivantes : elles sont évoquées dans les descriptions détaillées de chaque étape.

4. **Exclure** les aspects "non-fonctionnels" du code : cette étape sera décrite dans le Chapitre 6 (Section 6.1).

5. **Restructurer automatiquement** le code, en effectuant des transformations vers un autre code Fortran, ou exprimé dans une extension de Fortran accepté par les outils d'analyse statique utilisés par la suite, comme décrit dans la Section 6.2 du Chapitre 6.

6. **Décomposer** certains des sous-programmes initiaux du code en nouveaux sous-programmes, comme décrit dans la Section 6.3 du Chapitre 6.

7. **Inférer les profils** des prédicats et opérations spécifiant les sous-programmes. La méthode d'inférence est décrite dans le Chapitre 7.

8. **Inférer la spécification partielle des sortes** et compléter l'inférence des profils de l'étape précédante, comme décrit dans la Section 8.1 du Chapitre 8. Cette analyse peut être effectuée "en parallèle" avec la suivante pour éviter d'avoir à traiter en une fois le code entier.

9. **Délimiter les types abstraits de données** comme proposé dans la Section 8.3 du Chapitre 8 et décomposer le code en fragments dont chacun correspond le mieux possible à un type abstrait de données. Ces fragments seront analysés davantage et spécifiés un par un.

10. **Inférer les domaines des opérations partielles** de la spécification concrète du fragment de code, comme décrit dans la Section 8.2 du Chapitre 8 et la Section 9.9 du Chapitre 9.

11. **Abstraire** : appliquer à la main des transformations de l'organigramme du fragment de code, chercher à la main les invariants des boucles et étudier le code jusqu'à avoir compris sa fonction sur un niveau plus abstrait que celui de la spécification concrète, comme expliqué dans la Section 3.7 ci-dessus.

12. **Axiomatiser** les sortes, prédicats et opérations de la spécification concrète du fragment de code : cette étape est illustrée sur l'exemple principal de cette thèse, dans le Chapitre 9.

Chapitre 4

Analyse statique de codes Fortran77

Résumé

Ce chapitre contient un survol des techniques et outils de l'analyse statique, focalisé sur l'analyse de codes programmés en Fortran77 et sur les résultats susceptibles d'être utiles dans la rétro-ingénierie.

Nous commençons, dans la Section 4.1, par une définition de l'analyse statique et nous décrivons les informations qu'elle peut fournir dans la Section 4.2.

La Section 4.3 donne, à l'aide de deux exemples, une idée la nature des problèmes posés par cet analyse et des approches proposées pour les résoudre.

Dans la Section 4.4, nous décrivons et comparons quelques outils opérationnels réalisant l'analyse statique de codes Fortran77.

4.1 En quoi consiste l'analyse statique?

L'*analyse statique* est le recueil de certaines informations sur un programme au moyen d'une analyse du code source ne nécessitant pas l'exécution du programme sur des données réelles : c'est une analyse pouvant s'effectuer au moment de la compilation ("compile time") et non pas lors de l'exécution ("run time"), à la différence du test de logiciel. Contrairement au test, l'analyse statique fournit des propriétés du programme qui sont vraies pour toutes les exécutions possibles. C'est pourquoi elle est intéressante comme aide à l'élaboration d'une spécification.

L'analyse syntaxique

Tout analyse statique commence par une *analyse syntaxique* qui permet la construction des tables de symboles et fournit directement des informations **ne prenant pas en compte l'ordre d'exécution des instructions**. L'analyse syntaxique repose sur une définition de la *grammaire* du langage de programmation du code à analyser.

L'analyse de flot de contrôle

À partir des informations fournies par l'analyse syntaxique du code, nous pouvons construire le *graphe de contrôle* (ou organigramme). La construction, analyse, et éventuellement transformation de celui-ci constituent l'*analyse de flot de contrôle*. Les nœuds du graphe sont des *"basic blocks"* (blocs de base) : soit des suites d'instructions sans branchements dans le flot de contrôle ni appels de sous-programme, soit des instructions conditionnelles, `GOTO`, têtes de boucle `DO` ou appels de sous-programme. Les arcs dirigés indiquent des séquences d'exécution (apparemment) possibles des nœuds. Un *chemin d'exécution* est une suite de nœuds dans le graphe de contrôle, dont chaque nœud (sauf le dernier) est relié au suivant par un arc de la bonne orientation.

Pour effectuer l'analyse *inter-procédurale* (c'est-à-dire d'un code comportant plus d'un sous-programme, par opposition à l'analyse *intra-procédurale*), il faut construire le *graphe des appels*. Ce graphe a des

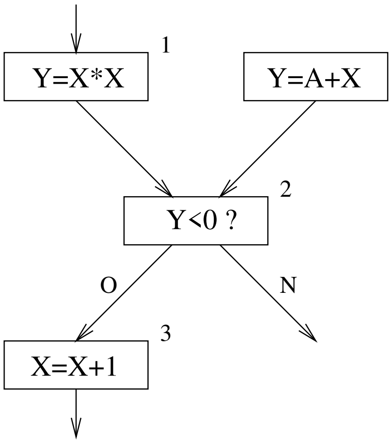


FIG. 4.1: Un exemple d'organigramme avec chemin non exécutable : le chemin composé des blocs numérotés 1,2,3 n'est jamais suivi puisque x^2 et donc y est toujours positif

sous-programmes comme nœuds. Dans la version la plus simple de ce graphe, chaque arc dirigé indique l'existence d'au moins un site d'appel du sous-programme cible de l'arc, depuis le sous-programme source.

L'analyse de flot de données

L'*analyse de flot de données* emploie aussi le graphe de contrôle, mais pour fournir une description des **variables** à différents **points** dans un fragment de code. Elle inclut l'analyse "classique" de flot de données et l'*exécution symbolique*. Le *"data dependence"* [1] étend l'analyse de flot de données avec d'autres techniques afin d'extraire plus de renseignements des fragments de code comportant des boucles `DO` et des accès aux éléments de tableaux.

Dans la suite de ce chapitre, nous parlerons surtout de l'analyse de flot de données.

Les chemins non exécutables

L'analyse statique prend donc en compte tous les chemins d'exécution apparemment possibles. Notons qu'un même chemin d'exécution peut passer plusieurs fois par le même point dans le code : à cause d'une boucle ou d'appels nombreux d'un même sous-programme.

> Prenons comme exemple un appel de sous-programme se trouvant à l'intérieur d'une boucle : l'analyse statique ne différencie pas les différents appels, chacun avec ses propres **valeurs** de paramètres effectifs, résultant des itérations successives de la boucle. Elle ne manipule qu'un seul *site d'appel.*

L'inconvénient de l'analyse statique est que les résultats peuvent être entachés d'une certaine imprécision. En effet, un programme peut comporter des chemins d'exécution apparents, c'est-à-dire, qui apparaissent dans son organigramme mais en réalité ne peuvent être exécutés, voir l'exemple de la Figure 4.1. Il y a des techniques permettant d'éliminer certains chemins non exécutables mais le problème général est indécidable.

Une sémantique partielle ou approximative du programme

Le premier domaine de l'analyse statique, et celui qui a donné lieu à la plus grande part des travaux, est celui de la construction de compilateurs. Cette application exige une mise en œuvre entièrement automatique et relativement efficace (complexité linéaire en la taille du programme, de préférence). La plupart des techniques d'analyse statique sont donc spécialisées pour le recueil de certaines informations partielles sur le programme. Ces informations sont exprimées dans un domaine plus pauvre que celui

1. Ce chapitre contient plusieurs termes qui ne sont pas traduits de l'anglais, car la littérature sur ce domaine existe presque exclusivement en anglais et certains termes en anglais sont bien établis et employés partout.

des sémantiques habituelles de programmes et sont souvent approximatives. En effet, à l'approximation incontournable due à la prise en compte de chemins d'exécution jamais suivis, peut s'ajouter d'autres approximations afin d'assurer l'efficacité du calcul voire la calculabilité tout court.

L'exception est l'exécution symbolique [2], qui a pour objectif de fournir une information précise équivalente à la sémantique axiomatique de [Hoa69], mais qui est semi-automatique, peu efficace et n'est généralement mise en œuvre que de façon très partielle (par exemple : sur un sous-ensemble restreint du langage).

L'interprétation abstraite est l'étude théorique des techniques d'analyse statique dont les résultats sont approximatifs. Son objectif est la justification d'algorithmes d'analyse, en comparant la sémantique approximative du programme fournie par l'analyse statique à une sémantique exacte.

4.2 Les informations apportées par l'analyse statique

L'analyse syntaxique

La simple analyse syntaxique du programme permet de fournir, pour chaque instruction ou bloc de base, des informations telles que :

- l'ensemble des variables dont la valeur est lue (au sens large, y compris, par exemple, pour la copier sur un fichier de sortie, pour l'utiliser comme indice dans un tableau) : on dira par la suite, les *utilisations* (ou lectures) de variable ("uses" en anglais),
- l'ensemble des affectations de valeurs aux variables (y compris en y copiant une valeur lue dans un fichier d'entrée) : on dira par la suite, les *modifications* (ou écritures) de variable (appelées "definitions" dans la littérature),
- pour une instruction contenant un appel à un sous-programme, les *paramètres effectifs* de l'appel,
- etc.

Sans même suivre les enchaînements possibles des instructions individuelles, on peut regrouper et comparer ces données à la manière des générateurs de références croisées des compilateurs, afin d'obtenir des informations comme :

- les variables jamais modifiées ou jamais utilisées,
- les sous-programmes externes et ceux qui ne sont pas appelés,
- des instances d'incompatibilité de types,
- des comparaisons des déclarations différentes d'un même bloc `COMMON`,
- les ensembles de paramètres effectifs possibles pour chaque sous-programme,
- les variables pouvant être modifiées et celles pouvant être utilisées au cours de l'exécution d'un sous-programme,
- etc.

L'analyse de flot de contrôle

Ensuite, l'analyse de flot de contrôle, permet d'identifier les boucles (et leur nœud d'entrée) dans les programmes non structurés, à l'aide de la relation de *dominance* sur les nœuds [ASU86, Hec77], ainsi que les morceaux de code vers lesquels il n'y a pas d'instruction de transfert de contrôle.

2. La terminologie dans ce domaine est très floue : ce que nous appelons exécution symbolique (en suivant certains auteurs d'articles récents et pour employer un terme bien connu) est peut-être plus correctement appelée *interprétation symbolique* ou *évaluation (globale) symbolique*, puisqu'il concerne l'analyse de l'ensemble des chemins d'exécution. Certains auteurs réservent le terme exécution symbolique à l'analyse d'un seul chemin. L'*évaluation partielle* utilise une approche similaire afin de transformer le programme, ce qui n'est pas le but ici.

L'analyse de flot de données

Le suivi des enchaînements possibles des instructions individuelles, permet aussi d'effectuer l'analyse de flot de données afin de calculer, à partir des données recueillies sur chaque instruction :

- pour un point donné, les *"reaching definitions"*, i.e. les modifications de variables pouvant être intervenues le long des différents chemins menant à ce point et pouvant ne pas avoir été supprimées par d'autres avant l'arrivée à ce point ;
- pour un point donné, les variables susceptibles d'être utilisées (*"live variables"*) dans les différents chemins partant de ce point ;
- pour chaque point de modification d'une variable donnée, les points d'utilisation possibles et *vice-versa* (appelé *"def-use chains"* resp. *"use-def chains"* dans la littérature)
- etc.

Nous pouvons connaître ainsi les "dépendances" d'une valeur de variable, c'est-à-dire : les variables et constantes intervenant dans l'expression de sa valeur au point de modification, et pour chacune de ces variables, les variables et constantes intervenant dans une "reaching definition", et ainsi de suite.

Nous pouvons aussi extraire la *"slice"* (tranche) [Wei84] du programme associée à une variable et à une instruction donnée, c'est-à-dire :

- soit l'extrait exécutable du programme qui reprend toutes les instructions pouvant influer sur la valeur de cette variable à ce point et qui reproduit le comportement du programme à leur égard (*"backward slice"*),
- soit l'extrait du programme qui peut dépendre de la valeur à ce point de cette variable (*"forward slice"*).

L'exécution symbolique

Plus généralement, le but de l'exécution symbolique est de fournir pour chaque point du programme :

- pour chaque chemin permettant d'arriver à ce point, la contrainte sur la valeur des variables d'entrée du programme qui assure que c'est ce chemin qui sera emprunté : l'ensemble de ces contraintes partitionne les ensembles de valeurs possibles des variables d'entrée,
- et pour chacune de ces contraintes, une expression close définissant la valeur à ce point des variables de sortie et variables locales, en fonction des valeurs des variables d'entrée.

La précision des résultats de l'analyse de flot de données

Prenons le cas où plusieurs chemins d'exécution mènent au même point du programme. Chaque chemin correspond à une contrainte sur les valeurs des variables d'entrée et chaque chemin induit une valeur symbolique pour les variables. Dans le résultat fourni par l'exécution symbolique à ce point, chaque valeur possible des variables est conditionnée par la contrainte du chemin. Or la contrainte d'un chemin peut être insatisfiable, ce qui exclurait la valeur correspondante des valeurs possibles.

Les techniques d'analyse de flot de données autres que l'exécution symbolique fournissent un résultat qui n'est pas conditionné par une contrainte sur les valeurs des variables d'entrée, ou qui n'intègre qu'une partie de la vraie contrainte et peut donc être approximatif. En règle générale, l'analyse de flot de données ne peut pas répondre précisément à des questions telles que "quel est l'ensemble des variables qui changent de valeur pendant l'exécution de ce fragment de code?", car la réponse exacte dépend du chemin d'exécution suivi, et donc des valeurs des entrées. Elle peut donner deux réponses :

1. l'ensemble (appelé *"must"*) de variables qui changent de valeur dans **tout** chemin apparemment possible : cette réponse peut être une sous-approximation pour des valeurs particulières des entrées car elle risque d'exclure des variables, par rapport à la réponse exacte, ou

2. l'ensemble (appelé *"may"*) de variables qui changent de valeur sur **au moins un** des chemins apparemment possibles : cette réponse peut être une sur-approximation pour des valeurs particulières des entrées car elle risque d'inclure trop de variables, par rapport à la réponse exacte. Elle peut aussi être une sur-approximation de la vraie réponse pour l'ensemble des valeurs possibles des entrées, car elle peut inclure des variables modifiées par des chemins d'exécution non exécutables en réalité.

Selon l'utilisation qui en sera faite, il faut donc décider laquelle des deux réponses est *prudente* (en anglais : "conservative" ou "safe") : est-ce qu'il faut la réponse *"must"* dans laquelle on est sûr de ne pas retrouver une variable qui ne change pas de valeur, ou bien la réponse *"may"*, dans laquelle on est sûr de retrouver toute variable qui peut changer de valeur ?

4.3 La mise en œuvre de l'analyse de flot de données

Cette section a pour objectif de fournir une idée de la nature des problèmes posés par l'analyse de flot de données et d'indiquer les approches proposées dans la littérature pour les résoudre. La description des outils mettant en œuvre ces approches sera le sujet de la section suivante. Plusieurs présentations synthétiques de l'analyse de flot de données "classique" existent dans la littérature [ASU86, ZC91, Hec77, Ken81].

La mise en œuvre de l'analyse de flot de données sera introduite par deux exemples simplifiés ne prenant en compte ni les appels de sous-programme, ni la manipulation de tableaux, ni les alias de variables.

Nous supposons donné un graphe de contrôle avec un nœud d'entrée, e, et un nœud de sortie, s. L'ensemble $pred(n)$ des *prédécesseurs* d'un nœud n dans le graphe est l'ensemble des nœuds reliés à n par un arc qui débute dans le prédécesseur.

4.3.1 Un premier exemple - les "reaching definitions"

Dans le premier exemple, il s'agit de trouver les *modifications* de variable pouvant être valides au moment de l'exécution d'un nœud n. On appellera cet ensemble $reach(n)$. Il s'agit d'une analyse *"may"*. On annote chaque nœud n du graphe de contrôle avec les modifications de variable qu'il contient : c'est l'ensemble $gen(n)$. Si un nœud contient plusieurs modifications successives de la même variable, alors la dernière modification supprime les autres, car le flot de contrôle à l'intérieur d'un nœud est toujours séquentiel, par définition. On dit qu'une modification d de variable v atteignant le nœud n est *tuée* ("killed") par n si $gen(n)$ contient une autre modification de v. Une fois qu'on connaît toutes les modifications de variable du programme, on peut alors ajouter une deuxième annotation aux nœuds : l'ensemble $kill(n)$ des modifications tuées dans ce nœud. En effet pour chaque modification d'une variable v dans $gen(n)$, on ajoute à $kill(n)$ toutes les autres modifications de v générées quelque part dans le programme.

La fonction de transfert

Appelons $MODS$ l'ensemble de toutes les modifications possible de variable. La valeur de $reach(n)$ est une partie de $MODS$.

Définissons une *fonction de transfert* $f_n : \mathcal{P}(MODS) \to \mathcal{P}(MODS)$ pour le noeud n qui, pour une valeur de $reach(n)$ en entrée, calcule l'ensemble des modifications à la sortie du noeud. Le résultat de f_n est donc l'union des modifications en entrée qui ne sont pas tuées dans n et des nouvelles modifications introduites dans n :

$$\forall r \in \mathcal{P}(MODS).\ f_n(r) = (r - kill(n)) \cup gen(n)$$

Les équations de flot de données

Ensuite, on doit prendre en compte les enchaînements possibles des nœuds du graphe afin de calculer $reach$ pour chacun.

Le nœud d'entrée e est un cas spécial, aucune modification ne l'atteint :

$$reach(e) = \emptyset$$

Pour tout autre nœud n, l'ensemble des modifications de variables pouvant l'atteindre est l'union de modifications arrivant de tous les prédécesseurs de n :

$$reach(n) = \bigcup_{m \in pred(n)} f_m(reach(m)) \quad \text{si} \quad n \neq e$$

Cette équation est un exemple d'*équation de flot de données*. Il y a la même équation pour chaque nœud du graphe, ce qui donne un système d'équations simultanées pour le graphe entier. Si le graphe contient des boucles alors le système d'équations est récursif.

Un algorithme itératif

La solution de ce système d'équations de flot de données est facilement obtenue par un algorithme itératif. Cet algorithme initialise $reach(n)$ à $\emptyset$ pour chaque nœud. Ensuite, il propage les modifications dans le graphe de contrôle en recalculant à chaque itération la valeur de $reach(n)$ pour tous les nœuds, un par un, à l'aide des équations de flot de données. Il s'arrête quand il atteint un point fixe (qui existe dans ce cas, comme nous allons l'expliquer plus loin).

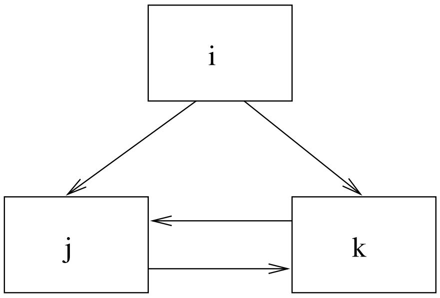


FIG. 4.2: Flot de contrôle dans un exemple de fragment de code non réductible

Plusieurs approches ont permis d'améliorer l'efficacité de la résolution des équations de flot de données ([ASU86, RP86, Ken81]). Cependant, les techniques employant une *grammaire attribuée* doivent traiter le `GOTO` de façon ad-hoc et les approches d'*élimination* ne sont applicables qu'aux graphes de contrôle *réductibles* (informellement : dont les boucles n'ont qu'un point d'entrée, pour une définition précise voir [ASU86, Ken81] : un contre-exemple est donné dans la Figure 4.2). Or l'utilisation exclusive des primitives de contrôle structuré garantit des graphes réductibles. Des programmes écrits en Fortran peuvent ne pas être réductibles, mais des études de codes réels ont montré que la vaste majorité des codes employant des `GOTO` sont tout de même réductibles [ASU86, Ken81]. Toutefois, les codes EDF que nous avons analysés contiennent au moins une boucle irréductible : dans le sous-programme `ODIDAV` de *SILODI*.

4.3.2 Un autre exemple - l'exécution symbolique

L'exécution symbolique ([CR81]) repose sur une "exécution" du code où les valeurs des variables d'entrée sont représentées par des expressions symboliques. Par exemple, si la valeur de M dans un programme Fortran est lue dans un fichier, alors on lui attribue la valeur symbolique α et après exécution de l'instruction `L=5*M`, on attribue à L la valeur 5α. On appelle *état symbolique* les valeurs symboliques de toutes les variables à un point donné du programme.

Les prédicats de branche

On annote chaque arc du graphe de contrôle par la condition devant être satisfaite pour qu'il soit emprunté lors de l'exécution. On appellera de telles étiquettes des *prédicats de branche* :

- ils sont mis à *vrai* quand il n'y a qu'un transfert de contrôle possible ;
- à la sortie d'une instruction `IF/THEN/ELSE` ou `IF` *logique* le prédicat de la branche `THEN` est l'expression paramètre du `IF` et celui de la branche `ELSE` sa négation ;
- dans le cas d'un `IF` *arithmétique*, les prédicats expriment respectivement l'infériorité de l'expression à 0, son égalité à 0 et sa supériorité à 0 ...

Les conditions de chemin

Pour un chemin donné du graphe, la *condition de chemin* est la conjonction des prédicats de branche de tous les arcs le composant, après avoir substitué dans chaque prédicat les valeurs des variables de l'état symbolique effectives au moment du passage par l'arc. Évidemment, une condition de chemin incohérente indique un chemin qui ne peut être suivi.

La forme du résultat de l'exécution symbolique

Si un seul chemin possible mène à un nœud n (autre que le nœud e), alors le résultat de l'exécution symbolique à l'entrée de n est la conjonction de la condition du chemin et de l'état symbolique à la fin du chemin. Dans le cas de plusieurs chemins menant à n, le résultat de l'exécution symbolique à l'entrée de n est la disjonction des résultats à la sortie des nœuds précédents. Ce résultat peut être présenté sous forme d'expression “case” où chaque état symbolique possible est conditionné par sa condition de chemin.

Les fonctions de transfert

Les nœuds de branchement ont pour effet de modifier la condition de chemin, et les autres, l'état symbolique. Pour ces derniers, le résultat de l'exécution symbolique à la sortie est la conjonction de celui à l'entrée et de la modification de l'état symbolique effectuée par le nœud. Toutefois, dans le cas de la réaffectation d'une variable qui a déjà une valeur, nous remplaçons l'ancienne valeur dans l'état symbolique. C'est pourquoi il faut substituer dans chaque nouveau prédicat de branche les valeurs courantes des variables données par l'état symbolique. Si le résultat de l'exécution symbolique à l'entrée du nœud est sous la forme d'une disjonction, alors pour chaque terme de la disjonction, nous pouvons faire la conjonction de l'état symbolique et de la modification effectué par le nœud. Ceci à cause de la distributivité de la conjonction logique. Ainsi, nous gérons séparément les états symboliques et les conditions de chemin.

Le résultat de l'exécution symbolique à la sortie d'un nœud de branchement est la conjonction du résultat à l'entrée et du prédicat de branche pour la branche suivie. Dans le cas d'une disjonction à l'entrée du nœud, nous pouvons faire la conjonction du prédicat de branche avec chaque condition de chemin.

La démarche

Hormis les éventuelles boucles, l'exécution symbolique s'effectue depuis l'entrée du graphe vers la sortie, en suivant “en parallèle” les différents chemins d'exécution de façon à avoir toujours exécuté tous les nœuds précédant le nœud courant. Avant chaque branchement du flot de contrôle, les différents prédicats de branche sont évalués, à partir de l'état symbolique courant, afin de détecter toute incohérence avec la condition courante du chemin. Un démonstrateur de théorèmes ou un système de résolution de contraintes peut être utilisé à cette fin, mais comme le problème général est indécidable, il peut y avoir des chemins pour lesquels on poursuit l'interprétation symbolique sans savoir s'ils sont faisables ou non. La mise en œuvre de l'exécution symbolique pose trois problèmes majeurs :

1. le temps de calcul demandé par ces techniques,

2. l'explosion combinatoire du nombre de chemins possibles s'il y a trop de points de décision dans le code

3. et le traitement des boucles. Si la première itération d'une boucle donne des expressions pour les valeurs des variables qui ne permettent pas de déterminer si la condition de sortie de la boucle est satisfaite, et si une deuxième itération a pour effet de changer ces valeurs, alors on devrait effectuer la deuxième itération. Deux chemins sont alors créés, l'un pour le cas où on sort de la boucle après une seule itération, et l'autre pour le cas où la deuxième itération est effectuée. Mais si on suit ce principe et si les itérations suivantes n'apportent pas plus de précisions sur la satisfaction ou non de la condition de sortie, tout en continuant de modifier les valeurs des variables, alors on ne terminera jamais l'exécution symbolique de la boucle.

   Dans ce cas, la solution consiste à exprimer la valeur des variables modifiées dans la boucle, ainsi que la condition de sortie, en fonction du nombre d'itérations. On cherche d'abord à exprimer la valeur de chaque variable après k itérations en fonction de sa valeur après $k-1$ itérations. À partir de ces définitions et de la valeur des variables à l'entrée de la boucle, on obtient par induction leur valeur après j itérations. Ensuite, on évalue la condition de sortie à partir de cette valeur des variables et obtient donc une expression du nombre d'itérations effectuées. Les expressions symboliques ainsi obtenues représentent tous les chemins, même si le nombre d'itérations varie d'un chemin à l'autre.

   Or les définitions inductives ne peuvent pas toujours être trouvées automatiquement. Il faut l'intervention de l'utilisateur et même pour ce dernier le problème peut être trop difficile. Des méthodes d'approximation sont proposées dans [CC77]. Dans [CPPGM91] on propose de continuer l'exécution symbolique à la sortie de la boucle, en donnant des valeurs symboliques fraîches aux variables modifiées (de façon alors inconnue) par la boucle.

4.3.3 Un modèle théorique général

La recherche des “reaching definitions” et l'exécution symbolique représentent les deux extrêmes de difficulté des problèmes d'analyse de flot de données. Ces deux problèmes, ainsi que celui des “live variables” évoqué dans la Section 4.2 et de nombreux autres problèmes classiques d'analyse de flot de données sont des interprétations différentes d'un même modèle théorique.

Le but d'un problème donné d'analyse de flot de données est de trouver pour tout nœud n du graphe de contrôle, la valeur $p(n)$ d'une certaine propriété à l'entrée du nœud, étant donné $p(e)$ (la valeur de p à l'entrée du programme) et la fonction de transfert f_n (qui prend $p(n)$ en entrée et rend la valeur de p à la sortie de n) pour tous les nœuds[3]. Dans le cas de l'exécution symbolique de nœuds de branchement, il y a une fonction de transfert pour chaque sortie du nœud.

Monotone data flow analysis framework

Or les problèmes d'analyse de flot de données que nous avons évoqués satisfont deux conditions :

1. Les valeurs possibles de la propriété en question forment un *treillis complet* dont, intuitivement, l'ordre représente la quantité d'informations. La borne supérieure (**sup**) du treillis traduit l'effet combiné des différents nœuds précédents[4].

   Une relation d'ordre $\leq$: $L \times L$ est une relation réflexive ($\forall a \in L.\ a \leq a$), antisymétrique ($\forall a, b \in L.\ (a \leq b \wedge b \leq a) \Rightarrow a = b$) et transitive ($\forall a, b, c \in L.\ (a \leq b \wedge b \leq c) \Rightarrow a \leq c$).

   Un *ensemble partiellement ordonné* (*“poset”*) $< L, \leq >$ est un ensemble L muni d'une relation d'ordre $\leq$: $L \times L$.

   Soit $H \subseteq L$. Tout $a \in L$ tel que $\forall b \in H.\ b \leq a$ s'appelle *majorant* de H (notons qu'a n'appartient pas forcément à H). Si H n'est pas vide, le plus petit des majorants s'appelle *borne supérieure*

3. En fait pour certains problèmes d'analyse statique - par exemple celui des “live variables”, dits de *flot en arrière* par opposition à ceux comme celui des définitions atteignables dits de *flot en avant* - il s'agit de définir la valeur de p à l'entrée du nœud en fonction de la valeur en sortie, et de propager les informations vers l'arrière depuis la valeur de p à la sortie de s, mais le problème résultant est le *dual* de celui décrit ici.

4. Dans la littérature, on s'arrange souvent pour définir le treillis de façon à utiliser le **inf**, mais nos exemples sont beaucoup plus intuitifs si on emploie le treillis dual, i.e. avec le **sup** ($\sqcup$).

> ou **sup** de H ($\sqcup H$) (*"join"* en anglais). Le *minorant* et *borne inférieure* ou **inf** de H ($\sqcap H$) (*"meet"* en anglais) ont des définitions duales (où $\leq$ est remplacé par $\geq$).
>
> Si $\sqcup H$ et $\sqcap H$ existent pour tout $H \subseteq L$ alors nous avons un *treillis complet* : $< L, \leq, \perp, \top, \sqcup, \sqcap >$, où $\perp = \sqcap L$ et $\top = \sqcup L$.

Dans le cas des "reaching definitions", le treillis est $< \mathcal{P}(MODS), \subseteq, \emptyset, MODS, \cap, \cup >$.

Dans le cas de l'exécution symbolique, les éléments du treillis sont des formules de la logique du premier ordre (implicitement quantifiées par les variables d'entrée) caractérisant des états de la mémoire (c'est-à-dire, des ensembles de variables du programme avec leur valeur). $\leq$ est l'inférence logique $\Rightarrow$, $\sqcup$ est la disjonction logique $\vee$, $\sqcap$ est la conjonction logique $\wedge$, $\perp$ est $false$ (ne caractérisant aucun état de la mémoire) et $\top$ est $true$ (caractérisant tous les états de la mémoire).

2. Les fonctions de transfert $f_n : L \to L$ sont *monotones* : ceci garantit l'existence de points fixes pour les fonctions de transfert et ainsi pour le système d'équations de flot de données.

> Une fonction $f : L \to L$ est *monotone* si et seulement si $\forall a, b \in L.\ a \leq b \Rightarrow f(a) \leq f(b)$.
>
> Corollaire : $f : L \to L$ est *monotone* si et seulement si $\forall a, b \in L.\ f(a \sqcup b) \leq f(a) \sqcup f(b)$.
>
> Un *point fixe* d'une fonction $f : L \to L$ est un élément a de L tel que $f(a) = a$.
>
> Soit L un treillis complet et $f : L \to L$ monotone, alors les points fixes de f (avec le même ordre que pour L) forment un treillis complet non vide.

Tout problème d'analyse de flot de données avec ces deux propriétés est une instance de *"monotone data flow analysis framework"* (MDF) [KU77]. L'analyse statique étant définie comme la prise en compte de tous les chemins d'exécutions apparemment possibles, la bonne valeur de $p(n)$ est la solution *"meet over all paths"* (MOP) : le **sup** des valeurs de p à la sortie de chaque chemin menant à n depuis e. Appelons $PATH(n)$ l'ensemble de ces chemins. La valeur de p à la sortie du chemin $c \in PATH(n)$ est la *composition* f_path des fonctions de transfert f_i pour tous les nœuds i de c (dans l'ordre), appliquée à $p(e)$. Mais $p(e) = \perp$. La solution MOP est donc la solution au système d'équations :

$$p(n) = \bigcup_{c \in PATH(n)} f\_path(\perp) \text{pour tous les noeuds} \quad n \quad \text{sauf} \quad e$$

Le problème général de la recherche de la solution MOP d'un MDF est indécidable. Toutefois, si les fonctions de transfert sont non seulement monotones mais *distributives* alors [KU77] a démontré que la solution MOP pour tous les nœuds est égale au plus petit point fixe (**lfp**) du système d'équations de flot de données (la solution LFP). C'est la solution trouvée par l'algorithme itératif décrit pour les "reaching definitions".

> Une fonction totale $f : L \to L$ est *distributive* si et seulement si $\forall a, b \in L.\ f(a \sqcup b) = f(a) \sqcup f(b)$.

Or dans le cas des fonctions de transfert monotones mais pas distributives, la solution LFP est plus petite que la solution MOP et donc en est une approximation *prudente* (voir Section 4.2).

Les propriétés du treillis déterminent si un algorithme itératif prenant $\perp$ comme valeur initiale, tel que celui décrit pour le problème des "reaching definitions", termine ou non. S'il termine (comme pour les "reaching definitions"), alors il calcule la solution LFP. S'il ne termine pas, ou si l'itération est trop long, d'autres approches sont parfois possibles, comme celles proposées pour l'exécution symbolique ou dans [CC77].

L'interprétation abstraite

L'*interprétation abstraite* [CC77, CC79, Cou81] propose de comparer les valeurs calculées par l'analyse statique au treillis complet d'une sémantique formelle exacte des programmes, afin de prouver formellement que l'analyse statique fournit toujours une approximation prudente de la propriété étudiée.

Cette démarche n'est intéressante que pour l'analyse statique de propriétés "sémantiques" du programme (et non pas "syntaxiques" comme les "reaching definitions") et dans les cas où ce n'est qu'une valeur approximative de la propriété étudiée qui est calculée.

Appelons le treillis de la sémantique exacte D et supposons que les valeurs calculées par l'analyse statique forment un autre treillis complet $D^\sharp$. Définissons une *fonction d'abstraction* $\alpha : D \rightarrow D^\sharp$ telle que la plus petite sur-approximation d'une valeur $d \in D$ calculée par l'analyse statique est $\alpha(d)$. Si nous pouvons définir une autre fonction (la *fonction de concrétisation*) $\gamma : D^\sharp \rightarrow D$ telle que α est γ sont *adjointes* alors nous savons que l'analyse statique calcule toujours une approximation prudente.

Soient D et $D^\sharp$ des treillis complets, $\alpha : D \rightarrow D^\sharp$ et $\gamma : D^\sharp \rightarrow D$ sont *adjointes* si et seulement si

- α et γ sont monotones et
- $\forall d \in D, d^\sharp \in D^\sharp.\ \alpha(d) \leq d^\sharp \Leftrightarrow d \leq \gamma(d^\sharp)$

Définissons

- une relation **approx** : $D^\sharp \times D$ telle que
  $\forall d \in D, d^\sharp \in D^\sharp.\ d^\sharp$ **approx** $d \Leftrightarrow \alpha(d) \leq d^\sharp$
- et une autre relation **approx** : $D^\sharp \rightarrow D^\sharp \times D \rightarrow D$ telle que
  $\forall f : D \rightarrow D, f^\sharp : D^\sharp \rightarrow D^\sharp.\ f^\sharp$ **approx** $f \Leftrightarrow \forall d \in D, d^\sharp \in D^\sharp.\ d^\sharp$ **approx** $d \Rightarrow f^\sharp(d^\sharp)$ **approx** $f(d)$.

Alors nous avons :

1. $\forall d_1, d_2 \in D, d_1^\sharp, d_2^\sharp \in D.\ d_1^\sharp$ **approx** $d_1 \wedge d_2^\sharp$ **approx** $d_2 \Rightarrow d_1^\sharp \sqcup d_2^\sharp$ **approx** $d_1 \sqcup d_2$
2. $\forall f : D \rightarrow D, f^\sharp : D^\sharp \rightarrow D^\sharp.\ f^\sharp$ **approx** $f \Rightarrow$ **lfp** $f^\sharp$ **approx lfp** f

Cette relation idéale entre la sémantique du programme et les résultats de l'analyse statique est souvent difficile à établir en pratique [CF97]. D'autres approches [Mar97] proposent alors de relâcher certaines des contraintes ci-dessus et de prouver des propriétés moins fortes. Cependant, peu d'outils opérationnels d'analyse statique ont été prouvés de cette façon, encore moins en employant la sémantique formelle de toutes les constructions d'un langage de programmation, très compliquée à définir pour un langage tel que Fortran77.

4.3.4 L'analyse inter-procédurale

Examinons le cas où l'analyse statique doit tenir compte des appels de sous-programme. On peut effectuer l'analyse inter-procédurale en faisant l'*expansion en-ligne* des appels de sous-programme: on remplace chaque site d'appel par le code du sous-programme appelé, en substituant les paramètres formels par les paramètres effectifs. Ayant éliminé ainsi tous les appels de sous-programme, on n'a plus qu'à effectuer l'analyse statique intra-procédurale du code résultant et on obtient des informations avec la même précision. L'absence de récursion dans Fortran77 rend cette approche faisable, mais elle nécessite de dupliquer l'analyse de tout sous-programme appelé depuis plusieurs sites. De plus, elle ne permet pas de caractériser un sous-programme donné, ce qui est gênant dans notre cadre de rétro-ingénierie.

Il y a des approches permettant de minimiser le nombre d'analyses de chaque sous-programme, mais au risque d'obtenir une information moins précise que pour l'expansion en-ligne.

L'approche extrême dans ce sens, adoptée souvent dans les générateurs de références croisées, est de ne pas analyser le sous-programme mais de supposer que chacun de ses paramètres peut être modifié et peut être utilisé.

Une autre approche consiste à trouver des informations *sommaires* pour chaque sous-programme :

- les paramètres et variables globales pouvant être modifiés,
- les paramètres et variables globales pouvant être utilisés,
- les paramètres et variables globales toujours modifiés,
- ...

et d'intégrer ces informations dans l'analyse du sous-programme appelant à chaque site d'appel, après substitution des paramètres effectifs pour les paramètres réels.

Dans le cas d'un graphe d'appels sans cycles, les sous-programmes peuvent être analysés dans l'ordre donné en remontant le graphe des appels depuis les feuilles, de façon à assurer que chaque site d'appel sera analysé après le sous-programme appelé. L'analyse s'effectue alors en une seule fois.

4.3.5 Prise en compte des alias

Quand deux identificateurs différents de variables nomment la même zone de mémoire ou nomment des zones de mémoire qui se chevauchent, on dit que les deux variables sont des *alias*. En Fortran77, ce phénomène peut être induit par :

1. L'instruction `EQUIVALENCE` évoquée dans le Chapitre 3 (Section 3.1.5) : cette instruction permet de déclarer dans un sous-programme p une variable l qui partagera le même emplacement mémoire qu'une autre variable c. On dit que les variables l et c sont des *alias statiques*. Dans le code de p, les utilisations (resp. modifications) de l et de c peuvent être entrelacées. Dans l'analyse statique de p, il faut donc traiter toute utilisation (resp. modification) de l comme une utilisation (resp. modification) de c, et *vice-versa*.

2. L'appel d'un sous-programme p avec passage par référence des paramètres : chaque paramètre effectif e devient l'alias du paramètre formel f qu'il instancie. On dit que les variables f et e sont des *alias dynamiques* puisque la relation d'alias est créée par l'appel au sous-programme. Contrairement au cas précédent, les deux noms ne sont pas utilisés dans le même sous-programme - le nom f remplace le nom e pour ce qui concerne le code de p, mais évidemment e peut être lui-même un paramètre formel du sous-programme appelant p.

   On peut donc effectuer d'abord l'analyse statique de chaque sous-programme avec les noms locaux, et déterminer ainsi, par exemple, si p peut utiliser ou modifier son paramètre f. Ensuite, on prendrait en compte les alias crées par le passage par référence des paramètres, en comptant un site d'appel à p comme un point d'utilisation (resp. modification) possible de chaque paramètre effectif e instanciant le paramètre formel f utilisé (resp. modifié) par p.

   Toutefois, dans le cas d'un site d'appel du sous-programme p avec le même paramètre effectif e pour deux paramètres formels différents $f1$ et $f2$, les deux paramètres formels $f1$ et $f2$ deviennent aussi des alias car ils partagent la même adresse en mémoire. Or ce cas ressemble à celui de l'instruction `EQUIVALENCE`: dans le code de p les utilisations (resp. modifications) de $f1$ et de $f2$ peuvent être entrelacées. Si l'on effectue l'analyse statique de ce site d'appel particulier de p (ou s'il se trouve que pour **tous** les sites d'appels de p, $f1$ et $f2$ sont des alias), alors en analysant le code de p, il faut traiter toute utilisation (resp. modification) de $f1$ comme une utilisation (resp. modification) de $f2$, et *vice-versa*.

   Un cas similaire à celui-ci se produit quand une variable de bloc `COMMON` est passée comme paramètre effectif (instanciant le paramètre formel $f1$) dans un appel à un sous-programme p comportant sa propre référence à cette adresse dans le bloc `COMMON`. Pour ce site d'appel, une variable globale ($f2$) a comme alias un paramètre formel ($f1$).

   Dans ces cas, l'utilisation des informations sommaires décrite dans la Section 4.3.4 empêche de prendre en compte le contexte des alias pour un site particulier d'appel. Supposons que p soit composé uniquement d'un `IF` à deux branches, une branche contenant une modification de $f1$ et l'autre branche contenant une modification de $f2$. Les informations sommaires pour p vont être les suivantes :

   (a) $f1$ peut être modifié

   (b) $f2$ peut être modifié

   (c) aucune variable n'est toujours modifiée (c'est-à-dire, dans tous les chemins possibles d'exécution).

   Or dans le cas où $f1$ et $f2$ sont des alias, $f1$ (ou $f2$) est toujours modifiée par p.

   Deux façons de prendre en compte dans l'analyse de flot de données de tels cas créés par les alias dynamiques, sont proposées dans la littérature :

   (a) La première façon consiste à prendre en compte à chaque point de modification ou d'utilisation de variable, uniquement les alias dynamiques effectifs à ce point. Le contexte d'appel est pris en compte pour chaque site d'appel d'une procédure.

(b) La deuxième façon est une approximation obtenue en regroupant toutes les instances d'alias dynamique, sans prendre en compte leur enchaînement, et en effectuant leur fermeture transitive et symétrique : on considère toujours comme alias deux variables qui sont des alias à un point quelconque de l'exécution ou deux variables qui deviennent (à deux points distincts) alias d'une même troisième variable. En effet, on partitionne les variables du code en des classes d'équivalence, et ceci une seule fois. Évidemment, on regroupera dans chaque classe d'équivalence des noms de variables employés dans des sous-programmes différents, mais on sera sûr de prendre en compte aussi le cas évoqué ci-dessus.

3. Des déclarations différentes des variables à l'intérieur du même bloc `COMMON`, par exemple dans le sous-programme appelant p il y a déclaration d'un bloc `COMMON` appelée $DONNEES$ et comportant le seul tableau entier A de dimension 2, tandis que dans p, $DONNEES$ est déclaré comme un bloc comportant deux variables entières, $DEBUT$ et FIN. C'est encore un exemple d'alias statique, mais qui ressemble à celui du passage par référence des paramètres : dans le code de p les variables $DEBUT$ et FIN remplacent $A(1)$ et $A(2)$.

Pour être vraiment utile, la prise en compte des alias doit traiter de façon précise le cas où un tableau (ou partie de tableau) est alias d'un autre. Les parties de tableau concernées peuvent ne pas avoir le même nombre de dimensions.

> Par exemple : comme paramètre effectif, l'élément d'indice 100 d'un tableau A composé d'une seule ligne de 300 éléments, et comme paramètre formel, un tableau B composé de deux lignes de 50 éléments.

4.3.6 Traitement des tableaux

Nous avons expliqué dans le Chapitre 3 (Section 3.1.3) l'utilisation très répandue des tableaux dans les programmes Fortran. Or, dans le cas d'une modification ou d'une utilisation d'un élément de tableau dont l'indice est une variable, on ne sait pas *a priori* quel élément est référencé.

La solution de facilité, très prudente dans le sens expliqué dans la Section 4.1, est de traiter un tel cas comme une modification (resp. utilisation) de tous les éléments du tableau. Cette solution empêche de distinguer les modifications et les utilisations d'éléments différents, ou d'ensembles d'éléments différents, d'un même tableau. Elle ne peut représenter un paramètre effectif composé de certains éléments (mais pas tous) d'un tableau.

Ce problème est adressé par la communauté travaillant sur la parallélisation automatique de programmes, qui étudient surtout des vieux programmes Fortran. Leur but est de savoir si les ensembles d'éléments d'un tableau concernés par deux points d'accès peuvent avoir une intersection non vide. Or les points d'accès aux tableaux sont souvent situés dans des boucles `DO`, et dans la parallélisation de codes Fortran on cherche, entre autre, des possibilités de parallélisation de différentes itérations de boucles `DO`.

Deux familles d'algorithmes ont été développées :

1. Les algorithmes de *"data dependence"* qui comparent les points d'accès à l'intérieur de boucles `DO` deux par deux et décident si le chevauchement est possible : si l'indice du tableau dans chaque cas est une fonction linéaire des variables `DO`, si les variables des boucles `DO` sont entières et si les bornes des boucles `DO` sont des expressions entières, alors le problème s'exprime comme la recherche d'une solution appartenant aux entiers à une équation linéaire dans une région définie par les bornes des boucles.

   Par exemple dans le fragment de code Fortran suivant :

   ```
   ligne 1         DO 4 I = 2, 10
   ligne 2           A(I) = B(I) + 1
   ligne 3           D(I) = A(I - 1)
   ligne 4       4 CONTINUE
   ```

   le problème est de savoir si un même élément du tableau `A` peut être modifié par l'instruction de la ligne 2 lors d'une itération de la boucle et utilisé par l'instruction de la ligne 3 lors d'une autre itération. Soit x la valeur d'`I` pour une itération de la boucle `DO` et y sa valeur dans une autre itération. L'indice de lélément d'`A` modifié dans la première de ces itérations est égal à

x, et l'indice de lélément d'`A` utilisé dans l'autre itération est égal à $y-1$. Nous vérifions donc l'existence d'une solution à l'équation $x = y - 1$ dans la région $(2 \leq x) \wedge (y \leq 10)$ donnée par les bornes de la boucle.

Pour certaines équations, l'existence d'une solution ne peut pas être connue avec certitude - du moins dans un temps raisonnable, mais une réponse partielle peut être donnée par un algorithme rapide. Le traitement des tableaux multi-dimensionnels est plus compliqué.

2. Les algorithmes qui manipulent une description plus ou moins exacte de l'ensemble d'indices des éléments d'un même tableau référencés par une ligne de code, boucle `DO`, sous-programme, ... En effet, afin de décider si deux fragments de code peuvent être exécutés en parallèle, ces algorithmes ont besoin d'informations *sommaires* sur les indices des éléments de chaque tableau pouvant être référencés quelque part dans le fragment. Les descriptions des différentes parties d'un même tableau référencées par les diverses instructions composant le fragment de code sont alors réunies en une seule description sommaire. Dans le cas d'un fragment contenant des appels de sous-programme, le descripteur des indices exprimé en termes des paramètres formels du sous-programme, doit être "traduit" vers un descripteur exprimé en termes des paramètres effectifs. Ceci afin de propager les informations sommaires vers les sites d'appel.

   Certaines de ces approches emploient, pour des raisons d'efficacité ou d'économie d'espace mémoire, une description, qui n'est pas toujours exacte. A l'inexactitude intrinsèque de l'analyse statique s'ajoute alors l'inexactitude de la description de l'ensemble d'éléments. Ainsi la description des informations *"may"* peut être une sur-approximation à cause :

   - des éléments référencés uniquement au cours de chemins non exécutables, impossibles à éliminer,
   - de la nature approximative de la description d'un ensemble arbitraire d'éléments ou
   - d'autres approximations introduites lors des opérations sur les descripteurs telles que le calcul des descriptions sommaires ou la traduction des descriptions vers les sites d'appel

   Ces approches comprennent :

   - celle de [CK87, Hav94] permettant la description des parties de tableaux de certaines formes appelées *"regular sections"* : par exemple une ligne, une colonne, une diagonale, des "rayures", mais pas une région triangulaire ;
   - celle de [TIF86], qui consiste à calculer les inégalités linéaires exprimant les bornes de polyèdres, appelées *régions*, à l'intérieur des tableaux. Elle ne peut qu'approximer les polyèdres non convexes. Son utilisation de la programmation linéaire implique un calcul de complexité exponentielle dans le pire des cas, mais qui semble être polynomiale en pratique [Cre96].

   Par contre, [LY88] fournit une description exacte appelée *"atom image"* pour chaque point d'accès de chaque tableau et les informations sommaires sont juste des listes (arbitrairement longues) d'"atom images".

4.4 Les outils disponibles

La réalisation de tout outil d'analyse statique avec une couverture totale de Fortran77 (i.e. analysant tout le langage et non un sous-ensemble) est difficile, à cause notamment

- des `GOTO` (voir Section 4.3.3),
- des alias statiques (voir Section 4.3.5) et
- de l'importance des accès aux tableaux (voir Section 4.3.6).

ainsi que des autres spécificités du langage évoquées dans la Section 3.1 du Chapitre 3.

Il existe plusieurs outils d'analyse statique du Fortran77 réalisés aux années 70 et 80 et qui, à cause de leur âge, souffrent d'un manque d'ouverture (les informations qu'ils recensent ou calculent ne sont pas toutes disponibles pour l'utilisateur), et de plus, n'intègrent pas les techniques les plus récentes. Les outils plus modernes sont plus généraux, mais à cause de l'essor d'autres langages de programmation, ils offrent souvent peu ou pas d'aide à l'analyse de code Fortran77.

Il existe de nombreux outils effectuant une analyse statique minimale de Fortran, sans analyse de flot de données. Ces outils se limitent à la génération de références croisées, ainsi qu'à la détection d'anomalies de programmation pouvant indiquer des erreurs, vérifiant souvent le respect de la norme *ANSI* ou d'autres règles de bonne programmation. Nous avons décrit l'outil *Ftnchek* à titre d'exemple.

En effet, à ses débuts l'analyse de flot de données a surtout été mise en œuvre au sein des compilateurs optimisés. En ce qui concerne l'exécution symbolique de code Fortran, plusieurs prototypes d'outils ont été réalisés par des universitaires à cette époque [Cow88] mais il semblerait que leur mise en œuvre de façon moins expérimentale a posé des problèmes, car nous n'en avons pas trouvé de plus modernes !

Il existe aussi des générateurs d'outils, plus récents, permettant en principe de réaliser soi-même une analyse statique plus poussée mais qui laissent souvent à l'utilisateur la définition de la grammaire de Fortran (mais ce n'est pas le cas pour *Refine/Fortran*) et demandent toujours un certain effort de programmation.

Les outils plus récents réalisés par des chercheurs universitaires en rétro-ingénierie ou ré-ingénierie et disponibles dans le domaine public s'appliquent à d'autres langages que Fortran, par exemple : [HM93] pour le langage C, [HR92] pour un sous-ensemble de Pascal et [MWT94] pour C, C++ et COBOL.

Finalement, il y a les outils réalisés par les chercheurs universitaires en parallélisation de codes Fortran. Ces outils n'ont pas été réalisés dans le but d'effectuer une analyse statique générale, mais étant récents, ils sont ouverts, puissants et intègrent les meilleures techniques, en particulier en ce qui concerne le traitement des éléments de tableaux. En revanche, ils n'effectuent qu'une exécution symbolique limitée et adaptée aux besoins de la parallélisation (l'*analyse symbolique* [Hav94]).

Par ailleurs, les outils *FORESYS* (plus précisément, l'outil *FOREST* apparenté à Foresys) et *PIPS* proposent des fonctions de *restructuration* de code afin de le simplifier : détection de structures `while` ou `repeat/until` réalisées à l'aide des `GOTO`, élimination de tests inutiles,... Il s'agit de transformation plutôt que d'analyse statique, et ces fonctionnalités seront discutées dans la Section 6.2 du Chapitre 6.

Parmi les outils décrits par la suite, nous avons pu essayer nous-mêmes *Ftnchek*, *FORESYS* pour ce qui concerne la fonction d'auto-documentation, et *PIPS*. À titre d'exemple, nous présentons dans les Annexes A et C quelques résultats de l'analyse par *Ftnchek* et *FORESYS* du sous-programme `PRCHEN` présenté dans l'Annexe G. Plusieurs exemples des résultats des analyses de *PIPS* sont présentés dans le Chapitre 5. Le but est surtout d'illustrer la forme des résultats des différents outils. Pour illustrer toutes les petites différences entre les outils sur le plan de la précision de l'analyse, il aurait fallu élaborer une procédure de test spéciale, mais ceci est hors de notre propos.

4.4.1 *Ftnchek*

Outil du domaine public, *Ftnchek* [5] se limite au recoupement d'informations provenant de l'analyse syntaxique du code. Il génère une documentation des variables des sous-programmes (simplement classées comme paramètres, variables locales ou variables de bloc `COMMON`) et détecte des anomalies de dimensionnement ou de typage et autres fautes potentielles de programmation mais sans prendre en compte le flot de contrôle. Voir l'Annexe A pour plus d'informations, ainsi qu'un exemple des résultats.

4.4.2 *Hindsight*

Il se distingue des autres outils ressemblant à *Ftnchek* par sa génération, sous forme graphique, des graphes des appels et, surtout, des graphes de contrôle des sous-programmes. Voir Annexe B.

4.4.3 *FORESYS*

FORESYS [6] effectue l'analyse inter-procédurale de flot de données. Il ajoute alors aux fonctionnalités semblables à celles de *Ftnchek*, la génération de documentation des sous-programmes indiquant quels

5. Développé par Robert MONIOT, *Ftnchek* est disponible, avec d'autres outils, à http://www.fortran.com/fortran/tools.html

6. *FORESYS* est une marque déposée par CONNEXITE, et commercialisé par SIMULOG, voir http://www.simulog.fr/FR/html/prods/foresys.html

sont les paramètres et variables déclarées dans des blocs `COMMON` qui sont certainement utilisés, ceux qui sont peut-être utilisés, ceux certainement modifiés et ceux peut-être modifiés. Cependant, il ne précise pas si l'utilisation peut intervenir avant la modification (et ainsi ne distingue pas les paramètres d'entrée), ni les éléments référencés en cas de tableaux.

FORESYS propose la parallélisation automatique du code source. Pour le calcul des dépendances, il doit analyser les éléments des tableaux effectivement lus et modifiés, mais ces informations ne sont pas disponibles à l'utilisateur. L'évaluation partielle est proposée aussi : à partir des informations fournies par l'utilisateur sur les valeurs de certaines variables, il cherche à éliminer du code des branches ou des calculs qui sont inutiles pour ces valeurs, sans toutefois fournir à l'utilisateur d'autres résultats que le code transformé.

Voir l'Annexe C pour plus d'informations, ainsi qu'un exemple des résultats.

4.4.4 *Refine/Fortran*

Refine/Fortran [7] est un générateur d'outils d'analyse statique ou de transformation de codes Fortran77 conçu pour l'aide à la rétro-ingénierie/ré-ingénierie. L'analyse syntaxique est pré-codée et les références croisées du genre de celles fournis par *Ftnchek* sont fournies. Il n'y a pas de traitement spécifique des éléments de tableaux. Un "toolkit" est proposé pour la manipulation et la visualisation des résultats de l'analyse syntaxique et l'utilisateur peut programmer d'autres fonctionnalités en *Lisp*. Voir Annexe D pour plus d'informations.

4.4.5 *PIPS*

Développé à l'École des Mines de Paris [8] [IJT91], *PIPS* est un outil du domaine public de parallélisation de programmes Fortran77. Il effectue l'analyse de flot de données inter-procédurale et offre de nombreuses fonctionnalités d'analyse statique. Il prend en compte l'ordre d'exécution des modifications et utilisations de valeurs d'une même variable et en cas de tableaux, caractérise par une *région* les éléments référencés, parfois de façon précise et sinon par une sur-approximation. *PIPS* prend en compte le contexte des appels en supposant le code analysé "complet" (voir la discussion de la Section 3.8 du Chapitre 3). En conclusion, parmi les outils d'analyse statique de Fortran que nous avons étudiés, *PIPS* est le plus puissant et le plus avancé. Nous le décrivons avec plus de détails dans le chapitre suivant.

7. *Refine/Fortran* est une marque déposée de Reasoning Systems Inc., voir http://www.reasoning.com
8. voir http://www.cri.ensmp.fr/pips/index.html

Chapitre 5

Régions de *PIPS* et adéquation à nos besoins

Résumé

PIPS étant ce que nous avons trouvé de plus puissant et de plus avancé parmi les outils d'analyse statique de Fortran, nous décrivons plus en détail, dans la première section de ce chapitre, sa représentation de parties de tableaux.

Dans la deuxième section du chapitre, nous étudions l'adéquation à la rétro-ingénierie de cette représentation.

5.1 Les régions de *PIPS*

Les régions de tableau de *PIPS* caractérisent l'ensemble des indices des éléments d'un même tableau référencés par une ou plusieurs instructions d'un programme [Cre96]. *PIPS* calcule les bornes linéaires, paramétrées par des variables scalaires entières du programme, de ces régions.

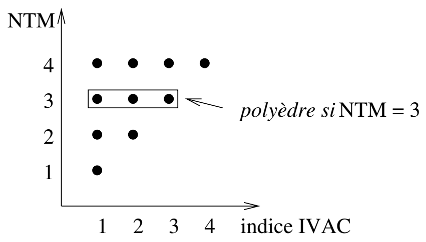
Une région $\mathcal{R}$ de *PIPS* pour un tableau de dimension d est donc une fonction de l'ensemble Σ des *stores* (valeurs courantes des variables) vers tous les polyèdres d'entiers (les indices des éléments référencés), qui appartiennent à $\mathcal{P}(\mathbb{Z}^d)$:

$$\mathcal{R} : \Sigma \rightarrow \mathcal{P}(\mathbb{Z}^d)$$

Par exemple, la région caractérisant l'ensemble des éléments d'`IVAC` référencés par la boucle `DO` suivante

```
      DO 9 IL = 1, NTM
         ZIR = IVAC(IL)
         ...
    9 CONTINUE
```

est la fonction qui, pour une valeur (NTM) donnée de `NTM`, rend l'intervalle $[1, NTM]$. Notons que cet intervalle est vide si NTM est inférieure à 1.



5.1.1 Sur-approximations et sous-approximations

La meilleure sur-approximation polyédrique de l'ensemble d'éléments référencés est le plus petit polyèdre contenant (au moins) ces éléments. Pour le calcul des *"summary regions"* (expliquée dans la Section 5.1.2 ci-dessous), *PIPS* a besoin de réunir plusieurs régions d'un même tableau en une seule. Il emploie alors l'opération de calcul de l'enveloppe convexe.

Notons $\sqsubseteq$, $\sqcap$ et $\sqcup$ l'inclusion, l'intersection et l'union, respectivement, sur les ensembles, généralisées ("lifted") aux fonctions. Les régions avec $\sqsubseteq$ comme relation d'ordre forment un treillis complet (voir Section 4.3.3 du Chapitre 4) :

$$< \mathcal{R} : \Sigma \underset{convex}{\longrightarrow} \mathcal{P}(\mathbb{Z}^d), \sqsubseteq, \lambda\sigma.\emptyset, \lambda\sigma.\mathbb{Z}^d, \text{enveloppe convexe}, \sqcap > .$$

La sémantique exacte correspondante est le treillis complet des fonctions vers les ensembles arbitraires d'entiers :

$$< \mathcal{R} : \Sigma \rightarrow \mathcal{P}(\mathbb{Z}^d), \sqsubseteq, \lambda\sigma.\emptyset, \lambda\sigma.\mathbb{Z}^d, \sqcup, \sqcap > .$$

Les deux treillis peuvent être comparés (voir Section 4.3.3 du Chapitre 4), afin de prouver que le polyèdre calculé est bien la meilleure sur-approximation.

Par exemple, l'ensemble des éléments de `NRLO` modifiés par la boucle `DO` suivante

```
DO I = 1, NL
   K1 = NRLO(I)
   IF (K1.GT.0) THEN
     IF (NDOB(K1).LE.0) NRLO(I) = -1
   ELSE
     NRLO(I) = 0
   ENDIF
END DO
```

est sur-approximée par la région qui, pour une valeur (NL) donnée de `NL`, rend l'intervalle $[1, NL]$.

En ce qui concerne les sous-approximations, il faudrait trouver le plus grand polyèdre contenant juste des éléments forcément référencés, mais celui-ci n'est pas bien défini. En effet, pour un ensemble arbitraire d'éléments, il est possible de définir plusieurs polyèdres inclus dans l'ensemble et qui regroupent le même nombre maximal d'éléments. Ils sont donc incomparables et il n'y a pas de treillis complet dans le cas des sous-approximations.

Par exemple, l'instruction suivante fait référence à deux éléments du tableau `NCHA` qui ne sont pas nécessairement contigus.

```
IF (NCHA(KI).GT.NCHA(KJ)) THEN
```

L'ensemble composé des indices `KI` et `KJ`, est sous-approximé par le polyèdre convexe composé de la seule valeur de `KI`, mais aussi par celui composé de la seule valeur de `KJ`.

5.1.2 Régions EXACT et MAY

PIPS ne calcule donc pas une sous-approximation des régions, mais la description exacte (appelée région EXACT) des éléments référencés lorsque c'est possible et sinon, une sur-approximation (appelée région MAY). En l'absence de description exacte, *PIPS* emploie comme sous-approximation la fonction rendant toujours le polyèdre vide. La sur-approximation par défaut est le tableau entier. Dans le cas où un tableau est paramètre formel d'un sous-programme dont la dimension est déclarée par un astérisque, la borne inférieure des indices est 1 et la borne supérieure n'est pas précisée. La sur-approximation par défaut est alors une région MAY sans borne supérieure et avec 1 comme borne inférieure.

Rappelons qu'en analyse de flot de données classique, on appelle *"may"* l'union des résultats pour tous les chemins d'exécution et *"must"* leur intersection (voir Section 4.2 du Chapitre 4). Si *PIPS* peut exprimer un prédicat sur les variables appartenant à une branche d'un test (*prédicat de branche*, voir Section 4.3.2 du Chapitre 4) comme une expression linéaire sur des scalaires entiers, alors il l'intègre dans la région de cette branche. Le prédicat vérifié par la branche constitue une contrainte sur le store mais pas nécessairement sur les valeurs des indices ϕ_1, ϕ_2, ... des éléments du tableau. En effet, il y a deux cas de figure :

- les variables contraintes par le prédicat étaient déjà des paramètres de la région et une contrainte supplémentaire peut (si elle n'est pas redondante avec les autres) constituer une contrainte supplémentaire sur les valeurs des indices prises par les éléments de tableaux, ou
- le prédicat de branche ajoute de nouvelles variables et donc de nouvelles dimensions au polyèdre décrit par la région, mais n'a aucun effet sur les valeurs possibles des indices : la *projection* du nouveau polyèdre sur les anciens axes donne l'ancien polyèdre.

La région devient alors une fonction sur le store qui a comme résultat le polyèdre vide en cas de non satisfaction du prédicat de branche et le polyèdre (non vide) sinon. Grâce à cette possibilité d'intégration du prédicat de branche dans la description de la région, les régions EXACT fournissent des descriptions des éléments référencés qui sont plus précises que la solution *"must"* classique. Par ailleurs, *PIPS* peut effectuer un test de faisabilité des solutions du polyèdre convexe, c'est-à-dire, tester s'il est toujours vide. Ce test donne une sous-approximation rationnelle "conservative" (prudente) des éventuelles solutions entiers [Cre96].

Par exemple, les éléments de `NXUT_P` utilisés par la boucle `DO` du fragment de code suivant

```
IF (NXU.GT.0) THEN
  DO I = 1, NBY
    NX5(I_NXUT-1+I) = NXUT_P(I)
  ENDDO
ELSE
ENDIF
```

sont caractérisés par la région EXACT qui, pour une valeur donnée (NBY) de `NBY`, rend l'intervalle $[1, NBY]$ si la valeur de `NXU` est supérieure à zéro et le polyèdre vide sinon.

Toutefois, la région pour le fragment entier est MAY car toute région d'instruction `IF` est toujours MAY.

Une instruction comportant une seule modification (resp. utilisation) possible d'un tableau donné, donne toujours lieu à une région *PIPS* EXACT pour cette instruction. Mais une seule instruction comportant deux modifications (resp. utilisations) possibles à des éléments d'un même tableau peut donner lieu à une région MAY pour l'instruction. En effet, c'est la propagation et la combinaison par *PIPS* des régions EXACT pour des références individuelles qui résultent souvent en des régions MAY, comme nous allons expliquer dans la suite de cette sous-section. Comme chaque opération sur les régions est susceptible d'entraîner une perte de précision, *PIPS* teste l'exactitude du résultat et crée une région MAY en cas d'échec du test. Certains de ces tests ne sont pas exacts [Cre96], mais sont, comme le test de faisabilité, des sous-approximations rationnelles "conservatives" (prudentes) des solutions entières recherchées.

Perte de précision dans le calcul des "summary regions"

PIPS représente un fragment de code par un arbre syntaxique, exceptés les fragments de code non structurés qui sont représentés par un graphe de contrôle attaché à cet arbre. *PIPS* calcule une région des références (dans un mode donné : READ, WRITE, IN ou OUT, voir les Sections 5.1.4 et 5.1.5 ci-dessous) pour chaque tableau pour chaque instruction et puis propage ces régions récursivement vers la racine de l'arbre syntaxique du sous-programme. À chaque pas de propagation, les différentes régions (pour un mode donné) d'un même tableau sont unies en une région résumée (pour ce mode) pour le nœud de l'arbre syntaxique. C'est la *"summary region"*.

C'est ici que l'opération d'enveloppe convexe employée pour unir deux polyèdres convexes en un autre polyèdre convexe peut introduire de l'imprécision. Dans le cas de tableaux d'une dimension, le résultat de l'union de deux régions EXACT est MAY si elles peuvent ne pas être contiguës ou superposées.

Les sous-programmes sont analysés dans l'ordre, en commençant par ceux qui constituent les "feuilles" de l'arbre des appels et en remontant vers la racine de l'arbre (analyse "bottom-up" ou ascendante). Une fois l'analyse de tous les sous-programmes d'une même profondeur dans l'arbre des appels terminée, les régions résumées pour chacun de ces sous-programmes sont propagées vers les instructions d'appel. Les régions résumées du sous-programme appelé peuvent ainsi être traitées comme des références aux tableaux par l'instruction d'appel, comme expliqué dans le Chapitre 4 (Section 4.3.4).

Perte de précision dans la propagation au début du fragment de code

Une région caractérise tous les accès (dans un mode donné) à un même tableau contenus dans un fragment de code. Ce fragment de code fait partie d'un sous-programme (même s'il contient des appels d'autres sous-programmes) avec son propre *espace de nommage* composé des variables locales, des paramètres formels et des variables déclarées dans des blocs `COMMON`. La région est paramétrée par le *store*, c'est-à-dire par les valeurs des variables de l'espace de nommage à un moment donné de l'exécution. Les valeurs des variables peuvent varier pendant l'exécution du fragment de code. Afin de combiner des régions (caractérisant des références différentes à un même tableau dans le fragment de code), il faut qu'elles soient toutes exprimées en fonction du même store. Dans *PIPS*, les régions calculées pour un certain fragment de code sont donc des fonctions du store **au début de l'exécution de ce fragment**. Il faut alors pouvoir exprimer les valeurs des variables au moment de l'accès au tableau en fonction de leurs valeurs au début de l'exécution du fragment. Si la valeur de la variable au point d'accès n'est pas une expression linéaire de variables scalaires, *PIPS* ne peut pas employer cette variable comme paramètre de la région, qui devient alors une sur-approximation.

Par exemple, les éléments de `NRTJ1` utilisés dans l'appel de `PREFIX` du fragment de `BESTI1` (légèrement modifié, voir Section 5.2.2) suivant sont caractérisés par une région EXACT dont la borne supérieure est une fonction de la variable `NT`.

```
...
NT = F_NT
...
IF (NT.LE.0) THEN
   NT = 1
ELSE
ENDIF
...
CALL PREFIX(CARL,..., NRTJ1,...)
```

mais comme il y a deux valeurs possibles pour `NT` au site d'appel (`F_NT` ou 1), ces éléments sont caractérisés au point d'entrée de `BESTI1` par une région MAY.

Dans le cas du calcul des “summary regions” pour un sous-programme, *PIPS* ré-exprime toutes les régions en fonction du store au point d'entrée du sous-programme. Les variables locales du sous-programme sont alors éliminées des paramètres des régions.

Perte de précision dans la traduction des régions

Afin de propager une région pour un sous-programme vers l'appel de celui-ci depuis un autre sous-programme, il faut traduire la région de l'espace de nommage du sous-programme appelé vers celui de l'appelant. Il faut traduire toutes les variables paramètres de la région. Toutefois, comme expliqué dans la Section 4.3.5, des alias dynamiques de tableaux peuvent ne pas être dimensionnés de la même façon. Parfois il n'est pas possible de traduire précisément les bornes linéaires des valeurs d'indices d'un tableau dimensionné d'une façon en bornes linéaires du même tableau dimensionné de façon différente. Une région EXACT est alors traduite en région MAY. Un phénomène similaire peut se présenter en cas de traduction d'un tableau dont le type déclaré varie entre les sous-programmes.

Perte de précision dans l'analyse de code non structuré et certaines boucles `DO`

Certaines structures de contrôle du code mal structuré, réalisées par exemple à l'aide de `GOTO`, donnent lieu à des régions MAY. C'est le cas des `GOTO` conditionnels à l'intérieur de boucles `DO`. Par ailleurs, *PIPS* n'effectue pas de calcul de point fixe pour des boucles réalisées à l'aide de `GOTO`, alors les régions pour ces boucles sont sur-approximées par le tableau entier.

En ce qui concerne les boucles `DO`, *PIPS* peut calculer une région EXACT pour le nid de boucles entier (sans besoin d'itérer vers un point fixe), au cas où

- le corps de boucles ne contient pas de `GOTO`,
- il arrive à exprimer les bornes de boucle comme des expressions linéaires des variables scalaires entières

– et l'incrément vaut 1 ou -1.

La région calculée intègre implicitement la condition que la valeur absolue de la limite supérieure de la boucle est strictement plus grande que celle de la limite inférieure (car si la limite supérieure n'est pas plus grande que la limite inférieure, la boucle est simplement "sautée" sans être exécutée).

Conditions du calcul des régions EXACT

En conclusion, étant donné un fragment de code pour lequel la région d'un même tableau référencée (dans un mode donné) par chaque composant est EXACT, *PIPS* devrait pouvoir réunir ces régions en une région cumulée EXACT pour le fragment de code entier si :

– le fragment ne contient pas de `GOTO`,

– les bornes et l'incrément des boucles `DO` satisfont les conditions énoncées ci-dessus,

– en cas d'instruction conditionnelle, *PIPS* arrive à exprimer la condition comme une expression linéaire de scalaires entiers,

– en cas d'appel à des sous-programmes, *PIPS* peut traduire la région EXACT d'un paramètre formel du sous-programme en région EXACT sur le paramètre effectif,

– *PIPS* peut ré-exprimer (sans perte de précision) la région pour chaque composant du fragment de code en fonction des valeurs **au début de l'exécution du fragment de code** de variables scalaires et

– l'union des régions (polyèdres convexes) est un polyèdre convexe.

Voir l'exemple de la Section 5.1.4 ci-dessous.

Types d'imprécision

Nous pouvons définir trois types d'imprécision dans la caractérisation d'un ensemble d'éléments par une région MAY :

1. l'imprécision du *"may"* : l'ensemble des éléments référencés dépend du chemin d'exécution suivi (et peut inclure des éléments référencés uniquement dans des chemins effectivement non exécutables) ;

2. la sur-approximation d'un polyèdre non convexe par un polyèdre convexe ;

3. la sur-approximation des bornes.

PIPS ne distingue pas ces trois types d'imprécision dans les régions MAY.

5.1.3 Ajout de la précondition aux régions

Nous avons décrit, au début de cette section, l'ajout du prédicat de branche à la région d'une instruction conditionnelle. Le prédicat de branche fait partie des *préconditions* que *PIPS* calcule à chaque nœud de l'arbre syntaxique du code. Les préconditions calculées sont toutefois limitées aux contraintes linéaires sur des variables scalaires entières.

Une précondition est composée

– de prédicats de branche, qui peuvent être des égalités ou inégalités, et

– des égalités résultant d'affectations de valeurs aux variables.

Certains éléments d'une précondition déterminent les bornes de la région : comme nous l'avons expliqué au début de cette section, le prédicat de branche peut contraindre la valeur d'une variable paramètre de la région. De plus, certaines affectations de paramètres sont prises en compte dans la propagation de la région vers le début de chaque bloc de code, où l'inverse de la précondition est employée pour exprimer la région en fonction du store au début du bloc.

Les autres éléments de la précondition ajoutent des dimensions au polyèdre, et ainsi des contraintes sur le store, mais ne modifient pas les valeurs possibles des **indices du tableau** au point de référence. En d'autres termes, la projection du polyèdre sur les anciens axes n'est pas modifiée par ces derniers éléments de la précondition.

Par exemple, la précondition au point de lecture d'`IVAC` dans le fragment de code suivant extrait du sous-programme `CAGAIN` (voir aussi l'exemple de la Section 5.1.4 ci-dessous) :

```
   DO 9 IL = 1, NTM
      ZIR = IVAC(IL)
      ...
 9 CONTINUE
```

comprend

- les contraintes suivantes sur les valeurs possibles des indices d'`IVAC` au point de lecture :
  - le fait que la valeur de l'indice doit être supérieure à 0, provenant de la borne inférieure de la boucle `DO` et de la déclaration (par un astérisque) de la dimension d'`IVAC`, et
  - l'inégalité $IL \leq NTM$ provenant de la borne supérieure de la boucle `DO`.
- ainsi que de nombreuses contraintes sur le store qui ne contraignent pas les valeurs des indices d'`IVAC`, telles que
  - le prédicat de branche $1 \leq IBT$ et
  - l'égalité $NXF = 0$ provenant d'une affectation.

Toutefois, *PIPS* ajoute systématiquement toutes les préconditions associées à chaque région pour un fragment de code donné.

Supposons maintenant que nous voulions tester l'égalité de deux régions d'un même tableau. Le résultat de cette comparaison doit être indépendant des contraintes sur des variables qui n'ont aucun effet sur les valeurs possibles des indices du tableau. En d'autres termes, il faut comparer non pas les régions avec toute leur précondition, mais la projection sur les axes des indices et paramètres des deux régions. *PIPS* propose (sous forme de fonctions *C*) les fonctionnalités permettant de réaliser cette comparaison.

5.1.4 Régions READ et WRITE : exemple de sortie *PIPS*

Les régions caractérisant les éléments d'un tableau *utilisés* par un fragment de code s'appellent les régions READ, et les régions caractérisant les éléments *modifiés*, les régions WRITE.

Prenons comme exemple le fragment de `CAGAIN` présenté dans la Figure 5.1. Chaque instruction est annotée par les régions READ et WRITE calculées pour certains tableaux.

Une région est délimitée par $<$ et $>$. Elle contient d'abord le nom du tableau avec, entre parenthèses, la valeur de l'indice de chaque dimension représentée par `PHI1`, `PHI2`, ... Ensuite il y a le *mode* : `R` pour READ ou `W` pour WRITE. Après, vient la précision (`MAY` ou `EXACT`). Finalement, entre accolades, il y a la liste des contraintes sur les valeurs des indices et sur les valeurs des autres variables.

Notons :

- Dans cet exemple, les préconditions ajoutent de nombreuses contraintes qui ne changent pas les valeurs possibles des indices.
- De la région EXACT caractérisant l'accès inconditionnel à `IVAC` indexé par la variable de boucle dans l'instruction numéro 0004 , *PIPS* calcule une région EXACT pour la boucle `DO` de l'instruction 0003 : c'est le premier exemple de la Section 5.1.1.
- De deux régions EXACT contiguës caractérisant les accès à `LHAC` aux instructions 0007 et 0008, *PIPS* calcule une région EXACT pour la boucle `DO` de l'instruction 0003.
- Le fait de calculer systématiquement une région cumulée entraîne une perte de précision dans le cas des deux accès à `NCHA` à l'instruction 0011. La région cumulée pour cette instruction est MAY car les deux éléments référencés ne sont pas forcément contigus (c'est le deuxième exemple de la Section 5.1.1). Les régions engendrées par ces accès pour les deux boucles `DO` imbriquées aux instructions 0009 et 0010 sont donc MAY aussi. Cependant, chaque accès seul engendre la même région EXACT pour la boucle extérieure.

```
      SUBROUTINE CAGAIN (NTM, INTM, LHAC, NCHA, EHA, IVAC, M_GAIN)
   ...

C  ...
C  <IVAC(PHI1)-R-EXACT-{1<=PHI1, PHI1<=NTM, I_ISUI==3NSO+18,
C    I_LCHBG==15, MNL==7560, NNL==7560, NXF==0, NXO==0, 1<=IBT,
C    IBT<=2}>
C  <LHAC(PHI1)-R-EXACT-{1<=PHI1, PHI1<=1+NTM, I_ISUI==3NSO+18,
C    I_LCHBG==15, MNL==7560, NNL==7560, NXF==0, NXO==0, 1<=IBT,
C    IBT<=2, 1<=NTM}>
C  <NCHA(PHI1)-R-MAY-{1<=PHI1, I_ISUI==3NSO+18, I_LCHBG==15,
C    MNL==7560, NNL==7560, NXF==0, NXO==0, 1<=IBT, IBT<=2, 1<=NTM}>

C -----------------------------------------------------------
C       CALCUL DES DIFFERENTS TERMES DE LA MATRICE HTH
C -----------------------------------------------------------

      DO 9 IL = 1, NTM                                          0003

C  <IVAC(PHI1)-R-EXACT-{PHI1==IL, 1<=PHI1, I_ISUI==3NSO+18,
C    I_LCHBG==15, MNL==7560, NNL==7560, NXF==0, NXO==0, 1<=IBT,
C    IBT<=2, IL<=NTM}>

         ZIR = IVAC(IL)                                         0004

         IF (ZIR.LT.1.) ZIR = 1.                                0005

C  <LHAC(PHI1)-R-EXACT-{PHI1==IL, 1<=PHI1, I_ISUI==3NSO+18,
C    I_LCHBG==15, MNL==7560, NNL==7560, NXF==0, NXO==0, 1<=IBT,
C    IBT<=2, IL<=NTM}>

         IDEB = LHAC(IL)                                        0007

C  <LHAC(PHI1)-R-EXACT-{PHI1==IL+1, I_ISUI==3NSO+18, I_LCHBG==15,
C    MNL==7560, NNL==7560, NXF==0, NXO==0, 1<=IBT, IBT<=2, 1<=IL,
C    IL<=NTM}>

         IFIN = LHAC(IL+1)-1                                    0008

C  ...
C  <NCHA(PHI1)-R-MAY-{IDEB<=PHI1, 1<=PHI1, PHI1<=IFIN,
C    I_ISUI==3NSO+18, I_LCHBG==15, MNL==7560, NNL==7560, NXF==0,
C    NXO==0, 1<=IBT, IBT<=2, 1<=IL, IL<=NTM}>

         DO 10 KI = IDEB, IFIN                                  0009

C  ...
C  <NCHA(PHI1)-R-MAY-{IDEB<=PHI1, 1<=PHI1, PHI1<=IFIN,
C    I_ISUI==3NSO+18, I_LCHBG==15, MNL==7560, NNL==7560, NXF==0,
C    NXO==0, 1<=IBT, IBT<=2, IDEB<=KI, KI<=IFIN, 1<=IL, IL<=NTM}>

            DO 7 KJ = IDEB, IFIN                                0010

C  <NCHA(PHI1)-R-MAY-{IDEB<=PHI1, KI+KJ<=PHI1+IFIN, 1<=PHI1,
C    PHI1+IDEB<=KI+KJ, PHI1<=IFIN, I_ISUI==3NSO+18, I_LCHBG==15,
C    MNL==7560, NNL==7560, NXF==0, NXO==0, 1<=IBT, IBT<=2, IDEB<=KI,
C    IDEB<=KJ, KI<=IFIN, KJ<=IFIN, 1<=IL, IL<=NTM}>

               IF (NCHA(KI).GT.NCHA(KJ)) THEN                   0011
         ...
9        CONTINUE                                               0019
```

FIG. 5.1: Fragment de `CAGAIN`, annoté par *PIPS*

- La région pour `NCHA` à l'instruction 0009 a des bornes exactes pour `PHI1` égales aux valeurs des variables locales scalaires `IDEB` et `IFIN`, mais il s'agit d'adressage indirect. Les valeurs de ces variables sont copiées de deux éléments du tableau `LHAC`. *PIPS* ne peut donc pas les exprimer en fonction de variables scalaires, alors la seule borne des valeurs de l'indice de `NCHA` à la instruction 0003 est celle venant du dimensionnement (par un astérisque) du tableau.

5.1.5 Régions IN et OUT

Les régions IN de *PIPS* caractérisent les ensembles d'éléments d'un tableau dont la valeur au début de l'exécution du fragment de code peut être *utilisée* par le fragment (i.e. avant toute modification éventuelle de l'élément par le fragment). Une région IN d'un fragment de code est calculée à partir de la différence pour chaque instruction entre la région READ de l'instruction et la région WRITE des instructions exécutées avant celle-ci dans le fragment.

Par exemple, dans le fragment de `PRXPLO` suivant, *PIPS* ne peut pas caractériser précisément les éléments d'`IPCO` utilisés par `PRCHEN` et ils sont alors sur-approximés par le tableau entier. Mais les `NS` premiers éléments d'`IPCO` sont modifiés par ce fragment de code avant l'appel de `PRCHEN`. La région IN calculée pour `IPCO` est donc celle qui, pour une valeur donnée (NS) de `NS`, rend le polyèdre composé de tous les entiers supérieurs à NS et à 0.

```
do I = 1, NS
   IPCO(I) = 0
   IPCE(I) = 0
end do
do I = 1, NL
   N1 = NRLO(I)
   N2 = NRLE(I)
   CALL PRCHEN (I, N1, N2, IPCO, ICCO, NRLE)
   CALL PRCHEN (I, N2, N1, IPCE, ICCE, NRLO)
end do
```

Notons que le calcul de la sur-approximation d'une région IN repose sur la **sous**-approximation des régions WRITE si *PIPS* n'a pas pu calculer la région EXACT. C'est la raison pour laquelle la sous-approximation est définie (voir Section 5.1.1).

Les régions OUT d'un fragment caractérisent les ensembles d'éléments modifiés et dont la nouvelle valeur est utilisée dans la suite de l'exécution (après la sortie du fragment).

Les régions OUT sont les seules calculées par une analyse "descendante", à partir de la racine de l'arbre syntaxique du sous-programme qui est la racine de l'arbre des appels. Par conséquent, lors du calcul d'une région OUT pour un sous-programme p et tableau t une borne supérieure résultant de la déclaration de dimensionnement du tableau $t0$ du sous-programme $p0$ appelant (directement ou indirectement) p, où $t0$ instancie t, peut être ajoutée parmi les contraintes. Cette borne n'est pas propagée lors du calcul des éventuelles régions IN de t pour p puisqu'elles sont calculées de façon "ascendante". La comparaison des régions IN et OUT de t pour p est alors compromise: aucune région IN ne sera trouvée identique à une région OUT.

5.1.6 La précision de l'analyse de *PIPS* dans la pratique

L'analyse par *PIPS* des codes que nous avons étudiés (avant restructuration, voir ci-dessous) donne lieu à très peu de régions EXACT. Évidemment, ceci est du en partie aux pertes de précision induites par le calcul des régions IN et OUT et des "summary regions" et décrites dans la Section 5.1.2.

Analyse par *PIPS* des structures de contrôle des codes analysés

Une autre raison est les structures de contrôle employées dans ces codes. En effet, de nombreux sous-programmes de *COURCIRC* emploient une boucle `DO` avec échappement du sous-programme (réalisé par un `GOTO`) au milieu de la boucle, en cas d'erreur (voir l'exemple présenté dans la Figure 6.1 et expliqué dans la Section 6.1 du Chapitre 6). *PIPS* ne peut pas trouver de bornes exactes aux régions pour ce type de boucles. *ESTIME* contient aussi des `GOTO` conditionnels au sein des boucles `DO`, ainsi que de nombreuses boucles réalisées à l'aide de `GOTO`.

Toutefois la précision de l'analyse de *PIPS* est améliorée par des transformations du code proposées dans le Chapitre 6, à savoir : l'élimination des tests d'erreur et la restructuration automatique du code.

```
C  <ICCO(PHI1)-R-MAY-{1<=PHI1, 1<=NSO}>
C  <IPCO(PHI1)-R-EXACT-{1<=PHI1, PHI1<=NSO}>

      DO 3 I=1,NSO
         ...
C  <IPCO(PHI1)-R-EXACT-{PHI1==I, 1<=PHI1, ND==0, I<=NSO}>

         IC = IPCO(I)

    4    IF (IC.LE.0) GOTO 3
            ...
C  <ICCO(PHI1)-R-EXACT-{PHI1==IC, 1<=PHI1, 1<=I, I<=NSO}>

    5       IC = ICCO(IC)
            GOTO 4
    3 CONTINUE
```

FIG. 5.2: La lecture par `CASDEG` de toutes les listes d'une table partitionneuse

En effet la restructuration automatique peut remplacer certains `GOTO` au sein des boucles par des `IF`. Par contre, peu de boucles réalisées à l'aide de `GOTO` sont automatiquement transformées en boucles `while`[1]. En tout cas, *PIPS* n'est pas capable d'analyser précisément les boucles `while` qui ne peuvent pas être caractérisées uniquement en fonction de variables scalaires (voir Section 5.1.6).

Analyse par *PIPS* des catégories d'accès aux tableaux des codes analysés

La dernière raison pour l'imprécision de *PIPS* est l'utilisation très fréquente d'adressage indirect, c'est-à-dire l'emploi de la valeur d'un élément de tableau (en passant par une variable intermédiaire, comme Fortran77 impose de le faire) comme indice d'un autre (voire du même) tableau. *PIPS* est incapable de caractériser les régions résultantes.

Nous avons trouvé dans les codes analysés que les accès aux tableaux appartiennent à quatre catégories principales. Les trois premières catégories caractérisent typiquement les accès aux trois catégories de structure de données décrites dans le Chapitre 3 (Section 3.3.5). Les trois dernières catégories emploient l'adressage indirect et nous avons déjà dit que *PIPS* ne peut pas caractériser les bornes exactes d'un ensemble d'éléments référencés par adressage indirect. Décrivons chacune de ces catégories d'accès au tableau et sa caractérisation par *PIPS*.

1. Employer comme indice une variable dont la valeur est une fonction linéaire de variables scalaires et de constantes. Cette méthode est employée, par exemple,

   - pour lire un tableau réalisant le stockage des attributs d'une classe d'objets, comme décrit dans le Chapitre 3 (Section 3.3.5),

     par exemple la lecture du tableau `IVAC` qui stocke la variance de chacune des télé-mesures numérotées de 1 à `NTM` présentée dans la Figure 5.1,

   ou

   - pour initialiser diverses structures de données.

   Si un accès de ce type est employé dans une boucle `DO`, alors *PIPS* peut trouver les bornes exactes de la région sous les conditions évoquées dans la Section 5.1.2.

2. La première catégorie d'accès par adressage indirect est la lecture d'une table partitionneuse de listes réalisée de la façon décrite dans le Chapitre 3 (Section 3.3.5).

   Un exemple est l'extrait de `CASDEG` présenté dans la Figure 5.2.

   Soit i (la valeur d'`IPCO(IC)` dans notre exemple) la valeur codée du début d'une liste de la table, c'est-à-dire :

   - 0 si la liste est vide et

1. La boucle `while` n'appartient pas à la syntaxe standard de Fortran77 mais *PIPS* accepte et analyse les `while`.

```
      SUBROUTINE CAJAC (NTM, NH, VCAL, IPCO, IPCE, ICCO, ICCE,
     +                  NRLO, NRLE, NRGR, NRCO, NRTJ, ZIMP, NXUT,
     +                  IVAC, LITA, LHAC, NCHA, EHA)
   ...

C  <LITA(PHI1)-R-EXACT-{1<=PHI1, PHI1<=NTM, 1<=IBT, IBT<=2}>
C  <NRLO(PHI1)-R-MAY-{1<=PHI1, 1<=IBT, IBT<=2, 1<=NTM}>
C  <NXUT(PHI1)-R-MAY-{1<=PHI1, 1<=IBT, IBT<=2, 1<=NTM}>
C  ...

C -----------------------------------------------------------
C       CONSTRUCTION DU JACOBIEN UNE LIGNE PAR TM
C -----------------------------------------------------------

      DO 3 K = 1, NTM                                          0002

C  <LITA(PHI1)-R-EXACT-{PHI1==K, 1<=PHI1, 1<=IBT, IBT<=2, K<=NTM}>

         LMES = IABS(LITA(K))                                  0003

         NOUV = LMES/10                                        0004
         ...

C  <NXUT(PHI1)-R-EXACT-{PHI1==NOUV, 1<=PHI1, 1<=IBT, IBT<=2, 1<=K,
C    K<=NTM, 1<=KTI, KTI<=10}>

         LNXU = NXUT(NOUV)                                     0020

         NOV = LNXU/10                                         0021
         LTY = LNXU-10*NOV                                     0022

C  <NRLO(PHI1)-R-EXACT-{PHI1==NOV, 1<=PHI1, LTY+10NOV==LNXU, 1<=IBT,
C    IBT<=2, 1<=K, K<=NTM, 1<=KTI, KTI<=10, 10NOV<=LNXU,
C    LNXU<=9+10NOV}>

         IO = NRLO(NOV)                                        0023
         ...
3        CONTINUE                                              0033
```

FIG. 5.3: Fragment de `CAJAC` présentant l'accès à `NRLO` dans une boucle `DO`, annoté par *PIPS*

– l'indice du premier élément de la liste stockée dans le tableau T (`ICCO` dans notre exemple) sinon.

Dans la lecture de la liste débutant par i, les éléments de T utilisés sont ceux indexés par $T^n[i]$ pour toutes les valeurs naturelles de n telles que l'élément $T^{n-1}[i]$ est utilisé et est supérieure à zéro (où $T^0[i] = i$, $T^1[i] = T[i]$, $T^2[i] = T[T[i]]$ ...).

Nous pouvons définir des bornes de cet ensemble d'indices pour le cas le plus général (qui risquent d'être des sur-approximations dans des cas particuliers) : la borne inférieure est 1 et la borne supérieure est **la borne supérieure de toutes les valeurs d'éléments de T.**

PIPS est incapable d'exprimer la borne supérieure dans ce cas, car il n'a pas de notation pour la borne supérieure de toutes les valeurs d'éléments d'un tableau.

Soit $T0$ (`IPCO` dans notre exemple) le tableau employé pour stocker toutes les têtes de liste.

Supposons que toutes les listes indexées par les n (`NSO` dans notre exemple) premiers éléments de $T0$ soient lues.

Alors les bornes des indices des éléments de $T0$ utilisés sont 1 et n.

Évidemment, *PIPS* peut calculer la région EXACT pour $T0$ dans ce cas, car il s'agit d'une instance de la catégorie d'accès précédent.

3. La deuxième catégorie d'accès par adressage indirect est la lecture des listes stockées de la façon décrite dans le Chapitre 3 (Section 3.3.5).

   Un exemple en est la lecture des plages de `NCHA` indexées par des éléments de `LHAC`, présentée dans la Figure 5.1.

   Soit $T0$ (`LHAC` dans notre exemple) le tableau employé pour stocker les débuts des listes stockées dans le tableau T (`NCHA` dans notre exemple).

   Soit i (`IL` dans notre exemple) l'indice d'une liste stockée dans T et n (`NTM` dans notre exemple) le nombre de listes.

   Dans le cas de la lecture d'une seule liste, les bornes des indices des éléments de T utilisés sont les valeurs de $T0[i]$ et de $T0[i+1]-1$.

   PIPS peut calculer la région EXACT de T dans ce cas mais elle est exprimée en fonction des valeurs des variables locales employées pour stocker les valeurs lues dans $T0$, et non pas en fonction des valeurs d'éléments de $T0$.

   Dans le cas de la lecture de toutes les listes indexées par les n premiers éléments de $T0$, les bornes des indices des éléments de $T0$ utilisés sont 1 et $n+1$.

   PIPS peut calculer la région EXACT dans ce cas.

   Les bornes des indices des éléments de T utilisés dans ce cas sont les valeurs de $T0[1]$ et de $T0[n+1]-1$.

   PIPS ne peut pas exprimer ces bornes.

4. La dernière catégorie d'accès est juste le cas général des indirections successives. Elle est illustré par l'exemple suivant.

   Dans le fragment de `CAJAC` présenté dans la Figure 5.3, le tableau `NRLO` est utilisé dans une boucle `DO` mais avec comme indice, non pas la variable `K` de la boucle, mais la variable locale `NOV`. La valeur de `NOV` dépend de la valeur du tableau `NXUT` indexée par la variable locale `NOUV` dont la valeur dépend de la valeur du tableau `LITA` indexée par la variable `K` de la boucle `DO`. Les bornes des indices des éléments de `NRLO` utilisés sont les bornes des valeurs de `NXUT` indexées par les différentes valeurs de `NOUV`. Les bornes de la valeur de `NOUV` dans le cas général sont une fonction des bornes des valeurs possibles de la plage de `LITA` comprise entre les bornes de la boucle `DO`.

PIPS peut trouver la région EXACT pour `LITA` dans ce cas (il s'agit d'un exemple de la première catégorie d'accès au tableaux décrit ci-dessus), mais ne peut pas trouver les bornes des indices des éléments de `NRLO` utilisés, ni celles pour `NXUT`.

En conclusion, *PIPS* ne peut pas toujours calculer les bornes exactes des éléments de chaque tableau utilisés, dans les exemples typiques des accès aux tableaux rencontrés dans les codes que nous avons analysés.

5.2 L'emploi de *PIPS* pour la rétro-ingénierie

PIPS fournit une analyse puissante du flot des données des codes Fortran qui semble intéressante pour la rétro-ingénierie étant donné la précision de l'analyse des références aux tableaux. Toutefois, *PIPS* a été conçu pour satisfaire les besoins de la parallélisation et les régions qu'il calcule ne sont pas toujours bien adaptées aux analyses de rétro-ingénierie que nous avons envisagées. Dans cette section, nous étudions l'adéquation des régions *PIPS* pour trois des besoins en analyse statique que nous avons constatés dans notre cadre de rétro-ingénierie :

1. l'aide à la compréhension du code,
2. l'analyse de l'appel par référence et
3. la détection des *vrais paramètres* d'entrée et de sortie des sous-programmes.

5.2.1 Des régions pour aider à la compréhension du code

Ne pas analyser chaque élément de tableau comme n'importe quel scalaire mais analyser des tableaux entiers ou, pour plus de précision, des plages de tableaux, peut faciliter la compréhension du code Fortran.

Prenons le cas où un programmeur

1. choisit de stocker certaines données dans un tableau (ou plage de tableau) au lieu d'employer des variables scalaires et
2. crée une boucle comportant une instruction avec référence à un élément du tableau indexé par une variable dont la valeur varie avec les itérations successives de la boucle.

La boucle a pour effet d'exécuter cette instruction sur différents éléments de la plage du tableau. Même si la boucle ne fait pas référence à tous les éléments à l'intérieur de la plage, elle la "traverse" en traitant de la même façon plusieurs éléments. Nous pouvons conclure que cette plage de tableau fait partie d'une structure de données, et donc qu'il serait utile d'analyser, dans le cadre de la compréhension du code, cette plage (par opposition à l'analyse de ses éléments individuels ou du tableau entier).

L'exemple le plus évident de ce cas de figure est l'emploi de plusieurs tableaux (ou plages de tableau) pour réaliser une classe d'objets avec attributs comme décrit dans le Chapitre 3 (Section 3.3.5). Dans ce cas, chaque plage stocke un ensemble de valeurs homogènes : tous les éléments de la plage peuvent être traités de la même façon.

Mais lorsqu'une plage de tableau est employée dans la réalisation d'autres structures de données, les éléments ne sont pas forcément homogènes car ils peuvent servir non seulement à représenter les données stockées, mais aussi la structure elle-même. Par ailleurs, il peut exister des contraintes sur les combinaisons des valeurs des différents éléments de la plage. Dans ce cas, il ne faut pas chercher un "sens" pour chacun des éléments arbitraires, mais pour la plage toute entière.

Définissons la *plage de tableau traversée par une boucle* comme l'enveloppe convexe des éléments d'un tableau référencés par la boucle.

Adéquation des régions *PIPS*

Prenons le cas d'une boucle `DO` comportant une référence à un tableau, et dont la région de ce tableau calculée par *PIPS* pour la boucle est EXACT. Cette région est exactement la plage du tableau qui participe à la réalisation d'une structure de données.

Si l'exécution de l'instruction de la boucle faisant référence au tableau était conditionnelle, alors les éléments effectivement référencés ne formeraient pas un polyèdre convexe et la région *PIPS* serait MAY. Mais la plage du tableau participant à la réalisation de la structure de données serait la même. En effet, pour caractériser la plage d'un tableau intervenant dans la réalisation d'une structure de données, il suffit de connaître les **bornes** des indices de l'ensemble des éléments pouvant être référencés par la boucle : la convexité de cet ensemble ne nous intéresse pas. Contrairement à *PIPS*, nous voudrions faire une distinction entre les régions dont les **bornes** sont exactes et les régions dont les **bornes** sont sur-approximées. Une région aux bornes sur-approximées ne précise pas la plage du tableau intervenant dans la réalisation d'une structure de données. Elle ne fait que signaler son existence.

Dans le cas d'autres formes de boucle, la région trouvée par *PIPS* est exactement la plage du tableau qui participe à la réalisation d'une structure de données, à condition que

- la boucle comporte une seule référence au tableau et
- les bornes de la région soient exactes.

Par contre toute "summary region" qui réunit les régions engendrées par des références différentes au même tableau

- dans une même boucle ou
- dans deux boucles d'un même sous-programme,
- voire dans deux sous-programmes différents appelés par un autre,

ne peut pas être exploitée directement par nos analyses.

En conclusion, les régions les plus exploitables dans le cadre de la compréhension du code sont celles engendrées par une seule référence au tableau dans une boucle et dont nous savons si leurs bornes sont exactes. Appelons celles-ci les *régions de tableau engendrées par des boucles*. Elles caractérisent les *plages de tableau traversées par des boucles*. Les régions trouvées par *PIPS* n'en sont qu'une approximation.

5.2.2 L'analyse de l'appel par référence

L'*exploitation de l'appel par référence* décrite dans le Chapitre 3 (Section 3.1.6) a pour effet de passer au sous-programme un ensemble d'éléments du tableau borné en bas par le paramètre effectif, mais dont la borne supérieure ne peut être déterminée (ou approximée) que par l'analyse statique du sous-programme. Ce cas de figure est fréquent dans les codes que nous avons analysés.

Le passage de bornes inférieures d'ensembles d'éléments dont la borne supérieure est inconnue n'est pas ce que nous voulons spécifier dans la spécification concrète. Par ailleurs, nous avons expliqué dans la section précédente pourquoi il est intéressant d'analyser les *plages de tableau traversées par des boucles*. Dans un cas d'exploitation de l'appel par référence, nous voulons connaître quelles plages traversées par des boucles sont effectivement passées. C'est ainsi que nous pouvons savoir quels sous-programmes accèdent à la même plage d'un tableau donné, ou aux plages différentes.

En règle générale, nous pouvons supposer qu'un tableau ne contient qu'une seule plage traversée par des boucles. Cette plage représente alors les bornes effectives du tableau. Dans certains cas, nous pouvons trouver des tableaux contenant une plage qui elle même est partagée en plusieurs autres plages. En particulier, nous voulons savoir s'il existe des tableaux composés de plusieurs plages disjointes et qui ne sont pas réunies dans une autre plage plus grande. Dans ce dernier cas, les sous-programmes accédant à des plages différentes du même tableau peuvent traiter des structures de données distinctes.

En d'autres termes, il nous faut un outil d'analyse statique capable de différencier les accès aux différentes plages d'un même tableau et de

1. trouver les bornes des plages et
2. comparer les plages entre elles, c'est-à-dire
   (a) reconnaître que deux plages sont distinctes ou qu'elles ont une intersection non vide et
   (b) reconnaître que deux plages sont identiques.

La simulation de l'allocation dynamique de la mémoire, de la façon décrite dans la Section 3.3.2, repose sur l'exploitation de l'appel par référence. Des plages contiguës d'un même "grand" tableau sont adressées comme autant de "petits" tableaux "virtuels". Chaque plage a sa propre "sens" mais le seul "sens" du tableau entier est le regroupement des plages. Étudions l'analyse par *PIPS* de cette simulation de l'allocation dynamique.

Prenons comme exemple le tableau `NA1` de `BESTI1`. Les débuts des plages successives de ce tableau sont indexés par les variables

`I_IPCO`, `I_IPCE`, `I_ICCO`, `I_ICCE`, `I_NRLO`, `I_NRLE`, `I_NRGR`, `I_NRCO` et `I_NRTJ`,

dont les valeurs sont calculées par le fragment suivant de `BESTI1`:

```
I_IPCO = 19
I_IPCE = I_IPCO + NS
I_ICCO = I_IPCE + NS
I_ICCE = I_ICCO + NL
I_NRLO = I_ICCE + NL
I_NRLE = I_NRLO + NL
I_NRGR = I_NRLE + NL
I_NRCO = I_NRGR + NG
I_NRTJ = I_NRCO + NC
```

Appelons les plages respectives du tableau `NA1`, ainsi que les tableaux qu'elles simulent,

`IPCO`, `IPCE`, `ICCO`, `ICCE`, `NRLO`, `NRLE`, `NRGR`, `NRCO` et `NRTJ`.

En effet, elles instancient des paramètres formels de ces noms dans la plupart des sous-programmes appelés par `BESTI1`, mais avec quelques exceptions. Quand nous parlons d'un accès au "tableau `NRLE` ", par exemple, nous voulons dire un accès à un tableau paramètre formel du sous-programme qui peut être instancié par la plage de `NA1` dont le début est indexé par `I_NRLE`.
Voici, par exemple, la déclaration du sous-programme `PRXPLO`:

```
 SUBROUTINE PRXPLO(IPCO, IPCE, ICCO, ICCE,
&                 NDOB, NRLO, NRLE, NRGR,
&                 NRCO, NRTJ)
```

et son point d'appel dans `PREFIX`:

```
 CALL PRXPLO(NA1(I_IPCO), NA1(I_IPCE), NA1(I_ICCO), NA1(I_ICCE),
&           NDOB,        NA1(I_NRLO), NA1(I_NRLE), NA1(I_NRGR),
&           NA1(I_NRCO), NA1(I_NRTJ))
```

Un rétro-ingénieur qui voudrait connaître les longueurs des différentes plages d'un "grand tableau" procéderait probablement comme suit : il supposerait les plages contiguës et déduirait la longueur de chaque plage de la valeur de l'indice de début de la plage suivante.

Dans le cas des indices du tableau `NA1`, cette approche implique que

- la longueur de `I_IPCO`, `I_IPCE`, `I_ICCO` et `I_ICCE` est NS,
- la longueur de `I_NRLO` et `I_NRLE` est NL,
- la longueur de `I_NRGR` est NG et
- la longueur de `I_NRCO` est NC

mais ne permet pas de deviner la longueur de `NRTJ`.

Évidemment un outil automatique tel que *PIPS* ne peut pas faire de telles suppositions.

En fait, l'indice `I_NRGR` est employée dans les sous-programmes `CACONT` et `CADSTM` comme point de départ d'une seule plage (instanciant le paramètre formel `NRGC` de ces deux sous-programmes) de longueur `NG` + `NC` recouvrant donc les deux plages contiguës de `NA1`, `NRGR` et `NRCO`, tandis que dans le sous-programme `PRXPLO`, `I_NRGR` sert à indexer la seule plage `NRGR` de longueur `NG`.

Caractérisation de la borne supérieure des plages pour chaque point de référence

PIPS devrait trouver la borne supérieure de chaque plage, non pas à partir de la borne inférieure de la plage "suivante" mais à partir des accès aux éléments de la plage. Cependant, comme nous l'avons expliqué (dans la Section 5.1.6), *PIPS* ne peut pas caractériser de façon précise tous les types d'accès.

En effet, *PIPS* n'arrive pas à trouver les bornes exactes des régions engendrées par

- trois des cinq accès à `IPCO` ou `IPCE`[2] : il s'agit de cas où la valeur de l'indice dépend de la valeur d'un autre tableau,
- quatre des cinq accès à `ICCO` ou `ICCE`, qui stockent les fins de listes chaînées (et lors de la lecture des listes, l'indice a donc une valeur calculée récursivement, et dépendante de la valeur d'un autre élément du tableau) et
- six des onze accès à `NRLO` ou `NRLE` pour des raisons identiques aux précédentes.

Mais sur le code restructuré automatiquement pour éliminer le plus possible de `GOTO` (voir Section 5.1.6), *PIPS* arrive à trouver les bornes exactes des régions engendrées par

- les cinq références aux éléments de `NRTJ`,
- les deux références aux éléments de `NRGR`,
- les deux références aux éléments de `NRCO`,
- les deux références aux éléments d'une plage recouvrant les tableaux `NRGR` et `NRCO`
- la référence aux éléments d'une grande plage recouvrant les tableaux `NRLO`, `NRLE`, `NRGR`, `NRCO` et `NRTJ`.

Traduction des régions dans un même système de référence

Nous venons de décrire la caractérisation par *PIPS* des accès aux paramètres formels, de divers sous-programmes, instanciés par des plages d'un grand tableau, **pour la boucle la plus petite contenant l'accès au tableau**. C'est-à-dire que chaque région est paramétrée par la valeur au moment de l'exécution de la boucle des variables de l'*espace de nommage* du sous-programme dans lequel se trouve cette boucle. Nous ne pouvons pas comparer les différentes plages du grand tableau référencées, sans traduire les régions correspondantes dans un même système de référence.

- D'une part, il s'agit de la traduction des régions dans l'espace de nommage d'un sous-programme qui appelle directement ou indirectement tous les autres sous-programmes contenant des accès au grand tableau. Cette traduction est effectuée par *PIPS* afin de propager récursivement les accès effectués dans des sous-programmes appelés vers le point d'appel dans l'appelant.

  Rappelons que pour effectuer la propagation des accès effectués dans des sous-programmes appelés, *PIPS* calcule la "summary region" de toutes les modifications (resp. utilisations) d'éléments d'un même tableau par un sous-programme, ce qui entraîne une perte de l'information sur les régions engendrées par chaque point d'accès.

  Un premier exemple des effets est le suivant :
  Les régions caractérisant les deux modifications possibles dans `PRCHEN` au tableau `ICCO` :

  - l'un avec comme indice `IL`, un paramètre formel de `PRCHEN` instancié par la variable de la boucle `DO` de `PRXPLO` qui contient les appels à `PRCHEN`,
  - l'autre avec comme indice la variable locale `KP` dont la valeur est calculée récursivement,

  sont regroupés en une région pour l'appel à `PRCHEN` dont les bornes ne sont pas exactes. Si nous renommons le tableau au différents points d'accès afin de forcer *PIPS* à calculer séparément la région pour chaque point d'accès, alors il trouve une première région dont les bornes exactes sont les bornes de la boucle `DO` et une deuxième région aux bornes sur-approximées. Les bornes exactes de la première région sont donc "perdues" dans le calcul de la "summary region".

2. Les tableaux `IPCO` et `IPCE` sont traités de façon symétrique, ainsi que les deux tableaux `ICCO` et `ICCE` et les deux tableaux `NRLO` ou `NRLE`.

Régions caractérisant la plage `IPCO`

EXACT	[19,`18+F_NS`] si `18+F_NS` $\leq$ 7118
MAY	[19,7118]

Régions caractérisant la plage `ICCO`

MAY	[`19+2F_NS`,`18+F_NL+2F_NS`] si $bi \geq 1$ et $bs \leq 7118$
MAY	[`19+2F_NS`,7118] si `19+2F_NS` ≥ 1
MAY	[`19+2F_NS`,7118] si `F_NS` ≥ 1

Régions caractérisant la plage `NRLO`

EXACT	[19+2`F_NL` +2`F_NS`,18+3`F_NL` +2`F_NS`] si $bi \geq 1$ et $bs \leq 7118$
MAY	[19+2`F_NL` +2`F_NS`,18+3`F_NL` +2`F_NS`] si $bi \geq 1$ et $bs \leq 7118$
MAY	[19+2`F_NL` +2`F_NS`,7118] si 19+2`F_NL` +2`F_NS` ≥ 1
MAY	[19+2`F_NL` +2`F_NS`,7118] si `F_NS` ≥ 1

Région caractérisant l'union des plages `NRLO`, `NRLE`, `NRGR`, `NRCO` et `NRTJ`

MAY	[19+2`F_NL` +2`F_NS`,7118] si $bi \geq 1$

Régions caractérisant la plage `NRGR`

EXACT	[19+4`F_NL` +2`F_NS`,18+4`F_NL` +`F_NG` +2`F_NS`] si $bi \geq 1$ et $bs \leq 7118$
MAY	[19+4`F_NL` +2`F_NS`,18+4`F_NL` +`F_NG` +2`F_NS`] si $bi \geq 1$ et $bs \leq 7118$

Régions caractérisant l'union des plages `NRGR` et `NRCO`

EXACT	[19+4`F_NL` +2`F_NS`,18+4`F_NL` +`F_NC` +`F_NG` +2`F_NS`] si $bi \geq 1$ et $bs \leq 7118$
MAY	[19+4`F_NL` +2`F_NS`,18+4`F_NL` +`F_NC` +`F_NG` +2`F_NS`] si $bi \geq 1$ et $bs \leq 7118$

Régions caractérisant la plage `NRCO`

EXACT	[19+4`F_NL` +`F_NG` +2`F_NS`,18+4`F_NL` +`F_NC` +`F_NG` +2`F_NS`] si $bi \geq 1$ et $bs \leq 7118$
MAY	[19+4`F_NL` +`F_NG` +2`F_NS`,18+4`F_NL` +`F_NC` +`F_NG` +2`F_NS`] si $bi \geq 1$ et $bs \leq 7118$

Régions caractérisant la plage `NRTJ`

MAY	[19+4`F_NL` +`F_NC` +`F_NG` +2`F_NS`,7118] si $bi \geq 1$
MAY	[19+4`F_NL` +`F_NC` +`F_NG` +2`F_NS`,7118] si $bi \geq 1$ et $1 \leq$ `F_NS`

FIG. 5.4: Régions calculées par *PIPS* au point d'entrée de `BESTI1` pour les accès à `NA1`

Comme deuxième exemple : dans `PREFIX`, le calcul des "summary regions" a pour effet d'unir les régions engendrées par des références aux différentes plages de `NA1` en une seule région aux bornes sur-approximées.

Pour évaluer les possibilités de *PIPS* en l'absence du calcul des "summary regions", nous l'avons testé sur une version d'*ESTIME* modifiée en renommant, à chaque point d'accès, chaque tableau instancié par une plage de `NA1`. Ceci afin de "tromper" le calcul des "summary regions".

- D'autre part, pour chaque sous-programme et pour chaque région calculée
  - pour une boucle contenant un accès au tableau ou
  - pour un appel à un autre sous-programme

  au milieu du sous-programme, il faut exprimer la région en fonction du store au point d'entrée du sous-programme, comme nous l'avons expliqué dans la Section 5.1.2.

  Dans notre cas, il s'agit de traduire les régions dans l'espace de nommage de `BESTI1` et les ré-exprimer en fonction du store au point d'entrée de celui-ci. Les bornes exactes des régions sont des fonctions linéaires des variables `NL`, `NS`, ... qui sont déclarées dans une zone `COMMON` et visibles dans tous les sous-programmes intervenant dans la traduction et ne changent pas de valeurs, sauf pour la variable `NT`. La traduction est donc faite sans perte de précision pour les régions aux bornes exactes, sauf en ce qui concerne celles paramétrées par `NT`.

  En fait, dans le code analysé, nous avons simulé la lecture du fichier `'ASDMFX'` par la copie de dix nouveaux paramètres formels **scalaires** d'entrée ajoutés à `BESTI1`, comme nous le décrivons dans le Chapitre 6) ci-dessous (Section 6.1.3). Les valeurs lues dans `'ASDMFX'` et affectées aux variables `NL`, `NS`, ... ,`NT` sont simulées alors par les nouveaux paramètres d'entrée de `BESTI1`, `F_NL`, `F_NS`, ... ,`F_NT`.

  Seule `NT` peut être modifiée suite à l'affectation de la valeur lue dans `'ASDMFX'`. Alors, *PIPS* ne peut pas ré-exprimer les bornes de région paramétrées par `NT` en fonction de la valeur au point d'entrée de `BESTI1` de `F_NT`. Il remplace donc les bornes supérieures exactes des régions concernées (et notamment celle caractérisant la grande plage recouvrant les tableaux `NRLO`, `NRLE`, `NRGR`, `NRCO` et `NRTJ`), par la sur-approximation.

  Cependant, l'ultime modification de `NT` est toujours exécutée avant tous les accès dont les régions sont paramétrées par `NT`. Dans de tels cas, on obtiendrait un résultat plus précis s'il était possible de comparer deux régions obtenues à deux points différents d'un même sous-programme en ré-exprimant l'une en fonction du store de l'autre, mais sans être obligé de ré-exprimer les deux en fonction du store initial.

  Les régions de `NA1` obtenues de cette façon au point d'entrée de `BESTI1` sont présentées dans la Figure 5.4. Le chiffre 7118 provient de la déclaration, dans `BESTI1`, de la dimension de `NA1`. À cause de ce dimensionnement, toutes les régions de la Figure 5.4 sont bornées en bas par 1 et en haut par 7118. Pour les régions dont l'expression des bornes paramétrées par le store est trop longue, nous avons employé la notation *bi* pour la borne inférieure et *bs* pour la borne supérieure dans les contraintes sur ces bornes.

Comparaison des régions

Les régions engendrées par les différents points d'accès à des éléments de `NA1` peuvent alors être comparées, soit à la main, soit en programmant une extension à *PIPS* permettant d'appeler les utilitaires de comparaison de polyèdres qui sont proposées sous forme de fonctions *C*.

Toutefois, des contraintes provenant des prédicats de branche vérifiées par certaines régions mais pas par d'autres peuvent fausser les comparaisons. L'idéal serait alors de pouvoir comparer deux régions **sous condition que le prédicat de branche de l'une soit vérifié**.

> Par exemple, la contrainte $1 \leq$ `F_NS` est incluse dans certaines des régions montrées dans la Figure 5.4 paramétrées par `F_NS`. Nous avons besoin de pouvoir comparer une région comportant cette contrainte avec une autre qui ne la comporte pas, **en supposant que la contrainte soit satisfaite**.

Cependant, dans le cas de deux régions aux bornes sur-approximées, le test d'égalité n'a pas de sens. Nous allons trouver identiques deux régions ayant les mêmes sur-approximations des bornes alors que ces régions peuvent représenter deux ensembles complètement disjoints d'éléments de `NA1`. Il en est de même pour le test d'intersection non vide de deux régions aux bornes sur-approximées. Par contre, nous pouvons établir

- l'inclusion d'une région aux bornes exactes ou sur-approximées dans une autre aux bornes exactes ou
- l'intersection non vide de deux régions aux bornes exactes.

À cause de la sur-approximation de l'union des plages `NRLO`, `NRLE`, `NRGR`, `NRCO` et `NRTJ`, nous ne pouvons pas détecter l'emploi de l'indice `I_NRLO` (mis à 19+2F_NL +2F_NL par `BESTI1`) pour indexer le début de deux plages différentes.

Conclusion

En conclusion, **en l'absence du calcul des "summary regions"** *PIPS* permet dans cet exemple de

- trouver des bornes exactes pour la plupart des régions de boucle engendrées par une référence à un élément d'un paramètre formel instancié par une plage de `NA1`,
- établir dans le cas de bornes exactes, que chaque indice `I_IPCO`, `I_IPCE`, ... est employé pour indexer un ensemble d'éléments de `NA1` disjoint des autres ensembles, sauf en ce qui concerne un des deux cas de superposition évoqués ci-dessus et
- détecter le cas de superposition concernant des régions pour lesquelles les bornes exactes ont été trouvées.

PIPS ne peut pas trouver de bornes exactes pour certaines régions. Il faut vérifier à la main dans ces cas, que la borne supérieure de l'ensemble d'éléments référencés en employant l'un des indices `I_IPCO`, `I_IPCE`, ... est bien celle à laquelle on s'attend.

Comme nous l'avons expliqué dans la Section 5.1.6, cette vérification nécessite une connaissance des bornes des valeurs possibles des éléments de certains tableaux (ou plages de `NA1`, ...).

Pour les tableaux (tels `IPCO` et `ICCO`) modifiés par le fragment de code analysé, il n'est pas trop difficile de trouver à la main les bornes des valeurs pouvant être modifiées. C'est une information que la technique d'*exécution symbolique* a pour vocation de fournir (voir Section 4.3.2 du Chapitre 4), mais pour le tableau `ICCO` (modifié par `PRCHEN`), par exemple, il faut trouver l'invariant de boucle.

Pour les tableaux (tels `NRLO` et `NRLE`) dont la valeur provient d'un fichier d'entrée, il faudrait trouver la plus petite *précondition* sur la valeur de ces fichiers qui garantit les bornes de l'ensemble des éléments de `NA1` référencés par une boucle particulière. Ainsi, pour les plages de `NRLO` et de `NRLE` dont les indices sont compris entre 1 et `NL`, nous posons que la valeur des éléments est limitée à l'intervalle entre -1 et `NS`. Ceci nous permet de trouver les bornes de la région d'`IPCO` engendrée par le point de lecture dans `PRCHEN`, avec comme indice la variable `N1`, car `N1` est mise à la valeur d'un élément de `NRLO` ou de `NRLE`. Le prédicat de branche vérifié par ce point de lecture inclut la condition que la valeur de `N1` soit supérieure à zéro. La région d'`IPCO` engendrée par ce point de lecture est donc bornée par 1 et `NS`.

5.2.3 Régions pour les vrais paramètres d'entrée et de sortie

Nous pouvons maintenant donner une définition des *vrais paramètres* d'entrée et de sortie d'un sous-programme Fortran77. Nous voulons une définition des paramètres qui permet de constater si deux sous-programmes différents traversent deux plages différentes du même tableau.

Références en mode *IN* ou *OUT*

Suivant *PIPS*, définissons une *référence en mode IN* à une variable scalaire ou élément de tableau dans un sous-programme comme une *utilisation* de la variable par une instruction du sous-programme qui peut être exécutée avant toute *modification* éventuelle de la même variable par le même sous-programme. Les *références en mode IN* sont alors les *"upwardly exposed uses"* de la littérature en anglais de l'analyse du flot des données.

Définissons une *référence en mode OUT* à une variable scalaire ou élément de tableau dans un sous-programme comme une *modification* de la variable par une instruction du sous-programme et dont la valeur peut être *utilisée* à l'extérieur du sous-programme dans la suite de l'exécution du code. Nous essayons, par cette définition, d'exclure des paramètres de sortie, les modifications de variable qui ne servent qu'à stocker des valeurs intermédiaires utilisées uniquement à l'intérieur du sous-programme. En anglais, les *références en mode OUT* sont les *"definitions"* qui sont *"live"* à la sortie du sous-programme.

Nous supposons que les entrées/sorties autres que par variable sont

- soit simulées par des paramètres supplémentaires (comme expliqué pour les entrées depuis les fichiers dans la Section 6.1.3 du Chapitre 6)
- soit prises en compte dans la définition ci-dessus, mais remarquons que dans le cas des *références en mode OUT* vers une imprimante ou un écran, il n'y a pas besoin de vérifier la relecture de la valeur !

Notons qu'une "variable" Fortran avec une valeur initialisée lors de la compilation du code par une instruction `PARAMETER`, `DATA` ou `BLOCK DATA` n'est pas en réalité une variable, mais une constante nommée. Les constantes ne peuvent évidemment pas être des vrais paramètres de sous-programme.

Plages traversées en mode *IN* ou *OUT*

Dans le cas de références à plusieurs éléments d'un même tableau, nous reprenons la notion de *plage de tableau traversée par une boucle* développée dans la Section 5.2.1. C'est pourquoi nous n'incluons pas dans les vrais paramètres, des ensembles arbitraires d'éléments d'un tableau, mais seules des plages de tableau.

Si le point de référence n'est pas situé dans une boucle, alors le vrai paramètre est la plage regroupant les éléments du même tableau pouvant être référencés **à ce même point de référence**. Cette plage peut être composée d'un seul élément, mais ne l'est pas nécessairement : prenons par exemple le cas du point de référence qui est un appel d'un autre sous-programme.

Définissons donc la *plage d'un tableau traversée* par le sous-programme *en mode IN* (resp. *OUT*) comme suit :
pour chaque référence à un élément ou autre plage de tableau, c'est le plus petit ensemble d'éléments du tableau, *enveloppe convexe* de tous ceux pouvant être *référencées en mode IN* (resp. *OUT*) à ce point.

Vrais paramètres de sortie

Les *vrais paramètres de sortie* d'un sous-programme regroupent alors

1. chaque variable scalaire pouvant être *référencée en mode OUT* par le sous-programme et
2. chaque *plage* de tableau pouvant être *traversée* par le sous-programme *en mode OUT*. Dans le cas d'un point de référence à l'intérieur de plusieurs boucles imbriquées, il s'agit de la plage traversée par la boucle externe.

Selon cette définition, si le sous-programme contient deux points de référence d'un même tableau, qui modifient deux ensembles différents d'éléments, regroupés dans deux *plages* différentes (et en supposant que toutes les modifications soient *utilisées* à l'extérieur du sous-programme), alors le sous-programme a deux paramètres différents de sortie dans le même tableau.

Vrais paramètres d'entrée

Les *vrais paramètres d'entrée* sont définis

- en remplaçant *OUT* par *IN* dans la définition ci-dessus,
- et en ajoutant éventuellement les paramètres supplémentaires d'entrée suivants :
  1. Une copie de chaque paramètre de sortie scalaire qui n'appartient pas aux paramètres d'entrée et qui peut ne pas être modifié par les sous-programme, mais dont la valeur à la sortie du sous-programme peut être utilisée au retour de l'appel. En effet, une variable dont la modification

par le sous-programme est *"may"* peut ne pas être modifiée par le sous-programme mais avoir sa valeur utilisée au retour de l'appel. Or nous ne pouvons pas donner au sous-programme une liste de paramètres de sortie qui varie avec le chemin d'exécution suivi (i.e. avec les valeurs des paramètres d'entrée). Dans ce cas, nous considérons alors que la valeur en entrée est "copiée" à la sortie. Il faut donc que cette variable soit paramètre d'entrée aussi, même si elle n'est pas *utilisée* par le sous-programme.

2. Une copie de chaque paramètre de sortie non scalaire qui n'appartient pas aux paramètres d'entrée et dont au moins un élément peut être utilisé au retour de l'appel du sous-programme sans avoir été modifié par les sous-programme. En effet, la valeur de cette plage du tableau à la sortie du sous-programme peut être composée en partie de sa valeur à l'entrée et en partie des nouvelles valeurs de certains éléments. Les valeurs à l'entrée du sous-programme de certains éléments de la plage peuvent apparaître dans la plage à la sortie du sous-programme, même si elles ne sont pas *utilisées* par le sous-programme.

Paramètres des plages

Les bornes d'une plage de tableau qui est vrai paramètre d'un sous-programme dépendent souvent des valeurs des vrais paramètres d'entrée (scalaires ou autres plages) de ce sous-programme. Si nous définissons, comme vrais paramètres, des plages incluant les éléments référencés pour toutes les valeurs possibles des paramètres d'entrée, alors nous perdons l'information sur ces dépendances. Pour plus d'informations, nous pouvons définir donc des plages dont les bornes sont paramétrées par les paramètres d'entrée du sous-programme. Toutefois, si nous voulons pouvoir comparer les vrais paramètres de **tous les sous-programmes d'un même code**, alors il faut que toutes les plages soient paramétrées, non pas par les paramètres d'entrée de leur sous-programme, mais par les paramètres d'entrée du code.

Adéquation des régions *PIPS*

Les régions IN et OUT calculées par *PIPS* ne caractérisent pas les vrais paramètres d'entrée et de sortie tels qu'ils sont définis ici. En l'absence du calcul des "summary regions", les régions IN et OUT d'un sous-programme caractériseraient tous les vrais paramètres sauf les paramètres supplémentaires d'entrée dont l'ajout est expliqué ci-dessus. Les régions *PIPS* pour un sous-programme sont paramétrées par la partie scalaire du *store* de l'*espace de nommage* du sous-programme au point d'entrée, c'est-à-dire : par les paramètres d'entrée scalaires du sous-programme.

Par ailleurs, dans le cas où seuls certains éléments d'une plage de tableau paramètre de sortie d'un sous-programme sont modifiés, *PIPS* peut être incapable de vérifier avec précision si uniquement ces éléments de la plage sont utilisés suite au retour du sous-programme. Il risque alors d'ajouter une région d'un paramètre d'entrée qui n'existe pas en réalité.

Le problème se pose en particulier dans le cas d'un tableau déclaré mais pas encore initialisé. Dans le code Fortran77, un tableau peut être déclaré dans un sous-programme mais pas modifié, et son adresse peut être passée le long d'une chaîne d'appels de sous-programme avant la modification effective de quelques-uns de ses éléments au sein d'un sous-programme que nous appellerons `SP`. Prenons la plage du tableau englobant les éléments pouvant être modifiés par `SP` et dont les nouvelles valeurs peuvent être utilisées au retour de l'appel de `SP`. Supposons que **tous** les éléments de la plage ne sont pas **toujours** modifiés par `SP`. À la sortie de `SP`, il peut y avoir des éléments de la plage qui sont toujours dans le même état que lors de la déclaration du tableau. Toutefois, il faut supposer que les valeurs de ces éléments ne sont jamais utilisées au retour de l'appel de `SP`, car la lecture de la valeur de toute variable Fortran77 non initialisée donne des résultats aléatoires et nous supposons que les codes que nous analysons ne contiennent pas de bogues (voir la Section 3.2 du Chapitre 3).

Dans ce cas, nous ne voulons pas faire de cette plage un paramètre d'entrée de `SP`. En entrée de `SP` le tableau n'a pas de valeur, il a juste une adresse. La valeur de type tableau est effectivement "créée" au sein de `SP` et la plage de tableau devrait donc être un paramètre de sortie uniquement.

Dans le cas d'une région non convexe ou aux bornes sur-approximées, plutôt que de demander à *PIPS* de vérifier de façon imprécise que seuls les éléments modifiés par `SP` sont utilisés par la suite, nous pouvons obtenir de meilleurs résultats en lui demandant de vérifier qu'il n'existe aucun point de modification (antérieure à l'appel de `SP`) de tout paramètre effectif pouvant instancier ce paramètre.

> Un exemple de ce cas de figure est la modification des éléments d'`ICCO` par `PRXPLO` (à travers son appel de `PRCHEN`). *PIPS* ne peut pas caractériser avec précision les éléments modifiés, ni ceux (les mêmes) utilisés par la suite (dans `CASDEG`, par exemple, voir Section 5.1.6). Alors, il ne va pas éliminer la possibilité qu'un élément d'`ICCO` non modifié par `PRXPLO` soit utilisé par la suite. Mais `ICCO` est toujours instancié par des plages de tableau vierges.

En l'absence des régions OUT

Remarquons qu'il est souvent difficile dans la pratique d'analyser le code entier afin de déterminer les valeurs utilisées à l'extérieur du fragment étudié. Dans ce cas, nous pouvons supposer qu'une variable modifiée par le fragment mais dont cette valeur n'est pas utilisée par lui est un vrai paramètre de sortie : nous supposons que la modification sert à quelque chose. Nous savons aussi qu'un bloc `COMMON` modifié par le fragment mais déclaré par aucun sous-programme à l'extérieur du fragment ne peut pas être un paramètre de sortie. Dans le cas d'autres variables modifiées par le fragment et dont les nouvelles valeurs sont utilisées par lui, l'approche la plus prudente, si nous sommes incapables de vérifier si ces valeurs sont utilisées à l'extérieur du fragment aussi, est d'inclure ces variables dans les paramètres de sortie.

Conclusion

Nous avons discuté dans quelle mesure le calcul des régions de tableaux effectué par *PIPS* est susceptible de nous aider dans la rétro-ingénierie. Dans le Chapitre 7 ci-dessous, ces idées sont précisées dans la description d'une analyse spécialisée que nous pouvons réaliser à l'aide de *PIPS*. Mais d'abord nous allons discuter la préparation du code à l'analyse.

Chapitre 6

Transformations du code source

Résumé

Diverses transformations du code Fortran sont décrites et discutées dans ce chapitre, qui commence par une brève introduction

Le sujet de la Section 6.1 est la transformation du code afin d'exclure certains aspects "non-fonctionnels" qui peuvent nuire aux résultats de l'analyse statique ou à la clarté de la spécification concrète.

Dans la Section 6.2, nous proposons la restructuration du code, pour les mêmes raisons, mais aussi pour aider à la compréhension du code par le rétro-ingénieur.

Dans la Section 6.3, il s'agit de la "décomposition" de certains sous-programmes en sous-programmes plus petits. Nous voulons départager les traitements de données "indépendantes" et encapsuler chaque boucle dans son propre sous-programme. Ceci afin de rapprocher la structuration du code Fortran en sous-programmes à la structuration de la spécification algébrique en prédicats et opérations, et faciliter ainsi l'axiomatisation de ces derniers.

Finalement nous discutons, dans la Section 6.4, les solutions possibles au problème des variables Fortran qui prennent plusieurs valeurs successives auxquelles nous ne voulons pas nécessairement donner la même sorte.

Nous concluons en précisant les transformations que nous avons effectué sur notre exemple principal avant de construire la spécification concrète.

Introduction

Comme expliqué dans le Chapitre 3 (Section 3.6), nous proposons d'effectuer certaines transformations sur le code avant de le spécifier, afin d'aboutir à une spécification concrète claire. Ces transformations sont justifiées par le fait qu'elles ne modifient pas la sémantique du code, ou du moins ne modifient que des aspects "inintéressants" qui ne seront pas décrits par la spécification concrète.

En principe, il s'agit de transformer le code Fortran initial en un autre code Fortran, mais nous verrons dans la suite qu'il est parfois intéressant d'étendre le langage des résultats des transformations, en supposant que nous soyons capables d'effectuer l'analyse statique du langage étendu.

En effet, les transformations sont motivées et justifiées par une analyse statique préalable du code, mais les résultats de cette analyse sont "jetés" après la transformation. Une fois toutes les transformations réalisées, le code est au niveau d'abstraction désiré pour effectuer les analyses statiques définitives proposées dans la suite de cette partie et conduisant directement à l'élaboration de la spécification concrète. C'est pourquoi il faut prendre en compte les effets potentiels de certaines transformations proposées sur les résultats des analyses ultérieures et aussi effectuer certaines transformations avant d'autres.

Nous décrivons brièvement aussi dans ce chapitre des transformations plus poussées et dont le résultat n'est pas pris en compte dans l'analyse statique ultérieure car trop loin du code initial, mais qui servent à mieux comprendre le code, comme nous l'avons expliqué dans la Section 3.7 du Chapitre 3.

Les transformations proposées ci-dessous sont très fastidieuses à réaliser à la main. Il faudrait les réaliser autant que possible à l'aide d'un outil automatique mais à notre connaissance, la restructuration est la seule transformation proposée dans ce chapitre qui est offerte (dans une certaine mesure) par

```
    ...
C                 ENTREES SORTIES PAR ARGUMENT
C *******************************************************************
C *          SIGNIFICATION               .  NOM   . DIMENSION.TYPE  *
  ...
C * SORTIES
C * ***********************************                             *
C * MATRICE INVERSE                      . B      . (N,N)    . C*16 *
C *                                      .        .          .      *
C * REJET D'ERREUR                       . IER    .    1     . I*4  *
C *  97  : PIVOT NUL (INFERIEUR A 1E-6)                             *
C *******************************************************************
     ...
     DATA EPSIL /1.E-6/
     ...
     DO 100 L = 1, M-1
     ...
       VABS3 = ABS(B(L,L))
       IF (VABS3.LT.EPSIL) THEN
         IER = 97
         RETURN
       ENDIF
     ...
 100 CONTINUE
     ...
     END
```

FIG. 6.1: Extrait de `XINVMX` montrant le "break" de la boucle `DO`

des outils opérationnels. La réalisation d'outils capables d'effectuer les autres transformations sort du propos de cette étude. La mise en œuvre de tels outils est assez délicate (par exemple, que faire des commentaires ?). Nous proposons ici, de façon relativement informelle, des transformations que nous avons testées à la main sur quelques exemples de code.

6.1 Exclusion des aspects "non-fonctionnels"

Comme expliqué dans le Chapitre 3 (Section 3.6), nous voulons exclure de la spécification concrète certains aspects "non-fonctionnels" du code. Le choix des aspects à exclure de chaque code analysé doit surtout être fait par l'être humain. Nous proposons d'exclure ces aspects par une transformation du code qui est "sur mesure" dans la plupart des cas et donc difficile à automatiser.

6.1.1 Élimination ou abstraction des tests de légitimité des valeurs

Prenons le cas d'un test de la valeur d'une variable qui provoque une sortie avec message d'erreur en cas de résultat négatif. L'échappement avec message d'erreur complique la structure de contrôle du code.

Deux exemples sont :

1. le test à la ligne 58 de `PREFIX` (présenté dans l'Annexe E) que la valeur lue d'un fichier et copiée dans `NS` est inférieure ou égale à la constante (définie par une instruction `PARAMETER`) `NMS` et
2. le test dans `XINVMX` (dont un extrait est présenté dans la Figure 6.1) que la valeur de `VABS` calculée par le sous-programme est supérieure ou égale à la constante (définie par une instruction `DATA`) `EPSIL`.

Ce dernier exemple est typique de nombreux sous-programmes de *COURCIRC*, où le "break" depuis le milieu d'une boucle `DO` en cas d'erreur empêche *PIPS* de trouver des limites aux régions référencées par la boucle.

Or une certaine notion d'abstraction consiste à ne pas spécificier le test de la valeur et la sortie en cas d'erreur, mais

- soit éliminer tout simplement le test du code et donc de la spécification concrète (c'est ce que nous faisons dans le deuxième exemple ci-dessus)
- soit l'éliminer du code et spécifier d'une autre façon la contrainte sur les valeurs, mais sans spécifier le comportement du code en cas de valeur illégitime. Par exemple, une inégalité imposée entre un paramètre d'entrée et une constante peut faire partie de la spécification de la *sorte* du paramètre. Ce serait possible dans le premier exemple ci-dessus.

  Toutefois, il faut vérifier si les informations détruites par cette transformation augmenteraient la précision de l'analyse statique : éliminer de `PREFIX` la condition de branche que `NS` est inférieure ou égal à `NMS` empêche *PIPS* de comparer une région paramétrée par `NS` à une région paramétrée par `NMS` (voir le Chapitre 5). En l'occurrence, cette comparaison n'est pas nécessaire.

6.1.2 Élimination des messages de déboguage et statistiques

Souvent, de nombreux messages sont écrits à l'écran ou à l'imprimante afin de rendre compte du progrès du calcul. La sortie ou le format de ces messages sont parfois conditionnés par la valeur de certaines variables représentant le “mode” d'exécution. À notre avis, considérer ces messages comme faisant partie des sorties du code, complique trop la spécification concrète. Toutefois, leur contenu est parfois utile à la compréhension du code. Nous proposons donc

- de supprimer les variables représentant le “mode” d'exécution, le format des sorties ... ,
- de supprimer des messages qui impriment simplement les valeurs des variables, l'entrée ou la sortie des routines, l'heure, ... et
- de transformer en commentaires des messages qui expliquent de façon textuelle le progrès du calcul[1].

6.1.3 Abstraction des détails des entrées/sorties sur fichier

La programmation en Fortran des entrées/sorties sur fichier est assez compliquée. Dans *ESTIME*, certaines sont gérées par les utilitaires `NLIRE` et `NECRIR`, dont le paramètre formel `CTAB` déclaré comme un tableau de caractères est instancié dans les appels par des tableaux de tous les types, sans respect des types déclarés. Nous ne souhaitons pas compliquer la spécification concrète par la spécification détaillée de ces entrées/sorties, (ni de compliquer l'analyse automatique par ces sous-programmes pas “propres”), car nous proposons de “faire abstraction” des entrées/sorties fichier en les remplaçant dans le code par des paramètres supplémentaires ajoutés au sous-programme dans lequel la lecture (resp. écriture) du fichier a lieu, ainsi qu'à tous les sous-programmes appelant celui-ci, directement ou indirectement. Chaque paramètre supplémentaire représente le contenu d'un fichier, qui est variable, et non pas le **nom** du fichier, constant. Si le contenu d'un fichier d'entrée (resp. sortie) est copié dans (resp. depuis) plusieurs variables stockées en mémoire de façon contiguë (comme dans l'exemple cité dans la Section 3.3.1 du Chapitre 3), alors le contenu de ce fichier est simulé par autant de nouveaux paramètres d'entrée (resp. sortie). Dans le cas où le contenu du fichier est copié dans (resp. depuis) un tableau, il est simulé par un nouveau paramètre de type tableau.

> Par exemple, le sous-programme `NLIRE` est employé pour copier le contenu d'un fichier dans un tableau ou bloc `COMMON`. Nous proposons de remplacer chaque appel de `NLIRE` qui sert à remplir un tableau par :
>
> - un paramètre supplémentaire d'entrée du même type que le tableau et
> - un appel à un autre sous-programme qui copie le contenu de ce paramètre d'entrée dans le tableau.

1. L'outil *FORESYS* propose un premier pas dans ce sens sous la forme d'une transformation qui rapproche à chaque instruction `WRITE`, l'instruction `FORMAT` correspondant, au lieu de grouper les instructions `FORMAT` à la fin du fichier.

Chaque appel de `NLIRE` qui sert à initialiser plusieurs variables d'un bloc `COMMON` est remplacé par

- un paramètre supplémentaire d'entrée et
- une instruction d'affectation

pour chacune des variables initialisées.

Ainsi l'appel de `NLIRE` par `BESTI1` pour copier les données stockées dans le fichier `'ASDMFX'` dans les variables `NL`, `NS`, ... , `NT` de bloc `COMMON` (employées pour dimensionner les tableaux virtuels dans la simulation de l'allocation dynamique de mémoire), est simulé par l'ajout des à `BESTI1` des paramètres d'entrée scalaires `F_NL`, `F_NS`, ... , `F_NT` et la copie des valeurs de ces paramètres dans `NL`, `NS`, ... , `NT`, respectivement.

6.1.4 Abstraction de la simulation de l'allocation dynamique

Dans la Section 5.2.2 du Chapitre 5 ci-dessus, nous avons expliqué la difficulté posée par l'analyse automatique de la simulation de l'allocation dynamique dans les codes *ESTIME* et *COURCIRC*. La solution que nous avons essayée est de garder la simulation de l'allocation dynamique dans le code analysé pour l'élaboration de la spécification concrète, et de compter sur une analyse statique très précise des accès aux tableaux.

Une autre solution à ce problème aurait été de transformer le code afin d'effectuer l'abstraction de la simulation de l'allocation dynamique, avant les analyses conduisant à l'élaboration de la spécification concrète. Il s'agit de remplacer chacun des grands tableaux par les petits tableaux qu'il simule. Il faut détecter chaque cas où un élément d'un grand tableau est employé dans un appel de sous-programme pour instancier un paramètre formel déclaré comme tableau de dimension inconnu. Dans ces cas, l'élément du grand tableau serait remplacé par un petit tableau. Le petit tableau correspond alors à la plage du grand tableau débutant par cet élément.

Cette approche est difficile à automatiser car elle ne serait justifiée que par une analyse préliminaire très précise des accès aux tableaux afin de détecter d'éventuels cas de chevauchement des plages du grand tableau effectivement référencées.

Dans le Chapitre 5 (Section 5.2.2), nous avons donné l'exemple de l'indice `I_NRGR` qui est employé pour indexer deux plages différentes de `NA1`, dont l'une inclut l'autre.

Dans le Chapitre 5 (Section 5.1.6), nous avons expliqué que dans certains cas l'analyse statique ne peut fournir les bornes exactes de la plage référencée par un fragment de code. Dans ces cas, il faut vérifier à la main le non-chevauchement des plages.

Dans *COURCIRC*, il s'agit d'éliminer aussi les `EQUIVALENCE` entre noms d'un même bloc `COMMON` (ce qui nécessite de vérifier d'abord que chaque indice est toujours employé avec le même identificateur de tableau), ainsi que les appels aux sous-programmes de la "bibliothèque" de gestion de la simulation de l'allocation dynamique.

Dans *ESTIME*, nous pourrions dimensionner les petits tableaux par les dimensions maximales constantes (`NMS`, `NML`, ... dont la valeur est fixée par une instruction `PARAMETER`) fournies, mais dans *COURCIRC* nous ne connaissons pas les dimensions maximales. En effet, la solution idéale serait alors de pouvoir réaliser l'analyse statique sur un code Fortran où certains tableaux sont dimensionnés par des variables (ou sinon, ne sont pas dimensionnés du tout).

6.2 Restructuration

La *restructuration* est la transformation du code source afin de simplifier l'architecture ou la structure de contrôle sans changer la sémantique. Nous parlerons ici de la restructuration relative à la structure de contrôle.

Le code et aussi sa spécification sont plus faciles à comprendre si la structure de contrôle est la plus simple possible. En particulier, la restructuration d'une boucle peut faciliter l'extraction d'un invariant qui permet de comprendre le rôle de la boucle. C'est pourquoi notre but est de simplifier le plus possible les structures de contrôle spécifiées.

Les transformations élémentaires

La restructuration résulte de la composition de nombreux petits pas de transformation, qui peuvent être sélectionnés de façon entièrement automatique ou interactivement. Ces transformations élémentaires sont relativement faciles à justifier. Elles incluent, par exemple :

- le dépliage (*"unrolling"*) de boucles,
- la duplication d'instructions à la place de l'emploi d'un `GOTO`,
- le réordonnancement d'instructions,
- l'élimination de tests ou de variables locales inutiles,
- l'élimination de la duplication d'instructions à travers l'ajout de nouveaux `GOTO` vers des étiquettes existantes et
- la re-formation de boucles.

L'élimination de tests ou de variables locales sont justifiées par la propagation des affectations de valeur, ainsi que des contraintes sur les valeurs impliquées par les résultats de tests. Les déplacements d'instructions sont justifiés par les *dépendances* entre les instructions : étant données une *modification* de variable et une *utilisation* de la valeur de la modification, nous pouvons déplacer l'une ou l'autre à condition de ne pas modifier toutes les dépendances entre utilisations et modifications de variables dans le fragment de code concerné.

Une approche consiste à constituer une bibliothèque de transformations élémentaires, chacune avec leur condition d'application. L'idéal est alors de prouver formellement que chaque transformation élémentaire conserve la sémantique du code si sa condition d'application est respectée. C'est ce qui est fait dans le système interactif de [Bul90], qui manipule une représentation du code exprimée dans un "wide spectrum language" formel.

Il existe des outils de restructuration entièrement automatique. Nous avons essayé celui (appelé *FOREST*) qui fait partie de *FORESYS* et celui intégré à *PIPS* (voir la Section 4.4 du Chapitre 4). Nous verrons dans la suite quelques exemples des limites de leur puissance.

La transformation des structures de contrôle hérités du FortranIV

Il s'agit de transformer

- les instances de `GOTO` *calculé* en `IF/THEN/ELSE` et `GOTO`,
- les instances de `IF` *arithmétique* en `IF/THEN/ELSE` et
- certaines utilisations du `IF` *logique* en `IF/THEN/ELSE`

afin de faciliter la compréhension et l'analyse ultérieure du code.

L'outil *FOREST* effectue ces transformations automatiquement.

La reconnaissance de boucles `while`

La boucle `while` n'existant pas dans le Fortran, il ne s'agit pas d'une transformation Fortran- Fortran mais

- soit d'une transformation dont le résultat est exprimé dans une extension de Fortran qu'il faut ensuite pouvoir analyser [2],
- soit de mettre en évidence les boucles `while` sans transformer le code.

2. *PIPS* accepte les codes contenant des boucles `while`.

Prenons comme exemple la boucle suivante du sous-programme `CAPIV`:

```
      ...
      KCH  = IPCO(NOUV)
   30    IF (KCH.LE.0) GOTO 1
         ...
    4    KCH = ICCO(KCH)
      GOTO 30
    1 RETURN
      END
```

Cette boucle réalisée à l'aide d'un `GOTO` est équivalente à une boucle `while` avec comme condition de sortie

$KCH \leq 0$
et comme invariant à la $n$ième itération

$KCH = ICCO^{n-1}[IPCO[NOUV]]$
(où
$ICCO^0[IPCO[NOUV]] = IPCO[NOUV]$,
$ICCO^1[IPCO[NOUV]] = ICCO[IPCO[NOUV]]$,
$ICCO^2[IPCO[NOUV]] = ICCO[ICCO[IPCO[NOUV]]], \ldots$)

L'outil de restructuration automatique proposé par *PIPS* ne reconnaît pas cet exemple de boucle `while`, mais elle est reconnue par *FOREST*, voir la Figure C.3 de l'Annexe C.

Restructurations plus poussées

Les buts (parfois conflictuels) d'autres restructurations sont :

- de regrouper les sorties de boucle,
- de regrouper les cas dans les tests,
- d'éliminer des variables locales ou des tests par une simplification de la structure de contrôle,
- d'obtenir un code structuré,
- ...

Prenons comme exemple le sous-programme `PRCHEN` présenté dans l'Annexe G (mais sans l'appel à `MOVEK`, éliminé parce qu'il ne sert qu'à la mise au point du logiciel). Nous avons employé ([WP96], [WP97] et [eNWP97]) une méthode de transformation manuelle d'organigramme pour l'analyse de ce sous-programme car nous croyons qu'il est plus facile pour un être humain de détecter des transformations intéressantes dans un (petit) organigramme que dans le code source. Nous avons obtenu le code structuré présenté dans la Figure 6.2, qui élimine le test de la ligne 61 du code initial, en remplaçant la première itération de la boucle du code initial par les deux instructions avant la boucle `while` dans le code structuré.

Ni l'outil de restructuration automatique de *PIPS*, ni *FOREST* ne trouvent cette simplification (voir l'Annexe C), qui nécessite le dépliage de la première itération de la boucle.

Comme nous l'avons expliqué dans le Chapitre 3 (Section 3.7), d'autres restructurations sont effectuées afin de **comprendre** le code, mais leur résultats ne peuvent pas être exprimés en Fortran non étendu et ils sont trop loin du code initial pour être spécifiés au niveau concret.

Ainsi, avant d'arriver à la **compréhension** totale de `PRCHEN`, il nous a fallu ajouter à des transformations vers un nouveau code toujours en syntaxe Fortran, les autres transformations vers un langage étendu décrites dans [WP96, WP97, eNWP97].

```
C     --- ------------------------------------------------------------ ---
C     --- INTRODUCTION D'UN ARC DANS UN CHAINAGE D'ARCS A LEURS ORIGINE ---
C     --- ------------------------------------------------------------ ---
      SUBROUTINE PRCHEN(IL,N1,N2,IPCO,ICCO,NRLE)
      ...
      DIMENSION IPCO(*), ICCO(*), NRLE(*)

      KCH = IPCO(N1)
      if ((N1.GT.0).AND.(N2.GT.0)) then
C       --- ------------------------------------------------------- ---
C       ---       ARC RACCORDANT DEUX SOMMETS OBSERVABLES            ---
C       ---       INITIALISATION , ENTREE DANS LA CHAINE             ---
C       --- ------------------------------------------------------- ---
        if ((KCH.GT.0).AND.(NRLE(KCH).LT.N2) then
C         --- ----------------------------------------------------- ---
C         ---                 CHAINE NON-VIDE                        ---
C         --- ----------------------------------------------------- ---
          KP  = KCH
          KCH  = ICCO(KCH)
C         --- ----------------------------------------------------- ---
C         ---     CLASSEMENT DES ARCS PAR LEUR SOMMETS EXTREMITES    ---
C         --- ----------------------------------------------------- ---
          do while ((KCH.GT.0).AND.(NRLE(KCH)).LT.N2))
            KP  = KCH
            KCH  = ICCO(KCH)
          end do
          ICCO(KP) = IL
        else
C         --- ----------------------------------------------------- ---
C         ---                CAS D UN DEBUT DE CHAINE                ---
C         --- ----------------------------------------------------- ---
          IPCO(N1) = IL
        endif
        ICCO(IL) = KCH
      endif
      RETURN
      END
```

FIG. 6.2: Restructuration manuelle de PRCHEN

Élimination de variables locales

Le code est plus facile à comprendre par un être humain si aux points de lecture, les variables autres que les paramètres d'entrée sont annotées par leur valeur au cas où une expression “simple” peut être trouvée pour celle-ci. La fonction d'exécution symbolique intégrée à l'outil *PIPS* réalise une telle annotation limitée aux cas d'une seule modification possible, d'une valeur constante ou égale à une expression linéaire de variables scalaires entières.

En ce qui concerne certaines variables locales servant à stocker une valeur intermédiaire, nous pouvons remplacer l'identificateur par la valeur et éliminer la variable. En particulier, si nous nous permettons d'étendre le langage des résultats des transformations, nous pouvons éliminer un certain nombre de variables locales en permettant les variables indexées dans les expressions et dans certaines instructions où elles ne sont pas permises par Fortran.

Par exemple, dans le code restructuré de `PRCHEN` présenté dans la Figure 6.2, la variable `KCH` peut être remplacée par `IPCO(N1)` dans certaines instructions et par `ICCO(KP)` dans d'autres. Les lignes ultimes du sous-programme sont ainsi transformées en les lignes suivantes et nous reconnaissons deux cas d'insertion d'un élément dans une liste chaînée :

```
          ICCO(IL) = ICCO(KP)
          ICCO(KP) = IL
        else
C         --- ------------------------------------------------------- ---
C         ---                  CAS D UN DEBUT DE CHAINE               ---
C         --- ------------------------------------------------------- ---
          ICCO(IL) = IPCO(N1)
          IPCO(N1) = IL
        endif
      endif
      RETURN
      END
```

Toutefois, l'élimination de variables locales peut nuire à la compréhension du code dans les cas où la variable locale représente un concept clairement identifié dans le domaine d'application,

> par exemple : la valeur de la variable `NPOSTI` de `QIOUVR` est facilement exprimée en terme des variables d'entrée, mais elle est employée dans de nombreuses instructions et représente le concept de numéro de poste.

Élimination de paramètres

Un premier cas de figure consiste en des sous-programmes avec paramètres déclarés qui ne sont pas référencés par le sous-programme. C'est le cas du paramètre `NDTM` de `PRXPLO` (voir Annexe F). Des déclarations de bloc `COMMON` peuvent parfois être éliminées aussi. Cependant, cette transformation n'est pas nécessaire si les vrais paramètres sont détectés comme nous proposons dans le Chapitre 7.

Le deuxième cas de figure est celui où la valeur de l'un des paramètres effectifs dépend de la même façon dans tous les appels d'un sous-programme, des valeurs des autres paramètres effectifs. Dans ce cas nous pourrions transformer le code afin d'éliminer ce paramètre. Ceci est un exemple d'analyse qui utilise le contexte des appels, comme expliqué dans le Chapitre 3 (Section 3.8).

> Par exemple, dans les deux appels de `PRCHEN` depuis `PRXPLO`, le paramètre effectif instanciant le paramètre formel `N2` vaut le paramètre effectif instanciant le paramètre formel `NRLE` indexé par le paramètre effectif instanciant le paramètre formel `IL`. Nous pourrions donc éliminer le paramètre formel `N2`, le remplaçant dans toutes les *utilisations* par `NRLE(IL)`.

Cette dernière transformation peut faciliter la compréhension du code. Cependant, il ne faut pas éliminer de tels paramètres des opérations de la spécification concrète, car le sous-programme spécifié ne remplit pas alors la même fonction que le sous-programme du code initial Fortran. En effet, cette transformation ne change pas la sémantique du fragment de code composé du sous-programme et de ses appels, mais elle change la sémantique du sous-programme (contrairement à l'élimination des paramètres qui ne sont pas référencés par le code). Elle compte parmi les transformations évoquées dans le Chapitre 3 (Section 3.7) qui ne sont pas prises en compte dans la spécification concrète.

6.3 Décomposition des sous-programmes

Comme expliqué dans le Chapitre 3 (Section 3.4), la structuration d'un code Fortran en sous-programmes ne correspond pas souvent à la structuration d'une spécification algébrique en prédicats et opérations. Or nous pouvons tenter par des méthodes automatiques ou semi-automatiques de reconnaître certains cas de figure et d'effectuer une première décomposition du code avant son analyse qui le rapproche un peu plus de la décomposition de la spécification algébrique. Il s'agit d'une transformation Fortran-Fortran ayant pour effet de créer des appels de nouveaux sous-programmes à l'intérieur des sous-programmes initiaux et donc

- de définir des paramètres d'entrée et de sortie,
- d'isoler le fragment de code correspondant au corps de chaque sous-programme et
- de le remplacer dans le code initial par un appel au nouveau sous-programme.

Toutefois, nous proposons de garder tous les sous-programmes initiaux afin de garder le lien entre la spécification concrète et le code, comme expliqué dans le Chapitre 3 (Section 3.6).

Notons que chaque nouveau sous-programme créé par la transformation que nous proposons aura normalement un seul point d'appel. En effet, nous ne proposons pas ici de "factoriser" différents fragments de code presque identiques par la définition d'un nouveau sous-programme. C'est un autre sujet de recherche (voir [BYM+98]) que nous n'avons pas eu le temps d'adresser, mais qui semble prometteur pour des sous-programmes tels que `ODIDAV`, `ODIDAM` et `ODIBAR` ou `FACTO`.

Nous proposons ici deux phases successives de transformation (avec une éventuelle réitération de la première phase sur les résultats de la deuxième) : la première phase a pour but d'encapsuler, dans de nouveaux sous-programmes différents, les traitements indépendants de données différentes, et la deuxième phase sert à encapsuler chaque boucle dans son propre sous-programme. Ces transformations sont à effectuer après la restructuration mais avant tout renommage des variables ayant plusieurs valeurs successives (voir Section 6.4 ci-dessous).

Séparer les traitements de données différentes

Une opération d'une spécification algébrique n'a qu'un paramètre de sortie mais les sous-programmes que nous avons analysés en ont souvent plusieurs. Dans certains cas il s'agit de structures de données réalisées à l'aide de plusieurs variables Fortran (voir la Section 3.4 du Chapitre 3), et dans ces cas nous voulons préserver le tuple des paramètres de sortie car il a un "sens". Mais d'autres sous-programmes regroupent des traitements de variables différentes qui pourraient être séparés. De tels sous-programmes existent pour deux raisons principales :

1. Le programmeur a voulu regrouper des traitements "similaires" en quelque sorte, par exemple : toutes les initialisations de données. Comme il sait que certaines données seront initialisées ensemble avant d'effectuer un calcul, alors il ne crée pas un sous-programme pour l'initialisation de chaque structure de données. Or dans une spécification algébrique les différentes opérations sur les structures de données de la même *sorte* sont souvent regroupées dans un module correspondant à un type abstrait de données, mais il n'y a aucune raison de regrouper en une opération des sous-opérations similaires sur des données différentes.

2. Le programmeur a voulu économiser une structure de sous-programme ou une structure de contrôle, alors il l'a faite partager par plusieurs traitements. Le partage du structure de contrôle est un exemple d'"interleaving" (entrelacement), qui est courant dans les vieux codes [RSW95]. L'entrelacement nuit à la clarté du code et nous ne voulons pas la spécifier dans la spécification concrète.

```
C       --- ------------------------------------------------------------ ---
C       ---     CONSTRUCTION DU VECTEUR D'EXPLORATION NODALE             ---
C       ---     OU DESCRIPTION DES SUCCESSEURS DES SOMMETS               ---
C       --- ------------------------------------------------------------ ---
      SUBROUTINE PRXPLO(IPCO,IPCE,ICCO,ICCE,NDOB,NRLO,NRLE,NRGR,NRCO,NRTJ)
      ...
      COMMON/DMFX/  NXL,NL,NXP,NXJ,NT,NC,NG,NBY,NS,JOUR(5)
      COMMON/IPAC/  SIGA,SIGR,NSO,NTMA,NTMR,LBLC(7,2),NFA,NBFA,NFR,
     +              NBFR ,NXU,BROB

      DIMENSION NRLO(*), NRLE(*), NRGR(*), NRCO(*), NRTJ(*)
      DIMENSION IPCO(*), IPCE(*), ICCO(*), ICCE(*), NDOB(*)

      NSO = NS
      CALL PRXPL1(IPCO,IPCE,ICCO,ICCE,NDOB,NRLO,NRLE)
      CALL PRXPL2(NDOB,NRGR)
      CALL PRXPL3(NDOB,NRCO)
      CALL PRXPL4(NDOB,NRTJ)
      RETURN
      END
```

Fig. 6.3: Décomposition de `PRXPLO` : 1ère phase

```
C       --- ----------------------------------------------------------- ---
C       --- CALCUL D IPCO,IPCE,ICCO,ICCE,NRLO,NRLE                      ---
C       --- ----------------------------------------------------------- ---
      SUBROUTINE PRXPL1(IPCO,IPCE,ICCO,ICCE,NDOB,NRLO,NRLE)
      ...
      COMMON/DMFX/  NXL,NL,NXP,NXJ,NT,NC,NG,NBY,NS,JOUR(5)
      COMMON/IPAC/  SIGA,SIGR,NSO,NTMA,NTMR,LBLC(7,2),NFA,NBFA,NFR,
     +              NBFR ,NXU,BROB

      DIMENSION NRLO(*),NRLE(*),IPCO(*),IPCE(*),ICCO(*),ICCE(*),NDOB(*)

C       --- ----------------------------------------------------------- ---
C       ---  ON ELIMINE LES ELEMENTS N'APPARTENANT PAS A LA COMPOSANTE  ---
C       ---  CONNEXE PRINCIPALE (NDOB=0)                                ---
C       --- ----------------------------------------------------------- ---
      do I = 1, NL
         K1 = NRLO(I)
         if (K1.GT.0) then
           IF (NDOB(K1).LE.0) NRLO(I) = -1
         else
           NRLO(I) = 0
         endif
         K1 = NRLE(I)
         IF (K1.GT.0) then
           IF (NDOB(K1).LE.0) NRLE(I) = -1
         else
           NRLE(I) = 0
         endif
      end do
C       --- ----------------------------------------------------------- ---
C       ---                    INITIALISATION                           ---
C       --- ----------------------------------------------------------- ---
      do I = 1, NS
         IPCO(I) = 0
         IPCE(I) = 0
      end do
C       --- ----------------------------------------------------------- ---
C       ---                CONSTRUCTION DES CHAINAGES                   ---
C       --- ----------------------------------------------------------- ---
      do I = 1, NL
         N1 = NRLO(I)
         N2 = NRLE(I)
         CALL PRCHEN (I, N1, N2, IPCO, ICCO, NRLE)
         CALL PRCHEN (I, N2, N1, IPCE, ICCE, NRLO)
      end do
    7 RETURN
      END
```

FIG. 6.4: Le nouveau sous-programme PRXPL1

Prenons comme premier exemple le sous-programme PRXPLO, dont le code est présenté dans l'Annexe F, et qui sert à initialiser les tableaux décrivant les différents raccordements dans le réseau.

Les boucles DO des lignes 77, 85 et 93 servent à initialiser les tableaux NRGR, NRCO et NRTJ, respectivement. Dans chaque boucle, le tableau NDOB peut être lu, et le tableau à modifier est certainement lu. Il est évident qu'il n'y a pas d'interaction entre ces boucles (la sortie de chacune n'est pas lue par une autre), qui sont regroupées ici simplement parce qu'elles réalisent des tâches similaires. Nous pourrions définir un sous-programme différent pour chacune de ces boucles.

Chacune des boucles DO des lignes 64 et 105 initialise deux tableaux : il s'agit des tableaux NRLO et NRLE dans le premier cas, et IPCO et IPCE dans le deuxième cas. Ces deux boucles sont des exemples d'entrelacement afin d'économiser une structure de contrôle. Chaque boucle pourrait être décomposée en deux boucles "concurrentes", sans interaction entre les deux.

La boucle DO de la ligne 114 est aussi un exemple d'entrelacement afin d'économiser une structure de contrôle. Elle pourrait être décomposée en deux boucles "concurrentes" et sans interaction, dont chacune duplique les modifications des variables N1 et N2 mais comporte un seul des appels à

```
C       --- ---------------------------------------------------------- ---
C       --- CALCUL DE NRGR                                             ---
C       --- ---------------------------------------------------------- ---
      SUBROUTINE PRXPL2(NDOB,NRGR)
      ...
      COMMON/DMFX/  NXL,NL,NXP,NXJ,NT,NC,NG,NBY,NS,JOUR(5)
      DIMENSION NDOB(*),NRGR(*)

C       --- ---------------------------------------------------------- ---
C       ---  ON ELIMINE LES ELEMENTS N'APPARTENANT PAS A LA COMPOSANTE ---
C       ---  CONNEXE PRINCIPALE (NDOB=0)                               ---
C       --- ---------------------------------------------------------- ---
      do I = 1, NG
         K1 = NRGR(I)
         if (K1.GT.0) then
           IF (NDOB(K1).LE.0) NRGR(I) = -1
         else
           NRGR(I) = 0
         endif
      end do
      RETURN
      END
```

FIG. 6.5: Le nouveau sous-programme `PRXPL2` (`PRXPL3` et `PRXPL4` sont similaires)

> `PRCHEN`. Toutefois, ces deux boucles lisent les valeurs de `NRLO` et `NRLE` initialisées dans celle de la ligne 64.
>
> Prenons comme deuxième exemple le sous-programme `PRCHEN` après la restructuration présentée dans la Figure 6.2. Ce sous-programme a une structure de contrôle compliquée qui a pour effet de lire les variables IL, N1, N2, IPCO, ICCO, NRLE et de modifier soit le seul tableau `ICCO`, soit les deux tableaux `IPCO` et `ICCO`. Il serait possible de définir deux sous-programmes, dont l'un modifie `IPCO` si besoin et l'autre modifie `ICCO`, mais ce deuxième sous-programme prendrait en entrée la valeur d'`IPCO` **avant modification**. Comme nous avons expliqué dans le Chapitre 2, les deux tableaux implémentent ensemble une table de listes de numéros de ligne, alors dans ce cas il est normal qu'un sous-programme modifie les deux.
>
> Prenons comme troisième exemple le sous-programme `CAPIV` après restructuration présenté dans la Figure C.3 de l'Annexe C. La structure de contrôle est une boucle `while` dans laquelle les tableaux `EHA` et `NCHA` et les scalaires `PIVO` et `IX` sont modifiés. Il s'agit d'un autre exemple d'entrelacement afin d'économiser une structure de contrôle car la modification de `PIVO` peut être extraite dans un sous-programme spécifique. Cependant, les modifications des trois autres variables sont plus difficiles à séparer, car à chaque itération de la boucle, la valeur (éventuellement modifiée) d'`IX` est employée comme indice dans les modifications des tableaux.

Nous proposons l'heuristique suivante afin de détecter des cas d'entrelacement et autres regroupements de traitements de variables différentes qui pourraient être séparés. Il s'agit de décomposer le sous-programme en unités fonctionnelles, chacune avec un nombre minimal de paramètres de sortie, pouvant être exécutées en concurrence. C'est-à-dire que nous pouvons reproduire les effets de l'exécution du sous-programme initial en exécutant les sous-programmes composants **dans n'importe quel ordre**. Évidemment, nous n'avons pas l'intention de les exécuter en parallèle, mais ceci est un gage de leur indépendance. En effet, nous supposons que s'il est impossible de séparer en deux traitements concurrents une modification de deux variables par le même sous-programme, alors les deux variables peuvent servir à réaliser ensemble une structure de données. Le point faible le plus évident de cette heuristique est le cas d'une structure de données réalisée par plusieurs tableaux avec une opération d'initialisation qui consiste à remettre **indépendamment** à zéro tous les tableaux. Notre heuristique conduit à la décomposition inutile de ce genre de sous-programme en l'initialisation de chaque **composant** de la structure de données.

Nous proposons de commencer par trouver pour chaque *vrai paramètre de sortie* du sous-programme (voir Section 5.2.3 du Chapitre 5), son "slice" arrière [Wei84] du sous-programme, c'est-à-dire : l'extrait du sous-programme qui permet de calculer juste la valeur de ce paramètre. Toutefois, nous proposons

```
C       --- ------------------------------------------------------------ ---
C       --- INTRODUCTION D'UN ELEMENT DE MATRICE J DU A UN ELEMENT D'INJECTION
C       --- ------------------------------------------------------------ ---
      subroutine CAPIV(IX,PIVO,NOUV,IPCO,ICCO,NRLX,NCHA,EHA,ZIMP,ATG,SUSL)
C
C.. Formal Arguments ..
      integer IX,NOUV,IPCO(*),ICCO(*),NRLX(*),NCHA(*)
      real PIVO,EHA(*),ZIMP(*),ATG(*),SUSL(*)
C
C.. Local Scalars ..
      integer KCH,NHK,NHKD
      real ELM
C
C.. Intrinsic Functions ..
      intrinsic COS
C
C.. Common Blocks ..
      ...
      common /COMSOL/ SPGM,KER,IBT,BF4L,B4TROP
      ...
      call CAPIV1(PIVO,NOUV,IPCO,ICCO,ZIMP,ATG,SUSL)
      call CAPIV2(IX,NOUV,IPCO,ICCO,NRLX,NCHA,EHA,ZIMP,ATG)
      end
```

FIG. 6.6: Décomposition de `CAPIV` : 1ère phase

```
C       --- ------------------------------------------------------------ ---
C       --- CALCUL DE PIVO
C       --- ------------------------------------------------------------ ---
      subroutine CAPIV1(PIVO,NOUV,IPCO,ICCO,ZIMP,ATG,SUSL)
C
C.. Formal Arguments ..
      integer NOUV,IPCO(*),ICCO(*)
      real PIVO,ZIMP(*),ATG(*),SUSL(*)
C
C.. Local Scalars ..
      integer KCH
      real ELM
C
C.. Intrinsic Functions ..
      intrinsic COS
C
C.. Common Blocks ..
      ...
      common /COMSOL/ SPGM,KER,IBT,BF4L,B4TROP
      ...
      KCH = IPCO(NOUV)
      do while (KCH .gt. 0)
        ELM = COS(ATG(KCH)) / ZIMP(KCH)
        PIVO = PIVO - ELM
C       --- ------------------------------------------------------------ ---
C       PRISE EN COMPTE DU TERME DU A LA 1/2 SUSCEPTANCE DANS LES CALCULS
C       EN REACTIF
C       --- ------------------------------------------------------------ ---
        if (IBT .eq. 2) then
          PIVO = PIVO + 2*SUSL(KCH)
        end if
        KCH = ICCO(KCH)
      end do
C     --- ------------------------------------------------------------ ---
C               IL N'Y A PLUS D'ARC AYANT SON ORIGINE EN NOUV
C     --- ------------------------------------------------------------ ---
      RETURN
      end
```

FIG. 6.7: Le nouveau sous-programme `CAPIV1`

```
C       --- ----------------------------------------------------------- ---
C       --- CALCUL DE IX,EHA,NCHA
C       --- ----------------------------------------------------------- ---
      subroutine CAPIV2(IX,NOUV,IPCO,ICCO,NRLX,NCHA,EHA,ZIMP,ATG)
C
C.. Formal Arguments ..
      integer IX,NOUV,IPCO(*),ICCO(*),NRLX(*),NCHA(*)
      real EHA(*),ZIMP(*),ATG(*)
C
C.. Local Scalars ..
      integer KCH,NHK,NHKD
      real ELM
C
C.. Intrinsic Functions ..
      intrinsic COS
      ...
      NHKD = 0
      KCH = IPCO(NOUV)
      do while (KCH .gt. 0)
        ELM = COS(ATG(KCH)) / ZIMP(KCH)
C       --- ----------------------------------------------------------- ---
C       PRISE EN COMPTE DU TERME DU A LA 1/2 SUSCEPTANCE DANS LES CALCULS
C       EN REACTIF
C       --- ----------------------------------------------------------- ---
        NHK = NRLX(KCH)
C       --- ----------------------------------------------------------- ---
C                           SOMMET DEJA VU
C       --- ----------------------------------------------------------- ---
        if (NHK .eq. NHKD) then
          EHA(IX) = EHA(IX) + ELM
        else
          IX = IX + 1
          EHA(IX) = ELM
          NCHA(IX) = NHK
          NHKD = NHK
        endif
        KCH = ICCO(KCH)
      end do
C     --- ----------------------------------------------------------- ---
C               IL N'Y A PLUS D'ARC AYANT SON ORIGINE EN NOUV
C     --- ----------------------------------------------------------- ---
      RETURN
      end
```

FIG. 6.8: Le nouveau sous-programme CAPIV2

d'unir les slices de deux paramètres de sortie du même sous-programme dans les cas suivants :

1. le "slice" d'un paramètre comporte une *modification* d'un autre paramètre du même sous-programme ou
2. la valeur (en entrée ou en sortie) d'un paramètre de sortie peut être *utilisée* dans le "slice" d'un autre paramètre du même sous-programme.

Nous proposons de créer un nouveau sous-programme correspondant à chaque "slice" résultant (sauf dans le cas d'un seul slice résultant qui contient tout le sous-programme initial). Chaque nouveau sous-programme a donc comme paramètres un sous-ensemble des paramètres du sous-programme dont il est extrait.

> Le résultat de l'application de cette transformation à PRXPL0 (notre premier exemple ci-dessus) est de créer les nouveaux sous-programmes PRXPL1, PRXPL2, PRXPL3 et PRXPL4 présentés dans les Figures 6.4 et 6.5 et de remplacer le code de PRXPL0 présenté dans l'Annexe F par des appels à ces nouveaux sous-programmes, comme illustré par la Figure 6.3. Notons que le nouveau sous-programme PRXPL1 créé par notre heuristique (et présenté dans la Figure 6.4) regroupe trois traitements successifs que nous allons finalement spécifier par des opérations différentes :
>
> – des modifications de NRLO et NRLE,

- l'initialisation à 0 d'IPCO et IPCE
- et l'utilisation de ces modifications dans des modifications ultérieures d'IPCO, IPCE, ICCO et ICCE.

Notre heuristique ne permet pas de départager ces traitements successifs des mêmes paramètres du sous-programme. Toutefois, nous verrons que dans ce cas, la phase suivante de décomposition des sous-programmes les départage.

Notre heuristique ne modifie pas PRCHEN.

Le résultat pour CAPIV est de remplacer le code présenté dans la Figure C.3 de l'Annexe C par celui présenté dans 6.6, qui contient des appels aux nouveaux sous-programmes CAPIV1 et CAPIV2 présentés dans les Figures 6.7 et 6.8.

Dans une approche similaire, [LD99] emploient, pour décider de la décomposition d'un sous-programme, un jeu de règles heuristiques visant à mesurer la *cohésion* des paramètres de sortie.

Extraction des boucles

La deuxième phase de décomposition consiste à créer un nouveau sous-programme pour chaque boucle dans le code Fortran. En effet, comme nous l'avons expliqué dans le Chapitre 3 (Section 3.4), chaque boucle est spécifiée par une opération spécifique. Les boucles DO sont faciles à délimiter, et nous pouvons nous appuyer sur l'analyse automatique pour décider leurs "paramètres" d'entrée et de sortie. Par contre, en ce qui concerne les boucles réalisées par les GOTO, sans l'aide de l'analyse automatique il peut être difficile de cerner, non seulement les "paramètres" mais aussi l'étendue de la boucle.

Cette phase de décomposition peut être suivie d'une autre itération de la première phase sur les paramètres de sortie des nouveaux sous-programmes.

Cette transformation extrait les trois boucles du sous-programme PRXPL1 (créé par la première phase de transformation et présenté dans la Figure 6.4) et les encapsulent dans trois nouveaux sous-programmes. Cependant, l'application de la première phase de transformation à ces nouveaux sous-programmes a pour effet de départager les modifications de données différentes effectuées "en parallèle" par chaque boucle. C'est pourquoi nous présentons, dans la Figure 6.9, une nouvelle version de PRXPL1 faisant appel aux six nouveaux sous-programmes présentés dans les Figures 6.10, 6.11 et 6.12. Il faudra attendre l'inférence des profils avant de pouvoir "factoriser" les nouveaux sous-programmes PRXPL5 et PRXPL6 en deux appels d'un même sous-programme. Il en est de même pour PRXPL7 et PRXPL8, ainsi que PRXPL9 et PRXPL0.

Nous présentons dans la Figure 6.13 le résultat de l'encapsulation (dans le nouveau sous-programme PRCHE1) de la boucle while de la version de PRCHEN après restructuration présentée dans la Figure 6.2.

Cette transformation crée aussi deux nouveaux sous-programmes encapsulant les boucles while de CAPIV1 et de CAPIV2 (que nous ne présentons pas ici).

```
C        --- ----------------------------------------------------------- ---
C        --- CALCUL D IPCO,IPCE,ICCO,ICCE,NRLO,NRLE                      ---
C        --- ----------------------------------------------------------- ---
      SUBROUTINE PRXPL1(IPCO,IPCE,ICCO,ICCE,NDOB,NRLO,NRLE)
      ...
      COMMON/DMFX/  NXL,NL,NXP,NXJ,NT,NC,NG,NBY,NS,JOUR(5)

      DIMENSION NRLO(*),NRLE(*),IPCO(*),IPCE(*),ICCO(*),ICCE(*),NDOB(*)

      CALL PRXPL5(NDOB,NRLO)
      CALL PRXPL6(NDOB,NRLE)
      CALL PRXPL7(IPCO)
      CALL PRXPL8(IPCE)
      CALL PRXPL9(IPCO,ICCO,NRLO,NRLE)
      CALL PRXPL0(IPCE,ICCE,NRLE,NRLO)
      RETURN
      END
```

FIG. 6.9: Décomposition de PRXPL1

```
C       --- ----------------------------------------------------------- ---
C       --- BOUCLE CALCUL DE NRLO                                        ---
C       --- ----------------------------------------------------------- ---
      SUBROUTINE PRXPL5(NDOB,NRLO)
      ...
      COMMON/DMFX/  NXL,NL,NXP,NXJ,NT,NC,NG,NBY,NS,JOUR(5)
      DIMENSION NDOB(*),NRLO(*)

C       --- ----------------------------------------------------------- ---
C       ---  ON ELIMINE LES ELEMENTS N'APPARTENANT PAS A LA COMPOSANTE  ---
C       ---  CONNEXE PRINCIPALE (NDOB=0)                                ---
C       --- ----------------------------------------------------------- ---
      do I = 1, NL
         K1 = NRLO(I)
         if (K1.GT.0) then
           IF (NDOB(K1).LE.0) NRLO(I) = -1
         else
           NRLO(I) = 0
         endif
      end do
      RETURN
      END
```

FIG. 6.10: Le nouveau sous-programme PRXPL5 (PRXPL6 est similaire)

```
C       --- ----------------------------------------------------------- ---
C       --- BOUCLE INIT D IPCO                                           ---
C       --- ----------------------------------------------------------- ---
      SUBROUTINE PRXPL7(IPCO)
      ...
      COMMON/DMFX/  NXL,NL,NXP,NXJ,NT,NC,NG,NBY,NS,JOUR(5)

      DIMENSION IPCO(*)

C       --- ----------------------------------------------------------- ---
C       ---                    INITIALISATION                            ---
C       --- ----------------------------------------------------------- ---
      do I = 1, NS
         IPCO(I) = 0
      end do
      RETURN
      END
```

FIG. 6.11: Le nouveau sous-programme PRXPL7 (PRXPL8 est similaire)

```
C       --- ----------------------------------------------------------- ---
C       --- BOUCLE CALCUL D IPCO,ICCO                                    ---
C       --- ----------------------------------------------------------- ---
      SUBROUTINE PRXPL9(IPCO,ICCO,NRLO,NRLE)
      ...
      COMMON/DMFX/  NXL,NL,NXP,NXJ,NT,NC,NG,NBY,NS,JOUR(5)

      DIMENSION NRLO(*),NRLE(*),IPCO(*),ICCO(*)

C       --- ----------------------------------------------------------- ---
C       ---                CONSTRUCTION DES CHAINAGES                    ---
C       --- ----------------------------------------------------------- ---
      do I=1,NL
         N1 = NRLO(I)
         N2 = NRLE(I)
         CALL PRCHEN (I, N1, N2, IPCO, ICCO, NRLE)
      end do
    7 RETURN
      END
```

FIG. 6.12: Le nouveau sous-programme PRXPL9 (PRXPL0 est similaire)

```
C     --- ------------------------------------------------------------ ---
C     --- INTRODUCTION D'UN ARC DANS UN CHAINAGE D'ARCS A LEURS ORIGINE ---
C     --- ------------------------------------------------------------ ---
      SUBROUTINE PRCHEN(IL,N1,N2,IPCO,ICCO,NRLE)
      ...
      DIMENSION IPCO(*), ICCO(*), NRLE(*)

      KCH = IPCO(N1)
      if ((N1.GT.0).AND.(N2.GT.0)) then
C       --- ------------------------------------------------------- ---
C       ---       ARC RACCORDANT DEUX SOMMETS OBSERVABLES           ---
C       ---       INITIALISATION , ENTREE DANS LA CHAINE            ---
C       --- ------------------------------------------------------- ---
        if ((KCH.GT.0).AND.(NRLE(KCH).LT.N2) then
C         --- ----------------------------------------------------- ---
C         ---                CHAINE NON-VIDE                        ---
C         --- ----------------------------------------------------- ---
          KP  = KCH
C         --- ----------------------------------------------------- ---
C         ---     CLASSEMENT DES ARCS PAR LEUR SOMMETS EXTREMITES   ---
C         --- ----------------------------------------------------- ---
          CALL PRCHE1(IL,N2,KP,ICCO,NRLE)
          KCH  = ICCO(KP)
          ICCO(KP) = IL
        else
C         --- ----------------------------------------------------- ---
C         ---              CAS D UN DEBUT DE CHAINE                 ---
C         --- ----------------------------------------------------- ---
          IPCO(N1) = IL
        endif
        ICCO(IL) = KCH
      endif
      RETURN
      END

C     --- ------------------------------------------------------------ ---
C     --- BOUCLE DE PRCHEN                                             ---
C     --- ------------------------------------------------------------ ---
      SUBROUTINE PRCHE1(IL,N2,KP,ICCO,NRLE)
      ...
      DIMENSION ICCO(*), NRLE(*)

      KCH  = ICCO(KP)
      do while ((KCH.GT.0).AND.(NRLE(KCH)).LT.N2))
        KP  = KCH
        KCH  = ICCO(KCH)
      end do
      RETURN
      END
```

FIG. 6.13: Décomposition finale de `PRCHEN`

Notons que la décomposition automatique que nous proposons ici n'élimine pas de la spécification concrète les opérations qui effectuent plusieurs traitements (non récursifs) de suite sur les mêmes données : c'est au rétro-ingénieur de décider si une telle opération devrait être décomposée en d'autres opérations élémentaires dont elle représente la composition.

6.4 Distinguer les valeurs successives d'une variable

Le problème

À une variable des spécifications algébriques est associée une seule valeur, mais une variable du code Fortran dénote une adresse mémoire où peuvent être stockées plusieurs valeurs successives. Or certaines analyses effectuées sur le code Fortran ne donneront pas de bons résultats si nous confondons les valeurs successives qu'une variable peut prendre.

Il est sûr que la tâche du programmeur est facilitée si chaque variable a un sens clair, qui ne change pas. Cependant, dans le but d'économiser la place mémoire, il peut employer une même variable pour stocker des valeurs intermédiaires dans le calcul d'une valeur définitive, ou d'autres valeurs successives qui représentent à peu près la même chose, mais de façon différente. Parfois une variable locale "de travail" est employée tout simplement comme endroit de stockage de valeurs intermédiaires de calcul différents. Un cas spécial est présenté par les variables des différentes boucles `DO` d'un même sous-programme qui portent souvent le même identificateur (`I`, par exemple).

Notons que nous allons effectuer des analyses statiques, et il n'est donc pas question de distinguer les valeurs différentes affectées à une même variable lors des itérations successives d'une même boucle. Toutefois, nous pourrions envisager de distinguer la valeur de la variable à l'entrée de la boucle et celle à la sortie.

> Prenons comme premier exemple les tableaux `IPCO` paramètres formels des sous-programmes `PRXPLO`, `PRCHEN`, ... et alias dynamiques, ainsi que les tableaux `ICCO` paramètres formels des mêmes sous-programmes et alias dynamiques. Par abus de langage et pour simplifier la discussion, nous parlerons comme de deux seules variables "le tableau `IPCO` " et "le tableau `ICCO` ". Comme expliqué dans le Chapitre 2, ces tableaux servent à stocker des listes de numéros de ligne. Le tableau `IPCO` est indexé par les numéros de sommet et stocke des pointeurs vers des listes chaînées stockées dans `ICCO`. Un élément valant 0 dans `IPCO` représente une liste vide.
>
> La boucle `DO` de `PRXPLO` isolée dans le nouveau sous-programme `PRXPL7` (voir la Figure 6.11) initialise à 0 tous les éléments d'`IPCO` indexés par un numéro de sommet. Dans chaque itération de la boucle `DO` suivante (isolée dans le nouveau sous-programme `PRXPL9`, voir Figure 6.12), les deux tableaux sont passés vers `PRCHEN`, qui peut les modifier, remplaçant notamment une valeur de 0 dans `IPCO` par une valeur d'indice d'`ICCO`. À la fin de cette deuxième boucle `DO`, les deux tableaux ont leurs valeurs définitives.
>
> À tout moment de l'exécution après la sortie de la première boucle `DO` ci-dessus, la valeur des deux tableaux peut être interprétée de la même façon comme une table de listes de numéros de ligne. La valeur à la sortie de la première boucle `DO` représente évidemment le cas spécial de la table vide, et la valeur à la sortie de la deuxième boucle `DO` le cas spécial de la table entièrement remplie. Cependant, les mêmes opérations sont appliquées à toutes les valeurs successives de la paire de tableaux et nous pouvons leur donner la même sorte. Par conséquent, confondre les valeurs successives ne nuit pas aux résultats des analyses statiques dans ce cas.
>
> Prenons comme deuxième exemple les tableaux `IPTAI` paramètres formels des sous-programmes `CACONT`, `CAISO`, `CAEDI`, ... et alias dynamiques. La valeur d'`IPTAI` à la sortie de `CACONT` est lue par `CAISO` et ensuite passée à `CAEDI`. Nous parlerons comme d'une seule variable "le tableau `IPTAI` ".
>
> Le tableau `IPTAI` est indexé par les numéros des sommets du réseau et, dans sa forme définitive, sert à stocker des pointeurs vers les informations sur les "télé-mesures" relatives à chaque sommet. Ces informations étant stockées dans le tableau `NTAI` (plus exactement, les tableaux `NTAI` paramètres formels de `CAISO`, ... et alias dynamiques) indexé par les numéros de télé-mesure. La valeur définitive d'`IPTAI` est le résultat de plusieurs calculs successifs pour prendre en compte les différentes télé-mesures rentrant dans l'"injection" pour chaque sommet. Ces calculs sont effectués par des boucles `DO` successives, d'abord dans `CACONT` et ensuite dans `CAEDI`. Les valeurs intermédiaires sont successivement stockées dans `IPTAI`.
>
> La première boucle `DO` de `CACONT` initialise à 0 ou à -1 tous les éléments d'`IPTAI` indexés par un numéro de sommet. Ensuite plusieurs boucles successives
>
> – modifient certains de ces éléments sans les lire et
> – utilisent certains éléments mais ne les modifient pas et
> – utilisent certains éléments et les modifient ensuite.
>
> Une boucle ultime utilise tous les éléments indexés par un numéro de sommet. Pendant cette phase du calcul, la valeur des éléments indexés par un numéro de sommet représente
>
> – une injection non définie si l'élément vaut -1
> – une injection nulle si l'élément vaut 0
> – une injection forcée à zéro si l'élément vaut la constante `I7F7F`
> – sinon, le nombre de télé-mesures rentrant dans l'injection du sommet.

Ensuite, une autre boucle `DO` sert à modifier la façon dont les valeurs sont codées. À la suite de cette boucle, la valeur des éléments indexés par un numéro de sommet représente

- une injection non définie si l'élément vaut -1
- une injection nulle si l'élément vaut 0
- une injection forcée à zéro si l'élément vaut la constante `I7F7F`
- sinon, le nombre cumulé de télé-mesures rentrant dans les injections des sommets, du premier jusqu'à celui-ci (et donc un pointeur vers le début d'une plage du tableau `NTAI`).

Une instruction supplémentaire met la valeur de l'élément indexé par le nombre total de sommets $+1$, au nombre total des télé-mesures pris en compte (et donc l'adresse de fin de l'ultime plage de `NTAI`).

Dans `CAEDI`, tous les éléments d'`IPTAI` indexés par un numéro de sommet sont lus et ensuite ceux qui valaient `I7F7F` sont mis à 0. C'est la valeur définitive du tableau dans laquelle

- la valeur des éléments indexés par un numéro de sommet représente
  - une injection non définie si l'élément vaut -1
  - une injection nulle si l'élément vaut 0
  - sinon, le nombre cumulé de télé-mesures rentrant dans les injections des sommets, du premier jusqu'à celui-ci (et donc un pointeur vers le début d'une plage du tableau `NTAI`).
- et l'élément suivant vaut le nombre total des télé-mesures rentrant dans les injections de tous les sommets (et donc l'adresse de fin de l'ultime plage de `NTAI`).

Or pour les analyses ultérieures d'inférence des sortes pour les profils ou de détection de type abstrait de données (décrites dans la suite de cette partie), il est important de distinguer les valeurs d'`IPTAI` qui ont des significations différentes. Il faut donc distinguer surtout les valeurs où un élément indique le nombre de télé-mesures relatives à ce sommet, et celles où il indique un nombre cumulé.

Par contre, confondre les valeurs successives des premières boucles de `CACONT` ne nuit probablement pas aux résultats de ces analyses. Toutefois, il est difficile d'imaginer comment nous pourrions détecter de façon automatique le fait que la valeur ultime donnée à `IPTAI` dans `CACONT` est fondamentalement différente de celles d'avant.

"SSA form"

Or, il existe une transformation de programmes que nous pourrions appliquer avant l'analyse statique et dans laquelle les variables sont renommées de façon à assurer que les points de *modification* possibles sont les mêmes pour tout point d'*utilisation* d'une même variable. Il s'agit du *"Single Static Assignment form"* ("SSA form") [CFR+91].

La technique s'appuie sur l'ajout, pour chaque variable V, de "pseudo affectations" d'une "ϕ-function" des différentes modifications possibles à certains points dans le code (voir l'exemple de code Fortran dans la Figure 6.14 et ce code mis sous "SSA form" dans la Figure 6.15). Une "ϕ-function" est une fonction inconnue jusqu'à l'exécution du programme et dénotée par la variable ϕ. La valeur de la "ϕ-function" ne nous intéresse pas, l'important est qu'elle représente un ensemble unique de points de modification possibles. Les occurrences de V dans le code sont ensuite renommées pour assurer qu'à chaque point d'utilisation correspond un seul point de modification (de "ϕ-function" ou non).

La technique consiste donc à effectuer les étapes suivantes pour chaque variable V :

1. décider tous les points d'insertion de "pseudo-affectation",
2. donner un nouveau nom unique (disons un indice unique au nom initial), à chaque point de modification et à chaque pseudo-affectation,
3. à chaque point d'insertion décidé pour V (renommé V_n à ce point), insérer la "pseudo-affectation" de la "ϕ-function" des modifications différentes de V (modifications renommées V_i, V_j, ...) arrivant à ce point par les différents chemins d'exécution et
4. à chaque point d'utilisation, renommer V en V_n : le seul point de modification ou pseudo-affectation l'atteignant.

```
      DO 10 I = 1, 5
        IF (I + L = J) L = L + 4
   10 CONTINUE
```

FIG. 6.14: Fragment de code employant des scalaires uniquement

$L_0 = \ldots$
```
      DO 10 I = 1, 5
```
$L_1 = \phi(L_0, L_3)$
```
        IF (I + L1 = J ) L2 = L1 + 4
```
$L_3 = \phi(L_1, L_2)$
```
   10 CONTINUE
```

FIG. 6.15: Code employant des scalaires mis sous "SSA form"

```
      L0 = ...
      DO 10 I = 1, 5
        IF (I = 1) THEN
          L1 = L0
        ELSE
          L1 = L3
        ENDIF
        IF (I + L1 = J) L2 = L1 + 4
        IF (I + L1 = J) THEN
          L3 = L2
        ELSE
          L3 = L1
        ENDIF
   10 CONTINUE
```

FIG. 6.16: Code employant des scalaires avec renommage des variables

La mise sous SSA form n'est pas une transformation du code Fortran vers un code Fortran équivalent, car les "ϕ-function" n'existent évidemment pas en Fortran. L'analyse statique peut être adaptée pour analyser cette forme intermédiaire du code et prendre en compte les "ϕ-function" ou sinon, chaque pseudo-affectation d'une "ϕ-function" peut être remplacée dans le code par plusieurs affectations normales, une à la fin de chaque branche menant au point d'insertion de la pseudo-affectation (comme montré pour notre exemple de code dans la Figure 6.16). Ceci donne un code équivalent au code initial, mais qu'il faut analyser avec précaution.

Dans [CFR+91], on propose de traiter chaque *modification* d'un élément de tableau comme une *modification* du tableau entier précédée par une *utilisation* cachée, car les valeurs des autres éléments sont "copiées" dans la nouvelle valeur du tableau. Un exemple de code Fortran employant un tableau est donné dans la Figure 6.17 et sa mise sous "SSA form" dans la Figure 6.18. [KS98], par contre, propose de faciliter la parallélisation de codes Fortran à l'aide d'une extension (appelée "Array SSA form") de la notion de "ϕ-function" permettant de traiter une *modification* d'un élément de tableau comme une *modification* du tableau entier qui, contrairement au cas des scalaires, ne *tue* pas la *modification* précédente mais est à combiner avec elle. Le résultat de la mise sous "Array SSA form" de notre exemple de code est présenté dans la Figure 6.19.

La notion de *région*, qui consiste à traiter les *modifications* d'éléments d'un même tableau effectuées au sein d'une boucle - ou autre fragment de code - comme des *modifications* d'une *région* du tableau, est orthogonale à celle du "SSA form". Il nous semble que la meilleure façon d'étendre la mise sous "SSA form" à la notion de région serait d'effectuer une première analyse afin de détecter les régions, suivie par la mise sous "SSA form" (en adoptant l'extension aux tableaux proposée dans [CFR+91]) afin de distinguer les valeurs successives d'une même région. Toutefois, la difficulté, expliquée dans le Chapitre 5, à comparer les régions (définies de façon très contextuelle) entre elles, aurait pour conséquence une multiplicité de régions apparemment différentes, chacune avec peu de valeurs successives.

Nous pourrions envisager alors de transformer tout le code à analyser en "SSA form" étendu aux régions de tableaux. Comme illustré par les exemples précédents, le coût de la détection des quelques

```
   DO 10 I = 1, 5
     IF (L(I) = J) L(I) = 0
10 CONTINUE
```

FIG. 6.17: Fragment de code employant le tableau L

```
   L0 = ...
   DO 10 I = 1, 5
     L1 = φ(L0, L3)
     IF (L1(I) = J ) L2 = Update(L1, I, 0)
     L3 = φ(L1, L2)
10 CONTINUE
```

FIG. 6.18: Code employant un tableau mis sous “SSA form”

```
   L0 = ...
   DO 10 I = 1, 5
     L1 = φ(L0, L4)
     IF (L1(I) = J ) THEN
       L2(I) = 0
       L3 = φ(L1, L2)
     ENDIF
     L4 = φ(L1, L3)
10 CONTINUE
```

FIG. 6.19: Code employant un tableau mis sous “Array SSA form”

cas où les valeurs successives d'une même variable changent réellement de sens, est le renommage inutile dans les nombreux cas où la variable garde en fait le même sens.

Autre solution

Une autre approche serait d'adapter les analyses proposées dans la suite de cette partie afin de différencier “au vol” les valeurs successives d'une même variable.

Conclusion

Nous avons proposé dans ce chapitre diverses transformations du code Fortran à effectuer avant les ultimes analyses pour l'élaboration de la spécification concrète. En ce qui concerne l'abstraction de la simulation de l'allocation dynamique de mémoire décrite dans la Section 6.1.4 et la mise sous “SSA form” décrite dans la Section 6.4, il s'agissait d'une discussion des de l'intérêt de transformations possibles, mais que nous n'avons pas effectuées sur des exemples de code réel. Nous avons effectué sur l'exemple principal de cette thèse toutes les transformations proposées dans ce chapitre sauf ces deux dernières, et c'est le résultat des transformations effectuées qui est spécifié dans la spécification concrète décrite dans le Chapitre 9.

Chapitre 7

Inférence des profils

Résumé

Dans ce chapitre, nous décrivons comment nous proposons d'inférer les *profils* des *prédicats* et *opérations* spécifiant les sous-programmes du code Fortran. Notre inférence est basée sur l'analyse des *alias dynamiques* entre *vrais paramètres* formels et effectifs de chaque appel de sous-programme, les paramètres étant caractérisés par des *régions*. Nous commençons par une introduction dans la Section 7.1.

Dans la Section 7.2, nous définissons la caractérisation des vrais paramètres d'entrée et de sortie des sous-programmes par des régions légèrement différentes de celles de *PIPS*.

La Section 7.3 contient la définition de la relation *dynamic_alias* entre deux régions.

Dans la Section 7.4, nous expliquons qu'en ce qui concerne les régions aux bornes exactes, les *classes d'équivalence* engendrées par cette relation regroupent des régions de la même *sorte*, et nous discutons les inférences prudentes et une heuristique à employer dans le cas de régions aux bornes sur-approximées. Les définitions des Sections 7.2, 7.3 et 7.4 sont illustrées sur un exemple restreint à un argument de certains profils de notre exemple principal.

La Section 7.5 présente tous les résultats obtenus par cette inférence pour notre exemple principal.

Dans la Section 7.6, nous expliquons et justifions certains choix et hypothèses implicites de notre analyse et dans la Section 7.7, nous décrivons le prototype réalisé comme une extension de *PIPS* qui a servi à la tester et affiner.

Finalement, dans la Section 7.8, nous comparons notre inférence des profils à des travaux similaires décrits dans la littérature.

7.1 Le problème

Une fois les transformations du code terminées, nous débutons l'élaboration de la spécification concrète. Nous allons spécifier les sous-programmes Fortran initiaux, ainsi que ceux résultant de la décomposition décrite dans le Chapitre 6 (Section 6.3). La première étape dans leur spécification est de fixer des *profils* cohérents pour l'ensemble des sous-programmes. Dans ce chapitre, nous décrivons comment inférer des profils à l'aide de l'analyse statique (voir aussi [WP99]). Cette analyse permet de décrire chaque *vrai paramètre* formel de sous-programme par une *région* et chaque *sorte* par l'ensemble de ces régions qui y appartient. Ce résultat permet donc de définir des profils cohérents mais dans lesquels les sortes sont nommées de façon arbitraire. Il permet aussi de connaître certaines des valeurs appartenant à chaque sorte, mais la **spécification** des sortes sera décrite dans le chapitre suivant. Nous ne nous occupons pas non plus pour l'instant de la nature *totale* ou *partielle* des opérations : nous décrivons comment la décider, ainsi que le *domaine* des opérations partielles, dans la suite de cette partie.

Les objectifs

Comme expliqué dans le Chapitre 3 (Section 3.6), nous proposons de ne pas employer dans les profils de la spécification des sortes correspondant aux types Fortran déclarés. Ceci parce que

– les types Fortran sont très peu nombreux,

- les déclarations de type ne sont pas toujours respectées et
- nous perdons l'information apportée par les paramètres effectifs instanciant chaque paramètre formel (voir Section 3.8 du Chapitre 3).

  Par exemple, dans *ESTIME* les numéros de sommet et les numéros de ligne appartiennent aux nombres positifs. Cependant, la spécification concrète du code est beaucoup plus claire si nous distinguons

  - les numéros de sommet
  - des numéros de ligne

  et

  - les tableaux stockant des valeurs égales à zéro ou un numéro de sommet
  - des tableaux stockant des valeurs égales à zéro ou un numéro de ligne

  La façon la plus naturelle de les distinguer dans la spécification concrète est par leur sorte.

À ce niveau de granularité des sortes, nous ne pouvons pas supposer qu'un tableau a la même sorte sur toute son étendue. Nous savons que les tableaux employés dans la simulation de l'allocation dynamique de la mémoire (voir Section 3.3.2 du Chapitre 5) stockent des informations de natures différentes, indexées de différentes façons dans les différentes "plages" du tableau. Traiter tout tableau comme une entité indivisible en ce qui concerne sa sorte conduirait à affecter la même sorte à toutes les plages d'un tel tableau et ainsi à tous les paramètres formels d'opérations pouvant être instanciés par une des plages. Nous voulons une analyse capable de mettre en évidence l'existence des plages d'un même tableau stockant des informations de nature différentes, afin de leur affecter des sortes différentes.

La décomposition des tableaux que nous proposons

Nous avons expliqué dans le Chapitre 5 (Section 5.2.1), pourquoi il peut être intéressant dans une analyse à des fins de rétro-ingénierie de manipuler non pas des éléments individuels de tableau mais toute plage de tableau traversée par une boucle. Nous proposons donc d'affecter des sortes aux variables scalaires et aux plages de tableaux

- traversées par une boucle ou
- composées d'un seul élément d'indice constant [1].

Dans le cas d'un tableau composé de plusieurs plages disjointes qui sont passées à des sous-programmes différents (et traversées par des boucles), et si le tableau entier n'est traversé par aucune boucle, nous définissons alors un paramètre, avec sa propre sorte, par plage. C'est le cas des tableaux employés dans la simulation de l'allocation dynamique de la mémoire (voir Section 3.3.2 du Chapitre 5).

Ceci implique que chaque plage de tableau traversée par une boucle et ayant sa propre sorte sera spécifiée comme une variable (en l'occurrence, un tableau) à part entière, car elle "simule" un tableau entier dans le code. Dans le cas de la simulation de l'allocation dynamique de la mémoire, le regroupement de plusieurs plages dans un même "grand" tableau ne sert qu'à économiser de l'espace mémoire et nous n'avons pas besoin de le spécifier.

Une inférence de type spécialisée

Nous proposons alors d'inférer les profils des opérations à l'aide d'une inférence de type spécialisée ne prenant en compte que les appels de sous-programme et leurs paramètres formels mais étendue aux régions de tableaux. Notre analyse ne prend pas en compte les déclarations de type Fortran. Elle est

1. Nous n'avons pas rencontré des exemples de plages composées d'un seul élément d'indice constant dans les codes analysés, alors nous ne développons pas leur traitement, mais nous pouvons imaginer que l'analyse des tableaux réalisant des `records` donneraient lieu à de telles plages.

fondée sur une analyse des tableaux suffisamment fine pour pouvoir intégrer l'analyse de *l'exploitation de l'appel par référence* discutée dans le Chapitre 5 (Section 5.2.2) [2].

La règle centrale de cette analyse est exprimée de façon intuitive comme suit :

> tous les paramètres effectifs pouvant instancier un même paramètre formel f d'un sous-programme p sont spécifiés avec la même sorte que celle de f dans le profil du prédicat ou de l'opération spécifiant p.

Nous allons étudier dans la suite de ce chapitre la mise en application de cette règle, ainsi que les hypothèses qui la justifient.

7.2 Régions pour l'inférence des profils

Les plages de tableau traversées par des boucles sont caractérisées par les *régions* de tableau trouvées par *PIPS* pour les boucles (voir le Chapitre 5). Notre inférence des profils est fondée donc sur les régions.

Nous allons étendre la notion de région aux scalaires (régions de 0 dimension), comme il est possible de faire pour les analyses de *PIPS*. Nous supposons que nous disposons de l'analyse des régions IN et OUT de *PIPS*, comportant les extensions proposées dans le Chapitre 5. Résumons ces propositions. Il faut étendre l'analyse de *PIPS* afin de :

- propager une région pour chaque référence à une variable scalaire ou à un élément de tableau, sans calculer la “summary region” (Section 5.2.1),
- rendre les régions IN et OUT comparables (voir Section 5.1.5) et permettre de comparer des régions avec prédicats de branche différents (voir Section 5.2.2),
- dans le cas d'une région OUT pour un sous-programme sans la même région IN,
  - vérifier si la région OUT est *“may”* (voir Section 4.2) ou non convexe et
  - dans le cas d'une région de tableau, vérifier que tous les paramètres effectifs sont déjà initialisés en entrée du sous-programme,

  et si c'est le cas, en ajouter une copie aux régions IN pour le sous-programme (voir Section 5.2.3)
- et distinguer les régions à bornes exactes de celles à bornes sur-approximées (voir Section 5.2.1).

Comme nous n'employons pas les “summary regions”, un même sous-programme peut avoir au point d'entrée plusieurs régions d'un même tableau pour chaque mode. Mais un autre sous-programme ne peut pas avoir de régions du `même` tableau, car chaque nom de tableau appartient à l'espace de nommage d'un seul sous-programme. Dans les figures, les noms de toutes les variables devraient être préfixés par le nom du sous-programme, mais nous avons enlevé les préfixes dans les cas où il n'y a pas d'ambiguïté, dans un souci de lisibilité.

> Nous prenons comme exemple les paramètres formels appelés `IPCO` ou `IPCE` de certains sous-programmes d'*ESTIME*. Le graphe des appels de ces sous-programmes (qui ne fait pas apparaître les appels d'autres sous-programmes) est présenté dans la Figure 7.1. Ce graphe des appels est annoté par des extraits des appels Fortran de sous-programme et des déclarations de sous-programme.
>
> Dans cet exemple, le sous-programme `PRCHEN` déclare un paramètre formel appelé `IPCO` et le sous-programme `PRXPLO` déclare un paramètre du même nom qui peut instancier celui de `PRCHEN`. Mais le tableau de `PRCHEN` n'est pas **le même** que celui de `PRXPLO` (et peut aussi être instancié par le tableau `IPCE` de `PRXPLO`).

Appelons les régions trouvées par l'analyse modifiée ainsi IN-*exact* ou OUT-*exact* en cas de bornes exactes et IN-*approx* ou OUT-*approx* en cas de bornes sur-approximées. Bien évidemment, les régions scalaires ont toujours des bornes exactes.

2. Notons que nous supposons ici que l'abstraction de la simulation de l'allocation dynamique évoquée dans la Section 6.1.4 du Chapitre 6 n'est pas effectuée sur le code analysé. Si nous pouvions être sûrs de pouvoir éliminer toutes les instances de l'exploitation de l'appel par référence de cette façon, alors une analyse plus simple que celle décrite dans ce chapitre suffirait pour inférer les profils des sous-programmes transformés. Mais il faudrait étudier d'autres codes pour juger si l'abstraction de l'exploitation de l'appel par référence est vraiment faisable dans tous les cas.

7.3 L'inférence des alias dynamiques

Décrivons maintenant, de façon précise, l'analyse statique que nous proposons.

Nous voulons inférer des profils pour un ensemble de sous-programmes, en indiquant quels arguments de sous-programmes différents ont la même sorte. Mais chaque sous-programme a son propre espace de nommage des valeurs qu'il manipule. La première étape est donc l'étude de toutes les façons de nommer dans les différents sous-programmes les valeurs qui sont passées de l'un à l'autre au cours de l'exécution du code : ce sont les alias dynamiques.

Les régions des paramètres

Nous avons déjà défini les *vrais paramètres d'entrée* (resp. *sortie*) dans le Chapitre 5 (Section 5.2.3).

Les *régions des paramètres d'entrée* (resp. *sortie*) d'un sous-programme sont toutes les régions IN-*exact* et IN-*approx* (resp. OUT-*exact* et OUT-*approx*) à son point d'entrée.

Dans le cas de plusieurs régions IN-*exact* (resp. OUT-*exact*) identiques pour le même sous-programme, nous n'en retenons qu'une, car chacune d'elles est la description exacte du même *vrai paramètre*.

Par contre, dans le cas de plusieurs régions *approx* identiques, nous les retenons toutes, car chacune d'elles peut être la description d'un vrai paramètre différent. En effet, deux références différentes à un même tableau peuvent être sur-approximées par la même région *approx*. Les deux plages de tableau caractérisées peuvent avoir des bornes différentes, voire être complètement disjointes.

Les régions des paramètres concernés par notre exemple sont présentées dans la Figure 7.2.

Notons qu'une région *approx* peut caractériser une plage de tableau qui est vide en réalité : le vrai paramètre correspondant n'existe pas. En effet, un outil d'analyse statique ne peut pas toujours déterminer avec précision si une référence est en mode IN ou OUT.

Les régions IN-*approx* de `BESTI1`, `PREFIX`, `PRXPL0` et `PRXPL1` caractérisent des plages de tableau qui n'existent en réalité que si `PRXPL9` et `PRXPL0` lisent des éléments d'`IPCO` (resp. `IPCE`) qui ne sont pas initialisés antérieurement par `PRXPL7` (resp. `PRXPL8`). Or *PIPS* ne peut pas déterminer si tel est le cas, car il ne sait pas caractériser de façon précise les plages de `PRXPL9:IPCO` et `PRXPL0:IPCE` pouvant être traversées par les boucles `DO` de `PRXPL9` et `PRXPL0` (voir Section 5.2.2 du Chapitre 5).

Les régions des paramètres globaux

Les vrais paramètres d'entrée et de sortie comprennent des (plages de) variables qui ne sont pas déclarées comme paramètres du sous-programme mais comme variables de blocs `COMMON` : nous appelons ceux-ci les *paramètres globaux*.

Notre analyse est fondée sur l'instanciation des paramètres formels des sous-programmes par les paramètres effectifs mais les paramètres globaux ne sont pas instanciés par des paramètres effectifs. En effet, une variable d'un bloc `COMMON` peut avoir des noms différents dans des sous-programmes différents, mais ceci est une instance d'*alias statique*, et non pas d'*alias dynamique* comme dans le cas des autres paramètres. En fait, un paramètre global de sous-programme peut ne pas être déclaré du tout dans les appelants immédiats. Évidemment il faut prendre en compte les alias statiques. Nous les ajoutons aux ensembles de régions représentant les sortes après l'analyse décrite ici.

Notons que ceci n'implique pas que des variables globales n'interviennent jamais dans notre analyse : un paramètre global d'un sous-programme peut aussi être paramètre **effectif** d'un autre et dans ce cas peut être inclus dans une sorte par notre analyse.

Les *régions des paramètres globaux* sont donc les régions de paramètres qui sont régions de variables déclarées comme variables de blocs `COMMON`.

Dans notre exemple les plages de `BESTI1:NA1` sont des paramètres globaux de `BESTI1` et paramètres effectifs de `PREFIX` et `CAMAT`.

```
SUB. BESTI1

  CALL PREFIX(...,NA1,...)
  SUB. PREFIX(...,NA1,...)

    CALL PRXPL0(NA1(I_IPCO),NA1(I_IPCE),NA1(I_ICCO),NA1(I_ICCE),...,NA1(I_NRLO),NA1(I_NRLE),...)
    SUB. PRXPL0(IPCO,       IPCE,       ICCO,       ICCE,       ...,NRLO,       NRLE,       ...)

      CALL PRXPL1(IPCO,IPCE,ICCO,ICCE,...,NRLO,NRLE)
      SUB. PRXPL1(IPCO,IPCE,ICCO,ICCE,...,NRLO,NRLE)

        CALL PRXPL7(IPCO)
        SUB  PRXPL7(IPCO)

        CALL PRXPL8(IPCE)
        SUB. PRXPL8(IPCE)

        CALL PRXPL9(IPCO,ICCO,NRLO,NRLE)
        SUB. PRXPL9(IPCO,ICCO,NRLO,NRLE)

          CALL PRCHEN(...,IPCO,ICCO,NRLE)
          SUB. PRCHEN(...,IPCO,ICCO,NRLE)

        CALL PRXPL0(IPCE,ICCE,NRLE,NRLO)
        SUB. PRXPL0(IPCE,ICCE,NRLE,NRLO)

          CALL PRCHEN(...,IPCE,ICCE,NRLO)
          SUB. PRCHEN(...,IPCO,ICCO,NRLE)

  CALL CAMAT(...,NA1,...)
  CALL CAMAT(...,NA1,...)
  SUB. CAMAT(...,NA1,...)

    CALL CACONT(...,NA1,NA1(I_NRLO),NA1(I_NRLE),...)
    SUB. CACONT(...,NA1,NRLO,       NRLE,       ...)

      CALL CASDEG(...,NA1(I_IPCO),NA1(I_ICCO),NRLE)
      CALL CASDEG(...,NA1(I_IPCE),NA1(I_ICCE),NRLO)
      SUB. CASDEG(...,IPCO,       ICCO,       NRLE)

    CALL CAJAC(...,NA1(I_IPCO),NA1(I_IPCE),NA1(I_ICCO),NA1(I_ICCE),NA1(I_NRLO),NA1(I_NRLE),...)
    SUB. CAJAC(...,IPCO,       IPCE,       ICCO,       ICCE,      ,NRLO,       NRLE,       ...)

      CALL CAPIV(...,IPCO,ICCO,NRLE)
      CALL CAPIV(...,IPCE,ICCE,NRLO)
      SUB. CAPIV(...,IPCO,ICCO,NRLX)

        CALL CAPIV1(...,IPCO,ICCO,...)
        SUB. CAPIV1(...,IPCO,ICCO,...)

        CALL CAPIV2(...,IPCO,ICCO,NRLX,...)
        SUB. CAPIV2(...,IPCO,ICCO,NRLX,...)
```

FIG. 7.1: Graphe d'appels annoté des sous-programmes de notre exemple

sous-prog.	mode et précision		polyèdre
BESTI1:	IN-*approx*	NA1	[19+2F_NS,7118] si 19+2F_NS ≥ 1
PREFIX:	IN-*approx*	NA1	[I_IPCO+NS,$+\infty$[si I_IPCO+NS ≥ 1
PREFIX:	IN-*approx*	NA1	[I_IPCE+NS,$+\infty$[si I_IPCE+NS ≥ 1
PREFIX:	OUT-*exact*	NA1	[I_IPCO,I_IPCO+NS-1] si I_IPCO ≥ 1
PREFIX:	OUT-*approx*	NA1	[I_IPCO,$+\infty$[si I_IPCO ≥ 1
PREFIX:	OUT-*exact*	NA1	[I_IPCE,I_IPCE+NS-1] si I_IPCE ≥ 1
PREFIX:	OUT-*approx*	NA1	[I_IPCE,$+\infty$[si I_IPCE ≥ 1
PRXPL0:	IN-*approx*	IPCO	[NS+1,$+\infty$[si NS+1 ≥ 1
PRXPL0:	IN-*approx*	IPCE	[NS+1,$+\infty$[si NS+1 ≥ 1
PRXPL0:	OUT-*exact*	IPCO	[1,NS]
PRXPL0:	OUT-*approx*	IPCO	[1,$+\infty$[
PRXPL0:	OUT-*exact*	IPCE	[1,NS]
PRXPL0:	OUT-*approx*	IPCE	[1,$+\infty$[
PRXPL1:	IN-*approx*	IPCO	[NS+1,$+\infty$[si NS+1 ≥ 1
PRXPL1:	IN-*approx*	IPCE	[NS+1,$+\infty$[si NS+1 ≥ 1
PRXPL1:	OUT-*exact*	IPCO	[1,NS]
PRXPL1:	OUT-*approx*	IPCO	[1,$+\infty$[
PRXPL1:	OUT-*exact*	IPCE	[1,NS]
PRXPL1:	OUT-*approx*	IPCE	[1,$+\infty$[
PRXPL7:	OUT-*exact*	IPCO	[1,NS]
PRXPL8:	OUT-*exact*	IPCE	[1,NS]
PRXPL9:	IN-*approx*	IPCO	[1,$+\infty$[
PRXPL9:	OUT-*approx*	IPCO	[1,$+\infty$[
PRXPLO:	IN-*approx*	IPCE	[1,$+\infty$[
PRXPLO:	OUT-*approx*	IPCE	[1,$+\infty$[
PRCHEN:	IN-*exact*	IPCO	[N1] si N1 ≥ 1
PRCHEN:	OUT-*exact*	IPCO	[N1] si N1 ≥ 1
CAMAT:	IN-*exact*	NA1	[I_IPCO,I_IPCO+NSO-1] si I_IPCO ≥ 1
CAMAT:	IN-*exact*	NA1	[I_IPCE,I_IPCE+NSO-1] si I_IPCE ≥ 1
CAMAT:	IN-*approx*	NA1	[I_IPCO,$+\infty$[si I_IPCO ≥ 1
CAMAT:	IN-*approx*	NA1	[I_IPCE,$+\infty$[si I_IPCE ≥ 1
CACONT:	IN-*exact*	NA1	[I_IPCO,I_IPCO+NSO-1] si I_IPCO ≥ 1
CACONT:	IN-*exact*	NA1	[I_IPCE,I_IPCE+NSO-1] si I_IPCE ≥ 1
CASDEG:	IN-*exact*	IPCO	[1,NSO]
CAJAC:	IN-*approx*	IPCO	[1,$+\infty$[
CAJAC:	IN-*approx*	IPCE	[1,$+\infty$[
CAPIV:	IN-*exact*	IPCO	[NOUV] si NOUV ≥ 1
CAPIV1:	IN-*exact*	IPCO	[NOUV] si NOUV ≥ 1
CAPIV2:	IN-*exact*	IPCO	[NOUV] si NOUV ≥ 1

FIG. 7.2: Régions de paramètres formels pour notre exemple

Les régions des paramètres formels

Appelons *vrais paramètres formels* les vrais paramètres d'entrée et de sortie sans les paramètres globaux.

Les *régions des paramètres formels* sont donc les régions des paramètres d'entrée et de sortie qui ne sont pas régions des paramètres globaux.

Toutes les régions présentées dans la Figure 7.2 sont des régions de paramètres formels, sauf celle de `BESTI1:NA1`.

Les régions des paramètres effectifs

Les *régions des paramètres effectifs* sont le résultat de la traduction effectuée par *PIPS* (voir Section 5.1.2 du Chapitre 5) des régions des paramètres formels dans l'espace de nommage du sous-programme appelant.

Les régions au point d'entrée des paramètres effectifs

Les *régions au point d'entrée des paramètres effectifs* sont le résultat de l'expression par *PIPS* des régions des paramètres effectifs au point d'appel du sous-programme en fonction du *store* au point d'entrée de l'appelant (voir Section 5.1.2 du Chapitre 5).

Notons le cas spécial d'un appel de sous-programme situé dans une boucle, avec un paramètre effectif qui peut varier d'itération en itération. La région du paramètre effectif caractérise le paramètre effectif pour une itération arbitraire de la boucle en fonction d'une variable locale à la boucle. Dans ce cas, *PIPS* élimine de la région les variables qui varient d'itération en itération, afin de la ré-exprimer en terme du store au point d'entrée de l'appelant. Il calcule ainsi une seule région pour la boucle entière qui contient toutes les régions des itérations individuelles. Nous allons revenir dans la suite sur le traitement de ce cas.

Dans notre exemple, la région de paramètre formel de `PRCHEN`

`PRCHEN:` IN-*exact* `IPCO`[`N1`] si `N1` ≥ 1

caractérise un seul élément d'`IPCO` en fonction de la valeur de `N1`. Il y a deux paramètres effectifs pouvant instancier ce vrai paramètre formel, l'un dans `PRXPL9` et l'autre dans `PRXPL0`. À titre d'exemple, la région au point d'appel du paramètre effectif de `PRXPL9` est :

`PRXPL9:` *exact* `IPCO`[`N1`] si `N1` ≥ 1

Le point d'appel est situé dans une boucle `DO` et `PRXPL9:N1` y est égale à la valeur de l'élément de `NRLO` indexé par la variable de boucle. Différents éléments de `PRXPL9:IPCO` peuvent donc être référencés dans les différentes itérations de la boucle.

La région du paramètre formel de `PRXPL9`:

`PRXPL9:` IN-*approx* `IPCO`$[1,+\infty[$

caractérise toutes les itérations de la boucle.

La situation est similaire en ce qui concerne

- le point d'appel dans `PRXPL0`,
- la région OUT de `PRCHEN`

  `PRCHEN:` OUT-*exact* `IPCO`[`N1`] si `N1` ≥ 1
- et la région du paramètre formel de `CAPIV`

  `CAPIV:` IN-*exact* `IPCO`[`NOUV`] si `NOUV` ≥ 1

  dont l'appel est situé dans une boucle de `CAJAC` dans laquelle la valeur instanciant `CAPIV:NOUV` varie d'itération en itération.

Les instanciations

Appelons *instanciations* toutes les paires

<région de paramètre formel,région au point d'entrée de paramètre effectif>

et définissons une relation *instantiated_by* qui lie les deux régions de chaque instanciation.

Toutefois, dans le cas spécial (évoqué dans la section précédente) d'un sous-programme dont les appels sont toujours situés dans des boucles, nous allons faire la distinction introduite dans la Section 7.1, entre

1. les régions de paramètres formels qui sont
   - soit engendrées elles-mêmes par une boucle interne
   - soit composées d'un seul élément d'indice constant
2. et celles composées d'un seul élément dont l'indice dépend des autres paramètres d'entrée du sous-programme appelé et varie avec les itérations de la boucle.

Dans le premier cas, nous considérons qu'il n'y a pas d'instanciation du paramètre formel exprimable en fonction du store au point d'entrée de l'appelant. L'effet sera de donner sa propre sorte au paramètre formel du sous-programme appelé et donc de spécifier la décomposition de la plage de tableau traversée par la boucle en plages plus petites, dont chacune est traitée par une itération de la boucle.

Dans le deuxième cas, la "plage traversée" par le sous-programme appelé consiste en un élément variable de la plage traversée par la boucle des appels. Si tous les appels du sous-programme sont situés dans des boucles traversant la même plage, nous pouvons caractériser le paramètre formel du sous-programme appelé de la même façon que la plage traversée par les appelants. L'effet sera de donner la même sorte au paramètre formel et à la plage traversée par l'appelant et de spécifier le passage de toute la plage à chaque itération de la boucle. Concrètement, nous traduisons donc la région au point d'entrée du paramètre effectif, qui caractérise la région de la boucle, dans l'espace de nommage du sous-programme appelé. Nous étendons cette traduction "en arrière" aux éventuels sous-programmes appelés par celui-ci, et ainsi de suite. Dans ce cas l'instanciation est formée

- de la "traduction en arrière" de la région du paramètre effectif et
- de la région du paramètre effectif.

> Dans notre exemple, le paramètre d'entrée de `PRCHEN` appartient au deuxième cas ci-dessus. Son *instanciation* pour l'appel depuis `PRXPL9` n'est donc pas :
>
> <`PRCHEN:` IN-*exact* `IPCO` [`N1`] si `N1` ≥ 1, `PRXPL9:` IN-*approx* `IPCO` [1,+∞[>
>
> mais :
>
> <`PRCHEN:` IN-*approx* `IPCO` [1,+∞[, `PRXPL9:` IN-*approx* `IPCO` [1,+∞[>
>
> les instanciations pour `CAPIV` sont :
>
> <`CAPIV:` IN-*approx* `IPCO` [1,+∞[, `CAJAC:` IN-*approx* `IPCO` [1,+∞[>
>
> et :
>
> <`CAPIV:` IN-*approx* `IPCE` [1,+∞[, `CAJAC:` IN-*approx* `IPCO` [1,+∞[>
>
> et l'instanciation pour `CAPIV1`:
>
> <`CAPIV1:` IN-*approx* `IPCO` [1,+∞[, `CAPIV:` IN-*approx* `IPCO` [1,+∞[>

7.3.1 La relation *dynamic_alias*

Nous définissons une relation, *dynamic_alias*, qui lie deux régions référençant la même zone en mémoire selon un chemin particulier d'exécution. Nous pouvons considérer que la "vraie" variable est la zone en mémoire et les régions liés par la relation *dynamic_alias* sont deux noms pour une "même" variable. Notons que tout comme *instantiated_by*, cette relation n'est pas symétrique.

La relation *dynamic_alias* inclut *instantiated_by*. Par ailleurs, s'il existe une instanciation $< f, e >$ pour un site d'appel du sous-programme $P3$ depuis le sous-programme $P2$, et si $P2$ est appelé par $P1$ alors le paramètre effectif pour ce site d'appel peut être aussi le paramètre **formel** dans une instanciation

sous-prog.	mode et précision		polyèdre
PRXPL1:	IN-*approx*	IPC0	[NS+1,+∞[si NS+1 ≥ 1
PRXPL0:	IN-*approx*	IPC0	[NS+1,+∞[si NS+1 ≥ 1
PREFIX:	IN-*approx*	NA1	[I_IPC0+NS,+∞[si I_IPC0+NS ≥ 1
BESTI1:	*approx*	NA1	[19+F_NS,7118] si 19+F_NS ≥ 1
PRXPL1:	IN-*approx*	IPCE	[NS+1,+∞[si NS+1 ≥ 1
PRXPL0:	IN-*approx*	IPCE	[NS+1,+∞[si NS+1 ≥ 1
PREFIX:	IN-*approx*	NA1	[I_IPCE+NS,+∞[si I_IPCE+NS ≥ 1
BESTI1:	IN-*approx*	NA1	[19+2F_NS,7118] si 19+2F_NS ≥ 1
PRXPL7:	OUT-*exact*	IPC0	[1,NS]
PRXPL1:	OUT-*exact*	IPC0	[1,NS]
PRXPL0:	OUT-*exact*	IPC0	[1,NS]
PREFIX:	OUT-*exact*	NA1	[I_IPC0,I_IPC0+NS-1] si I_IPC0 ≥ 1
BESTI1:	*exact*	NA1	[19,18+F_NS] si 18+F_NS ≤ 7118
PRXPL8:	OUT-*exact*	IPCE	[1,NS]
PRXPL1:	OUT-*exact*	IPCE	[1,NS]
PRXPL0:	OUT-*exact*	IPCE	[1,NS]
PREFIX:	OUT-*exact*	NA1	[I_IPCE,I_IPCE+NS-1] si I_IPCE ≥ 1
BESTI1:	*exact*	NA1	[19+F_NS,18+2F_NS] si 18+2F_NS ≤ 7118
PRCHEN:	IN-*approx*	IPC0	[1,+∞[
PRXPL9:	IN-*approx*	IPC0	[1,+∞[
PRXPL1:	*approx*	IPC0	[1,+∞[
PRCHEN:	OUT-*approx*	IPC0	[1,+∞[
PRXPL9:	OUT-*approx*	IPC0	[1,+∞[
PRXPL1:	OUT-*approx*	IPC0	[1,+∞[
PRXPL0:	OUT-*approx*	IPC0	[1,+∞[
PREFIX:	OUT-*approx*	NA1	[I_IPC0,+∞[si I_IPC0 ≥ 1
BESTI1:	*approx*	NA1	[19,7118]
PRCHEN:	IN-*approx*	IPC0	[1,+∞[
PRXPL0:	IN-*approx*	IPCE	[1,+∞[
PRXPL1:	*approx*	IPCE	[1,+∞[
PRCHEN:	OUT-*approx*	IPC0	[1,+∞[
PRXPL0:	OUT-*approx*	IPCE	[1,+∞[
PRXPL1:	OUT-*approx*	IPCE	[1,+∞[
PRXPL0:	OUT-*approx*	IPCE	[1,+∞[
PREFIX:	OUT-*approx*	NA1	[I_IPCE,+∞[si I_IPCE ≥ 1
BESTI1:	*approx*	NA1	[19+F_NS,7118] si 19+F_NS ≥ 1
CASDEG:	IN-*exact*	IPC0	[1,NS0]
CACONT:	IN-*exact*	NA1	[I_IPC0,I_IPC0+NS0-1] si I_IPC0 ≥ 1
CAMAT:	IN-*exact*	NA1	[I_IPC0,I_IPC0+NS0-1] si I_IPC0 ≥ 1
BESTI1:	*exact*	NA1	[19,18+F_NS] si 18+F_NS ≤ 7118
CASDEG:	IN-*exact*	IPC0	[1,NS0]
CACONT:	IN-*exact*	NA1	[I_IPCE,I_IPCE+NS0-1] si I_IPCE ≥ 1
CAMAT:	IN-*exact*	NA1	[I_IPCE,I_IPCE+NS0-1] si I_IPCE ≥ 1
BESTI1:	*exact*	NA1	[19+F_NS,18+2F_NS] si 18+2F_NS ≤ 7118
CAPIV1:	IN-*approx*	IPC0	[1,+∞[
CAPIV:	IN-*approx*	IPC0	[1,+∞[
CAJAC:	IN-*approx*	IPC0	[1,+∞[
CAMAT:	IN-*approx*	NA1	[I_IPC0,+∞[si I_IPC0 ≥ 1
BESTI1:	*approx*	NA1	[19,7118]
CAPIV1:	IN-*approx*	IPC0	[1,+∞[
CAPIV:	IN-*approx*	IPC0	[1,+∞[
CAJAC:	IN-*approx*	IPCE	[1,+∞[
CAMAT:	IN-*approx*	NA1	[I_IPCE,+∞[si I_IPCE ≥ 1
BESTI1:	*approx*	NA1	[19+F_NS),7118] si 19+F_NS ≥ 1

FIG. 7.3: Listes d'alias pour notre exemple

pour l'appel de $P2$ par $P1$. En effet, *dynamic_alias* est la fermeture transitive d'*instantiated_by*, c'est-à-dire[3] :

$$instantiated\_by(f,e) \Rightarrow dynamic\_alias(f,e) \tag{7.1}$$

$$dynamic\_alias(r1,r2) \wedge dynamic\_alias(r2,r3) \Rightarrow dynamic\_alias(r1,r3) \tag{7.2}$$

7.3.2 Les listes d'alias

Appelons *liste d'alias* une liste regroupant le plus grand nombre possible de régions de telle façon que chaque élément (sauf le premier) instancie celui se trouvant avant lui, et chaque élément (sauf le dernier) est instancié par celui après. Lors de l'appel du sous-programme le plus profond dans le graphe des appels parmi ceux concernés, tous les éléments d'une liste d'alias référencent la même zone de mémoire.

Les listes d'alias pour notre exemple sont présentées dans la Figure 7.3. Les listes sont séparées par des lignes blanches. Les listes commençant par une région de `CAPIV2` ne sont pas incluses, faute de place, mais elles sont identiques à celles commençant par une région de `CAPIV1`, sauf que `CAPIV1` est remplacé par `CAPIV2`.

Les régions ultimes de certaines listes n'ont pas de mode. C'est parce qu'elles sont des régions (au point d'entrée) de paramètre effectif mais ne sont pas des régions de paramètre formel de l'appelant. C'est-à-dire, elles ne sont ni IN ni OUT pour l'appelant.

7.4 L'inférence des sortes

Tous les alias dynamiques d'une région de paramètre formel donnée peuvent référencer la “même” variable et donc ont la même sorte, du moins en l'absence de coercitions implicites de sortes. Nous allons maintenant expliquer le regroupement des régions en ensembles de régions de la même sorte.

7.4.1 La relation *same_sort* pour les régions *exact*

Nous définissons une relation binaire et symétrique entre régions, *same_sort*.

$$same\_sort(r1,r2) \Rightarrow same\_sort(r2,r1) \tag{7.3}$$

Cette relation inclut *dynamic_alias* car deux régions liées par *dynamic_alias* représentent juste deux noms différents pour la même zone en mémoire au même moment et nous ne prenons pas en compte pour l'instant la possibilité de coercitions “implicites” de sorte.

$$dynamic\_alias(f,e) \Rightarrow same\_sort(f,e) \tag{7.4}$$

Toute paire d'éléments d'une même liste d'alias est donc liée par *same_sort*.

Si un sous-programme est appelé depuis deux sites d'appel avec deux paramètres effectifs différents, alors les deux instanciations, $< f, e1 >$ et $< f, e2 >$ du même paramètre formel, f, sont dans deux listes d'alias identiques jusqu'à l'élément f et différentes à partir de celui-ci. D'après notre règle principale d'inférence (de Section 7.1), tous les éléments de ces deux listes ont la même sorte :

$$same\_sort(f,e1) \wedge dynamic\_alias(f,e2) \Rightarrow same\_sort(e1,e2) \tag{7.5}$$

Dans notre exemple, cette règle a pour effet d'affecter la même sorte aux éléments des deux listes d'alias commençant par une région de `CASDEG`.

3. Dans ces formules de logique du premier ordre, comme dans celles qui suivent, nous supposons la quantification universelle de toutes les variables.

Par ailleurs, si deux régions *exact* aux bornes identiques instancient deux régions de paramètres formels différentes, alors il s'agit

- soit exactement du même paramètre effectif qui instancie deux paramètres formels différents
- soit exactement du même vrai paramètre formel en entrée et en sortie du sous-programme.

Nous inférons alors que tous les éléments des deux listes d'alias concernées ont la même sorte :

$$exact(e) \wedge same\_sort(f1, e) \wedge dynamic\_alias(f2, e) \Rightarrow same\_sort(f1, f2) \quad (7.6)$$

Cette règle assure la transitivité de la relation *same_sort* **dans le cas de région *exact*** . Dans ce cas, elle est réflexive aussi. En fait, deux régions *exact* identiques sauf en ce qui concerne leur mode (c'est à dire, une région IN-*exact* et une région OUT-*exact* aux mêmes bornes) ont la même sorte. Soit *same_limits* la relation entre deux régions partageant les mêmes bornes :

$$exact(r1) \wedge exact(r2) \wedge same\_limits(r1, r2) \Rightarrow same\_sort(r1, r2). \quad (7.7)$$

Sur les régions *exact*, la relation *same_sort* est donc une *relation d'équivalence* et les sortes sont juste les *classes d'équivalence.*

Dans notre exemple, la règle 7.6 a pour effet d'affecter la même sorte aux éléments de chaque paire de listes terminant par la même région *exact* de `BESTI1:NA1`.

7.4.2 La relation *same_sort* pour les régions *approx*

Les trois premières règles (7.3, 7.4 et 7.5) d'inférence de la relation *same_sort* s'appliquent aux régions *exact* et aux régions *approx*.

Toutefois, si la même région *approx* instancie deux régions de paramètres formels différentes, alors nous ne pouvons pas, en règle générale, en tirer la même conclusion que dans le cas de régions *exact* (règle 7.6). Ceci parce que deux plages du même tableau qui sont en réalité complètement disjointes peuvent être caractérisées par la même région *approx*. La relation *same_sort* n'est pas transitive dans ce cas. En effet, **une région *approx* peut avoir deux sortes différentes car elle peut caractériser deux vrais paramètres différents**.

L'exception est le cas d'une région *exact* de paramètre formel d'un sous-programme appelé dans une boucle, qui a été "élargie" à une région *approx* pour toutes les itérations de la boucle, comme expliqué dans la Section 7.3. Dans ce cas, nous savons que toutes les instances de régions *approx* dérivées de cette façon de la même région *exact* (ou de deux régions *exact* aux mêmes bornes) caractérisent le même paramètre formel. Nous pouvons conclure donc que les régions ont la même sorte. Il faudrait étendre les règles 7.6 et 7.7 de façon à prendre en compte ce cas, mais nous ne le faisons pas ici, car l'expression de la règle étendue est trop compliquée.

Dans notre exemple, nous affectons donc la même sorte aux éléments des quatre listes d'alias commençant par

`PRCHEN:` IN-*approx* `IPCO` $[1,+\infty[$

ou

`PRCHEN:` OUT-*approx* `IPCO` $[1,+\infty[$

ainsi qu'aux éléments des deux listes d'alias contenant

`CAPIV:` IN-*approx* `IPCO` $[1,+\infty[$

7.4.3 La description des sortes

Chaque sorte est décrite par le plus grand ensemble de régions tel que chaque élément est lié par la relation *same_sort* à tous les autres éléments.

Les ensembles de régions appartenant à une même sorte dans notre exemple sont présentées dans la Figure 7.4. Les ensembles de régions sont séparés par des lignes blanches.

sous-prog.	mode et précision		polyèdre
PRXPL1:	IN-*approx*	IPCO	[NS+1,$+\infty$[si NS+1 ≥ 1
PRXPLO:	IN-*approx*	IPCO	[NS+1,$+\infty$[si NS+1 ≥ 1
PREFIX:	IN-*approx*	NA1	[I_IPCO+NS,$+\infty$[si I_IPCO+NS ≥ 1
BESTI1:	*approx*	NA1	[19+F_NS,7118] si 19+F_NS ≥ 1
PRXPL1:	IN-*approx*	IPCE	[NS+1,$+\infty$[si NS+1 ≥ 1
PRXPLO:	IN-*approx*	IPCE	[NS+1,$+\infty$[si NS+1 ≥ 1
PREFIX:	IN-*approx*	NA1	[I_IPCE+NS,$+\infty$[si I_IPCE+NS ≥ 1
BESTI1:	IN-*approx*	NA1	[19+2F_NS,7118] si 19+2F_NS ≥ 1
PRXPL7:	OUT-*exact*	IPCO	[1,NS]
PRXPL1:	OUT-*exact*	IPCO	[1,NS]
PRXPLO:	OUT-*exact*	IPCO	[1,NS]
PREFIX:	OUT-*exact*	NA1	[I_IPCO,I_IPCO+NS-1] si I_IPCO ≥ 1
CASDEG:	IN-*exact*	IPCO	[1,NSO]
CACONT:	IN-*exact*	NA1	[I_IPCO,I_IPCO+NSO-1] si I_IPCO ≥ 1
CAMAT:	IN-*exact*	NA1	[I_IPCO,I_IPCO+NSO-1] si I_IPCO ≥ 1
BESTI1:	*exact*	NA1	[19,18+F_NS] si 18+F_NS ≤ 7118
PRXPL8:	OUT-*exact*	IPCE	[1,NS]
PRXPL1:	OUT-*exact*	IPCE	[1,NS]
PRXPLO:	OUT-*exact*	IPCE	[1,NS]
PREFIX:	OUT-*exact*	NA1	[I_IPCE,I_IPCE+NS-1] si I_IPCE ≥ 1
CACONT:	IN-*exact*	NA1	[I_IPCE,I_IPCE+NSO-1] si I_IPCE ≥ 1
CAMAT:	IN-*exact*	NA1	[I_IPCE,I_IPCE+NSO-1] si I_IPCE ≥ 1
BESTI1:	*exact*	NA1	[19+F_NS,18+2F_NS] si 18+2F_NS ≤ 7118
PRCHEN:	IN-*approx*	IPCO	[1,$+\infty$[
PRXPL9:	IN-*approx*	IPCO	[1,$+\infty$[
PRXPL1:	*approx*	IPCO	[1,$+\infty$[
PRXPLO:	IN-*approx*	IPCE	[1,$+\infty$[
PRXPL1:	*approx*	IPCE	[1,$+\infty$[
PRCHEN:	OUT-*approx*	IPCO	[1,$+\infty$[
PRXPL9:	OUT-*approx*	IPCO	[1,$+\infty$[
PRXPL1:	OUT-*approx*	IPCO	[1,$+\infty$[
PRXPLO:	OUT-*approx*	IPCO	[1,$+\infty$[
PREFIX:	OUT-*approx*	NA1	[I_IPCO,$+\infty$[si I_IPCO ≥ 1
BESTI1:	*approx*	NA1	[19,7118]
PRXPLO:	OUT-*approx*	IPCE	[1,$+\infty$[
PRXPL1:	OUT-*approx*	IPCE	[1,$+\infty$[
PRXPLO:	OUT-*approx*	IPCE	[1,$+\infty$[
PREFIX:	OUT-*approx*	NA1	[I_IPCE,$+\infty$[si I_IPCE ≥ 1
BESTI1:	*approx*	NA1	[19+F_NS,7118] si 19+F_NS ≥ 1
CAPIV1:	IN-*approx*	IPCO	[1,$+\infty$[
CAPIV:	IN-*approx*	IPCO	[1,$+\infty$[
CAJAC:	IN-*approx*	IPCO	[1,$+\infty$[
CAMAT:	IN-*approx*	NA1	[I_IPCO,$+\infty$[si I_IPCO ≥ 1
BESTI1:	*approx*	NA1	[19,7118]
CAJAC:	IN-*approx*	IPCE	[1,$+\infty$[
CAMAT:	IN-*approx*	NA1	[I_IPCE,$+\infty$[si I_IPCE ≥ 1
BESTI1:	*approx*	NA1	[19+F_NS,7118] si 19+F_NS ≥ 1
CAPIV2:	IN-*approx*	IPCO	[1,$+\infty$[

FIG. 7.4: Sortes pour notre exemple avec traitement prudent des régions *approx*

sous-prog.	mode et précision		polyèdre
PRXPL7:	OUT-*exact*	IPCO	[1,NS]
PRXPL1:	OUT-*exact*	IPCO	[1,NS]
PRXPL0:	OUT-*exact*	IPCO	[1,NS]
PREFIX:	OUT-*exact*	NA1	[I_IPCO,I_IPCO+NS-1] si I_IPCO ≥ 1
CASDEG:	IN-*exact*	IPCO	[1,NSO]
CACONT:	IN-*exact*	NA1	[I_IPCO,I_IPCO+NSO-1] si I_IPCO ≥ 1
CAMAT:	IN-*exact*	NA1	[I_IPCO,I_IPCO+NSO-1] si I_IPCO ≥ 1
BESTI1:	*exact*	NA1	[19,18+F_NS] si 18+F_NS ≤ 7118
PRXPL8:	OUT-*exact*	IPCE	[1,NS]
PRXPL1:	OUT-*exact*	IPCE	[1,NS]
PRXPL0:	OUT-*exact*	IPCE	[1,NS]
PREFIX:	OUT-*exact*	NA1	[I_IPCE,I_IPCE+NS-1] si I_IPCE ≥ 1
CACONT:	IN-*exact*	NA1	[I_IPCE,I_IPCE+NSO-1] si I_IPCE ≥ 1
CAMAT:	IN-*exact*	NA1	[I_IPCE,I_IPCE+NSO-1] si I_IPCE ≥ 1
BESTI1:	*exact*	NA1	[19+F_NS,18+2F_NS] si 18+2F_NS ≤ 7118
PRCHEN:	IN-*exact*	IPCO	[1,PREFIX:NS]
PRXPL9:	IN-*exact*	IPCO	[1,PREFIX:NS]
PRXPL0:	IN-*exact*	IPCE	[1,PREFIX:NS]
PRCHEN:	OUT-*exact*	IPCO	[1,PREFIX:NS]
PRXPL9:	OUT-*exact*	IPCO	[1,PREFIX:NS]
PRXPL0:	OUT-*exact*	IPCE	[1,PREFIX:NS]
CAPIV1:	IN-*exact*	IPCO	[1,CAMAT:NSO]
CAPIV:	IN-*exact*	IPCO	[1,CAMAT:NSO]
CAJAC:	IN-*exact*	IPCO	[1,CAMAT:NSO]
CAJAC:	IN-*exact*	IPCE	[1,CAMAT:NSO]
CAPIV2:	IN-*exact*	IPCO	[1,CAMAT:NSO]

FIG. 7.5: La sorte créée par l'application de l'heuristique à notre exemple

7.4.4 Une heuristique pour le traitement des régions *approx*

En supposant que chaque instance de la même région *approx* caractérise un paramètre différent, avec sa propre sorte, nous obtenons un résultat très *prudent* (dans le sens expliqué dans la Section 4.2 du Chapitre4). Nous tendons ainsi vers une explosion du nombre de sortes apparemment différentes. La seule solution possible est de chercher des *heuristiques* permettant de décider quand est-ce que deux régions qui ne sont pas toutes les deux *exact* caractérisent le même vrai paramètre. Chaque application de telles heuristiques peut être signalée au rétro-ingénieur, ce qui lui permet, pour avoir plus de confiance dans les résultats de l'analyse, de vérifier à la main l'égalité effective des deux vrais paramètres.

En effet, il semble raisonnable de supposer que la même plage de tableau peut parfois être caractérisée par une région aux bornes exactes pour un point de référence, et par une région aux bornes sur-approximées pour un autre point de référence.

L'heuristique suivante donne de bons résultats dans notre cas :

> si une région *exact* et une région *approx* du même tableau ont la même borne inférieure, et si la borne supérieure de la région *approx* est supérieure ou égale à celle de la région *exact*, alors nous supposons que les deux régions caractérisent le même plage du tableau.

Cette heuristique doit être appliquée avec beaucoup de précaution :

1. Si elle pouvait s'appliquer à plusieurs paires de régions et si la région *approx* d'une des paires résulte d'un calcul basé sur la région *approx* d'une autre paire, alors il faut l'appliquer sur les différentes paires "dans le bon ordre".

   Dans notre exemple, cette heuristique s'appliquerait

   - aux régions de `PRXPL1:IPCO` appartenant aux listes d'alias commençant par
     - la région OUT-*approx* de `PRXPL9:IPCO` ou
     - la région OUT-*exact* de `PRXPL7:IPCO`
   - et aussi aux régions de `BESTI1:NA1` appartenant aux listes d'alias commençant par
     - la région IN-*approx* de `PRXPL1:IPCO` ou
     - la région OUT-*exact* de `PRXPL8:IPCE`

   mais `PRXPL1` appelle `PRXPL9` et la région IN-*approx* de `PRXPL1:IPCO` est le résultat d'un calcul basé sur la région OUT-*approx* de `PRXPL9:IPCO`. L'application de l'heuristique à la première paire de régions ci-dessus fait disparaître la région *approx* de la dernière paire.

2. Dans le cas où deux régions *exact* différentes d'un même tableau ont la même borne inférieure qu'une région *approx* (et des bornes supérieures différentes mais toutes deux inférieures ou égales à celle de la région *approx*), cette heuristique devient inapplicable car elle implique deux "vraies" bornes supérieures différentes pour la même région *approx*.

   `BESTI1` a deux régions de paramètre effectif différentes du tableau `NA1` qui sont *exact* et ont comme borne inférieure `NRGR`, voir Chapitre 5 (Section 5.2.2).

3. Une fois la paire de régions choisie, il faut remplacer la région *approx* par la région *exact* ayant la même borne inférieure, propager la nouvelle borne supérieure exacte jusqu'au début de la liste d'alias et puis recalculer les régions IN-*approx* et OUT-*approx* avant de ré-appliquer les règles d'inférence ci-dessus.

   Dans notre exemple, cette heuristique s'applique aussi aux régions de `CAMAT:NA1` des listes d'alias commençant par

   - la région IN-*approx* de `CAJAC:IPCO` ou
   - la région IN-*exact* de `CASDEG:IPCO`.

   Or la région *exact* de cette paire est de longueur `CAMAT:NSO` mais `CAMAT:NSO` ne peut pas être traduit dans l'espace de nommage de `CAJAC`, alors il faut donner à la région de `CAJAC:IPCO` une borne supérieure exprimée dans l'espace de nommage de `CAMAT`.

Notons que l'heuristique a pour effet d'éliminer des duplications de régions de paramètres formels *approx* (voir Section 7.3).

Dans notre exemple, cette heuristique a pour effet de

- faire disparaître les régions IN-*approx* de `BESTI1`, `PREFIX`, `PRXPL0` et `PRXPL1`,
- remplacer toutes les autres régions *approx* par des régions *exact* et
- regrouper dans une seule sorte toutes les régions présentées dans la Figure 7.5.

Il appartient alors au rétro-ingénieur de vérifier à la main que `PRXPL9` (resp. `PRXPL0`) traverse les `NS` premiers éléments d'`IPCO` (resp. `IPCE`) et que `CAJAC` traverse les `CAMAT:NSO` premiers éléments d'`IPCO` et d'`IPCE`. Il s'agit de trouver la précondition sur les valeurs de `NRLO` et `NRLE` décrite à la fin de la Section 5.2.2 du Chapitre 5.

7.5 Profils des sous-programmes de la spécification concrète

Les profils inférés pour les sous-programmes intervenant dans la spécification concrète du Chapitre 9 par la méthode que nous venons de décrire (en employant l'heuristique de la Section 7.4.4 et en supposant pour l'instant que toutes les opérations sont totales) sont les suivants :

$$\begin{aligned}
&PRXPL1 : S_NL, S_NS, S_NDOB, S_NRLx, S_NRLx \\
&\qquad\qquad \to S_IPCx, S_IPCx, S_ICCx, S_ICCx, S_NRLx, S_NRLx \\
&PRXPL7 : S_NS \to S_IPCx \\
&PRXPL8 : S_NS \to S_IPCx \\
&PRXPL9 : S_NL, S_IPCx, S_ICCx, S_NRLx, S_NRLx \to S_IPCx, S_ICCx \\
&PRXPL0 : S_NL, S_IPCx, S_ICCx, S_NRLx, S_NRLx \to S_IPCx, S_ICCx \\
&PRCHEN : S_IL, S_N1, S_N2, S_IPCx, S_ICCx, S_NRLx \to S_IPCx, S_ICCx \\
&PRCHE1 : S_IL, S_KP, S_ICCx, S_N2, S_NRLx \to S_KP
\end{aligned}$$

où

- S_NL est la sorte de `NL` ;
- S_NS est la sorte de `NS` ;
- S_NDOB est la sorte des plages de `PRXPL1:NDOB`, ... composés des `NS`[4] premiers éléments ;
- S_NRLx est la sorte des plages de `PRXPL1:NRLO`, `PRXPL1:NRLE`, `PRXPL9:NRLO`, `PRXPL9:NRLE`, `PRXPL0:NRLO`, `PRXPL0:NRLE`, `PRCHEN:NRLE`, `PRCHE1:NRLE`, ... composés des `NL` premiers éléments ;
- S_IPCx est la sorte des plages de `PRXPL1:IPCO`, `PRXPL1:IPCE`, `PRXPL7:IPCO`, `PRXPL8:IPCE`, `PRXPL9:IPCO`, `PRXPL0:IPCE`, `PRCHEN:IPCO`, ... composés des `NS` premiers éléments (c'est la sorte trouvée dans l'exemple des Sections 7.3 et 7.4) ;
- S_ICCx est la sorte des plages de `PRXPL1:ICCO`, `PRXPL1:ICCE`, `PRXPL9:ICCO`, `PRXPL0:ICCE`, `PRCHEN:ICCO`, `PRCHE1:ICCO`, ... composés des `NL` premiers éléments ;
- S_IL est la sorte de `PRCHEN:IL` et `PRCHE1:IL` ;
- S_N1 est la sorte de `PRCHEN:N1` ;
- S_N2 est la sorte de `PRCHEN:N2` et `PRCHE1:N2` et
- S_KP est la sorte de `PRCHE1:KP`.

4. Nous mettons dorénavant le nom des variables globales `NS` et `NL` sans le nom de sous-programme en préfixe car, contrairement à d'autres variables de bloc `COMMON`, celles-ci ne changent pas de nom de sous-programme en sous-programme, ni de valeur.

et où les applications de l'heuristique sont justifiées par

1. poser comme précondition qu'à l'entrée de `PRXPL9` et `PRXPL0` les `NL` premiers éléments de `PRXPL9:NRLO` et de `PRXPL0:NRLE`, sont inférieurs ou égaux à `NS` et

2. prouver à la main (par induction) que tous les éléments d'`ICCO` et `NRLE` que `PRCHEN` peut lire sont parmi les `NL` premiers.

Notons que la plage d'`ICCO` (resp. `ICCE`) qui est paramètre d'entrée de `PRXPL9` (resp. `PRXPL0`) est vide en réalité : aucun élément d'`PRXPL9:ICCO` (resp. `PRXPL0:ICCE`) n'est utilisé (à travers un appel à `PRCHEN`) avant d'être modifié. Mais la preuve de ceci demande un raisonnement par induction et donc l'intervention du rétro-ingénieur.

7.6 Discussion

Nous expliquons dans cette section quelques choix que nous avons fait dans l'élaboration de notre inférence des profils.

7.6.1 Pourquoi ne pas recourir à l'inférence de type classique?

Le profil d'un prédicat ou d'une opération d'une spécification algébrique est la spécification de la *signature* ("type signature") du sous-programme spécifié. Le résultat de notre analyse est donc équivalent à un ensemble cohérent de signatures pour les sous-programmes, mais comme nous ne prenons pas en compte les déclarations de type Fortran et n'avons pas de *types de base*, ces signatures sont construites à partir de **variables de type**. À chaque variable de type correspond une *sorte* de la spécification concrète. Comparons notre analyse à l'inférence de type classique.

L'inférence de type polymorphe classique ([Mil78]) est appliquée surtout aux programmes écrits dans des langages fonctionnels et déjà partiellement typés par des déclarations de type. Cette analyse repose sur un jeu de types de base et de constructeurs de type prédéfinis. On affecte une variable de type différente à chaque terme sans type déclaré et les règles d'inférence de type sont appliquées afin d'unifier des termes de type contenant des variables et trouver des types cohérents pour toutes les déclarations et applications de fonctions du programme, y compris celles concernant les opérations primitives telles que l'affectation, les tests d'égalités ou inégalités et les opérations arithmétiques. Certaines fonctions, telles que les opérations arithmétiques, sont souvent équipées d'un type prédéfini non polymorphe, mais toute fonction dont le type est inconnu est supposée polymorphe par défaut. Les types sont ceux de la logique d'ordre supérieure. Quant aux tableaux, aucun traitement spécifique n'est proposé et surtout pas la possibilité de donner des types différents à des plages différentes d'un même tableau.

À cause de l'utilisation intensive des opérations arithmétiques dans les codes que nous avons analysés, l'inférence de type classique serait incapable de distinguer les différents types d'entiers ou de tableaux d'entiers décrits ci-dessus. En effet, pour éviter un résultat très près des types Fortran déclarés, il faudrait considérer comme polymorphes les opérations arithmétiques, ainsi que les autres utilitaires employés sur des entiers et tableaux d'entiers.

Sinon, nous pourrions imaginer de distinguer positifs, naturels et entiers, mais ceci ne permettrait pas de faire les distinctions que nous désirons. De la même façon, nous pourrions distinguer différents types de tableaux en intégrant la **dimension** du tableau dans son type. Toutefois, la dimension effective de tous les tableaux résultant de la simulation de l'allocation dynamique, n'est jamais déclarée explicitement et les deux tableaux `ICCO` et `NRLO` ont la même dimension **effective** mais nous ne voulons pas leur donner la même sorte.

Par ailleurs, le "subtype polymorphism" semble suffisant pour décrire les codes Fortran analysés. Nous n'avons trouvé aucun exemple de "parametric polymorphism" (tel que le type d'un sous-programme traitant n'importe quelle liste, indépendamment du type de ses éléments), ni de sous-programmes *de deuxième ordre* prenant un autre sous-programme comme paramètre. En tout cas, nous sommes incapables de spécifier des sous-programmes de deuxième ordre dans le langage *CASL* de spécification algébrique [5].

5. Un code comportant un sous-programme de deuxième ordre ne pourrait être spécifié qu'après sa transformation afin

Contrairement à l'inférence classique de type, nous n'avons pas de types de base et ne prenons pas en compte les déclarations de type Fortran. Nous nous restreignons aux types des termes de la logique du premier ordre et n'affectons des types (plus exactement des sortes, car ils n'ont pas de "sens" pour l'instant) qu'aux paramètres formels de sous-programmes (tels que définis ci-dessus). Au lieu de présenter notre analyse comme l'unification de variables de type, nous avons choisi de la présenter comme l'inférence d'une relation entre paramètres formels du même type (et donc sorte). Ceci afin d'expliquer comment différentes occurrences de ce qui semble être le même paramètre n'ont pas toujours le même type.

7.6.2 Pourquoi ne pas avoir pris les sous-sortes en compte ?

Nous avons supposé qu'il n'y a pas de coercition implicite de sortes et nous avons ignoré d'éventuelles relations de sous-sorte entre les sortes. Expliquons pourquoi.

1. La première raison est le coût de la prise en compte des sous-sortes : les résultats de l'analyse sont plus difficiles à exploiter.

   En effet, nous aurions pu prendre en compte dans notre analyse les possibilités que *CASL* permet de spécifier, à savoir :

   - une relation d'ordre sur les sortes et
   - la coercition implicite dans le cas d'un paramètre effectif d'entrée appartenant à une sous-sorte de la sorte du paramètre formel.

   Nous aurions défini alors une relation binaire *sub_or_same_sort* sur les régions et remplacé les règles 7.4, 7.5 et 7.6 par :

   $$dynamic\_alias(f, e) \Rightarrow sub\_or\_same\_sort(e, f) \qquad (7.8)$$

   L'inconvénient est que le système d'inférence résultant n'aurait pas permis d'inférer l'égalité des sortes de deux paramètres (sauf si ils sont caractérisés par deux régions *exact* aux bornes identiques). Le résultat de la prise en compte des sous-sortes aurait été donc

   - un plus grand nombre de sortes
   - et une relation d'ordre partielle difficile à exploiter.

2. La deuxième raison est que l'intérêt théorique de la prise en compte des sous-sortes s'est révélé illusoire pour les exemples que nous avons trouvés dans les codes analysés.

   En effet, les sous-sortes permettent en théorie

   - de spécifier des sous-programmes polymorphes en terme du *"subsort polymorphism"* et
   - de conclure à une relation de sous-sorte dans certains cas d'inclusion de régions.

   Nous expliquons dans la suite de cette section que dans les exemples que nous avons trouvé, la spécification en terme de sous-sortes n'était pas adaptée.

Nous ne prenons donc pas en compte les sous-sortes dans l'élaboration des profils des opérations, mais nous allons les prendre en compte dans l'analyse des accès aux tableaux au sein des sous-programmes, comme expliqué dans le chapitre suivant.

---

d'éliminer le sous-programme par l'*expansion en-ligne*. Il s'agit de remplacer chaque appel au sous-programme par le corps du sous-programme après substitution des paramètres formels par les paramètres effectifs.

7.6.3 Le traitement du chevauchement des régions

Deux régions d'un même tableau peuvent être ni identiques, ni complètement disjointes. Il peut s'agir d'inclusion ou de chevauchement partiel.

Un cas spécial est celui des régions engendrées par deux boucles imbriquées. Dans ce cas, nous savons que la région de la boucle externe inclut celle de la boucle interne et que cette inclusion ne signifie pas l'inclusion des sortes des deux régions.

Que pouvons nous inférer alors dans les cas de régions engendrées par des boucles non imbriquées?

Dans le cas d'inclusion d'une région quelconque dans une autre région *approx*, nous ne pouvons rien inférer car rien ne prouve l'inclusion des paramètres caractérisés par ces deux régions. De la même façon, le chevauchement partiel d'une région *approx* et d'une autre région quelconque ne signifie rien.

Par contre, dans le cas d'inclusion d'une région quelconque dans une autre région *exact*, nous savons qu'il y a réellement inclusion des deux paramètres caractérisés. Dans ce cas, l'explication la plus évidente pour l'inclusion stricte est que la région la plus petite appartient à une sous-sorte de la sorte de la région la plus grande. Nous aurions pu modifier notre analyse afin de prendre en compte les sous-sortes et faire cette inférence automatique. L'exemple suivant démontre pourquoi nous préférons simplement signaler l'inclusion au rétro-ingénieur.

> Deux vrais paramètres formels de `PRXPLO` sont caractérisés par deux régions différentes, l'un du tableau `NRGR` et l'autre du tableau `NRCO`. Ces régions ont comme alias dynamiques au niveau de `BESTI1` deux régions *exact* différentes de `NA1`, toutes deux incluses dans une autre région *exact* de `NA1`. Cette dernière est alias dynamique
>
> – de la région de `NRGC` caractérisant le paramètre formel de `CACONT` et
> – de la région de `NRGC` caractérisant le paramètre formel de `CADSTM`.
>
> En l'occurrence, le tableau `NRGR` représente un attribut des *groupes*, et le tableau `NRCO` le même attribut des *consommations*, mais dans `CACONT` et `CADSTM` les deux peuvent être traités de la même façon, ce qui permet d'économiser quelques instructions. Or nous n'avons pas envie de spécifier cette économie dans la spécification concrète, où les *groupes* et les *consommations* sont spécifiés par deux types abstraits de données différents. C'est pourquoi la meilleure solution dans cet exemple est simplement de signaler la situation au rétro-ingénieur.

Dans le cas d'une région *approx* ayant exactement les mêmes bornes que celles d'une région *exact*, nous pouvons

– soit inférer que la sorte de la région *approx* est plus petite ou égale à celle de la région *exact* et procéder comme dans le cas de l'inclusion stricte

– soit employer l'heuristique de la Section 7.4.4, en d'autres termes, supposer que les bornes de la région *approx* sont en fait exactes, mais l'outil d'analyse statique n'a pas pu le démontrer.

Nous ne pouvons pas expliquer le chevauchement partiel de deux régions *exact* et nous signalons alors une anomalie.

Ces inférences sont résumées dans la table de la Figure 7.6.

type d'intersection *	*exact* * *exact*	*approx* * *exact*	*exact* * *approx*	*approx* * *approx*
=	=	$\leq$ possible	$\leq$ possible	-
$\subset$ boucles imbriquées	-	-	-	-
$\subset$ autre	$<$ possible	$<$ possible	-	-
autre	anomalie	-	-	-

FIG. 7.6: Relation inférée entre les sortes en cas d'intersection de deux régions

7.6.4 Les hypothèses implicites

La règle centrale de notre analyse introduite dans la Section 7.1 repose sur deux hypothèses implicites :

1. que nous pouvons affecter une seule sorte à chaque vrai paramètre formel, c'est-à-dire qu'il n'y a pas de sous-programmes polymorphes et

2. que les valeurs successives d'une même variable Fortran appartiennent à la même sorte.

Traitement du polymorphisme

L'analyse que nous proposons des sous-programmes dédiés au suivi des réseaux électriques devraient permettre de distinguer les sortes des différents entiers ou tableaux d'entiers cités en exemple dans la Section 7.1. Cependant des sous-programmes "utilitaires" peuvent être employés pour effectuer des opérations de base sur des entiers ou des tableaux de toutes les sortes.

Or nous avons déjà expliqué comment prendre en compte les sous-sortes. L'analyse ainsi modifiée des sous-programmes "utilitaires" aurait pour effet de créer une sur-sorte (la sorte de tous les entiers) des sortes des numéros de sommet et des numéros de ligne.

Mais cela compliquerait inutilement l'exploitation des résultats dans le but de trouver des profils pour les sous-programmes "utilitaires" qui ne sont pas ceux qui vont nous aider le plus à comprendre le code. Par ailleurs, c'est surtout ces sous-programmes qui ne respectent pas les types Fortran déclarés car parfois ils ne font pas réellement état d'une sur-sorte de leurs arguments, mais juste d'une "astuce" de programmation que nous voulons exclure de la spécification concrète.

> Par exemple, les deux premiers paramètres déclarés en Fortran du sous-programme `MOVEK` ont comme type déclaré un tableau de caractères. En réalité, ce sous-programme est employé surtout pour copier un tableau de n'importe quel type dans un autre. Mais il est parfois appelé avec un scalaire comme deuxième paramètre effectif !

Cependant, sans modification des règles d'inférence pour prendre en compte le "subsort polymorphism", l'analyse d'un sous-programme "utilitaire" risque de mettre dans la même sorte des régions caractérisant des paramètres de significations complètement différentes. Une autre approche au problème est donc d'essayer de détecter dans le code les sous-programmes qui sont "utilitaires" et d'en éliminer les appels avant d'effectuer cette analyse. Le choix des sous-programmes à éliminer est évidemment subjectif et détermine la granularité des sortes inférées. Toutefois, nous avons déjà proposé (dans la Section 3.9 du Chapitre 3 et la Section 6.1.3 du Chapitre 6) de substituer par un code "équivalent" des appels de sous-programmes gérant les entrées et sorties fichier, ainsi que ceux (comme `MOVEK`) qui ne respectent pas des règles de bonne programmation telles que celles proposées par des outils d'analyse statique. Notons par ailleurs que l'analyse décrite ici se limite aux sous-programmes codés en Fortran et ne prend donc pas en compte les fonctions utilitaires pré-codées qui font partie du langage. Les autres sous-programmes utilitaires peuvent souvent être identifiés par leurs sites d'appel nettement plus nombreux et moins localisés que ceux des sous-programmes dédiés au suivi des réseaux électriques.

Valeurs successives de sortes différentes pour une même variable

Nous avons expliqué dans le Chapitre 6 (Section 6.4) que les variables Fortran sont parfois réutilisées pour stocker des valeurs intermédiaires dans le calcul d'une valeur définitive, voire des valeurs d'une signification complètement différente. Une approche à ce problème est la transformation du code dans l'esprit du "SSA form" (voir Chapitre 6, Section 6.4) avant d'effectuer l'analyse décrite ici. Il devrait être possible de trouver une solution beaucoup moins coûteuse spécialisée à cette analyse. Il s'agirait de modifier les règles d'inférence afin de ne plus considérer comme identiques deux occurrences d'une même région si une modification de la valeur de la région a pu s'effectuer entre les deux. Nous n'avons pas eu le temps d'étudier cette possibilité.

7.7 Implémentation dans *PIPS*

Nous avons réalisé une première implémentation de l'analyse décrite ici comme extension de l'outil *PIPS*. Le but de cette réalisation était une étude rapide de faisabilité. C'est pourquoi elle est basée sur les "summary regions" de *PIPS* sans les extensions énumérées dans la Section 7.2. Un test approximatif des régions *exact* parmi les régions MAY est effectué et seule l'analyse *prudente* décrite dans la Section 7.4.2 est réalisée. C'est donc la réalisation d'une analyse nettement plus approximative que celle décrite ici.

Cette réalisation a été testé sur *SYSDIC COURCIRC SILODI* (voir le Chapitre 2). Le résultat est un nombre trop élevé de sortes : le nombre de sortes détectées dans les trois cas est environ la moitié du nombre de régions manipulées. Il est vrai que nous pouvons nous attendre à un nombre élevé de sortes pour ces codes à cause de leurs graphes des appels peu profonds et avec peu de branchements. Toutefois, cette réalisation de l'analyse éclate inutilement de nombreuses sortes à cause des nombreuses approximations. Les résultats fournis par cette réalisation ont aidé à affiner l'analyse décrite ici mais pour réaliser celle-ci il faudrait étendre le calcul des régions de *PIPS*, ce qui demande beaucoup plus de travail et sort du cadre de cette thèse.

7.8 Comparaison à d'autres travaux

D'autres travaux rapportés dans la littérature ressemblent à notre approche, notamment :

- d'autres travaux sur l'analyse des alias : ceux-ci s'adressent surtout aux problèmes posés par les codes programmés en *C* dans lesquels il y a peu de références aux tableaux mais il s'agit d'analyser les pointeurs. Cependant les travaux de [ZRL96] et [Ste96] sont comparables à notre approche, ainsi que
- l'inférence de type dans *Lackwit* [OJ97].

L'objectif des travaux décrits dans [ZRL96] et [Ste96] est l'analyse *"flow-insensitive"* (i.e. sans tenir compte du flot de contrôle) de l'emploi des pointeurs dans les codes *C*. [ZRL96] prend en compte non seulement (comme nous) des noms différents dans des sous-programmes différents pour la même zone en mémoire ou pointeur, mais aussi toutes les références différentes formées à partir des pointeurs, du "de-referencing" et de l'accès aux champs des structures. Il définit une relation entre deux "noms" qui peuvent, à cause d'une affectation d'une valeur à un pointeur ou d'un appel d'un sous-programme avec passage d'un pointeur, référencer la même zone en mémoire à un moment de l'exécution. De la fermeture transitive, réflexive et symétrique de cette relation, il engendre une relation d'équivalence sur les "noms" qui permet de partitionner le code analysé afin de décider pour chaque fragment résultant s'il faut appliquer une analyse "flow-sensitive" des alias de pointeurs.

[Ste96] emploie l'inférence de type dans un système non-standard de types comme moyen efficace d'effectuer l'analyse "flow-insensitive" des pointeurs dans les codes programmés en *C*. Dans son système de types, un "type" représente un ensemble d'adresses de variables, à savoir : toutes les adresses qui peuvent être indiquées par le même pointeur. Chaque "type" a un "composant" représentant les adresses pouvant être indiquées par les variables de ce "type". Tous les pointeurs à une même adresse n'ont pas toujours le même "type" mais ont toujours le même "composant". Le système de type représente alors un graphe : le "storage shape graph", et l'égalité entre "types" n'est pas l'égalité classique structurelle mais celle qu'on obtiendrait par le "tagging" des "types".

[OJ97] décrit l'application aux codes programmés en *C* de l'inférence de type classique ([Mil78]) mais avec l'ajout du "tagging". Le but étant de détecter des variables ayant la même "représentation" et donc le même "type". Les règles d'inférence donnent à deux variables le même "type" si la valeur de l'une est affectée à l'autre ou si l'une instancie l'autre dans un appel de sous-programme. Un paramètre formel peut être polymorphe, c'est-à-dire avoir plusieurs "types".

Nous exploitons la similarité entre analyse des alias et inférence de types, non pas pour employer l'inférence de types non standard pour l'analyse des alias, mais afin d'employer l'analyse des alias pour trouver des "types" non polymorphes pour les paramètres des sous-programmes et construire une spécification algébrique. Notre analyse est spécifique au langage Fortran et basée sur les *régions* de tableaux.

Chapitre 8

D'autres analyses

Résumé

Dans ce chapitre, nous décrivons et discutons les autres analyses que nous avons réalisées à la main avant de procéder à l'élaboration de la spécification concrète décrite dans le chapitre suivant.

Dans la Section 8.1, il s'agit d'affiner les profils présentés dans le chapitre précédent et d'amorcer la définition des sortes des plages de tableau par une analyse intra- et inter-procédurale des références aux tableaux, en prenant en compte les boucles `DO` et les affectations.

Dans la Section 8.2, nous proposons une analyse de la terminaison des boucles et de l'emploi d'opérations primitives partielles de Fortran, afin de détecter des contraintes sur les valeurs des entrées des sous-programmes qui ne sont explicitées ni par les déclarations de type Fortran, ni par les analyses précédentes.

La Section 8.3 propose une heuristique de détection de réalisations de types abstraits de données dans des codes Fortran, basée sur l'analyse des points d'utilisation des tableaux. Il nous semble que la meilleure façon d'aborder la compréhension d'un code Fortran entier est d'isoler et d'étudier un par un des réalisations de types abstraits de données.

8.1 Inférence intra- et inter-procédurale de sortes

Nous complétons les résultats de l'inférence des profils du Chapitre 7 par une analyse intra- et inter-procédurale des références aux tableaux, en prenant en compte les boucles `DO` et les affectations au sein des sous-programmes. Nous avons effectué cette analyse à la main pour le fragment de code dont la spécification fournit l'exemple principal de cette thèse, mais il s'agit d'une inférence de type qui pourrait sans doute être automatisée. Nous n'avons pas eu le temps d'entreprendre son automatisation, ni même sa formalisation complète : nous nous contentons ici de donner quelques principes illustrés par des exemples.

En effet cette analyse est assez délicate puisque, contrairement au cas de L'inférence des profils, où nous obtenons de bons résultats sans prendre en compte les coercitions implicites dues à un ordre sur les sortes, ici nous sommes obligés de considérer que chaque branchement conditionnel dans le code peut impliquer une coercition de sorte. Car un branchement impose une contrainte (le *prédicat de branche*, voir Section 4.3.2 du Chapitre 4) sur la valeur à partir de ce point de la variable testée et la restreint ainsi à une sous-sorte de sa sorte initiale.

Les résultats de notre analyse sont

1. des informations sur les **valeurs** appartenant à certaines sortes et
2. des **relations** entre les sortes :
   - une sorte est la sous-sorte d'une autre dont les éléments respectent une certaine contrainte ou
   - une sorte est l'union d'autres sortes ou de sortes et de constantes.

Le point de départ de notre analyse est l'ensemble de profils inféré par la méthode du Chapitre 7 et présenté dans la Section 7.5 du même chapitre.

Analyse de boucles DO

Une boucle `DO` n'est exécutée que si sa limite supérieure est supérieure ou égale à sa limite inférieure. Sinon, elle est simplement sautée. Une boucle `DO` contient donc un branchement caché. Pour une boucle `DO` de la forme de celles trouvées dans les sous-programmes de notre exemple, dans laquelle la limite inférieure est 1 et la limite supérieure variable :

```
DO I = 1, L
```

nous affectons donc une sorte, disons S_L, à la limite supérieure de la boucle qui inclut 1 et peut inclure des valeurs inférieures à 1. Ensuite nous affectons la sous-sorte de S_L restreinte aux valeurs positives à la variable `I`.

> Ainsi, des boucles `DO` de `PRXPL9` et de `PRXPL0` dont les bornes supérieures sont égales à `NL` et les indices sont alias dynamiques de `PRCHEN:IL`, nous inférons que
>
> – S_IL (la sorte de `IL`) est la sous-sorte de S_NL (la sorte de `NL`) qui regroupe les valeurs positives.

Analyse des affectations

Étant donnée une instruction dans le code affectant une valeur constante c à une variable v de sorte S_v, nous pouvons inférer que S_v inclut c. En cas de l'affectation de la valeur d'une autre variable a de sorte S_a, nous inférons que S_v inclut S_a ou une sous-sorte de celle-ci en cas d'affectation dans une branche conditionnelle. En analysant toutes les affectations possibles de toutes les variables appartenant à S_v, nous pouvons construire la définition complète de S_v comme l'union de (sous-)sortes et/ou de valeurs constantes.

Nous analysons donc les affectations afin de faire des inférences sur les sortes des indices ou des valeurs des éléments de tableaux.

Analyse des tests

Nous pouvons aussi faire quelques inférences à partir des tests, mais elles sont relativement faibles. En cas d'un test d'égalité d'une variable v à une constante c, nous pouvons conclure que S_v inclut c. S'il s'agit d'une inégalité, telle que tester si la valeur de v est inférieure ou égale à une constante c, alors nous pouvons inférer seulement que S_v inclut au moins une valeur inférieure ou égale à c. De la même façon, un test de l'égalité de deux variables implique seulement que leurs sortes ont une intersection non vide et les tests d'inégalité des valeurs de deux variables implique qu'il existe au moins une paire de valeurs appartenant aux sortes de ces variables qui satisfont cette inégalité.

Inférences de sortes des indices et de valeurs d'éléments de tableaux

Notre objectif est d'inférer un ensemble cohérent de sortes pour les indices et pour les éléments de toutes les sortes de plages de tableaux trouvées par l'inférence des profils. Or dans l'analyse du Chapitre 7, nous avons cherché des sortes pour les plages et non pas pour leurs indices ou leurs éléments, car

– nous ne voulions pas donner la même sorte à deux plages de tableaux ayant les indices et les éléments des même sortes, mais servant à réaliser des structures de données différentes, et

– différents éléments d'une même plage de tableau peuvent avoir des significations différentes.

Cependant, après avoir différencié les plages de tableaux à l'aide de l'inférence des profils, nous pouvons chercher à caractériser par une seule sorte tous les éléments d'une plage, même s'il s'agit d'une sorte qui est l'union des sortes des différents éléments. Dans ce dernier cas, nous simulons l'union de sortes par la relation de sur-sorte.

Soit $Array[Index, Value]$ la sorte de tout tableau dont les indices sont de sorte $Index$ et les valeurs des éléments de sorte $Value$. L'objectif de cette analyse est de pouvoir définir chaque sorte f d'une plage de tableau comme une sous-sorte de la sorte $Array[Index\_f, Value\_f]$ de tous les tableaux des indices de la sorte de ceux des tableaux de f et des éléments de la sorte de ceux des tableaux de f. Nous allons dire que f est plus petite ou égale[1] à $Array[Index\_f, Value\_f]$ puisque nous ne savons pas si chaque élément d'une plage de tableau de sorte f peut prendre toutes les valeurs de la sorte $Value\_f$. En effet, au cas où les plages de tableaux de sorte f servent à réaliser d'autres structures de données, il peut y avoir des contraintes supplémentaires sur les valeurs de certains éléments de f.

Par exemple, de la boucle `DO` de `PRXPL9` dans laquelle les valeurs des `NL` premiers éléments de `NRLO` et de `NRLE` sont utilisés, avec la variable de boucle `IL` comme indice, nous concluons que la sorte des **indices** des plages de tableau de sorte S_NRLx est la sorte (S_IL) d'`IL`.

Trouvons la sorte des **éléments** des plages de tableau de sorte S_NRLx :
De l'affectation dans `PRXPL9` et `PRXPLO` de la valeur de `NRLO(IL)` (resp. `NRLE(IL)`) à `PRXPL9:N1` (resp. `PRXPLO:N1`), nous concluons que les valeurs des éléments des plages de tableau de sorte S_NRLx ont la sorte (S_N1) de `N1` et donc

– $S\_NRLx < Array[S\_IL, S\_N1]$

Ensuite, de l'utilisation, dans une branche conditionnelle de `PRCHEN`, d'`IPCO(N1)`, nous concluons que la sorte des **indices** des plages de tableau de sorte S_IPCx inclut S_N1 restreinte aux valeurs positives.

Trouvons la sorte des **éléments** des plages de tableau de sorte S_IPCx :
De l'affectation de 0 aux `NS` premiers éléments d'`IPCO` (resp. `IPCE`) effectuée par `PRXPL7` (resp. `PRXPL8`), suivie par l'affectation éventuelle (dans l'appel de `PRCHEN` depuis la boucle de `PRXPL9` (resp. `PRXPLO`)), de `PRCHEN:IL` à chacun des `NS` premiers éléments d'`IPCO` (resp. `IPCE`), nous concluons que la sorte des valeurs des éléments des plages de tableau de sorte S_IPCx est la sorte (S_IL) d'`IL` augmentée par la valeur 0 et donc

– soit S_N1Pos la sous-sorte de S_N1 qui regroupe les valeurs positives,
– soit S_IL0 la sorte S_IL augmentée de la valeur 0,
– $S\_IPCx < Array[S\_N1Pos, S\_IL0]$

L'analyse du flot de données dans `PRCHEN` révèle que l'affectation de la valeur d'`IL` à `ICCO(KP)` se situe dans une branche conditionnée par la supériorité de la valeur de `KP` à zéro et suit l'affectation à `KP` d'une valeur de soit `IPCO`, soit `ICCO`. En fait dans toutes les références à une plage de sorte S_ICCx, la valeur de l'indice est copiée d'un élément d'une plage de sorte S_IPCx ou S_ICCx et est supérieure à zéro. Nous concluons que la sorte des **indices** des plages de sorte S_ICCx est l'union des sortes des valeurs des éléments des plages de sorte S_IPCx ou S_ICCx, restreinte aux positifs. Mais quelle est la sorte des **éléments** des plages de tableau de sorte S_ICCx ?

La seule affectation d'une valeur à un élément de plage de sorte S_ICCx a lieu dans `PRCHEN`. L'analyse du flot de données révèle que cette affectation de la valeur de `KCH` à `ICCO(IL)` suit l'affectation à `KCH` d'une valeur soit d'`IPCO`, soit d'`ICCO`. Nous concluons que la sorte des valeurs des éléments de plages de sorte S_ICCx est celle des valeurs des éléments des plages de sorte S_IPCx et donc :

– $S\_ICCx < Array[S\_IL, S\_IL0]$.

De l'affectation dans `PRXPL9` et `PRXPLO` de la valeur de `NRLE(IL)` (resp. `NRLO(IL)`) à `PRXPL9:N2` (resp. `PRXPLO:N2`), nous concluons que

– $S\_N2 = S\_N1$

De l'affectation dans `PRCHEN` d'une valeur à `KP`, nous concluons :

– $S\_KP = S\_IL$.

Ces inférences nous permettent d'affiner les profils trouvés par la méthode du Chapitre 7 (en substituant S_IL pour S_KP et S_N1 pour S_N2) et d'amorcer la définition des sortes des plages de tableau.

1. Comme dans *CASL*, nous utilisons le terme sous-sorte pour les sortes qui sont plus petites ou égales à la sur-sorte.

8.2 Détection d'autres contraintes sur les paramètres

Notre motivation pour l'inférence des profils décrite dans le Chapitre 7 était le constat que le programmeur peut prendre en compte des contraintes sur les valeurs des paramètres d'entrée d'un sous-programme qui ne sont pas explicitées par leur type Fortran déclaré. Toutefois, le but de l'inférence des profils n'était pas d'**expliciter** ces contraintes. Nous devons détecter les contraintes et les intégrer dans la spécification concrète. Notons qu'il peut s'agir non seulement de contraintes sur la valeur d'un paramètre d'entrée en isolation, mais aussi de contraintes sur des **combinaisons** des valeurs de plusieurs paramètres d'entrée.

Comme nous étudions des sous-programmes écrits pour un contexte connu à l'avance (et non pas des routines de "bibliothèque"), une approche à la détection de telles contraintes est l'étude des paramètres effectifs. Ainsi,

- d'une part l'analyse du chapitre 7 nous a permis de prendre en compte que les entrées d'un sous-programme peuvent être les sorties d'un autre. Nous verrons dans le Chapitre 9 un exemple de comment nous pourrons étendre les résultats de cette analyse aux **"groupes"** de paramètres formels.
- D'autre part, l'analyse de la Section 8.1 nous a permis de prendre en compte les entrées qui sont des variables de boucles `DO` ou des valeurs lues dans d'autres tableaux.

Cependant, l'analyse des paramètres effectifs ne suffit pas pour connaître toutes les contraintes implicites sur les valeurs des paramètres d'entrée des sous-programmes, car nous ne connaissons pas les valeurs possibles des entrées depuis l'extérieur du code. Une approche complémentaire est donc l'étude de deux aspects de chaque sous-programme :

- la terminaison et
- l'emploi d'opérations primitives partielles de Fortran.

La terminaison

Nous supposons que les boucles des sous-programmes doivent terminer, ce qui peut induire des contraintes sur les valeurs des paramètres d'entrée.

> Par exemple, le sous-programme `PRCHE1` utilise l'élément du paramètre d'entrée `ICCO` indexé par l'autre paramètre d'entrée `KP`, et ensuite la suite d'éléments $\texttt{ICCO}^n\texttt{(KP)}$, jusqu'à la satisfaction de la condition de la boucle `while`. La boucle ne terminera pas si la valeur de l'un des éléments utilisés est égale à son indice. En fait, il faut qu'à chaque itération de la boucle un élément différent soit utilisé, avec une valeur différente de celles des autres éléments. Ceci jusqu'à la terminaison de la boucle soit par l'utilisation de la valeur zéro soit par la satisfaction de l'autre composant de la condition de la boucle `while`. La terminaison de cette boucle impose donc une contrainte sur combinaisons des valeurs en entrée des paramètres `KP` et `ICCO` de `PRCHE1`.

Les opérations partielles primitives de Fortran77

La valeur initiale, avant toute affectation, d'un élément de tableau Fortran77 dépend du compilateur employé, et parfois de l'état de la mémoire au moment de l'exécution du code. Par conséquent, le résultat de la lecture d'un élément de tableau auquel aucune valeur n'a été affectée n'est pas défini. En effet, l'opération de lecture d'un élément de tableau Fortran77 est *partielle* et il incombe au programmeur de s'assurer que les éléments de tableau ne sont jamais utilisés par son code avant d'être modifiés. Le fait que l'opération de lecture d'un élément de tableau soit partielle implique que de nombreux sous-programmes ne doivent accepter en entrée que des tableaux dont les "bons" éléments sont déjà initialisés. Or le type Fortran du tableau n'indique pas quels éléments ont une valeur définie.

La région IN du tableau calculée par *PIPS* caractérise les éléments dont la valeur à l'entrée du sous-programme sera utilisée et doit donc avoir déjà été modifiée. En cas de région EXACT c'est donc cette région qui nous donne la contrainte que nous cherchons ici. Contrairement à l'analyse pour l'inférence des profils, où nous avons choisi d'analyser les **bornes** de la plage de tableau traversée au lieu des éléments individuels utilisés, la convexité de la région est importante ici (voir l'exemple suivant).

Notons par ailleurs qu'il existe d'autres opérations primitives partielles de Fortran77, mais nous ne discutons ici que des exemples des implications de la partialité de la lecture des tableaux.

> Par exemple, tous les `NL` premiers éléments de `NRLO` sont utilisés par la boule `DO` de `PRXPL9`. `PRXPL9` ne doit donc prendre en entrée que des valeurs de `NRLO` et `NL` pour lesquelles les `NL` premiers éléments de `NRLO` ont été modifiés. Cette contrainte sur les valeurs de `NRLO` et `NL` en entrée de `PRXPL9` est exprimée de façon précise par la région IN EXACT calculée par *PIPS*.
>
> Mais *PIPS* est incapable de caractériser de façon précise les éléments d'`ICCO` utilisés par `PRCHEN`. Ces éléments sont indexés par `KCH` dont la valeur est calculée récursivement dans la boucle de `PRCHE1`. Nous avons décidé (dans la Section 7.5 du Chapitre 7) que la plage d'`ICCO` utilisée est **bornée** par les indices 1 et `NL` mais il n'est pas vrai que **tous** les `NL` premiers éléments d'`ICCO` doivent être modifiés avant l'appel de `PRCHEN`. Par ailleurs, `NL` n'est pas un paramètre de `PRCHEN`, alors l'utilisation d'`ICCO(KCH)` induit une contrainte sur les valeurs en entrée des paramètres de `PRCHEN`, qui doit être exprimée de façon récursive.
>
> L'utilisation par `PRCHEN` des éléments de `NRLE` indexés par `KCH` induit une contrainte similaire sur les valeurs de ses paramètres `IPCO`, `N1`, `ICCO` et `NRLE`.

La prise en compte de différentes façons de toutes les contraintes sur les valeurs des paramètres d'entrée des sous-programmes, sera illustrée sur l'exemple principal de cette thèse dans le Chapitre 9.

8.3 Détection de réalisations de types abstraits de données

Les profils inférés dans le Chapitre 7 et complétés par l'analyse de la Section 8.1 nous fournissent pour chaque sorte, les sous-programmes qui ont des paramètres d'entrée ou de sortie de cette sorte. C'est-à-dire qu'une même valeur de cette sorte affectée à une variable par l'un de ces sous-programmes est ensuite utilisée par d'autres. Face à des identificateurs de paramètres formels restreints à six lettres et des commentaires souvent cryptiques, l'analyse des opérations effectuées par ces différents sous-programmes sur les valeurs de la même sorte que le paramètre, peut nous aider à comprendre le "sens" du paramètre. Or un type (abstrait) de données [2] est composé d'un ensemble de valeurs et des fonctions et prédicats permettant de calculer, d'écrire, de lire, ... ces valeurs. Il nous semble que la meilleure façon d'aborder la compréhension d'un code Fortran entier est d'isoler et d'étudier un par un des réalisations de types abstraits de données. Par ailleurs, les spécifications algébriques sont en règle générale structurées en types abstraits de données.

Trois difficultés sont induites par cette approche :

1. Les structures de données autres que celles fournies par les types primitifs de Fortran sont réalisées en Fortran à l'aide de plusieurs variables scalaires ou fragments de tableau. Or chaque opération ou prédicat appartenant au type abstrait de données correspondant ne fait pas toujours référence à tous les composants de la structure de données. C'est-à-dire qu'il y aura un sous-programme Fortran avec comme paramètres certains des composants de la structure de données, un autre avec certains autres composants, ... Pour retrouver tous les sous-programmes Fortran participant à la réalisation d'un même type abstrait de données, il ne suffit donc pas de chercher tous les sous-programmes avec (au moins) un paramètre d'une certaine sorte : il faut trouver l'ensemble de sortes qui réalisent la structure de données et étudier tous les sous-programmes faisant référence à des variables d'au moins une sorte appartenant à cet ensemble.

2. Même après la décomposition proposée dans le Chapitre 6 (Section 6.3), la fonctionnalité des sous-programmes faisant référence aux structures de données ne correspond pas toujours aux simples opérations d'accès. Souvent, l'accès est mélangé avec le calcul pour lequel il est effectué.

3. Plus généralement, un type abstrait de données est supposé parfaitement encapsulé, d'où l'intérêt de les employer comme critère de décomposition du code : chaque type abstrait de données peut être analysé indépendamment des autres. Toutefois, les vieux codes Fortran n'étaient certainement pas conçus comme un ensemble de types abstraits de données parfaitement indépendants et la

2. Pour le distinguer des types du langage Fortran, nous allons appeler *type abstrait de données* un type de données composé non seulement d'un ensemble de valeurs de ce type mais aussi de fonctions et de prédicats.

frontière entre l'un et l'autre peut être difficilement détectable dans le code. Finalement, ce sera au rétro-ingénieur de décider de la structure de la spécification concrète. C'est une instance des limites de l'automatisation.

La première difficulté dans la détection de réalisations de type abstrait de données est donc de trouver des ensembles de sortes qui participent à la réalisation d'une même structure de données. Nous proposons des heuristiques que nous avons appliquées à la main aux codes analysés. Elles ont permis de détecter des instances des quatre modes typiques d'accès aux tableaux décrits dans le Chapitre 5 (Section 5.1.6). Elles sont basées sur l'analyse des indices dans les références de tableaux afin de détecter des références "en série" - une première référence à un élément de tableau fournit l'indice pour la référence suivante - ou "en parallèle" - un même indice sert dans les références de fragments de tableaux de plusieurs sortes. Nos heuristiques sont assez prudentes, conçues pour éviter de signaler des ensembles de sortes sans signification, elles n'indiquent souvent que des "noyaux" de réalisations de type abstrait de données. C'est au rétro-ingénieur d'enrichir pendant la spécification détaillée ces ensembles "noyaux" de sortes.

1. Nous proposons une analyse des sortes des plages de tableau car la plupart des structures de données nécessitent un tableau pour leur réalisation.

2. Pour une sorte $S1$ de plage de tableau donnée, et pour chaque point d'*utilisation* d'une plage de cette sorte, nous proposons d'étudier les *dépendances* de la valeur d'indice employée. Il s'agit de chercher d'autres plages de tableau pouvant être *utilisées* dans la *modification* de la valeur de l'indice. Si nous trouvons que dans **tous** les points d'utilisation de toutes les plages de tableau de sorte $S1$, la valeur de l'indice peut dépendre d'une plage de tableau de sorte $S2$, alors nous concluons que $S1$ et $S2$ interviennent dans la réalisation du même type abstrait de données.

   Nous proposons d'analyser de cette façon uniquement les points d'utilisation parce que nous avons trouvé de nombreux cas où l'initialisation d'une structure de données emploie un mode d'accès autre que celui employé dans son utilisation.

   Cette analyse permet, par exemple,

   - de regrouper les sortes S_IPCx et S_ICCx qui servent à réaliser une *table partitionneuse* (voir la Section 2.4.1 du Chapitre 2) de listes de numéros de ligne indexée par les numéros de sommet ;
   - de regrouper les sortes des plages de tableau appelées (dans la plupart des sous-programmes) `LHAC`, `NCHA` et `EHA` et qui servent à stocker une matrice creuse (voir Section 3.3.5 du Chapitre 3) et
   - de regrouper les sortes des plages de tableau appelées (dans la plupart des sous-programmes) `IDEB`, `ISUI`, `INDL` et `ELM`. Ces plages de tableau servent à réaliser une autre représentation de matrice creuse : `IDEB` stockant le numéro du premier élément non zéro de chaque ligne de la matrice, `ISUI` une liste chaînée des numéros des éléments non zéro suivants, `INDL` le numéro de colonne de chaque élément non zéro et `ELM` la valeur de chaque élément non zéro. Les sortes des tableaux `IDEB` et `ISUI` servent à réaliser une table similaire à celle réalisée par S_IPCx et S_ICCx mais indexée par une sorte différente et stockant des éléments d'une sorte différente.
     Toutefois, cette heuristique ne permet pas de détecter qu'il y a une façon alternative d'accéder à la matrice : au lieu d'un accès direct d'`IDEB` effectué à certains points d'utilisation, d'autres points d'utilisation emploient un tableau normalement appelé `LORD` pour lire dans un certain ordre des valeurs d'indice d'`IDEB`. Elle n'indique pas non plus qu'il y a un autre tableau normalement appelé `NONU` qui stocke le nombre d'éléments non zéro de chaque ligne. C'est une instance de la détection, à l'aide de l'heuristique, d'un "noyau" de sortes que le rétro-ingénieur doit enrichir à la main.

3. Une fois l'analyse précédente effectuée, nous pouvons analyser les sortes des plages de tableau qui ne sont pas systématiquement adressées à l'aide d'une autre plage de tableau. Ces sortes sont regroupées en ensembles ayant des indices de la même sorte. De tels ensembles de sortes de plages de tableaux peuvent être des réalisations des "classes d'objets" (voir Section 3.3.5 du Chapitre 3).

   Il faut appliquer cette heuristique après la précédente puisqu'elle aurait eu pour effet d'assimiler les plages de tableau de sorte S_ICCx à des attributs des lignes, si nous n'avions pas détecté que, bien qu'étant indexés par des numéros de ligne, ils sont toujours référencés à l'aide d'une plage de tableau de sorte S_IPCx.

Par ailleurs, il faut distinguer les valeurs successives des plages de tableau dans cette analyse, car la représentation d'une structure de données peut changer au cours du calcul des valeurs, comme illustré par les exemples du Chapitre 6 (Section 6.4).

Une fois le "noyau" de sortes trouvé, il faut lui ajouter tous les sous-programmes qui ont des paramètres d'une sorte appartenant à cet ensemble. Toutefois, il ne faut pas ajouter des sous-programmes qui n'effectuent que le simple passage des valeurs vers d'autres sous-programmes (en ce qui concerne ces sortes). Nous pouvons reconnaître ces sous-programmes grâce au fait que les régions de leurs paramètres formels (des sortes concernées) sont juste les régions au point d'entrée des paramètres effectifs des sous-programmes qu'ils appellent à leur tour (voir Section 7.3 du Chapitre 7). Finalement, nous y ajoutons les sortes d'éventuelles bornes variables des plages de tableau. Le résultat est la réalisation de type abstrait de données détectée par cette méthode.

Pour l'exemple de S_IPCx et S_ICCx, le résultat final est :

- les sortes S_IPCx, S_ICCx, S_NS et S_NL et
- les sous-programmes `PRXPL7`, `PRXPL8`, `PRXPL9`, `PRXPL0`, `PRCHEN`, `PRCHE1`, `CASDEG`, `CAPIV1` et `CAPIV2`.

Les opérations primitives sur la structure de données réalisées à l'intérieur de ces sous-programmes sont :

- l'initialisation de la table à une liste vide pour chaque numéro de sommet, réalisée par `PRXPL7` et `PRXPL8` ;
- l'ajout d'un élément à un certain point de la liste, réalisé par certaines instructions de `PRCHEN` ;
- la lecture de la liste de numéros de ligne pour un numéro de sommet donné réalisé par certaines instructions de `PRCHEN`, `CASDEG`, `CAPIV1` et `CAPIV2` ;
- tester si une liste de numéros de ligne est vide comme effectué par certaines instructions de `PRCHEN`, `PRCHE1`, `CASDEG`, `CAPIV1` et `CAPIV2` ;
- rendre la tête d'une liste non vide de numéros de ligne comme effectué par certaines instructions de `PRCHEN`, `PRCHE1`, `CASDEG`, `CAPIV1` et `CAPIV2` et
- rendre la queue d'une liste non vide de numéros de ligne comme effectué par certaines instructions de `PRCHEN`, `PRCHE1`, `CASDEG`, `CAPIV1` et `CAPIV2`.

Ceci est l'interprétation du type abstrait de données le plus élémentaire : une table partitionneuse de listes de numéros de ligne indexée par des numéros de sommet. Un autre point de vue consiste à prendre en compte tout le reste de chaque sous-programme ci-dessus (en plus des quelques instructions qui réalisent l'une des opérations primitives énumérées ci-dessus). Par exemple : prendre en compte l'ordre sur les numéros de ligne qui détermine le point d'insertion de chaque numéro de ligne par `PRCHEN`. Il faut alors augmenter le type abstrait de données de la sorte S_NRLx. Le type abstrait de données augmentée représente une table de co-cycles des sommets d'un réseau, dans laquelle chaque co-cycle est stocké comme une liste de numéros de ligne ordonnée d'une certaine façon (voir la Section 2.4 du Chapitre 2). Ce type abstrait de données fait partie lui-même d'une réalisation d'un modèle du réseau électrique.

De nombreux travaux sur la détection de réalisations de types abstraits de données ou de classes d'objets sont décrits dans la littérature [GKS97, SR97]. Ils proposent diverses analyses :

- soit des variables globales référencées dans chaque sous-programme (afin de détecter des classes d'objets)
- soit des types des paramètres de chaque sous-programme (afin de détecter des types abstraits de données).

L'objectif de ces analyses est de dégager des ensembles de sous-programmes dont chacun est associé à un ensemble de variables globales référencées ou types de paramètres. Dans le cas idéal, chaque ensemble de sous-programmes et de variables globales (resp. types de paramètres) serait disjoint des autres. L'expérience prouve que les codes réels ne correspondent pas à ce modèle. Diverses méthodes, dont l'analyse statistique ([CCM96]) et *l'analyse de concepts* ([LS97, SR97]), sont alors proposées pour distinguer parmi toutes les paires de sous-programme et variable globale référencée (resp. type de paramètre), celles qui sont significatives.

Notre approche est spécialisée aux programmes codés en Fortran. Ses points originaux sont les suivants :

- Au lieu des variables globales référencées ou des types Fortran des paramètres déclarés, elle est basée sur les sortes inférées (par la méthode du Chapitre 7) des plages de tableau effectivement référencées (vrais paramètres), ainsi que les sortes des indices inférées par la méthode de la Section 8.1 ci-dessus.
- Elle commence par une analyse de la façon d'accéder aux éléments des plages de chaque sorte, afin de regrouper les sortes de plages de tableau qui semblent intervenir dans la réalisation du même type abstrait de données. Ensuite, elle associe à chaque ensemble de sortes tous les sous-programmes ayant des paramètres **d'une ou de plusieurs** des sortes de l'ensemble.
- Le premier volet de l'analyse de l'accès aux tableaux est limité au points d'*utilisation*. Elle est basée sur les *dépendances* des valeurs d'indices : nous cherchons des instances de tableaux référencés à l'aide d'un indice calculé à partir d'une valeur d'élément d'un autre tableau. Dans ces cas, ce dernier tableau n'est pas toujours un **paramètre** du sous-programme qui contient le point de référence : parfois c'est la valeur de l'indice qui est passée en paramètre.
- Le deuxième volet de l'analyse de l'accès aux tableaux met simplement dans le même ensemble les sortes de plage de tableau ayant la même sorte d'indice.

Chapitre 9

La spécification concrète de PRCHEN

Résumé

Ce chapitre se veut un guide à l'élaboration de la spécification concrète du code, après les transformations et analyses que nous avons proposées. L'élaboration d'une spécification concrète est illustrée sur le fragment de code qui constitue notre exemple principal. L'ordre des sections du chapitre a été choisi pour faciliter l'explication, non pas parce qu'il faut élaborer les différents aspects de la spécification concrète dans l'ordre dans lequel ils sont traités ici.

La première section sert simplement à préciser l'étendue de la spécification concrète montrée dans l'Annexe J.

La Section 9.2 explique quel fragment de sous-programme est spécifié par chaque opération (sauf les opérations auxiliaires), ainsi que les noms des opérations. Dans la Section 9.3, nous expliquons comment nous avons décidé quelles opérations seraient cachées, et lesquelles définies en "local".

La Section 9.4 décrit la spécification partielle du langage Fortran dont cette spécification est une extension.

Les dimensions du modèle du réseau pris en entrée par ce fragment de code sont stockées dans des variables et servent comme dimensions **effectives** de certains tableaux ou plages de tableaux. Les dimensions fixes des tableaux concernés sont calculés à partir des dimensions maximales permises du modèle du réseau, qui sont constantes. Dans les Sections 9.5 et 9.6, nous expliquons comment nous avons décidé de spécifier cet état de faits.

Dans la Section 9.7, nous expliquons les tuples de sortes que nous sommes obligés de spécifier, et ceux que nous avons décidé de spécifier ou d'inclure dans des profils, et dans la Section 9.8 nous justifions notre décision de spécifier l'engendrement fini de ces tuples.

Finalement, la Section 9.9 explique comment nous avons spécifié les domaines de certaines opérations, une fois leurs profils fixés.

9.1 Les sous-programmes spécifiés

L'objectif de la spécification concrète montrée dans l'Annexe J est la spécification du sous-programme `PRCHEN` par l'opération $PRCHEN$; elle inclut aussi la spécification de certains fragments de **`PRXPLO`**. En effet, `PRCHEN` sert à écrire les fragments de tableau de sorte S_IPCx et S_ICCx qui forment le "noyau" d'un type abstrait de données (voir la Section 8.3 du chapitre précédent). Nous incluons donc dans la spécification concrète montrée dans l'Annexe J, non seulement la spécification de **`PRCHEN`**, mais aussi celle de l'initialisation de ces fragments de tableaux et celle du calcul de leur valeur définitive à travers les appels de **`PRCHEN`**.

9.2 Les opérations non-auxiliaires

Pour chaque fragment de code qui ne constituent pas un sous-programme entier, nous employons dans cette spécification concrète un nom d'opération qui indique le sens que nous avons trouvé pour le fragment, car nous proposons de construire la spécification concrète après l'étude détaillée, manuelle et automatique, du code spécifié, comme expliqué dans le Chapitre 3 (Section 3.7).

La décomposition de `PRXPLO` proposée dans le Chapitre 6 (Section 6.3) nous a aidé à comprendre et à analyser ce sous-programme. Cependant, l'étude détaillée du code effectuée à la main, ainsi que la détection du type abstrait de données représentant la table des co-cycles des sommets (voir la Section 8.3 du Chapitre 8) nous amène à spécifier une décomposition légèrement modifiée. Ainsi le sous-programme `PRXPL1` du Chapitre 6 n'est pas spécifié, mais le fragment de celui-ci qui regroupe l'initialisation et le remplissage des deux tables de listes de numéros de ligne est spécifié par l'opération *construit_tables_cocycles*.

La spécification concrète "factorise" les sous-programmes qui ne diffèrent que par les noms des paramètres et les spécifie par une seule opération. Ainsi les sous-programmes `PRXPL7` et `PRXPL8`, créés par décomposition de `PRXPLO`, sont tous les deux spécifiés par l'opération *table_vide* (mais voir la Section 9.7) et les sous-programmes `PRXPL9` et `PRXPLO` sont spécifiés par *remplit_table_cocycles*.

De façon similaire, le fragment de `PRXPL1` spécifié par *construit_tables_cocycles* peut être décomposé en deux "slices" symétriques, chacune spécifiée par *construit_table_cocycles* (qui contient le mot "table" au singulier).

Le sous-programme `PRCHE1` obtenu par décomposition de `PRCHEN` selon la méthode du Chapitre 6, est spécifié par de `PRXPLO`, sont tous les deux spécifiés par l'opération *table_vide* (mais voir la Section 9.7) et les sous-programmes `PRXPL9` et `PRXPLO` sont spécifiés par *point_inser*, et le corps de la première `IF` de `PRCHEN` par l'opération *insere*.

9.3 Les parties cachées ou définies en locale

Parmi les opérations et prédicats de la spécification concrète, seul $PRCHEN$ correspond exactement à un sous-programme entier du code Fortran. Or les sortes, prédicats et opérations de la spécification concrète devraient spécifier l'interface avec l'extérieur du fragment de code spécifié. Tout prédicat ou opération qui n'est pas caché devrait spécifier un sous-programme qui peut être appelé par un autre fragment de code. Tout élément de toute sorte qui n'est pas cachée devrait avoir une représentation valide dans les structures de données du code. La partie "cachée" de la spécification concrète spécifie la partie "privée" ou intérieure des sous-programmes. C'est pourquoi toutes les opérations autres que $PRCHEN$ sont cachées dans la spécification concrète montrée dans l'Annexe J.

Cette spécification concrète emploie quelques prédicats et opérations auxiliaires. Chacun correspond à deux instructions au plus du code Fortran et représente une opération de bas niveau sur les tables partitionneuses ou les listes non vides sans doublons. Cependant, même certaines de ces opérations sont "spécialisées" à la tâche remplie par `PRXPLO` et `PRCHEN`. C'est pourquoi nous ne donnons pas à ces prédicats et opérations le même statut que celui des autres opérations représentant un fragment plus conséquent du code, mais les définissons en "local".

9.4 La spécification des types et opérations primitives de Fortran

La spécification concrète s'appuie sur la spécification des types et des opérations primitives du Fortran. L'idéal serait de disposer d'une spécification algébrique de tout le langage, mais une telle spécification du Fortran n'existe pas et son élaboration est une tâche non triviale qui sort du cadre de cette thèse. Alors nous proposons de nous contenter des quelques éléments du langage qui nous suffisent pour cet exemple et dont la spécification algébrique est relativement immédiate. Toutefois, nous ne spécifions pas fidèlement le Fortran mais une vision déjà un peu abstraite du langage. En effet, nous faisons abstraction de quelques contraintes du langage de programmation afin de rendre plus claire la spécification des sous-programmes.

Le type `INTEGER` de Fortran et les constantes, opérations arithmétiques et inégalités de base sont spécifiés par la sorte *Int* des entiers et les prédicats et opérations sur les entiers que nous supposons spécifiés dans un module appelé BASICS (qui n'est pas présenté ici).

Le type des tableaux d'entiers de Fortran et leur écriture et lecture sont spécifiés par une instanciation de la spécification générique ARRAY[**sort** *Index*][ELEM] présenté, avec d'autres spécifications "standard", dans l'Annexe I. Dans cette spécification des tableaux, l'opération d'écriture *write __ at __ in __* est totale, mais l'opération de lecture *read __ at __* n'est définie que pour les éléments du tableau déjà écrits (voir Section 8.2 du Chapitre 8). C'est-à-dire que cette spécification n'impose pas une valeur initiale par défaut pour les éléments d'un tableau jamais écrits. Toutefois, nous spécifions par la constante (surchargée) *new* la valeur initiale des tableaux (et non pas des éléments de tableau) de toutes les sortes, pour lesquels la lecture n'est définie pour aucun élément. Ceci nous permet de spécifier l'engendrement fini des sortes des tableaux. En réalité, la valeur initiale des éléments d'un tableau Fortran déclaré mais non encore écrit varie avec le compilateur, voire avec l'état de la mémoire au moment de la définition du tableau. Nous spécifions donc par une constante, une valeur variable mais hors du domaine des opérations observatrices.

Nous nous permettons d'employer non seulement la sorte *Int* qui spécifie le type Fortran `INTEGER`, mais aussi les sous-sortes *Pos* et *Nat* de celle-ci, (que nous supposons spécifiées aussi dans le module Basics) spécifiant les entiers positifs et les naturels respectivement, avec coercition implicite de la sous-sorte vers la sur-sorte. L'emploi de ces sous-sortes nous permet

- de spécifier comme *Pos* la sorte de
  - l'indice d'un tableau déclaré avec 1 comme borne inférieure des indices ou dont la dimension est déclaré par une astérisque et
  - la variable d'une boucle `DO` avec 1 comme borne inférieure, ainsi que
- de spécifier explicitement la coercition de type (vers le sous-type) implicite qui a souvent lieu dans le code Fortran suite à un test.

  Par exemple, dans une branche du code conditionnée par un test de supériorité ou égalité à zéro, le programmeur sait que la variable de type `INTEGER` testée a une valeur positive qui peut donc être employée comme indice d'un tableau dont les indices commencent par 1.

Les sortes *Int*, *Pos* et *Nat*, ainsi que les tableaux indexés par ces sortes sont spécifiés comme infinis. Évidemment, dans un langage de programmation tous les chiffres et tous les tableaux sont bornés et en Fortran, il faut expliciter les bornes des valeurs des indices pour chaque dimension de chaque tableau, mais il s'agit d'un aspect purement opérationnel des codes que nous n'avons pas envie d'ajouter à la spécification.

9.5 Dimensions maximales du réseau électrique

Les variables globales `NS` et `NL` d'*ESTIME*, qui représentent le nombre total de sommets (resp. lignes) dans le réseau électrique, peuvent prendre comme valeur tout entier inférieur ou égal à la constante `NMS` (resp. `NML`) (comme nombre maximal de sommets (resp. lignes)). Dans le code Fortran les limites, `NMS` et `NML`, aux valeurs de `NS` (resp. `NL`) sont nécessaires afin de pouvoir limiter les dimensions de certains tableaux.

Or il serait possible de refléter ces bornes dans les spécifications, notamment en spécifiant des tableaux indexés par des intervalles finis sur *Pos*. Cependant, nous avons décidé que ce serait compliquer les spécifications avec des détails relativement inintéressants. Rien n'empêche d'introduire plus tard ces aspects dans les spécifications, mais nous avons choisi de ne pas spécifier les constantes `NMS` et `NML` et de permettre alors à `NS` et à `NL` de prendre tout entier comme valeur.

9.6 Sorte des numéros de sommet et de ligne

La sorte *S_N1Pos* (resp. *S_IL*) est la sorte des indices des fragments de tableaux de sorte *S_IPCx* (resp. *S_NRLx* ou *S_ICCx*). Or les valeurs des indices de ces tableaux sont les entiers positifs entre 1 et la valeur de la variable globale `NS` (resp. `NL`) compris, à condition que cette dernière soit supérieure à 0. Toutefois, nous ne pouvons pas tenir compte de la borne supérieure variable dans la spécification de ces sortes : ce serait spécifier les "types dépendants" c'est-à-dire un type (en l'occurrence, un intervalle sur les positifs) qui dépend de la **valeur** d'un paramètre d'entrée (`NS` resp. `NL`) du code spécifié. Ni le langage *CASL*, ni la plupart des autres langages de spécification algébrique, ne permettent de spécifier les types dépendants.

En fait, nous n'avons pas besoin de spécifier les types dépendants : l'important est de spécifier comment les valeurs de `NS` et `NL` constituent des **bornes effectives** (et non pas les bornes fixes discutées dans la Section 9.5) aux autres valeurs des sortes S_N1Pos (resp. S_IL) et ainsi aux plages de tableaux de sorte S_IPCx (resp. S_NRLx ou S_ICCx). Ceci est fait soit à travers la spécification des tuples de sortes, soit à travers la spécification du domaine des opérations, comme décrit dans les sections suivantes.

Par ailleurs, grâce aux commentaires du code, nous savons que les variables des sortes S_N1Pos et S_IL représentent des numéros de sommet (resp. de ligne) du réseau électrique, alors nous appelons dorénavant ces sortes $NumS$ (resp. $NumL$). De la même façon, nous remplaçons le nom de sorte S_IL0 par $NumL\_ou\_0$ et S_N1 par S_Val_NRLx (comme sorte des valeurs des plages de tableau de sorte S_NRLx).

Les sortes $NumS$ et $NumL$ sont donc spécifiées dans les modules NumeroSommet et NumeroLigne comme des numéros positifs à travers une égalité de sortes entre $NumS$ (resp. $NumL$) et Pos. De même, la sorte $NumL\_ou\_0$ est spécifiée comme égale à la sorte, Nat, des naturels et par conséquent $NumL$ est une sous-sorte de $NumL\_ou\_0$. Une sorte $NumS\_ou\_neg$ est spécifiée comme égale à la sorte, Int, des entiers et la sorte S_Val_NRLx est spécifiée ensuite comme une sous-sorte de $NumS\_ou\_neg$ et sur-sorte de $NumS$.

9.7 Les tuples

Dans la spécification concrète, nous nous permettons de définir des tuples de valeurs qui n'existent pas explicitement dans le code. Dans le code Fortran, tous les paramètres du même sous-programme forment en quelque sorte un tuple, mais il n'y a pas moyen de grouper certains paramètres à l'intérieur de cet ensemble.

En effet, dans le cas d'un sous-programme Fortran avec paramètres de sortie multiples, nous sommes bien obligés de spécifier le résultat de l'opération correspondante comme un tuple (c'est le cas de `PRCHEN`, par exemple), mais nous pouvons aussi grouper en tuples des paramètres d'entrée ou des sous-ensembles des paramètres de sortie.

Ceci est intéressant quand toutes les combinaisons des valeurs des deux sortes ne sont pas permises dans le tuple. Par exemple, le sous-programme `PRCHEN` a comme paramètres de sortie deux plages de tableau des sortes S_IPCx et S_ICCx. Nous créons donc le tuple de sortes S_IPCx_ICCx pour regrouper dans la spécification les valeurs de ces deux variables en une seule sortie. Or `PRCHEN` a aussi deux paramètres d'entrée de ces mêmes sortes et nous constatons que les paramètres effectifs pouvant instancier ces paramètres d'entrée sont juste

- le résultat d'un autre appel de `PRCHEN` (dans une itération antérieure de la boucle de `PRXPL9` ou de `PRXPL0`) ou
- la paire de valeurs
  - d'une plage de tableau de sorte S_IPCx résultat d'un appel de `PRXPL9` ou de `PRXPL0` et
  - d'une plage de tableau déclarée mais pas encore écrite, de sorte S_ICCx.

  Cette dernière paire de valeurs étant aussi parmi les sorties possibles de `PRCHEN`.

En d'autres termes, les paires de valeurs de ces deux sortes possibles à la sortie de `PRCHEN` sont juste celles possibles à l'entrée. Or ces paires de valeurs sont celles pouvant être interprétées comme une *table partitionneuse* (voir la Section 2.4.1 du Chapitre 2) de listes de numéros de ligne indexée par les numéros de sommet.

C'est pourquoi nous définissons la sorte S_IPCx_ICCx non pas comme le **produit cartésien** des sortes S_IPCx et S_ICCx, mais comme une **sous-sorte** de celui-ci : la sous-sorte permettant de représenter les tables partitionneuses. Les paires de plages d'`IPCO` et d'`ICCO` en entrée de `PRCHEN` appartiennent aussi à cette sous-sorte et nous remplaçons donc dans le profil de `PRCHEN` les deux paramètres d'entrée des sortes S_IPCx et S_ICCx par un seul de sorte S_IPCx_ICCx.

Nous nous permettons aussi de spécifier les fragments de code qui correspondent à peu près aux sous-programmes `PRXPL7` et `PRXPL8` (voir Section 9.2 ci-dessous) par une opération (*table_vide*) avec un résultat non pas de la sorte S_IPCx (trouvée pour `PRXPL7` et `PRXPL8` par l'inférence des profils décrite dans le Chapitre 7), mais de la sorte S_IPCx_ICCx. Cette opération spécifie alors la création d'une table de listes vides.

Nous avons décidé de définir aussi la sorte S_KP_ICCx pour représenter les listes sans doublons non vides pouvant être extraites des tables de sorte S_IPCx_ICCx et notamment toutes les paires de paramètres effectifs instanciant `KP` et `ICCO` de `PRCHE1`. Nous remplaçons donc, dans le profil de `PRCHE1`, les deux paramètres d'entrée de sortes S_KP et S_ICCx par un paramètre de sorte S_KP_ICCx.

Nous proposons de rendre les spécifications plus lisibles en spécifiant toujours de la même façon standardisée tous les tuples de sortes. Nous créons donc une spécification générique du produit cartésien de sortes pour chaque arité de tuple requise (les spécifications PAIR, TRIPLE,...). Le produit cartésien de chaque tuple de sortes est spécifié en instanciant la spécification générique avec la spécification des sortes composantes du tuple. Le nom de la sorte construite est paramétré par les sortes composantes (ex. $Pair[NumL, Array[NumL, NumL\_ou\_0]]$). L'opération de construction de tout produit cartésien est représentée par des $<<\ >>$ et une notation infixe, ex.

$<< KP\ ICCO >>$.

Nous donnons aux projections du tuple des noms standardisés ($1st\_of$ pour la première projection, $2nd\_of$ pour la deuxième,...). Nous évitons d'employer ces projections dans les axiomes, préférant, si possible, dénoter une variable de la sorte du tuple par l'application du constructeur aux composants. Cependant, pour écrire un axiome dans lequel une opération prend en argument un composant d'un tuple résultat d'une autre opération, nous sommes obligés d'employer la projection, par exemple dans l'axiome définissant l'opération *table_vide* dans la spécification concrète de l'Annexe J.

Par ailleurs, en instanciant la spécification générique du tuple, nous pourrions aussi renommer la sorte du tuple, ainsi que les opérations de construction et de projection, de façon à pouvoir employer des noms spécifiques à ce tuple et plus significatifs que les noms "standardisés".

Ensuite, les sortes qui ne regroupent que certaines combinaisons de valeurs des sortes $S_1, S_2, \ldots$ peuvent être spécifiées comme sous-sorte du produit cartésien $S_1 * S_2 * \ldots$ Dans le cas des sortes S_IPCx_ICCx et S_KP_ICCx il s'agit de "produits dépendants" : les valeurs possibles de l'un des composants du tuple dépendent de la valeur d'au moins un autre composant. Notons que d'autres sous-sortes du produit cartésien $S_1 * S_2 * \ldots$ ne sont pas des produits dépendants car elles regroupent les mêmes valeurs que le produit cartésien de certaines sous-sortes de $S_1, S_2, \ldots$.

Évidemment, les sous-sortes du produit cartésien peuvent avoir leur propres constructeurs. Les opérations de projection définies sur le produit cartésien peuvent être appliquées aussi aux sous-sortes de celui-ci, à travers la coercition implicite.

9.8 L'engendrement fini

Nous pouvons définir la sorte S_IPCx_ICCx par la contrainte récursive entre les deux valeurs qui assure qu'elles peuvent être lues comme une table partitionneuse. Mais toutes les valeurs de cette sorte sont créées et écrites uniquement à l'intérieur du sous-programme `PRXPLO` et plus précisément par `PRCHEN` et le fragment de code spécifié par l'opération *table_vide*. Une spécification plus concise de cette sorte est donc qu'elle est finiment engendrée par *table_vide* et $PRCHEN$.

De la même façon, nous définissons la sorte S_KP_ICCx comme celle qui est finiment engendrée à partir de valeurs de la sorte S_IPCx_ICCx par l'opération auxiliaire $list\_at\ \_\ in\ \_$. En effet, toutes les paires de paramètres effectifs instanciant le paramètre d'entrée de sorte S_KP_ICCx de `PRCHE1` sont des résultats de $list\_at\ \_\ in\ \_$. Mais `PRCHE1` est spécifiée par une opération récursive (*point_inser*), dont les arguments doivent représenter non seulement les valeurs de `KP` et d'`ICCO` passées dans l'appel depuis `PRCHEN`, mais aussi leurs valeurs dans les itérations successives de la boucle `DO` de `PRCHE1`. Ces valeurs sont les résultats d'applications successives de l'opération $tail\_of$. À première vue alors, la sorte S_KP_ICCx devrait être finiment engendré par les deux opérations $list\_at\ \_\ in\ \_$ et $tail\_of$. Mais en fait, toutes les listes résultant d'une application de $tail\_of$ pourraient aussi être résultats de $list\_at\ \_\ in\ \_$, alors cette dernière opération suffit pour engendrer toute la sorte S_KP_ICCx.

9.9 Les domaines des opérations

Le *domaine* d'une opération partielle a pour effet d'imposer des contraintes sur les valeurs des arguments qui ne sont pas imposées par leur sorte. Parfois il s'agit d'une contrainte sur la valeur d'un argument seul qui pourrait être exprimée en remplaçant la sorte de l'argument dans le profil par la sous-sorte regroupant les valeurs qui respectent cette contrainte. Dans ce cas, le spécificateur a le choix entre une opération totale avec un certain profil et une opération partielle avec un profil employant la sous-sorte. Lorsqu'il s'agit de contraintes sur les **combinaisons** de valeurs de différents arguments, le spécificateur a le choix entre regrouper les paramètres liés par chaque contrainte dans un tuple (voir la section précédente) afin de définir une opération totale d'arité réduite, ou de spécifier les contraintes à travers le domaine d'une opération partielle. En effet, le domaine d'une opération totale inclut les mêmes valeurs que le produit cartésien des sortes de ses arguments et le domaine d'une opération partielle inclut les mêmes valeurs qu'une sous-sorte du produit cartésien, qui peut être un produit dépendant.

Nous avons expliqué dans le Chapitre 8 (Section 8.2) comment la terminaison des boucles et l'appel d'opérations primitives partielles de Fortran, et notamment la lecture de tableaux, peuvent induire des contraintes sur les valeurs des paramètres d'entrée des sous-programmes.

Le premier exemple présenté dans la Section 8.2 est la contrainte induite par la boucle de `PRCHE1` sur les valeurs en entrée de ses paramètres. L'un des autres exemples est la contrainte sur les valeurs en entrée des paramètres de `PRCHEN` induite par la lecture d'`ICCO`. Or il n'est pas nécessaire d'exprimer ces deux contraintes à travers les domaines de $PRCHEN$ et de *point_inser* (spécifiant `PRCHE1`), car nous avons changé les profils de ces opérations pour y intégrer des paires d'arguments et ces contraintes sont vérifiées par toutes les valeurs des sortes S_KP_ICCx (resp. S_IPCx_ICCx).

Parfois le domaine d'une opération *op* est la conséquence logique du domaine des opérations partielles employées dans sa définition. Dans ce cas nous n'avons pas besoin d'inclure la définition explicite du domaine d'*op* dans la spécification : nous pouvons choisir de l'inclure pour plus de clarté, mais elle est redondante. C'est le cas de l'opération *remplit_table_cocycles* spécifiant `PRXPL9` dans la spécification concrète montrée dans l'Annexe J. Le résultat de *remplit_table_cocycles* est défini si et seulement si

- la valeur de son premier argument est 1 et le résultat d'une certaine application de $PRCHEN$ est défini ou
- la valeur de son premier argument n'est pas 1 et le résultat d'une autre application de $PRCHEN$ est défini.

Or le résultat d'une application d'opération n'est défini que si ses arguments sont définis. Dans la définition de *remplit_table_cocycles* les arguments des applications de $PRCHEN$ intègrent les lectures de `NRLO` qui induisent la contrainte sur les valeurs de `NRLO` et de `NL` expliquée dans le deuxième exemple de la Section 8.2. Cette contrainte est donc impliquée par la définition de *remplit_table_cocycles* et il n'est pas nécessaire de l'intégrer dans une définition explicite du domaine de *remplit_table_cocycles*.

Toutefois, quand la définition d'une opération *op* inclut l'application d'un prédicat au résultat d'une opération partielle, alors il faut définir explicitement le domaine d'*op*. Ceci parce que dans *CASL* le résultat (la vérité) d'un prédicat est toujours défini, même quand il est appliqué à un argument qui n'est pas défini. Le dernier exemple de la Section 8.2 est une contrainte qu'il faut exprimer de façon récursive sur les valeurs en entrée des paramètres de `PRCHEN`. Cette contrainte est exprimée dans la spécification récursive des domaines de $PRCHEN$ et de *point_inser* (spécifiant `PRCHE1`) dans la spécification concrète montrée dans l'Annexe J.

Deuxième partie

Élaboration et justification de la spécification abstraite

Chapitre 10

Élaboration des spécifications abstraites

Résumé

Ce chapitre décrit la spécification la plus abstraite du fragment de code dont nous avons décrit la spécification concrète dans la partie précédente, ainsi que les spécifications intermédiaires et de moins en moins abstraites. Chaque spécification intermédiaire intègre un choix d'implémentation.

Après une introduction expliquant notre démarche, chacune des huit premières sections du chapitre décrit une des spécifications successives, en commençant par la plus abstraite.

La spécification concrète, celles décrites dans ce chapitre et les correspondances entre toutes ces spécifications, constituent le résultat concret de notre démarche de rétro-ingénierie. Avant de procéder, dans la suite de cette partie, à la justification formelle des correspondances entre les spécifications des différents niveaux, nous concluons ce chapitre par un rappel de ce que les spécifications permettent d'élucider, c'est-à-dire : tout ce que la rétro-ingénierie nous a appris sur ce fragment de code.

Introduction

Une fois la spécification concrète élaborée, nous procédons à la spécification de ce que nous croyons être la fonctionnalité du fragment de code étudié, au niveau d'abstraction décrit dans le Chapitre 1. Ensuite, nous devons vérifier la "correction" de cette spécification abstraite du code. Il s'agit d'expliquer comment et dans quelle mesure le code satisfait cette spécification abstraite et, éventuellement, d'indiquer les modifications à apporter à cette dernière.

Dans cette partie, nous allons décrire l'élaboration de la spécification la plus abstraite du fragment de code dont nous avons décrit la spécification concrète dans la partie précédente, ainsi qu'une démarche de justification de cette spécification. Nous espérons que les conclusions que nous tirons de la mise en œuvre de notre approche sur cet exemple seront pertinentes pour de nombreux autres codes.

En s'inspirant du processus de développement de logiciels à l'aide des méthodes formelles décrit dans le Chapitre 1 (Section 1.1), nous décomposons le passage de la spécification la plus abstraite à la spécification concrète en plusieurs étapes d'implémentation, en élaborant plusieurs spécifications intermédiaires. Nous élaborons les spécifications intermédiaires en partant de la spécification la plus abstraite et en "descendant" vers la spécification concrète. Pour chaque choix d'implémentation contribuant à la différence entre la spécification la plus abstraite et la spécification concrète, nous élaborons une nouvelle spécification intermédiaire qui traduit l'effet de ce choix sur la spécification précédente. Nous avons expliqué dans le Chapitre 1 (Section 1.1) que la spécification concrète est *OP-complète* (c'est-à-dire : décrit complètement les procédures de construction des éléments des types abstraits de données et les autres procédures d'accès aux données et ne nécessite plus de choix d'implémentation). En "descendant" vers la spécification concrète, les spécifications intermédiaires successives vont donc comporter de plus en plus de définitions *OP-complètes*, substituées, par des choix successifs d'implémentation, aux définitions déclaratives et imprécises des spécifications plus abstraites.

Les choix d'implémentation nous sont suggérés par notre compréhension du rôle du fragment de code à plusieurs niveaux d'abstraction, suite à notre étude de ce fragment et de tous les sous-programmes que nous croyons participer à la réalisation d'un type abstrait de données (voir la Section 8.3 du Chapitre 8) [1]. Nous allons voir aussi qu'une connaissance des entrées et sorties du code entier est cruciale à l'explication de l'abstraction.

En effet, nous avons compris pendant l'étude du code en préparation de l'élaboration de la spécification concrète (voir la Section 2.4 du Chapitre 2), que

- au niveau le plus abstrait, le fragment de code étudié réalise le calcul du co-cycle des sommets d'un réseau;
- au niveau le moins abstrait, les paramètres formels `IPCO` et `ICCO` de `PRCHEN` réalisent une table partitionneuse de listes;
- à un niveau intermédiaire d'abstraction, `PRCHEN` sert à réaliser l'insertion dans une liste ordonnée de cette table;
- au même niveau d'abstraction que le précédent, l'ordre des listes sert uniquement à optimiser l'algorithme de `CASDEG`,
- la table de listes ne sert donc qu'à stocker des résultats intermédiaires qui pourraient aussi bien être recalculés à chaque fois et
- le code entier prend en entrée une représentation d'un réseau électrique, qu'il modifie et puis sort comme résultat.

Remarquons ici, que l'ordonnancement des décisions d'implémentation n'est pas évident et la complexité des spécifications intermédiaires en dépend beaucoup.

Nous essayons donc de reproduire la démarche supposée par les méthodes formelles d'un développeur de code Fortran qui doit satisfaire la spécification abstraite et qui, par une suite de choix d'implémentation, arrive au fragment de code étudié. Toutefois, nous réalisons des spécifications abstraites dans un style légèrement différent de celui conseillé dans le cadre du développement de logiciels. Ceci à cause de notre désir de confronter le code Fortran à notre intuition concernant les types abstraits de données réalisés.

En effet, il est souhaitable lors de l'élaboration d'une spécification abstraite, de laisser imprécis le plus possible de détails, en d'autres termes, de réaliser la spécification la plus "loose" possible. Par exemple, nous croyons que `PRCHEN` aide au calcul d'ensembles de lignes. Dans le cas du développement d'un logiciel, il serait préférable d'employer dans la spécification la plus abstraite, une spécification générique minimale des ensembles. Par exemple, la spécification juste de la sorte de l'ensemble vide et du prédicat d'appartenance d'un élément à l'ensemble. Mais cette spécification a comme modèles les ensembles, les multi-ensembles, les listes, les listes sans doublons, les listes ordonnées, ...

Pour traduire notre intuition que le code étudié sert à construire des ensembles, et la confronter à la spécification concrète, nous préférons employer la spécification monomorphe (ou *initiale*) qui a les ensembles comme seul modèle. En d'autres termes, nous préférons nous servir de la notion précise et familière d'ensemble plutôt que d'une vague notion de conteneur d'éléments.

Pour la même raison, nous employons à un niveau plus bas d'abstraction la spécification classique des tables au lieu d'une spécification plus "loose".

Quant aux prédicats et opérations des types abstraits de données standard, nous voulons les employer dans les axiomes, mais non pas imposer systématiquement l'implémentation de tous. Nous en cachons donc la plupart, à l'aide du mécanisme de "hiding", comme nous verrons dans les Sections 10.1 et 10.6 ci-dessous.

---

1. En fait, l'idéal serait de prendre comme fragment de code à spécifier, la réalisation du type abstrait de données entier, mais ce fragment est trop important pour exposer son abstraction en détail ici.

10.1 La spécification la plus abstraite : Abstract1

C'est la spécification la plus abstraite du calcul effectué par le fragment de `PRXPLO` compris entre les lignes de code 100 et 119, et par `PRCHEN`. Elle spécifie ce que nous croyons être l'abstraction de la fonctionnalité de ce fragment de code Fortran, à savoir : le calcul du co-cycle (l'ensemble des lignes connectées au sommet) pour chaque sommet du réseau électrique. Elle commence alors par la spécification d'un modèle du réseau électrique composé de lignes et de sommets et qui traduit non seulement ce que nous avons compris de ce fragment de code Fortran, mais aussi des connaissances plus générales acquises de l'étude d'autres parties du code *ESTIME* et auprès des ingénieurs d'EDF. Cependant, ce modèle du réseau électrique est très réduit, et simplifié pour ce qui concerne les sommets.

ABSTRACT1 est présentée dans l'Annexe K et sa structure est illustrée par la Figure 10.1 [2].

10.1.1 La spécification des sommets

Les sommets sont représentés par des objets de la sorte *Sommet* spécifiés dans le module SOMMET-ABSTRAIT. Ils ont divers attributs ; nous ne spécifions ici que l'identificateur et un autre attribut, auquel nous donnons le nom symbolique $attribS1$, pour indiquer qu'il pourrait le premier de plusieurs autres attributs. Ces deux attributs sont représentés sous forme des opérations totales unaires *id_de* et $attribS1\_de$ sur *Sommet*. La sorte *Sommet* est *partitionnée par id_de* : c'est-à-dire que chaque objet de sorte *Sommet* (nous dirons dans la suite “sommet” au lieu d'“objet de la sorte *Sommet*”) a un identificateur qu'il ne partage avec aucun autre sommet.

Notons que cette spécification des sommets permet à un sommet de combiner n'importe quelles valeurs des attributs autres que l'identificateur et notamment, à deux sommets aux identificateurs différents d'avoir les mêmes valeurs pour tous leurs autres attributs.

10.1.2 La spécification des lignes

Les lignes sont représentées par des objets de la sorte *Ligne* spécifiés dans le module LIGNE-ABSTRAITE. Une ligne est orientée (avec une origine et une extrémité) et sert, *a priori*, à connecter deux sommets. Cependant, une ligne peut ne pas être connectée, à son origine comme à son extrémité. Les objets de la sorte *Ligne* (nous dirons “lignes”) ont donc une propriété qui est le fait d'être connectée ou non à l'origine, représentée par le prédicat unaire *est_connectee_orig* sur *Ligne*, et une autre propriété “symétrique” qui est le fait d'être connectée ou non à l'extrémité(représentée par *est_connectee_extrem*). En cas de connexion à l'origine (resp. extrémité), une ligne a comme attribut le sommet auquel elle est connectée. Ces attributs sont donc représentés par les deux opérations partielles unaires *origine_de* et *extremite_de* sur *Ligne*, définies juste en cas de connexion. Parmi les autres attributs des lignes, nous spécifions ici l'identificateur et un autre attribut nommé symboliquement $attribL1$. Les deux étant représentés par des opérations totales unaires sur *Ligne*.

Les lignes ne peuvent pas combiner n'importe quelles valeurs des attributs : en cas de connexion à l'origine et à l'extrémité, l'origine de la ligne doit être différente de l'extrémité. Rien n'empêche deux lignes d'identificateurs différents d'être connectées “en parallèle” aux mêmes sommets origine et extrémité. Par ailleurs, contrairement au cas des sommets, deux lignes différentes peuvent partager le même identificateur, sous condition que les seules propriétés qui changent de valeur entre les deux lignes sont la connexion ou non à l'origine ou à l'extrémité. Ceci parce que le code spécifié [3] doit pouvoir représenter une modification de l'état de connexion d'une ligne physique modélisée.

En effet, chaque état ou “vue” d'une même ligne physique modélisée est représenté par sa propre valeur de sorte *Ligne*. Plusieurs valeurs peuvent représenter successivement donc la même ligne physique au cours du calcul. Cependant, chaque ligne physique a son propre identificateur unique, alors les différents objets représentant différents états d'un même ligne physique partagent la même valeur de cet attribut.

2. Notez que dans les illustrations de structure des spécifications, nous n'incluons pas les paramètres dans les noms des spécifications génériques (telles que SET). Par ailleurs, le module SET est représenté par un rectangle à coins arrondis car il s'agit d'un module “standard” présenté dans l'Annexe (I) et non pas dans l'Annexe K. Les flèches représentent l'extension d'un module par un autre. Plusieurs flèches qui se joignent à un point représentent l'instanciation d'une spécification générique.

3. En fait, il ne s'agit pas ici du fragment spécifié, mais d'autres fragments de `PRXPLO` et d'autres sous-programmes d'*ESTIME*.

Notons que les sommets spécifiés dans SOMMETABSTRAIT sont, par contre, immuables : en spécifiant la partition de la sorte *Sommet* par la valeur de l'identificateur, nous interdisons l'existence de deux "vues" du même sommet physique [4].

10.1.3 La spécification du réseau

Les réseaux sont représentés par des objets de sorte *Reseau* spécifiés dans le module RESEAU-ABSTRAIT. Un réseau "contient" un ensemble de sommets et un ensemble de lignes comme spécifié par les opérations
$sommets\_de : Reseau \rightarrow Set[Sommet]$ et
$lignes\_de : Reseau \rightarrow Set[Ligne]$.
Un réseau peut regrouper n'importe quels ensembles de sommets et de lignes pourvu que les contraintes axiomatisées dans RESEAUABSTRAIT soient respectées :

- les lignes peuvent être connectées uniquement à des sommets appartenant aussi au réseau,
- deux lignes ne peuvent pas être connectées aux mêmes sommets avec des orientations différentes et
- deux lignes du même réseau ne peuvent pas partager le même identificateur.

Remarquons

- que la sorte *Reseau* regroupe des réseaux de toutes les topologies possibles (et non pas seulement celle du "réseau français") et
- que chaque objet de sorte *Reseau* "inclut" les propriétés des lignes y appartenant et des sommets connectés à ces lignes, donc représente un seul **état** de ces objets physiques.

10.1.4 La spécification des ensembles de lignes

Comme expliqué au début de ce chapitre, les ensembles de lignes sont spécifiés par instanciation de la spécification générique "standard" SET[ELEM]. Cependant, nous ne voulons pas imposer l'implémentation de tous les prédicats et opérations standard de ce type abstrait de données. Nous cachons donc, à l'aide du mécanisme du "hiding" tous les prédicats et opérations sur les ensembles. En effet, nous n'avons pas besoin de spécifier la construction d'ensembles arbitraires de lignes et leurs observateurs classiques, mais seulement la construction des ensembles de lignes sortant d'un sommet ou y entrant.

10.1.5 La spécification des lignes entrant ou sortant d'un sommet

Le module ENSEMBLESLIGNESSORTANTETENTRANT s'appuie sur la spécification précédente du réseau pour spécifier les opérations
$lignes\_sortant : Sommet * Reseau \rightarrow ?\ Set[Ligne]$ et [5]
$lignes\_entrant : Sommet * Reseau \rightarrow ?\ Set[Ligne]$.
Ces opérations ne sont définies que si le sommet passé en argument est connectée à une ligne appartenant au réseau. Elles fournissent l'ensemble des lignes connectées à l'origine (resp. extrémité) au sommet donné, et à l'extrémité (resp. origine) à un autre sommet quelconque, c'est-à-dire, le co-cycle.

Les définitions des deux opérations sont symétriques, ce qui n'oblige pas des implémentations symétriques mais en fait nous verrons dans la suite que les implémentations que nous proposons pour les deux sont symétriques. Par conséquent, il suffira souvent de décrire l'implémentation de *lignes_sortant*.

Rappelons que ces deux opérations spécifient le calcul principal effectué par le fragment de code spécifié.

4. En réalité, dans un contexte plus large que celui de PRXPLO, les sommets ne sont pas immuables, mais nous nous permettons cette simplification qui ne change rien à notre problème et qu'il sera facile de lever par la suite.
5. Le symbole $\rightarrow ?$ est employé dans *CASL* pour indiquer une opération partielle.

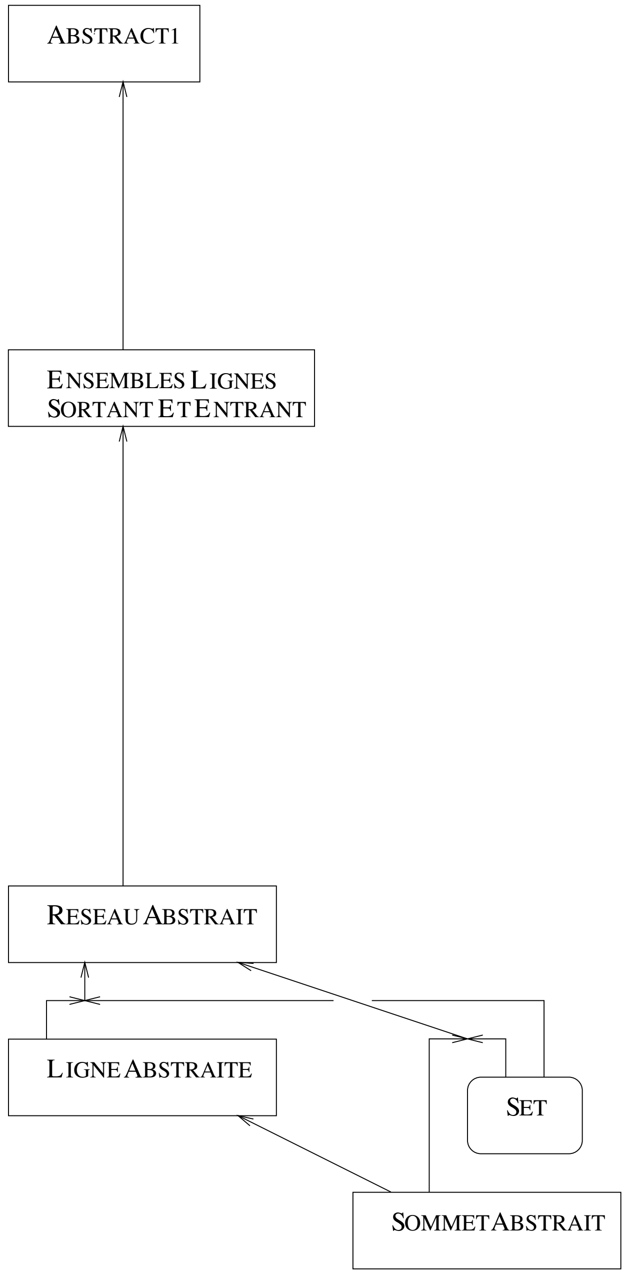


FIG. 10.1: Structure d'ABSTRACT1

10.2 La spécification Abstract2 : l'implémentation du réseau

10.2.1 La décision d'implémentation

Cette étape d'implémentation traduit la décision de regrouper dans la structure de données de sorte *Reseau* les informations sur tous les sommets et toutes les lignes du réseau. Les sommets et les lignes de chaque réseau sont numérotés, ce qui permet de **représenter chaque sommet ou ligne par son numéro dans un réseau donné** car ce numéro sert d'indice dans le réseau de toutes les informations sur le sommet ou la ligne.

10.2.2 Modification structurelle

La spécification ABSTRACT2 est présentée dans l'Annexe L et sa structure est illustrée par la Figure 10.2.

Le module RESEAUABSTRAIT de la spécification ABSTRACT1, ainsi que les modules SOMMET-ABSTRAIT et LIGNEABSTRAITE qu'il enrichit, sont ici remplacés par IMPLRESEAU, IMPLSOMMETSDE-RESEAU et IMPLLIGNESDERESEAU, ainsi que les modules (NUMEROSOMMET, NUMEROLIGNE et ARRAY) qu'ils enrichissent.

Le module ENSEMBLESLIGNESSORTANTETENTRANT est remplacé par ENSEMBLESNUMSLIGNES-SORTANTETENTRANT.

10.2.3 Motivation

L'implémentation représentée par ABSTRACT2 est motivée par le constat que les opérations *lignes_sortant* et *lignes_entrant* ne sont définies que pour les sommets **du réseau** et rendent un ensemble de lignes **du même réseau**. Pour le calcul du résultat de ces opérations, il suffit de représenter les sommets et lignes appartenant au réseau donné en entrée.

Nous implémentons le réseau comme une "base de données" de toutes les valeurs des attributs de ses sommets ainsi que de ses lignes. Chaque sommet ou ligne du réseau a un numéro qui sert de "pointeur" dans cette base. À partir du réseau, nous pouvons récupérer toutes les informations sur les sommets et lignes qui sont nécessaires à notre calcul.

L'intérêt de cette implémentation est de permettre une optimisation des codes qui manipulent un petit nombre de réseaux différents. Dans ces codes, chaque réseau est représenté par une structure de données coûteuse en mémoire, mais les autres structures (tuples d'arguments, ensembles,...) représentant plusieurs sommets ou lignes du même réseau peuvent être composées de simples pointeurs (les numéros de sommet ou de ligne).

10.2.4 La spécification des numéros de sommet et de ligne

Chaque sommet (resp. ligne) du réseau est numéroté(e) avec un numéro compris dans l'intervalle entre 1 et le nombre total de sommets (resp. lignes) dans le réseau. Deux sommets (resp. lignes) du même réseau ne partagent jamais le même numéro. La numérotation des sommets (resp. lignes) d'un réseau donné définit donc une *bijection* entre

- un intervalle (débutant par 1) sur les numéros positifs et
- les sommets (resp. lignes) du réseau.

Cette implémentation est intéressante pour les codes qui servent à "modifier" les valeurs de certains attributs de certains sommets ou de certaines lignes du réseau : d'une valeur de réseau en entrée ce genre de code calcule une autre valeur de réseau, mais sans ajouter ou retirer des sommets ou des lignes, ni changer **la bijection entre les numéros de sommet (resp. ligne) et leur identificateur**. Or nous avons vu dans la section 10.1 qu'il y a une autre bijection entre les identificateurs de sommet ou ligne et les objets physiques représentés. Si le réseau est implémenté de la façon que nous décrivons ici, alors ce genre de code manipule plusieurs valeurs de sorte *Reseau*, mais dans lesquelles un même indice indique différentes "vues" du même sommet ou ligne physique.

Les numéros de sommet appartiennent à la sorte $NumS$ spécifiée dans le module NUMEROSOMMET et les numéros de ligne à la sorte $NumL$, spécifiée dans NUMEROLIGNE. Nous avons expliqué dans la description de la spécification concrète (voir Chapitre 9) que $NumS$ et $NumL$ y sont spécifiées comme des sortes égales à Pos.

10.2.5 La spécification des tableaux

Le réseau est réalisé sous la forme de plusieurs tableaux, chacun servant à stocker la valeur pour chaque sommet (resp. ligne) de l'un des attributs des sommets (resp. lignes) spécifiés dans ABSTRACT1. (sauf en ce qui concerne l'exception expliquée dans la Section 10.2.6 ci-dessous). Ces tableaux sont indexés par les numéros de sommet en ce qui concerne les tableaux stockant les attributs de sommet, et par les numéros de ligne pour les tableaux stockant les attributs de ligne.

Une conséquence de cette implémentation est qu'il suffit de passer juste les tableaux concernés, au lieu de tout le réseau, aux fragments de code qui ne lisent que certains attributs des sommets ou des lignes.

Les tableaux sont des instanciations de la spécification générique ARRAY [**sort** $Index$][ELEM] déjà décrite dans le Chapitre 9 (Section 9.4).

10.2.6 La spécification du réseau

Le réseau contient un tableau par attribut de sommet ou de ligne sauf en ce qui concerne la connexion à l'origine et à l'extrémité des lignes. Dans ces cas, un seul tableau sert à stocker le fait d'être connecté ou non, ainsi qu'un pointeur vers le sommet en cas de connexion. Il y a donc deux tableaux, l'un représentant l'origine des lignes et l'autre leur extrémité, qui stockent pour chaque ligne du réseau

- soit une valeur de 0 en cas de non connexion,
- soit le numéro du sommet connecté.

Les opérations
$\_\_.NS : Reseau \to NumS\_ou\_0$ et
$\_\_.NL : Reseau \to NumL\_ou\_0$
fournissent le nombre total de sommets (resp. lignes) du réseau et ainsi, au cas où le nombre total n'est pas 0, le numéro de sommet (resp. ligne) maximal (le numéro minimal étant toujours 1).

10.2.7 La spécification des numéros des lignes sortant ou entrant

À cause de la bijection pour chaque réseau entre les sommets (resp. lignes) du réseau et les numéros de sommet (resp. ligne) pour ce réseau, les opérations
$lignes\_sortant : Sommet * Reseau \to ?\ Set[Ligne]$ et
$lignes\_entrant : Sommet * Reseau \to ?\ Set[Ligne]$
du module ENSEMBLESLIGNESSORTANTETENTRANT d'ABSTRACT1 peuvent être remplacées ici par les opérations
$nums\_lignes\_sortant : NumS * Reseau \to ?\ Set[NumL]$ et
$nums\_lignes\_entrant : NumS * Reseau \to ?\ Set[NumL]$
d'ENSEMBLESNUMSLIGNESSORTANTETENTRANT. Ces dernières calculent les ensembles de numéros de ligne sortant (resp. entrant) pour chaque numéro de sommet du réseau.

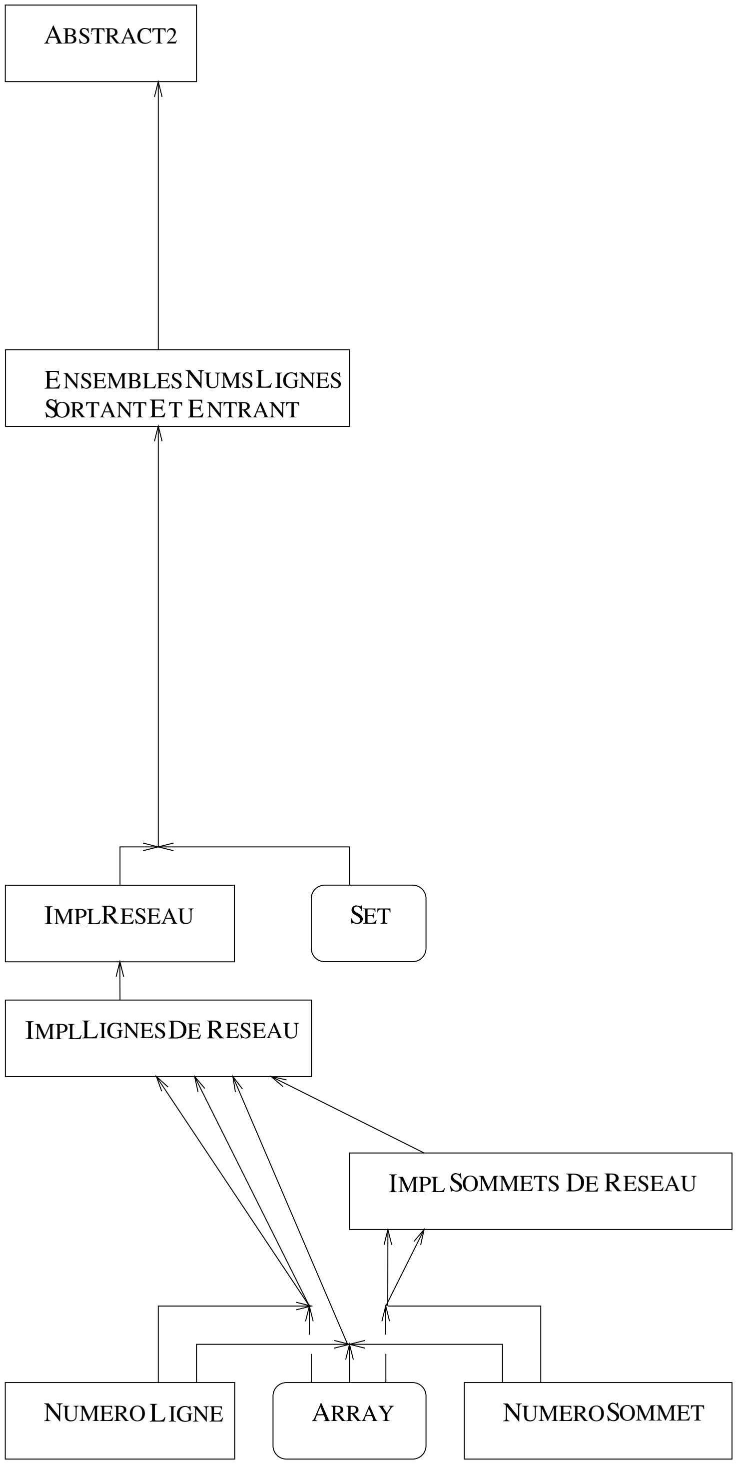


Fig. 10.2: Structure d'Abstract2

10.3 La spécification Abstract3 : l'implémentation des ensembles

10.3.1 La décision d'implémentation

Cette étape d'implémentation traduit la décision d'implémenter les ensembles de numéros de ligne par des listes sans doublons de numéros de ligne.

10.3.2 Modification structurelle

La spécification ABSTRACT3 est présentée dans l'Annexe M et sa structure est illustrée par la Figure 10.3 [6].

Le module ENSEMBLESNUMSLIGNESSORTANTETENTRANT d'ABSTRACT2 est remplacé ici par LISTESNUMSLIGNESSORTANTETENTRANT.

Le module SET[ELEM] d'ABSTRACT2 est remplacé par LISTDIFFELEMS[ELEM], ainsi que le module LIST[ELEM] qu'il enrichit.

10.3.3 Motivation

Afin d'arriver à une implémentation d'ABSTRACT2 traduisible en langage de programmation, nous devons spécifier une procédure calculatoire pour le stockage des numéros de ligne appartenant aux ensembles crées par *nums_ligne_sortant*. Il est évident aussi que la lecture des résultats de *nums_ligne_sortant* nécessitera une procédure calculatoire de parcours à la place du prédicat *is_in*. Il s'agit alors de simuler les ensembles spécifiés par la spécification standard SET par une autre structure de données qui se prête plus facilement aux procédures calculatoires de stockage et de parcours.

10.3.4 La spécification des listes sans doublons

Nous commençons par la spécification générique "standard" des listes, LIST[ELEM], dans laquelle *List*[*Elem*] est la sorte des listes d'éléments de sorte *Elem*. La spécification générique "standard" LISTDIFFELEMS[ELEM] enrichit celle-ci par une définition de la sorte *LDE*[*Elem*] des listes sans doublons, comme une sous-sorte de *List*[*Elem*]. C'est la spécification la plus abstraite possible des listes sans doublons : aucun prédicat ni opération spécifique n'est défini dans LISTDIFFELEMS. Implicitement, la sorte *LDE*[*Elem*] "hérite" de l'engendrement libre par le résultat de la projection d'*empty* depuis *List*[*Elem*] vers *LDE*[*Elem*] et par la restriction de *cons* à *LDE*[*Elem*], avec projection du résultat. Une injection des objets de sorte *LDE*[*Elem*] dans la sorte *List*[*Elem*] est aussi spécifiée implicitement par la déclaration de sous-sorte, permettant d'appliquer les prédicats *is_empty* et *is_in* aux objets de *LDE*[*Elem*].

Cependant, nous ne voulons pas imposer l'implémentation du type abstrait de données "standard" spécifié par LIST, et nous cachons donc *List*[*NumL*] dans ABSTRACT3. Ceci a l'effet de cacher aussi les prédicats et opérations définies sur *List*[*NumL*] [7].

10.3.5 La spécification des listes sans doublons des numéros des lignes sortant ou entrant

Dans le module LISTESNUMSLIGNESSORTANTETENTRANT, l'opération

$nums\_ligne\_sortant : NumS * Reseau \rightarrow ?\ Set[NumL]$

du module ENSEMBLESNUMSLIGNESSORTANTETENTRANT d'ABSTRACT2 est remplacée par l'opération

$nums\_ligne\_sortant : NumS * Reseau \rightarrow ?\ LDE[NumL]$

qui construit des listes sans doublons de numéros de lignes sortant. La situation est analogue en ce qui concerne les lignes entrant.

6. Notez que le module IMPLRESEAU est représenté par un rectangle hachuré car il appartient aussi à la spécification ABSTRACT2 et par conséquent est montré dans l'Annexe L mais pas dans l'Annexe M. Par ailleurs les modules enrichis par IMPLRESEAU n'apparaissent que dans la Figure 10.2, car les étapes ultérieures d'implémentation n'y touchent pas.

7. C'est pour cette raison que le module LIST est montré comme caché dans la Figure 10.3. Toutefois, cette manière d'illustrer les parties cachées d'une spécification est forcément approximative car nous ne pouvons pas illustrer de cette façon le "hiding" de certains symboles (tels que les prédicats et opérations sur *Set*[*Elem*] cachés dans ABSTRACT1), d'un module comportant d'autres symboles qui ne sont pas cachés.

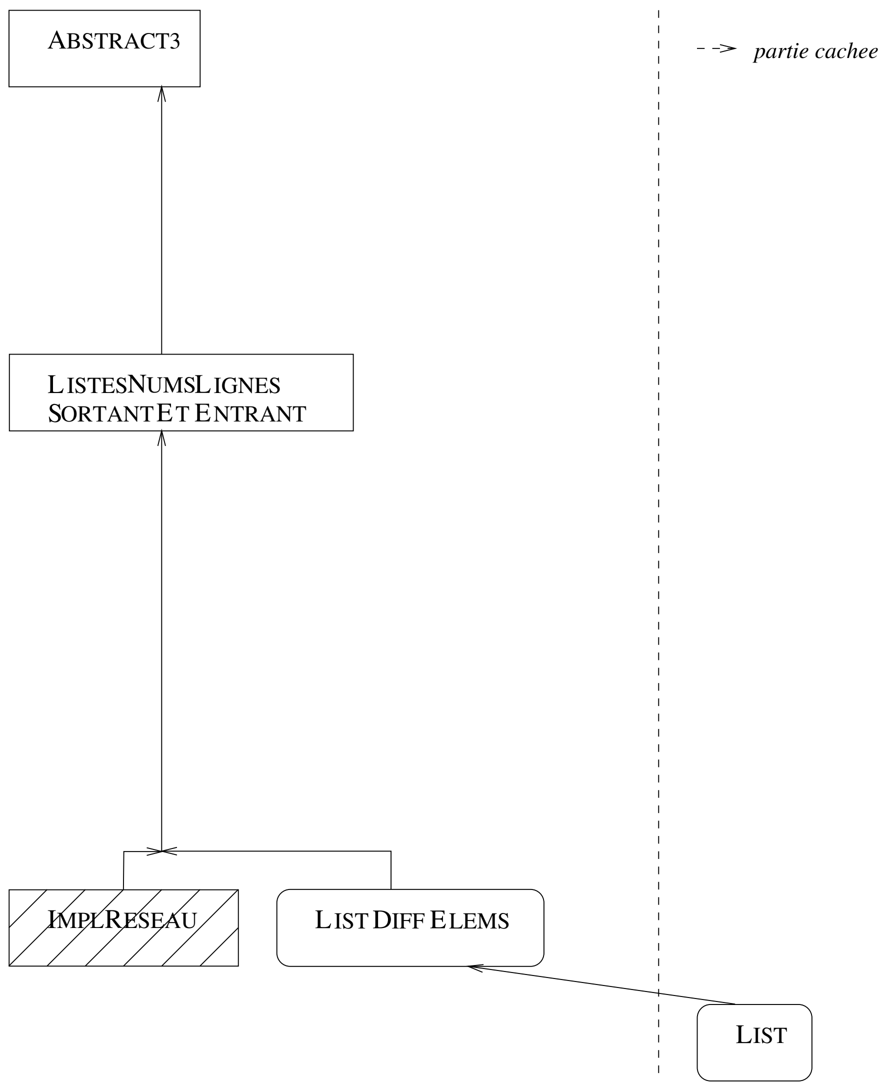


FIG. 10.3: Structure d'ABSTRACT3

10.4 Abstract4 : spécification imprécise des ordres dans les listes

10.4.1 La décision d'implémentation

Cette étape d'implémentation traduit la décision d'ordonner les listes de numéros de ligne sortant de façon à regrouper les numéros des lignes connectées à un même sommet extrémité et les listes de numéros de ligne entrant de façon à regrouper les numéros des lignes connectées à un même sommet origine.

10.4.2 Modification structurelle

La spécification ABSTRACT4 est présentée dans l'Annexe N et sa structure est illustrée par la Figure 10.4.

ABSTRACT4 est juste une extension d'ABSTRACT3, qui sert à la préciser un peu plus. Le module IMPLRESEAU d'ABSTRACT3 est étendu dans ABSTRACT4 à RESEAUAVECORDRES et la spécification générique standard LISTDIFFELEMS[ELEM] à LISTDIFFELEMSWITHPARAMETRISEDORDER[ELEMWITHPARAMETRISEDORDER]. Ce dernier module est instancié par RESEAUAVECORDRES et ensuite étendu à LISTENUMSLIGNEAVECORDREEXTREM et LISTENUMSLIGNEAVECORDREORIG.

LISTESNUMSLIGNESSORTANTETENTRANTAVECORDRES est l'extension des deux instanciations de LISTDIFFELEMSWITHPARAMETRISEDORDER par RESEAUAVECORDRES. Il remplace LISTESNUMSLIGNESSORTANTETENTRANT d'ABSTRACT3.

10.4.3 Motivation

Dans ABSTRACT3 nous avons spécifié les éléments des listes construites par les opérations *nums_ligne_sortant* et *nums_ligne_entrant*, mais nous n'avons pas défini l'ordre des éléments. Or il n'est pas possible d'arriver à un algorithme déterministe pour la construction de ces listes, et donc à une spécification que nous pouvons traduire en code, sans décider l'ordre des éléments dans les listes. En fait, notre choix d'implémentation est orienté par des considérations d'efficacité, car les aspects opérationnels interviennent de plus dans les spécifications intermédiaires successives.

10.4.4 Les critères d'ordonnancement choisis

Nous choisissons pour chaque cas (lignes sortant et lignes entrant) un critère qui facilite le comptage par l'opération implémentée par le sous-programme `CASDEG` des *liaisons* sortant ou entrant d'un noeud, `CASDEG` étant l'un des deux "utilisateurs" du fragment de code spécifié ici.

En effet, `CASDEG` calcul le *degré* d'un sommet, c'est-à-dire le nombre de liaisons sortant ou entrant. Une liaison est une connexion entre deux sommets et peut donc correspondre à plusieurs lignes en parallèle. `CASDEG` parcourt donc la liste de numéros de lignes sortant (resp. entrant) d'un sommet, en incrémentant son compteur de liaisons à chaque fois qu'il s'agit d'une ligne connectée (à l'autre bout) à un nouveau sommet extrémité (resp. origine). L'algorithme est plus efficace si la liste de numéros de lignes est construite de telle façon que les numéros des lignes constituant une liaison y sont contigus.

Notons qu'il n'est pas possible d'exprimer de façon procédurale l'algorithme efficace de comptage employé dans `CASDEG`, ni les critères d'ordre des éléments des listes de numéros de ligne sortant et entrant dont il dépend, à un niveau plus élevé d'abstraction. Ceci parce qu'ils concernent l'ordre des éléments dans une liste, et non pas une propriété d'ensembles.

10.4.5 La spécification des ordres

Dans le module RESEAUAVECORDRES d'ABSTRACT4 nous spécifions l'ordre

$\_\_ <_e \_\_ \; dans \; \_\_ : NumL * NumL * Reseau$

pour les éléments des listes de numéros de ligne sortant d'un même sommet, ainsi que l'ordre

$\_\_ <_o \_\_ \; dans \; \_\_ : NumL * NumL * Reseau$

pour les éléments des listes de numéros de ligne entrant dans un même sommet.

Il s'agit d'ordres stricts mais partiels : tous les numéros de lignes connectées aux deux extrémités doivent être comparables par les deux ordres.

Les ordres sont spécifiés de façon "loose", mais en précisant qu'ils doivent avoir pour effet de regrouper les numéros des lignes connectées au même sommet extrémité (resp. origine) : autrement dit, si trois numéros de ligne sont rangés dans l'ordre $\_ <_e \_$ *dans* r pour un réseau r, et si la ligne du premier numéro et celle du troisième numéro sont connectées au même sommet extrémité, alors celle du deuxième numéro aussi. La spécification ne précise pas l'ordre relatif des numéros de lignes d'une même liaison (connectées au même sommet extrémité), ni l'ordre des liaisons.

10.4.6 La spécification des listes ordonnées

Les ordres $\_ <_e \_$ *dans* _ et $\_ <_o \_$ *dans* _ ne sont pas des vraies relations d'ordre spécifiées par une spécification "standard" d'ordre strict partiel. Les vraies relations d'ordre sont binaires (dans ce cas, elles auraient deux arguments de sorte $NumL$) mais les ordres que nous avons spécifiés sur les numéros de ligne dépendent de propriétés des **lignes indiquées par ces numéros dans un réseau donné** et varient donc de réseau en réseau. Le module ElemWithParametrisedOrder spécifie donc une relation d'"ordre" à trois arguments qui est anti-symétrique, anti-réflexive et transitive (comme une vraie relation d'ordre binaire) si nous fixons la valeur du troisième argument.

ListDiffElemsWithParametrisedOrder définit un prédicat caractérisant les listes ordonnées par un ordre à trois arguments, ainsi qu'une opération d'insertion qui respecte cet ordre et permet alors la construction de listes ordonnées. Notons que, contrairement au cas des vraies relations d'ordre binaires, nous ne pouvons pas caractériser **par une sorte** les listes ordonnées par $\_ <_e \_$ *dans* _ (resp. $\_ <_o \_$ *dans* _) **pour une certaine valeur du réseau** car il s'agirait de spécifier un *type dépendant* (voir Section 9.6 du Chapitre 9). Alors $LDE[NumL]$ est la sorte la plus petite possible pour les listes ordonnées de cette façon.

ListDiffElemsWithParametrisedOrder est instancié de deux façons différentes par ReseauAvecOrdres, une fois pour chaque ordre.

10.4.7 La spécification des listes ordonnées de numéros de ligne sortant ou entrant

Finalement, ListesNumsLignesSortantEtEntrantAvecOrdres étend le module ListesNumsLignesSortantEtEntrant d'Abstract3 par la spécification des ordres des listes de numéros des lignes sortant et entrant.

Les ordres sur les numéros de ligne, les prédicats caractérisant les listes ordonnées par ces ordres et les opération d'insertion les respectant sont tous cachés dans Abstract4. Nous les employons dans les axiomes mais ne voulons pas imposer leur implémentation en tant que sous-programmes Fortran.

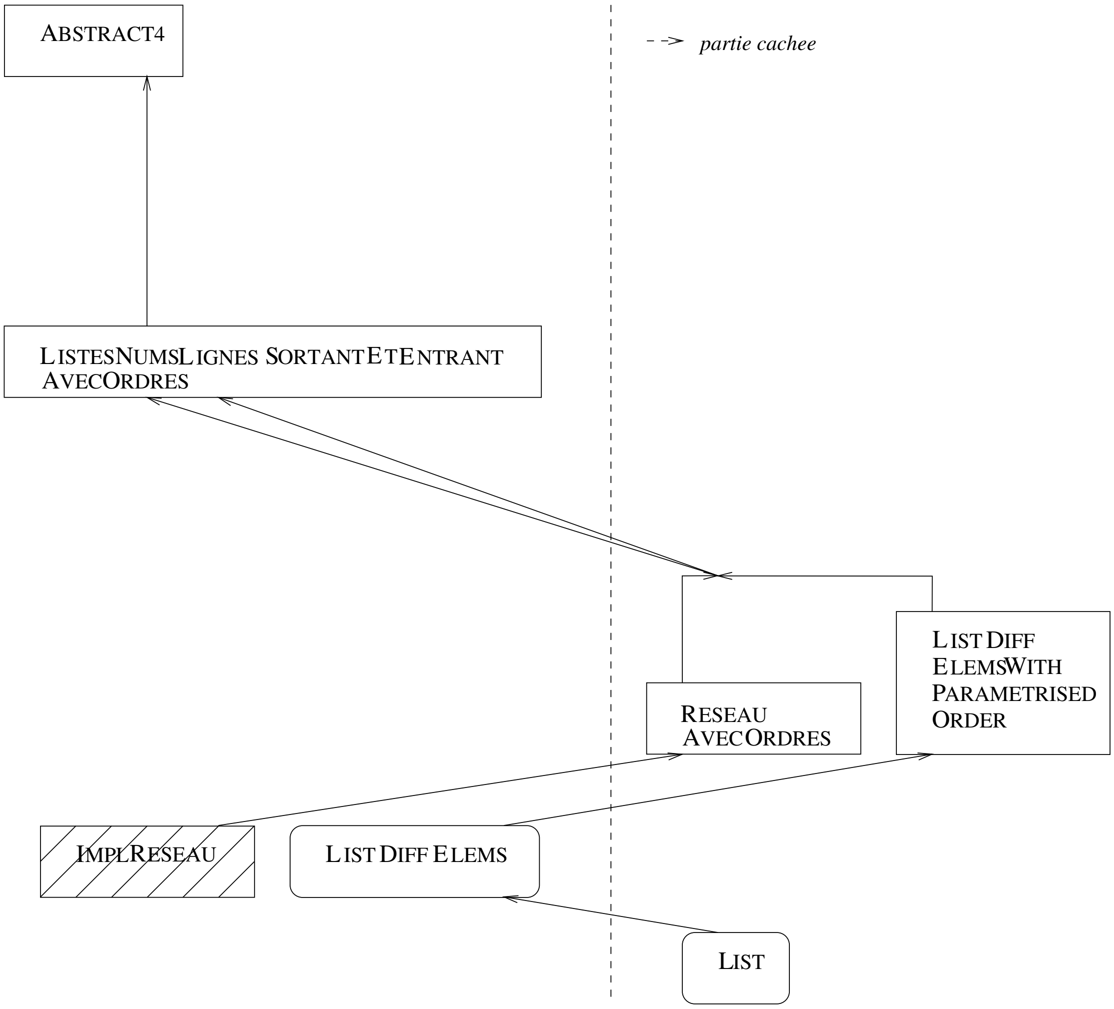


FIG. 10.4: Structure d'ABSTRACT4

10.5 La spécification Abstract5 : l'implémentation des ordres

10.5.1 La décision d'implémentation

Cette étape d'implémentation traduit un choix d'implémentation calculatoire, en terme d'opérations de base sur les tableaux, naturels et positifs, des deux ordres partiels sur les numéros de lignes spécifiés de façon "loose" par ABSTRACT4.

10.5.2 Modification structurelle

La spécification ABSTRACT5 est présentée dans l'Annexe O et sa structure est illustrée par la Figure 10.5.

Les modules RESEAUAVECORDRES et LISTESNUMSLIGNESSORTANTETENTRANTAVECORDRES d'ABSTRACT4 sont remplacés dans ABSTRACT5 par RESEAUAVECORDRESIMPLEMENTES et LISTES-NUMSLIGNESSORTANTETENTRANTAVECORDRESIMPLEMENTES, respectivement. Les deux instanciations de LISTDIFFELEMSWITHPARAMETRISEDORDER par RESEAUAVECORDRES d'ABSTRACT4 sont "factorisées" dans ABSTRACT5 par une seule instanciation par RESEAUAVECORDRESIMPLEMENTES.

10.5.3 Motivation

Dans ABSTRACT4, deux ordres sur les numéros de lignes, paramétrés par le réseau, étaient spécifiés de façon imprécise. Il s'agit ici de préciser la spécification et de l'exprimer en terme d'une comparaison de valeurs lues dans des tableaux composant le réseau, afin de pouvoir la traduire en code.

Par ailleurs, en étendant de façon symétrique la spécification des deux ordres symétriques d'ABSTRACT4, nous les "factorisons" par un seul ordre. Celui-ci n'est pas paramétré par le réseau, mais par l'un des deux tableaux symétriques du réseau résultats des opérations $\_.NRLO$ et $\_.NRLE$.

10.5.4 L'implémentation des ordres

Dans RESEAUAVECORDRESIMPLEMENTES, les deux ordres
$\_ <_e \_ \; dans \; \_ : NumL * NumL * Reseau$ et
$\_ <_o \_ \; dans \; \_ : NumL * NumL * Reseau$
spécifiés dans RESEAUAVECORDRES sont implémentés par des comparaisons des valeurs stockées pour les numéros de ligne concernés, dans l'un des tableaux faisant partie de l'implémentation du réseau.

Dans le cas du premier ordre, il s'agit du tableau résultat de l'opération $\_.NRLE$ qui stocke le numéro de sommet connecté à l'extrémité de chaque ligne. Dans le cas de l'autre ordre, il s'agit du tableau résultat de l'opération $\_.NRLO$ qui stocke le numéro de sommet connecté à l'origine de chaque ligne. Dans le cas des deux ordres, si la même valeur est stockée dans le tableau pour les deux numéros de ligne, alors l'ordre habituel sur les positifs est employé.

Comme les deux ordres sont engendrés de la même façon par des tableaux différents, tous les deux peuvent être spécifiés en terme d'un troisième ordre
$\_ <_a \_ \; dans \; \_ : NumL * NumL * Array[NumL, NumS\_ou\_0]$
paramétré non pas par le réseau, mais par un tableau.

En fait, les ordres $\_ <_e \_ \; dans \; \_$ et $\_ <_o \_ \; dans \; \_$ étaient cachés dans ABSTRACT4 et ici ils ne sont même pas spécifiés. Il suffit de spécifier $\_ <_a \_ \; dans \; \_$ et de modifier les axiomes.

Ainsi, les listes de numéros de ligne ordonnées par $\_ <_e \_ \; dans \; \_$ ou par $\_ <_o \_ \; dans \; \_$ sont ordonnées aussi par $\_ <_a \_ \; dans \; \_$ pour des tableaux différents. Par conséquent, au lieu de spécifier deux opérations
$insert\_ord\_extrem : NumL * LDE[NumL] * Reseau \to ?\; LDE[NumL]$ et
$insert\_ord\_orig : NumL * LDE[NumL] * Reseau \to ?\; LDE[NumL]$,
et deux prédicats
$\_ \; est\_ord\_extrem\_dans \; \_ : LDE[NumL] * Reseau$ et
$\_ \; est\_ord\_orig\_dans \; \_ : LDE[NumL] * Reseau$,
comme dans ABSTRACT4, il nous suffit dans ABSTRACT5 de spécifier (toujours de façon déclarative) la seule opération
$insert : NumL * LDE[NumL] * Array[NumL, NumS\_ou\_0] \to ?\; LDE[NumL]$

et le seul prédicat
$\_\_ \; est\_ord\_dans \; \_ : LDE[NumL] * Array[NumL, NumS\_ou\_0]$.

C'est ce prédicat qui est employé dans les axiomes de ListesNumsLignesSortantEtEntrantAvecOrdresImplementes définissant l'ordre des listes des numéros des lignes sortant et entrant.

Comme dans Abstract4, nous cachons l'ordre sur les numéros de ligne, le prédicat caractérisant les listes ordonnées par cet ordre et l'opération d'insertion le respectant.

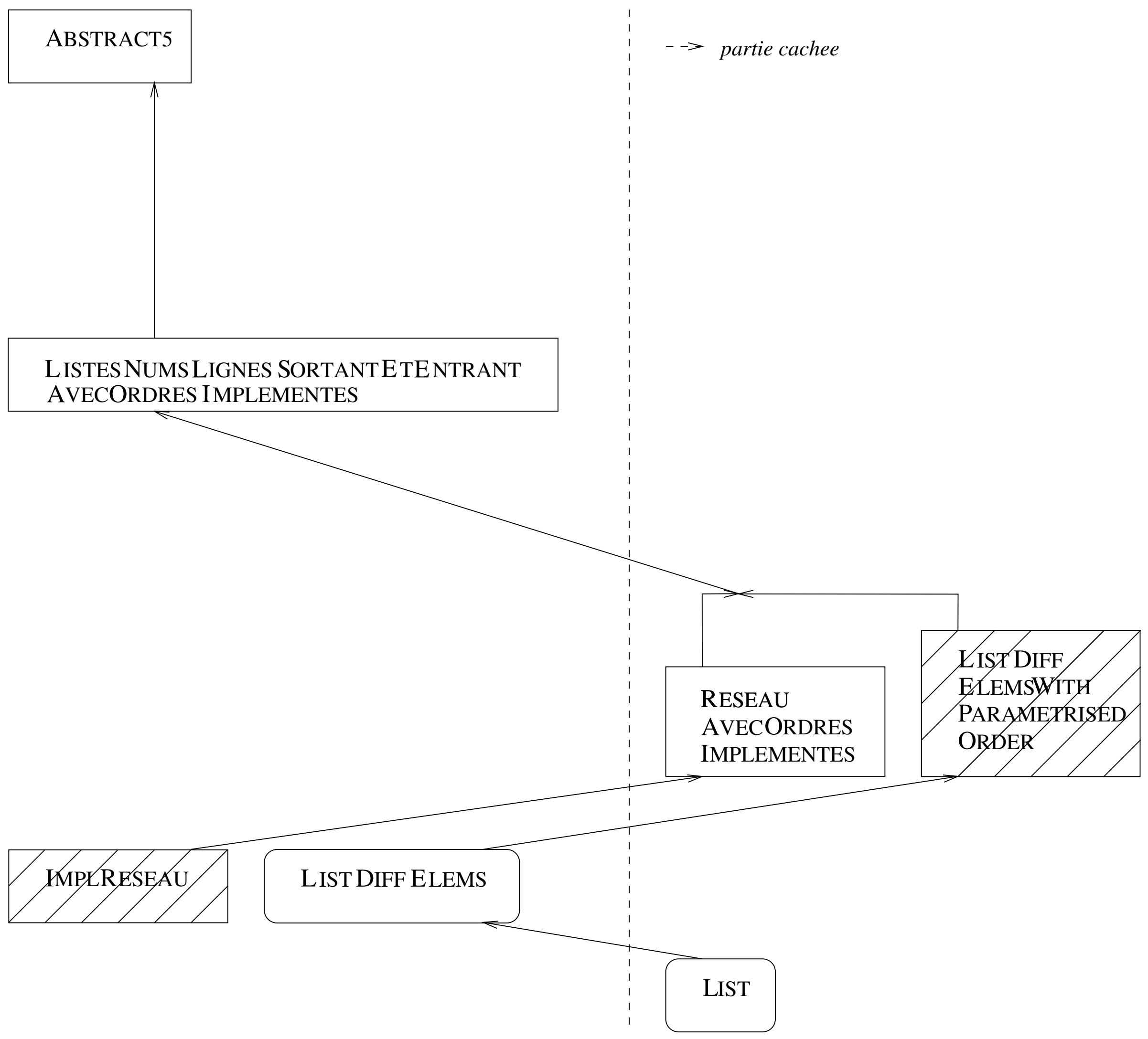


FIG. 10.5: Structure d'ABSTRACT5

10.6 La spécification Abstract6 du stockage des listes

10.6.1 La décision d'implémentation

Cette étape d'implémentation traduit la décision de pré-calculer et de stocker dans deux tables, les listes de numéros de ligne sortant et entrant pour tous les numéros de sommet du réseau.

10.6.2 Modification structurelle

La spécification ABSTRACT6 est présentée dans l'Annexe P et sa structure est illustrée par la Figure 10.6.

Le module LISTESNUMSLIGNESSORTANTETENTRANTAVECORDRESIMPLEMENTES de ABSTRACT5 est remplacé dans ABSTRACT6 par CONSTRUCTABLENUMSLIGNESSORTANTOUENTRANT, ainsi que les modules (TABLENUMSLIGNE et TABLE [**sort** *Key*][**sort** *Item*]) qu'il enrichit.

10.6.3 Motivation

Cette étape d'implémentation représente un premier pas vers une procédure de calcul des numéros de ligne sortant ou entrant. Son intérêt est purement opérationnel : elle permet au code spécifié d'effectuer une fois pour toutes, et de façon efficace, le calcul des listes des numéros des lignes entrant et sortant pour tous les sommets d'un réseau donné et d'en mémoriser les résultats, au lieu de refaire le calcul à chaque appel de la fonction spécifiée par *nums_lignes_sortant* ou *nums_lignes_entrant*.

En effet, la seule procédure de construction de la liste des numéros des lignes sortant ou entrant pour un numéro de sommet donné, consiste à parcourir toutes les lignes du réseau, à tester la connexion de chacune au sommet de ce numéro, et à ajouter un par un les bons numéros de ligne à la liste (à l'aide de l'opération *insert*, par exemple). Comme nous serons amenés à appeler la fonction spécifiée par *nums_lignes_sortant* de nombreuses fois sur le même sommet ou pour de nombreux sommets différents, alors il est beaucoup plus économique en temps de calcul d'effectuer un seul parcours des lignes afin de calculer les numéros de ligne sortant pour tous les sommets à la fois.

Cette étape d'implémentation décompose donc le calcul des numéros de ligne sortant ou entrant d'un sommet en trois étapes, qu'il faut réaliser dans l'ordre mais qu'il n'est pas nécessaire de réaliser ensemble. En effet, les deux premières étapes peuvent être réalisées une fois pour toutes et la troisième autant de fois que nécessaire. Ces trois étapes sont :

1. l'extraction du réseau des composants nécessaires à la construction des tables (par les opérations déjà définies $\_\_.NS$, $\_\_.NL$, $\_\_.NRLO$ et $\_\_.NRLE$)
2. la construction des tables stockant le graphe des fonctions *nums_lignes_sortant* ou *nums_lignes_entrant* pour un réseau donné (par l'opération *construit_table_cocycles* définie dans CONSTRUCTABLENUMSLIGNESSORTANTOUENTRANT) et
3. la lecture d'une table pour connaître des numéros de ligne sortant ou entrant pour un numéro de sommet donné du réseau (par l'opération *retrieve* définie dans TABLE).

Les opérations *nums_lignes_sortant* et *nums_lignes_entrant* qui correspondent à la composition des opérations de ces trois étapes ne sont même pas spécifiées dans ABSTRACT6, afin de permettre la réalisation dispersée des trois étapes.

Dans l'étape d'implémentation précédente, nous avons profité de la symétrie des numéros de ligne sortant et entrant en spécifiant un seul ordre paramétré. Dans celle-ci, nous en profitons en spécifiant une seule table, une seule opération de construction et une seule opération de lecture.

10.6.4 La spécification d'une table de listes sans doublons de numéros de ligne indexée par les numéros de sommet

Cette table est spécifiée par l'instanciation d'une spécification générique "standard" et déclarative d'une table TABLE [**sort** *Key*][**sort** *Item*] comportant la définition de la table initiale vide (la constante *new*) et les opérations d'ajout d'un item à la table (*add*) et de la lecture de l'item référencé par une clef donnée (*retrieve*). Hormis l'ordre des arguments des opérations, cette spécification est en fait une simple traduction de ARRAY [**sort** *Index*][Elem]. Nous cachons dans ABSTRACT6 les opérations définies sur la table, afin de les employer dans les axiomes sans imposer leur implémentation.

Dans TABLENUMSLIGNE, les clefs sont instanciées par les numéros de sommet et les items par les listes sans doublons de numéros de ligne.

10.6.5 La spécification de la construction des tables de listes des numéros des lignes sortant ou entrant

Le module CONSTRUCTABLENUMSLIGNESSORTANTOUENTRANT, définit l'opération

$$\begin{aligned}construit\_table\_cocycles : {} & NumS\_ou\_0 * NumL\_ou\_0 * \\ & Array[NumL, NumS\_ou\_0] * Array[NumL, NumS\_ou\_0] \\ & \rightarrow ?\ Table[NumS, LDE[NumL]]\end{aligned}$$

qui construit une table de listes sans doublons de numéros de ligne, indexée par les numéros de sommet, dont les éléments des listes et leur ordre sont déterminés par les deux tableaux passés en argument. D'abord, cette opération initialise la table en y ajoutant une liste vide pour chaque numéro de sommet (spécifiée à l'aide de l'opération auxiliaire récursive *empty __ upto __*). Ensuite, elle parcourt tous les numéros de ligne et pour chaque ligne connectée aux deux extrémités, insère son numéro dans la liste appropriée de la table (spécifiée à l'aide de l'opération auxiliaire récursive *remplit_table_cocycles*). L'insertion du numéro de ligne dans la table en respectant l'ordre sur les numéros de ligne, est spécifiée (à l'aide de l'opération auxiliaire *insertTab*) comme la composition des opérations *retrieve*, *insert* et *add*. L'opération *construit_table_cocycles* est donc définie de façon calculatoire, mais en terme d'opérations primitives qui, elles, sont définies de façon déclarative (dans TABLE et LISTDIFFELEMSWITHPARAMETRISEDORDER).

Appliquée deux fois aux composants d'un même réseau (mais avec les tableaux inversés), *construit_table_cocycles* sert à construire deux tables stockant les listes des numéros de lignes sortant et entrant pour chaque numéro de sommet du réseau.

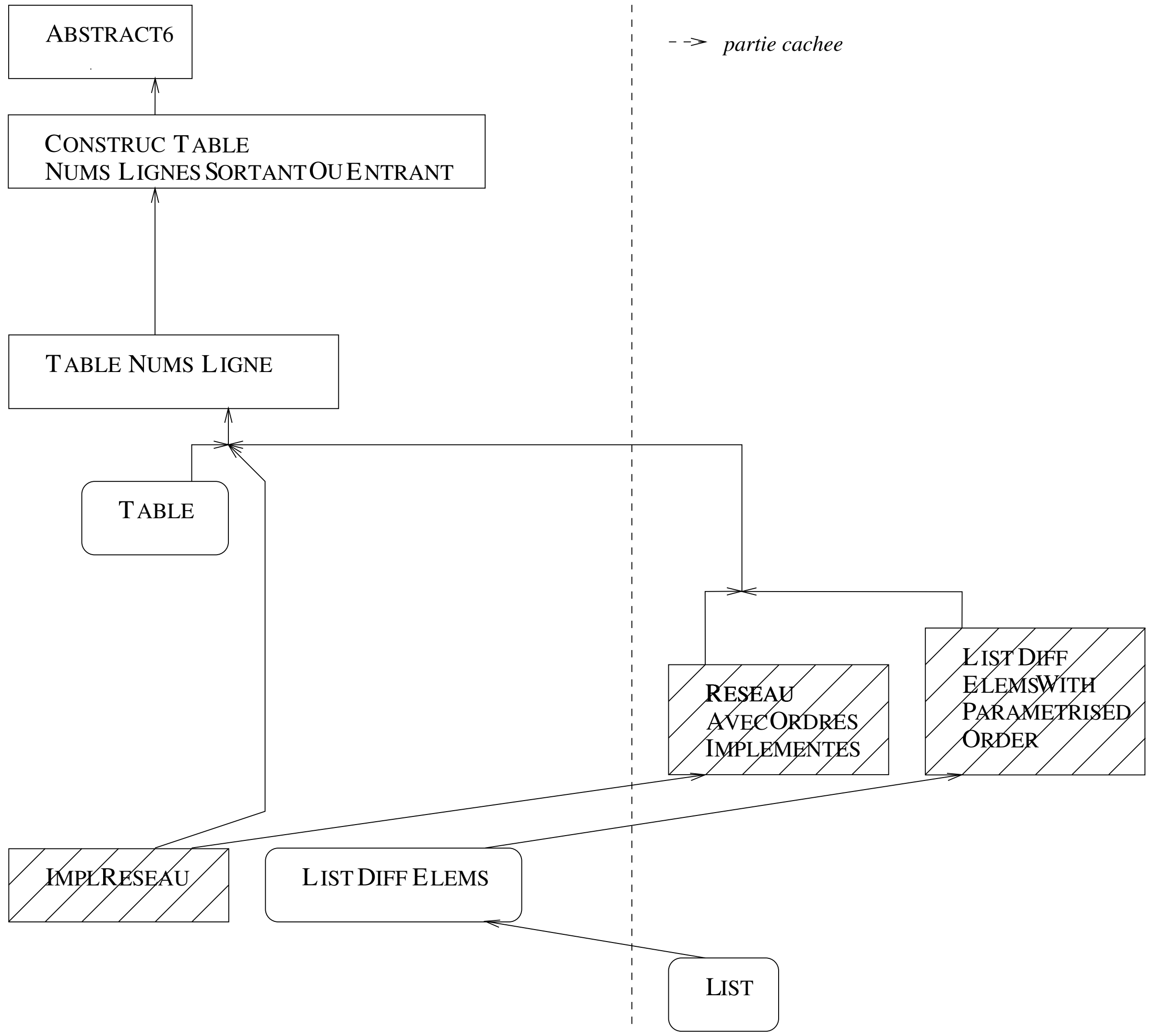


FIG. 10.6: Structure d'ABSTRACT6

10.7 Abstract7 : tables partitionneuses et algorithme d'insertion

10.7.1 La décision d'implémentation

Cette étape d'implémentation traduit le choix d'employer une certaine classe de tables (les tables partitionneuses) pour le stockage des numéros de ligne sortant ou entrant, et un algorithme d'insertion de chaque numéro de ligne s'appuyant sur les opérations primitives de cette classe de tables.

10.7.2 Modification structurelle

La spécification ABSTRACT7 est présentée dans l'Annexe Q et sa structure est illustrée par la Figure 10.7.

Les modules CONSTRUCTABLEPARTITIONNEUSENUMSLIGNESSORTANTOUENTRANT et INSERENUMLIGNE d'ABSTRACT7 remplacent CONSTRUCTABLENUMSLIGNESSORTANTOUENTRANT, LISTENUMSLIGNEAVECORDREIMPLEMENTE, RESEAUAVECORDRESIMPLEMENTES et LISTDIFFELEMSWITHPARAMETRISEDORDER d'ABSTRACT6.

TABLE et TABLENUMSLIGNE sont remplacées par OPTABLEPARTITIONINGLISTS et TABLEPARTITIONNEUSENUMSLIGNE.

10.7.3 Motivation

Dans la table la plus générale spécifiée par TABLE [**sort** *Key*][**sort** *Item*], la seule façon d'ajouter un élément à une liste item de la table, est de construire une nouvelle liste contenant l'élément et d'ajouter toute cette liste à la table à la place de l'ancienne.

Cette étape d'implémentation est motivée par le constat que les tables de listes des numéros des lignes sortant ou entrant ont comme propriété d'être *partitionneuses* (voir la Section 2.4.1 du Chapitre 2). Or les tables de listes partitionneuses peuvent être implémentées d'une façon qui, non seulement est très économique en espace mémoire, mais aussi permet une implémentation efficace des opérations d'ajout d'un seul élément à une liste de la table, ne nécessitant pas le remplacement d'une liste entière par une autre.

Le choix des tables partitionneuses pour le stockage des listes des numéros des lignes sortant ou entrant conduit donc à l'emploi des primitives d'insertion d'un seul élément dans la procédure d'insertion de chaque numéro de ligne sortante ou entrante.

10.7.4 La spécification des tables partitionneuses

Les tables partitionneuses peuvent être vues comme une sous-sorte des tables générales spécifiées par TABLE [**sort** *Key*][**sort** *Item*], mais dans OPTABLEPARTITIONINGLISTS nous spécifions directement les tables partitionneuses de façon opérationnelle, sans nous appuyer sur TABLE [**sort** *Key*][**sort** *Item*]. En effet, les tables partitionneuses sont spécifiées avec une opération *retrieve* qui est la restriction de celle des tables générales et avec la même constante *new*, mais avec leurs propres constructeurs : *empty* _ *at* _, *consTab* et *insert* _ *after* _ *in* _ remplacent l'*add* des tables générales.

10.7.5 La spécification de l'algorithme d'insertion

Dans INSERENUMLIGNE l'opération auxiliaire *insertTab* du module CONSTRUCTABLENUMSLIGNESSORTANTOUENTRANT d'ABSTRACT6 est spécifiée de façon calculatoire comme opération non cachée. La spécification ne s'appuie plus sur l'opération *insert* spécifiée de façon déclarative sur les listes ordonnées dans LISTDIFFELEMSWITHPARAMETRISEDORDER[ELEMWITHPARAMETRISEDORDER] d'ABSTRACT6, mais implémente directement

- l'insertion de l'élément en tête de la liste appropriée à l'aide de la primitive *consTab* spécifiée dans OPTABLEPARTITIONINGLISTS ou sinon,

– le parcours la liste lue dans la table à la recherche du bon point d'insertion (à l'aide de opération auxiliaire récursive $point\_inser$), suivi par l'insertion du numéro de ligne à ce point à l'aide de la primitive $insert \_\_ after \_\_ in \_\_$.

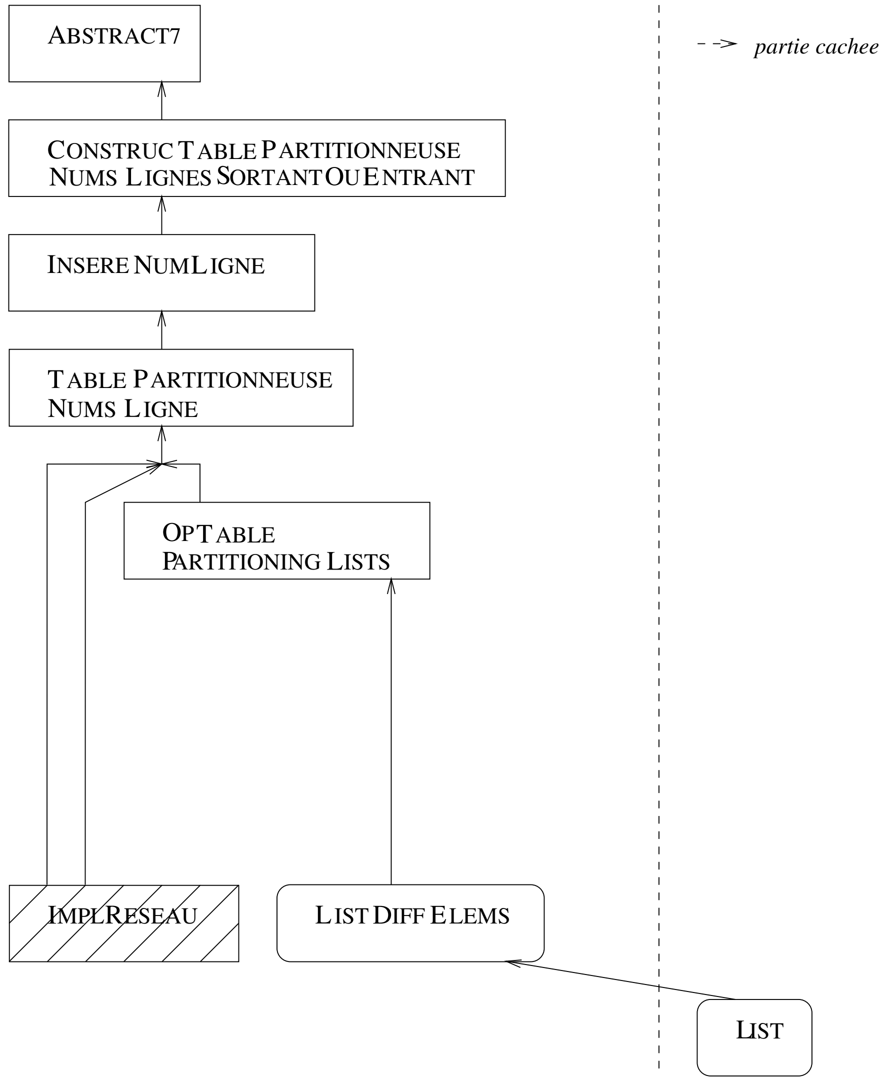


Fig. 10.7: Structure d'Abstract7

10.8 Abstract8 : l'implémentation des tables partitionneuses

10.8.1 La décision d'implémentation

Cette étape d'implémentation traduit le choix d'implémentation des tables de listes partitionneuses de numéros de ligne indexées par les numéros de sommet, ainsi que les opérations définies dessus.

10.8.2 Modification structurelle

La spécification ABSTRACT8 est présentée dans l'Annexe R et sa structure est illustrée par la Figure 10.8.

Les modules LISTDIFFELEMS[ELEM] et OPTABLEPARTITIONINGLISTS [**sort** *Key*][ELEM] d'ABSTRACT7, ainsi que le module caché LIST[ELEM] qu'ils enrichissent, sont remplacés dans ABSTRACT8 par IMPLLISTDIFFELEMS[ELEMORNIL] et IMPLTABLEPARTITIONINGLISTS [**sort** *Key*] [ELEMORNIL]

Par conséquent, les modules TABLEPARTITIONNEUSENUMSLIGNE, INSERENUMLIGNE et CONSTRUCTABLEPARTITIONNEUSENUMSLIGNESSORTANTOUENTRANT d'ABSTRACT7 doivent être remplacés par IMPLTABLENUMSLIGNE, IMPLINSERENUMLIGNE et IMPLCONSTRUCTABLENUMSLIGNESSORTANTOUENTRANT. Ces derniers sont identiques aux modules qu'ils remplacent, sauf qu'ils enrichissent IMPLTABLEPARTITIONINGLISTS au lieu d'OPTABLEPARTITIONINGLISTS.

10.8.3 Motivation

Il s'agit de définir une implémentation *OP-complète* et ne nécessitant plus de choix d'implémentation, en terme des prédicats et opérations de base sur les positifs, naturels et tableaux. Celle choisie est économique en mémoire et comprend une implémentation efficace des opérations spécifiées dans OPTABLEPARTITIONINGLISTS, ainsi que des observateurs des listes stockées.

10.8.4 La spécification d'un schéma d'implémentation partielle des listes

La spécification IMPLLISTDIFFELEMS[ELEMORNIL] spécifie un schéma d'implémentation des listes sans doublons spécifiées par LISTDIFFELEMS[ELEM] dans le contexte des tables de listes sans doublons.

ELEMORNIL spécifie une nouvelle sorte *Elem_or_nil*, qui est l'union de la sorte *Elem* des éléments de la liste et d'une constante (pour représenter la fin de la liste) *nil*. La sorte *Elem* est donc sous-sorte d'*Elem_or_nil*. Il s'agit d'une généralisation de la relation entre les sortes *Pos* et *Nat*.

Nous spécifions les tableaux de sorte *Array*[*Elem*, *Elem_or_nil*], indexés par des objets de sorte *Elem* et stockant des objets de sorte *Elem_or_nil*. La liste sans doublons est alors implémentée par certaines paires composées d'un objet de sorte *Elem_ou_nil* et d'un tableau de sorte *Array*[*Elem*, *Elem_or_nil*].

Nous spécifions dans IMPLLISTDIFFELEMS[ELEMORNIL] juste le schéma d'implémentation

– de la sorte *LDE*[*Elem*] des listes sans doublons et

– des observateurs sur *LDE*[*Elem*].

Nous ne spécifions pas l'implémentation des constructeurs *empty* et (la restriction de) *cons* sur les listes sans doublons, car le constructeur (et observateur) *tail_of* sur les listes sans doublons et l'opération *retrieve* sur les tables de listes sont les seuls constructeurs nécessaires de listes sans doublons.

En d'autres termes, IMPLLISTDIFFELEMS[ELEMORNIL] spécifie **l'interprétation comme liste sans doublons**, de certaines paires d'objets de sortes *Elem_ou_nil* et *Array*[*Elem*, *Elem_or_nil*].

La liste est réalisée sous forme de liste chaînée de la façon décrite dans la Section 2.4.1 du Chapitre 2 (et illustrée dans le schéma de la Figure 2.3, en supposant que la valeur zéro est spécifiée par *nil*, les naturels par *Elem_ou_nil* et les positifs par *Elem*).

De cette façon, nous pouvons interpréter

– comme la liste vide, toute paire dont le premier composant est *nil*

– et comme liste sans doublons non vide toute paire respectant la condition suivante :

le premier composant n'est pas *nil* et si nous prenons

– comme premier élément de la liste la valeur du premier composant de la paire (projeté sur *Elem*), v_1, et
– comme *n*ième élément de la liste la valeur v_n indexée dans le tableau par v_{n-1}, sauf si celle-ci est *nil*,

alors la liste est sans "boucles" (toutes les valeurs v_n sont différentes).

Pour que la liste soit finie, il faut ajouter à cette condition le fait que la lecture "enchaînée" des valeurs que nous venons de décrire, termine toujours par la lecture de la valeur *nil*.

En caractérisant de cette façon les paires qui peuvent être interprétées comme une liste sans doublons, nous ignorons, évidemment, les valeurs dans le tableau indexées par des indices autres que les éléments de la liste.

10.8.5 La spécification d'un schéma d'implémentation des tables

La spécification IMPLTABLEPARTITIONINGLISTS [**sort** *Key*][ElemOrNil] spécifie un schéma d'implémentation des tables de listes partitionneuses par certaines paires de tableaux.

Les listes stockées dans la table sont réalisées de la façon décrite ci-dessus. La nature partitionneuse de l'ensemble des listes permet alors de stocker dans le même tableau les queues de toutes les listes non vides de la table, comme il est décrit dans la Section 2.4.1 du Chapitre 2 (et illustré dans le schéma de la Figure 2.4).

La double interprétation des valeurs de sorte *Elem*

– comme élément de la liste

– et aussi comme pointeur vers l'élément suivant

et le stockage des queues de toutes les listes dans le même tableau, rendent cette implémentation particulièrement économique en espace mémoire.

Contrairement au schéma d'implémentation de la liste sans doublons, IMPLTABLEPARTITIONINGLISTS [**sort** *Key*][ElemOrNil] spécifie non seulement l'implémentation des observateurs mais aussi celle des constructeurs. La sorte $TabPartList[Key, Elem]$ regroupe juste les paires de tableaux construites par les constructeurs spécifiés. Par ailleurs, l'engendrement fini des listes sans doublons par l'opération *retrieve* est spécifié ici.

En effet, *retrieve* construit la liste référencée dans la table par la clef k, en formant la paire de valeurs composée

– de la valeur dans le premier tableau indexée par k

– et du deuxième tableau.

et qui représente (de la façon décrite dans la section précédente) une liste sans doublons.

Remarquons que représenter le deuxième composant de chaque liste par le deuxième tableau de la paire représentant la table, n'est possible qu'en ignorant les valeurs indexées dans ce tableau par des indices autres que les éléments de la liste. Voici l'intérêt de l'implémentation de la liste sans doublons expliquée dans la section précédente.

10.8.6 Instanciation du schéma d'implémentation de la table

Le module IMPLTABLENUMSLIGNE instancie le schéma d'implémentation spécifiée par IMPLTABLEPARTITIONINGLISTS [**sort** *Key*][ElemOrNil] :

– la sorte *Key* de la spécification générique est instanciée par *NumS*,

– la sorte *Elem* par *NumL* et

– la sorte *Elem_or_nil* par *NumL_ou_0*.

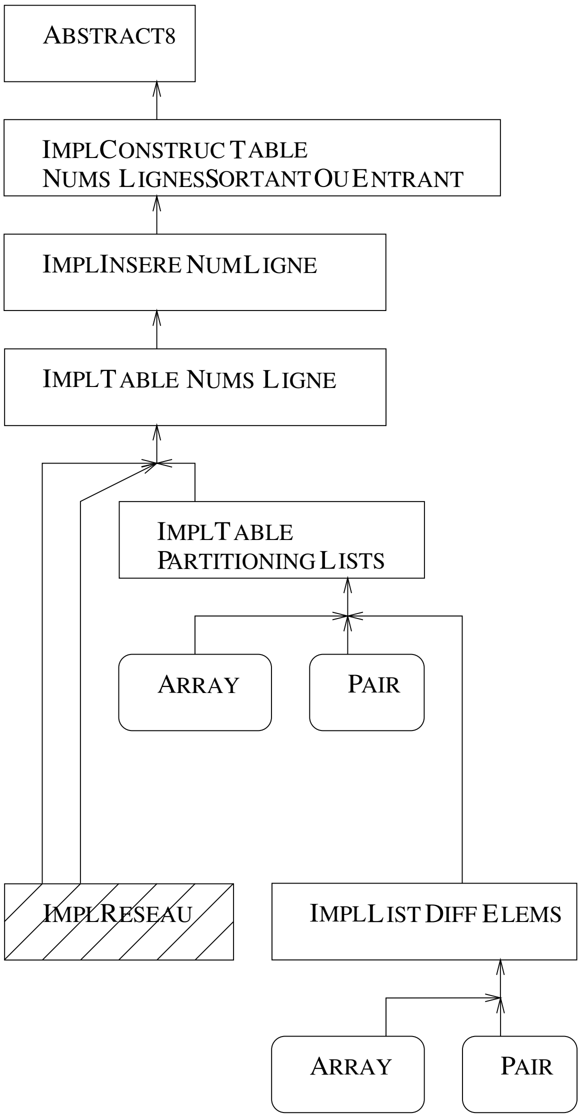


FIG. 10.8: Structure d'ABSTRACT8

Conclusion

ABSTRACT8 est l'ultime spécification intermédiaire entre ABSTRACT1 et la spécification concrète, car nous avons réussi à "expliquer" entièrement la spécification concrète en termes d'ABSTRACT1 à travers les choix d'implémentation concrétisés par les spécifications intermédiaires successives.

En prenant en compte les correspondances naturelles entre les paires de spécifications successives, notre démarche a permis d'expliciter :

- le modèle du réseau de distribution ;
- l'emploi des tables de listes de positifs pour stocker le *co-cycle* de chaque sommet pour un réseau donné (et d'indiquer l'intérêt du stockage du co-cycle de tous les sommets du réseau) ;
- l'ordre des éléments de ces listes (et d'indiquer, quand les spécifications décrites ici sont mises avec celles du sous-programme `CASDEG`, l'intérêt de cet ordre dans le calcul du *degré* de chaque sommet) ;
- sous quelles conditions le sous-programme `PRCHEN` effectue l'insertion de chaque élément de ces listes, de façon à respecter cet ordre ;
- la propriété d'une table de listes de positifs qui conditionne son implémentation par deux tableaux de naturels indexés par les positifs ;
- le double codage d'un élément d'une liste de cette table et de l'adresse de la suite de la liste et
- le double codage de l'état de la connexion et du numéro de sommet connecté, pour une extrémité d'une ligne du réseau.

Cependant, nous verrons dans le chapitre suivant que cette "explication", qui nous satisfait intuitivement, ne correspond pas à la définition formelle de l'*implémentation correcte* .

Chapitre 11

La correction de l'implémentation

Résumé

Nous démontrons dans ce chapitre qu'au lieu de nous aider à prouver l'*implémentation correcte* d'ABSTRACT1 par la spécification concrète, nos spécifications intermédiaires démontrent que l'implémentation n'est pas correcte.

Après une brève introduction, nous rappelons, dans la Section 11.1, la définition de l'implémentation correcte d'une spécification algébrique par une autre.

Dans les sous-sections successives de la Section 11.2, nous reprenons chacune des étapes d'implémentation du chapitre précédant en les comparant (informellement) à cette définition et en indiquant comment certaines ne la satisfont pas.

Nous décrivons alors, dans les sous-sections successives de la Section 11.3, des extensions conservatrices des spécifications intermédiaires qui sont des implémentations correctes d'ABSTRACT1.

Cependant, nous expliquons dans la Section 11.4 que nous n'avons pas envie de simplement adopter les spécifications étendues comme les "vraies" spécifications intermédiaires.

Introduction

Nous avons réussi à élaborer une suite de spécifications intermédiaires, de moins en moins abstraites, dont chacune caractérise l'un des choix d'implémentation qui expliquent comment `PRCHEN` satisfait la spécification abstraite. Nous trouvons cette explication persuasive. Mais nous démontrons dans ce chapitre qu'au lieu de nous aider à prouver l'*implémentation correcte* d'ABSTRACT1 par la spécification concrète, nos spécifications intermédiaires démontrent que l'implémentation n'est pas correcte.

Rappelons que le développement de logiciels à l'aide des méthodes formelles décrit dans la Section 1.1 du Chapitre 1 repose sur la preuve de la correction de chaque étape d'implémentation. Pour prouver le respect de la spécification la plus abstraite par la spécification concrète, il faut prouver que chaque nouvelle spécification construite respecte la précédente[1]. Normalement, nous démontrons que chaque spécification est une *implémentation correcte* de la précédente.

11.1 L'implémentation correcte

Nous donnons ici une définition "opérationnelle" de l'*implémentation correcte* . Les fondements théoriques permettant d'arriver à cette définition sont expliqués dans [Wir90], [ONS93] [ST88], [BH96] et [BCH99].

Une spécification algébrique SP_A est "implémentée" dans ce sens par **une autre spécification algébrique** SP_I [2]. Chacune des spécifications SP_A et SP_I spécifient de nombreux programmes Fortran[3].

1. En fait, la preuve de chaque étape d'implémentation ne suffit pas pour prouver la correction de la suite d'étapes dans certains cas d'implémentation comportementale (voir [ONS93]), mais cette exception ne nous concerne pas ici.

2. Certains auteurs décrivent comment nous obtenons SP_I par l'enrichissement d'une première spécification SP_C, afin d'assurer l'implémentation de SP_A. Cet aspect ne nous concerne pas pour l'instant : nous supposons l'enrichissement fait.

3. En effet, nous prenons comme *sémantique* d'une spécification algébrique de signature Σ, un ensemble de Σ-*structures*.

Pour que l'implémentation soit *correcte*, SP_I doit spécifier tout ou une partie de l'ensemble de programmes spécifié par SP_A (sauf dans le cas de l'*implémentation comportementale* où il s'agit de spécifier un ensemble de programmes qui **se comportent comme** tout ou une partie de l'ensemble de programmes spécifié par SP_A, comme nous expliquons avec plus de précision dans la suite).

En d'autres termes, SP_I ne doit pas permettre un programme qui ne satisfait pas SP_A mais peut exclure certains programmes satisfaisant SP_A. Dans ce dernier cas, il s'agit d'une inclusion stricte des deux ensembles de programmes que nous appellerons *raffinement* : SP_I impose plus de contraintes que SP_A.

Nous ne vérifions pas directement l'inclusion de la classe de programmes spécifiés par SP_I dans celle de SP_A, mais son corollaire : que les axiomes (ainsi que les *contraintes extra-logiques*, c'est-à-dire, dans notre cas, l'engendrement libre ou fini des sortes) de SP_A sont inclus dans SP_I ou dans les conséquences logiques des axiomes de SP_I.

11.1.1 Le cas sans "hiding"

Supposons dans un premier temps que le "hiding" n'est employé dans aucune des deux spécifications. Pour démontrer la correction de l'implémentation, il faut

- trouver une correspondance entre SP_A et une partie de SP_I et ensuite
- démontrer que les axiomes et contraintes extra-logiques de SP_I impliquent ceux de SP_A. Ceci suppose un système de preuve complet et correct et, de préférence, un outil d'aide à la démonstration de théorèmes. Nous n'allons pas développer cet aspect car nous allons voir que sur notre exemple nous rencontrons déjà des problèmes dans la recherche de la bonne correspondance entre les spécifications.

La correspondance entre SP_A et SP_I peut comporter plusieurs aspects que nous précisons et illustrons par la suite :

- la traduction de certains symboles de SP_I qui correspondent à ceux de SP_A, mais ont un autre nom ;
- l'"oubli" des symboles de SP_I qui ne signifient rien dans SP_A ;
- la "restriction" de SP_I afin d'exclure de certaines sortes la **partie** qui ne signifie rien dans SP_A,
- en cas d'implémentation comportementale, le remplacement, dans les axiomes de SP_A, de la notion d'égalité entre termes des sortes "non-observables" par l'*équivalence comportementale.*

La traduction et l'oubli

La traduction de SP_I, ainsi que l'"oubli" des symboles qui ne signifient rien dans SP_A, sont définis par un *morphisme* (que nous appellerons *morphisme d'implémentation*)

$\sigma : \Sigma_A \to \Sigma_I$

depuis la signature, Σ_A, de SP_A vers la signature, Σ_I, de SP_I. Nous devons pouvoir retrouver dans SP_I :

- une sorte correspondant à chaque sorte de SP_A,
- un prédicat du bon profil correspondant à chaque prédicat de SP_A et
- une opération du bon profil correspondant à chaque opération de SP_A.

Si σ envoie un symbole x_A de Σ_A sur un symbole x_I de Σ_I, alors nous dirons par abus de langage, que x_I *implémente* x_A.

En ce qui concerne la correspondance entre opérations, l'opération op_I implémentant une opération **partielle** op_A peut être

- totale ou

Une même Σ-structure peut être dénotée par de nombreuses spécifications de signature Σ. Une seule Σ-structure peut exprimer la sémantique de nombreux programmes Fortran équivalents et nous supposons d'habitude que la sémantique de chaque programme est exprimée par une seule Σ-structure (pour une signature Σ donnée). Cependant, nous verrons dans le chapitre suivant que la sémantique d'un même programme Fortran peut être exprimée par plusieurs Σ-structures.

– partielle, si le domaine d'op_I inclut (ou est égal à) celui d'op_A (après application de σ).

Si le domaine d'op_I est plus grand que celui d'op_A, alors op_I doit respecter juste sur le domaine d'op_A les autres axiomes de SP_A la définissant. Évidemment, une opération totale op_A ne peut être implémentée que par une opération totale op_I.

La restriction et l'équivalence comportementale

Regardons de plus près la correspondance entre les termes d'une sorte s_A de SP_A et de la sorte s_I de SP_I qui l'implémente. Supposons d'abord que SP_A spécifie un ensemble de *constructeurs* de termes de sorte s_A et, éventuellement, des égalités entre termes engendrés de différentes façons, et que SP_I spécifie un ensemble de constructeurs de termes de sorte s_I et, éventuellement, des égalités entre termes engendrés de différentes façons. Le morphisme σ fait correspondre chaque constructeur de s_A à un constructeur de s_I. Il y a trois cas possibles :

1. Le cas le plus facile est celui où

   – chaque constructeur de terme de sorte s_I dans SP_I implémente un constructeur de terme de sorte s_A dans SP_A et

   – chaque égalité entre termes de sorte s_A spécifiée (explicitement ou par conséquence logique des axiomes) dans SP_A est spécifiée aussi (sur les termes correspondants) dans SP_I.

   La correspondance entre s_I et s_A est immédiate.

2. Souvent, il y a dans SP_I des constructeurs

   – qui ne correspondent à aucun constructeur de s_A et

   – qui permettent d'engendrer des termes de sorte s_I qui ne sont pas spécifiés comme égaux à un autre terme engendré par un constructeur implémentant un constructeur de SP_A.

   La **partie** de s_I qui exclut ces termes est la partie engendrée par les constructeurs implémentant ceux de s_A. C'est cette partie de s_I qui implémente s_A et il faut restreindre s_I à cette partie pour les besoins de la preuve que les axiomes de SP_I impliquent ceux de SP_A.

   Un exemple est l'implémentation de la sorte (s_A) des ensembles de numéros positifs par la sorte (s_I) des listes sans doublons de positifs, dans laquelle

   – l'ensemble vide est implémenté par la liste vide et

   – l'opération d'ajout d'un élément à l'ensemble est implémenté par une opération qui insère un élément, s'il n'est pas encore présent, dans la liste de façon à respecter l'ordre habituel $<$ sur les positifs.

   Parmi toutes les listes sans doublons (engendrées par la liste vide et par la restriction de *cons*), c'est juste les listes ordonnées (engendrées par la liste vide et l'opération d'insertion) qui implémentent les ensembles.

   CASL nous offre la possibilité de spécifier par une sous-sorte de s_I (les listes ordonnées sans doublons de positifs, dans notre exemple) la partie implémentant s_A. Dans la correspondance entre SP_A et SP_I employée dans la justification de l'implémentation correcte, s_A correspondrait alors à cette sous-sorte de s_I, et la définition de la restriction de s_I ne serait pas nécessaire.

3. Parfois, des égalités entre termes de s_A spécifiées dans SP_A ne sont pas spécifiées dans SP_I entre les termes correspondants de s_I. Cependant, l'implémentation de s_A par s_I peut être justifiée comme un cas d'implémentation comportementale, si c'est le "comportement" des termes de s_A que nous avons réellement voulu spécifier dans SP_A. C'est-à-dire, en spécifiant l'égalité entre deux termes de s_A, nous avons voulu spécifier l'égalité entre les résultats (n'appartenant pas à s_A) de l'application de certains des observateurs de s_A aux termes. Nous disons que s_A n'est pas *observable*. Il s'agit

alors de remplacer en quelque sorte dans les axiomes de SP_A, la relation habituelle d'égalité entre les termes de s_A par l'équivalence comportementale.

Un exemple est l'implémentation dans ABSTRACT3 des ensembles de numéros positifs par des listes sans doublons de positifs, où deux listes contenant exactement les mêmes éléments (mais ordonnées de deux façons différentes) représentent le même ensemble. Ceci constitue une implémentation comportementale si le comportement de l'ensemble est donné par le résultat des observateurs *is_empty* et *is_in*.

Parfois, les constructeurs de s_A ne sont pas spécifiés par SP_A, mais en comparant les sortes s_A et s_I, nous voyons qu'un seul objet de s_A peut correspondre à plusieurs objets de s_I (mais chaque objet de s_A correspond à au moins un objet de s_I). Un exemple est donné par l'implémentation spécifiée par ABSTRACT2 de la sorte *Reseau* spécifiée par ABSTRACT1. Dans ABSTRACT2, les sommets et les lignes d'un réseau sont numérotés et donc chaque objet de la sorte *Reseau* d'ABSTRACT1 correspond à tous les objets de la sorte *Reseau* d'ABSTRACT2 regroupant effectivement les mêmes sommets et lignes, mais avec les diverses numérotations possibles. Cette étape d'implémentation serait un exemple du troisième cas ci-dessus si les constructeurs des sortes *Reseau* d'ABSTRACT1 et d'ABSTRACT2 étaient spécifiés. Mais en l'occurrence, ils ne le sont pas et les axiomes d'ABSTRACT1 n'impliquent aucune contrainte d'égalité entre deux objets de sorte *Reseau* obtenus de deux façons différentes, alors la définition de l'équivalence comportementale n'est pas nécessaire.

11.1.2 Le cas avec "hiding"

Prenons maintenant le cas où Σ_A regroupe les symboles non cachés d'une spécification SP'_A avec "hiding", la signature de tous les symboles cachés ou non étant Σ'_A.

Dans ce cas, il faut démontrer :

- l'existence d'une extension que nous appelons SP'_I (et dont nous appelons la signature Σ'_I) de SP_I, qui est une implémentation correcte de SP'_A sans "hiding", en employant le morphisme d'implémentation
  $\sigma : \Sigma'_A \to \Sigma'_I$
- que σ renvoie Σ_A sur (une partie de) Σ_I et
- que l'extension SP'_I de SP_I est *conservatrice*, i.e. qu'elle n'implique aucune formule sur les termes de Σ_I qui n'était pas déjà une conséquence logique de SP_I (ni restreint les modèles de Σ_I par des contraintes extra-logiques ni élargit la classe de modèles de Σ_I en employant le "hiding").

Notons qu'un cas spécial se présente quand la partie cachée étend de façon conservatrice la partie non cachée de la spécification abstraite, c'est-à-dire quand SP'_A sans le "hiding" est une extension conservatrice de SP_A. Dans ce cas, SP_A et SP'_A spécifient exactement les mêmes programmes : elles sont *équivalentes*.

Par ailleurs, si Σ_I regroupe les symboles "visibles" d'une spécification SP''_I de signature Σ''_I, alors les symboles cachés de SP''_I ne sont pas pris en compte dans la correspondance entre sa signature et Σ_A. Toutefois, la partie cachée de SP''_I (ou un fragment de celle-ci) peut coïncider avec l'extension conservatrice vers SP'_I nécessaire à la démonstration de correction.

11.2 Pourquoi notre implémentation n'est pas correcte

Pour que notre spécification concrète soit une implémentation correcte de la spécification la plus abstraite, il faudrait que chaque spécification intermédiaire soit une implémentation correcte de la précédente. Ce n'est pas le cas pour les étapes d'implémentation ABSTRACT1 - ABSTRACT2, ABSTRACT5 - ABSTRACT6 et ABSTRACT6 - ABSTRACT7, ni pour le passage d'ABSTRACT8 à la spécification concrète.

11.2.1 L'étape d'implémentation "incorrecte" Abstract1 - Abstract2

Nous avons expliqué dans la section précédente qu'une implémentation correcte repose sur un morphisme entre les signatures des deux spécifications. Dans cette étape d'implémentation, un calcul sur les sommets et lignes est remplacé par un calcul sur les numéros de sommet et de ligne et le morphisme

- $Sommet \mapsto NumS$
- $Ligne \mapsto NumL$
- $Reseau \mapsto Reseau$
- $lignes\_sortant \mapsto nums\_lignes\_sortant$

est respecté par le profil de $nums\_lignes\_sortant$. Toutefois, l'application de ce morphisme aux profils des prédicats et opérations unaires sur les sortes *Sommet* et *Ligne* d'ABSTRACT1 (tels que $origine\_de : Ligne \to ?\ Sommet$) ne donne des profils d'aucun prédicat ou opération d'ABSTRACT2. En effet, dans ABSTRACT2 les prédicats et opérations qui semblent correspondre (tels que $num\_orig\_de : NumL * Reseau \to ?\ NumS$) ont tous un argument supplémentaire de sorte *Reseau*. C'est parce que les objets des sortes *Sommet* (resp. *Ligne*) ont plusieurs attributs que les numéros de sommet (resp. ligne) d'ABSTRACT2 ne suffisent pas seuls à les représenter.

Par ailleurs, le profil des opérations *sommets_de* et *lignes_de* d'ABSTRACT1 ne correspondent, après application de ce morphisme, à aucun profil d'opération d'ABSTRACT2.

11.2.2 L'implémentation comportementale correcte d'Abstract2 par Abstract3

Nous avons dit déjà (dans la Section 11.1.1) que cette étape d'implémentation est un exemple d'implémentation comportementale. En effet, nous choisissons dans ABSTRACT3 de simuler les ensembles de numéros de ligne par les listes sans doublons de numéros de ligne, c'est-à-dire :

- la sorte $Set[NumL]$ par $LDE[NumL]$,
- la constante cachée *empty* de sorte $Set[NumL]$ par la projection sur $LDE[NumL]$ de la constante *empty* de la sorte cachée $List[NumL]$,
- le prédicat caché *is_empty* sur $Set[NumL]$ par la restriction à $LDE[NumL]$ du prédicat *is_empty* sur $List[NumL]$,
- le prédicat caché *is_in* sur $Set[NumL]$ par la restriction à $LDE[NumL]$ du prédicat *is_in* sur $List[NumL]$ et
- l'opération cachée $add : NumL * Set[NumL] \to Set[NumL]$ par une opération **totale** au bon profil et comportement. Il convient de définir dans une extension conservatrice d'ABSTRACT3 l'opération

  $consLDE : NumL * LDE[NumL] \to ?\ LDE[NumL]$

  comme la restriction à $LDE[NumL]$ de l'opération *cons* sur $List[NumL]$, avec projection du résultat sur $LDE[NumL]$, ainsi que l'opération

  $cons\_if\_not\_in : NumL * LDE[NumL] \to LDE[NumL]$
  qui

  - renvoie la valeur inchangée de la liste au cas où l'élément à ajouter y appartient déjà et
  - sinon est égale à $consLDE$.

  L'opération $cons\_if\_not\_in$ implémente *add*.

Le morphisme d'implémentation traduit cette correspondance entre les symboles, une fois ABSTRACT3 étendue de façon conservatrice par la définition de ces prédicats et opérations sur $LDE[NumL]$.

L'engendrement libre de $List[NumL]$ par *empty* et *cons* implique l'engendrement libre de sa sous-sorte $LDE[NumL]$ par la projection d'*empty* sur $LDE[NumL]$ et *consLDE*. Ceci implique l'engendrement fini de $LDE[NumL]$ par la projection d'*empty* sur $LDE[NumL]$ et *cons_if_not_in*, et la contrainte d'engendrement de $Set[NumL]$ est satisfaite.

Cependant, la spécification standard des ensembles enrichie par ABSTRACT2 comprend un axiome précisant que deux ensembles sont égaux si et seulement si ils ont les mêmes éléments. Cet axiome n'est pas satisfait pour les listes sans doublons. En effet, différentes listes sans doublons peuvent contenir les mêmes éléments, mais dans des ordres différents.

L'implémentation d'ABSTRACT2 par ABSTRACT3 est donc une implémentation comportementale. Nous nous permettons d'interpréter l'égalité entre éléments $s1$ et $s2$ de $Set[NumL]$ dans les axiomes d'ABSTRACT2 par l'équivalence comportementale. Nous disons alors que $s1$ et $s2$ sont équivalents si et seulement si

- *is_empty* donne le même résultat pour $s1$ et pour $s2$ et
- *is_in* donne toujours le même résultat pour $s1$ et pour $s2$, quel que soit le numéro de ligne.

Deux listes sans doublons sont équivalentes dans ce sens si et seulement si elles ont les mêmes éléments, et nous pouvons considérer l'axiome de ABSTRACT2 satisfait.

11.2.3 L'implémentation correcte d'Abstract3 par Abstract4

ABSTRACT4 est une extension conservatrice d'ABSTRACT3 et donc trivialement une implémentation correcte.

11.2.4 L'implémentation correcte d'Abstract4 par Abstract5

Il faut définir une extension conservatrice d'ABSTRACT5 afin de démontrer comment elle implémente les prédicats et opérations cachés d'ABSTRACT4 et ainsi, satisfait les axiomes d'ABSTRACT4 spécifiant l'ordonnancement des résultats des opérations *nums_lignes_sortant* et *nums_lignes_entrant*.

L'extension conservatrice définit

- $\_\_ <_e \_\_dans\_\_$ comme la composition de $\_\_.NRLE$ et de $\_\_ <_a \_\_dans\_\_$,
- $\_\_ <_o \_\_dans\_\_$ comme la composition de $\_\_.NRLO$ et de $\_\_ <_a \_\_dans\_\_$,
- *__est_ord_extrem_dans__* comme la composition de $\_\_.NRLE$ et de *__est_ord_dans__*,
- *__est_ord_orig_dans__* comme la composition de $\_\_.NRLO$ et de *__est_ord_dans__*,
- *insert_ord_extrem* comme la composition de $\_\_.NRLE$ et d'*insert* et
- *insert_ord_orig* comme la composition de $\_\_.NRLO$ et d'*insert*,

et le morphisme d'implémentation est l'identité.

11.2.5 L'étape d'implémentation "incorrecte" Abstract5 - Abstract6

Le morphisme de signatures pour cette étape d'implémentation est l'identité, mais les opérations *nums_lignes_sortant* et *nums_lignes_entrant* d'ABSTRACT5 ne correspondent à aucune opération d'ABSTRACT6. C'est parce que ces deux opérations sont implémentées par des compositions des opérations $\_\_.NS$, $\_\_.NL$, $\_\_.NRLO$, $\_\_.NRLE$, *construit_table_cocycles* et *retrieve* d'ABSTRACT6. ABSTRACT6 n'inclut pas la définition de ces compositions d'opérations, car tout l'intérêt de cette étape d'implémentation est de séparer le calcul des numéros de ligne sortant, de leur emploi.

11.2.6 L'étape d'implémentation "incorrecte" Abstract6 - Abstract7

Nous avons envie de poser comme morphisme de signatures pour cette étape d'implémentation, l'identité pour tous les symboles de la signature sauf $Table[NumS, LDE[NumL]]$, qui serait envoyé sur $TabPartList[NumS, NumL]$. Mais cette dernière est une sous-sorte de la première et ne peut contenir toutes les tables engendrées par l'opération totale *add* définie sur $Table[NumS, LDE[NumL]]$.

11.2.7 L'implémentation comportementale correcte d'Abstract7 par Abstract8

La sorte $List[NumL]$ (qui "instancie" $List[Elem]$) est cachée dans ABSTRACT7 et les listes avec doublons ne sont pas implémentées par ABSTRACT8. Afin de prouver l'implémentation correcte d'ABSTRACT7 par ABSTRACT8, il faut alors étendre d'abord cette dernière de façon conservatrice afin de démontrer l'implémentation de $List[NumL]$. Il s'agit de

- étendre ABSTRACT8 par LIST[ELEM],
- par la définition de la relation de sous-sorte entre $LDE[Elem]$ et $List[Elem]$,
- ainsi que par la définition de l'opération partielle $consLDE$ comme la restriction à $LDE[Elem]$ de $cons$ (mais avec un résultat dans $LDE[Elem]$)
- et finalement, ajouter la définition de $consLDE$ en terme de $retrieve$ et de $consTab$:

LIST[ELEM]
and
IMPLTABLEPARTITIONINGLISTS[**sort** Key;][ELEMORNIL]
then local
  pred $\_\mathit{is\_different\_elems} : List[Elem]$;
  vars $l : List[Elem]$;
  axiom $l\ \mathit{is\_different\_elems}$
    $\Leftrightarrow\ (l\ \mathit{is\_empty} \vee$
      $(\neg\ \mathit{head\_of}\ l\ \mathit{is\_in}\ \mathit{tail\_of}\ l \wedge\ \mathit{tail\_of}\ l\ \mathit{is\_different\_elems}))$;
within
sort $LDE[Elem] = \{\ l : List[Elem]\ \bullet\ l\ \mathit{is\_different\_elems};\ \}$
op $consLDE : Elem * LDE[Elem] \to ?\ LDE[Elem]$
vars $e : Elem;\ l : LDE[Elem];\ k : Key;\ t : TabPartList[Key, Elem]$;
axiom $consLDE(e, l) = cons(e, l)\ as\ LDE[NumL]$;
  $consLDE(e, retrieve(k, t)) = retrieve(k, consTab(e, k, t))$;

Par ailleurs, rappelons qu'en interprétant certains objets de sorte $Pair[Elem\_ou\_nil, Array[Elem, Elem\_or\_nil]]$ comme des listes sans doublons d'éléments de sorte $Elem$, comme expliqué dans la Section 10.8 et illustré dans la Figure 2.3, nous ignorons les valeurs dans le tableau indexées par des indices autres que les éléments de la liste.

En effet, deux tableaux différents peuvent représenter la même queue de liste, à condition qu'ils aient les mêmes valeurs juste aux positions indexées par les éléments de la liste. De la même façon, la liste vide est représentée par un premier composant de valeur nil et n'importe quel tableau. Deux paires dans lesquelles les tableaux sont différents, peuvent représenter alors deux listes sans doublons contenant les mêmes éléments aux mêmes positions. Il s'agit d'une implémentation *comportementale* de la liste sans doublons.

Comme nous l'avons remarqué dans la Section 10.3, la déclaration de $LDE[Elem]$ comme sous-sorte de $List[Elem]$ impose une obligation de preuve de l'engendrement libre des objets de la sorte $LDE[Elem]$ définie dans ABSTRACT8. Les axiomes définissant les observateurs de $LDE[Elem]$ sont "hérités" aussi de LIST[ELEM].

Alors, afin de prouver

- l'engendrement libre de $LDE[Elem]$ par $empty$ et $cons$ composés avec la projection sur $LDE[Elem]$,
- la satisfaction de la définition d'$\mathit{is\_empty}$ dans LIST[ELEM] par la composition de la projection sur $LDE[Elem]$ et de $\mathit{is\_empty}$ sur $LDE[Elem]$,
- la satisfaction de la définition de $\mathit{head\_of}$ dans LIST[ELEM] par la composition de la projection sur $LDE[Elem]$ et de $\mathit{head\_of}$ sur $LDE[Elem]$,
- la satisfaction de la définition de $\mathit{tail\_of}$ dans LIST[ELEM] par la composition de la projection sur $LDE[Elem]$, de $\mathit{tail\_of}$ sur $LDE[Elem]$ et de l'injection dans $List[Elem]$

nous

- ajoutons à ABSTRACT8 la définition d'une constante distinguée parmi toutes les valeurs représentant la liste vide :

  IMPLLISTDIFFELEMS[ELEMORNIL]
  then
  op $empty : LDE[Elem]$;
  axiom $empty =<< nil \;\; new >>$;

- définissons dans ABSTRACT7 l'équivalence comportementale ec entre deux listes sans doublons de la façon suivante :

  LISTDIFFELEMS[ELEM]
  then
  pred $ec : LDE[Elem] * LDE[Elem]$;
  vars $l1, l2 : LDE[Elem]$;
  axiom $ec(l1, l2)$
  $\Leftrightarrow$ (($l1$ $is\_empty \Leftrightarrow l2$ is_empty) $\wedge$
  ($l1$ is_empty $\vee$
  ($\neg$ ($l1$ is_empty) $\wedge$
  $head\_of\ l1 = head\_of\ l2$ $\wedge$
  $ec(tail\_of\ l1, tail\_of\ l2)$)));

- et remplaçons dans les axiomes et la contrainte d'engendrement libre de ABSTRACT7, l'égalité par l'équivalence comportementale.

11.2.8 L'étape d'implémentation "incorrecte" Abstract8 - Concrete

L'implémentation d'ABSTRACT8 par CONCRETE repose sur le morphisme qui est l'identité pour tous les symboles sauf $TabPartList[Nums, NumL]$, qui correspond à S_IPCO_ICCO. Cependant, ce morphisme ne permet pas de trouver dans (la partie non "locale"de) CONCRETE des symboles correspondant à :

1. la sorte $LDE[NumL]$: en fait, seule la sorte S_KP_ICCx correspondant à la sous-sorte de $LDE[NumL]$ regroupant les listes non vides, est définie ;

2. le prédicat (is_empty) et les opérations ($head\_of$ et $tail\_of$, new, $empty\_\_at\_\_$, $consTab$, $insert\_\_after\_\_in$ et $retrieve$) définis sur les sortes $LDE[NumL]$ et $TabPartList[Nums, NumL]$, ainsi que l'opération $insertTab$: en effet, les axiomes de la spécification concrète emploient divers prédicats et opérations auxiliaires définis en "local", correspondant pour la plupart, soit aux restrictions à S_KP_ICCx des opérations définies dans ABSTRACT8 sur $LDE[NumL]$ (telles que $list\_at\_in\_\_ : NumS * S\_IPCx\_ICCx \rightarrow? S\_KP\_ICCx$ qui est la restriction à S_KP_ICCx de $retrieve$), soit aux compositions des prédicats et opérations manquants (telles que $is\_empty\_at : S\_IPCx\_ICCx * NumS$ qui est la composition de $retrieve$ et d'is_empty). Ces prédicats et opérations sont tous implémentés par une ou deux instructions de Fortran. De la même façon, les opérations implémentant $consTab$ et $insert\_\_after\_\_in$ en trois instructions Fortran ne sont définies qu'en "local" ;

3. la sorte $Reseau$ et les opérations ($\_\_.NS$, $\_\_.IdSommets$, $\_\_.AttS1$, $\_\_.NL$, $\_\_.IdLignes$, $\_\_.AttL1$, $\_\_.NRLO$ et $\_\_.NRLE$) permettant d'extraire du réseau le nombre de sommets, nombre de lignes et tableaux stockant les valeurs des attributs pour tous les sommets (resp. lignes) d'un réseau, les prédicats et opérations ($est\_connectee\_orig$, $est\_connectee\_extrem$, id, $attribS1$, $attribL1$, num_orig et num_extrem) d'accès aux propriétés des sommets et lignes du réseau, ainsi que les sortes ($IdSommet$, $AttribS1$, $IdLigne$ et $AttribL1$) des attributs autres que l'origine et l'extrémité des sommets et des lignes, les sortes des tableaux stockant la valeur de ces derniers attributs et les opérations définies sur ces tableaux.

La non-implémentation de la sorte $LDE[NumL]$ et des prédicats et opérations sur cette sorte et sur $TabPartList[Nums, NumL]$ ressemblent à l'implémentation "incorrecte" d'ABSTRACT5 par ABSTRACT6, mais elle intègre deux aspects propres au passage de l'ultime spécification abstraite vers la spécification concrète :

1. Nous avons expliqué dans la Partie 1 que les tuples de valeurs ne sont pas explicités dans le code Fortran, alors la définition dans la spécification concrète d'une sorte telle que S_KP_ICCx regroupant certains tuples de valeurs est un choix de rétro-ingénieur. Nous aurions aussi bien pu choisir de caractériser par une sorte les tuples de valeurs implémentant la sorte $LDE[NumL]$. De la même façon nous avons choisi de spécifier certaines séquences d'instructions Fortran dans des prédicats et opérations auxiliaires, afin d'améliorer la lisibilité de la spécification concrète. Ces séquences d'instructions ne sont pas encapsulées sous forme de sous-programme Fortran, alors ce choix aussi est assez arbitraire.

2. Par ailleurs, nous savons que dans le style de programmation employé à l'époque du code analysé, il était rare qu'une opération effectuée en trois instructions Fortran soit encapsulée comme sous-programme, ce qui explique que les opérations $consTab$ et $insert\_\_after\_\_in$ spécifiées dans ABSTRACT8 ne sont définies qu'en "local" dans la spécification concrète.

La non-implémentation de la sorte *Reseau*, ainsi que les prédicats et opérations agissant dessus et autres sortes associées, a une autre explication. Il s'agit des sortes, prédicats et opérations qui ne sont pas définis de façon OP-complète dans ABSTRACT8, car la spécification concrète est OP-complète mais ABSTRACT8 ne l'est pas. Normalement (comme nous l'avons expliqué dans le Chapitre 1), le but d'un développement de code à l'aide des méthodes formelles est d'arriver à une spécification OP-complète, étendant (de façon conservatrice) la spécification du langage de programmation. Pour des besoins de clarté ABSTRACT8, ainsi que toutes les autres spécifications abstraites depuis ABSTRACT1, incluent la spécification de quelques sortes et opérations qui ne sont pas implémentées par le fragment de code étudié, **mais qui sont implémentées par d'autres fragments d'*ESTIME*** . Notons que cette explication ne s'applique pas aux sortes, prédicats et opérations d'ABSTRACT1, ABSTRACT5 et ABSTRACT6 qui ne sont pas implémentés par ABSTRACT2 ABSTRACT6 et ABSTRACT7 respectivement, car ils ne sont implémentés par aucun autre fragment d'*ESTIME*.

11.3 Une implémentation correcte d'Abstract1

Nous pouvons étendre les spécifications ABSTRACT2 à ABSTRACT8 afin qu'elles soient des implémentations correctes d'ABSTRACT1.

11.3.1 L'extension Abstract7′ d'Abstract7

Rappelons que ABSTRACT7 n'est par une implémentation correcte d'ABSTRACT6 car elle n'implémente pas les tables non partitionneuses.

Nous étendons donc ABSTRACT7 (comme illustré dans la Figure 11.1) par

- le module IMPLCORRECTE7TABLENUMSLIGNE qui étend
  - TABLE instancié de façon à définir la sorte $Table[NumS, LDE[NumL]]$ et
  - TABLEPARTITIONNEUSENUMSLIGNE,

  et explicite la relation de sous-sorte entre $TabPartList[NumS, NumL]$ et $Table[NumS, LDE[NumL]]$, définissant partiellement l'opérations *add* sur cette dernière en terme des opérations sur la première et

- le module IMPLCORRECTE7CONSTRUCTABLENUMSLIGNESSORTANTOUENTRANT qui définit une opération $construit\_table\_cocycles$ avec résultat de sorte $Table[NumS, LDE[NumL]]$ comme la composition de l'opération du même nom avec résultat de sorte $TabPartList[NumS, NumL]$ et de l'injection du résultat dans la sur-sorte.

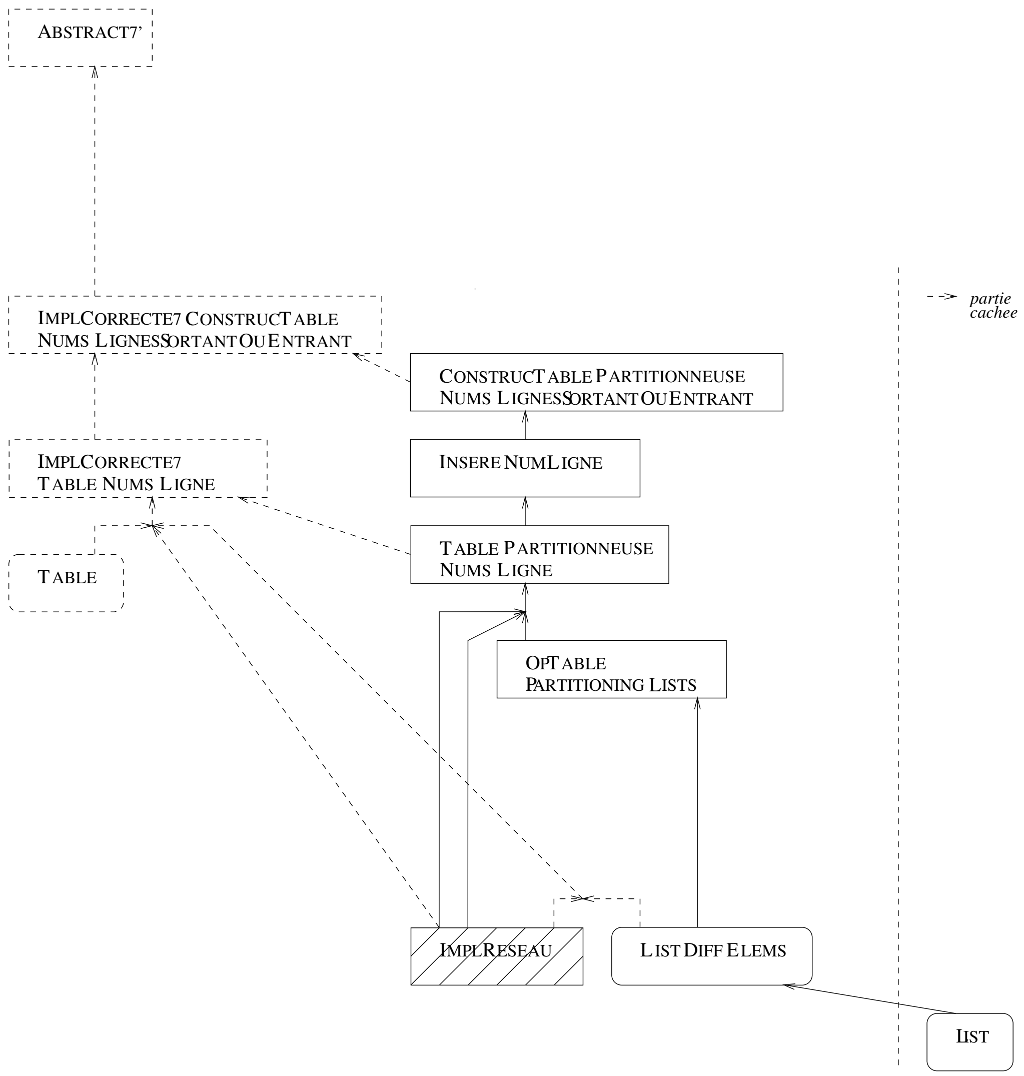


Fig. 11.1: Structure d'Abstract7 $'$

Cette extension d'ABSTRACT7 s'appelle ABSTRACT7′ et est présentée dans l'Annexe U.

ABSTRACT7′ est une implémentation correcte d'ABSTRACT6 dans laquelle

- le module TABLENUMSLIGNE d'ABSTRACT6 est implémenté par les modules TABLEPARTITIONNEUSENUMSLIGNE et IMPLCORRECTE7TABLENUMSLIGNE d'ABSTRACT7′
- CONSTRUCTABLENUMSLIGNESSORTANTOUENTRANT est implémenté par CONSTRUCTABLESPARTITIONNEUSENUMSLIGNESSORTANTOUENTRANT et IMPLCORRECTE7CONSTRUCTABLENUMSLIGNESSORTANTOUENTRANT.

ABSTRACT7′ est une extension conservatrice d'ABSTRACT7. Toutefois, elle ne permet pas de définir la construction des objets de la sorte $Table[NumS, LDE[NumL]]$ en termes de prédicats et d'opérations spécifiant des opérations primitives de Fortran. Une implémentation d'ABSTRACT7 ne peut être OP-complète que si elle ajoute (aux conséquences des axiomes) la définition en termes de prédicats et d'opérations spécifiant des opérations primitives de Fortran de chaque terme de $Table[NumS, LDE[NumL]]$.

ABSTRACT8 est étendue de la même façon à ABSTRACT8′, sauf que

- le module IMPLCORRECTE7TABLENUMSLIGNE est remplacé dans ABSTRACT8′ par un module (du même nom sauf que le chiffre 7 du nom est remplacé par 8) identique, sauf qu'il enrichit IMPLTABLENUMSLIGNE au lieu de TABLEPARTITIONNEUSENUMSLIGNE et
- le module IMPLCORRECTE7CONSTRUCTABLENUMSLIGNESSORTANTOUENTRANT est remplacé dans ABSTRACT8′ par un module (du même nom sauf que le 7 est remplacé par 8) identique sauf qu'il enrichit IMPLCONSTRUCTABLENUMSLIGNESSORTANTOUENTRANT et IMPLCORRECTE8 TABLENUMSLIGNE.

11.3.2 L'extension Abstract6′ d'Abstract6

Pour qu'ABSTRACT6 soit une implémentation correcte d'ABSTRACT5, il faut l'étendre à ABSTRACT6′ par la définition (dans le module IMPLCORRECTE6LISTESNUMSLIGNESSORTANTETENTRANT) des opérations *nums_lignes_sortant* et *nums_lignes_entrant* comme la composition de $\_\_.NS$, $\_\_.NL$, $\_\_.NRLO$, $\_\_.NRLE$, *construit_table_cocycles* et *retrieve*.

ABSTRACT6′ est présentée dans l'Annexe T et sa structure dans la Figure 11.2.

ABSTRACT6′ est une implémentation correcte d'ABSTRACT5 dans laquelle le module LISTES-NUMSLIGNESSORTANTETENTRANT est implémenté par IMPLCORRECTE6LISTESNUMSLIGNESSORTANT ETENTRANT.

ABSTRACT6′ est une extension conservatrice d'ABSTRACT6.

ABSTRACT7′ et ABSTRACT8′ sont étendues de la même façon (mis à part les modifications des extensions afin d'enrichir les bons modules, et la modification des numéros dans les noms des modules, comme expliqué dans le cas de ABSTRACT8′) à ABSTRACT7″ et ABSTRACT8″.

11.3.3 L'extension Abstract2′ d'Abstract2

Nous avons dit dans la Section 11.2.1 que les numéros de sommet et de ligne ne peuvent être une implémentation correcte des sortes *Sommet* et *Ligne*.

C'est pourquoi nous étendons ABSTRACT2 (comme illustré dans la Figure 11.3) par les modules SOMMETABSTRAIT et LIGNEABSTRAITE.

Le module IMPLCORRECTERESEAU spécifie **de façon déclarative non calculatoire**

- les relations pour chaque réseau entre sommets resp. lignes et numéros, à travers
  - les opérations
    $sommet : NumS * Reseau \rightarrow ?\ Sommet$ et
    $ligne : NumL * Reseau \rightarrow ?\ Ligne$
    qui construisent un objet de sorte *Sommet* (resp. *Ligne*) à partir d'un objet de sorte $NumS$ (resp. $NumL$) et d'un objet de sorte *Reseau* et sont définies pour les numéros de sommet (resp. ligne) inférieure ou égal au nombre total fourni pour le réseau par $\_\_.NS$ (resp. $\_\_.NL$),

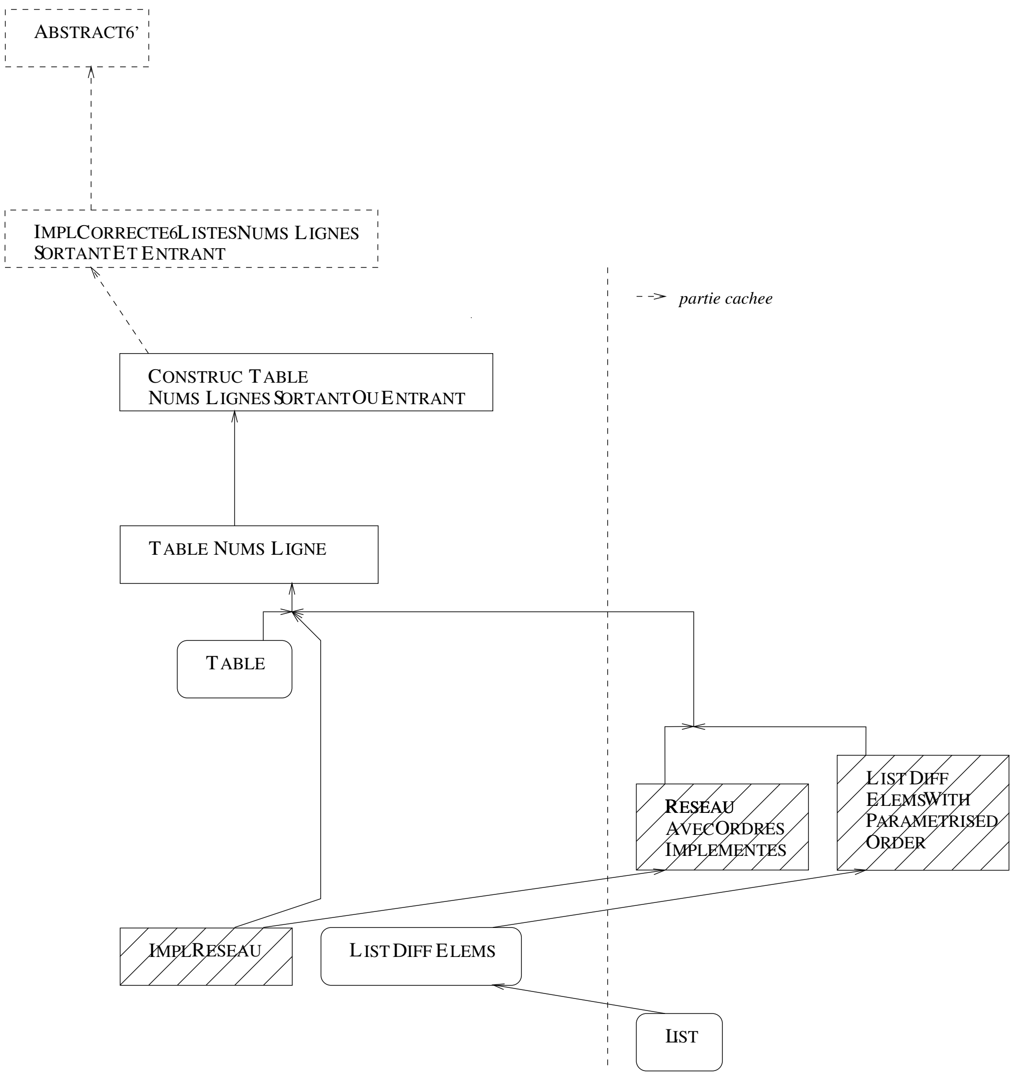


Fig. 11.2: Structure d'Abstract6 ′

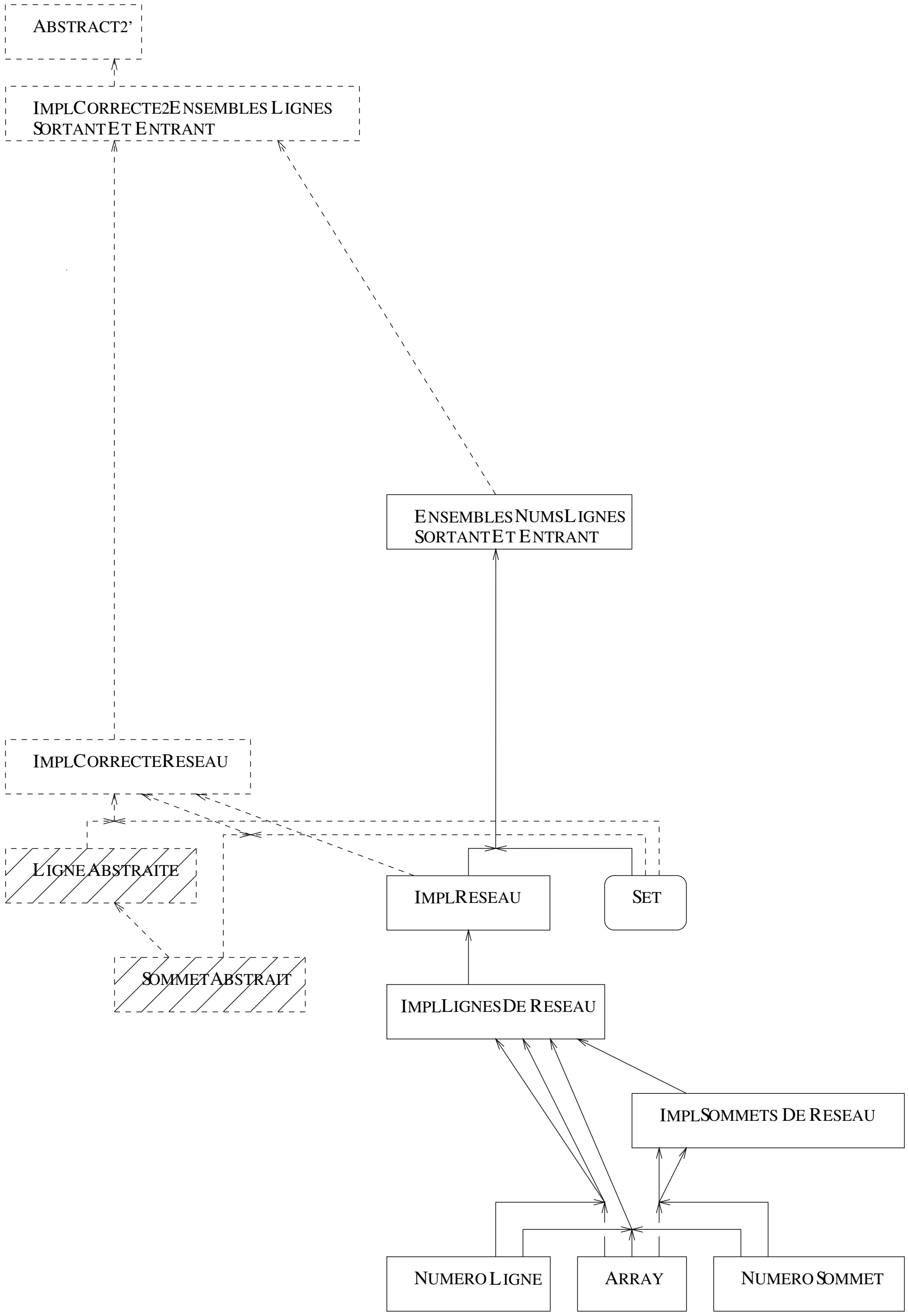


FIG. 11.3: Structure d'ABSTRACT2'

  - leurs inverses, les opérations
    $num : Sommet * Reseau \rightarrow ? NumS$ et
    $num : Ligne * Reseau \rightarrow ? NumL$,
- les opérations *sommets_de* et *lignes_de* à partir des opérations *num*.

Il spécifie aussi l'implémentation des prédicats et opérations unaires sur *Sommet* et *Ligne* **pour un réseau donné contenant le sommet ou la ligne en question** et il contient deux axiomes précisant que chaque sommet et chaque ligne appartient toujours à au moins un réseau. Grâce à ces axiomes, les implémentations des prédicats et opérations unaires sur *Sommet* resp. *Ligne* sont définies sur tous les objets de ces deux sortes.

ImplCorrecteReseau contient aussi deux axiomes précisant que les sortes *Sommet* et *Ligne* sont *partitionnées* par les valeurs des attributs (c'est-à-dire : deux sommets ayant les mêmes valeurs pour tous leur attributs sont égaux). Ces axiomes sont nécessaires à la preuve des axiomes d'Abstract1 contenant des égalités entre objets de sorte *Sommet* resp. *Ligne*.

Le module ImplCorrecte2EnsemblesLignesSortantEtEntrant spécifie **de façon déclarative non calculatoire aussi** le calcul de *lignes_sortant* resp. *lignes_entrant* à partir de *nums_lignes_sortant* resp. *nums_lignes_entrant* et des opérations *num*.

Appelons Abstract2′ cette extension d'Abstract2, présentée dans l'Annexe S.

Abstract2′ est une implémentation correcte d'Abstract1 dans laquelle

- le module ReseauAbstrait est implémenté par ImplReseau et ImplCorrecteReseau et
- le module EnsemblesLignesSortantEtEntrant est implémenté par EnsemblesNums-LignesSortantEtEntrant et ImplCorrecte2EnsemblesLignesSortantEtEntrant.

Notons cependant que cette implémentation correcte repose, comme nous l'avons expliqué dans la Section 11.1.1, sur une implémentation de la sorte *Reseau* d'Abstract1 par une sorte (la sorte *Reseau* d'Abstract2) contenant beaucoup plus d'objets, qui n'est justifiée que par le fait que l'égalité entre deux objets de sorte *Reseau* n'intervient jamais dans Abstract1.

Abstract2′ est une extension conservatrice d'Abstract2. Toutefois, elle ne permet pas de définir la construction des sortes *Sommet* et *Ligne* en termes de prédicats et d'opérations spécifiant les opérations primitives de Fortran. Par conséquent, une implémentation de Abstract2′ ne peut être OP-complète que si elle ajoute (aux conséquences des axiomes) la définition en termes de prédicats et d'opérations spécifiant des opérations primitives de Fortran de chaque terme des sortes *Sommet* et *Ligne*.

Les spécifications Abstract3 à Abstract5, Abstract6′, Abstract7″ et Abstract8″ sont étendues de la même façon (mis à part les modifications des extensions afin d'enrichir les bons modules et la modification des numéros dans les noms des modules), créant ainsi les spécifications Abstract3′ à Abstract5′, Abstract6″, Abstract7‴ et Abstract8‴.

11.4 Le problème posé par l'implémentation correcte d'Abstract1

La structure d'Abstract8‴ est illustrée dans la Figure 11.4. Elle est une implémentation correcte d'Abstract1 et une extension conservatrice d'Abstract8. Toutefois, elle n'est pas OP-complète et même si nous ajoutions la spécification OP-complète de la partie implémentée par d'autres fragments d'*ESTIME* (et délimitée dans la Section 11.2.8 ci-dessus), il manquerait toujours la spécification OP-complète de la construction des sortes *Sommet*, *Ligne* et $Table[NumS, LDE[NumL]]$.

Nous pourrions étendre la spécification concrète de la même façon que Abstract8, afin qu'elle aussi soit une implémentation correcte d'Abstract1. Mais d'une part ce serait de donner une spécification fausse du fragment de `PRXPLO` et de `PRCHEN` étudié et d'autre part, la spécification concrète ne serait plus OP-complète.

Les modules composant toutes les spécifications abstraites étendues de la façon décrite dans la section précédente, sont présentés de façon schématique dans la Figure 11.5[4]. Les modules hachurés dans cette figure sont ceux ajoutés dans les extensions décrites ci-dessus et *raffinés* par aucune spécification ultérieure, mais simplement “copiés” de spécification en spécification(sauf pour les noms des modules qu'ils

4. La Figure 11.5 ne montre pas les modules enrichis par ImplReseau.

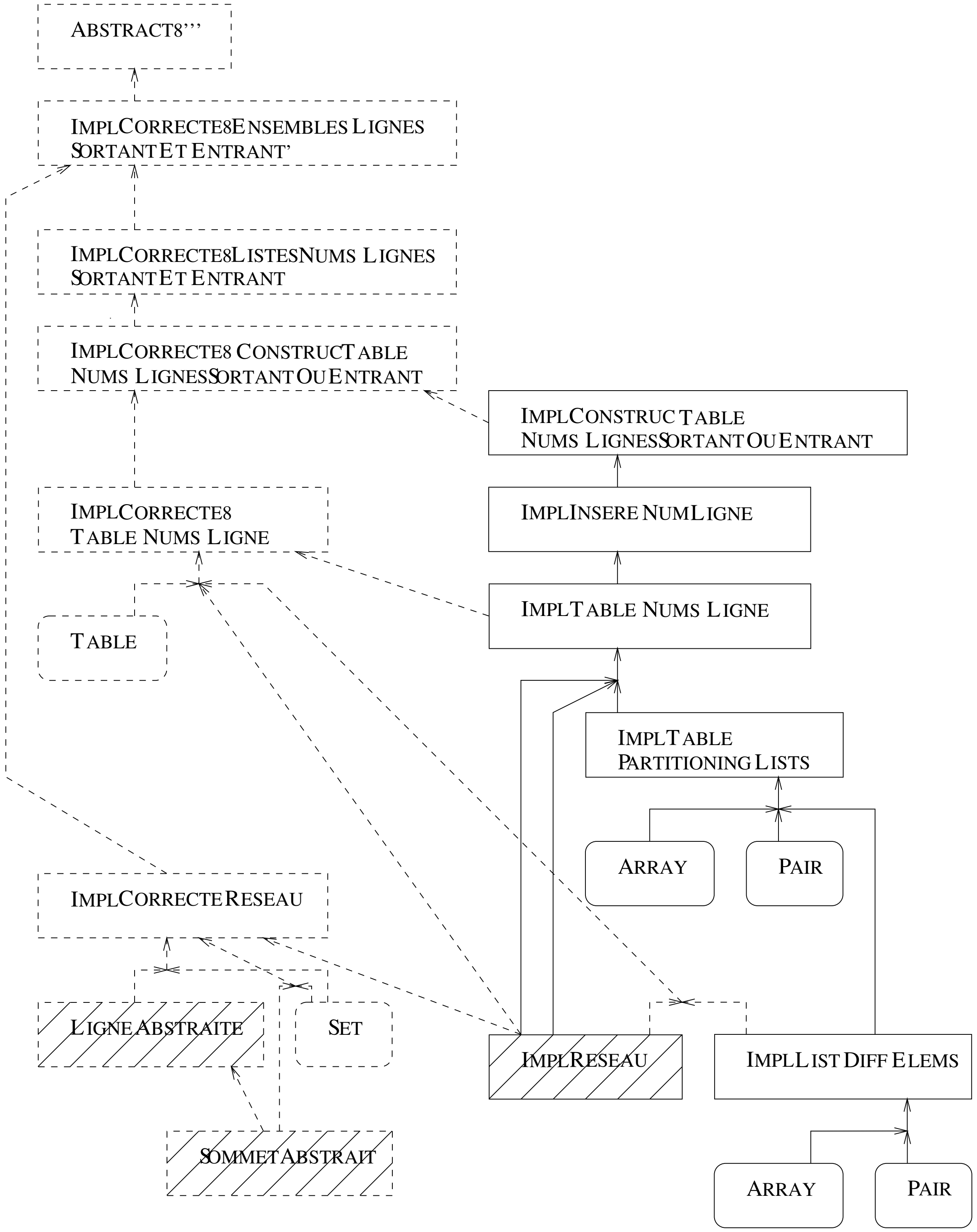


FIG. 11.4: Structure d'ABSTRACT8 '''

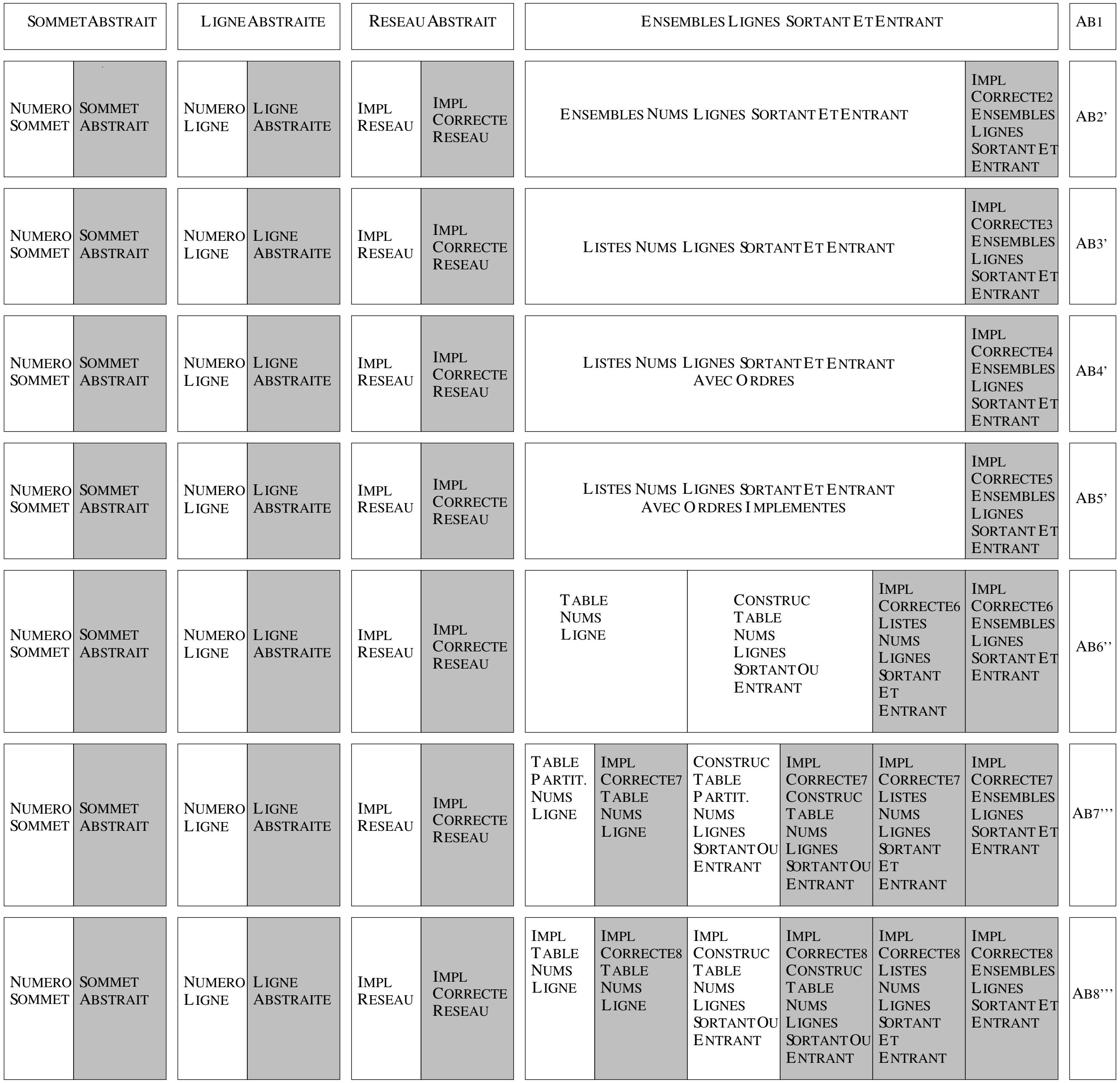


Fig. 11.5: L'implémentation correcte

enrichissent). Ils ne sont implémentés ni par le fragment de code spécifié par la spécification concrète, ni par aucun autre fragment d'*ESTIME*. Nous sommes convaincus qu'il n'y a "pas besoin" de les implémenter et pourtant, sans eux Abstract8 n'est pas une implémentation correcte d'Abstract1. C'est pourquoi nous allons proposer, dans le chapitre suivant, une relation entre deux spécifications algébriques qui est moins contraignante que l'implémentation correcte.

Chapitre 12

Notre définition d'une abstraction correcte

Résumé

Nous commençons ce chapitre par un rappel de

- pourquoi ABSTRACT8 n'est pas une *implémentation correcte* d'ABSTRACT1 et
- pourquoi nous voulons garder ABSTRACT1 comme **abstraction** d'ABSTRACT8.

(en laissant de coté l'abstraction de la spécification concrète par ABSTRACT8 qui ne sert qu'à illustrer des points spécifiques à la première étape d'abstraction d'une spécification concrète).

Dans la Section 12.1, nous allons rappeler que nous nous trouvons dans la situation particulière de l'étude d'un fragment d'un code entier optimisé, expliquer pourquoi cette situation ne correspond pas au paradigme de l'implémentation correcte des spécifications algébriques et caractériser le cas général de l'implémentation d'un fragment d'un code entier optimisé.

Dans les Sections 12.2, 12.3 et 12.4, nous allons définir trois cas particuliers, tirés de nos trois étapes d'implémentation "incorrectes", d'implémentation d'un fragment d'un code entier optimisé. Les définitions sont nécessairement longues et fastidieuses, mais la Section 12.4 contient aussi une caractérisation de l'implémentation d'ABSTRACT1 par ABSTRACT2 en terme de l'implémentation (correcte) d'un *produit dépendant* par un autre.

Finalement, nous allons nous inspirer de cette étude de l'implémentation d'un fragment d'un code entier optimisé pour terminer ce chapitre, dans la Section 12.5, par une définition de l'*abstraction correcte* qui répond à nos attentes.

Introduction

En supposant que la spécification concrète représente fidèlement le code Fortran, le problème exposé dans le chapitre précédent nous laisse face à deux possibilités :

- soit ABSTRACT1 n'est pas la bonne abstraction du code
- soit nous avons besoin d'autres relations entre spécifications algébriques, en plus de l'inverse de l'implémentation correcte, pour décrire notre idée d'une abstraction utile et pertinente d'un fragment de code Fortran.

Peut-être qu'il ne fallait pas abstraire :

- les numéros de sommet et de ligne par les sommets et les lignes,
- la construction et la lecture d'une table stockant le *graphe* d'une fonction par la fonction elle-même et

– une table partitionneuse par une table plus générale.

Les modules SommetAbstrait et LigneAbstraite d'Abstract1 spécifient des **types abstraits de données** . Ils spécifient la représentation et l'interrogation des attributs, de tous les sommets et lignes possibles, sans tenir compte des réseaux auxquels ils appartiennent (car les réseaux ne sont spécifiés que dans le module ReseauAbstrait, qui est une extension de SommetAbstrait et de LigneAbstraite).

Par contre, Abstract2 spécifie la réalisation de tous les sommets et lignes **de chaque réseau**. L'interrogation des attributs d'un sommet ou d'une ligne n'est jamais effectuée sans passer comme argument supplémentaire le réseau auquel il ou elle appartient. C'est tout ce qui est nécessaire au calcul du co-cycle, mais **ce n'est pas une implémentation des types abstraits de données** bien encapsulés et indépendants.

Or nous avons employé des types abstraits de données dans Abstract1 parce que la **séparation des notions** de sommet, de ligne et de réseau rendent la spécification plus claire et non pas parce que nous avons supposé que le code Fortran étudié est réellement composé de réalisations bien encapsulées et indépendantes de ces types abstraits de données.

De façon similaire, nous avons choisi d'employer dans Abstract6 une spécification des tables sans aucune contrainte sur les données stockées, simplement parce qu'elle spécifie un type abstrait de données classique bien connu et facilite ainsi la **compréhension** de la spécification, et non pas parce que nous croyons que les structures de données du fragment de code étudié suffisent à représenter toutes les tables du type abstrait de données classique. De la même façon que nous aurions pu employer une spécification "loose" des ensembles dans Abstract1, nous aurions pu employer dans Abstract6 une spécification des tables qui permettait à l'opération *add* d'être partielle, c'est-à-dire : une spécification "loose" de la classe de toutes les tables, mais aussi de certaines sous-classes de celle-ci. La spécification dans Abstract6 des tables les plus classiques, et leur restriction dans Abstract7 aux tables partitionneuses, traduisent une idée de raffinement qui ne correspond pas à l'implémentation correcte.

Quant à l'implémentation d'une fonction de deux arguments (*nums_lignes_sortant*) par

– le calcul et stockage de son graphe pour une valeur donnée de l'un des arguments (le réseau) et

– la lecture de ce graphe,

la notion d'implémentation correcte ne permet en aucun cas de "raffiner" l'une des opérations spécifiées en la remplaçant par plusieurs autres, dont elle est la composition. Cependant, nous avons envie d'effectuer l'abstraction qui est l'inverse de ce "raffinement" et de "faire abstraction" de la mémorisation des résultats dans une table.

L'inverse de l'implémentation correcte semble alors insuffisant pour caractériser la notion d'abstraction qu'il nous faut, mais l'intérêt des spécifications **formelles** est de pouvoir vérifier que deux spécifications sont liées par une relation **précise**.

Nous proposons alors de définir une notion d'abstraction correcte qui est moins contraignante que l'inverse de l'implémentation correcte et qui permet d'exprimer de quelle façon et sous quelles conditions une spécification algébrique peut être une abstraction d'une autre.

12.1 L'implémentation d'un fragment d'un code entier optimisé

12.1.1 Le problème

Les symboles visibles d'une spécification algébrique spécifient l'interface du code spécifié avec l'extérieur. Ainsi, les prédicats et opérations décrivent les services fournis au monde extérieur et donc encapsulés sous forme de sous-programme Fortran. C'est-à-dire que l'existence de certains sous-programmes Fortran et, dans une certaine mesure, leurs profils, sont fixés par la spécification la plus abstraite. En effet, l'implémentation peut ajouter des prédicats ou opérations ou simuler une sorte par une autre, mais la définition de l'implémentation correcte rappelée dans le chapitre précédent exige le respect, moyennant le morphisme d'implémentation, des profils visibles de la spécification abstraite.

La spécification de cette interface du code avec l'extérieur peut nécessiter la définition d'autres sortes, prédicats et opérations, soit parce que c'est la seule façon d'exprimer certaines propriétés, soit pour augmenter la clarté et la compréhensibilité de la spécification. Si nous ne voulons pas imposer l'implémentation de ces sortes, prédicats et opérations issus de la *décomposition interne* de la spécification, alors il faut les *cacher* à l'aide du mécanisme du *"hiding"*. Ce mécanisme est généralement présenté dans la littérature comme la façon de spécifier les données et fonctionnalités "privées" d'un module. Une spécification moins abstraite, SP_I, **avec une décomposition interne différente** peut être une implémentation correcte d'une spécification avec "hiding", SP_A, si SP_I peut être étendue de façon conservatrice à une implémentation correcte, SP_I', de SP_A sans "hiding" (voir le chapitre précédent, Section 11.1.2).

Dans le cas d'une fonction ou type abstrait de données de "bibliothèque" (ou d'un "composant" dans un processus de construction de logiciels à partir de composants), nous ne connaissons pas d'avance les autres codes "clients" du code spécifié. C'est pourquoi son interface doit être figée dès la spécification la plus abstraite et les profils de l'interface doivent employer autant que possible des sortes d'autres types abstraits de données "standard".

Notre cas est différent : nous spécifions non pas des fonctions de bibliothèque, mais des fragments d'un code entier plus grand. Par conséquent, les seuls "clients" d'un fragment de code que nous analysons se trouvent à l'intérieur du même code. Dans notre cadre de rétro-ingénierie, les autres types abstraits de données employés et la décomposition en prédicats et opérations de la spécification la plus abstraite sont choisis afin de la rendre la plus **claire** et la plus **abstraite** possible et non pas afin de fixer les interfaces entre les fragments du code entier. Car en développant les spécifications intermédiaires décrites dans le Chapitre 10, nous avons trouvé que les profils des services fournis à l'extérieur, ainsi que la décomposition interne, décidés pour la spécification abstraite du fragment de code étudié ne sont pas toujours bien adaptés aux spécifications à des niveaux inférieurs d'abstraction. À deux niveaux différents d'abstraction, nous voulons décrire le rôle du fragment de code, soit en termes des sommets, lignes et de la fonction *lignes_sortant*, soit en termes des numéros de sommet, numéros de ligne et de la construction et de la lecture d'une table stockant le graphe de la fonction *nums_lignes_sortant*.

En effet, la suite de spécifications intermédiaires proposée dans le Chapitre 10 décrit l'implémentation de la spécification la plus abstraite en adaptant les types abstraits de données spécifiés **au contexte d'appel**, afin d'"optimiser" l'implémentation, c'est-à-dire : économiser de l'espace mémoire ou du temps d'exécution. Cette suite de spécifications intermédiaires décrit le raffinement, non seulement de certains types abstraits de données de la spécification abstraite (par exemple : l'implémentation des ensembles par des listes sans doublons), mais aussi de sa décomposition interne (par exemple : l'opération cachée *add* sur les ensembles est remplacée par l'opération cachée *cons* sur les listes mais *cons* n'est pas une implémentation correcte d'*add*) et même des profils définissant l'interface du fragment de code spécifié avec le reste du code (par exemple : le remplacement de l'opération *lignes_sortant* par *nums_lignes_sortant* d'abord et ensuite, par les opérations de construction et de lecture des tables).

12.1.2 Le problème du point de vue de la spécification du code entier

La seule façon de justifier la *correction* des étapes d'implémentation d'un fragment d'un code entier "optimisé" de cette façon, est d'élaborer la spécification du code entier et de cacher (à l'aide du mécanisme de "hiding") tous les symboles sauf ceux de l'interface du code entier avec l'extérieur. Ceci revient à traiter chaque code entier réalisé par EDF comme un seul composant. Presque tous les raffinements de la spécification abstraite du code entier concerneraient alors des symboles cachés et seraient justifiés par des extensions conservatrices.

De ce point de vue, notre exemple est la spécification d'un fragment du code *ESTIME* qui (supposons le) prend en entrée un objet de sorte *Reseau* et produit en sortie un autre objet de sorte *Reseau*.

La spécification la plus abstraite du code entier *ESTIME* contiendrait une opération *estime* : *Reseau* $\rightarrow$ *Reseau* visible et divers symboles cachés dont *Sommet*, *Ligne* et *lignes_sortant*. Les axiomes de cette spécification définiraient le résultat d'*estime* en termes des prédicats et opérations cachés, dont *lignes_sortant* :

spec ESTIME1 =
  ABSTRACT1
  ...
then
ops *estime* : *Reseau* $\rightarrow$ *Reseau*;
  ...

vars $r : Reseau$;
axioms $estime(r) =$
$\ldots$
$\forall\ s : Sommet \bullet ($
$s\ is\_in\ sommets\_de\ r$
$\Rightarrow \ldots lignes\_sortant(s, r) \ldots)$
$\ldots;$
hide $Sommet$,
$Ligne$,
$Set[Sommet]$,
$Set[Ligne]$,
$\ldots$
end

ESTIME1 serait correctement implémentée par une autre spécification définissant *estime* en termes de l'opération $nums\_lignes\_sortant$:

spec ESTIME2 =
ABSTRACT2
$\ldots$
then
ops $estime : Reseau \rightarrow Reseau$;
$\ldots$
vars $r : Reseau$;
axioms $estime(r) =$
$\ldots$
$\forall\ ns : NumS \bullet ($
$ns \leq r.NS$
$\Rightarrow \ldots nums\_lignes\_sortant(ns, r) \ldots)$
$\ldots;$
hide $nums\_lignes\_sortant$,
$\ldots$
end

ESTIME2 serait correctement implémentée par une autre spécification définissant *estime* en termes des opérations *retrieve* et $construit\_table\_cocycles$ sur la sorte $Table[NumS, LDE[NumL]]$:

spec ESTIME6 =
ABSTRACT6
$\ldots$
then
ops $estime : Reseau \rightarrow Reseau$;
$\ldots$
vars $r : Reseau$;
axioms $estime(r) =$
$\ldots$
$\forall\ ns : NumS \bullet ($
$ns \leq r.NS$
$\Rightarrow \ldots retrieve(ns, construit\_table\_cocycles(r.NS, r.NL, r.NRLO, r.NRLE)) \ldots)$
$\ldots;$
hide $Table[NumS, LDE[NumL]]$,
$\ldots$
end

ESTIME6 serait correctement implémentée par une autre spécification (ESTIME7) définissant *estime* en termes des restrictions des opérations *retrieve* et $construit\_table\_cocycles$ à la sous-sorte $TabPartList[NumS, NumL]$ de $Table[NumS, LDE[NumL]]$ (et dont nous ne montrons pas l'extrait, car il est identique à celui d'ESTIME6, sauf qu'ESTIME7 étend ABSTRACT7 et non pas ABSTRACT6).

ESTIME7 serait correctement implémentée par une spécification concrète dans laquelle *estime* serait définie de façon *OP-complète* en terme des symboles de la spécification algébrique de Fortran, éventuellement à travers d'autres prédicats et opérations, cachés ou non.

De ce point de vue, nous savons que l'implémentation incorrecte de l'opération *lignes_sortant* par *nums_lignes_sortant* que nous avons proposé dans le Chapitre 10 (Section 10.2) suffit pour l'implémentation correcte d'*ESTIME* si nous arrivons à définir *estime* dans ESTIME2 en terme de *nums_lignes_sortant* et sans employer *lignes_sortant*. C'est-à-dire, nous savons que l'implémentation correcte d'*ESTIME* ne nécessite pas, dans ce cas, l'implémentation de *lignes_sortant* par un appel de *nums_lignes_sortant*, comme spécifié dans l'extension de ABSTRACT2 proposée dans la Section 11.3.3 du chapitre précédent.

Malheureusement, nous n'avons pas une vision aussi complète de la spécification d'*ESTIME*.

12.1.3 Notre point de vue

En effet, nous ne pouvons pas adapter l'approche que nous venons de décrire, car la rétro-ingénierie du code entier en une fois semble complètement irréaliste. Mais une spécification abstraite de juste un fragment du code entier, dans laquelle **tous** les symboles sont cachés, n'a aucun sens. Par ailleurs, la définition de l'implémentation correcte d'une spécification avec "hiding" est très permissive et nous présente un problème difficile de preuve dans le cas général, à savoir : comment trouver la bonne extension conservatrice. C'est pourquoi proposer l'utilisation systématique du "hiding" nous semble une mauvaise solution pour notre problème.

Nous nous plaçons alors dans le cadre de l'implémentation de la spécification d'un fragment du code entier, dans laquelle tous les symboles ne peuvent pas être cachés, et avec une connaissance limitée de la spécification du reste du code. Dans ce cadre, notre problème est de savoir dans quels cas (ou sous quelles conditions) nous pouvons nous permettre de *raffiner la signature de la spécification du fragment du code*. C'est-à-dire, permettre une spécification de l'implémentation qui n'inclut qu'une partie d'une implémentation correcte, la partie exclue étant une extension conservatrice du reste.

12.1.4 Caractérisation du problème de notre point de vue

Nous nous permettons de justifier les étapes d'implémentation "incorrectes" d'un fragment d'un code plus grand par une extension conservatrice, comme si les symboles concernés étaient cachés. Cette situation est illustrée par la Figure 12.1.

Dans cette situation, nous voulons caractériser et trouver une justification pour la partie de SP'_I qui n'est pas implémentée.

Dans ce but, nous décomposons chaque étape "incorrecte" d'implémentation en deux types de sous-étape :

- celles dans lesquelles un raffinement classique (et implémentation correcte) de la spécification est effectuée et
- celles dans lesquelles il n'y a pas de raffinement, alors le nombre de modèles de la spécification ne change pas mais la signature est modifiée : appelons *"signature shift"* les transformations de spécification effectuées dans ces sous-étapes.

Dans les étapes de "signature shift" nous avons la situation illustrée par la Figure 12.2. La signature (Σ_I) de SP_I ne contient pas certains symboles de Σ_A mais, éventuellement, contient certains symboles

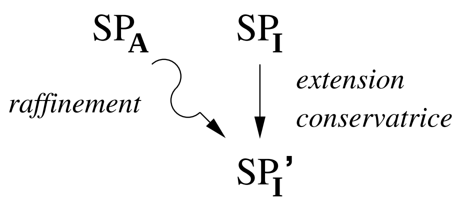


FIG. 12.1: Implémentation justifiée par une extension conservatrice

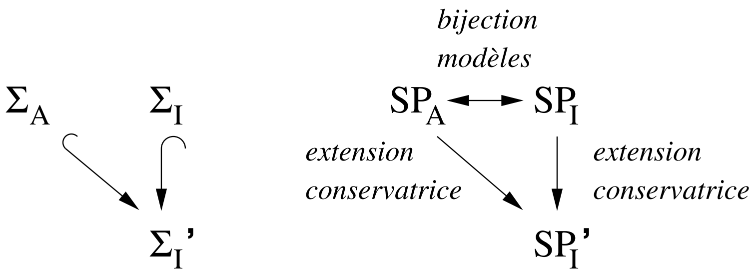


FIG. 12.2: "Signature shift" justifiée par une extension conservatrice

qui ne sont pas présents dans Σ_A. La signature de SP_I' contient au moins tous les symboles de Σ_A et $SP_I'|_{\Sigma_A}$ (SP_I' avec "hiding" de tous les symboles n'appartenant pas à la signature de SP_A) est *équivalente* à SP_A. Mais ceci implique que SP_I' est non seulement une extension conservatrice de SP_I, mais aussi de SP_A. Il y a donc un "mapping" bijectif entre les modèles de SP_A et ceux de SP_I.

Apparemment, toute SP_I dont SP_I' est une extension conservatrice, serait justifiée dans une étape de *"signature shift"*. Mais comme nous effectuons cette transformation sur la spécification d'un fragment d'un code plus grand, nous devons justifier la réduction de la signature de SP_I' vers Σ_I par rapport aux spécifications des autres fragments du code. Il faut vérifier que l'extension de SP_I vers SP_I' est conservatrice pour les spécifications de tous les autres fragments du code.

Nous allons expliquer comment les trois cas d'étape d'implémentation "incorrecte" exposés par notre exemple sont des instances de ce schéma général et caractériser pour chaque cas la transformation de "signature shift" de la spécification SP_A en une spécification SP_I ayant le même nombre de modèles mais une signature différente et justifiée de la façon que nous venons de décrire. Ceci dans l'espoir que ces trois cas sont typiques de ceux que nous rencontrerions dans d'autres codes d'EDF, et que la caractérisation de la transformation de SP_A en SP_I représente une façon plus intuitive et plus facile de justifier ce genre d'étape d'implémentation que la construction générale présentée ci-dessus. Commençons par le cas le plus facile.

12.2 L'implémentation par décomposition

Nous avons expliqué (dans la Section 10.6 du Chapitre 10 et la Section 11.2.5 du Chapitre 11) que l'intérêt de l'implémentation incorrecte dans ABSTRACT6 de l'opération *nums_lignes_sortant* d'ABSTRACT5 par six autres opérations, dont la composition est une implémentation correcte de *nums_lignes_sortant*, est de **séparer**, dans l'implémentation de *nums_lignes_sortant*, le pré-calcul du graphe de cette fonction, de la relecture des résultats de ce calcul.

Décomposons cette étape d'implémentation en deux sous-étapes :

- une première étape de raffinement, dans laquelle *nums_lignes_sortant* et *nums_lignes_entrant* sont définies comme des compositions d'autres opérations : cette étape est une implémentation correcte
- et l'étape de *"signature shift"* dans laquelle les symboles *nums_lignes_sortant* et *nums_lignes_entr-ant* sont éliminés de la signature. Dans ce cas, aucun nouveau symbole est ajouté à SP_I dans cette étape, alors les deux spécifications SP_A et SP_I' de notre schéma général coïncident.

Appelons la *décomposition d'une opération* la relation entre deux spécifications SP_A et SP_I si

1. une opération, *op*, de spécification SP_A n'est pas inclue dans SP_I
2. et l'extension conservatrice (SP_I') de SP_I qui consiste à lui ajouter la définition d'*op* comme une **composition** de plusieurs autres prédicats ou opérations *op*1, *op*2, ..., est *équivalente* à SP_A.

Cette définition s'étend naturellement au cas de plusieurs opérations, ainsi qu'à celui des prédicats.

Appelons l'*implémentation par décomposition* la combinaison des deux sous-étapes d'implémentation.

L'implémentation de SP_A par une spécification SP_I qui en est une *implémentation par décomposition* semble justifiée dans tous les cas où SP_A spécifie un fragment d'un code entier, à condition que l'opération p sur laquelle porte l'implémentation par décomposition ne fasse pas partie de l'interface du code entier avec l'extérieur. Appelons $TOUT_A$ la spécification du code entier, qui comprend SP_A, mais dans laquelle tous les symboles de SP_A sont cachés et appelons $TOUT_I$ une implémentation correcte de $TOUT_A$ comprenant SP_I, mais dans laquelle tous les symboles de SP_I sont cachés. Tout axiome de $TOUT_A$ exprimé en terme d'op peut être reformulé dans $TOUT_I$ en terme de la composition d'$op1$, $op2$, ...

12.3 Le raffinement des profils

Nous avons expliqué (dans la Section 10.7 du Chapitre 10) que l'intérêt du remplacement dans ABSTRACT7 de la sorte $Table[NumS, LDE[NumL]]$ d'ABSTRACT6 par sa sous-sorte $TabPart\text{-}List[NumS, NumL]$, est que $TabPartList[NumS, NumL]$ admet une implémentation astucieuse qui n'est pas valable pour la sorte $Table[NumS, LDE[NumL]]$ toute entière.

Nous pouvons décomposer cette étape d'implémentation en deux sous-étapes de *"signature shift"*.

- Dans la première, nous remplaçons $Table[NumS, LDE[NumL]]$ par $TabPartList[NumS, NumL]$ comme sorte du résultat de l'opération *construit_table_cocycles*, ainsi que la sorte de la constante cachée *new*, car les résultats de ces opérations appartiennent toujours à cette sous-sorte. Nous définissons aussi la restriction à $TabPartList[NumS, NumL]$ de l'observateur *retrieve* défini sur $Table[NumS, LDE[NumL]]$.
- Dans la deuxième sous-étape, nous décidons de ne pas implémenter la sorte $Table[NumS, LDE[NumL]]$, ni les opérations non cachées *construit_table_cocycles* et *retrieve* définies sur cette sorte. Nous gardons donc dans la spécification de l'implémentation la sorte $TabPartList[NumS, NumL]$ et les restrictions de *construit_table_cocycles* et *retrieve* à cette sorte. Cette sous-étape est justifiée par notre conviction que la sorte $Table[NumS, LDE[NumL]]$ n'est plus "nécessaire" dans la spécification ABSTRACT7. Nous supposons alors, en quelque sorte, que l'observateur *retrieve* ne sera pas appliqué, dans les spécifications des autres fragments du code, à d'autres objets de sorte $Table[NumS, LDE[NumL]]$, mais uniquement à ceux appartenant à $TabPartList[NumS, NumL]$, et notamment ceux créés par ce fragment de code.

Nous allons définir deux relations entre spécifications, la première correspondant à la première sous-étape ci-dessus et la deuxième aux deux sous-étapes mises ensemble.

Mais d'abord nous avons besoin de la définition de la *restriction* à la sous-sorte d'un prédicat ou d'une opération. Il y a deux définitions possibles. Prenons l'exemple de la restriction aux naturels de l'opération de soustraction sur les entiers :

$moins : Int * Int \to Int.$

Elle peut être définie soit comme l'opération totale qui donne un entier comme résultat

$moins : Nat * Nat \to Int,$

soit comme l'opération partielle avec un naturel comme résultat

$moins : Nat * Nat \to ?\ Nat.$

Comme nous voulons caractériser la modification de la sorte du résultat dans les profils des opérations, nous adoptons ci-dessous cette dernière définition.

Étant données une sorte s et une sous-sorte s' de s, appelons la *restriction à la sous-sorte s' d'une opération op* dont le profil contient s, l'opération op', telle que

1. le profil d'op' est le résultat du remplacement de **toutes** les occurrences de s dans le profil d'op par s',
2. si dans le profil d'op, le résultat est d'une sorte autre que s, alors le domaine d'op' est obtenu en excluant du domaine d'op, tous les tuples de valeurs des arguments dans lesquels la valeur d'un argument dont la sorte est s dans le profil d'op, n'appartient pas à s',

3. si dans le profil d'op, le résultat est de sorte s, alors le domaine d'op' regroupe tous les tuples de valeurs des arguments d'op qui donnent un résultat dans s', et qui n'incluent pas une valeur d'un argument dont la sorte est s dans le profil d'op, qui n'appartient pas à s',
4. si le résultat d'op est de sorte s, alors pour tous les arguments dans le domaine d'op', le résultat d'op' est égal à la projection sur s' du résultat d'op appliquée aux mêmes arguments (avec injection implicite des arguments de sorte s' dans s) et
5. si le résultat d'op n'est pas de sorte s, alors pour tous les arguments dans le domaine d'op', le résultat d'op' est égal au résultat d'op appliquée aux mêmes arguments (avec injection implicite des arguments de sorte s' dans s)

Appelons la *restriction à la sous-sorte s' d'un prédicat* pr dont le profil contient s, le prédicat pr' tel que

1. le profil de pr' est le résultat du remplacement de toutes les occurrences de s dans le profil de pr par s' et
2. la vérité de pr' est le résultat de l'application de pr aux arguments de pr' (avec injection implicite des arguments de sorte s' dans s).

Appelons *raffinement des profils* la relation entre deux spécifications SP_A et SP_I si

1. SP_I contient la définition d'une sorte s' dont nous pouvons démontrer qu'elle est une sous-sorte d'une sorte s de SP_A,
2. dans les profils d'une ou plusieurs opérations $op1$, $op2$, ... , de SP_A, la sorte de leurs résultats est déclarée comme s, mais les résultats de ces opérations appartiennent toujours à s' (c'est-à-dire : la projection sur s' est définie) et ces opérations sont remplacées dans SP_I par des opérations $op1'$, $op2'$, ... , dont les profils sont le résultat du remplacement dans les profils d'$op1$, $op2$, ... , de toutes les occurrences de s par s',
3. SP_I définit la restriction à s' de tous les observateurs définis dans SP_A sur s
4. et la spécification $(SP_I'|_{\Sigma_A})$ obtenue en ajoutant à SP_I
   (a) si nécessaire, la définition de la relation de sous-sorte entre s' et s et
   (b) la définition des opérations $op1$, $op2$, ... , telle qu'$op1'$, $op2'$, ... , sont la *restriction* à s' d'$op1$, $op2$, ...

   et en cachant s' (si elle n'est pas définie dans SP_A) est *équivalente* à SP_A.

L'implémentation de SP_A par une spécification SP_I qui en est un *raffinement des profils* , semble justifiée dans tous les cas où SP_A spécifie un fragment d'un code entier, à condition que les opérations $op1$, $op2$, ... , sur lesquelles porte le raffinement des profils, ne fassent pas partie de l'interface du code entier avec l'extérieur. Tout axiome de $TOUT_A$ exprimé en terme d'une ou de plusieurs opérations parmi $op1$, $op2$, ... , peut être reformulé dans $TOUT_I$ en terme des restrictions de ces opérations, avec, au besoin, l'emploi des restrictions des observateurs.

Appelons *raffinement des profils avec élimination de la sur-sorte* la relation entre deux spécifications SP_A et SP_I si

1. la sorte s de SP_A est remplacée par la sorte s' dans tous les profils de SP_I
2. et la spécification $(SP_I'|_{\Sigma_A})$ obtenue en ajoutant à SP_I
   (a) la définition de s et de la relation de sous-sorte entre s' et s et
   (b) tous les prédicats et opérations de SP_A dont le profil contient s, définis de telle façon que les prédicats et opérations qui les remplacent dans SP_I sont leurs *restrictions* à s'

   est *équivalente* à SP_A.

L'implémentation de SP_A par une spécification SP_I qui en est un *raffinement des profils avec élimination de la sur-sorte* , semble justifiée dans tous les cas où SP_A spécifie un fragment d'un code entier, à condition que

1. tous les prédicats et opérations restreintes à la sous-sorte s' par le raffinement des profils n'appartiennent pas à l'interface du code entier avec l'extérieur,

2. s n'est dans aucun profil de l'interface du code entier avec l'extérieur, ou sinon, qu'elle est définie par la spécification d'un autre fragment du code et

3. aucun observateur non caché restreint à la sous-sorte par le raffinement des profils n'est appliqué par un autre fragment de code à un argument n'appartenant pas à s', sauf si l'observateur est redéfini sur la bonne sorte par la spécification d'un autre fragment du code.

12.4 Caractérisation de l'implémentation d'Abstract1 par Abstract2

L'objectif introduit dans la Section 12.1 ci-dessus dans le cas de l'étape d'implémentation ABSTRACT1 - ABSTRACT2 est le suivant : nous voulons savoir sous quelles conditions nous pouvons implémenter la spécification du fragment d'un code entier exprimée en termes des sortes *Sommet* et *Ligne*, par une spécification du même fragment exprimée en termes des sortes $NumS$ et $NumL$, sachant que $NumS$ et $NumL$ ne sont pas des implémentations correctes de *Sommet* et *Ligne*.

Notons que si le réseau était constant et contenait tous les sommets de la sorte *Sommet* et toutes les lignes de la sorte *Ligne*, alors nous pourrions spécifier une bijection (constante) entre *Sommet* et $NumS$ et une autre entre *Ligne* et $NumL$. Dans ce cas, une spécification de $NumS$, $NumL$, de tableaux **constants** et de prédicats et opérations sur $NumS$ et $NumL$ serait une implémentation correcte de la spécification par LIGNEABSTRAITE des sortes *Sommet* et *Ligne*. Par ailleurs, $NumS$ et $NumL$ seraient des intervalles sur *Pos* bornés en haut par les constantes $r.NS$ et $r.NL$, respectivement (pour le réseau constant r). Nous pourrions implémenter alors les prédicats et opérations unaires sur *Sommet* et *Ligne* par des prédicats et opérations unaires sur $NumS$ et $NumL$ effectuant la lecture de tableaux constants. Plus loin dans le raffinement des spécifications, nous pourrions définir deux relations d'ordre (classiques, ne prenant pas le réseau comme argument supplémentaire) sur $NumL$ et les listes stockées dans les tables construites par `PRXPLO` seraient alors des listes ordonnées (classiques).

Remarquons que cette vision des choses correspond assez bien à notre compréhension intuitive immédiate du fragment de code étudié. En effet, nous avons "fait abstraction", dans une certaine mesure, dans notre compréhension intuitive du code, du fait que le réseau était variable.

En fait, le réseau est forcément variable, car c'est le paramètre d'entrée du code entier. La valeur du réseau paramétrise toutes les relations intuitives que nous venons d'évoquer. Ainsi, la "bijection" entre les sommets et les numéros de sommet **varie avec la valeur du réseau**. C'est pourquoi l'implémentation correcte de la spécification de la sorte *Sommet* par la spécification de la sorte $NumS$ est impossible et nous sommes obligés de caractériser la numérotation des sommets non pas comme la *simulation* de la sorte *Sommet* par la sorte $NumS$, mais comme l'ajout de la sorte $NumS$ et l'implémentation de chaque opération sur *Sommet* par la composition de

- la recherche de l'objet correspondant de sorte $NumS$ (pour le réseau donné),
- l'appel d'une opération sur $NumS$ et
- au besoin, la construction du sommet ou ligne correspondant à chaque numéro de sommet resp. ligne (pour le réseau donné) dans le résultat.

Mais cette caractérisation ne reflète pas le fait qu'au lieu d'effectuer ce genre d'aller-retour entre *Sommet* et $NumS$ pour chaque opération spécifiée sur *Sommet*, l'implémentation peut simplement basculer complètement à une représentation des sommets sous forme de numéros de sommet pour le réseau donné.

En effet, nous pouvons imaginer une implémentation de la sorte *Sommet* sous forme du tuple des valeurs de tous les attributs des sommets. Chaque objet de la sorte *Sommet* serait représenté par le tuple des valeurs lues dans les tableaux pertinents du réseau, à l'indice égal au numéro de sommet. De la même façon, chaque objet de la sorte *Ligne* serait représenté par un tuple des valeurs, lues pour la plupart dans les tableaux pertinents du réseau, à l'indice égal au numéro de ligne, mais comprenant aussi deux composants du tuple qui seraient eux mêmes les tuples représentant les sommets connectés à l'origine et à l'extrémité (du moins dans le cas des lignes connectées aux deux extrémités). L'implémentation de la fonction *lignes_sortant* nécessiterait la construction de l'ensemble de toutes les lignes, à partir de l'ensemble des numéros fourni par *nums_lignes_sortant*, en lisant tous les valeurs nécessaires dans les tableaux et en composant d'abord les tuples et ensuite l'ensemble de tous les tuples. C'est clair que cette procédure de construction du résultat de *lignes_sortant* n'est effectivement implémentée nulle part dans le code Fortran d'*ESTIME* et c'est pourquoi nous croyons que la spécifier dans ABSTRACT2 serait donner une représentation fausse du code.

En conclusion, notre problème pour le cas de l'implémentation d'ABSTRACT1 par ABSTRACT2, est comment caractériser l'implémentation des sortes *Sommet* et *Ligne* par les sortes *NumS* et *NumL* **dans le contexte d'un réseau.** Évidemment, cette implémentation repose sur les relations entre les sortes *Sommet* et *Reseau*, les sortes *Ligne* et *Reseau*, et les arguments et résultats de ces sortes des prédicats et opérations d'ABSTRACT1. Dans le cas de notre exemple, ces relations correspondent à une propriété du fragment de code étudié que nous pouvons exprimer de façon intuitive comme

> tout sommet ou ligne manipulé par le code appartient à un réseau manipulé par le code.

L'implémentation se justifie si le reste du code partage cette propriété.

Nous allons caractériser cette propriété et le raffinement d'ABSTRACT1 par ABSTRACT2 plus précisément en s'appuyant sur une idée qui traduit notre compréhension intuitive du rôle du réseau dans l'implémentation. Cette idée est la suivante :

> ce ne sont pas les sortes *Sommet*, *Ligne* et *Reseau* qui sont implémentées par ABSTRACT2, mais les sortes d'un produit dépendant de *Sommet* et de *Reseau* et d'un produit dépendant de *Ligne* et de *Reseau*.

Cependant, les spécifications ABSTRACT1 et ABSTRACT2 ne contiennent pas de produits dépendants, alors il faut les reformuler en termes de produits dépendants afin de démontrer que ABSTRACT2 reformulée ainsi implémente ABSTRACT1 reformulée.

En fait, dans une première sous-étape de raffinement (et implémentation correcte), nous ajoutons à ABSTRACT1 deux axiomes (présents aussi dans ABSTRACT2 ′, voir Section 11.3.3 du Chapitre 11) précisant que tout sommet et toute ligne appartient à au moins un réseau :

$\forall\ s : Sommet.\exists\ r : Reseau.s\ is\_in\ sommets\_de\ r$;

$\forall\ l : Ligne.\exists\ r : Reseau.l\ is\_in\ lignes\_de\ r$;

créant ainsi la spécification POST-ABSTRACT1. Nous verrons ci-dessous que ceci permet de justifier, dans l'étape suivante, le remplacement des prédicats et opérations totales unaires sur *Sommet* et *Ligne* par des prédicats (resp. opérations) sur la sorte des produits dépendent des sommets (resp. lignes) et du réseau d'appartenance ;

Et ABSTRACT2 résulte non seulement de l'implémentation des produits dépendants dans le sens que nous allons décrire ci-dessous, mais aussi d'une ultime sous-étape d'implémentation dans laquelle les opérations *sommets_de* et *lignes_de* d'ABSTRACT1 sont effectivement implémentées par $\_\_.NS$ et $\_\_.NL$. Nous allons traiter cette sous-étape d'implémentation par la suite, pour l'instant nous ajoutons à ABSTRACT2 les définitions suivantes :

$sommets\_de : Reseau \rightarrow ?\ Set[NumS]$

$lignes\_de : Reseau \rightarrow ?\ Set[NumL]$

$\forall ns : NumS.ns\ is\_in\ sommets\_de(r) \iff ns \leq r.NS;$

$\forall nl : NumL.nl\ is\_in\ lignes\_de(r) \iff nl \leq r.NL;$

créant ainsi la spécification PRE-ABSTRACT2.

Décrivons maintenant comment PRE-ABSTRACT2 reformulée en termes des produits dépendants est une implémentation correcte de POST-ABSTRACT1 reformulée en termes des produits dépendants.

12.4.1 La reformulation en termes de produits dépendants de Pre-Abstract2

Nous transformons les profils dans PRE-ABSTRACT2 afin d'y éliminer les sortes *NumS*, *NumL*, *Set*[*NumS*] et *Set*[*NumL*], en employant les sortes des produits dépendants :

- d'un numéro de sommet et d'un réseau contenant le sommet (appelons la sorte de ce produit dépendant *NumSommetDeReseau*),
- d'un numéro de ligne et d'un réseau contenant la ligne (produit de sorte *NumLigneDeReseau*),

- d'un ensemble de numéros de sommets du même réseau et du réseau auquel ils appartiennent (produit de sorte $EnsembleNumsSommetsDeMemeReseau$) et

- d'un ensemble de numéros de lignes du même réseau et du réseau auquel elles appartiennent (produit de sorte $EnsembleNumsLignesDeMemeReseau$).

Appelons *opération à profil dépendant* toute opération de PRE-ABSTRACT2 qui

- prend en entrée un objet de sorte $Reseau$,
- prend en entrée ou rend en sortie un ou plusieurs objets de sortes parmi $NumS$, $NumL$, $Set[NumS]$ ou $Set[NumL]$,
- (dans le cas d'arguments de sorte $NumS$ ou $NumL$) restreint, par son domaine, les valeurs de sorte $NumS$ ou $NumL$, en entrée à celles inférieures ou égales au nombre de sommets resp. lignes du réseau en entrée,
- (dans le cas d'arguments de sorte $Set[NumS]$ ou $Set[NumL]$) restreint, par son domaine, les valeurs de sorte $Set[NumS]$ ou $Set[NumL]$ en entrée à celles ne contenant que les numéros de sommet resp. ligne inférieurs ou égaux au nombre de sommets resp. lignes du réseau en entrée,
- (dans le cas où le résultat est de sorte $NumS$ ou $NumL$) a un résultat inférieur ou égal au nombre de sommets resp. lignes du réseau en entrée et
- (dans le cas où le résultat est de sorte $Set[NumS]$ ou $Set[NumL]$) a un résultat ne contenant que des numéros de sommet resp. ligne inférieur ou égal au nombre de sommets resp. lignes du réseau en entrée.

Appelons *prédicat à profil dépendant* tout prédicat de PRE-ABSTRACT2 qui

- prend en entrée un objet de sorte $Reseau$, ainsi qu'un ou plusieurs objets de sorte $NumS$, $NumL$, $Set[NumS]$ ou $Set[NumL]$,
- n'est vrai que pour les numéros de sommet ou de ligne (ou ensembles ne contenant que des numéros de sommet ou de ligne) inférieurs au nombre de sommets resp. lignes du réseau et
- n'est jamais appliqué, dans les axiomes de la spécification à un numéro de sommet (resp. ligne) ou à un ensemble contenant un numéro de sommet (resp. ligne), inférieur au nombre de sommets (resp. lignes) de l'argument de sorte $Reseau$.

Nous transformons les profils de tous les prédicats et toutes les opérations à profils dépendants de PRE-ABSTRACT2, en employant les sortes des produits dépendants évoqués plus haut pour exprimer à travers les profils toutes les contraintes énumérées dans notre définition de prédicat resp. d'opération à profil dépendant, sauf les deux dernières contraintes sur les opérations à profil dépendant. En effet, nous pouvons contraindre par des produits dépendants du **profil**, tout sommet ou ligne du résultat à appartenir à un réseau, mais il faut un **axiome** pour le contraindre à appartenir à celui en entrée.

Or une telle transformation de la spécification peut être justifiée par le fait que la spécification transformée spécifie exactement les mêmes codes Fortran que la spécification initiale.

En effet, nous avons expliqué dans le Chapitre 3 que les paramètres d'entrée et de sortie, les tuples de paramètres et les domaines des sous-programmes ne sont pas explicités en Fortran.

Prenons le cas d'un sous-programme avec n paramètres d'entrée implémentant une opération (et non pas un prédicat). S'il ne peut pas accepter toutes les combinaisons de valeurs des n paramètres d'entrée, alors ce sous-programme peut être spécifié :

- soit par une opération partielle avec n arguments

- soit par une opération totale unaire sur le *produit dépendant* des n paramètres.

Les sémantiques mathématiques des deux opérations sont différentes à cause de leurs profils différents mais nous pourrions les considérer comme "équivalentes" parce qu'elles sont des spécifications alternatives du même sous-programme Fortran.

Par ailleurs,

- une opération qui calcule un résultat r en fonction de la valeur de ses paramètres d'entrée et

- une opération identique sauf qu'elle construit comme résultat le produit dépendant de r et de la valeur de l'un des paramètres d'entrée

spécifient le même sous-programme Fortran. Nous pourrions aussi considérer ces deux opérations comme “équivalentes”.

De façon similaire, un sous-programme de deux arguments implémentant un prédicat et contenant une lecture de tableau qui n'est pas définie pour certaines combinaisons des valeurs en entrée peut être spécifié

- soit par un prédicat avec deux arguments qui n'est pas vrai pour les combinaisons de valeurs d'arguments en question
- soit par un prédicat unaire sur le *produit dépendant* des deux paramètres.

Appelons *équivalence en Fortran* la relation de *“signature shift”* entre deux spécifications algébriques $SP1$ et $SP2$ dont l'une est le résultat du remplacement dans l'autre de certains prédicats ou opérations par d'autres prédicats resp. opérations qui leur sont “équivalentes” dans l'une des trois façons décrites ci-dessus, c'est-à-dire, plus précisément :

- Une opération partielle $op1$ de $SP1$ dont deux de ses paramètres d'entrée $p1$ et $p2$ sont des sortes $S1$ et S, est remplacée dans $SP2$ par une opération $op2$ (partielle ou totale) du même profil, sauf que $p1$ et $p2$ sont remplacés par un seul paramètre $p12$ de la sorte d'un produit dépendant, $S1S$, de $S1$ et de S, regroupant les mêmes valeurs de ses deux composants ($p121$ et $p122$) que celles permises par le domaine d'$op1$.

  Le domaine d'$op1$ regroupe les mêmes tuples des valeurs de ses paramètres que les tuples des valeurs de

  - $p121$,
  - $p122$ et
  - des paramètres d'$op2$ autres que $p12$

  regroupés par le domaine d'$op2$.

  Les deux spécifications ayant la *même sémantique sur le premier composant du tuple*, dans le sens suivant. En ajoutant à $SP2$ la définition d'$op1$ comme le résultat d'$op2$ appliquée aux mêmes arguments, sauf que la valeur du premier composant ($p121$) du paramètre $p12$ d'$op2$ est la valeur du paramètre $p1$ d'$op1$, et la valeur du deuxième composant ($p122$) du paramètre $p12$ d'$op2$ est la valeur du paramètre p d'$op1$:

  op $op1 : \ldots * S1 * S * \ldots$

  $\forall\ p12 : S1S.op1(\ldots, 1st\_of\ p12, 2nd\_of\ p12, \ldots) = op2(\ldots, p12, \ldots)$

  et en *cachant* la sorte $S1S$, nous obtenons une spécification équivalente à $SP1$.

- Ou une opération $op1$ de $SP1$ avec un résultat de sorte $S2$ et, parmi ses paramètres d'entrée, le paramètre p de sorte S, est remplacée dans $SP2$ par une opération $op2$ du même profil, sauf que son résultat a la sorte d'un produit dépendant, $S2S$, de $S2$ et de S.

  Une conséquence des axiomes de $SP2$ étant que le deuxième composant du résultat d'$op2$ est égal à p :

  $2nd\_of\ op2(\ldots, p, \ldots) = p$

  et les deux spécifications ayant la *même sémantique sur le premier composant du tuple*, dans le sens suivant. En ajoutant à $SP2$ la définition d'$op1$ comme le premier composant du résultat d'$op2$ sur les mêmes arguments :

  op $op1 : \ldots * S * \ldots \to S2$

  (En fait, $op1$ peut être totale ou partielle.)

  $\forall\ p : S.op1(\ldots, p, \ldots) = 1st\_of\ op2(\ldots, p, \ldots)$

  et en *cachant* la sorte $S2S$, nous obtenons une spécification équivalente à $SP1$.

- Ou la combinaison des deux cas précédents : une opération partielle $op1$ de $SP1$ dont deux de ses paramètres d'entrée $p1$ et $p2$ sont des sortes $S1$ et S, et avec un résultat de sorte $S2$, est remplacée dans $SP2$ par une opération $op2$ (partielle ou totale) du même profil, sauf que $p1$ et $p2$ sont remplacés par un seul paramètre $p12$ de la sorte d'un produit dépendant, $S1S$, de $S1$ et de S, regroupant les mêmes valeurs de ses deux composants ($p121$ et $p122$) que celles permises par le domaine d'$op1$ et son résultat a la sorte d'un produit dépendant, $S2S$, de $S2$ et de S.

Le domaine d'$op1$ regroupe les mêmes tuples des valeurs de ses paramètres que les tuples des valeurs de

- $p121$,
- $p122$ et
- des paramètres d'$op2$ autres que $p12$

regroupés par le domaine d'$op2$.

Le résultat d'$op2$ étant toujours égal à la paire formée par le résultat d'$op1$ (pour les mêmes arguments, sauf que la valeur du premier composant ($p121$) du paramètre $p12$ d'$op2$ est la valeur du paramètre $p1$ d'$op1$, et la valeur du deuxième composant ($p122$) du paramètre $p12$ d'$op2$ est la valeur du paramètre p d'$op1$) et la valeur de p.

Les deux spécifications ayant la *même sémantique sur le premier composant du tuple*, dans le sens suivant. En ajoutant à $SP2$ la définition d'$op1$:

op $op1 : \ldots * S1 * S * \ldots \rightarrow ?\ S2$

$\forall\ p12 : S1S.op1(\ldots, 1st\_of\ p12, 2nd\_of\ p12, \ldots) = 1st\_of\ op2(\ldots, p12, \ldots)$

et en *cachant* les sortes $S1S$ et $S2S$, nous obtenons une spécification *équivalente* à $SP1$.

- Ou un prédicat $pr1$ de $SP1$ dont deux de ses paramètres d'entrée $p1$ et $p2$ sont des sortes $S1$ et S, est remplacée dans $SP2$ par un prédicat $pr2$ du même profil, sauf que $p1$ et $p2$ sont remplacés dans $pr2$ par un seul paramètre $p12$ de la sorte d'un produit dépendant, $S1S$, de $S1$ et de S, regroupant au moins toutes les valeurs de ses deux composants ($p121$ et $p122$) pour lesquelles $pr1$ est vrai.

  Les deux spécifications ayant la *même sémantique sur le premier composant du tuple*, dans le sens suivant. En ajoutant à $SP2$ la définition de $pr1$ comme le résultat de $pr1$ appliquée aux mêmes arguments, sauf que la valeur du premier composant ($p121$) du paramètre $p12$ de $pr2$ est la valeur du paramètre $p1$ de $pr1$, et la valeur du deuxième composant ($p122$) du paramètre $p12$ de $pr2$ est la valeur du paramètre p de $pr1$:

  pred $pr1 : \ldots * S1 * S * \ldots$

  $\forall\ p12 : S1S.pr1(\ldots, 1st\_of\ p12, 2nd\_of\ p12, \ldots) = pr2(\ldots, p12, \ldots)$

  (où $1st\_of$ et $2nd\_of$ sont les projections sur les tuples définies dans la spécificationgénérique "standard" Pair présentée dans l'Annexe I), et en *cachant* la sorte $S1S$, nous obtenons une spécification équivalente à $SP1$.

Cette définition s'étendant naturellement à plusieurs prédicats ou opérations.

La spécification résultant de cette transformation de Pre-Abstract2 est Pre-Abstract2PD, présentée dans l'Annexe W. Elle constitue la reformulation de Pre-Abstract2 en termes de produits dépendants.

12.4.2 La reformulation de Post-Abstract1 en termes de produits dépendants

Nous transformons les profils dans Post-Abstract1 afin d'y éliminer les sortes *Sommet*, *Ligne*, *Set*[*Sommet*] et *Set*[*Ligne*], en employant les sortes des produits dépendants :

- d'un sommet et d'un réseau auquel il appartient (appelons la sorte de ce produit dépendant *SommetDeReseau*),
- d'une ligne et d'un réseau auquel elle appartient (produit de sorte *LigneDeReseau*),
- d'un ensemble de sommets du même réseau et du réseau auquel ils appartiennent (produit de sorte *EnsembleSommetsDeMemeReseau*) et
- d'un ensemble de lignes du même réseau et du réseau auquel elles appartiennent (produit de sorte *EnsembleLignesDeMemeReseau*).

Il y a trois cas à traiter :

1. celui des profils contenant au moins un résultat ou argument de sortes parmi *Sommet*, *Ligne*, *Set*[*Sommet*] et *Set*[*Ligne*], ainsi qu'un argument de la sorte *Reseau*, dans notre exemple il s'agit des opérations suivantes :

$sommets\_de : Reseau \to Set[Sommet]$

$lignes\_de : Reseau \to Set[Ligne]$

$lignes\_sortant : Sommet * Reseau \to ?\ Set[Ligne]$ et

$lignes\_entrant : Sommet * Reseau \to ?\ Set[Ligne]$;

2. celui des profils contenant plusieurs occurrences de sortes parmi *Sommet*, *Ligne*, $Set[Sommet]$ ou $Set[Ligne]$, d'arguments ou de résultats contraints à appartenir au même réseau, mais ne contenant pas la sorte *Reseau*, dans notre exemple il s'agit des opérations

   $origine\_de : Ligne \to ?\ Sommet$ et

   $extremite\_de : Ligne \to ?\ Sommet$;

3. celui des profils contenant un seul argument appartenant à l'une des sortes *Sommet*, *Ligne*, $Set[Sommet]$ ou $Set[Ligne]$, mais ne contenant pas un argument ou résultat de la sorte *Reseau*, dans notre exemple il s'agit des prédicats et opérations unaires sur *Sommet* et *Ligne*, tels

   $\_\_est\_connectee\_orig : Ligne$ et

   $id\_de\_\_ : Ligne$,

   autres que ceux dans le premier cas.

Les opérations du premier cas peuvent être transformées de façon *équivalente en Fortran*, comme pour les prédicats et opérations d'ABSTRACT2, car elles ont toutes des *profils dépendants* (si nous adaptons la définition de profils dépendants donnée pour ABSTRACT2 aux produits dépendants évoqués ci-dessus).

Pour les prédicats et opérations du troisième cas, il s'agit de remplacer la sorte *Sommet* resp. *Ligne* par la sorte *SommetDeReseau* resp. *LigneDeReseau* dans les profils. Le résultat de chaque prédicat ou opération au profil modifié ne dépend pas du deuxième composant du produit dépendant en entrée, mais uniquement du premier composant, qui est l'argument du prédicat resp. de l'opération à l'ancien profil. Chaque prédicat au profil modifié doit être vrai si et seulement si le prédicat à l'ancien profil est vrai pour le premier composant de son argument. Le résultat de chaque opération au profil modifié doit être égal au résultat de l'opération à l'ancien profil appliquée au premier composant de son argument.

C'est pour obtenir dans ce cas la *même sémantique sur le premier composant du tuple* qu'avant la modification des profils, que nous avons implémenté ABSTRACT1 par POST-ABSTRACT1. En effet, nous avons ajouté à ABSTRACT1 les axiomes assurant que les domaines des prédicats et opérations aux profils modifiés regroupent comme premiers composants des tuples, toutes les valeurs de la sorte *Sommet* resp. *Ligne*.

Pour les opérations du deuxième cas, il faut modifier la sorte de l'argument et du résultat, en ajoutant un deuxième composant de sorte *Reseau* aux deux. Contrairement aux opérations du troisième cas, il n'y a pas d'argument de sorte *Reseau*, mais, dans le même esprit que pour les transformations *équivalentes en Fortran*, nous ajoutons à l'axiomatisation des opérations aux nouveaux profils la contrainte que le deuxième composant du résultat est égal au deuxième composant de l'argument. Contrairement au troisième cas, le deuxième composant du résultat de chacune de ces opérations aux profils modifiés, varie alors en fonction du deuxième composant de l'argument.

Après ces modifications, les sortes *Sommet*, *Ligne*, $Set[Sommet]$ et $Set[Ligne]$ n'apparaissent dans aucun profil de prédicat ou d'opération non cachés[1]. Nous pouvons dire alors que l'implémentation de ces sortes n'est pas "nécessaire". Nous cachons alors ces sortes.

Appelons *reformulation en produits dépendants* la relation de *"signature shift"* entre deux spécifications SP_A et SP_I telle que

- SP_A contient des sortes S, $S1$, $S2$, … et SP_I contient la sorte S, et les sortes de produits dépendants de S avec $S1$, $S2$, … et dans SP_I les sortes S, $S1$, $S2$, … sont cachées ;
- certaines opérations (celles du premier cas ci-dessus) de SP_A sont remplacés dans SP_I par d'autres, de la même façon que dans une relation d'*équivalence en Fortran* entre les deux spécifications ;

1. Ce n'est vrai que si les opérations de construction et de projection des produits dépendants sont définies uniquement sur les sortes cachées des produits cartésiens, comme dans l'exemple que nous présentons en annexe.

- certains prédicats (ceux du troisième cas ci-dessus) de SP_A sont remplacés dans SP_I par des prédicats des mêmes profils, sauf qu'un argument $p1$ du prédicat pr_A de SP_A de sorte $S1$ ou $S2$ ou ... est remplacé par un argument $p12$ du prédicat pr_I de SP_I de la sorte du produit dépendant de $S1$ resp. $S2$ resp. ... et de S ;
- certaines opérations (celles du troisième cas ci-dessus) de SP_A sont remplacées dans SP_I par des opérations des mêmes profils, sauf que chaque argument $p1$ de l'opération op_A de SP_A de sorte $S1$ ou $S2$ ou ... est remplacé par un argument $p12$ de l'opération op_I de SP_I de la sorte du produit dépendant de $S1$ resp. $S2$ resp. ... et de S ;
- certaines opérations (celles du deuxième cas ci-dessus) de SP_A sont remplacées dans SP_I par des opérations des mêmes profils, sauf que
  - chaque argument $p1$ de l'opération op_A de SP_A de sorte $S1$ ou $S2$ ou ... est remplacé par un argument $p12$ de l'opération op_I de SP_I de la sorte du produit dépendant de $S1$ resp. $S2$ resp. ... et de S et
  - le résultat de sorte $S1$ ou $S2$ ou ... d'op_A est remplacé dans op_I par un résultat de la sorte du produit dépendant de $S1$ resp. $S2$ resp. ... et de S ;
- la contrainte suivante est respectée pour chaque opération op_I de SP_I remplaçant une opération de SP_A dont la sorte du résultat est $S1$ ou $S2$ ou ... (où SiS est la sorte d'un produit dépendant de $S1$ ou $S2$ ou ... et de S) :
  $\forall\ p12 : SiS.2nd\_of\ op_I(\ldots, p12, \ldots) = 2nd\_of\ p12$
- et SP_A et SP_I ont la *même sémantique en ce qui concerne le premier composant* de chaque produit dépendant, dans le sens suivant :
  - en enlevant de SP_I le "hiding" des sortes $S1$, $S2$, ... $S1$, $S2$, ... ;
  - en ajoutant à SP_I les définitions des opérations du premier cas ci-dessus remplacées dans SP_A, comme pour obtenir l'*équivalence en Fortran* ;
  - en ajoutant à SP_I la définition suivante de chaque prédicat pr_A du troisième cas ci-dessus remplacé dans SP_I par pr_I :
    $\forall\ p12 : SiS.pr_A(\ldots, 1st\_of\ p12, \ldots) \iff pr_I(\ldots, p12, \ldots)$;
  - en ajoutant à SP_I la définition suivante de chaque opération op_A du troisième cas ci-dessus remplacée dans SP_I par op_I :
    $\forall\ p12 : SiS.op_A(\ldots, 1st\_of\ p12, \ldots) = op_I(\ldots, p12, \ldots)$;
  - en ajoutant à SP_I la définition suivante de chaque opération op_A du troisième cas ci-dessus remplacée dans SP_I par op_I :
    $\forall\ p12 : SiS.op_A(\ldots, 1st\_of\ p12, \ldots) = 1st\_of\ op_I(\ldots, p12, \ldots)$;
  - et en cachant les sortes des produits dépendants de $S1$, $S2$, ... et de S,

  nous obtenons une spécification $(SP_I'|_{\Sigma_A})$ *équivalente* à SP_A.

La spécification résultant de cette transformation de POST-ABSTRACT1 est POST-ABSTRACT1PD, présentée dans l'Annexe V.

12.4.3 L'implémentation de Post-Abstract1 reformulée par Pre-Abstract2 reformulée

PRE-ABSTRACT2PD est une implémentation correcte de POST-ABSTRACT1PD. Cette implémentation correcte repose sur l'implémentation des produits dépendants de POST-ABSTRACT1PD par les produits dépendants de PRE-ABSTRACT2PD :

- *SommetDeReseau* est implémentée par *NumSommetDeReseau*,
- *LigneDeReseau* est implémentée par *NumLigneDeReseau*,
- *EnsembleSommetsDeMemeReseau* est implémentée par *EnsembleNumsSommetsDeMemeReseau* et
- *EnsembleLignesDeMemeReseau* est implémentée par *EnsembleNumsLignesDeMemeReseau*.

Un morphisme évident relie les autres prédicats et opérations des deux spécifications. Cette implémentation est inhabituelle, mais apparemment parfaitement correcte. Elle intègre l'implémentation de la sorte *Reseau* justifiée dans la description de l'implémentation correcte de la Section 11.3.3 du chapitre précédent. Elle est justifiée par l'extension conservatrice PRE-ABSTRACT2PD′ (présentée dans l'Annexe X) de PRE-ABSTRACT2PD contenant les définitions des symboles cachés de POST-ABSTRACT1PD. Comme ABSTRACT2 ′ (voir Section 11.3.3 du Chapitre 11), PRE-ABSTRACT2PD′ contient des opérations *sommet* et *ligne* (spécifiant la construction des objets de sorte *Sommet* resp *Ligne* à partir du réseau et d'un numéro de sommet resp. ligne), ainsi que deux axiomes précisant que les sortes *Sommet* et *Ligne* sont partitionnées par les valeurs des attributs (nécessaires à la preuve des axiomes de POST-ABSTRACT1PD contenant des égalités entre objets de ces sortes).

12.4.4 L'implémentation contextuelle

Nous avons présenté une explication de l'implémentation d'ABSTRACT1 par ABSTRACT2 en termes de produits dépendants. Dans cette explication, nous avons tenté de concrétiser notre compréhension intuitive du fragment de code. Nous avons expliqué (dans la Section 10.4 du Chapitre 10) que les listes des numéros de ligne construites par le code ne sont pas des listes ordonnées selon la spécification habituelle. Toutefois, si nous transformions chaque liste en une liste de produits dépendants de numéro de ligne et du réseau, en construisant pour chaque élément de la liste, le tuple regroupant cet élément et le réseau, alors cette liste serait une liste ordonnée. Car ces produits dépendants seraient ordonnés par une relation binaire d'ordre. Le réseau est le contexte implicite sous-jacent à notre compréhension intuitive du code et la reformulation des spécifications en termes de produits dépendants peut aider à faire le lien entre cette compréhension intuitive et les spécifications intermédiaires proposées dans le Chapitre 10.

Bien sûr, nous ne proposons pas de garder comme spécifications intermédiaires des spécifications exprimées en termes de produits dépendants, trop compliquées à comprendre. La reformulation en termes de produits dépendants nous sert simplement à caractériser et à comprendre un certain type d'étape d'implémentation. Définissons alors la relation d'*implémentation contextuelle* entre deux spécifications SP_A et SP_I telle que SP_I reformulée en termes de produits dépendants par une transformation *équivalente en Fortran* est une implémentation correcte de SP_A transformée par une *reformulation en produits dépendants* . L'implémentation correcte reposant sur l'implémentation des produits dépendants de SP_A reformulée par ceux de SP_I reformulée.

L'implémentation de SP_A par une spécification SP_I qui en est une implémentation contextuelle, semble justifiée dans tous les cas où SP_A spécifie un fragment d'un code entier, à condition que nous arrivons à reformuler toutes les spécifications des autres fragments de $TOUT_A$ par une reformulation en produits dépendants. C'est-à-dire, à condition que

- tous les prédicats et opérations contenant une sorte d'un premier composant de l'un des produits dépendants de l'implémentation contextuelle appartiennent à l'un des trois cas évoqués dans la Section 12.4.2 ci-dessus [2] (et ont donc leurs profils modifiés dans la reformulation en produits dépendants),

- ces prédicats et opérations de SP_A ne font pas partie de l'interface du code entier avec l'extérieur et

- pour chaque sorte de produit dépendant, pour chaque objet appartenant à la sorte du premier composant, la sorte du produit dépendant contient au moins un tuple dont le premier composant est cet objet.

12.4.5 L'implémentation de Pre-Abstract2 par Abstract2

Les opérations
$sommets\_de\ \_\_ : Reseau \to Set[Sommet]$ et
$lignes\_de\ \_\_ : Reseau \to Set[Ligne]$

2. Sans doute, nous pourrions élargir la définition de ces cas, par exemple, aux prédicats et opérations manipulant des lignes ou des sommets de réseaux différents, mais nous ne le faisons pas ici par crainte de trop compliquer notre explication.

d'ABSTRACT1 sont des *opérations à profil dépendant* implémentées dans PRE-ABSTRACT2 par les opérations

$nums\_sommets\_de\ \_\_ : Reseau \rightarrow Set[NumS]$ et

$nums\_lignes\_de\ \_\_ : Reseau \rightarrow Set[NumL]$,

mais dans ABSTRACT2 par

$\_\_.NS : Reseau \rightarrow NumS\_ou\_0$ et

$\_\_.NL : Reseau \rightarrow NumL\_ou\_0$.

Ceci parce qu'il s'agit du cas spécial des opérations ayant comme résultat l'ensemble des numéros de **tous** les sommets (resp. toutes les lignes) du réseau. Dans ce cas, l'ensemble de numéros de sommet (resp. ligne) forme, par définition de la numérotation, un intervalle sur les positifs, compris entre la valeur 1 et le nombre de sommets (resp. lignes) dans le réseau. Alors, au lieu d'implémenter cet intervalle comme n'importe quel autre ensemble, nous l'implémentons dans ABSTRACT2 par le numéro de sommet (resp. ligne) qui est la borne supérieure de l'intervalle, ou la valeur 0 en cas de l'ensemble vide.

Le test is_in d'appartenance d'un numéro de sommet (resp. ligne) à cet ensemble particulier est implémenté par le test de son infériorité ou égalité à la borne supérieure (ou 0).

L'implémentation de PRE-ABSTRACT2 par ABSTRACT2 est

- un double cas de raffinement des profils,

  1. le profil de $nums\_sommets\_de$ est raffiné à

     $nums\_sommets\_de : Reseau \rightarrow EnsembleTousNumsSommetsDuReseau$,

     avec l'élimination de la sur-sorte $Set[NumS]$, qui n'apparaît plus dans aucun profil et

  2. dans le cas de $nums\_lignes\_de$, la sur-sorte, $Set[NumL]$, n'est pas éliminée, car elle est employée dans les profils de $nums\_lignes\_sortant$ et $nums\_lignes\_entrant$.

- suivi par une implémentation correcte : la sous-sorte $EnsembleTousNumsSommetsDuReseau$ de $Set[NumS]$ regroupant tous les numéros contenus dans un intervalle commençant par 1, est implémentée par la sorte $NumS\_ou\_0$, et is_in est implémenté par $\leq$, ce qui fait coïncider les opérations $nums\_sommets\_de$ et $\_\_.NS$. La sous-sorte $EnsembleTousNumsLignesDuReseau$ de $Set[NumL]$ est implémentée de façon similaire.

Notons que ceci donne deux "implémentations" différentes de la sorte $Set[Ligne]$ d'ABSTRACT1 : dans le cas de l'implémentation de l'opération $lignes\_sortant$ d'ABSTRACT1 elle est remplacée par $Set[NumL]$ et dans le cas de l'implémentation de $lignes\_de$ d'ABSTRACT1, par $NumL\_ou\_0$.

En effet, dans l'implémentation contextuelle nous avons remplacé $Sommet$ par $NumS$ et $Ligne$ par $NumL$ partout. Mais il n'y a pas toujours la même correspondance en ce qui concerne les sous-sortes des sortes "génériques", telles que $Set[\_\_]$, paramétrées soit par la sorte $Sommet$ ou la sorte $Ligne$, soit par $NumS$ ou $NumL$. Les sous-sortes de $Set[Ligne]$ caractérisées par une propriété des **lignes** ne correspondent pas à des sous-sortes de $Set[NumL]$ (car les numéros de ligne n'ont pas les mêmes propriétés que les lignes) et, inversement, nous pouvons définir sur les ensembles de numéros de ligne des propriétés (telles que la contiguïté) qui n'ont pas de sens pour les ensembles de lignes correspondant. Alors $Set[Ligne]$ est remplacée dans le profil par $Set[NumL]$ dans le cas de $lignes\_sortant$ et par $NumL\_ou\_0$ dans le cas de $lignes\_de$.

12.5 L'abstraction correcte

Nous proposons la définition suivante de l'*abstraction correcte* d'une spécification algébrique d'un fragment d'un code entier, par une autre. Une abstraction est *correcte* si elle est une instance de l'une des relations suivantes[3] :

3. Cette définition de l'*abstraction correcte* est partielle, car il serait sans doute possible de trouver d'autres relations entre spécifications algébriques caractérisant l'inverse d'autres étapes "typiques" d'implémentation, à ajouter à celles-ci.

12.5.1 L'inverse de l'*implémentation correcte*

Nous avons rappelé la définition de l'*implémentation correcte* dans la Section 11.1 du Chapitre 11. Son inverse est le type d'abstraction résultant naturellement du paradigme de développement de logiciels à l'aide des méthodes formelles et des travaux sur le raffinement des spécifications algébriques. Il donne une spécification abstraite qui permet plus de modèles (ou programmes Fortran) et est donc, dans ce sens, moins "précise" que la spécification dont elle est l'abstraction. En particulier,

- la spécification abstraite peut être moins **complète** que l'autre : un exemple est l'abstraction d'ABSTRACT4 contenant une spécification de listes ordonnées et précisant l'ordre des éléments de la liste résultat de *nums_lignes_sortant*, par ABSTRACT3 qui spécifie les listes mais pas les listes ordonnées et ne précise pas l'ordre des éléments du résultat de *nums_lignes_sortant*. Remarquons que dans ce cas, nous pouvons **justifier** l'"oubli" de l'ordre des éléments par une abstraction "parallèle" de l'utilisateur `CASDEG` du résultat de *nums_lignes_sortant*, contenant une spécification déclarative du calcul effectué par `CASDEG`, qui ne dépend pas de l'ordre des éléments des listes. L'abstraction d'une spécification de *nums_lignes_sortant* précisant l'ordre des éléments par une spécification qui ne le précise pas est une instance de l'abstraction par *relâchement de la postcondition* proposée par [GC99b].

- Ou l'abstraction peut "remplacer" un type abstrait de données par un autre moins **"riche"**. Il s'agit souvent de l'inverse d'une *implémentation comportementale*, car le type abstrait de données plus riche contient plus d'objets. Un premier exemple est l'abstraction du type abstrait de données spécifié par LISTDIFFELEMS dans ABSTRACT3 par le type abstrait de données spécifié par SET dans ABSTRACT2. Nous avons déjà remarqué (au début du Chapitre 10) que dans cet exemple, nous aurions pu choisir la forme précédente d'abstraction, vers une spécification "loose" des ensembles satisfaite aussi par les listes sans doublons. Un autre exemple est l'abstraction du réseau numéroté spécifié par ABSTRACT2 par le réseau non numéroté spécifié par ABSTRACT1.

L'inverse de l'implémentation correcte permet aussi l'abstraction d'une opération op_I par une opération partielle op_A de domaine plus restreint, à condition que la spécification du résultat de op_A soit respectée par op_I sur tout le domaine d'op_A. Dans ce cas, l'inverse de l'implémentation correcte correspond à l'abstraction par *renforcement de la précondition* proposée dans [GC99b], car la précondition est remplacée dans notre formalisme par le domaine. Dans un contexte de rétro-ingénierie d'un fragment d'un code entier, nous devons ajouter comme condition (ou justification) de cette abstraction qu'op_I n'est **appliquée** qu'aux arguments appartenant au domaine d'op_A.

Par exemple, l'opération d'insertion d'un élément dans une liste ordonnée n'est normalement définie que sur les listes ordonnées. Si nous décidons d'implémenter les listes ordonnées de la même façon que les listes non ordonnées, et l'opération d'insertion *insert* par une opération *insert+* qui traverse la liste en cherchant un élément plus petit que celui à insérer, et puis l'insère à ce point, alors op_I est définie sur toutes les listes, mais ne construit une liste ordonnée que si elle est appliquée à une liste ordonnée. Dans ce cas, et si *insert+* est toujours appliquée aux listes ordonnées, nous pouvons l'abstraire par *insert* définie uniquement sur les listes ordonnées. Éventuellement, nous pouvons aussi "abstraire" la sorte des listes par la sorte des listes ordonnées.

12.5.2 L'inverse de l'*implémentation par décomposition*

L'*implémentation par décomposition* d'un fragment d'un code entier est définie dans la Section 12.2. Son inverse permet de "faire abstraction" du stockage et de la relecture de résultats intermédiaires du calcul effectués par le code afin d'économiser du temps de calcul, en les remplaçant par une opération dont l'implémentation **n'est pas encapsulée sous forme de sous-programme Fortran** mais est diffusée dans le code. Le remplacement de plusieurs prédicats ou opérations par leur composition, permet d'éliminer le type abstrait de données servant au stockage des résultats intermédiaires.

Un exemple de ce type d'abstraction est le passage de la spécification (ABSTRACT6) de la construction et de la relecture d'une table stockant les résultats d'un calcul, vers la spécification de la fonction (*nums_lignes_sortant*) dont le graphe pour un réseau donné est stocké dans la table.

12.5.3 L'inverse du *raffinement des profils avec élimination de la sur-sorte*

Le *raffinement des profils avec élimination de la sur-sorte* d'un fragment d'un code entier est définie dans la Section 12.3. Son inverse permet d'"élargir" des profils de prédicats ou d'opérations et de reformuler la spécification en termes de types abstraits de données plus généraux et donc mieux connus (mais qui ne sont que **partiellement** implémentés par les structures de données et sous-programmes du code Fortran), en "faisant abstraction" de la spécialisation de ces types abstraits de données effectuée dans le code afin d'économiser de l'espace mémoire ou du temps d'exécution.

Il s'agit alors de l'abstraction de types abstraits de données **spécialisés à leur contexte d'utilisation** par des types abstraits de données plus "généraux" pouvant être employés dans des contextes plus larges.

Un exemple de ce type d'abstraction est le passage d'une spécification (ABSTRACT7) formulée en termes de tables de listes partitionneuses vers une spécification formulée en termes de tables plus générales.

Ce mécanisme d'abstraction semble être l'opposé de celui du renforcement de la précondition décrit à la fin de la Section 12.5.1, dans laquelle une opération et éventuellement une sorte sont abstraites par une opération partielle de domaine plus restreint et une sorte plus petite. Mais dans ce dernier cas, il s'agit de reconnaître le domaine "effectif" de l'opération, et ainsi son "vrai" sens.

12.5.4 L'inverse du *raffinement des profils*

Le *raffinement des profils* d'un fragment d'un code entier est définie dans la Section 12.3. Son inverse permet de "faire abstraction" des propriétés conférées par une implémentation particulière et qui permettent d'économiser de l'espace mémoire ou du temps d'exécution.

Par exemple, dans l'abstraction (dont l'inverse est décrit dans la Section 12.4.5) d'une spécification contenant une opération dont le résultat est de la sorte des ensembles de numéros contigus de lignes, par une spécification dans laquelle la sorte du résultat de cette opération est généralisée à la sorte de tous les ensembles de numéros de lignes, nous "faisons abstraction" de la propriété de contiguïté des numéros de ligne, afin de préparer l'abstraction par un réseau dans lequel les lignes ne sont pas numérotées, et où cette propriété n'a pas de sens.

12.5.5 L'inverse de l'*implémentation contextuelle*

L'*implémentation contextuelle* d'un fragment d'un code entier est définie dans la Section 12.4.4. Son inverse permet d'introduire dans la spécification abstraite une indépendance des types abstraits de données qui **n'est pas respectée par le code Fortran "optimisé" au contexte**.

Un exemple de ce type d'abstraction est le passage d'une spécification (ABSTRACT2) formulée en termes de numéros de sommet et de ligne, vers une spécification formulée en termes de sommet et de ligne, même quand les numéros de sommet ne sont pas une implémentation correcte des sommets et les numéros de ligne ne sont pas une implémentation correcte des lignes. Un "groupe" de sortes liées d'une certaine façon dans les profils de tous les prédicats et opérations est donc remplacé, dans l'abstraction de la spécification, par un autre "groupe" de sortes.

Conclusion

Nous proposons une méthode de rétro-ingénierie de codes Fortran77 vers des spécifications algébriques vraiment abstraites.

Notre méthode de rétro-ingénierie est surtout adaptée aux codes Fortran77 employés par EDF (au service Etudes de Réseaux de la Division Recherche et Développement). Notre approche est basée sur les méthodes formelles et plus particulièrement les spécifications algébriques. Elle se distingue des autres approches comparables par le niveau élevé d'abstraction visé. Il ne nous semble pas possible d'atteindre ce niveau d'abstraction sans une part manuelle d'analyse et de preuve, que nous cherchons tout de même à limiter.

La première phase est une suite d'étapes d'analyse et de transformation du code, culminant en la réalisation d'une spécification *concrète*. Elle inclut l'inférence des *profils* de la spécification à l'aide de l'outil *PIPS*.

Notre méthode commence par une première phase semi-automatique d'analyse, de transformation et de décomposition du code. Ensuite, le rétro-ingénieur crée deux spécifications algébriques de chaque fragment résultant de la décomposition : la spécification *concrète* et la spécification *abstraite*. Finalement, il élabore une justification formelle du fait que la spécification abstraite constitue une abstraction pertinente de la spécification concrète. Souvent, cette justification passe par la création d'autres spécifications algébriques à des niveaux d'abstraction intermédiaires.

En ce qui concerne la première phase, nous avons élaboré une "démarche" composée d'une suite d'étapes d'analyse ou de transformation. Certaines de ces étapes sont reconnues comme utiles par de nombreux auteurs du domaine. D'autres ressemblent à des propositions d'autres auteurs dans le domaine, mais nous les avons adaptées aux spécificités du Fortran77 et des codes du service Etudes de Réseaux analysés, en proposant de nouvelles heuristiques.

Nous proposons une étape qui est entièrement nouvelle : il s'agit de l'inférence des *profils* des *prédicats* et *opérations* de la spécification algébrique, à partir de l'analyse des "vrais" paramètres formels et effectifs des sous-programmes Fortran77. Nous voulons obtenir un ensemble cohérent de profils pour les prédicats et opérations spécifiant les sous-programmes du code, mais qui emploient des sortes d'une granularité plus fine que les types Fortran déclarés des paramètres. Nous "oublions" alors les déclarations de type Fortran et partons de la règle qu'un paramètre formel de sous-programme a la même sorte que tous les paramètres effectifs l'instanciant dans les différents sites d'appel. Il faut d'abord analyser le code afin de déterminer les "vrais" paramètres d'entrée et de sortie des sous-programmes, car les déclarations du code ne suffisent pas. Dans le cas de tableaux, nous n'obtenons pas des résultats de bonne qualité si nous supposons toujours que le vrai paramètre est tout le tableau. Car des ensembles différents d'éléments du même tableau peuvent être effectivement passés à différents sous-programmes (ou instancier différents paramètres du même sous-programme) et nous ne voulons pas toujours donner la même sorte à ces ensembles différents d'éléments. C'est pourquoi nous proposons une analyse qui prend en compte les éléments effectivement référencés.

Nous avons passé en revue de nombreux outils d'analyse statique des codes Fortran77, ainsi que les fonctions de restructuration de deux d'entre eux. Le plus puissant en analyse statique est *PIPS*, et nous avons testé et raffiné notre inférence des profils à l'aide d'un prototype réalisé comme une extension de cet outil. Les possibilités d'automatisation des autres étapes de la première phase sont indiquées, mais la réalisation de tous les outils nécessaires n'était pas possible dans le cadre de cette thèse.

À la suite de ces transformations et analyses, nous proposons de réaliser la spécification concrète de chaque fragment du code entier. Cette spécification se situe à un niveau d'abstraction suffisamment bas pour convaincre de la fidélité de sa représentation du code et pour pouvoir retrouver facilement dans le code les objets décrits par la spécification, mais suffisamment élevé pour être plus claire et plus compréhensible que le code lui-même. Ainsi, cette première spécification est déjà plus abstraite que celle qui serait fournie par un générateur semi-automatique, comme proposé par d'autres approches à la rétro-ingénierie. Nous donnons un guide de l'élaboration de la spécification concrète, en prenant comme exemple un fragment d'un code réel fourni par EDF.

La deuxième phase consiste en la réalisation et la justification d'une spécification *abstraite* du code. Nous proposons une définition de l'*abstraction correcte* d'une spécification algébrique qui est plus générale que l'inverse de l'*implémentation correcte* .

Dans notre expérience, la compréhension intuitive d'un code vient de

- les restructurations automatiques et manuelles afin de simplifier les structures de contrôle,
- l'analyse des conventions de codage dans les structures de données et
- la décomposition du code en fragments correspondant à des types abstraits de données, c'est-à-dire : en structures de données avec leurs opérations d'écriture et de lecture,

ainsi que de l'étude des commentaires et des documents éventuels. Nous croyons qu'une abstraction d'un niveau élevé traduit la compréhension intuitive du rétro-ingénieur (aidé par un expert du domaine dans certains cas) et ne peut être obtenue automatiquement.

Nous proposons alors de réaliser, sans se référer à la spécification concrète, la deuxième spécification algébrique du fragment de code étudié : c'est une spécification abstraite qui exprime la compréhension intuitive du code. Cette spécification doit décrire la finalité du calcul effectué par le fragment de code, en termes d'un modèle du réseau électrique, sans décrire comment il est réalisé. Elle doit donc être compréhensible par un expert en réseaux électriques qui n'est pas forcément un informaticien.

La spécification concrète est justifiée par sa proximité du code. Pour la justification de la spécification abstraite, nous la comparons à la spécification concrète en nous appuyant sur la notion très générale de l'*implémentation correcte* d'une spécification algébrique par une autre. L'inverse de l'implémentation correcte permet de caractériser l'abstraction fonctionnelle, l'abstraction comportementale et l'abstraction des données.

Cependant, nous nous sommes rendus compte que l'inverse de l'implémentation correcte ne suffit pas pour caractériser notre idée intuitive de l'abstraction d'un fragment d'un code entier. En effet, nous sommes obligés dans la pratique de décomposer un code d'EDF en fragments et d'en effectuer la spécification fragment par fragment. Mais ces fragments ne correspondent pas à des modules de "bibliothèque" logicielle ou à des "composants" dans une approche de programmation à partir de composants logiciels. Les fragments que nous étudions ont été réalisés pour un seul "utilisateur" : le code entier étudié. À cause des impératifs de l'époque de leur réalisation, les fragments de code sont donc le plus souvent spécialisés à leur contexte d'utilisation, afin d'économiser du temps d'exécution ou de la place mémoire.

Mais nous voulons une spécification abstraite la plus claire possible, ce qui implique d'employer le plus possible des types abstraits de données classiques et donc bien connus, et d'éviter une spécification abstraite tellement *"loose"* qu'elle n'a pas de modèle intuitif évident.

C'est pourquoi nous avons caractérisé le cas général de l'implémentation d'un fragment d'un code entier "optimisé", ainsi que trois instances que nous espérons typiques. Les inverses de ces instances typiques d'implémentation "incorrecte" permettent de

- "faire abstraction" du stockage des résultats intermédiaires des calculs,
- abstraire un type abstrait de données spécialisé au contexte d'utilisation par un type abstrait de données plus classique au contexte plus général et
- réaliser des définitions indépendantes de types abstraits de données dont des implémentations interdépendantes sont réalisées en réalité, pour des questions d'économie de temps d'exécution ou de place mémoire.

Est-ce que notre méthode atteint l'objectif fixé?

Nous avons expliqué dans l'introduction à cette thèse, que le but du développement de notre méthode de rétro-ingénierie était la construction d'une description des "vieux" codes Fortran77 du service Etudes de Réseaux, permettant de maîtriser leur évolution et leur remplacement.

Nous sommes convaincus de l'intérêt des résultats de notre approche.

Les résultats de notre approche sont illustrés par la spécification concrète du fragment de code que nous avons pris en exemple, ainsi que par la spécification la plus abstraite du même fragment et les spécifications à des niveaux d'abstraction intermédiaires, dont chacune intègre un choix d'implémentation. Il faut un langage (ou au moins un cadre) pour exprimer la description désirée du code et une description sous forme de spécification algébrique est facilement comprise, analysée et traduite vers d'autres formalismes, car elle est exprimée en logique du premier ordre. En même temps, le caractère formel du langage oblige à beaucoup de rigueur et garantit que la description ne contient rien d'implicite ou d'ambigu.

Notre démarche a permis d'expliciter, pour le fragment de code pris en exemple :

- le calcul effectué et le modèle du réseau sous-jacent ;
- l'intérêt de certains choix d'implémentation ;
- les conventions de codage des données ;
- la propriété des données qui conditionne leur stockage sous la forme choisie ;
- les conditions d'application des sous-programmes pour qu'elles réalisent les fonctions voulues et
- l'interdépendance des implémentations des types abstraits de données différents.

Par contre, l'obtention de nos résultats a demandé beaucoup de temps.

Il faudrait automatiser des étapes de l'analyse initiale, mais aussi reconnaître les codes respectant un style de programmation qui facilite cette analyse et encourager les programmeurs d'aujourd'hui à adopter ce style, voir à annoter leurs codes.

L'extension d'outils tels que *PIPS* permettrait de pallier au manque d'outils. En particulier, l'automatisation de l'inférence des profils des prédicats et opérations spécifiant tous les sous-programmes permettrait un grand pas en avant dans l'élaboration de la spécification concrète, ainsi que dans la détection de réalisations de types abstraits de données dans le code.

Toutefois, nous avons expliqué comment l'analyse est rendue particulièrement délicate lorsque les codes exploitent certaines possibilités offertes par Fortran77 et qui vont à l'encontre d'une programmation bien structurée, modulaire et respectant les types. En effet, nous pourrions réduire l'effort d'analyse

- soit en s'assurant que le code étudié respecte certaines propriétés qui permettent de simplifier les analyses,
- soit en tenant compte d'annotations que le programmeur aurait ajoutées à son code.

Le Fortran77 est employé même aujourd'hui pour des codes qui doivent être très efficaces. En encourageant les programmeurs à respecter un certain style de programmation ou, lorsque le respect de ce style enlève tout l'intérêt du Fortran77, à ajouter des annotations exprimées dans une extension du langage (comme celles définies dans l'extension *Anna* d'*Ada*), nous pourrions faciliter énormément les analyses ultérieures à des fins de rétro-ingénierie ou de validation :

- Ainsi, nous pourrions vérifier par une analyse statique automatique que toutes les boucles du code réalisées à l'aide de `GOTO` sont réductibles, et, mieux encore, que toutes les structures de contrôle sont des instances de schémas prédéfinis correspondant aux constructions de la programmation structurée.
- L'inférence de profils que nous proposons serait énormément simplifiée si nous pourrions vérifier aussi que c'est toujours la même plage d'un tableau donné qui est employée comme paramètre effectif dans des appels de sous-programme (ou sinon, que chacune des sous-plages est employée

dans une itération différente de la même boucle). À cause des limites de l'analyse statique il ne sera pas possible dans certains cas de vérifier entièrement cette propriété de façon automatique. Si elle n'est pas respectée, nous pourrions tout de même transformer le code de façon à abstraire la simulation de l'allocation dynamique, comme nous l'avons expliqué, à condition que celle-ci soit réalisée "proprement" à travers une bibliothèque d'utilitaires (comme dans le cas du code *COURCIRC*).

- De façon plus générale, s'interdire l'emploi de paramètres formels d'un sous-programme comme variables locales, voir annoter les sous-programmes avec leurs "vrais" paramètres d'entrée et de sortie (comme nous les avons définis) simplifierait les analyses.
- L'inférence de profils que nous proposons serait améliorée si nous étions sûrs de pouvoir toujours affecter la même sorte à toutes les valeurs successives prises par une même variable Fortran. C'est une propriété que nous ne pouvons pas vérifier automatiquement.
- De façon similaire, nous pourrions demander d'éviter le "double codage" (deux façons d'interpréter la valeur de la même variable) ou d'employer des fonctions explicites de "coercition" en présence du double codage.
- Au lieu d'employer nos heuristiques de décomposition de sous-programmes et de recherche de types abstraits de données, nous pourrions demander aux programmeurs Fortran d'éviter de regrouper des modifications indépendantes de données de sortes différentes dans le même sous-programme, d'éviter, par la duplication des sous-programmes "utilitaires", le polymorphisme (ce qui lèverait l'ultime difficulté dans l'inférence des profils) et ainsi de regrouper les sous-programmes en "modules" correspondant aux types abstraits de données.
- Finalement, si tous les aspects "non-fonctionnels" (messages d'erreur, ...) se trouvaient dans des branches conditionnées par le test d'une certaine variable, alors ces aspects pourraient être automatiquement éliminés du code, avant son analyse et sa spécification concrète, par évaluation partielle.

Pour la justification de la spécification abstraite, il faut des outils d'aide à la comparaison de spécifications des différents niveaux d'abstraction. Par contre, nous espérons avoir apporté une solution au problème de l'insuffisance de la notion d'*implémentation correcte* .

Nous avons passé beaucoup de temps aussi dans la justification de la spécification abstraite de notre exemple principal, surtout en ce qui concerne

- la manipulation de nombreuses spécifications algébriques à des niveaux d'abstraction différents (neuf dans cet exemple) : un outil réalisant la vérification d'un morphisme de signatures entre deux spécifications, la traduction correspondante de la spécification plus abstraite et la génération des obligations de preuve, aurait permis de gagner beaucoup de temps ici, et
- le passage de la spécification ABSTRACT2 vers ABSTRACT1, qui ne pose aucun problème de compréhension intuitive mais un problème difficile "technique". Cependant, la difficulté de la justification de cette étape nous a permis de constater que le type d'abstraction que nous désirons ne se limite pas à l'inverse de l'implémentation correcte. Nous espérons que le schéma général et les cas typiques d'abstraction que nous avons caractérisés à la suite de cette expérience, faciliteront les justifications de l'abstraction d'autres fragments de code.

Finalement, nous sommes convaincus qu'un effort important de rétro-ingénierie sur les premiers codes analysés serait vite amorti sur les suivants.

En effet, nous avons remarqué qu'il y a souvent de nombreuses similarités entre les codes différents employés par le service Etudes de Réseaux. Le même style de programmation, les mêmes structures de données et astuces de codage, sont dupliqués, à des détails près, dans de nombreux codes différents. C'est pourquoi nous sommes convaincus que l'effort de rétro-ingénierie diminuerait avec le nombre de codes du même service analysés. Cet effet serait probablement renforcé par la définition de *clichés* caractérisant les structures de données et algorithmes déjà spécifiées et la recherche de ces clichés dans de nouveaux codes.

Bibliographie

[Abr96] J.R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[AC94] B.L. Achee and Doris L. Carver. A greedy approach to object identification in imperative code. In *Proceedings of the Third Workshop on Program Comprehension, Washington, D.C.*, pages 4–11, November 1994.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Massachusetts, 1986.

[BBGK97] Mohamed S. Bendelloul, Zine E. Bouras, Saïd Ghoul, and Tahar Khammaci. Assistance à la compréhension de programmes : un modèle et un algorithme de fragmentation. *Génie Logiciel*, (45):32–42, Septembre 1997.

[BCH99] Michel Bidoit, Maria Victoria Cengarle, and Rolf Hennicker. Proof systems for structured specifications and their refinements. In Egidio Astesiano, Hans-Jörg Kreowski, and Bernd Krieg-Brückner, editors, *Algebraic Foundations of Systems Specification.* Springer, 1999. See http://www.lsv.ens-cachan.fr/Publis/.

[Ben98] Keith Bennet. Do program transformations help reverse engineering? In *Proceedings of the International Conference on Software Maintenance (ICSM'98), Bethseda, Maryland,* Washington D.C., USA, November 1998. IEEE Computer Society.

[BH96] Michel Bidoit and Rolf Hennicker. Behavioural theories and the proof of behavioural properties. *Theoretical Computer Science*, 165(1):3–55, 1996. See http://www.lsv.ens-cachan.fr/Publis/.

[BLB93] P.T. Breuer, K.C. Lano, and J.P. Bowen. Understanding programs through formal methods. In Hank van Zuylen, editor, *The REDO Compendium of Reverse Engineering for Software Maintenance*, pages 195–223. John Wiley & Sons, 1993.

[BM82] Victor R. Basili and Harlan D. Mills. Understanding and documenting programs. *IEEE Transactions on Software Engineering*, SE-8(3):270–283, May 1982.

[BM97] I. Baxter and M. Mehlich. Reverse engineering is reverse forward engineering. In Ira Baxter, Alex Quilici, and Chris Verhoef, editors, *Proceedings of the Fourth Working Conference on Reverse Engineering, October 6-8, 1997, Amsterdam, The Netherlands*, pages 104–113. IEEE Computer Society, 1997.

[BMW93] Ted J. Biggerstaff, B. G. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *Proceedings of the 15th International Conference on Software Engineering*, pages 482–498. IEEE Computer Society Press, April 1993.

[BST99] Michel Bidoit, Donald Sannella, and Andrzej Tarlecki. Architectural specifications in CASL. In *Proc. 7th Int. Conf. Algebraic Methodology and Software Technology (AMAST'98), Amazonia, Brazil, Jan. 1998*, volume 1584 of *Lecture Notes in Computer Science*, pages 341–357. Springer, 1999. See http://www.lsv.ens-cachan.fr/Publis/, revised and extended version available as LFCS report ECS-LFCS-99-407.

[Bul90] Tim Bull. An introduction to the WSL program transformer. In *Proceedings of the Conference on Software Maintenance, San Diego*, 1990. See http://www.dur.ac.uk/CSM/projects/ma/papers.html.

[BYM+98] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance (ICSM'98), Bethseda, Maryland*, pages 368–377, Washington D.C., USA, November 1998. IEEE Computer Society.

[Byr91] Eric J. Byrne. Software reverse engineering: A case study. *Software – Practice and Experience*, 21(12):1349–1364, December 1991.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conf. Record of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles, CA (Jan)*, pages 238–252, 1977.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Conf. Record of the 6th Annual ACM Symposium on Principles of Programming Languages (POPL), San Antonio, Texas (Jan)*, pages 269–282, 1979.

[CCM96] G. Canfora, A. Cimitile, and M. Munro. An improved algorithm for identifying objects in code. *Journal of Software Practice and Experience*, 26(1):25–48, January 1996.

[CdLM95] Aniello Cimitile, Andrea de Lucia, and Malcolm Munro. Qualifying reusable functions using symbolic execution. In *Proc. 2nd Working Conference on Reverse Engineering, Toronto, Canada, 1995*, pages 178–187, Washington, DC, 1995. IEEE Comp. Soc. Press.

[CF97] Baudoin Le Charlier and Pierre Flener. On the desirable link between theory and practice in Abstract Interpretation. In Pascal Van Hentenryck, editor, *proceedings of Static Analysis 4th International Symposium, SAS'97, Paris, France*, volume 1302 of *LNCS*, pages 379–387. Springer Verlag, September 1997.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing single static assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[Che98] Sophie Cherki. *Rétro-ingénierie de programmes Fortran utilisant les spécifications algébriques*. PhD thesis, Université de Paris-Sud, U.F.R. Scientifique d'Orsay, December 1998.

[CK87] David Callahan and Ken Kennedy. Analysis of interprocedural side effects in a parallel programming environment. In *proc. 1st Intnl. Conf. on Supercomputing, Athens*. Springer-Verlag, 1987.

[Cou81] Patrick Cousot. Semantic foundations of program analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis*, New Jersey, 1981. Prentice-Hall.

[Cow88] P. David Coward. Symbolic execution systems - a review. *Software Engineering Journal*, November 1988.

[CPPGM91] A. Coen-Porisini, F. De Paoli, C. Ghezzi, and D. Mandrioli. Software specialisation via symbolic execution. *IEEE Trans. on Software Engineering*, 17(9):884–899, September 1991.

[CR81] Lori A. Clarke and Debra J. Richardson. Symbolic evaluation methods for program analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis*, New Jersey, 1981. Prentice-Hall.

[Cre96] Béatrice Creusillet. *Array Region Analyses and Applications*. PhD thesis, Ecole des Mines de Paris, December 1996. See http://www.cri.ensmp.fr/pips/.

[EKMP82] H. Ehrig, H.-J. Kreowski, B. Mahr, and P. Padawitz. Algebraic implementation of abstract data types. *Theoretical Computer Science*, 20:209–263, 1982.

[eNWP97] J. Ryckbosch et N. Williams Preston. Rétro ingénierie des codes Fortran à l'aide des spécifications formelles: une étude de cas. Technical Report HR-32-97-010, EDF-DER, 1997.

[GC97] Gerald C. Gannod and Betty H.C. Cheng. A formal automated approach for reverse engineering programs with pointers. In *Proceedings of the 12th Automated Software Engineering Conference, (ASE'97)*, November 1997.

[GC99a] Gerald C. Gannod and Betty H.C. Cheng. A formal approach to reverse engineering: A case study. In *Proceedings of the 6th Working Conference on Reverse Engineering, (WCRE'99)*, Atlanta, GA, USA, October 1999.

[GC99b] Gerald C. Gannod and Betty H.C. Cheng. A specification matching based approach to reverse engineering. In *Proceedings of the 21st International Conference on Software Engineering, (ICSE'99)*, May 1999.

[GH93] John V. Guttag and James J. Horning. *Larch: languages and tools for formal specification.* Springer-Verlag, 1993. Epuisé, mais disponible au http://www.sds.lcs.mit.edu/spd/larch/index.html.

[GKS97] J.F. Girard, R. Koschke, and G. Schied. Comparison of abstract data type and abstract state encapsulation detection techniques for architectural understanding. In Ira Baxter, Alex Quilici, and Chris Verhoef, editors, *Proceedings of the Fourth Working Conference on Reverse Engineering, October 6-8, 1997, Amsterdam, The Netherlands*, pages 66–75. IEEE Computer Society, 1997.

[Hav94] Paul Havlak. Interprocedural symbolic analysis. Technical Report CRPC-TR9451-S, Department of Computer Science, Rice University, Texas, May 1994.

[Hec77] Matthew S. Hecht. *Flow Analysis of Computer Programs.* Elsevier North-Holland, New York, 1977.

[HIBM97] T. Dean Hendrix, James H. Cross II, Larry A. Barowski, and Karl S. Mathias. Tool support for reverse engineering multi-lingual software. In Ira Baxter, Alex Quilici, and Chris Verhoef, editors, *Proceedings of the Fourth Working Conference on Reverse Engineering, October 6-8, 1997, Amsterdam, The Netherlands*, pages 136–145. IEEE Computer Society, 1997.

[HM93] Mary Jean Harrold and Brian Malloy. A unified interprocedural program representation for a maintenance environment. *IEEE Trans. on Software Engineering*, 19(6):584–593, June 1993.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, October 1969.

[HP92] W. E. Howden and Suehee Pak. Problem domain, strutural and logical abstractions in reverse engineering. In *Proceedings of the International Conference on Software Maintenance 1992*, pages 214–224. IEEE Computer Society Press, November 1992.

[HR92] S. Horowitz and T. Reps. The use of program dependence graphs insoftware engineering. In *Proceedings of the 14th International Conference on Software Engineering, (May 11-15, Melbourne, Australia)*, pages 392–411, New York, 1992. ACM. Available at http://www.cs.wisc.edu/wpis/html/.

[IJT91] François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelisation: An overview of the PIPS project. In *proc. International Conference on Supercomputing, June 1991*, pages 144–151, 1991.

[JOR90] Richard J. LeBlanc Jr., Stephen B. Ornburn, and Spencer Rugaber. Recognizing design decisions in programs. *IEEE Software*, 7(1):46–54, January 1990.

[Ken81] Ken Kennedy. A survey of data flow analysis techniques. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis*, New Jersey, 1981. Prentice-Hall.

[KS98] Kathleen Knobe and Vivek Sarkar. Array SSA form and its use in parallelization. In *POPL 98, San Diego CA, USA*, pages 107–120. ACM, 1998.

[KU77] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.

[LD99] Arun Lakhotia and Jean-Christophe Deprez. Restructuring functions with low cohesion. In *Proceedings of the 6th Working Conference on Reverse Engineering, (WCRE'99)*, Atlanta, GA, USA, October 1999.

[LEW96] Jacques Loeckx, Hans-Dieter Ehrich, and Markus Wolf. *Specification of Abstract Data Types*. Wiley and Teubner, 1996.

[LS97] Christian Lindig and Gregor Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proceedings of the 19th International Conference on Software Engineering (ICSE'97), Boston, USA, May 1997*, pages 349–359, 1997. Available at http://www.cs.tu-bs.de/softech/people/snelting/papers.html.

[LY88] Zhiyuan Li and Pen-Chung Yew. Efficient interprocedural analysis for program parallelization and restructuring. *ACM SIGPLAN Notices*, 23(9):85–99, 1988.

[Mar97] Kim Marriott. Abstract Interpretation: a theory of approximate computation. In Pascal Van Hentenryck, editor, *proceedings of Static Analysis 4th International Symposium, SAS'97, Paris, France*, volume 1302 of *LNCS*, pages 367–378. Springer Verlag, September 1997.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(4):348–375, 1978.

[Mül96] Hausi A. Müller. Understanding software systems using reverse engineering technologies research and practice. In *Tutorial Proceedings ICSE-18 18th International Conference on Software Engineering (March 25-29, Berlin, Germany)*, 1996. See http://tara.uvic.ca/rigi/.

[MWT94] Hausi A. Müller, Kenny Wong, and Scott R. Tilley. Understanding software systems using reverse engineering technology. In *The 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences Proceedings (ACFAS)*, 1994. See http://tara.uvic.ca/rigi/concepts/acfas94.html.

[NEK94] Jim Q. Ning, Andre Engberets, and W. (Voytek) Kozaczynski. Automated support for legacy code understanding. *Communications of the ACM*, 37(5):50–57, May 1994.

[OJ97] Robert O'Callahan and Daniel Jackson. Lackwit: A program understanding tool based in type inference. In *International Conference on Software Engineering, Boston, MA, 1997*, pages 338–348, 1997.

[oLD99] The CoFI Task Group on Language Design. The common algebraic specification language summary, July 1999. See http://www.lsv.ens-cachan.fr/ bidoit/CoFI/Documents/CASL/Summary/index.html.

[ONS93] F. Orejas, M. Navarro, and A. Sanchez. Implementation and behavioural equivalence: A survey. In M. Bidoit and C. Choppy, editors, *Recent Trends in Data Type Specification, Proc. 8th Workshop on Specification of Abstract Data Types, Dourdan, France, 1991*, number 655 in Lecture Notes in Computer Science, pages 93–125, Berlin, 1993. Springer-Verlag.

[OT93] C.L. Ong and W.T. Tsai. Class and object extraction from imperative code. *JOOP*, pages 58–68, March–April 1993.

[QWZ97] A. Quilici, S. Woods, and Y. Zhang. New experiments with a constraint-based approach to program plan matching. In Ira Baxter, Alex Quilici, and Chris Verhoef, editors, *Proceedings of the Fourth Working Conference on Reverse Engineering, October 6-8, 1997, Amsterdam, The Netherlands*, pages 114–123. IEEE Computer Society, 1997.

[RP86] Barbara G. Ryder and Marvin C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3):277–316, Sept. 1986.

[RS97] A. Reeves and J. Schlesinger. Jackal: A hierarchical approach to program understanding. In Ira Baxter, Alex Quilici, and Chris Verhoef, editors, *Proceedings of the Fourth Working Conference on Reverse Engineering, October 6-8, 1997, Amsterdam, The Netherlands*, pages 84–93. IEEE Computer Society, 1997.

[RSW95] Spencer Rugaber, Kurt Stirewalt, and Linda M. Wills. The interleaving problem in program understanding. In *Proceedings of the 2nd Working Conference on Reverse Engineering (WCRE'95)*, pages 166–175. IEEE Computer Society Press, July 1995.

[SB96] Gokul V. Subramaniam and Eric J. Byrne. Deriving an object model from legacy Fortran code. In *Proc. 3rd International Conference on Software Maintenance, Monterey, CA*, pages 3–12, November 1996.

[SR97] Michael Siff and Thomas Reps. Identifying modules via concept analysis. In *ICSM '97: IEEE International Conference on Software Maintenance, (Sept. 29 - Oct. 3, Bari, Italy)*, Washington, DC, 1997. IEEE Computer Society. Available at http://www.cs.wisc.edu/wpis/html/.

[ST88] D.T. Sannella and A. Tarlecki. Towards formal development of programs from algebraic specifications: implementation revisited. *Acta Informatica*, (25):233–281, 1988.

[Ste96] B. Steensgaard. Points-to analysis in almost linear time. In *proc. Twentythird Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 1996*, pages 32–41, 1996. Available at http://www.research.microsoft.com/ rusa/papers.html.

[TIF86] Rémi Triolet, François Irigoin, and Paul Feautrier. Direct parallelisation of call statements. In *proc. SIGPLAN 86 Symposium on Compiler Construction, June 1986*, number July in SIGPLAN Notices, pages 176–185. ACM, 1986.

[vdBKV96] M.G.J. van den Brand, P. Klint, and C. Verhoef. Reverse engineering and system renovation - an annotated bibliography -. Technical report, Programming Research Group, University of Amsterdam, 1996. http://adam.wins.uva.nl/ x/reeng/REanno.html.

[War95] Martin Ward. A definition of abstraction. *Journal of Software Maintenance: Research and Practice*, 7(6):443–450, Nov–Dec 1995.

[Wei84] Mark Weiser. Program slicing. *IEEE Trans. on Software Engineering*, SE-10(4):352–357, July 1984.

[Wir90] M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 675–788, BW, 1990. Elseiver.

[WP96] N. Williams-Preston. Retro-ingénierie de logiciels Fortran à l'aide des spécifications algébriques. Technical Report Rapport de stage du DEA Sémantique, Preuves et Programmation, Université Pierre et Marie Curie Paris VI, September 1996.

[WP97] N. Williams-Preston. An experiment in reverse engineering using algebraic specifications. Technical Report 97-10, LSV, Ecole Normale Supérieure de Cachan, 1997. See http://www.lsv.ens-cachan.fr/Publis/.

[WP99] N. Williams-Preston. New type signatures for legacy Fortran subroutines. In *Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'99), Toulouse, France*, pages 76–85, September 1999. See http://www.lsv.ens-cachan.fr/Publis/.

[YLZ98] Hongji Yang, Xiaodong Liu, and Hussein Zedan. Tackling the abstraction problem for reverse engineering in a system re-engineering approach. In *proceedings of the International Conference on Software Maintenance (ICSM'98), Bethseda, Maryland*, Washington D.C., USA, November 1998. IEEE Computer Society. See http://www.cse.dmu.ac.uk/ xdl/.

[ZC91] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, New York, 1991.

[ZRL96] Sean Zhang, Barbara G. Ryder, and William Landi. Program decomposition for pointer aliasing: A step towards practical analyses. In *Proceedings of the 4th Symposium on the Foundations of Software Engineering, San Francisco, October, 1996*, 1996. Available at http://www.prolangs.rutgers.edu.

Index

Annexe A

L'outil *Ftnchek*

```
File prchen.f:
Warning in module PRCHEN file prchen.f:
   Identifiers of undeclared type
        B1              B10              B11              B12
       B13              B14              B15               B2
        B3               B4           B4TROP               B5
        B6               B7               B8               B9
      BF4L               BS              IBT             ICCO
      IPCO              KCH              KCR              KER
       KFB              KFR               KP               KS
         L               N1               N2             NRLE
      SPGM


 1 warning issued in file prchen.f
```

FIG. A.1: Les messages générés par *Ftnchek* pour `PRCHEN`

Ftnchek [1] est un outil du domaine public de génération de références croisées et de détection d'anomalies de programmation pouvant indiquer des erreurs. Programmé en *C*, il utilise les outils *UNIX lex* et *yacc*. On y trouve donc une grammaire *yacc* de Fortran77. *Ftnchek* effectue l'*analyse syntaxique* du code et génère des tables de symboles locales et globales. Chaque symbole est représenté par une structure en *C* avec des champs pour stocker les informations permettant de détecter des anomalies de programmation. *Ftnchek* ne fait aucune *analyse de flot de données*, ainsi il cherche des instances de variables *utilisées* avant d'être *modifiées*, mais dans le sens : l'instruction d'utilisation a un numéro de ligne inférieur à celui de l'instruction de modification, ce qui ne dit rien sur leur ordre d'exécution. En ce qui concerne les appels de sous-programmes, *Ftnchek* fait l'hypothèse que tout paramètre peut être à la fois utilisé et modifié. *Ftnchek* a de nombreuses options d'exécution permettant notamment :

- la génération du graphe des appels,
- la vérification de la concordance entre les paramètres formels et effectifs dans des sites d'appels aux sous-programmes, y compris dimensions des tableaux,
- la vérification de la concordance des déclarations différentes du même bloc `COMMON` : type et dimensions des variables,
- pour chaque bloc `COMMON`, de trouver les unités de programme qui y accèdent vraiment (contrairement à celles qui le déclarent)
- de détecter les sous-programmes existants mais jamais appelés ou appelés mais non-existants ; variables de blocs `COMMON` jamais utilisées ou utilisées mais jamais modifiées et variables locales non-utilisées,
- de générer des déclarations pour les variables non déclarées (voir la Figure A.2).

Ftnchek prend en compte les `EQUIVALENCE` mais, pour ce qui concerne les accès aux tableaux, considère chaque accès à un élément comme un accès au tableau entier.

Les messages générés par *Ftnchek* suite à la vérification de `PRCHEN` (dont le code source est présenté dans l'Annexe G) sont présentés dans la Figure A.1 et les déclarations dans la Figure A.2.

1. développé par Robert MONIOT et disponible au URL http://www.fortran.com/fortran/tools.html, où on peut trouver d'autres outils commerciaux et du domaine public

```
C====>Begin Module PRCHEN    File prchen.f      Undeclared variables
C
C     Argument variables
C
      INTEGER            ICCO(*),    IPCO(*),        L,       N1
      INTEGER            N2,         NRLE(*)
C
C     Local variables
C
      INTEGER            KCH,        KP
C
C     Common variables
C
      INTEGER            IBT,        KER
C
      LOGICAL*1          B4TROP,     BF4L
C
      REAL               SPGM(2)
C
C     Common variables
C
      LOGICAL*1          B1,         B10,        B11,        B12
      LOGICAL*1          B13,        B14,        B15,        B2
      LOGICAL*1          B3,         B4,         B5,         B6
      LOGICAL*1          B7,         B8,         B9,         BS
C
C     Common variables
C
      INTEGER            KCR,        KFB,        KFR,        KS
C
C====>End Module   PRCHEN    File prchen.f
```

FIG. A.2: Les déclarations générées par *Ftnchek* pour `PRCHEN`

Annexe B

Graphes de contrôle tracés par l'outil *Hindsight*

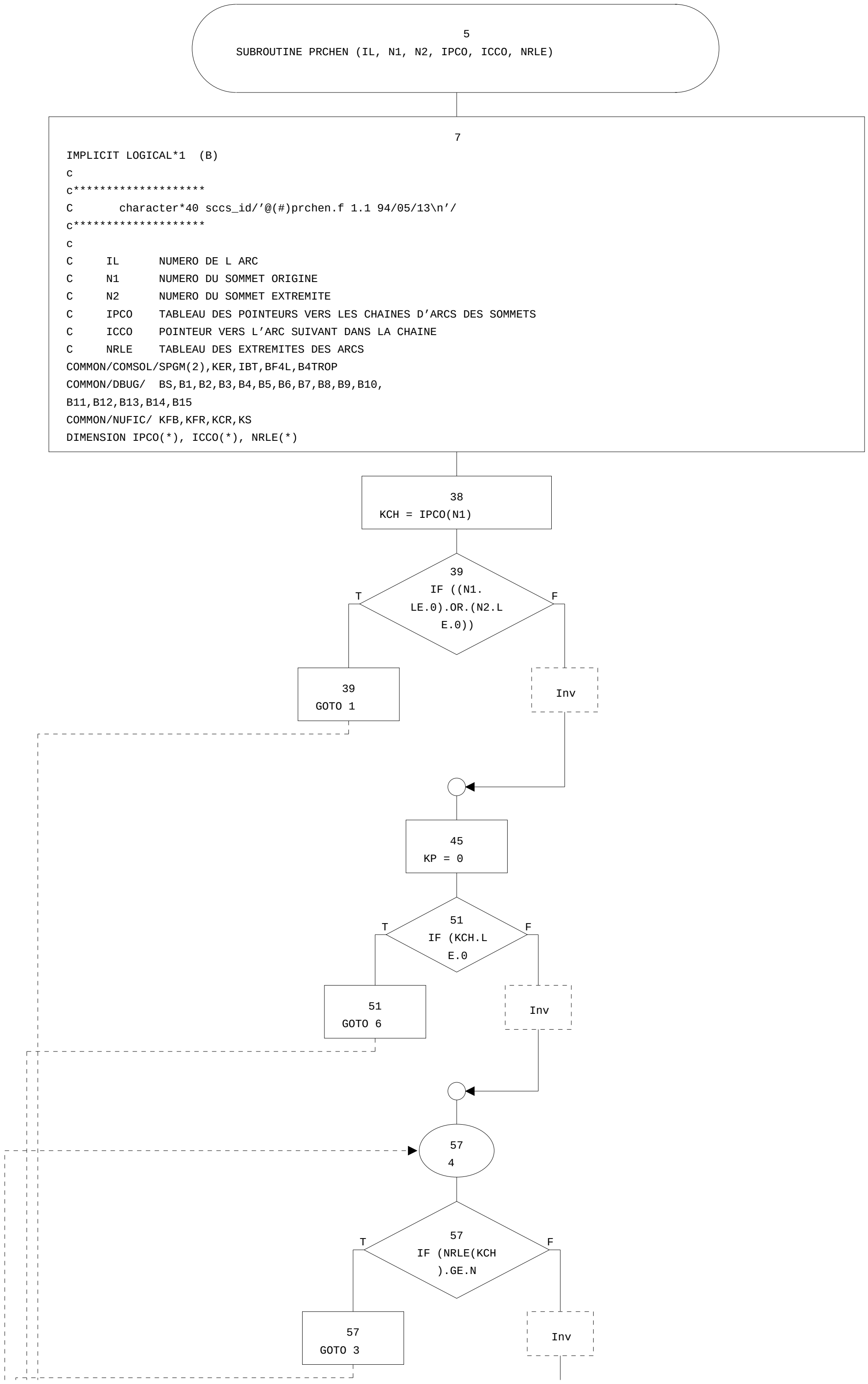


FIG. B.1: Le graphe de contrôle de PRCHEN tracé par *Hindsight* (début)

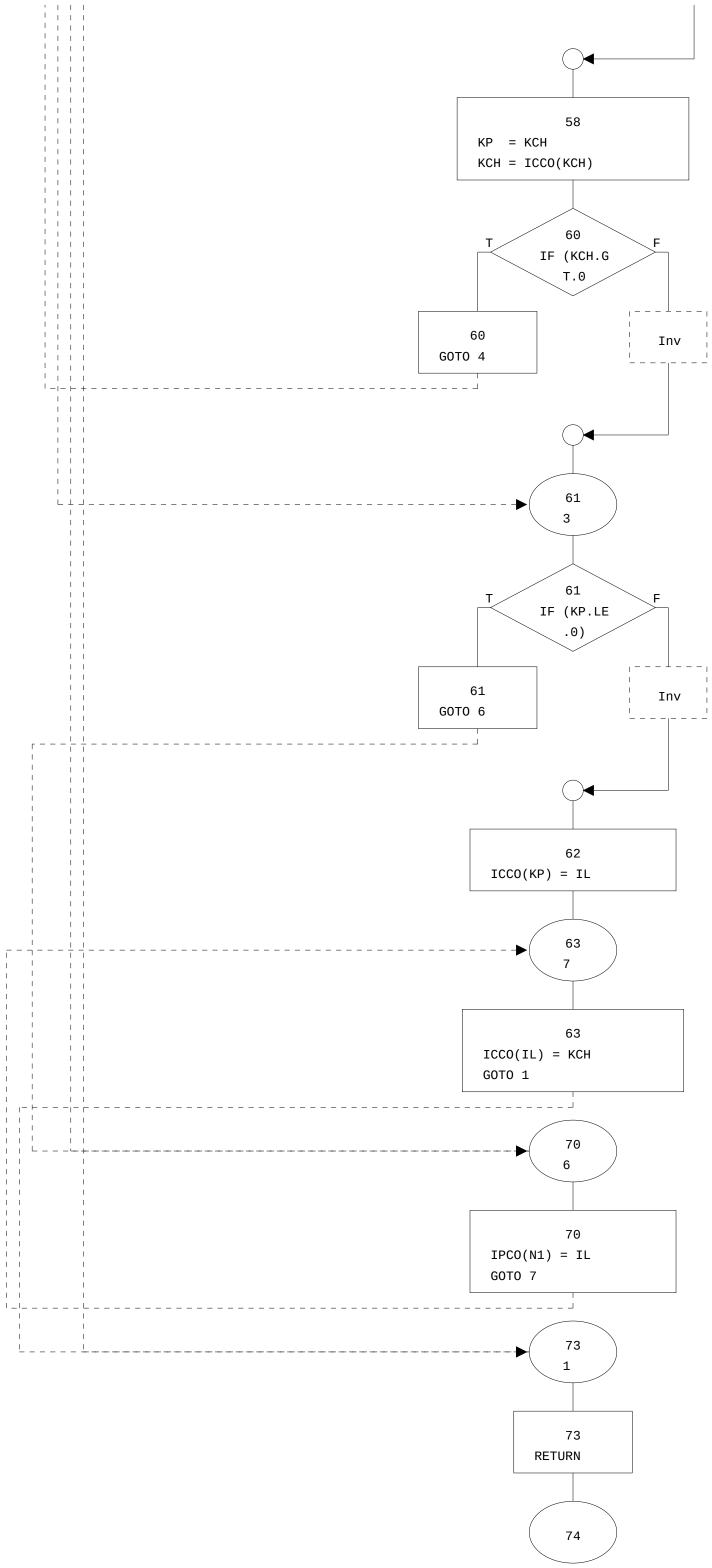


Fig. B.2: Le graphe de contrôle de `PRCHEN` tracé par *Hindsight* (fin)

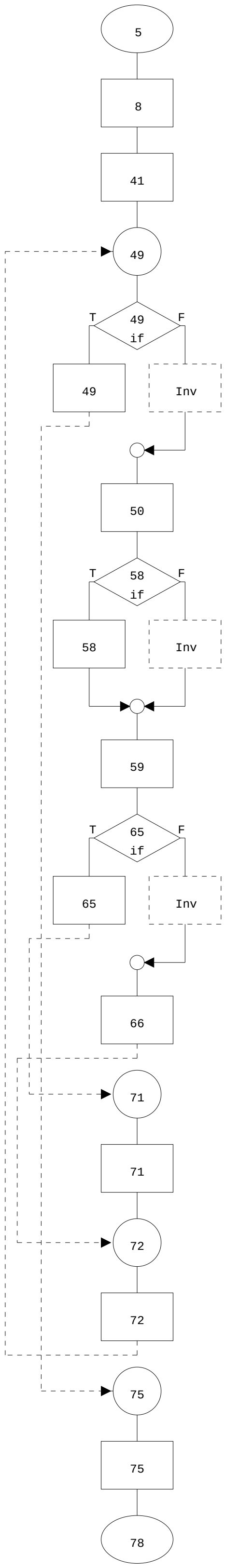


FIG. B.3: Le graphe de contrôle de `CAPIV` tracé par *Hindsight*

Annexe C

Les outils *FORESYS* et *FOREST*

```
Foresys Checker report on prchen in prchen.f

Standard-Warning-12 : Non standard size 1 for type logical
Local-Warning-65 : COMMON /COMSOL/ not used
Local-Warning-65 : COMMON /DBUG/ not used
Local-Warning-65 : COMMON /NUFIC/ not used
```

FIG. C.1: Les messages générés par *FORESYS* pour `PRCHEN`

FORESYS[1] est une suite d'outils pour code Fortran générés à l'aide de *Centaur* et commercialisés par la société *Simulog*. *FORESYS* effectue le même genre de génération de références croisées et de détection d'anomalies de programmation que *Ftnchek*. Les résultats pour `PRCHEN` (dont le code source est présenté dans l'Annexe G) sont présentés dans la Figure C.1.

Il a aussi une fonction d'auto-documentation, qui ajoute aux fichiers source des commentaires indiquant pour chaque unité de programme quels sont les paramètres et variables de zone `COMMON` qui sont certainement *utilisés*, ceux qui sont peut-être utilisés, ceux certainement *modifiés* et ceux peut-être modifiés. Cette annotation est le résultat d'une *analyse inter-procédurale de flot des données* qui prend en compte les instructions `EQUIVALENCE`, mais pas les indices des éléments de tableaux.

L'outil *FOREST* apparenté à *FORESYS* peut effectuer la *restructuration* automatique du code source Fortran77 avec un résultat codé soit en Fortran77, soit en Fortran77étendu ou Fortran90 avec les primitives de contrôle structuré (`while`, `case`, etc.). Les résultats de l'auto-documentation et de la restructuration en code Fortran77étendu des sous-programmes `PRCHEN` et `CAPIV` (dont les versions initiales sont présentées dans les Annexes G et H respectivement) sont présentés dans les Figures C.2 et C.3 respectivement.

FOREST réussit donc une certaine restructuration automatique de la procédure `PRCHEN`. Mais une restructuration plus poussée et qui élimine tous les `GOTO` est possible : elle résulte en le code source de la Figure 6.2 du Chapitre 6 et a été trouvée à la main lors d'une autre étude de cette procédure.

FOREST détecte la boucle *irréductible* du sous-programme `ODIDAV` de *SILODI* (voir la Section 4.3.1 du Chapitre 4).

FORESYS propose la parallélisation automatique du code source. Pour le calcul des *dépendances*, il doit analyser les *régions* des tableaux effectivement lues et modifiées, mais ces informations ne sont pas disponibles à l'utilisateur. L'*évaluation partielle* est proposée aussi : à partir des informations fournies par l'utilisateur sur les valeurs de certaines variables, il cherche à éliminer du code des branches ou des calculs qui sont inutiles pour ces valeurs, sans toutefois fournir à l'utilisateur d'autres résultats que le code transformé.

1. FORESYS est une marque déposée par CONNEXITE. FORESYS est commercialisé par SIMULOG, 1 rue James Joule, 78286 GUYANCOURT *tél* 01.30.12.27.00 *fax* 01.30.12.27.27 *e-mail* info@simulog.fr *www* http://www.simulog.fr/FR/html/prods/foresys.html

```
      subroutine PRCHEN(L,N1,N2,IPCO,ICCO,NRLE)
C
C.. Implicits ..
      implicit logical*1 (B)
C
C.. Formal Arguments ..
C.. In/Out Status: Maybe Read, Not Written ..
      integer L
C.. In/Out Status: Read, Not Written ..
      integer N1
C.. In/Out Status: Read, Not Written ..
      integer N2
C.. In/Out Status: Maybe Read, Maybe Written ..
      integer IPCO(*)
C.. In/Out Status: Maybe Read, Maybe Written ..
      integer ICCO(*)
C.. In/Out Status: Maybe Read, Not Written ..
      integer NRLE(*)
C
C.. Local Scalars ..
      integer KCH,KP
C
C.. External Calls ..
      external MOVEK
C
C.. Common Blocks ..
C... Variables in Common Block /COMSOL/ ...
      integer KER,IBT
      logical*1 BF4L,B4TROP
      real SPGM(2)
      common /COMSOL/ SPGM,KER,IBT,BF4L,B4TROP
C
C... Variables in Common Block /DBUG/ ...
      logical*1 BS,B1,B2,B3,B4,B5,B6,B7,B8,B9,B10,B11,B12,B13,B14,B15
      common /DBUG/ BS,B1,B2,B3,B4,B5,B6,B7,B8,B9,B10,B11,B12,B13,B14,
     &              B15
C
C... Variables in Common Block /NUFIC/ ...
      integer KFB,KFR,KCR,KS
      common /NUFIC/ KFB,KFR,KCR,KS
C
C ... Executable Statements ...
C
      KCH = IPCO(N1)
      KP = 0

      if (KCH > 0) then

        do while (NRLE(KCH) < N2)
          KP = KCH
          KCH = ICCO(KCH)
          if (KCH <= 0) goto 1000
        end do
 1000   if (KP > 0) then
          ICCO(KP) = L
          goto 1100
        end if
      end if

      IPCO(N1) = L
 1100 ICCO(L) = KCH
      end
```

FIG. C.2: Documentation et restructuration de PRCHEN par *FOREST*

```
C       --- ----------------------------------------------------------- ---
C       --- INTRODUCTION D'UN ELEMENT DE MATRICE J DU A UN ELEMENT D'INJECTION
C       --- ----------------------------------------------------------- ---
      subroutine CAPIV(IX,PIVO,NOUV,IPCO,ICCO,NRLX,NCHA,EHA,ZIMP,ATG,SUSL)
C
C.. Formal Arguments ..
      integer IX,NOUV,IPCO(*),ICCO(*),NRLX(*),NCHA(*)
      real PIVO,EHA(*),ZIMP(*),ATG(*),SUSL(*)
C
C.. Local Scalars ..
      integer KCH,NHK,NHKD
      real ELM
C
C.. Intrinsic Functions ..
      intrinsic COS
C
C.. Common Blocks ..
      ...
      common /COMSOL/ SPGM,KER,IBT,BF4L,B4TROP
      ...
      NHKD = 0
      KCH = IPCO(NOUV)
      do while (KCH .gt. 0)
        ELM = COS(ATG(KCH)) / ZIMP(KCH)
        PIVO = PIVO - ELM
C       --- ----------------------------------------------------------- ---
C     PRISE EN COMPTE DU TERME DU A LA 1/2 SUSCEPTANCE DANS LES CALCULS
C     EN REACTIF
C       --- ----------------------------------------------------------- ---
        if (IBT .eq. 2) then
          PIVO = PIVO + 2*SUSL(KCH)
        end if
        NHK = NRLX(KCH)
C       --- ----------------------------------------------------------- ---
C                             SOMMET DEJA VU
C       --- ----------------------------------------------------------- ---
        if (NHK .eq. NHKD) then
          EHA(IX) = EHA(IX) + ELM
        else
          IX = IX + 1
          EHA(IX) = ELM
          NCHA(IX) = NHK
          NHKD = NHK
        endif
        KCH = ICCO(KCH)
      end do
      end
```

FIG. C.3: Documentation et restructuration de CAPIV par *FOREST*

Annexe D

L'outil *Refine/Fortran*

Refine/Fortran est un outil d'aide à la rétro-ingénierie/ré-ingénierie de la société Reasoning Systems [1]. Outil de génération d'outils, il effectue une *analyse syntaxique* du code Fortran77 et génère une base de données orientée-objets. Un langage spécialisé et une bibliothèque de fonctions s'appliquant aux objets de la base de données permettent à l'utilisateur de coder sa propre analyse ou transformation du code. Par ailleurs, des fonctions permettant d'afficher des graphes (avec calcul automatique de la disposition des nœuds), d'indenter le code source et de réaliser des interfaces graphiques avec menus et liens hyper-texte, sont proposées.

Quelques fonctionnalités de base sont fournies sans nécessiter une programmation par l'utilisateur, en fait ce sont les références croisées du même genre que celles fournies par plusieurs outils :

- les indications de non conformité avec des guides de bonne programmation,
- le graphe des appels,
- les paramètres et variables de `COMMON` *modifiés*, ainsi que ceux *utilisés*, par chaque unité de programme, en prenant en compte les instructions `EQUIVALENCE` et les *alias* dus à l'instanciation des paramètres formels par des paramètres effectifs, mais ces derniers seulement à travers un niveau d'appel et non récursivement.

Il n'y a pas de traitement spécifique des éléments de tableaux.

1. Refine/Fortran est une marque déposée de Reasoning Systems, Inc., 3260 Hillview Avenue, Palo Alto, CA 94304 *tél* (415) 494-6201 *fax* (415) 494-8053 *e-mail* info-requests@reasoning.com *www* http://www.reasoning.com

Annexe E

Le sous-programme Fortran PREFIX

```
C       --- ------------------------------------------------------------ ---
C       ---              PREPARATION DE LA PARTIE FIXE                   ---
C       --- ------------------------------------------------------------ ---

C     KER=11: DEPASSEMENT DU NOMBRE MAX DE SOMMETS
C     KER=12: DEPASSEMENT DU NOMBRE MAX DE LIAISONS
C     KER=13: DEPASSEMENT DU NOMBRE MAX DE GROUPES
C     KER=14: DEPASSEMENT DU NOMBRE MAX DE CONSOMMATIONS
C     KER=15: DEPASSEMENT DU NOMBRE MAX DE TENSIONS
C     KER=16: DEPASSEMENT DU NOMBRE MAX DE SAXBY

        SUBROUTINE PREFIX (CARL, NA1, NA7, NDTM, NDOB, INTM)

        IMPLICIT LOGICAL *1 (B)
       character*40 sccs_id/'@(#)prefix.f 1.1 94/05/13\n'/

C       en sortie :
C       CARL   CARACTERISTIQUES DES LIAISONS
C       NA1    IPCO,IPCE,ICCO,ICCE,NRLO,NRLE,NRGR,NRCO,NRTJ,
C       NDOB
C       NA7    KZBY,NZBY ; LOCALISATION DES MESURES SAXBY
C       NDTM   TABLE DES INJECTIONS ENTIEREMENT DEFINIES
C       INTM   INDICATION POUR RETABLIR L'OBSERVABILITE

        integer*4 NTAZ
        INTEGER*2  NMP,NMS,NML,NMG,NMC,NMT,
     &            NMXJ,NBY1,IRDO,IRIN,IRDH,IRDG,
     &            NTF,NBTMX,NBSX,
     &            KCMM,NMVG,NNLX,NELX
C       NMS  =  NOMBRE MAX DE SOMMETS
C       NML  =  NOMBRE MAX DE LIAISONS
C       NMG  =  NOMBRE MAX DE GROUPES
C       NMC  =  NOMBRE MAX DE CONSOMMATIONS
C       NMT  =  NOMBRE MAX DE CAPTEURS DE TENSION
C       NBY1 =  NOMBRE MAX DE TM SAXBY
C       NTF  =  NOMBRE MAX DE T.M. FAUSSES
C       NMXJ =  NOMBRE DE TRONCONS DE JEU DE BARRE
C       NMP  =  NOMBRE MAX DE POSTES
C       NNLX  =  NOMBRE MAX D'ELEMENT DU JACOBIEN DANS FARA
C       NELX  =  NOMBRE MAX D'ELEMENT DU JACOBIEN DANS LAMBDA
C       NBTMX =  NOMBRE MAX DE TELEMESURE
C       NBSX = NOMBRE MAX D'ENTREE DU FICHIER DE LOCALISATION DES SAXBY (2*NBY1)
C       IRDO =  10 X REDONDANCE EN TELE INFORMATION
C       IRIN =  10 X NB MOYEN DE T.M. POUR UNE INJECTION
C       IRDH =  10 X NB MOYEN D'ELEMENT DE JACOBIEN PAR SOMMET
C       IRDG =  10 X NB MOYEN D'ELEMENT DE LA MATRICE DE GAIN PAR SOMMET
c       NTAZ =  45000    dimension de la table Z de FARA

        PARAMETER (NMP=500,  NMS=700, NML=1000, NMG=300, NMC=600, NMT=800)
        PARAMETER (NMXJ=1500, NBY1=50, IRDO=40, IRIN=15, IRDH=80, IRDG=108)
        PARAMETER (NTF=200,NBTMX=4200,NBSX=100,NTAZ=45000)
        PARAMETER (KCMM=30, NMVG=800,NNLX=(2*NML+NMS),NELX=(4*NML+NMS))

C       --- ------------------------------------------------------------ ---
C       CARL    ZIMP, ATG, SUSO, SUSE
C       Nx1     IPCO, IPCE, ICCO, ICCE, NRLO, NRLE, NRGR, NRCO, NRTJ
C       NB1     NRLO, NRLE, NRTJ
C       Nx2     IPTAI, NTAI
C       Nx3     LHAC, NCHA, EHA
C       Nx4     IVOL, IVEL, IVGR, IVCO, IVTM
C       Nx5     IVAC, LITA, ITMF, NXUT
C       Nx6     MVOL, MVEL, MVGR, MVCO, MVTV
C       M_GAIN  LCHBG, LORD, IDEB, NONU, ISUI, INDL, ELM
C       NA7     KZBY, NZBY
C       --- ------------------------------------------------------------ ---

        Integer*4       I_ATG, I_SUSO, I_SUSE
        Integer*4       I_IPCO, I_IPCE, I_ICCO, I_ICCE
        Integer*4       I_NRLO, I_NRLE, I_NRGR, I_NRCO, I_NRTJ
```

```
      Integer*4       IB_NRLO, IB_NRLE, IB_NRTJ
      Integer*4       I_IPTAI, I_NTAI
      Integer*4       I_LHAC, I_NCHA, I_EHA
      Integer*4       I_IVOL, I_IVEL, I_IVGR, I_IVCO, I_IVTM
      Integer*4       I_LCHBG, I_LORD, I_IDEB, I_NONU, I_ISUI
      Integer*4       I_INDL
      Integer*4       I_ELM
      Integer*4       I_IVAC, I_LITA, I_ITMF, I_NXUT, I_ITMF_P
      Integer*4       I_NXUT_P
      Integer*4       I_KZBY, I_NZBY
      Integer*4       I_TENS

      Common/I_CARL/  I_ATG, I_SUSO, I_SUSE
      Common/I_ESWE/  I_IPCO, I_IPCE, I_ICCO, I_ICCE,
     +                I_NRLO, I_NRLE, I_NRGR, I_NRCO, I_NRTJ
      Common/I_ESWC/  IB_NRLO, IB_NRLE, IB_NRTJ
      Common/I_ESWI/  I_IPTAI, I_NTAI
      Common/I_ESWA/  I_LHAC, I_NCHA, I_EHA
      Common/I_PTMA/  I_IVOL, I_IVEL, I_IVGR, I_IVCO, I_IVTM
      Common/I_GAIN/  I_LCHBG, I_LORD, I_IDEB, I_NONU, I_ISUI,
     +                I_INDL, I_ELM
      Common/I_ESWL/  I_IVAC, I_LITA, I_ITMF, I_NXUT, I_ITMF_P,
     +                I_NXUT_P
      Common/I_LRD/   I_KZBY, I_NZBY
      Common/I_STN/   I_TENS

      COMMON/COMSOL/    SPGM(2),KER,IBT,BF4L,B4TROP
      COMMON/DBUG/      BS,B1,B2,B3,B4,B5,B6,B7,B8,B9,B10,
     +                  B11,B12,B13,B14,B15
      COMMON/DMFX/      NXL,NL,NXP,NXJ,NT,NC,NG,NBY,NS,JOUR(5)
      COMMON/IPAC/      SIGA,SIGR,NSO,NTMA,NTMR,LBLC(7,2),NFA,NBFA,NFR,
     +                  NBFR,NXU,BROB
      COMMON/NUFIC/     KFB,KFR,KCR,KS
      COMMON/PAREG/     ZMAXI,ZMAXF,PRECA,PRECR,SEUA,SEUR,CAET,CRET,UCAL,
     +                  UDEP,UMIN,UMAX,CX,EPSDT,ZTMERO,FOURCH,
     +                  SURCOA,SURCOR,ZCONT,PONMIN,PONMAX,PIVNUL,
     +                  PONDNC,PONDAJ,DIVERM
      COMMON/TCAL/      VCAL,VCA2
      COMMON/TMPR/  DSOM(2,NMS)

      DIMENSION CARL(*), NA1(*), NA7(*), NDTM(*), NDOB(*), INTM(*)

C       --- ---------------------------------------------------------- ---
C       ---                    Debut de PREFIX                         ---
C       --- ---------------------------------------------------------- ---

      CALL MOVEK (SPGM, '*PREFIX*', 1, 1, 8)

      KER = 0
      IF (NS.GT.NMS) GOTO 4
      IF (NL.GT.NML) GOTO 5
      IF (NG.GT.NMG) GOTO 6
      IF (NC.GT.NMC) GOTO 7
      IF (NT.GT.NMT) GOTO 8
      IF (NBY.GT.NBY1) GOTO 10

      LON = 4*NS
      CALL NLIRE (DSOM, 'ASDSOM', LON, KER)
      IF (KER.NE.0) RETURN
```

```
C       --- ---------------------------------------------------------- ---
C       --- LECTURE DES PARAMETRES DE REGLAGE ET INITIALISATION EVENTUELLE-
C       --- ---------------------------------------------------------- ---

      CALL NLIRE (ZMAXI, 'ASESWZ', 50, KER)
c      IF (KER.NE.0) RETURN

      IF ((ZMAXI.GT.0.AND.ZMAXI.LT.100)
     +    .AND.(UDEP.GT.150.AND.UDEP.LT.300)
     +    .AND.(UMIN.GT.100.AND.UMIN.LT.350) .AND. (UMIN.LT.UMAX)
     +    .AND.(UMAX.GT.100.AND.UMAX.LT.350.)
     +    .AND.(UCAL.GT.150.AND.UCAL.LT.300.)
     +    .AND.(PONMIN.GE.1.AND.PONMAX.LE.32000.)
     +    .AND. (PONMIN.LE.PONMAX))
     +   GOTO 3

      IF (BS) WRITE(108,1006)
      ZMAXI  =  20.
      ZMAXF  =   1.
      PRECA  =  10.
      PRECR  =  10.
      SEUA   =   5.
      SEUR   =   5.
      CAET   =    .8
      CRET   =  80.
      UCAL   = 225.
      UDEP   = 235.
      UMIN   = 150.
      UMAX   = 280.
      CX     =   1.2
      EPSDT  =    .01
      ZTMERO =   5.
      FOURCH =   3.
      SURCOA =   1.
      SURCOR =   1.
      ZCONT  =  10.
      PONMIN =   1.
      PONMAX = 225.
      PIVNUL =    .00001
      PONDNC =   1.
      PONDAJ = 225.
      DIVERM =   3.

      CALL NECRIR (ZMAXI, 'ASESWZ', 50, KER)
      IF (KER.NE.0) RETURN

    3 VCAL = UCAL
      VCA2 = VCAL*VCAL

C       --- ---------------------------------------------------------- ---
C       ---     LECTURE DONNEES FIXES DANS LA BASE DU SYSDIC           ---
C       --- ---------------------------------------------------------- ---

      LON = 8*NL
      CALL NLIRE (CARL, 'ASLCAR', LON, KER)
      IF (KER.NE.0) RETURN

      LON = 2*NL + NG + NC + NT
      CALL NLIRE (NA1(I_NRLO), 'ASNROU', LON, KER)
      IF (KER.NE.0) RETURN

      LON = 2*NBY
      CALL NLIRE (NA7, 'ASNRSX', LON, KER)
      IF (KER.NE.0) RETURN

      LON = NL + 2*NS
      CALL NLIRE (INTM, 'ASINTM', LON, KER)
      IF (KER.NE.0) RETURN
```

```
      CALL NLIRE (NDTM, 'ASNDTM', NS, KER)
      IF (KER.NE.0) RETURN

C        --- ---------------------------------------------------------- ---
C        --- PRISE EN COMPTE DES RESULTATS DE L'OBSERVABILITE (CONNEXITE) --
C        --- ---------------------------------------------------------- ---

      CALL NLIRE (NDOB, 'ASNDOB', NS, KER)
      IF (KER.NE.0) RETURN

      do I=1,NS
         IF (NDOB(I).NE.0) NDOB(I) = 3
      end do

      CALL PRXPLO (NA1(I_IPCO), NA1(I_IPCE), NA1(I_ICCO), NA1(I_ICCE),
     +             NDOB,
     +             NA1(I_NRLO), NA1(I_NRLE), NA1(I_NRGR), NA1(I_NRCO),
     +             NA1(I_NRTJ), NDTM)
      RETURN

C        --- ---------------------------------------------------------- ---
C        ---     TRAITEMENT DES ERREURS SUR LES DIMENSIONS MAX          ---
C        --- ---------------------------------------------------------- ---

    4 IF (B9) WRITE(KS,1000) NS, NMS
      KER = 11
      RETURN

    5 IF (B9) WRITE(KS,1001) NL, NML
      KER = 12
      RETURN

    6 IF (B9) WRITE(KS,1002) NG, NMG
      KER=13
      RETURN

    7 IF (B9) WRITE(KS,1003) NC, NMC
      KER = 14
      RETURN

    8 IF (B9) WRITE(KS,1004) NT, NMT
      KER = 15
      RETURN

   10 IF (B9) WRITE(KS,1005) NBY, NBY1
      KER = 16
      RETURN

 1000 FORMAT (3X,'NOMBRE DE SOMMETS ',I3,' SUPERIEUR AU MAX: ',I3)
 1001 FORMAT (3X,'NOMBRE DE LIGNES ',I3,' SUPERIEUR AU MAX: ',I3)
 1002 FORMAT (3X,'NOMBRE DE GROUPES ',I3,' SUPERIEUR AU MAX: ',I3)
 1003 FORMAT (3X,'NOMBRE DE CONSOMMATIONS ',I3,' SUPERIEUR AU MAX: ',I3)
 1004 FORMAT (3X,'NOMBRE DE TENSIONS ',I3,' SUPERIEUR AU MAX: ',I3)
 1005 FORMAT (3X,'NOMBRE DE SAXBY ',I3,' SUPERIEUR AU MAX: ',I3)
 1006 FORMAT (3X,'ATTENTION PARAMETRES DE REGLAGE REINITIALISES')
      END
```

Annexe F

Le sous-programme Fortran PRXPLO

```
C        --- ------------------------------------------------------------ ---
C        ---     CONSTRUCTION DU VECTEUR D'EXPLORATION NODALE            ---
C        ---     OU DESCRIPTION DES SUCCESSEURS DES SOMMETS              ---
C        --- ------------------------------------------------------------ ---

      SUBROUTINE PRXPLO (IPCO, IPCE, ICCO, ICCE, NDOB,
     1                   NRLO, NRLE, NRGR, NRCO, NRTJ, NDTM)

      IMPLICIT LOGICAL *1 (B)
       character*40 sccs_id/'@(#)prxplo.f 1.1 94/05/13\n'/

        integer*4 NTAZ
        INTEGER*2  NMP,NMS,NML,NMG,NMC,NMT,
     &             NMXJ,NBY1,IRDO,IRIN,IRDH,IRDG,
     &             NTF,NBTMX,NBSX,
     &             KCMM,NMVG,NNLX,NELX
C       NMS  =  NOMBRE MAX DE SOMMETS
C       NML  =  NOMBRE MAX DE LIAISONS
C       NMG  =  NOMBRE MAX DE GROUPES
C       NMC  =  NOMBRE MAX DE CONSOMMATIONS
C       NMT  =  NOMBRE MAX DE CAPTEURS DE TENSION
C       NBY1 =  NOMBRE MAX DE TM SAXBY
C       NTF  =  NOMBRE MAX DE T.M. FAUSSES
C       NMXJ =  NOMBRE DE TRONCONS DE JEU DE BARRE
C       NMP  =  NOMBRE MAX DE POSTES
C       NNLX  =  NOMBRE MAX D'ELEMENT DU JACOBIEN DANS FARA
C       NELX  =  NOMBRE MAX D'ELEMENT DU JACOBIEN DANS LAMBDA
C       NBTMX =  NOMBRE MAX DE TELEMESURE
C       NBSX = NOMBRE MAX D'ENTREE DU FICHIER DE LOCALISATION DES SAXBY (2*NBY1)
C       IRDO =  10 X REDONDANCE EN TELE INFORMATION
C       IRIN =  10 X NB MOYEN DE T.M. POUR UNE INJECTION
C       IRDH =  10 X NB MOYEN D'ELEMENT DE JACOBIEN PAR SOMMET
C       IRDG =  10 X NB MOYEN D'ELEMENT DE LA MATRICE DE GAIN PAR SOMMET
c       NTAZ =  45000    dimension de la table Z de FARA

        PARAMETER (NMP=500,  NMS=700, NML=1000, NMG=300, NMC=600, NMT=800)
        PARAMETER (NMXJ=1500, NBY1=50, IRDO=40, IRIN=15, IRDH=80, IRDG=108)
        PARAMETER (NTF=200,NBTMX=4200,NBSX=100,NTAZ=45000)
        PARAMETER (KCMM=30, NMVG=800,NNLX=(2*NML+NMS),NELX=(4*NML+NMS))

C     IPCO   POINTEUR CHAINAGE DES LIGNES
C     IPCE   POINTEUR CHAINAGE DES LIGNES
C     ICCE   CHAINAGE DES LIGNES
C     ICCO   CHAINAGE DES LIGNES

C     NDOB(I)=     RANG D UN ST OBS DS LISTE ENSEMBLE DES STS
C                  -1 SI NON OBS

C     NRLO   RACCORDEMENT DES LIGNES A L ORIGINE
C     NRLE   RACCORDEMENT DES LIGNES A L EXTREMITE
C     NRGR   RACCORDEMENT DES GROUPES
C     NRCO   RACCORDEMENT DES CONSOMMATIONS
C     NRTJ   RACCORDEMENT DES TENSIONS
C            RACC= NUM ST OBS
C                  O SI NON RACCORDE
C                  -1 SI RACCORDE A ST NON OBS

C     NDTM   TABLE DES NOEUDS ENTIEREMENT DEFINIS

      COMMON/COMSOL/SPGM(2),KER,IBT,BF4L,B4TROP
      COMMON/DBUG/BS,B1,B2,B3,B4,B5,B6,B7,B8,B9,B10,B11,B12,B13,B14,B15
      COMMON/DMFX/NXL,NL,NXP,NXJ,NT,NC,NG,NBY,NS,JOUR(5)
      COMMON/IPAC/SIGA,SIGR,NSO,NTMA,NTMR,LBLC(7,2),NFA,NBFA,NFR,NBFR ,NXU,BROB
      COMMON/NUFIC/KFB,KFR,KCR,KS
      COMMON/TMPR/DSOM(2,NMS)

      DIMENSION NRLO(*), NRLE(*), NRGR(*), NRCO(*), NRTJ(*)
      DIMENSION IPCO(*), IPCE(*), ICCO(*), ICCE(*), NDOB(*), NDTM(*)
```

```
C       --- ----------------------------------------------------------- ---
C       ---                     Debut de PRXPLO                         ---
C       --- ----------------------------------------------------------- ---

      CALL MOVEK (SPGM, '*PRXPLO*', 1, 1, 8)

      NSO = NS

C       --- ----------------------------------------------------------- ---
C       ---  ON ELIMINE LES ELEMENTS N'APPARTENANT PAS A LA COMPOSANTE  ---
C       ---  CONNEXE PRINCIPALE (NDOB=0)                                 ---
C       --- ----------------------------------------------------------- ---

      DO 20 I = 1, NL
         K1 = NRLO(I)
         IF (K1.LE.0) GOTO 16
            IF (NDOB(K1).LE.0) NRLO(I) = -1
         GOTO 17
   16       NRLO(I) = 0
   17    K1 = NRLE(I)
         IF (K1.LE.0) GOTO 18
            IF (NDOB(K1).LE.0) NRLE(I) = -1
         GOTO 20
   18       NRLE(I) = 0
   20 CONTINUE

      DO 22 I = 1, NG
         K1 = NRGR(I)
         IF (K1.LE.0) GOTO 21
            IF (NDOB(K1).LE.0) NRGR(I) = -1
         GOTO 22
   21       NRGR(I) = 0
   22 CONTINUE

      DO 23 I = 1, NC
         K1 = NRCO(I)
         IF (K1.LE.0) GOTO 25
            IF (NDOB(K1).LE.0) NRCO(I) = -1
         GOTO 23
   25       NRCO(I) = 0
   23 CONTINUE

      DO 24 I = 1, NT
         K1 = NRTJ(I)
         IF (K1.LE.0) GOTO 26
            IF (NDOB(K1).LE.0) NRTJ(I) = -1
         GOTO 24
   26       NRTJ(I) = 0
   24 CONTINUE

C       --- ----------------------------------------------------------- ---
C       ---                     INITIALISATION                          ---
C       --- ----------------------------------------------------------- ---

      do I = 1, NS
         IPCO(I) = 0
         IPCE(I) = 0
      end do

C       --- ----------------------------------------------------------- ---
C       ---                 CONSTRUCTION DES CHAINAGES                  ---
C       --- ----------------------------------------------------------- ---

      do I=1,NL
         N1 = NRLO(I)
         N2 = NRLE(I)
         CALL PRCHEN (I, N1, N2, IPCO, ICCO, NRLE)
         CALL PRCHEN (I, N2, N1, IPCE, ICCE, NRLO)
      end do
```

```
C       --- ----------------------------------------------------------- ---
C       ---                    EDITIONS OPTIONNELLES                    ---
C       --- ----------------------------------------------------------- ---

      IF(.NOT.B1) GOTO 7
      WRITE (KS,1000)
      do I = 1, NS
         WRITE(KS,1001) I, (DSOM(J,I),J=1,2), IPCO(I), IPCE(I)
      end do
      WRITE (KS,1002)
      do I=1,NL
         WRITE(KS,1003) I, NRLO(I), NRLE(I), ICCO(I), ICCE(I)
      end do
      WRITE (KS,1004)
      do I = 1, NT
         WRITE(KS,1005) I, NRTJ(I)
      end do

    7 RETURN

 1000 FORMAT (///,'  CHAINAGE DES ARCS A LEURS EXTREMITES',//,7X,'POSTES',15X,'P+',10X,'P-',/)
 1001 FORMAT (1X,I5,1X,2A4,2(5X,I5))
 1002 FORMAT (/,2X,'LIGNES',5X,'NRLO',5X,'NRLE',5X,'ICCO',5X,'ICCE',/)
 1003 FORMAT (1X,5(I5,4X))
 1004 FORMAT (///,10X,'CAPTEURS DE TENSION',/,2X,'NUMERO',6X,'SOMMET',//)
 1005 FORMAT (1X,I6,6X,I6)
      END
```

Annexe G

Le sous-programme Fortran PRCHEN

```
C --- ------------------------------------------------------------
C --- INTRODUCTION D'UN ARC DANS UN CHAINAGE D'ARCS A LEURS ORIGINE -
C --- ------------------------------------------------------------

      SUBROUTINE PRCHEN (IL, N1, N2, IPCO, ICCO, NRLE)

      IMPLICIT LOGICAL*1  (B)
       character*40 sccs_id/'@(#)prchen.f 1.1 94/05/13\n'/

C     IL      NUMERO DE L ARC
C     N1      NUMERO DU SOMMET ORIGINE
C     N2      NUMERO DU SOMMET EXTREMITE
C     IPCO    TABLEAU DES POINTEURS VERS LES CHAINES D'ARCS DES SOMMETS
C     ICCO    POINTEUR VERS L'ARC SUIVANT DANS LA CHAINE
C     NRLE    TABLEAU DES EXTREMITES DES ARCS

      COMMON/COMSOL/SPGM(2),KER,IBT,BF4L,B4TROP
      COMMON/DBUG/BS,B1,B2,B3,B4,B5,B6,B7,B8,B9,B10,B11,B12,B13,B14,B15
      COMMON/NUFIC/KFB,KFR,KCR,KS

      DIMENSION IPCO(*), ICCO(*), NRLE(*)

C --- ------------------------------------------------------------
C ---Debut de PRCHEN ---
C --- ------------------------------------------------------------

      CALL MOVEK(SPGM,'*PRCHEN*',1,1,8)

C --- ------------------------------------------------------------
C ---ARC NE RACCORDANT PAS DEUX SOMMETS OBSERVABLES ---
C --- ------------------------------------------------------------

      KCH = IPCO(N1)
      IF ((N1.LE.0).OR.(N2.LE.0)) GOTO 1

C --- ------------------------------------------------------------
C ---INITIALISATION , ENTREE DANS LA CHAINE ---
C --- ------------------------------------------------------------

      KP = 0

C --- ------------------------------------------------------------
C ---CHAINE VIDE ---
C --- ------------------------------------------------------------

      IF (KCH.LE.0) GOTO 6

C --- ------------------------------------------------------------
C ---CLASSEMENT DES ARCS PAR LEUR SOMMETS EXTREMITES ---
C --- ------------------------------------------------------------

    4    IF (NRLE(KCH).GE.N2) GOTO 3
            KP  = KCH
            KCH = ICCO(KCH)
            IF (KCH.GT.0) GOTO 4
    3    IF (KP.LE.0) GOTO 6
            ICCO(KP) = IL
    7       ICCO(IL) = KCH
      GOTO 1

C --- ------------------------------------------------------------
C ---CAS D UN DEBUT DE CHAINE ---
C --- ------------------------------------------------------------

    6    IPCO(N1) = IL
      GOTO 7

    1 RETURN
      END
```

Annexe H

Le sous-programme Fortran CAPIV

```
C       --- ---------------------------------------------------------- ---
C       --- INTRODUCTION D'UN ELEMENT DE MATRICE J DU A UN ELEMENT D'INJECTION
C       --- ---------------------------------------------------------- ---

      SUBROUTINE CAPIV (IX,PIVO,NOUV,IPCO,ICCO,NRLX,NCHA,EHA,ZIMP,ATG,SUSL)

      IMPLICIT LOGICAL *1 (B)
       character*40 sccs_id/'@(#)capiv.f 1.1 94/05/13\n'/

C     IX      INDEX COURANT DE RANGEMENT DES ELEMENTS DE LA MATRICE
C     PIVO    PIVOT EN COURS DE CALCUL
C     NOUV    NUMERO DU SOMMET EN COURS D ANALYSE
C     IPCO    CHAINAGE DES ARCS
C     ICCO    CHAINAGE DES ARCS
C     NRLX    NOEUD EXTREMITE DES LIGNES
C     NCHA    JACOBIEN
C     EHA     JACOBIEN
C     ZIMP    IMPEDANCE DES LIGNES
C     ATG     ANGLE DE PERTE DES LIGNES
C     SUSL    SUSCEPTANCE DES LIGNES

      COMMON/COMSOL/SPGM(2),KER,IBT,BF4L,B4TROP
      COMMON/DBUG/BS,B1,B2,B3,B4,B5,B6,B7,B8,B9,B10,B11,B12,B13,B14,B15
      COMMON/NUFIC/KFB,KFR,KCR,KS

      DIMENSION IPCO(*),ICCO(*),NRLX(*)
      DIMENSION NCHA(*),EHA(*),ZIMP(*),ATG(*),SUSL(*)

C       --- ---------------------------------------------------------- ---
C       ---                    Debut de CAPIV                          ---
C       --- ---------------------------------------------------------- ---

      CALL MOVEK(SPGM,'*CAPIV *',1,1,8)

      NHKD = 0
      KCH  = IPCO(NOUV)

C       --- ---------------------------------------------------------- ---
C       ---     IL N Y A PLUS D'ARC AYANT SON ORIGINE EN NOUV          ---
C       --- ---------------------------------------------------------- ---

   30    IF (B3) WRITE(KS,1000) NOUV, KCH, IX, NHKD
         IF (KCH.LE.0) GOTO 1
         ELM  = COS(ATG(KCH))/ZIMP(KCH)
         PIVO = PIVO - ELM

C       --- ---------------------------------------------------------- ---
C     PRISE EN COMPTE DU TERME DU A LA 1/2 SUSCEPTAMCE DANS LES CALCULS
C     EN REACTIF
C       --- ---------------------------------------------------------- ---

         IF (IBT.EQ.2) PIVO = PIVO + 2*SUSL(KCH)
         NHK = NRLX(KCH)

C       --- ---------------------------------------------------------- ---
C       ---                    SOMMET DEJA VU                          ---
C       --- ---------------------------------------------------------- ---

         IF (NHK.EQ.NHKD) GOTO 5
            IX       = IX + 1
            EHA(IX)  = ELM
            NCHA(IX) = NHK
            NHKD     = NHK
         GOTO 4
    5       EHA(IX) = EHA(IX) + ELM
    4    KCH = ICCO(KCH)
      GOTO 30

    1 RETURN
 1000 FORMAT (1X,'NUM SOMM=',I5,5X,'KCH=',I5,5X,'IX=',I5,5X,'NHKD=',I5)
      END
```

Annexe I

Spécifications “standard” employées dans les autres spécifications

spec BASICS =
%%Spec. des Naturels, Positifs,...
...

---

spec ELEM =
%%Spec. de la sorte Elem, parametre de certaines specs generiques.
sort *Elem*;
end

---

spec PAIR[**sort** *S1*][**sort** *S2*] =
%%Spec. du produit cartesien des valeurs de deux sortes.
free type $Pair[S1, S2]$ ::= << _ _ >> ($1st\_of\_\_ : S1$; $2nd\_of\_\_ : S2$);
end

---

spec ARRAY[**sort** *Index*][ELEM] =
%%Spec. generique et abstraite d'un tableau Fortran (infini) d'une dimension.
generated type $Array[Index, Elem]$ ::=
new |
$write\_\_at\_\_in\_\_(Elem; Index; Array[Index, Elem])$;
op $read\_\_at\_\_ : Array[Index, Elem] * Index \to ?\ Elem$;
vars $i, i1 : Index;\ e : Elem;\ a : Array[Index, Elem]$;
axioms %%Op. read _ at _ definie uniquement pour les cases ecrites.
$\neg\ def\ read\ new\ at\ i$;
$def\ read(write\ e\ at\ i1\ in\ a) at\ i \Leftrightarrow ((i = i1) \vee\ def\ read\ a\ at\ i)$;
%%Defn. read _ at _.
$def\ read(write\ e\ at\ i1\ in\ a) at\ i \Rightarrow read(write\ e\ at\ i1\ in\ a) at\ i = e$
$when (i = i1)$
$else\ read\ a\ at\ i)$;
end

---

spec SET[ELEM] =
%%Spec. generique et abstraite des ensembles.
generated type $Set[Elem]$ ::= $empty \mid add\_\_to\_\_(Elem; Set[Elem])$
preds $\_\_is\_empty : Set[Elem]$;
$\_\_is\_in\_\_ : Elem * Set[Elem]$;
vars $e : Elem;\ s : Set[Elem]$;
axioms %%Defn. _ is_empty.
$s\ is\_empty \Leftrightarrow (s = empty)$;

%%Defn. _ is_in _
%%par induction structurelle.
$\neg e\ is\_in\ empty$;
$e1\ is\_in(add\ e2\ to\ s)$
$\Leftrightarrow ((e1 = e2) \vee e1\ is\_in\ s)$;

%%Deux ensembles aux memes membres sont egaux
$(\forall\ VarDecleElem \bullet (e\ is\_in\ s1 \Leftrightarrow e\ is\_in\ s2))$
$\Rightarrow s1 = s2$;
end

```
spec LIST[ELEM] =
%%Spec. generique et abstraite des listes.
   BASICS
then
free type List[Elem] ::= empty | cons(head_of__ : ?Elem; tail_of__ : ?List[Elem]);
preds __is_empty : List[Elem];
   __is_in__ : Elem * List[Elem];
op length_of__ : List[Elem] → Nat;
vars e : Elem; l : List[Elem];
axioms %%Defn. __ is_empty
   l is_empty ⇔ (l = empty);

   %%Defn. __ is_in __ par induction structurelle.
   e is_in l ⇔ ((e = head_of l) ∨ e is_in tail_of l);

   %%Defn. length_of __ par induction structurelle.
   length_of l =0
               when l is_empty
               else length_of(tail_of l) + 1;
end
```

```
spec LISTDIFFELEMS[ELEM] =
%%Spec. generique et abstraite des listes et listes sans doublons.
   LIST[ELEM]
then local
   pred    __is_distinct_elems : List[Elem];
   vars    l : List[Elem];
   axiom   l is_distinct_elems
           ⇔  (l is_empty ∨
                (¬ head_of l is_in tail_of l ∧  tail_of l is_distinct_elems));
within
sort %%La sorte des listes sans doublons
   %%i.e. chaque element n'a qu'une occurrence dans la liste.
   LDE[Elem] = { l : List[Elem] • l is_distinct_elems; }
end
```

```
spec TABLE[sort Key][sort Item] =
%%Spec. generique d'une table (infinie).
generated type Table[Key, Item] ::=
                          new |
                          add(Key; Item; Table[Key, Item]);
op retrieve : Key * Table[Key, Item] →? Item;
vars k, k1 : Key; i : Item; t : Table[Key, Item];
axioms %%Op. retrieve definie pour les clefs employees dans la construction de la table.
   ¬ def retrieve(k, new);
   def retrieve(k, add(k1, i, t)) ⇔ ((k = k1) ∨ def retrieve(k, t));
   %%Defn. retrieve.
   def retrieve(k, add(k1, i, t)) ⇒ retrieve(k, add(k1, i, t)) =i
                                                      when(k = k1)
                                                      else retrieve(k, t);
end
```

Annexe J

La spécification concrète

```
spec NumeroSommet = Basics
then
sort NumS = Pos;
  NumS_ou_neg = Int;
  S_Val_NRLx = { s : NumS_ou_neg • s ≥ -1; }
end
  ————————————————————————————————

spec NumeroLigne = Basics
then
sort NumL = Pos;
  NumL_ou_0 = Nat;
end
  ————————————————————————————————

spec Concrete =
%%Spec. concrete du sous-programme Fortran PRCHEN et d'une partie de PRXPLO
  %%Sorte Array[NumL,S_Val_NRLx].
  Array[NumeroLigne fit Index↦NumL]
      [NumeroSommet fit Elem↦S_Val_NRLx]
  and
  %%Sorte Array[NumS,NumL_ou_0].
  Array[NumeroSommet fit Index↦NumS]
      [NumeroLigne fit Elem↦NumL_ou_0]
  and
  %%Sorte Array[NumL,NumL_ou_0].
  Array[NumeroLigne fit Index↦NumL]
      [NumeroLigne fit Elem↦NumL_ou_0]
  and
  %%Sorte Pair[Array[NumS,NumL_ou_0],Array[NumL,NumL_ou_0]].
  Pair[sort Array[NumS, NumL_ou_0] fit S1↦Array[NumS, NumL_ou_0]]
      [sort Array[NumL, NumL_ou_0] fit S2↦Array[NumL, NumL_ou_0]]
  and
  %%Sorte Pair[NumL,Array[NumL,NumL_ou_0]].
  Pair[NumeroLigne fit S1↦NumL]
      [sort Array[NumL, NumL_ou_0] fit S2↦Array[NumL, NumL_ou_0]]
  and
  %%Sorte Pair[S_IPCx_ICCx,S_IPCx_ICCx].
  Pair[sort S_IPCx_ICCx fit S1↦S_IPCx_ICCx]
      [sort S_IPCx_ICCx fit S2↦S_IPCx_ICCx]
then
sorts %%La sorte des tableaux Fortran NRLO et NRLE.
  S_NRLx = Array[NumL, S_Val_NRLx];

  %%Les (sur-)sortes des tableaux Fortran IPCO et IPCE d'une part et ICCO
  %%et ICCE d'autre part: cette caracterisation ne tient pas compte du fait
  %%que les valeurs legitimes d'IPCO resp. IPCE dependent
  %%de la valeur d'ICCO resp. ICCE et vice-versa.
  S_IPCx = Array[NumS, NumL_ou_0];
  S_ICCx = Array[NumL, NumL_ou_0];


sorts %%S_IPCx_ICCx est la sorte des parametres de sortie de PRCHEN
  %%i.e: de la paire IPCO ou IPCE et ICCO ou ICCE.
  %%Cette sous-sorte de Pair[S_IPCx,S_ICCx] n'est pas le produit cartesien
  %%d'une sous-sorte de S_IPCx et d'une sous-sorte de S_ICCx
```

%%mais regroupe certains paires de valeurs de sortes S_IPCx et S_ICCx:
%%toutes celles pouvant etre interpretees comme une
%%table indexee par NumS, de listes de NumL sans doublons
%%et sans intersection i.e. chaque liste est distincte,
%%sans 'boucle' et a une fin (pour que PRCHEN s'arrete).
$S\_IPCx\_ICCx < Pair[S\_IPCx, S\_ICCx]$;

%%Sorte des listes non vides de NumL sans doublons stockees dans la table.
$S\_KP\_ICCx < Pair[NumL, S\_ICCx]$;

generated type S_IPCx_ICCx ::= $table\_vide$ | $PRCHEN$;

ops %%Les fragments de PRCHEN et de PRXPLO
%%Cree les 2 tables (lignes 100 a 119 de PRXPLO).
$construit\_tables\_cocycles : NumS\_ou\_0 * NumL\_ou\_0 * S\_NRLx * S\_NRLx \to? Pair[S\_IPCx\_ICCx, S\_IPCx\_ICCx]$;

%%Cree une table ("slice" des lignes 100 a 119 de PRXPLO).
$construit\_table\_cocycles : NumS\_ou\_0 * NumL\_ou\_0 * S\_NRLx * S\_NRLx \to? S\_IPCx\_ICCx$;

%%Initialise la table ("slice" des lignes 100 a 108 de PRXPLO).
$table\_vide : NumS\_ou\_0 \to S\_IPCx\_ICCx$;

%%Parcourt tous les numeros de ligne en appellant PRCHEN pour les
%%ajouter un a un a la table ("slice" des lignes 109 a 119 de PRXPLO).
$remplit\_table\_cocycles : NumL\_ou\_0 * S\_IPCx\_ICCx * S\_NRLx * S\_NRLx \to? S\_IPCx\_ICCx$;

%%Insere le 1er arg dans la liste indiquee par le 2eme de la table du 4eme
%%selon un ordre engendre par les 3eme et 5eme, si 2eme et 3eme > 0.
$PRCHEN : NumL * NumS\_ou\_0 * NumS\_ou\_0 * S\_IPCx\_ICCx * S\_NRLx \to? S\_IPCx\_ICCx$;

%%Insere le 1er arg dans la liste indiquee par le 2eme de la table du 4eme
%%selon un ordre engendre par les 3eme et 5eme (lignes 40 a 74 de
%%PRCHEN).
$insere : NumL * NumS * NumS * S\_IPCx\_ICCx * S\_NRLx \to? S\_IPCx\_ICCx$;

%%Parcourt la liste afin de chercher le bon point d'insertion (boucle de
%%PRCHEN).
$point\_inser : NumL * S\_KP\_ICCx * NumS * S\_NRLx \to? NumL$;

then local
preds %%Preds auxiliaires pour simplifier des axiomes.
$\_\_is\_empty\_at\_\_ : S\_IPCx\_ICCx * NumS$;
$\_\_has\_empty\_tail : S\_KP\_ICCx$;

ops %%Ops auxiliaires pour simplifier des axiomes.
$insert\_\_after\_\_in\_\_ : NumL * NumL * S\_IPCx\_ICCx \to? S\_IPCx\_ICCx$;
$insert\_\_front\_list\_at\_\_in\_\_ : NumL * NumS * S\_IPCx\_ICCx \to? S\_IPCx\_ICCx$;
$list\_at\_\_in\_\_ : NumS * S\_IPCx\_ICCx \to? S\_KP\_ICCx$;
$head\_of\_list\_at\_\_in\_\_ : NumS * S\_IPCx \to? NumL$;
$head\_of\_\_ : S\_KP\_ICCx \to NumL$;
$tail\_of\_\_ : S\_KP\_ICCx \to? S\_KP\_ICCx$;
$head\_of\_tail\_of\_\_ : S\_KP\_ICCx \to? NumL$;

vars $N1 : NumS;\ IL, KP : NumL;\ IPCx : S\_IPCx;\ ICCx : S\_ICCx;$
axioms %%Insere 1er arg apres 2eme dans une liste du 3eme.
$<< IPCx\ \ ICCx >>\ \in S\_IPCx\_ICCx$
$\Rightarrow insert\ IL\ after\ KP\ in\ << IPCx\ \ ICCx >>\ =$
$<< IPCx\ \ write\ IL\ at\ KP\ in(write(read\ ICCx\ at\ KP)at\ IL\ in\ ICCx) >>;$

%%Cons d'un numero de ligne et d'une liste de la table
$<< IPCx\ \ ICCx >>\ \in S\_IPCx\_ICCx$
$\Rightarrow insert\ IL\ front\_list\_at\ N1\ in\ << IPCx\ \ ICCx >>\ =$
$<< write\ IL\ at\ N1\ in\ IPCx\ \ write(read\ IPCx\ at\ N1)at\ IL\ in\ ICCx >>;$

%%Lecture dans la table d'une liste non vide.
$<< IPCx\ \ ICCx >>\ \in S\_IPCx\_ICCx$
$\Rightarrow list\_at\ N1\ in\ << IPCx\ \ ICCx >>\ =\ << (read\ IPCx\ at\ N1)as\ NumL\ \ ICCx >>;$

%%Lecture de la tête d'une liste non vide de la table.
$<< IPCx\ \ ICCx >>\ \in S\_IPCx\_ICCx$
$\Rightarrow head\_of\_list\_at\ N1\ in\ << IPCx\ \ ICCx >>\ = (read\ IPCx\ at\ N1)as\ NumL;$

%%Tete d'une liste non vide.
$<< KP\ \ ICCx >>\ \in S\_KP\_ICCx \Rightarrow head\_of\ << KP\ \ ICCx >>\ = KP;$

%%Queue non-vide d'une liste non-vide.
$<< KP\ \ ICCx >>\ \in S\_KP\_ICCx$
$\Rightarrow tail\_of << KP\ \ ICCx >>\ =\ << (read\ ICCx\ at\ KP)as\ NumL\ \ ICCx >>;$

%%Tete de la queue non vide d'une liste non vide.
$<< KP\ \ ICCx >>\ \in S\_KP\_ICCx$
$\Rightarrow head\_of\_tail\_of\ << KP\ \ ICCx >>\ = (read\ ICCx\ at\ KP)\ as\ NumL;$

%%Tests de liste vide.
$<< IPCx\ \ ICCx >>\ \in S\_IPCx\_ICCx$
$\Rightarrow << IPCx\ \ ICCx >>\ is\_empty\_at\ N1 \Leftrightarrow read\ IPCx\ at\ N1 \leq 0;$
$<< KP\ \ ICCx >>\ \in S\_KP\_ICCx$
$\Rightarrow << KP\ \ ICCx >>\ has\_empty\_tail \Leftrightarrow read\ ICCx\ at\ KP \leq 0;$

within
generated type $S\_KP\_ICCx\ ::=\ list\_at\_\_in\_\_;$

vars $NS0, N10 : NumS\_ou\_0;\ NL0, I : NumL\_ou\_0;\ N1 : NumS;\ IL : NumL;$
$IPCx\_ICCx : S\_IPCx\_ICCx;\ KP\_ICCx : S\_KP\_ICCx;$
$NRLO, NRLE, NRL1, NRL2, NRLx : S\_NRLx;$

axioms %%Defn. construit_tables_cocycles:
%%1er arg = nombre de sommets du reseau,
%%2me arg = nombre de lignes du reseau,
%%3me arg = origines des lignes et
%%4me arg = extremites des lignes.
$construit\_tables\_cocycles(NS0, NL0, NRLO, NRLE) =$
$=\ << construit\_table\_cocycles(NS0, NL0, NRLO, NRLE)$
$construit\_table\_cocycles(NS0, NL0, NRLE, NRLO) >>;$

%%Defn. construit_table_cocycles:
%%1er arg = nombre de sommets du reseau,
%%2me arg = nombre de lignes du reseau,
%%3me arg = l'une des extremites des lignes et
%%4me arg = l'autre extremite des lignes.
$construit\_table\_cocycles(NS0, NL0, NRL1, NRL2)$
$= remplit\_table\_cocycles(NL0, table\_vide(NS0), NRL1, NRL2);$

```
%%Defn. table_vide.
table_vide(NS0) =
    << new  new >>
    when (NS0 = 0)
    else << write 0 at(NS0 as NumS)in 1st_of(table_vide(NS0 − 1))  2nd_of(table_vide(NS0 − 1)) >>;

%%Defn. remplit_table_cocycles:
%%1er arg = dernier numero de ligne a inserer ou 0,
%%2me arg = table d'insertion,
%%3me arg = l'une des extremites des lignes et
%%4me arg = l'autre extremite des lignes.
remplit_table_cocycles(I, IPCx_ICCx, NRL1, NRL2) =
    IPCx_ICCx
    when I ≤ 0
    else PRCHEN(I as NumL,
                read NRL1 at (I as NumL),
                read NRL2 at (I as NumL),
                remplit_table_cocycles(I − 1, IPCx_ICCx, NRL1, NRL2),
                NRL2);

%%Defn. PRCHEN:
%%1er arg = numero de ligne a inserer,
%%2me arg = une extremite de cette ligne (numero sommet ou 0),
%%3me arg = l'autre extremite de cette ligne (numero sommet ou 0),
%%4me arg = table d'insertion et
%%5me arg = l'autre extremite pour toutes les lignes.
def PRCHEN(IL, N10, N20, IPCx_ICCx, NRLx)
⇒ PRCHEN(IL, N10, N20, IPCx_ICCx, NRLx) =
    IPCx_ICCx
    when ((N10 ≤ 0) ∨ (N20 ≤ 0))
    else insere(IL, N10 as NumS, N20 as NumS, IPCx_ICCx, NRLx);

%%Defn. insere:
%%1er arg = numero de ligne a inserer,
%%2me arg = une extremite de cette ligne (numero sommet),
%%3me arg = l'autre extremite de cette ligne (numero sommet),
%%4me arg = table d'insertion et
%%5me arg = l'autre extremite pour toutes les lignes.
def insere(IL, N1, N2, IPCx_ICCx, NRLx)
⇒ insere(IL, N1, N2, IPCx_ICCx, NRLx) =
    insert IL front_list_at N1 in IPCx_ICCx
    when ( IPCx_ICCx is_empty_at N1 ∨
        N2 ≤ read NRLx at (head_of_list_at N1 in IPCx_ICCx))
    else insert IL after point_inser(IL, list_at N1 in IPCx_ICCx, N2, NRLx) in IPCx_ICCx;

%%Defn. point_inser.
%%1er arg = numero de ligne a inserer,
%%2me arg = liste non-vide d'insertion,
%%3me arg = extremite qui decide ordre, de la ligne et
%%3me arg = extremite qui decide ordre, de toutes les lignes.
def point_inser(IL, KP_ICCx, N2, NRLx)
⇒ point_inser(IL, KP_ICCx, N2, NRLx) =
    head_of KP_ICCx
    when ( KP_ICCx has_empty_tail ∨
        N2 ≤ read NRLx at head_of_tail_of KP_ICCx)
    else point_inser(IL, tail_of KP_ICCx, N2, NRLx);
```

```
%%Domaines.
def PRCHEN(IL, N10, N20, IPCx_ICCx, NRLx)
⇔ (N10 ≤ 0 ∨
    N2 ≤ 0 ∨
    def insere(IL, N10 as NumS, N20 as NumS, IPCx_ICCx, NRLx);

def insere(IL, N1, N2, IPCx_ICCx, NRLx)
if def (read 1st_of(IPCx_ICCx) at N1) ∧
   ( IPCx_ICCx is_empty_at N1 ∨
     ( def(read NRLx at (head_of_list_at N1 in IPCx_ICCx)) ∧
       def(insert IL after point_inser(IL, list_at N1 in IPCx_ICCx, N2, NRLx) in IPCx_ICCx)));

def point_inser(IL, KP_ICCx, N2, NRLx)
if KP_ICCx has_empty_tail ∨
    ( def(read NRLx at head_of_tail_of KP_ICCx) ∧
      def point_inser(IL, tail_of KP_ICCx, N2, NRLx) );

hide construit_tables_cocycles,
  construit_table_cocycles,
  table_vide,
  remplit_table_cocycles,
  insere,
  point_inser
end
```

Annexe K

La spécification Abstract1

spec SOMMETABSTRAIT =
%%Spec. declarative des sommets du reseau electrique.
sorts *Sommet*;
IdSommet;
AttribS1;
%%AttribS2;
%%...
ops %%L'identificateur du sommet.
id_de__ : *Sommet* → *IdSommet*;
%%Un autre attribut quelconque.
attribS1_de__ : *Sommet* → *AttribS1*;
%%attribS2_de __: Sommet → AttribS2;
%%...
vars *s1*, *s2* : *Sommet*;
axioms %%Chaque sommet a un id unique a lui.
(*id_de s1* = *id_de s2*) ⇒ (*s1* = *s2*);
end

---

spec LIGNEABSTRAITE =
%%Spec. declarative des lignes electriques.
%%Sorte Sommet.
SOMMETABSTRAIT
then
sorts *Ligne*;
IdLigne;
AttribL1;
%%AttribL2;
%%...
preds %%Le fait d'etre connectee ou non a l'origine resp. extremite.
__est_connectee_orig : *Ligne*;
__est_connectee_extrem : *Ligne*;
ops %%L'identificateur de la ligne.
id_de__ : *Ligne* → *IdLigne*;
%%Un autre attribut quelconque.
attribL1_de__ : *Ligne* → *AttribL1*;
%%attribL2_de __: Ligne → AttribL2;
%%...
%%Le sommet connecte a l'origine, si est_connectee_orig.
origine_de__ : *Ligne* →? *Sommet*;
%%Le sommet connecte a l'extremite, si est_connectee_extrem.
extremite_de__ : *Ligne* →? *Sommet*;
vars *l*, *l1*, *l2* : *Ligne*;
axioms %%Ops origine_de et extremite_de definies pour les lignes connectees.
def origine_de l ⇔ *l est_connectee_orig*;
def extremite_de l ⇔ *l est_connectee_extrem*;

%%Les lignes ne sont pas des boucles.
¬(*l est_connectee_orig* ∧ (*origine_de l* = *extremite_de l*));

```
    %%Entre deux lignes du meme id,
    %%les seules proprietes qui peuvent varier sont
    %%la connexion ou non a l'origine et a l'extremite.
    (id_de l1 = id_de l2 ∧
     l1 est_connectee_orig ⇔ l2 est_connectee_orig ∧
     l1 est_connectee_extrem ⇔ l2 est_connectee_extrem )
    ⇒ l1 = l2;
end
```

```
spec ReseauAbstrait =
%%Spec. declarative d'un modele simple des reseaux electriques.
  %%Sorte Set[Sommet] et pred is_in.
  Set[SommetAbstrait fit Elem↦Sommet]
  and
  %%Sortes Ligne et Set[Ligne],
  %%preds est_connectee_orig, est_connectee_extrem et is_in et
  %%ops id_de, origine_de et extremite_de.
  Set[LigneAbstraite fit Elem↦Ligne]
then
sort Reseau;
op %%Les sommets et lignes appartenant au reseau.
  sommets_de__ : Reseau → Set[Sommet];
  lignes_de__ : Reseau → Set[Ligne];
vars l, l1, l2 : Ligne; r : Reseau;
axioms %%Un reseau ne contient jamais deux lignes du meme id.
  l1 is_in lignes_de r ∧ l2 is_in lignes_de r ∧ id_de l1 = id_de l2
  ⇒ l1 = l2;

  %%Chaque bout de chaque ligne du reseau est soit non-connecte,
  %%soit connecte a un sommet du reseau.
  (l is_in lignes_de r ∧ l est_connectee_orig)
  ⇒ origine_de l is_in sommets_de r;
  (l is_in lignes_de r ∧ l est_connectee_extrem)
  ⇒ extremite_de l is_in sommets_de r;

  %%Deux lignes peuvent etre connectees en parallele
  %%dans un reseau, mais pas dans des sens differents.
  (l is_in lignes_de r ∧ l est_connectee_orig ∧ l est_connectee_extrem)
  ⇒ ¬ ∃ l1 : Ligne •
      (l1 is_in lignes_de r ∧
       origine_de l = extremite_de l1 ∧
       extremite_de l = origine_de l1);
end
```

spec ENSEMBLESLIGNESSORTANTETENTRANT =
%%Spec. declarative des ensembles
%%de lignes sortant resp. entrant pour un sommet de reseau.
  %%Sortes Sommet, Ligne, Set[Ligne] et Reseau,
  %%preds est_connectee_orig, est_connectee_extrem et is_in et
  %%ops origine_de, extremite_de, sommets_de et lignes_de.
  RESEAUABSTRAIT
then
ops *lignes_sortant* : *Sommet* ∗ *Reseau* →? *Set*[*Ligne*];
  lignes_entrant : *Sommet* ∗ *Reseau* →? *Set*[*Ligne*];
vars *s* : *Sommet*; *l* : *Ligne*; *r* : *Reseau*;
axioms *def lignes_sortant*(*s*, *r*) ⇔ *s is_in sommets_de r*;
  %%Les lignes sortant
  %%d'un sommet sont connectees a ce sommet a l'origine
  %%mais aussi a un autre sommet quelconque a l'extremite.
  l is_in lignes_sortant(*s*, *r*)
  ⇔ *l is_in lignes_de r* ∧ *l est_connectee_extrem* ∧ *origine_de l* = *s*;

  def lignes_entrant(*s*, *r*) ⇔ *s is_in sommets_de r*;
  %%Les lignes entrant
  %%dans un sommet sont connectees a ce sommet a l'extremite
  %%mais aussi a un autre sommet quelconque a l'origine.
  l is_in lignes_entrant(*s*, *r*)
  ⇔ *l is_in lignes_de r* ∧ *l est_connectee_orig* ∧ *extremite_de l* = *s*;
end

---

spec ABSTRACT1 =
%%Spec. declarative d'un modele simple du reseau electrique et
%%des ensembles de lignes sortant resp. entrant
%%pour un sommet de reseau.
ENSEMBLESLIGNESSORTANTETENTRANT
hide *empty* : *Set*[*Sommet*],
  add__to__ : *Sommet* ∗ *Set*[*Sommet*] → *Set*[*Sommet*],
  __is_empty : *Set*[*Sommet*],
  __is_in__ : *Sommet* ∗ *Set*[*Sommet*]
  empty : *Set*[*Ligne*],
  add__to__ : *Ligne* ∗ *Set*[*Ligne*] → *Set*[*Ligne*],
  __is_empty : *Set*[*Ligne*],
  __is_in__ : *Ligne* ∗ *Set*[*Ligne*]

Annexe L

La spécification Abstract2

spec IMPLSOMMETSDERESEAU =
%%Spec. d'une implementation des sommets d'un reseau
%%a l'aide de tableaux.
%%Sortes NumS, NumS_ou_0, IdSommet, AttribS1,
%%Array[NumS,IdSommet] et Array[NumS,AttribS1] et op read __ at __.
ARRAY[NUMEROSOMMET **fit** $Index \mapsto NumS$][**sort** *IdSommet* **fit** $Elem \mapsto IdSommet$]
and
ARRAY[NUMEROSOMMET **fit** $Index \mapsto NumS$][**sort** *AttribS1* **fit** $Elem \mapsto AttribS1$]
then
sort *Reseau*;
ops %%Le nombre de sommets dans le reseau.
$\_\_.NS : Reseau \rightarrow NumS\_ou\_0;$
%%Stockage des attributs des sommets.
$\_\_.IdSommets : Reseau \rightarrow Array[NumS, IdSommet];$
$\_\_.AttS1 : Reseau \rightarrow Array[NumS, AttribS1];$
%%L'identificateur du sommet.
$id : NumS * Reseau \rightarrow ?\ IdSommet;$
%%Un autre attribut quelconque.
$attribS1 : NumS * Reseau \rightarrow ?\ AttribS1;$
vars $ns, ns1, ns2 : NumS;\ r, r1, r2 : Reseau;$
axioms %%Les tableaux stockant les attributs des sommets
%%sont effectivement dimensionnes par __.NS.
$def\ (read\ r.IdSommets\ at\ ns)\ if\ ns \leq r.NS;$
$def\ (read\ r.AttS1\ at\ ns)\ if\ ns \leq r.NS;$

%%Chaque sommet (tous reseaux confondus) a un id unique a lui.
$(ns1 \leq r1.NS \wedge ns2 \leq r2.NS \wedge$
$read\ r1.IdSommets\ at\ ns1 = read\ r2.IdSommets\ at\ ns2)$
$\Rightarrow read\ r1.AttribS1\ at\ ns1 = read\ r2.AttribS1\ at\ ns2;$

%%Dans un meme reseau, chaque numero designe un sommet different.
$ns1 \leq r.NS \wedge ns2 \leq r.NS \wedge read\ r.IdSommets\ at\ ns1 = read\ r.IdSommets\ at\ ns2$
$\Rightarrow (ns1 = ns2);$

%%Lecture des attributs des sommets.
$def\ id(ns, r) \Leftrightarrow ns \leq r.NS;$
$def\ id(ns, r) \Rightarrow (id(ns, r) = read\ r.IdSommets\ at\ ns);$
$def\ attribS1(ns, r) \Leftrightarrow ns \leq r.NS;$
$def\ attribS1(ns, r) \Rightarrow (attribS1(ns, r) = read\ r.AttribS1\ at\ ns);$
end

---

spec IMPLLIGNESDERESEAU =
%%Spec. d'une implementation des sommets et lignes d'un reseau
%%a l'aide de tableaux.
%%Sortes NumS, NumS_ou_0 et Reseau et op __.NS.
IMPLSOMMETSDERESEAU
and
%%Sortes NumL, NumL_ou_0, Array[NumL,IdLigne], Array[NumL,NumS_ou_0]
%%et Array[NumL,AttribL1] et ops read __ at __.
ARRAY[NUMEROLIGNE **fit** $Index \mapsto NumL$][**sort** *IdLigne* **fit** $Elem \mapsto IdLigne$]
and
ARRAY[NUMEROLIGNE **fit** $Index \mapsto NumL$][NUMEROSOMMET **fit** $Elem \mapsto NumS\_ou\_0$]
and
ARRAY[NUMEROLIGNE **fit** $Index \mapsto NumL$][**sort** *AttribL1* **fit** $Elem \mapsto AttribL1$]
then

preds %%Le fait d'etre connectee ou non a l'origine resp. extremite.
$est\_connectee\_orig : NumL * Reseau;$
$est\_connectee\_extrem : NumL * Reseau;$
ops %%Le nombre de lignes dans le reseau.
$\_\_.NL : Reseau \rightarrow NumL\_ou\_0;$
%%Stockage des attributs des lignes.
$\_\_.IdLignes : Reseau \rightarrow Array[NumL, IdLigne];$
$\_\_.AttL1 : Reseau \rightarrow Array[NumL, AttribL1];$
$\_\_.NRLO : Reseau \rightarrow Array[NumL, NumS\_ou\_0];$
$\_\_.NRLE : Reseau \rightarrow Array[NumL, NumS\_ou\_0];$
%%L'identificateur de la ligne.
$id : NumL * Reseau \rightarrow ?\ IdLigne;$
%%Un autre attribut quelconque.
$attribL1 : NumL * Reseau \rightarrow ?\ AttribL1;$
%%Le numero du sommet connecte a l'origine, si est_connectee_orig.
$num\_orig : NumL * Reseau \rightarrow ?\ NumS;$
%%Le numero du sommet connecte a l'extremite, si est_connectee_extrem.
$num\_extrem : NumL * Reseau \rightarrow ?\ NumS;$
vars $nl, nl1, nl2 : NumL;\ r, r1, r2 : Reseau;$
axioms %%Chaque bout de chaque ligne du reseau est soit non-connecte,
%%soit connecte a un sommet du reseau et
%%les tableaux stockant les attributs des lignes
%%sont effectivement dimensionnes par __.NL.
$def\ (read\ r.IdLignes\ at\ nl)\ if\ nl \leq r.NL;$
$def\ (read\ r.AttL1\ at\ nl)\ if\ nl \leq r.NL;$
$(read\ r.NRLO\ at\ nl \leq r.NS)\ if\ nl \leq r.NL;$
$(read\ r.NRLE\ at\ nl \leq r.NS)\ if\ nl \leq r.NL;$

%%Les lignes ne sont pas des boucles et
%%deux lignes peuvent etre connectees en parallele
%%dans un reseau, mais pas avec des orientations differentes.
$read\ r.NRLO\ at\ nl > 0 \wedge read\ r.NRLE\ at\ nl > 0$
$\Rightarrow \neg\ \exists\ nl1 : NumL \bullet (\ nl1 \leq r.NL \wedge$
$read\ r.NRLO\ at\ nl = read\ r.NRLE\ at\ nl1 \wedge$
$read\ r.NRLE\ at\ nl = read\ r.NRLO\ at\ nl1\ );$

%%Entre deux lignes du meme id (tous reseaux confondus),
%%les seules proprietes qui peuvent varier sont
%%la connexion ou non a l'origine et a l'extremite.
$(nl1 \leq r1.NL \wedge nl2 \leq r2.NL \wedge$
$read\ r1.IdLignes\ at\ nl1 = read\ r2.IdLignes\ at\ nl2)$
$\Rightarrow read\ r1.AttribL1\ at\ nl1 = read\ r2.AttribL1\ at\ nl2;$

$(nl1 \leq r1.NL \wedge nl2 \leq r2.NL \wedge$
$read\ r1.IdLignes\ at\ nl1 = read\ r2.IdLignes\ at\ nl2 \wedge$
$read\ r1.NRLO\ at\ nl1 > 0 \wedge read\ r2.NRLO\ at\ nl2 > 0)$
$\Rightarrow read\ r1.NRLO\ at\ nl1 = read\ r2.NRLO\ at\ nl2;$

$(nl1 \leq r1.NL \wedge nl2 \leq r2.NL \wedge$
$read\ r1.IdLignes\ at\ nl1 = read\ r2.IdLignes\ at\ nl2 \wedge$
$read\ r1.NRLE\ at\ nl1 > 0 \wedge read\ r2.NRLE\ at\ nl2 > 0)$
$\Rightarrow read\ r1.NRLE\ at\ nl1 = read\ r2.NRLE\ at\ nl2;$

%%Un reseau ne contient jamais deux lignes du meme id
%%(et donc chaque numero designe une ligne differente).
$nl1 \leq r.NL \wedge nl2 \leq r.NL \wedge read\ r.IdLignes\ at\ nl1 = read\ r.IdLignes\ at\ nl2$
$\Rightarrow nl1 = nl2;$

%%Lecture des attributs des lignes.
$est\_connectee\_orig(nl, r) \Leftrightarrow (nl \leq r.NL \wedge read\ r.NRLO\ at\ nl > 0);$
$est\_connectee\_extrem(nl, r) \Leftrightarrow (nl \leq r.NL \wedge read\ r.NRLE\ at\ nl > 0);$
$def\ id(nl, r) \Leftrightarrow nl \leq r.NL;$
$def\ id(nl, r) \Rightarrow (id(nl, r) = read\ r.IdLignes\ at\ nl);$
$def\ attribL1(nl, r) \Leftrightarrow nl \leq r.NL;$
$def\ attribL1(nl, r) \Rightarrow (attribL1(nl, r) = read\ r.AttribL1\ at\ nl);$
$def\ num\_orig(nl, r) \Leftrightarrow est\_connectee\_orig(nl, r);$
$def\ num\_orig(nl, r) \Rightarrow (num\_orig(nl, r) = read\ r.NRLO\ at\ nl);$
$def\ num\_extrem(nl, r) \Leftrightarrow est\_connectee\_extrem(nl, r);$
$def\ num\_extrem(nl, r) \Rightarrow (num\_extrem(nl, r) = read\ r.NRLE\ at\ nl);$
end

---

spec IMPLRESEAU = IMPLLIGNESDERESEAU
%%Spec. d'une implementation des sommets et lignes d'un reseau
%%a l'aide de tableaux.

---

spec ENSEMBLESNUMSLIGNESSORTANTETENTRANT =
%%Spec. declarative des ensembles des numeros
%%des lignes sortant resp. entrant pour un numero de sommet de reseau.
%%Sortes NumS, NumL, Set[NumL] et Reseau, pred is_in
%%et ops __.NL, __.NRLO, __.NRLE et read __ at __.
SET[IMPLRESEAU **fit** $Elem \mapsto NumL$]
then
ops $nums\_lignes\_sortant : NumS * Reseau \to ?\ Set[NumL];$
$nums\_lignes\_entrant : NumS * Reseau \to ?\ Set[NumL];$
vars $ns : NumS;\ nl : NumL;\ r : Reseau;$
axioms %%Defn. nums_lignes_sortant.
$def\ nums\_lignes\_sortant(ns, r) \Leftrightarrow (ns \leq r.NS);$
%%Les lignes sortant
%%d'un sommet sont connectees a ce sommet a l'origine
%%mais aussi a un autre sommet quelconque a l'extremite.
$nl\ is\_in\ nums\_lignes\_sortant(ns, r)$
$\Leftrightarrow\ nl \leq r.NL \wedge read\ r.NRLE\ at\ nl > 0 \wedge read\ r.NRLO\ at\ nl = ns;$

%%Defn. nums_lignes_entrant.
$def\ nums\_lignes\_entrant(ns, r) \Leftrightarrow (ns \leq r.NS);$
%%Les lignes entrant
%%dans un sommet sont connectees a ce sommet a l'extremite
%%mais aussi a un autre sommet quelconque a origine.
$nl\ is\_in\ nums\_lignes\_entrant(ns, r)$
$\Leftrightarrow\ nl \leq r.NL \wedge read\ r.NRLO\ at\ nl > 0 \wedge read\ r.NRLE\ at\ nl = ns;$
end

---

spec ABSTRACT2 =
%%Spec. d'une implementation des sommets et lignes d'un reseau a
%%l'aide de tableaux et spec. declarative des ensembles de numeros
%%des lignes sortant resp. entrant pour un numero de sommet de reseau.
ENSEMBLESNUMSLIGNESSORTANTETENTRANT
hide $empty : Set[NumL],$
$add\_\_to\_\_ : NumL * Set[NumL] \to Set[NumL],$
$\_\_is\_empty : Set[NumL],$
$\_\_is\_in\_\_ : NumL * Set[NumL]$

Annexe M

La spécification Abstract3

spec LISTESNUMSLIGNESSORTANTETENTRANT =
%%Spec. des elements des listes sans doublons des numeros des
%%lignes sortant resp. entrant pour un numero de sommet de reseau.
%%Sortes NumS, NumL, Reseau et LDE[NumL], pred is_in
%%et ops __.NS, __.NL, __.NRLO, __.NRLE et read __ at __.
LISTDIFFELEMS[IMPLRESEAU **fit** $Elem \mapsto NumL$]
then
ops $nums\_lignes\_sortant : NumS * Reseau \rightarrow? LDE[NumL]$;
$nums\_lignes\_entrant : NumS * Reseau \rightarrow? LDE[NumL]$;
vars $ns : NumS$; $nl : NumL$; $r : Reseau$;
axioms $def\ nums\_lignes\_sortant(ns, r) \Leftrightarrow (ns \leq r.NS)$;
%%Elements de la liste des numeros de lignes sortant.
$nl\ is\_in\ nums\_lignes\_sortant(ns, r)$
$\Leftrightarrow\ nl \leq r.NL \wedge read\ r.NRLE\ at\ nl > 0 \wedge read\ r.NRLO\ at\ nl = ns$;
$def\ nums\_lignes\_entrant(ns, r) \Leftrightarrow (ns \leq r.NS)$;
%%Elements de la liste des numeros de lignes entrant.
$nl\ is\_in\ nums\_lignes\_entrant(ns, r)$
$\Leftrightarrow\ nl \leq r.NL \wedge read\ r.NRLO\ at\ nl > 0 \wedge read\ r.NRLE\ at\ nl = ns$;
end

---

spec ABSTRACT3 =
%%Spec. d'une implementation des sommets et lignes d'un reseau
%%a l'aide de tableaux et spec. declarative des elements des
%%listes sans doublons de numeros des lignes sortant resp. entrant
%%pour un numero de sommet de reseau.
LISTESNUMSLIGNESSORTANTETENTRANT
hide $List[NumL]$

Annexe N

La spécification Abstract4

spec ElemWithParametrisedOrder =
%%Spec. d'un ordre strict partiel parametre par une variable.
sorts *Elem*;
  Parameter;
pred $\_\_is\_less\_than\_\_in\_\_ : Elem * Elem * Parameter$;
vars $a, b, c : Elem;\ p : Parameter$;
axioms %%Antisymetrique.
  $a\ is\_less\_than\ b\ in\ p \Rightarrow \neg\ b\ is\_less\_than\ a\ in\ p$;
  %%Antireflexif.
  $\neg\ a\ is\_less\_than\ a\ in\ p$;
  %%Transitif.
  $a\ is\_less\_than\ b\ in\ p \wedge b\ is\_less\_than\ c\ in\ p \Rightarrow a\ is\_less\_than\ c\ in\ p$;
end

---

spec ListDiffElemsWithParametrisedOrder[ElemWithParametrisedOrder] =
%%Spec. generique et declarative des listes sans doublons
%%ordonnees par un ordre parametre par une variable
%%et d'une operation d'insertion respectant cet ordre.
  %%Sortes Elem, Parameter et LDE[Elem], preds is_empty, is_in
  %%et __ is_less_than __ in __ et ops head_of et tail_of.
  ListDiffElems[ElemWithParametrisedOrder]
then
pred $\_\_is\_ordered\_in\_\_ : LDE[Elem] * Parameter$;
op $insert : Elem * LDE[Elem] * Parameter \rightarrow ?\ LDE[Elem]$;
vars $e : Elem;\ l : LDE[Elem];\ p : Parameter$;
axiom $l\ is\_ordered\_in\ p$
  $\Leftrightarrow\ (l\ is\_empty \vee$
    $(tail\_of\ l)\ is\_empty \vee$
    $(\quad head\_of\ l\ is\_less\_than(head\_of\ tail\_of\ l) in\ p \wedge$
      $tail\_of\ l\ is\_ordered\_in\ p))$;

  $def\ insert(e, l, p) \Leftrightarrow$
    $\neg(e\ is\_in\ l) \wedge$
    $l\ is\_ordered\_in\ p \wedge$
    $\forall\ m : Elem \bullet m\ is\_in\ l \Rightarrow (e\ is\_less\_than\ m\ in\ p \vee m\ is\_less\_than\ e\ in\ p)$;
  $def\ insert(e, l, p) \Rightarrow\ e\ is\_in\ insert(e, l, p)$;
  $def\ insert(e, l, p) \Rightarrow\ insert(e, l, p)\ is\_ordered\_in\ p$;
end

spec RESEAUAVECORDRES =
%%Spec. imprecise de deux ordres partiels sur les numeros de ligne
%%parametres par le reseau.
%%Sortes NumL et Reseau et ops __.NRLO, __.NRLE et read __ at __.
IMPLRESEAU
then
pred $\_\_ <_e \_\_dans\_\_ : NumL * NumL * Reseau$
$\_\_ <_o \_\_dans\_\_ : NumL * NumL * Reseau$
vars $nl1, nl2, nl3 : NumL;\ r : Reseau;$
axioms %%La relation d'ordre __ $<_e$ __ dans __ relie les numeros de toutes
%%les lignes connectees a l'extremite et appartenant au meme reseau.
$(\neg(nl1 = nl2) \wedge$
$read\ r.NRLE\ at\ nl1 > 0 \wedge$
$read\ r.NRLE\ at\ nl2 > 0)$
$\Rightarrow (nl1\ <_e\ nl2\ dans\ r \vee nl2\ <_e\ nl1\ dans\ r);$

%%Ranges dans l'ordre __ $<_e$ __ dans __,
%%les numeros des lignes connectees au meme sommet extremite
%%doivent etre contigus.
$(nl1\ <_e\ nl2\ dans\ r \wedge$
$nl2\ <_e\ nl3\ dans\ r \wedge$
$read\ r.NRLE\ at\ nl1 > 0 \wedge$
$read\ r.NRLE\ at\ nl2 > 0 \wedge$
$read\ r.NRLE\ at\ nl3 > 0 \wedge$
$read\ r.NRLE\ at\ nl1 = read\ r.NRLE\ at\ nl3)$
$\Rightarrow\ read\ r.NRLE\ at\ nl1 = read\ r.NRLE\ at\ nl2;$

%%La relation d'ordre __ $<_o$ __ dans __ relie les numeros de toutes
%%les lignes connectees a l'origine et appartenant au meme reseau.
$(\neg(nl1 = nl2) \wedge$
$read\ r.NRLO\ at\ nl1 > 0 \wedge$
$read\ r.NRLO\ at\ nl2 > 0)$
$\Rightarrow (nl1\ <_o\ nl2\ dans\ r \vee nl2\ <_o\ nl1\ dans\ r);$

%%Ranges dans l'ordre __ $<_o$ __ dans __,
%%les numeros des lignes connectees au meme sommet origine
%%doivent etre contigus.
$(nl1\ <_o\ nl2\ dans\ r \wedge$
$nl2\ <_o\ nl3\ dans\ r \wedge$
$read\ r.NRLO\ at\ nl1 > 0 \wedge$
$read\ r.NRLO\ at\ nl2 > 0 \wedge$
$read\ r.NRLO\ at\ nl3 > 0 \wedge$
$read\ r.NRLO\ at\ nl1 = read\ r.NRLO\ at\ nl3)$
$\Rightarrow\ read\ r.NRLO\ at\ nl1 = read\ r.NRLO\ at\ nl2;$
end

spec ListesNumsLignesSortantEtEntrantAvecOrdres =
%%Spec. des elements des listes sans doublons des numeros des
%%lignes sortant resp. entrant pour un numero de sommet de reseau.
%%et spec. imprecise de leurs ordres.
  %%Sortes NumS, NumL, Reseau et LDE[NumL], preds is_in et est_ord_extrem_dans
  %%et ops __.NS, __.NL, __.NRLO, __.NRLE et read __ at __.
  ListDiffElemsWithParametrisedOrder[ReseauAvecOrdres
    fit *Elem*$\mapsto$*NumL*,
      Parameter$\mapsto$*Reseau*,
      $\_\_is\_less\_than\_\_in\_\_ \mapsto \_\_ <_e \_\_dans\_\_$]
  with $\_\_is\_ordered\_in\_\_ \mapsto \_\_est\_ord\_extrem\_dans\_\_$,
    insert$\mapsto$*insert_ord_extrem*
  and
  %%Pred est_ord_orig_dans.
  ListDiffElemsWithParametrisedOrder[ReseauAvecOrdres
    fit *Elem*$\mapsto$*NumL*,
      Parameter$\mapsto$*Reseau*,
      $\_\_is\_less\_than\_\_in\_\_ \mapsto \_\_ <_o \_\_dans\_\_$]
  with $\_\_is\_ordered\_in\_\_ \mapsto \_\_est\_ord\_orig\_dans\_\_$,
    insert$\mapsto$*insert_ord_orig*
then
ops $nums\_lignes\_sortant : NumS * Reseau \to ?\ LDE[NumL]$;
  $nums\_lignes\_entrant : NumS * Reseau \to ?\ LDE[NumL]$;
vars $ns : NumS;\ r : Reseau$;
axioms $def\ nums\_lignes\_sortant(ns, r) \Leftrightarrow (ns \leq r.NS)$;
  %%Elements de la liste des numeros de lignes sortant.
  $nl\ is\_in\ nums\_lignes\_sortant(ns, r)$
  $\Leftrightarrow\ nl \leq r.NL \wedge read\ r.NRLE\ at\ nl > 0 \wedge read\ r.NRLO\ at\ nl = ns$;
  %%Ordonnancement de la liste des numeros de lignes sortant.
  $def\ nums\_lignes\_sortant(ns, r) \Rightarrow nums\_lignes\_sortant(ns, r)\ est\_ord\_extrem\_dans\ r$;

  $def\ nums\_lignes\_entrant(ns, r) \Leftrightarrow (ns \leq r.NS)$;
  %%Elements de la liste des numeros de lignes entrant.
  $nl\ is\_in\ nums\_lignes\_entrant(ns, r)$
  $\Leftrightarrow\ nl \leq r.NL \wedge read\ r.NRLO\ at\ nl > 0 \wedge read\ r.NRLE\ at\ nl = ns$;
  %%Ordonnancement de la liste des numeros de lignes entrant.
  $def\ nums\_lignes\_entrant(ns, r) \Rightarrow nums\_lignes\_entrant(ns, r)\ est\_ord\_orig\_dans\ r$;
end

---

spec Abstract4 =
%%Spec. d'une implementation des sommets et lignes d'un reseau
%%a l'aide de tableaux et spec. declarative et imprecise des
%%listes sans doublons de numeros des lignes sortant resp. entrant
%%pour un numero de sommet de reseau.
ListesNumsLignesSortantEtEntrantAvecOrdres
hide *List*[*NumL*],
  $\_\_ <_e \_\_dans\_\_$,
  $\_\_ <_o \_\_dans\_\_$,
  $\_\_est\_ord\_extrem\_dans\_\_$,
  $\_\_est\_ord\_orig\_dans\_\_$,
  insert_ord_extrem,
  insert_ord_orig

Annexe O

La spécification Abstract5

spec RESEAUAVECORDRESIMPLEMENTES =
%%Spec. en terme d'ops de base sur les tableaux de l'implementation
%%d'ordres sur les numeros de ligne, parametres par des tableaux.
%%Sortes NumL et Array[NumL,NumS_ou_0] et op read __ at __.
IMPLRESEAU
then
pred $\_\_ <_a \_\_dans\_\_ : NumL * NumL * Array[NumL, NumS\_ou\_0]$
vars $nl1, nl2 : NumL$; $r : Reseau$; $a : Array[NumL, NumS\_ou\_0]$;
axioms %%Defn. $\_\_ <_a \_\_$ dans __.
$nl1 \;<_a\; nl2 \; dans \; t$
$\Leftrightarrow read \; a \; at \; nl1 > 0 \wedge$
$read \; a \; at \; nl2 > 0 \wedge$
$(read \; a \; at \; nl1 < read \; a \; at \; nl2 \vee (\; (read \; a \; at \; nl1 = read \; a \; at \; nl2) \wedge (nl1 > nl2) \;) \;);$
end

---

spec LISTESNUMSLIGNESSORTANTETENTRANTAVECORDRESIMPLEMENTES =
%%Spec. des listes sans doublons des numeros des
%%lignes sortant resp. entrant pour un numero de sommet de reseau,
%%ordonnees par un ordre implemente de facon calculatoire.
%%Sortes NumS, NumL, Reseau et LDE[NumL], preds is_in et est_ord_dans
%%et ops __.NS, __.NL, __.NRLO, __.NRLE et read __ at __.
LISTDIFFELEMSWITHPARAMETRISEDORDER[RESEAUAVECORDRESIMPLEMENTES
fit $Elem \mapsto NumL$,
$Parameter \mapsto Reseau$,
$\_\_is\_less\_than\_\_in\_\_ \mapsto \_\_ <_a \_\_dans\_\_$]
with $\_\_is\_ordered\_in\_\_ \mapsto \_\_est\_ord\_dans\_\_$
then
ops $nums\_lignes\_sortant : NumS * Reseau \to ? \; LDE[NumL]$;
$nums\_lignes\_entrant : NumS * Reseau \to ? \; LDE[NumL]$;
vars $ns : NumS$; $r : Reseau$;
axioms $def \; nums\_lignes\_sortant(ns, r) \Leftrightarrow (ns \leq r.NS)$;
%%Elements de la liste des numeros de lignes sortant.
$nl \; is\_in \; nums\_lignes\_sortant(ns, r) \Leftrightarrow (nl \leq r.NL \wedge read \; r.NRLE \; at \; nl > 0 \wedge read \; r.NRLO \; at \; nl = ns)$;
%%Ordonnancement de la liste des numeros de lignes sortant.
$ns \leq r.NS \Rightarrow nums\_lignes\_sortant(ns, r) \; est\_ord\_dans \; r.NRLE$;

$def \; nums\_lignes\_entrant(ns, r) \Leftrightarrow (ns \leq r.NS)$;
%%Elements de la liste des numeros de lignes entrant.
$nl \; is\_in \; nums\_lignes\_entrant(ns, r) \Leftrightarrow (nl \leq r.NL \wedge read \; r.NRLO \; at \; nl > 0 \wedge read \; r.NRLE \; at \; nl = ns)$;
%%Ordonnancement de la liste des numeros de lignes entrant.
$ns \leq r.NS \Rightarrow nums\_lignes\_entrant(ns, r) \; est\_ord\_dans \; r.NRLO$;
end

---

spec ABSTRACT5 =
%%Spec. d'une implementation a l'aide de tableaux des sommets et lignes
%%d'un reseau et d'ordres sur les numeros de ligne et spec. declarative
%%des listes sans doublons ordonnees des numeros des lignes sortant resp.
%%entrant pour un numero de sommet de reseau.
LISTESNUMSLIGNESSORTANTETENTRANTAVECORDRESIMPLEMENTES
hide $List[NumL]$,
$\_\_ <_a \_\_dans\_\_$,
$\_\_est\_ord\_dans\_\_$,
$insert$
end

Annexe P

La spécification Abstract6

```
spec TableNumsLigne =
%%Spec. d'une table indexee par les numeros de sommet et stockant
%%des listes sans doublons de numeros de ligne.
  Table[ImplReseau fit Key↦NumS]
       [ListDiffElemsWithParametrisedOrder[ReseauAvecOrdresImplementes
                                           fit Elem↦NumL,
                                               Parameter↦Reseau,
                                               __is_less_than__in__↦__ <_a __dans__]
        fit Item↦LDE[NumL]]
  with __is_ordered_in__↦__est_ord_dans__
  ——————————————————————————————————————

spec ConstrucTableNumsLignesSortantOuEntrant =
%%Spec. operationnelle de la construction d'une table stockant
%%les numeros de ligne sortant ou entrant pour chaque numero sommet.
  %%Sortes NumS, NumS_ou_0, NumL, NumL_ou_0, Array[NumL,NumS_ou_0],
  %%Table[NumS,LDE[NumL]] et ops read__at__, insert, new, add, retrieve.
  TableNumsLigne
then
op construit_table_cocycles : NumS_ou_0 ∗ NumL_ou_0∗
                              Array[NumL, NumS_ou_0] ∗ Array[NumL, NumS_ou_0]
                              →? Table[NumS, LDE[NumL]];
then local
  ops     empty__upto__ : Table[NumS, LDE[NumL]] ∗ NumS_ou_0
                        → Table[NumS, LDE[NumL]];
          insertTab : NumL ∗ Table[NumS, LDE[NumL]]∗
                      NumS ∗ Array[NumL, NumS_ou_0]
                      →? Table[NumS, LDE[NumL]];
          remplit_table_cocycles : NumL_ou_0 ∗ Table[NumS, LDE[NumL]]∗
                                   Array[NumL, NumS_ou_0] ∗ Array[NumL, NumS_ou_0]
                                   →? Table[NumS, LDE[NumL]];
  vars    ns0 : NumS_ou_0; ns : NumS; nl0 : NumL_ou_0; nl : NumL;
          a1, a2 : Array[NumL, NumS_ou_0]; t : Table[NumS, LDE[NumL]];
  axioms  %%Defn. empty __ upto __.
          empty t upto ns0 =
             t
             when ns0 ≤ 0
             else add(ns0 as NumS, empty, empty t upto(ns0 − 1));

          %%Defn. insertTab:
          %%Ajoute numero par composition retrieve,insert et add.
          insertTab(nl, t, ns, a) = add(ns, insert(nl, retrieve(ns, t), a2), t);
          %%remplit_table_cocycles ajoute les numeros de ligne un a un, du 1er au dernier.
          def remplit_table_cocycles(nl0, t, a1, a2)
          ⇔ ∀ nl : NumL • (nl ≤ nl0 ⇒ read a1 at nl ≤ ns0) ∧ def(read a2 at nl));
          def remplit_table_cocycles(nl0, t, a1, a2) ⇒ remplit_table_cocycles(nl0, t, a1, a2) =
             t
             when (nl0 ≤ 0 ∨
                   read a1 at(nl0 as NumL) ≤ 0 ∨
                   read a2 at(nl0 as NumL) ≤ 0)
             else
             insertTab(nl0 as NumL, remplit_table_cocycles(nl0 − 1, t, a1, a2), read a1 at nl as NumS, a2);
within
vars ns0 : NumS_ou_0; nl0 : NumL_ou_0; a1, a2 : Array[NumL, NumS_ou_0];
axiom construit_table_cocycles(ns0, nl0, a1, a2) = remplit_table_cocycles(nl0, empty new upto ns0, a1, a2);
end
```

spec ABSTRACT6 =
%%Spec. d'une implementation a l'aide de tableaux des sommets et lignes
%%d'un reseau et d'ordres sur les numeros de ligne et spec. declarative
%%de la construction des tables stockant les listes sans doublons ordonnees
%%des numeros des lignes sortant/entrant pour chaque numero de sommet
%%d'un reseau.
CONSTRUCTABLENUMSLIGNESSORTANTOUENTRANT
hide $List[NumL]$,
  $\_\_ <_a \_\_dans\_\_$,
  $\_\_est\_ord\_dans\_\_$,
  $insert$,
  new,
  add

Annexe Q

La spécification Abstract7

```
spec OpTablePartitioningLists[sort Key;][Elem] =
%%Spec. generique operationnelle des tables stockant un ensemble
%%sans intersection de listes sans doublons.
  %%Sorte LDE[Elem] et pred is_in.
  ListDiffElems[Elem]
then
generated type TabPartList[Key, Elem] ::=
                    new |
                    empty__at__(TabPartList[Key, Elem]; Key) |
                    consTab(Elem; Key; TabPartList[Key, Elem])?;
op insert__after__in__ : Elem * Elem * TabPartList[Key, Elem]
                     →? TabPartList[Key, Elem];
  retrieve : Key * TabPartList[Key, Elem] →? LDE[Elem];
then local
  pred    __is_valid_key_in__ : Key * TabPartList[Key, Elem];
          __is_in__ : Elem * TabPartList[Key, Elem];
  vars    k, k1 : Key; e : Elem; t : TabPartList[Key, Elem];
  axioms  %%Les clefs valides sont
          %%celles employees dans la construction de la table.
          ¬(k is_valid_key_in new);
          k is_valid_key_in(empty t at k1) ⇔ ((k = k1) ∨ k is_valid_key_in t);
          k is_valid_key_in consTab(e, k1, t)
          ⇔ (def consTab(e, k1, t) ∧ k is_valid_key_in t);

          %%Teste la presence de l'element dans une des listes de la table.
          e is_in t ⇔ ∃ k : Key • e is_in(retrieve(k, t));
within
vars e, e1, e2 : Elem; k, k1, k2 : Key; t : TabPartList[Key, Elem]; l : LDE[Elem];
axioms %%Domaine retrieve.
  def retrieve(k, t) ⇔ k is_valid_key_in t;

  %%Domaine constructeur consTab.
  def consTab(e, k, t) ⇔ (k is_valid_key_in t ∧ ¬(e is_in t));

  %%Constructeur derive insert _ after __ in _.
  def(insert e2 after e1 in t) ⇔ (e1 is_in t ∧ ¬(e2 is_in t));
  def(insert e2 after e1 in(empty t at k))
      ⇒ insert e2 after e1 in(empty t at k) = empty(insert e2 after e1 in t)at k ;
  def(insert e2 after e1 in consTab(e, k, t))
      ⇒ insert e2 after e1 in consTab(e, k, t)
          =   consTab(e, k, consTab(e2, k, t))
              when (e = e1)
              else consTab(e, k, insert e2 after e1 in t);

  %%Defn. retrieve en terme constructeurs.
  def retrieve(k, empty t at k1) ⇒ retrieve(k, empty t at k1) =
      empty
      when(k = k1)
      else retrieve(k, t);
  def retrieve(k, consTab(e, k1, t)) ⇒ retrieve(k, consTab(e, k1, t)) =
      cons(e, retrieve(k, t)) as LDE[NumL]
      when(k = k1)
      else retrieve(k, t);
end
```

```
spec TablePartionneuseNumsLigne =
%%Spec. d'une table indexee par les numeros de sommet et stockant un
%%ensemble sans intersection de listes sans doublons de numeros de ligne.
   OpTablePartitioningLists[ImplReseau fit Key↦NumS]
       [ImplReseau fit Elem↦NumL]
   ————————————————————————————————

spec InsereNumLigne =
%%Spec. en terme preds et ops de base sur tables, listes, tableaux, positifs
%%d'un algorithme pour l'insertion d'un numero de ligne dans la table.
   %%Sortes NumL, Array[NumL,NumS_ou_0], LDE[NumL], TabPartList[NumS,NumL],
   %%preds is_empty, is_in et ops read__at__, head_of, tail_of, retrieve, insert__after__in__.
   TablePartionneuseNumsLigne
then
op insertTab : NumL * TabPartList[NumS, NumL]*
               NumS * Array[NumL, NumS_ou_0]
               →? TabPartList[NumS, NumL];
then local
   op       point_inser : NumL * LDE[NumL] * Array[NumL, NumS_ou_0] →? NumL;
   vars     nl : NumL; a : Array[NumL, NumS_ou_0]; l : LDE[NumL];
   axioms   %%Domaine point_inser.
            def point_inser(nl, l, a)
            ⇔ ¬ (l is_empty) ∧
                ∀ n : NumL • ((n is_in l ⇒ (n < nl)) ∧ ((n ≤ nl) ⇒ def read a at n));
            %%Parcourt la liste en cherchant le bon point d'insertion.
            def point_inser(nl, l, a) ⇒ point_inser(nl, l, a)
            =   head_of l
                when((tail_of l)is_empty ∨ (read a at nl ≤ read a at head_of(tail_of l)))
                else point_inser(nl, (tail_of l)as LDE[NumL], a);
within
vars nl : NumL; a1, a2 : Array[NumL, NumS_ou_0]; t : TabPartList[NumS, NumL];
axioms %%Defn. insertTab:
   %%1er arg = numero de ligne a inserer,
   %%2me arg = table listes numeros ligne,
   %%3me arg = clef liste d'insertion et
   %%4me arg = tableau pour choix point d'insertion.
   %%Domaine insertTab.
   def insertTab(nl, t, ns, a)
   ⇔ (∀ n : NumL; ns1 : NumS • n is_in retrieve(ns1, t) ⇒ n < nl ∧ def(read a at n)) ∧
       def retrieve(ns, t) ∧
       def(read a at nl);
   %%Ajoute le numero de ligne en tete de liste
   %%ou parcourt le reste de la liste en appellant point_inser.
   def insertTab(nl, t, ns, a) ⇒ insertTab(nl, t, ns, a)
   =   consTab(nl, ns, t)
       when( retrieve(ns, t)is_empty ∨
             (read a at nl ≤ read a at (head_of retrieve(ns, t))))
       else insert nl
            after point_inser(nl, retrieve(ns, t), a)
            in t;
end
```

spec CONSTRUCTABLEPARTITIONNEUSENUMSLIGNESSORTANTOUENTRANT =
%%Spec., en terme des preds et ops de base sur tables, listes, tableaux,
%%positifs et tuples, d'un algorithme pour la construction d'une table
%%partitionneuse de listes de numeros de ligne
%%dont les elements et leur ordre sont determines par deux tableaux.
  %%Sortes NumS, NumS_ou_0, NumL, NumL_ou_0, Array[NumL,NumS_ou_0],
  %%et TabPartList[NumS,NumL] et ops new et empty__at__.
  TABLEPARTIONNEUSENUMSLIGNE
  and
  %%Op insertTab.
  INSERENUMLIGNE
then
op *construit_table_cocycles* : $NumS\_ou\_0 * NumL\_ou\_0 *$
  $Array[NumL, NumS\_ou\_0] * Array[NumL, NumS\_ou\_0]$
  $\to ?\ TabPartList[NumS, NumL]$;
then local
  op *empty__upto__* : $TabPartList[NumS, NumL] * NumS\_ou\_0$
    $\to TabPartList[NumS, NumL]$;
  remplit_table_cocycles : $NumL\_ou\_0 * TabPartList[NumS, NumL] *$
    $Array[NumL, NumS\_ou\_0] * Array[NumL, NumS\_ou\_0]$
    $\to ?\ TabPartList[NumS, NumL]$;
  vars $ns0 : NumS\_ou\_0;\ nl0 : NumL\_ou\_0;\ a1, a2 : Array[NumL, NumS\_ou\_0]$;
  $t : TabPartList[NumS, NumL]$;
  axioms %%Defn. empty __ upto __.
  $empty\ t\ upto\ ns0 =$
    t
    $when\ ns0 \le 0$
    $else\ empty\ (empty\ t\ upto(ns0 - 1))\ at\ (ns0\ as\ NumS)$;

  %%Defn. remplit_table_cocycles:
  %%Ajoute les numeros de ligne un par un, du 1er au dernier,
  $def\ remplit\_table\_cocycles(nl0, t, a1, a2)$
  $\Leftrightarrow\ \forall\ nl : NumL \bullet (nl \le nl0 \Rightarrow read\ a1\ at\ nl \le ns0 \wedge def(read\ a2\ at\ nl))$;
  $def\ remplit\_table\_cocycles(nl0, t, a1, a2) \Rightarrow remplit\_table\_cocycles(nl0, t, a1, a2) =$
    t
    $when\ (nl0 \le 0 \vee$
      $read\ a1\ at(nl0\ as\ NumL) \le 0 \vee$
      $read\ a2\ at(nl0\ as\ NumL) \le 0)$
    $else$
    $insertTab(nl0\ as\ NumL, remplit\_table\_cocycles(nl0 - 1, t, a1, a2), read\ a1\ at\ nl\ as\ NumS, a2)$;
within
vars $ns0 : NumS\_ou\_0;\ nl0 : NumL\_ou\_0;\ a1, a2 : Array[NumL, NumS\_ou\_0]$;
axiom %%Defn. construit_table_cocycles.
  $construit\_table\_cocycles(ns0, nl0, a1, a2) = remplit\_table\_cocycles(nl0, empty\ new\ upto\ ns0, a1, a2)$;
end

---

spec ABSTRACT7 =
%%Spec. d'une implementation a l'aide de tableaux des sommets et lignes
%%d'un reseau et d'ordres sur les numeros de ligne et spec. declarative
%%de tables partitionneuses indexees par les numeros de sommet et
%%stockant des listes de numeros de ligne et spec. d'un algorithme pour
%%la construction des tables stockant les listes ordonnes des numeros
%%des lignes sortant/entrant pour chaque numero de sommet d'un reseau.
CONSTRUCTABLEPARTITIONNEUSENUMSLIGNESSORTANTOUENTRANT
hide $List[NumL]$

Annexe R

La spécification Abstract8

```
spec ElemOrNil =
%%Spec. du parametre des schemas d'implementation des listes
%%sans doublons et des tables de listes partitionneuses.
sort Elem;
free type Elem_or_nil ::= sort Elem | nil;
end
```

```
spec ImplListDiffElems[ElemOrNil] =
%%Spec. generique en terme des preds et ops de base sur tableaux et tuples
%%d'une implementation partielle des listes sans doublons : ses constructeurs
%%ne sont pas specifies ici, ni les limites de la sorte des listes.
  %%Sorte Array[Elem,Elem_or_nil] et op read__at__.
  Array[ElemOrNil fit Index↦Elem]
       [ElemOrNil fit Elem↦Elem_or_nil]
  and
  %%Sorte Pair[Elem_or_nil,Array[Elem,Elem_or_nil]]
  %%et ops <<__ __>>, start_of et end_of.
  Pair[sort Elem_or_nil fit S1↦Elem_or_nil]
       [sort Array[Elem, Elem_or_nil] fit S2↦Array[Elem, Elem_or_nil]]
       with 1st_of↦start_of, 2nd_of↦end_of
then
  %%La sorte des listes sans doublons est definie comme une sous-sorte
  %%de la paire, mais sans preciser quelle sous-sorte.
sort LDE[Elem] < Pair[Elem_or_nil, Array[Elem, Elem_or_nil]];
pred __is_empty : LDE[Elem];
ops head_of__ : LDE[Elem] →? Elem;
  tail_of__ : LDE[Elem] →? LDE[Elem];
vars l : LDE[Elem];

axioms %%Defn. is_empty.
  l is_empty ⇔ (start_of l = nil);

  %%Defn. head_of.
  head_of l = start_of l as Elem;

  %%Defn. tail_of.
  tail_of l =<< read(end_of l)at(start_of l as Elem)  end_of l >>;
end
```

```
spec ImplTablePartitioningLists[sort Key;][ElemOrNil] =
%%Spec. generique en terme des preds et ops de base sur tableaux et tuples
%%d'une implementation de tables stockant un ensemble sans intersection
%%de listes sans doublons.
   %%Sorte Array[Key,Elem_or_nil] et ops new, write__at__in__ et read__at__.
   Array[sort Key fit Index↦Key]
        [ElemOrNil fit Elem↦Elem_or_nil]
   and
   %%Sortes Elem, LDE[Elem], Pair[Array[Key,Elem_or_nil],Array[Elem,Elem_or_nil]] et
   %%ops write__at__in__, read__at__, <<__ __>>, starts_of, ends_of, head_of, tail_of.
   Pair[sort Array[Key, Elem_or_nil] fit S1↦Array[Key, Elem_or_nil]]
        [ImplListDiffElems[ElemOrNil] fit S2↦Array[Elem, Elem_or_nil]]
        with 1st_of↦starts_of, 2nd_of↦ends_of
then
sort TabPartList[Key, Elem] < Pair[Array[Key, Elem_or_nil], Array[Elem, Elem_or_nil]];
generated type LDE[Elem] ::= retrieve(Key; TabPartList[Key, Elem])?;
generated type TabPartList[Key, Elem] ::=
                         new |
                         empty__at__(TabPartList[Key, Elem]; Key) |
                         consTab(Elem; Key; TabPartList[Key, Elem])?;
op insert__after__in__ : Elem * Elem * TabPartList[Key, Elem] →? TabPartList[Key, Elem];
then local
   pred    __is_in__ : Elem * LDE[Elem];
           __is_in__ : Elem * TabPartList[Key, Elem];
   vars    e : Elem; l : LDE[Elem]; t : TabPartList[Key, Elem];
   axiom   e is_in l ⇔ ((e = head_of l) ∨ e is_in tail_of l);
           e is_in t ⇔  ∃ i : Key • e is_in retrieve(i, t);
within
vars k : Key e, e1, e2 : Elem; t : TabPartList[Key, Elem];
axioms %%Constructeur new (composant ends_of pas defini).
   starts_of new = new;

   %%Constructeur empty __ at __.
   empty t at k =<< write 0 at k in starts_of t  ends_of t >>;

   %%Constructeur consTab.
   def consTab(e, k, t) ⇔ (def read(starts_of t)at k ∧ ¬(e is_in t));
   def consTab(e, k, t) ⇒ consTab(e, k, t) =
        << write e at k in(starts_of t)
             write(read starts_of t at k)at e in(ends_of t) >>;

   %%Constructeur derive insert _ after __ in _.
   def insert e2 after e1 in t ⇔ (e1 is_in t ∧ ¬(e2 is_in t));
   def insert e2 after e1 in t ⇒ insert e2 after e1 in t =
        << starts_of t
             write e2 at e1 in(write(read(ends_of t)at e1)at e2 in(ends_of t)) >>;

   %%Observateur retrieve.
   retrieve(k, t) =<< read(starts_of t)at k  ends_of t >>;
end
```

spec IMPLTABLENUMSLIGNE =
%%Spec. en terme des preds et ops de base sur tableaux, positifs, nats
%%et tuples, d'une implementation de tables stockant un ensemble sans
%%intersection de listes sans doublons de numeros de ligne.
IMPLTABLEPARTITIONINGLISTS[IMPLRESEAU **fit** $Key \mapsto NumS$]
[IMPLRESEAU **fit** $Elem \mapsto NumL$, $Elem\_or\_nil \mapsto NumL\_ou\_0$, $nil \mapsto 0$]

---

spec IMPLINSERENUMLIGNE =
%%Spec. en terme preds et ops de base sur tables, listes, tableaux, positifs
%%d'un algorithme pour l'insertion d'un numero de ligne dans la table.
%%Sortes NumL, Array[NumL,NumS_ou_0], LDE[NumL], TabPartList[NumS,NumL],
%%preds is_empty, is_in et ops read__at__, head_of, tail_of, retrieve, insert__after__in__.
IMPLTABLENUMSLIGNES
then
op $insertTab : NumL * TabPartList[NumS, NumL]*$
$NumS * Array[NumL, NumS\_ou\_0]$
$\rightarrow ?\ TabPartList[NumS, NumL];$
then local
op $point\_inser : NumL * LDE[NumL] * Array[NumL, NumS\_ou\_0] \rightarrow ?\ NumL;$
vars $nl : NumL;\ a : Array[NumL, NumS\_ou\_0];\ l : LDE[NumL];$
axioms %%Domaine point_inser.
$def\ point\_inser(nl, l, a)$
$\Leftrightarrow \neg\ (l\ is\_empty) \wedge$
$\forall\ n : NumL \bullet ((n\ is\_in\ l \Rightarrow (n < nl)) \wedge ((n \leq nl) \Rightarrow\ def\ read\ a\ at\ n));$
%%Parcourt la liste en cherchant le bon point d'insertion.
$def\ point\_inser(nl, l, a) \Rightarrow point\_inser(nl, l, a)$
$=\ \ head\_of\ l$
$when((tail\_of\ l) is\_empty \vee (read\ a\ at\ nl \leq read\ a\ at\ head\_of(tail\_of\ l)))$
$else\ point\_inser(nl, (tail\_of\ l) as\ LDE[NumL], a);$
within
vars $nl : NumL;\ a1, a2 : Array[NumL, NumS\_ou\_0];\ t : TabPartList[NumS, NumL];$
axioms %%Defn. insertTab:
%%1er arg = numero de ligne a inserer,
%%2me arg = table listes numeros ligne,
%%3me arg = clef liste d'insertion et
%%4me arg = tableau pour choix point d'insertion.
%%Domaine insertTab.
$def\ insertTab(nl, t, ns, a)$
$\Leftrightarrow\ (\forall\ n : NumL;\ ns1 : NumS \bullet n\ is\_in\ retrieve(ns1, t) \Rightarrow n < nl \wedge def(read\ a\ at\ n)) \wedge$
$def\ retrieve(ns, t) \wedge$
$def(read\ a\ at\ nl);$
%%Ajoute le numero de ligne en tete de liste
%%ou parcourt le reste de la liste en appellant point_inser.
$def\ insertTab(nl, t, ns, a) \Rightarrow insertTab(nl, t, ns, a)$
$=\ \ consTab(nl, ns, t)$
$when(\ retrieve(ns, t) is\_empty \vee$
$(read\ a\ at\ nl \leq read\ a\ at\ (head\_of\ retrieve(ns, t))))$
$else\ insert\ nl$
$after\ point\_inser(nl, retrieve(ns, t), a)$
$in\ t;$
end

spec IMPLCONSTRUCTABLENUMSLIGNESSORTANTOUENTRANT=
%%Spec. en terme preds et ops de base sur tableaux, positifs, nats, tuples
%%d'une implementation de la construction d'une table partitionneuse
%%de listes de numeros de ligne
%%dont les elements et leur ordre sont determines par deux tableaux.
  %%Sortes NumS, NumS_ou_0, NumL, NumL_ou_0, Array[NumL,NumS_ou_0],
  %%et TabPartList[NumS,NumL] et ops new et empty__at__.
  IMPLTABLENUMSLIGNE
  and
  %%Op insertTab.
  IMPLINSERENUMLIGNE
then
ops $construit\_table\_cocycles : NumS\_ou\_0 * NumL\_ou\_0 *$
$Array[NumL, NumS\_ou\_0] * Array[NumL, NumS\_ou\_0]$
$\rightarrow ?\ TabPartList[NumS, NumL];$
then local
  ops $empty\_\_upto\_\_ : TabPartList[NumS, NumL] * NumS\_ou\_0$
$\rightarrow TabPartList[NumS, NumL];$
$remplit\_table\_cocycles : NumL\_ou\_0 * TabPartList[NumS, NumL] *$
$Array[NumL, NumS\_ou\_0] * Array[NumL, NumS\_ou\_0]$
$\rightarrow ?\ TabPartList[NumS, NumL];$
  vars $ns0 : NumS\_ou\_0;\ nl0 : NumL\_ou\_0;\ a1, a2 : Array[NumL, NumS\_ou\_0];$
$t : TabPartList[NumS, NumL];$
  axioms %%Defn. empty __ upto __.
$empty\ t\ upto\ ns0 =$
t
$when\ ns0 \leq 0$
$else\ empty\ (empty\ t\ upto(ns0 - 1))\ at\ (ns0\ as\ NumS);$

%%Defn. remplit_table_cocycles:
%%Ajoute les numeros de ligne un par un, du 1er au dernier,
$def\ remplit\_table\_cocycles(nl0, t, a1, a2)$
$\Leftrightarrow\ \forall\ nl : NumL \bullet (nl \leq nl0 \Rightarrow read\ a1\ at\ nl \leq ns0 \wedge def(read\ a2\ at\ nl));$
$def\ remplit\_table\_cocycles(nl0, t, a1, a2) \Rightarrow remplit\_table\_cocycles(nl0, t, a1, a2) =$
t
$when\ (nl0 \leq 0 \vee$
$read\ a1\ at(nl0\ as\ NumL) \leq 0 \vee$
$read\ a2\ at(nl0\ as\ NumL) \leq 0)$
$else$
$insertTab(nl0\ as\ NumL, remplit\_table\_cocycles(nl0 - 1, t, a1, a2), read\ a1\ at\ nl\ as\ NumS, a2);$
within
vars $ns0 : NumS\_ou\_0;\ nl0 : NumL\_ou\_0;\ a1, a2 : Array[NumL, NumS\_ou\_0];$
axiom %%Defn. construit_table_cocycles.
  $construit\_table\_cocycles(ns0, nl0, a1, a2) = remplit\_table\_cocycles(nl0, empty\ new\ upto\ ns0, a1, a2);$
end

---

spec ABSTRACT8 =
%%Spec. en terme preds et ops de base sur tableaux, positifs, nats, tuples
%%d'une implementation des sommets et lignes d'un reseau et de tables
%%partitionneuses indexees par les positifs et stockant des listes de positifs
%%dont les elements et leur ordre sont determines par deux tableaux
%%et pouvant stocker les numeros des lignes sortant resp. entrant
%%pour chaque numero de sommet d'un reseau.
IMPLCONSTRUCTABLENUMSLIGNESSORTANTOUENTRANT

Annexe S

La spécification Abstract2 $'$

spec IMPLCORRECTERESEAU =
%%Spec. de l'implementation totale du reseau.
%%Sortes Sommet et Set[Sommet].
SET[SOMMETABSTRAIT **fit** *Elem*↦*Sommet*]
and
%%Sortes Ligne et Set[Ligne].
SET[LIGNEABSTRAITE **fit** *Elem*↦*Ligne*]
and
IMPLRESEAU
then
ops %%Numerotation des sommets resp. lignes
$num : Sommet * Reseau \rightarrow ?\ NumS;$
$num : Ligne * Reseau \rightarrow ?\ NumL;$

%%Recuperation d'un sommet resp. ligne par son numero
$sommet : NumS * Reseau \rightarrow ?\ Sommet;$
$ligne : NumL * Reseau \rightarrow ?\ Ligne;$

%%Les sommets et lignes appartenant au reseau.
$sommets\_de\_\_ : Reseau \rightarrow Set[Sommet];$
$lignes\_de\_\_ : Reseau \rightarrow Set[Ligne];$
vars $s, s1, s2 : Sommet;\ l, l1, l2 : Ligne;\ ns : NumS;\ nl : NumL;\ r : Reseau;$
axioms %%Tout sommet resp. ligne appartient a au moins un reseau.
$\forall\ s : Sommet \bullet \exists\ r : Reseau \bullet s\ is\_in\ sommets\_de\ r;$
$\forall\ l : Ligne \bullet \exists\ r : Reseau \bullet l\ is\_in\ lignes\_de\ r;$

%%La sorte Sommet resp. Ligne est partitionnee par les valeurs des attributs.
$s1 = s2$
$\Leftrightarrow id\_de\ s1 = id\_de\ s2\ \wedge$
$\quad attribS1\_de\ s1 = attribS1\_de\ s2;$
$l1 = l2$
$\Leftrightarrow id\_de\ l1 = id\_de\ l2\ \wedge$
$\quad attribL1\_de\ l1(nlr1) = attribL1\_de\ l2\ \wedge$
$\quad l1\ est\_connectee\_orig \Leftrightarrow l2\ est\_connectee\_orig\ \wedge$
$\quad l1\ est\_connectee\_extrem \Leftrightarrow l2\ est\_connectee\_extrem\ \wedge$
$\quad origine\_de\ l1 = origine\_de\ l2\ \wedge$
$\quad extremite\_de\ l1 = extremite\_de\ l2;$

%%Defn. num:Sommet*Reseau ->? NumS.
%%Injective.
$((num(s1, r) = ns) \wedge (num(s2, r) = ns)) \Rightarrow s1 = s2;$
%%Inverse de sommet.
$def\ sommet(ns, r) \Rightarrow num(sommet(ns, r), r) = ns;$

%%Defn. num:Ligne*Reseau ->? NumL.
%%Injective.
$(num(l1, r) = nl \wedge num(l2, r) = nl) \Rightarrow l1 = l2;$
%%Inverse de ligne.
$def\ ligne(nl, r) \Rightarrow num(ligne(nl, r), r) = nl;$

%%Domaine sommet et implementation
%%id_de et attribS1_de.
$def\ sommet(ns, r) \Leftrightarrow ns \leq r.NS;$
$id\_de\ sommet(ns, r) = id(ns, r);$
$attribS1\_de\ sommet(ns, r) = attribS1(ns, r);$

%%Domaine ligne et implementation
%%est_connectee_orig,
%%est_connectee_extrem,
%%id_de,
%%attribL1_de,
%%origine_de et
%%extremite_de.
$def\ ligne(nl, r) \Leftrightarrow nl \leq r.NL;$
$ligne(nl, r)\ est\_connectee\_orig \Leftrightarrow est\_connectee\_orig(nl, r);$
$ligne(nl, r)\ est\_connectee\_extrem \Leftrightarrow est\_connectee\_extrem(nl, r);$
$id\_de\ ligne(nl, r) = id(nl, r);$
$attribL1\_de\ ligne(nl, r) = attribL1(nl, r);$
$def\ origine\_de\ ligne(nl, r) \Rightarrow origine\_de\ ligne(nl, r) = sommet(num\_orig(nl, r), r);$
$def\ extremite\_de\ ligne(nl, r) \Rightarrow extremite\_de\ ligne(nl, r) = sommet(num\_extrem(nl, r), r);$

%%Defn. sommets_de.
$s\ is\_in\ sommets\_de\ r \Leftrightarrow\ \exists\ ns : NumS \bullet sommet(ns, r) = s;$

%%Defn. lignes_de.
$l\ is\_in\ lignes\_de\ r \Leftrightarrow\ \exists\ nl : NumL \bullet ligne(nl, r) = l;$
end

---

spec IMPLCORRECTE2ENSEMBLESNUMSLIGNESSORTANTETENTRANT =
%%Spec. de l'implementation du calcul des ensembles
%%des lignes sortant resp. entrant pour chaque sommet du reseau.
IMPLCORRECTERESEAU
and
ENSEMBLESNUMSLIGNESSORTANTETENTRANT
then
ops $lignes\_sortant : Sommet * Reseau \to ?\ Set[Ligne];$
$lignes\_entrant : Sommet * Reseau \to ?\ Set[Ligne];$
vars $r : Reseau;\ l : Ligne;\ s : Sommet;$
axioms %%Defn. lignes_sortant.
$l\ is\_in\ lignes\_sortant(s, r)$
$\Leftrightarrow num(l, r)\ is\_in\ nums\_lignes\_sortant\ (num(s, r), r);$

%%Defn. lignes_entrant.
$l\ is\_in\ lignes\_entrant(s, r)$
$\Leftrightarrow num(l, r)\ is\_in\ nums\_lignes\_entrant\ (num(s, r), r);$
end

---

spec ABSTRACT2' = IMPLCORRECTE2ENSEMBLESNUMSLIGNESSORTANTETENTRANT
hide $empty : Set[NumL],$
$add\_\_to\_\_ : NumL * Set[NumL] \to Set[NumL],$
$\_\_is\_empty : Set[NumL],$
$\_\_is\_in\_\_ : NumL * Set[NumL]$
$empty : Set[Sommet],$
$add\_\_to\_\_ : Sommet * Set[Sommet] \to Set[Sommet],$
$\_\_is\_empty : Set[Sommet],$
$\_\_is\_in\_\_ : Sommet * Set[Sommet]$
$empty : Set[Ligne],$
$add\_\_to\_\_ : Ligne * Set[Ligne] \to Set[Ligne],$
$\_\_is\_empty : Set[Ligne],$
$\_\_is\_in\_\_ : Ligne * Set[Ligne]$

Annexe T

La spécification Abstract6 $'$

spec IMPLCORRECTE6LISTESNUMSLIGNESSORTANTETENTRANT =
%%Implementation correcte du module ListesNumsLignesSortantEtEntrant.
CONSTRUCTABLENUMSLIGNESSORTANTOUENTRANT
then
ops $nums\_lignes\_sortant : NumS * Reseau \to ?\ LDE[NumL]$;
$nums\_lignes\_entrant : NumS * Reseau \to ?\ LDE[NumL]$;
vars $ns : NumS$; $r : Reseau$;
axioms $nums\_lignes\_sortant(ns, r)$
$= retrieve(ns, construit\_table\_cocycles(r.NS, r.NL, r.NRLO, r.NRLE))$;
$nums\_lignes\_entrant(ns, r)$
$= retrieve(ns, construit\_table\_cocycles(r.NS, r.NL, r.NRLE, r.NRLO))$;
end

---

spec ABSTRACT6' =
IMPLCORRECTE6LISTESNUMSLIGNESSORTANTETENTRANT
hide $List[NumL]$
$\_\_ <_a \_\_dans\_\_$,
$\_\_est\_ord\_dans\_\_$,
$insert$,
new,
add

Annexe U

La spécification Abstract7 ′

spec IMPLCORRECTE7TABLENUMSLIGNE =
%%Implementation correcte du module TableNumsLigne.
%%Sorte Table[NumS,LDE[NumL]].
TABLE[IMPLRESEAU **fit** $Key \mapsto NumS$]
[LISTDIFFELEMS[IMPLRESEAU **fit** $Elem \mapsto NumL$] **fit** $Item \mapsto LDE[NumL]$]
and
%%Sorte TabPartList[NumS,NumL].
TABLEPARTIONNEUSENUMSLIGNE
then
sort %%Implique que new:TabPartList[NumS,NumL] est le
%%resultat de la projection de new:Table[NumS,LDE[NumL]] et
%%que retrieve:NumS*TabPartList[NumS,NumL]->?LDE[NumL] est la
%%restriction de retrieve:NumS*Table[NumS,LDE[NumL]]->?LDE[NumL].
$TabPartList[NumS, NumL] < Table[NumS, LDE[NumL]]$;
vars $ns : NumS$; $l : LDE[NumL]$; $t : Table[NumS, LDE[NumL]]$;
axiom %%Defn. d'add pour les tables partitionneuses en termes
%%des primitifs des tables partitionneuses.
$t \in TabPartList[NumS, NumL] \Rightarrow add(ns, l, t)$
$= empty\ (t\ as\ TabPartList[NumS, NumL])\ at\ ns$
$when\ l\ is\_empty$
$else\ consTab(head\_of\ \ l, ns, add(ns, tail\_of\ \ l, t)\ as\ TabPartList[NumS, NumL])$;
end

---

spec IMPLCORRECTE7CONSTRUCTABLENUMSLIGNESSORTANTOUENTRANT =
%%Implementation correcte du module ConstrucTableNumsLignesSortantOuEntrant.
%%Sorte Table[NumS,LDE[NumL]].
IMPLCORRECTE7TABLENUMSLIGNE
and
%%Op construit_table_cocycles avec resultat dans TabPartList[NumS,NumL].
CONSTRUCTABLEPARTIONNEUSENUMSLIGNESSORTANTOUENTRANT
then
op $construit\_table\_cocycles : NumS\_ou\_0 * NumL\_ou\_0 *$
$Array[NumL, NumS\_ou\_0] * Array[NumL, NumS\_ou\_0]$
$\rightarrow?\ Table[NumS, LDE[NumL]]$;
end

---

spec ABSTRACT7' =
IMPLCORRECTE7CONSTRUCTABLENUMSLIGNESSORTANTOUENTRANT
hide $List[NumL]$

Annexe V

La spécification Post-Abstract1PD

spec ENSEMBLESOMMETSDEMEMERESEAU1PD =
%%Spec. du produit dependant d'un ensemble de sommets et d'un reseau auquel elles appartiennent.
%%Sortes Pair[Set[Sommet],Reseau], Set[Sommet] et Reseau et op 2nd_of.
PAIR[SET[**sort** *Sommet* **fit** *Elem*↦*Sommet*] **fit** *S1*↦*Set*[*Sommet*]]
[**sort** *Reseau* **fit** *S2*↦*Reseau*]
then
sort $EnsembleSommetsDeMemeReseau < Pair[Set[Sommet], Reseau]$;
op $sommets\_de : Reseau \rightarrow EnsembleSommetsDeMemeReseau$;
sort $EnsembleSommetsDeMemeReseau = \{\ p : Pair[Set[Sommet], Reseau] \bullet \forall s : Sommet \bullet$
$s\ is\_in\ (1st\_of\ p) \Rightarrow s\ is\_in\ 1st\_of\ (sommets\_de\ (2nd\_of\ p));\ \}$
vars $r : Reseau$;
axiom $2nd\_of\ (sommets\_de\ r) = r$;
end

———————————————————————————

spec SOMMETDERESEAU1PD =
%%Spec. du produit dependant d'un sommet et d'un reseau auquel il appartient.
%%Sorte Pair[Sommet,Reseau], pred is_in et op sommets_de.
PAIR[ENSEMBLESOMMETSDEMEMERESEAU1PD**fit** *S1*↦*Sommet*]
[ENSEMBLESOMMETSDEMEMERESEAU1PD**fit** *S2*↦*Reseau*]
then
sort $SommetDeReseau < Pair[Sommet, Reseau]$;
pred $\_\_is\_in\_\_ : Pair[Sommet, Reseau] * Pair[Set[Sommet], Reseau]$;
sort $SommetDeReseau = \{\ p : Pair[Sommet, Reseau] \bullet p\ is\_in\ sommets\_de\ (2nd\_of\ p);\ \}$
vars $p1 : Pair[Sommet, Reseau];\ p2 : Pair[Set[Sommet], Reseau]$;
axiom %%Defn. is_in:Pair[Sommet,Reseau]*Pair[Set[Sommet],Reseau].
$p1\ is\_in\ p2$
$\Leftrightarrow ((1st\_of\ p1)\ is\_in\ (1st\_of\ p2) \wedge 2nd\_of\ p1 = 2nd\_of\ p2)$;
end

———————————————————————————

spec SOMMET1PD =
%%Sorte SommetDeReseau.
SOMMETDERESEAU1PD
then
sorts *IdSommet*;
AttribS1;
ops $id\_de\_\_ : SommetDeReseau \rightarrow IdSommet$;
$attribS1\_de\_\_ : SommetDeReseau \rightarrow AttribS1$;
vars $sr1, sr2 : SommetDeReseau$;
axioms %%Chaque sommet a un id unique a lui.
$(id\_de\ sr1 = id\_de\ sr2) \Rightarrow (1st\_of\ sr1 = 1st\_of\ sr2)$;

%%Les valeurs des attributs ne varient pas avec le reseau.
$1st\_of\ sr1 = 1st\_of\ sr2 \Rightarrow id\_de\ sr1 = id\_de\ sr2$;
end

spec ENSEMBLELIGNESDEMEMERESEAU1PD =
%%Spec. du produit dependant d'un ensemble de lignes et d'un reseau auquel elles appartiennent.
%%Sortes Pair[Set[Ligne],Reseau], Set[Ligne] et Reseau et op 2nd_of.
PAIR[SET[**sort** *Ligne* **fit** *Elem*$\mapsto$*Ligne*] **fit** *S1*$\mapsto$*Set*[*Ligne*]]
[**sort** *Reseau* **fit** *S2*$\mapsto$*Reseau*]
then
sort $EnsembleLignesDeMemeReseau < Pair[Set[Ligne], Reseau]$;
op $lignes\_de : Reseau \rightarrow EnsembleLignesDeMemeReseau$;
sort $EnsembleLignesDeMemeReseau = \{\ p : Pair[Set[Ligne], Reseau] \ \bullet\ \forall\ l : Ligne \bullet$
$l\ is\_in\ (1st\_of\ p) \Rightarrow l\ is\_in\ 1st\_of\ (lignes\_de\ (2nd\_of\ p));\ \}$
vars $r : Reseau$;
axiom $2nd\_of\ (lignes\_de\ r) = r$;
end

---

spec LIGNEDERESEAU1PD =
%%Spec. du produit dependant d'une ligne et d'un reseau auquel elle appartient.
%%Sortes Pair[Ligne,Reseau] et Pair[Set[Ligne],Reseau],
%%pred is_in et ops 1st_of, 2nd_of et lignes_de.
PAIR[ENSEMBLELIGNESDEMEMERESEAU1PD**fit** *S1*$\mapsto$*Ligne*]
[ENSEMBLELIGNESDEMEMERESEAU1PD**fit** *S2*$\mapsto$*Reseau*]
then
sort $LigneDeReseau < Pair[Ligne, Reseau]$;
pred $\_\_is\_in\_\_ : Pair[Ligne, Reseau] * Pair[Set[Ligne], Reseau]$;
sort $LigneDeReseau = \{\ p : Pair[Ligne, Reseau] \ \bullet\ p\ is\_in\ lignes\_de\ (2nd\_of\ p);\ \}$
vars $p1 : Pair[Ligne, Reseau];\ p2 : Pair[Set[Ligne], Reseau]$;
axiom %%Defn. is_in:Pair[Ligne,Reseau]*Pair[Set[Ligne],Reseau].
$p1\ is\_in\ p2$
$\Leftrightarrow ((1st\_of\ p1)\ is\_in\ (1st\_of\ p2) \wedge 2nd\_of\ p1 = 2nd\_of\ p2)$;
end

```
spec LIGNE1PD =
  %%Sorte SommetDeReseau.
  SOMMET1PD
  and
  %%Sorte LigneDeReseau et op 2nd_of.
  LIGNEDERESEAU1PD
then
sorts IdLigne;
  AttribL1;
preds __est_connectee_orig : LigneDeReseau;
  __est_connectee_extrem : LigneDeReseau;
ops id_de__ : LigneDeReseau → IdLigne;
  attribL1_de__ : LigneDeReseau → AttribL1;
  origine_de__ : LigneDeReseau →? SommetDeReseau;
  extremite_de__ : LigneDeReseau →? SommetDeReseau;
vars lr, lr1, lr2 : LigneDeReseau;
axioms %%Ops origine_de et extremite_de definies pour les lignes connectees.
  def origine_de lr ⇔ lr est_connectee_orig;
  def extremite_de lr ⇔ lr est_connectee_extrem;

  %%Les lignes ne sont pas des boucles.
  ¬(lr est_connectee_orig ∧ origine_de lr = extremite_de lr);

  %%Entre deux lignes du meme id,
  %%les seules proprietes qui peuvent varier sont
  %%la connexion ou non a l'origine et a l'extremite.
  (id_de lr1 = id_de lr2 ∧
   lr1 est_connectee_orig ⇔ lr2 est_connectee_orig ∧
   lr1 est_connectee_extrem ⇔ lr2 est_connectee_extrem )
  ⇒ 1st_of lr1 = 1st_of lr2;

  %%Les valeurs des attributs ne varient pas avec le reseau.
  1st_of lr1 = 1st_of lr2
  ⇒ id_de lr1 = id_de lr2 ∧
      lr1 est_connectee_orig ⇔ lr2 est_connectee_orig ∧
      lr1 est_connectee_extrem ⇔ lr2 est_connectee_extrem ∧
      origine_de lr1 = origine_de lr2 ∧
      extremite_de lr1 = extremite_de lr2;
end
```

spec RESEAU1PD =
%%Sorte LigneDeReseau, preds est_connectee_orig et est_connectee_extrem
%%et ops id_de, origine_de, extremite_de et 2nd_of.
LIGNE1PD
then
vars $lr, lr1, lr2 : LigneDeReseau;$
axioms %%Un reseau ne contient jamais deux lignes du meme id.
$2nd\_of\ lr1 = 2nd\_of\ lr2 \wedge$
$id\_de\ lr1 = id\_de\ lr2$
$\Rightarrow 1st\_of\ lr1 = 1st\_of\ lr2;$

%%Chaque bout de chaque ligne du reseau est soit non-connecte,
%%soit connecte a un sommet du reseau.
$def\ origine\_de\ lr \Rightarrow 2nd\_of\ (origine\_de\ lr) = 2nd\_of\ lr;$
$def\ extremite\_de\ lr \Rightarrow 2nd\_of\ (extremite\_de\ lr) = 2nd\_of\ lr;$

%%Deux lignes peuvent etre connectees en parallele
%%dans un reseau, mais pas dans des sens differents.
$(lr\ est\_connectee\_orig \wedge lr\ est\_connectee\_extrem)$
$\Rightarrow \neg (\exists\ lr1 : LigneDeReseau \bullet$
$(2nd\_of\ lr1 = 2nd\_of\ lr \wedge$
$origine\_de\ lr = extremite\_de\ lr1 \wedge$
$extremite\_de\ lr = origine\_de\ lr1)\ );$

%%Tout sommet et toute ligne appartient a au moins un reseau.
$\forall\ s : Sommet \bullet \exists\ sdr : SommetDeReseau \bullet s = 1st\_of\ sdr;$
$\forall\ l : Ligne \bullet \exists\ ldr : LigneDeReseau \bullet l = 1st\_of\ ldr;$
end

---

spec ENSEMBLESLIGNESSORTANTETENTRANT1PD =
%%Sortes SommetDeReseau, LigneDeReseau et EnsembleLignesDeMemeReseau,
%%preds est_connectee_orig, est_connectee_extrem et is_in et
%%ops origine_de et extremite_de.
RESEAU1PD
then
ops $lignes\_sortant : SommetDeReseau \to EnsembleLignesDeMemeReseau;$
$lignes\_entrant : SommetDeReseau \to EnsembleLignesDeMemeReseau;$
vars $sr : SommetDeReseau;\ lr : LigneDeReseau;$
axioms %%Les lignes sortant
%%d'un sommet sont connectees a ce sommet a l'origine
%%mais aussi a un autre sommet quelconque a l'extremite.
$lr\ is\_in\ lignes\_sortant(sr)$
$\Leftrightarrow lr\ est\_connectee\_extrem \wedge origine\_de\ lr = sr;$
$2nd\_of\ lignes\_sortant(sr) = 2nd\_of\ sr;$

%%Les lignes entrant
%%dans un sommet sont connectees a ce sommet a l'extremite
%%mais aussi a un autre sommet quelconque a l'origine.
$lr\ is\_in\ lignes\_entrant(sr)$
$\Leftrightarrow lr\ est\_connectee\_orig \wedge extremite\_de\ lr = sr;$
$2nd\_of\ lignes\_entrant(sr) = 2nd\_of\ sr;$
end

spec POST-ABSTRACT1PD =
ENSEMBLESLIGNESSORTANTETENTRANT1PD
hide *Sommet*,
Ligne,
Set[*Sommet*],
Set[*Ligne*],
Pair[*Sommet*, *Reseau*],
Pair[*Ligne*, *Reseau*],
Pair[*Set*[*Sommet*], *Reseau*],
Pair[*Set*[*Ligne*], *Reseau*]

Annexe W

La spécification Pre-Abstract2PD

```
spec NumSommetDeReseau2PD =
%%Spec. du produit dependant d'un numero de sommet et d'un reseau auquel il appartient.
  %%Sortes Pair[NumS,Reseau], NumS_ou_0 et Reseau
  %%et ops 1st_of et 2nd_of.
  Pair[NumeroSommet fit S1↦NumS][sort Reseau fit S2↦Reseau]
then
ops __.NS : Reseau → NumS_ou_0;
sort NumSommetDeReseau = { p : Pair[NumS, Reseau] • 1st_of p ≤ (2nd_of p).NS; }
end
```

```
spec ImplSommetsDeReseau2PD =
  %%Sortes NumS, NumS_ou_0, IdSommet, AttribS1,
  %%Array[NumS,IdSommet] et Array[NumS,AttribS1] et op read __ at __.
  Array[NumeroSommet fit Index↦NumS][sort IdSommet fit Elem↦IdSommet]
  and
  Array[NumeroSommet fit Index↦NumS][sort AttribS1 fit Elem↦AttribS1]
  and
  %%Sortes Reseau et NumSommetDeReseau et ops __.NS, 1st_of et 2nd_of.
  NumSommetDeReseau2PD
then
ops __.IdSommets : Reseau → Array[NumS, IdSommet];
  __.AttS1 : Reseau → Array[NumS, AttribS1];
  id : NumSommetDeReseau → IdSommet;
  attribS1 : NumSommetDeReseau → AttribS1;
vars ns : NumS; nsr, nsr1, nsr2 : NumSommetDeReseau; r : Reseau;
axioms %%Les tableaux stockant les attributs des sommets
  %%sont effectivement dimensionnes par __.NS.
  def (read r.IdSommets at ns) if ns ≤ r.NS;
  def (read r.AttS1 at ns) if ns ≤ r.NS;

  %%Chaque sommet (tous reseaux confondus) a un id unique a lui.
  id(nsr1) = id(nsr2) ⇒ attribS1(nsr1) = attribS1(nsr2);

  %%Dans un meme reseau, chaque numero designe un sommet different.
  (2nd_of nsr1 = 2nd_of nsr2 ∧ id(nsr1) = id(nsr2))
  ⇒ (nsr1 = nsr2);

  %%Lecture des attributs des sommets.
  id(nsr) = read ((2nd_of nsr).IdSommets) at (1st_of nsr);
  attribS1(nsr) = read ((2nd_of nsr).AttribS1) at (1st_of nsr);
end
```

```
spec NumLigneDeReseau2PD =
%%Spec. du produit dependant d'un numero de ligne et d'un reseau auquel elle appartient.
  %%Sortes Pair[NumL,Reseau], NumL_ou_0 et Reseau
  %%et ops 1st_of et 2nd_of.
  Pair[NumeroLigne fit S1↦NumL][sort Reseau fit S2↦Reseau]
then
ops __.NL : Reseau → NumL_ou_0;
sort NumLigneDeReseau = { p : Pair[NumL, Reseau] • 1st_of p ≤ (2nd_of p).NL; }
end
```

spec IMPLLIGNESDERESEAU2PD =
%%Sortes NumS, NumS_ou_0, Reseau et NumSommetDeReseau
%%et op __.NS.
IMPLSOMMETSDERESEAU2PD
and
%%Sortes NumL, NumL_ou_0, Array[NumL,IdLigne], Array[NumL,NumS_ou_0]
%%et Array[NumL,AttribL1] et ops read __ at __.
ARRAY[NUMEROLIGNE **fit** $Index \mapsto NumL$][**sort** $IdLigne$ **fit** $Elem \mapsto IdLigne$]
and
ARRAY[NUMEROLIGNE **fit** $Index \mapsto NumL$][NUMEROSOMMET **fit** $Elem \mapsto NumS\_ou\_0$]
and
ARRAY[NUMEROLIGNE **fit** $Index \mapsto NumL$][**sort** $AttribL1$ **fit** $Elem \mapsto AttribL1$]
and
%%Sorte NumLigneDeReseau et ops __.NL, 1st_of, 2nd_of et «__ __».
NUMLIGNEDERESEAU2PD
then
preds $est\_connectee\_orig : NumLigneDeReseau;$
$est\_connectee\_extrem : NumLigneDeReseau;$
ops $\_\_.IdLignes : Reseau \rightarrow Array[NumL, IdLigne];$
$\_\_.AttL1 : Reseau \rightarrow Array[NumL, AttribL1];$
$\_\_.NRLO : Reseau \rightarrow Array[NumL, NumS\_ou\_0];$
$\_\_.NRLE : Reseau \rightarrow Array[NumL, NumS\_ou\_0];$
$id : NumLigneDeReseau \rightarrow IdLigne;$
$attribL1 : NumLigneDeReseau \rightarrow AttribL1;$
$num\_orig : NumLigneDeReseau \rightarrow ?\ NumSommetDeReseau;$
$num\_extrem : NumLigneDeReseau \rightarrow ?\ NumSommetDeReseau;$
vars $nl : NumL;\ nlr, nlr1, nlr2 : NumLigneDeReseau;\ r : Reseau;$
axioms %%Les lignes du reseau sont soit non connectees
%%soit connectees a un sommet du reseau et
%%les tableaux stockant les attributs des lignes
%%sont effectivement dimensionnes par __.NL.
$def\ (read\ r.IdLignes\ at\ nl)\ if\ nl \leq r.NL;$
$def\ (read\ r.AttL1\ at\ nl)\ if\ nl \leq r.NL;$
$(read\ r.NRLO\ at\ nl \leq r.NS)\ if\ nl \leq r.NL;$
$(read\ r.NRLE\ at\ nl \leq r.NS)\ if\ nl \leq r.NL;$

%%Les lignes ne sont pas des boucles et
%%deux lignes peuvent etre connectees en parallele
%%dans un reseau, mais pas avec des orientations differentes.
$est\_connectee\_orig(nlr) \wedge est\_connectee\_extrem(nlr)$
$\Rightarrow \neg \exists\ nlr1 : NumLigneDeReseau \bullet$
$(2nd\_of\ nlr1 = 2nd\_of\ nlr\ \wedge$
$num\_orig(nlr) = num\_extrem(nlr1) \wedge$
$num\_extrem(nlr) = num\_orig(nlr1));$

%%Entre deux lignes du meme id (tous reseaux confondus),
%%les seules proprietes qui peuvent varier sont
%%la connexion ou non a l'origine et a l'extremite.
$id(nlr1) = id(nlr2) \Rightarrow attribL1(nlr1) = attribL1(nlr2);$

$id(nlr1) = id(nlr2) \wedge est\_connectee\_orig(nlr1) \wedge est\_connectee\_orig(nlr2)$
$\Rightarrow num\_orig(nlr1) = num\_orig(nlr2);$

$id(nlr1) = id(nlr2) \wedge est\_connectee\_extrem(nlr1) \wedge est\_connectee\_extrem(nlr2)$
$\Rightarrow num\_extrem(nlr1) = num\_extrem(nlr2);$

%%Un reseau ne contient jamais deux lignes du meme id
%%(et donc chaque numero designe une ligne differente).
$(2nd\_of\ nlr1 = 2nd\_of\ nlr2 \wedge id(nlr1) = id(nlr2))$
$\Rightarrow nlr1 = nlr2;$

%%Lecture des attributs des lignes.
$est\_connectee\_orig(nlr) \Leftrightarrow read\ ((2nd\_of\ nlr).NRLO)\ at\ (1st\_of\ nlr) > 0;$
$est\_connectee\_extrem(nlr) \Leftrightarrow read\ ((2nd\_of\ nlr).NRLE)\ at\ (1st\_of\ nlr) > 0;$
$id(nlr) = read\ ((2nd\_of\ nlr).IdLignes)\ at\ (1st\_of\ nlr);$
$attribL1(nlr) = read\ ((2nd\_of\ nlr).AttribL1)\ at\ (1st\_of\ nlr);$
$def\ num\_orig(nlr) \Leftrightarrow est\_connectee\_orig(nlr);$
$def\ num\_orig(nlr)$
$\Rightarrow num\_orig(nlr) =<< read\ ((2nd\_of\ nlr).NRLO)\ at\ (1st\_of\ nlr)\ \ 2nd\_of\ nlr >>;$
$def\ num\_extrem(nlr) \Leftrightarrow est\_connectee\_extrem(nlr);$
$def\ num\_extrem(nlr)$
$\Rightarrow num\_extrem(nlr) =<< read\ ((2nd\_of\ nlr).NRLE)\ at\ (1st\_of\ nlr)\ \ 2nd\_of\ nlr >>;$
end

---

spec ENSEMBLENUMSSOMMETSDEMEMERESEAU2PD =
%%Spec. du produit dependant d'un ensemble de numeros de sommet et
%%d'un reseau auquel ils appartiennent.
%%Sortes Pair[Set[NumS],Reseau] et NumS et pred is_in et ops 1st_of et 2nd_of.
PAIR[SET[NUMEROSOMMET **fit** *Elem*↦*NumS*] **fit** *S1*↦*Set*[*NumS*]]
[**sort** *Reseau* **fit** *S2*↦*Reseau*]
then
sort $EnsembleNumsSommetsDeMemeReseau = \{\ p : Pair[Set[NumS], Reseau]\ \bullet\ \forall\ ns : NumS \bullet$
$ns\ is\_in\ (1st\_of\ p) \Rightarrow ns \leq (2nd\_of\ p).NS;\ \}$
pred $\_\_is\_in\_\_ : NumSommetDeReseau * EnsembleNumsSommetsDeMemeReseau;$
vars $nsr : NumSommetDeReseau;\ snsr : EnsembleNumsSommetsDeMemeReseau;$
axiom %%Defn. is_in:NumSommetDeReseau*EnsembleNumsSommetsDeMemeReseau.
$nsr\ is\_in\ snsr$
$\Leftrightarrow ((1st\_of\ nsr)\ is\_in\ (1st\_of\ snsr) \wedge 2nd\_of\ nsr = 2nd\_of\ snsr);$
end

---

spec ENSEMBLENUMSLIGNESDEMEMERESEAU2PD =
%%Spec. du produit dependant d'un ensemble de numeros de ligne et
%%d'un reseau auquel elles appartiennent.
%%Sortes Pair[Set[NumL],Reseau] et NumL et pred is_in et ops 1st_of et 2nd_of.
PAIR[SET[NUMEROLIGNE **fit** *Elem*↦*NumL*] **fit** *S1*↦*Set*[*NumL*]]
[**sort** *Reseau* **fit** *S2*↦*Reseau*]
then
sort $EnsembleNumsLignesDeMemeReseau = \{\ p : Pair[Set[NumL], Reseau]\ \bullet\ \forall\ nl : NumL \bullet$
$nl\ is\_in\ (1st\_of\ p) \Rightarrow nl \leq (2nd\_of\ p).NL;\ \}$
pred $\_\_is\_in\_\_ : NumLigneDeReseau * EnsembleNumsLignesDeMemeReseau;$
vars $nlr : NumLigneDeReseau;\ snlr : EnsembleNumsLignesDeMemeReseau;$
axiom %%Defn. is_in:NumLigneDeReseau*EnsembleNumsLignesDeMemeReseau.
$nlr\ is\_in\ snlr$
$\Leftrightarrow ((1st\_of\ nlr)\ is\_in\ (1st\_of\ snlr) \wedge 2nd\_of\ nlr = 2nd\_of\ snlr);$
end

spec IMPLRESEAU2PD =
%%Spec. d'une implementation des sommets et lignes d'un reseau
%%a l'aide de tableaux.
%%Sorte NumSommetDeReseau et ops __.NS, 1st_of et 2nd_of.
IMPLSOMMETSDERESEAU2PD
and
%%Sorte NumLigneDeReseau et ops __.NL, 1st_of et 2nd_of.
IMPLLIGNESDERESEAU2PD
and
%%Sorte EnsembleNumsSommetsDeMemeReseau et ops is_in et 2nd_of.
ENSEMBLENUMSSOMMETSDEMEMERESEAU2PD
and
%%Sorte EnsembleNumsLignesDeMemeReseau et ops is_in et 2nd_of.
ENSEMBLENUMSLIGNESDEMEMERESEAU2PD
then
ops $nums\_sommets\_de : Reseau \rightarrow EnsembleNumsSommetsDeMemeReseau;$
$nums\_lignes\_de : Reseau \rightarrow EnsembleNumsLignesDeMemeReseau;$
vars $r : reseau;\ nsr : NumSommetDeReseau;\ nlr : NumLigneDeReseau;$
axioms $nsr\ is\_in\ (nums\_sommets\_de\ (2nd\_of\ nsr));$
$2nd\_of\ (nums\_sommets\_de\ r) = r;$
$nlr\ is\_in\ (nums\_lignes\_de\ (2nd\_of\ nlr));$
$2nd\_of\ (nums\_lignes\_de\ r) = r;$
end

---

spec ENSEMBLESNUMSLIGNESSORTANTETENTRANT2PD =
%%Sortes Reseau, NumSommetDeReseau, NumLigneDeReseau et
%%EnsembleNumsLignesDeMemeReseau,
%%preds is_in, est_connectee_extrem et est_connectee_orig et ops num_orig et num_extrem.
IMPLRESEAU2PD
then
ops $nums\_lignes\_sortant : NumSommetDeReseau \rightarrow EnsembleNumsLignesDeMemeReseau;$
$nums\_lignes\_entrant : NumSommetDeReseau \rightarrow EnsembleNumsLignesDeMemeReseau;$
vars $nsr : NumSommetDeReseau;\ nlr : NumLigneDeReseau;\ r : Reseau;$
axioms %%Les lignes sortant
%%d'un sommet sont connectees a ce sommet a l'origine
%%mais aussi a un autre sommet quelconque a l'extremite.
$nlr\ is\_in\ nums\_lignes\_sortant(nsr) \Leftrightarrow est\_connectee\_extrem(nlr) \wedge num\_orig(nlr) = nsr;$

%%Les lignes entrant
%%dans un sommet sont connectees a ce sommet a l'extremite
%%mais aussi a un autre sommet quelconque a origine.
$nlr\ is\_in\ nums\_lignes\_entrant(nsr) \Leftrightarrow est\_connectee\_orig(nlr) \wedge num\_extrem(nlr) = nsr;$
end

---

spec PRE-ABSTRACT2PD =
ENSEMBLESNUMSLIGNESSORTANTETENTRANT2PD
hide $Set[NumS],$
$\_\_is\_in\_\_ : NumSommetDeReseau * EnsembleNumsSommetsDeMemeReseau,$
$Set[NumS],$
$\_\_is\_in\_\_ : NumLigneDeReseau * EnsembleNumsLignesDeMemeReseau,$
$Pair[NumS, Reseau],$
$Pair[NumL, Reseau],$
$Pair[Set[NumS], Reseau],$
$Pair[Set[NumL], Reseau]$

Annexe X

La spécification Pre-Abstract2PD$'$

spec IMPLSOMMETPD =
%%Sorte NumSommetDeReseau.
PRE-ABSTRACT2PD
then
sort *Sommet*;
op *sommet* : *NumSommetDeReseau* → *Sommet*;
vars *nsr1*, *nsr2* : *NumSommetDeReseau*;
axioms %%Tout sommet appartient a au moins un reseau.
$\forall$ *s* : *Sommet* • $\exists$ *nsr* : *NumSommetDeReseau* • *sommet*(*nsr*) = *s*;

%%La sorte Sommet est partitionnee par les valeurs des attributs.
sommet(*nsr1*) = *sommet*(*nsr2*)
⇔ *id*(*nsr1*) = *id*(*nsr2*) ∧
attribS1(*nsr1*) = *attribS1*(*nsr2*);
end

---

spec IMPLENSEMBLESOMMETSPD =
%%Sortes EnsembleNumsSommetsDeMemeReseau et Set[Sommet] et pred is_in.
SET[IMPLSOMMETPD **fit** *Elem*↦*Sommet*]
then
op *sommets* : *EnsembleNumsSommetsDeMemeReseau* → *Set*[*Sommet*];
vars *s* : *Sommet*; *nsr* : *NumSommetDeReseau*; *snsr* : *EnsembleNumsSommetDeMemeReseau*;
axioms *s is_in sommets*(*snsr*) ⇒ ∃ *nsr* : *NumSommetDeReseau* •
nsr is_in snsr ∧ *sommet*(*nsr*) = *s*;
nsr is_in snsr ⇒ *sommet*(*nsr*) *is_in sommets*(*snsr*);
sommet(*nsr*) *is_in sommets*(*snsr*) ∧ *2nd_of nsr* = *2nd_of snsr*
⇒ (*1st_of nsr*) *is_in* (*1st_of snsr*);
end

---

spec IMPLPAIREENSEMBLESOMMETSRESEAUPD =
%%Sorte Pair[Set[Sommet],Reseau] et ops sommets et 2nd_of.
PAIR[IMPLENSEMBLESOMMETSPD **fit** *S1*↦*Set*[*Sommet*]][IMPLENSEMBLESOMMETSPD **fit** *S2*↦*Reseau*]
with *1st_of* __↦*sommets*
and
%%Sorte Pair[Sommet,Reseau] et ops sommet et 2nd_of.
PAIR[IMPLSOMMETPD **fit** *S1*↦*Sommet*][IMPLSOMMETPD **fit** *S2*↦*Reseau*]
with *1st_of* __↦*sommet*
then
sort *NumSommetDeReseau* < *Pair*[*Sommet*, *Reseau*];
EnsembleNumsSommetsDeMemeReseau < *Pair*[*Set*[*Sommet*], *Reseau*];
pred __*is_in*__ : *Pair*[*Sommet*, *Reseau*] ∗ *Pair*[*Set*[*Sommet*], *Reseau*];
vars *p1* : *Pair*[*Sommet*, *Reseau*]; *p2* : *Pair*[*Set*[*Sommet*], *Reseau*];
axiom %%Defn. is_in:Pair[Sommet,Reseau]*Pair[Set[Sommet],Reseau].
p1 is_in p2
⇔ (*sommet*(*p1*) *is_in sommets*(*p2*) ∧ *2nd_of p1* = *2nd_of p2*);
end

spec IMPLLIGNEPD =
  %%Sorte NumLigneDeReseau.
  PRE-ABSTRACT2PD
then
sort *Ligne*;
op $ligne : NumLigneDeReseau \rightarrow Ligne$;
vars $nlr1, nlr2 : NumLigneDeReseau$;
axioms %%Toute ligne appartient a au moins un reseau.
  $\forall\, l : Ligne \bullet \exists\, nlr : NumLigneDeReseau \bullet ligne(nlr) = l$;

  %%La sorte Ligne est partitionnee par les valeurs des attributs.
  $ligne(nlr1) = ligne(nlr2)$
  $\Leftrightarrow id(nlr1) = id(nlr2) \wedge$
    $attribL1(nlr1) = attribL1(nlr2) \wedge$
    $est\_connectee\_orig(nlr1) \Leftrightarrow est\_connectee\_orig(nlr2) \wedge$
    $est\_connectee\_extrem(nlr1) \Leftrightarrow est\_connectee\_extrem(nlr2) \wedge$
    $sommet(origine\_de\ nlr1) = sommet(origine\_de\ nlr2) \wedge$
    $sommet(extremite\_de\ nlr1) = sommet(extremite\_de\ nlr2)$;
end

---

spec IMPLENSEMBLELIGNESPD =
  %%Sortes EnsembleNumsLignesDeMemeReseau et Set[Ligne] et pred is_in.
  SET[IMPLLIGNEPD **fit** *Elem*↦*Ligne*]
then
op $lignes : EnsembleNumsLignesDeMemeReseau \rightarrow Set[Ligne]$;
vars $l : Ligne;\ nlr : NumLigneDeReseau;\ snlr : EnsembleNumsLigneDeMemeReseau$;
axioms $l\ is\_in\ lignes(snlr) \Rightarrow \exists\, nlr : NumLigneDeReseau \bullet nlr\ is\_in\ snlr \wedge ligne(nlr) = l$;
  $nlr\ is\_in\ snlr \Rightarrow ligne(nlr)\ is\_in\ lignes(snlr)$;
  $ligne(nlr)\ is\_in\ lignes(snlr) \wedge 2nd\_of\ nlr = 2nd\_of\ snlr$
  $\Rightarrow (1st\_of\ nlr)\ is\_in\ (1st\_of\ snlr)$;
end

---

spec IMPLPAIREENSEMBLELIGNESRESEAUPD =
  %%Sorte Pair[Set[Ligne],Reseau] et ops lignes et 2nd_of.
  PAIR[IMPLENSEMBLELIGNESPD **fit** *S1*↦*Set*[*Ligne*]][IMPLENSEMBLELIGNESPD **fit** *S2*↦*Reseau*]
  with *1st_of__*↦*lignes*
  and
  %%Sorte Pair[Ligne,Reseau] et ops ligne et 2nd_of.
  PAIR[IMPLLIGNEPD **fit** *S1*↦*Ligne*][IMPLLIGNEPD **fit** *S2*↦*Reseau*]
  with *1st_of__*↦*ligne*
then
sort $NumLigneDeReseau < Pair[Ligne, Reseau]$;
  $EnsembleNumsLignesDeMemeReseau < Pair[Set[Ligne], Reseau]$;
pred $\_\_is\_in\_\_ : Pair[Ligne, Reseau] * Pair[Set[Ligne], Reseau]$;
vars $p1 : Pair[Ligne, Reseau];\ p2 : Pair[Set[Ligne], Reseau]$;
axiom %%Defn. is_in:Pair[Ligne,Reseau]*Pair[Set[Ligne],Reseau].
  $p1\ is\_in\ p2 \Leftrightarrow (ligne(p1)\ is\_in\ lignes(p2) \wedge 2nd\_of\ p1 = 2nd\_of\ p2)$;
end

---

spec PRE-ABSTRACT2PD' =
  IMPLPAIREENSEMBLESOMMETSRESEAUPD
  and
  IMPLPAIREENSEMBLELIGNESRESEAUPD