Une méthodologie de spécification et de validation de systèmes hétérogènes fondée sur un modèle de contrats pour la conception des systèmes embarqués

Yann Glouche

N° d'ordre : 4016 ANNÉE 2009

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

Ecole doctorale MATISSE

présentée par

Yann Glouche

préparée à l'unité de recherche 6074 IRISA
Institut de Recherche en Informatique et Systèmes Aléatoires
IFSIC

Une méthodologie de spécification et de validation de systèmes hétérogènes fondée sur un modèle de contrats pour la conception des systèmes embarqués

Thèse soutenue à Rennes
le 10 Décembre 2009

devant le jury composé de :

Jean-Marc JEZEQUEL
Professeur à l'université de Rennes I/ président

Robert DE SIMONE
Directeur de recherche INRIA / rapporteur

Frédéric BONIOL
Professeur associé ENSEEIHT / rapporteur

Marc PANTEL
Maître de conférence INPT & ENSEEIHT/ examinateur

Thierry GAUTIER
Chargé de recherche INRIA / examinateur

Jean-Pierre TALPIN
Directeur de recherche INRIA / directeur de thèse

A mes parents,
à Vanessa.

Remerciements

Je remercie Jean-Marc JÉZÉQUEL, professeur à l'université de Rennes I, qui me fait l'honneur de présider ce jury.

Je remercie Robert DE SIMONE, directeur de recherche INRIA de Sophia-Antipolis, et Frédéric BONIOL, professeur associé à l'ENSEEIHT et ingénieur de recherche à l'ONERA, d'avoir bien voulu accepter la charge de rapporteur.

Je remercie Marc PANTEL, maître de conférences à l'INPT et à l'ENSEEIHT d'avoir bien voulu juger ce travail. Je remercie Thierry GAUTIER, chargé de recherche au centre INRIA de Rennes, d'avoir bien voulu juger ce travail, et de m'avoir constamment aidé à parfaire la rédaction de mes articles et de mes rapports durant ces trois années de doctorat.

Je remercie enfin Jean-Pierre TALPIN, directeur de recherche au centre INRIA de Rennes, qui a dirigé ma thèse, pour m'avoir accueilli dans l'équipe ESPRESSO, et pour m'avoir offert l'opportunité de travailler sur un sujet à la fois intéressant et plein de défis. Je tiens à te remercier tout particulièrement pour la confiance que tu m'as accordée.

Tous les membres de l'équipe ESPRESSO pour leur soutien amical et permanent, ainsi que pour le plaisir que j'ai eu à travailler avec eux. Je remercie en particulier Paul pour sa collaboration fructueuse, et Loïc pour ses nombreux conseils techniques.

Table des matières

Introduction

Cette thèse vise à élaborer une méthode de description et de validation des architectures embarquées à base de composants. Cette méthode repose sur la définition de concepts permettant la définition du contexte d'exécution de chaque composant ainsi que les propriétés devant être satisfaites par ce composant. La modélisation des hypothèses faites sur l'environnement et des garanties offertes par un composant correspond à la définition du concept de contrat. En outre, cette approche offre notamment la possibilité de spécifier les aspects temporels. Ce modèle sera ensuite utilisé pour définir le système de typage d'un langage de modules, et ainsi élaborer une méthodologie de conception de systèmes spécifiés avec le langage SIGNAL développé à l'IRISA dans l'équipe ESPRESSO.

D'importants secteurs industriels tels que l'aérospatiale, le transport ou encore les réseaux de télécommunication utilisent de plus en plus des outils informatiques dans le but de développer et concevoir des composants logiciels ou physiques interagissant avec leur environnement. La plupart des systèmes tels que les satellites, les trains, ou les téléphones mobiles sont construits à partir d'un ensemble de modules communiquant entre eux. Chacun de ces composants évolue en fonction de son contexte d'exécution et a un comportement qui lui est propre. L'interaction entre l'environnement et le composant est permanente. On parle de "systèmes embarqués" pour désigner les systèmes informatiques consacrés à des tâches spécifiques incluant souvent des contraintes de temps de calcul. Ces systèmes s'intègrent dans des dispositifs complets englobant souvent des parties électroniques, logicielles, ou mécaniques.

Les systèmes réactifs temps réel

Les systèmes générant un flot d'informations de sortie à partir d'un flot d'informations d'entrée, à une cadence fixée par l'environnement sont appelés des "systèmes réactifs" [HP85]. Dans le cas des applications dites "temps réel" [Ber89], un ensemble de contraintes temporelles telles que le temps de réponse à une modification de l'environnement, ou l'échantillonnage d'une donnée transmise par un capteur, devra être pris en considération.

La complexité des algorithmes et la distribution de calculs complexes sur des architectures hétérogènes assignent une place essentielle aux méthodes de vérification durant la phase de conception. En effet, les propriétés attendues dans une spécification de haut niveau doivent toujours être garanties lors de la mise en œuvre d'un composant. Cette vérification ne peut être laissée à l'attention du programmeur transcrivant des propriétés critiques en langage bas niveau. De plus, la maintenance et l'évolution de ces architectures sont délicates dans de telles circonstances. La conception de ces systèmes, ainsi que la preuve de leur correction sont des tâches extrêmement difficiles, sans compter que la moindre erreur peut avoir des conséquences très graves sur l'environnement dans le cas de "systèmes critiques". Il est donc crucial de pouvoir recourir à des outils s'appuyant sur des méthodes fiables pour aider au développement de tels systèmes. Cela constitue un défi majeur et l'un des objectifs les plus importants de l'informatique fondamentale et appliquée.

Ces dernières années, les méthodes de vérification formelle ont connu un développement spectaculaire, d'un point de vue théorique et applicatif. Un problème apparenté à la vérification est celui de la synthèse : il s'agit de construire un système à partir d'une spécification donnée. Ainsi, la synthèse satisfait par construction sa spécification, ce qui est un avantage pour la sûreté de ces système. Une méthodologie de construction efficace, basée sur des méthodes formelles, minimise le rôle de l'étape de vérification.

Le recours à des domaines fondamentaux pour la synthèse et la vérification de systèmes, tels que les automates ou la logique, bénéficiant de plusieurs siècles de recherche, permet de tirer profit de résultats théoriques puissants et de techniques algorithmiques efficaces.

La modélisation du temps

Le choix d'une représentation du temps adaptée peut simplifier le développement de systèmes réactifs temps réel. En effet, l'analyse de certaines propriétés temporelles est facilitée par une approche du temps judicieuse. On distingue en ce sens deux modèles majeurs : le modèle *asynchrone*, et le modèle *synchrone*.

Approche asynchrone

L'approche asynchrone permet de résoudre les problèmes sans faire référence à un temps logique. En effet, les programmes asynchrones ne sont pas soumis à la cadence d'une horloge. Un programme (ou processus) est constitué d'un nombre fini de tâches, s'exécutant de manière concurrente pour accomplir la mission attribuée au processus.

La durée d'exécution physique d'un processus est fortement influencée par plusieurs facteurs, tels que la politique d'ordonnancement des tâches ou les performances du processeur d'exécution. Ceci induit un indéterminisme, puisque la durée d'exécution physique n'est pas connue. Néanmoins, il est possible de considérer des échéances pour lesquelles un dépassement peut être toléré selon le niveau critique de la tâche.

Dans le cas des systèmes asynchrones : le temps est continu car une sortie peut être lue n'importe quand.

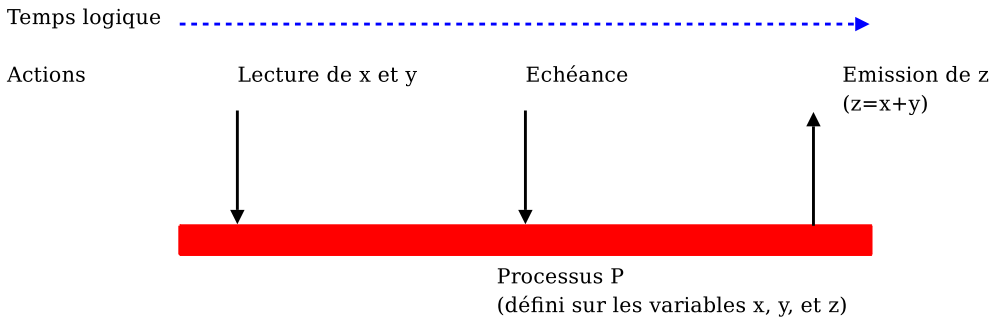


FIG. 1 – Processus asynchrone

Sur la figure 1, le processus P possède deux variables d'entrée x et y, ainsi qu'une variable de sortie z. Les instants de lecture des variables x et y sont bornés par une échéance.

Approche synchrone

Les systèmes synchrones [Hal98] sont cadencés par une horloge, permettant de déterminer, dans un référentiel temporel logique, les instants où un processus peut lire ses entrées, les instants où il émet des sorties, ainsi que le temps logique nécessaire à la réalisation des calculs du processus.

Les langages réactifs basés sur le modèle synchrone font les hypothèses fortes que :

- les communications sont instantanées,
- les opérations simples sont effectuées dans un temps logique nul.

Dans le modèle synchrone, un programme est représenté par une séquence ininterrompue d'événements logiques appelés *tops* d'horloge, auxquels sont associées une ou plusieurs opérations. Notons que lors d'un instant, les programmes synchrones parallélisent plusieurs opérations. Les systèmes modélisés avec l'approche synchrone sont totalement déterministes.

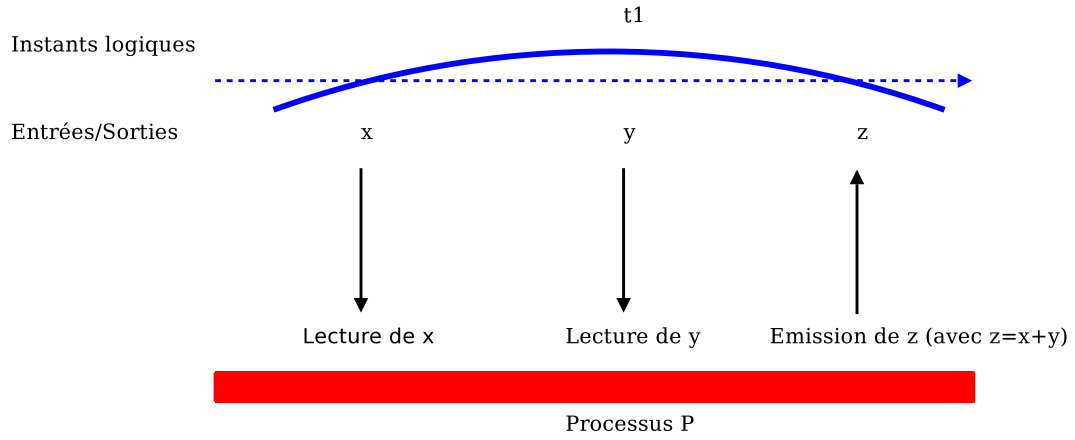


FIG. 2 – Processus synchrone

Sur la figure 2, le processus P possède deux variables d'entrée x et y, ainsi qu'une variable de sortie z. Les instants de lecture et d'émission des variables x, y et z sont connus. L'opération $x + y$ s'effectue dans un temps nul.

Les hypothèses du modèle synchrone sont vérifiées si l'on considère que la plate-forme d'exécution est suffisamment rapide pour réagir à des valeurs d'entrée et émettre les sorties correspondant aux tops d'horloge spécifiés. Il est donc essentiel de savoir si une implémentation matérielle d'un processus permet de satisfaire les hypothèses synchrones. Notons la possibilité d'utiliser des langages à flots de données pour modéliser de telles implémentations.

Parmi les modèles de spécification synchrones, le modèle multi-horloge ou polychrone se distingue par le fait qu'il permet de décrire des systèmes où chaque composant peut avoir sa propre horloge d'activation. Outre la validation formelle, il favorise l'approche orientée composant et le développement modulaire de systèmes à grande échelle. Le modèle d'exécution retenu dans le cadre de nos travaux s'inspire du modèle polychrone.

Contexte

Le projet ESPRESSO a pour objectif de définir des méthodologies et des outils de conception offrant un haut niveau de fiabilité dans le contexte de l'ingénierie des systèmes embarqués. Il s'appuie sur l'approche synchrone et plus généralement sur des bases mathématiques garantissant une fiabilité maximale. Les méthodes de spécification définies dans le cadre du projet s'appuient sur la réutilisation et le raffinement de composants réactifs mis en œuvre. Ces méthodes s'appliquent à des types d'architectures divers allant des circuits aux systèmes répartis.

L'équipe ESPRESSO développe une plate-forme d'aide à la conception de systèmes embarqués : POLYCHRONY. Cet outil vise à être principalement utilisé en avionique pour la conception de logiciels concurrents, l'exploration d'architecture, la simulation et la vérification. POLYCHRONY offre également un éditeur graphique, pour la génération de programmes écrits dans le langage SIGNAL.

Une méthodologie de conception possible consiste à raffiner de manière itérative les spécifications de chaque composant, jusqu'à leur implémentation, partant d'une description partielle de leur comportement et de leur interface avec l'environnement, à l'aide de propriétés d'abstraction.

Buts et motivations

Les travaux développés dans cette thèse s'inscrivent dans ce contexte technologique et visent à équiper POLYCHRONY d'un outillage formel, afin d'y intégrer un système répondant aux objectifs suivants :

- Modularité : la capacité de tester à tout moment la conformité entre la mise en œuvre d'un composant (implémentant une fonctionnalité logicielle ou décrivant un service prédéfini) et son interface (spécificiant ses contraintes de synchronisation et d'ordonnancement).
- Substituabilité : la capacité de vérifier la conformité des exigences globales du logiciel (contraintes de synchronisation et d'ordonnancement) lors du remplacement d'un composant par un autre.
- Compositionnalité : la capacité de vérifier l'adéquation entre le modèle d'une application et celui de son architecture d'exécution au regard d'exigences globales de déterminisme et d'absence de blocage.

Afin d'y parvenir, nous allons nous intéresser à mettre en œuvre les notions sémantiques, les moyens analytiques et les solutions algorithmiques étayant une méthodologie de conception dirigée par les modèles adaptée au calcul formel polychrone. La méthode devra intégrer la vérification des propriétés de chaque composant du système, la validation de chacune des interfaces par rapport au comportement de l'environnement.

Approche proposée

La méthode retenue ici est l'utilisation d'un concept de contrat défini par un couple hypothèses/garanties. Les hypothèses représentent les propriétés devant être satisfaites par le contexte d'exécution, pour qu'un composant puisse satisfaire les propriétés définies par les garanties. Les contrats seront utilisés pour la conception de processus SIGNAL. Plusieurs extensions de langages orientés objets tels que Java [LBR99] et C++ [DI99] intègrent une méthodologie de conception

basée sur les contrats. Le concept de contrat sera utilisé ici pour étendre le langage SIGNAL, afin d'opérer avec un système de types fondé sur un raisonnement basé sur les notions d'hypothèses/garanties.

Les contrats basés sur les notions d'hypothèses/garanties constituent un paradigme expressif pour une conception modulaire et compositionnelle de spécification de programmes. Ils sont devenus un concept important dans certains procédés employés par les outils de conception assistée par ordinateur, pour la conception de systèmes informatiques.

Nous élaborons ici les fondements pour la conception de systèmes embarqués basée sur la notion de contrats. Nous proposons ainsi une algèbre de contrats basés sur les hypothèses/garanties axée sur deux concepts simples :

- les hypothèses et les garanties des composants sont définies par des ensembles de processus satisfaisant une même contrainte que nous appellerons *filtres*.
- les filtres sont caractérisés par une structure d'algèbre booléenne.

Les choix effectués pour définir la structure des filtres permettent de définir une algèbre de Heyting sur l'ensemble des contrats. Cette structure riche caractérisant l'ensemble des contrats facilite les opérations d'abstraction, de raffinement, de composition et de normalisation des contrats. Un cadre de travail est ainsi défini, dans lequel les contrats sont utilisés pour vérifier la correction des hypothèses faites sur la définition d'un composant par son contexte d'utilisation, et pour fournir à l'environnement les garanties qui lui sont demandées.

Notre algèbre de contrats va permettre la définition d'un système de modules dont le paradigme de typage est basé sur la notion de contrats. Le type d'un module est un contrat exploitant les hypothèses et les garanties portant sur ses comportements. Ce système de types permet d'associer un module avec une interface qui peut être utilisée dans des variétés de scénarios tels que la vérification de la composabilité de modules, ou de supporter de manière efficace la compilation modulaire.

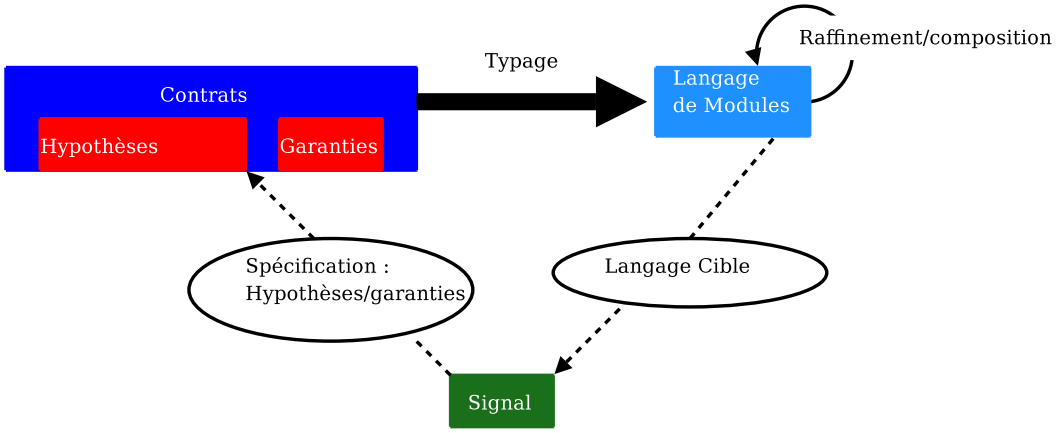


FIG. 3 – Raffinement de contrats spécifiés en SIGNAL.

Le modèle de contrats que nous avons développé peut s'adapter à n'importe quel langage, qu'il s'agisse de langage fonctionnel, orienté objets, ou flot de données. Cependant, le modèle reposant principalement sur la notion de comportement, la mise en œuvre de ce procédé est facilitée lorsque les spécifications ou les implémentations sont décrites avec des langages reposant eux-mêmes sur la notion de comportement. Il en va de même pour l'application du langage de modules comme extension d'un autre langage.

Bien que dans le cas d'étude présenté dans le chapitre 5 de cette thèse, nous avons fait le choix d'employer ce système pour vérifier des implémentations de processus décrites en SIGNAL, avec des spécifications de contrats elles-mêmes décrites en SIGNAL (voir figure [3]), la généricité de l'approche permet de ne pas nécessairement employer le même langage pour décrire les contrats et les processus à vérifier.

Organisation du document

Ce document comprend deux parties. La première présente les avantages et les inconvénients de méthodes et de modèles existants pour décrire les spécifications de systèmes temps réel, à partir desquels nous développons ensuite notre propre modèle. La seconde aborde en particulier l'application du modèle de contrats retenu pour définir le système de types d'un langage de modules. L'apport principal de cette thèse se situe dans le chapitre 2, où nous définissons un modèle de contrat, ainsi que dans le chapitre 4, où nous élaborons la sémantique d'un langage de modules basé sur notre modèle de contrats.

Partie I

Cette partie est consacrée à la présentation des méthodes de modélisation des propriétés des systèmes temps réel.

Chapitre 1. Le premier chapitre présente différentes techniques de modélisation existantes utilisées pour la spécification de différents aspects de la conception des architectures temps réel. Nous insistons notamment sur certaines méthodes centrées sur le concept des automates, permettant la modélisation des propriétés relatives à l'aspect temporel, ainsi que les techniques basées sur la théorie des ensembles de comportements. Nous n'oublions pas d'aborder la sémantique de langages caractéristiques incluant le raisonnement sur les pré/post-conditions. Ceci permettra de positionner notre méthodologie de conception par rapport aux travaux existants.

Chapitre 2. Le chapitre 2 est consacré à la présentation du modèle de contrat retenu pour la description des propriétés des composants. Nous débutons sur la définition des comportements permettant de définir les processus dans la sémantique du modèle. Nous développons ensuite la notion de filtre. Un filtre est un ensemble de processus satisfaisant une même contrainte. Cette structure de filtres est utilisée pour définir la structure des contrats. Dans cette structure, les contrats sont représentés par une paire de filtres dont l'un est utilisé pour décrire les hypothèses faites sur l'environnement d'exécution d'un composant, et l'autre pour décrire les garanties associées à ce composant. Nous étudierons des relations d'ordre et de composition définies pour chacune de ces structures ainsi que les propriétés qui en découlent.

Partie II

L'objet de cette partie est d'une part, de définir la sémantique d'un langage de modules basé sur la notion de contrat, et d'autre part, d'étendre SIGNAL avec ce langage de modules, pour intégrer le concept de contrat dans POLYCHRONY.

Chapitre 3. Le troisième chapitre présente le langage SIGNAL utilisé dans l'environnement POLYCHRONY pour la description de processus. Nous verrons la signification des principaux opérateurs dont la composition de processus, et l'écriture de processus simples, ainsi que les modèles de comportements et de marques sur lesquels reposent la sémantique du langage.

Chapitre 4. Le chapitre 4 décrit la sémantique d'un langage de modules dont le système de types est basé sur la notion de contrat. En particulier, nous ver-

rons comment les contrats sont utilisés pour définir le type des modules. D'autre part, la relation de raffinement définie sur l'ensemble des contrats permettra de définir facilement une relation de sous-typage de modules fondée sur le concept de substitution de modules.

Chapitre 5. Le cinquième et dernier chapitre présente un cas d'utilisation du modèle de contrat et du langage de modules. Le langage SIGNAL est utilisé pour définir les contrats caractérisant les signatures de modules, ainsi que les modules caractérisant les implémentations de processus. Nous illustrons avec cet exemple la simplicité d'appliquer notre langage de modules à un langage de programmation de systèmes temps réel comme SIGNAL, dont la sémantique est basée sur la théorie des traces. Les notions de substituabilité et de satisfaction de spécifications de processus SIGNAL sont directement induites par les relations de raffinement et de satisfaction définies sur la structure des contrats.

Première partie

Les modèles de contrats

Chapitre 1

Un tour d'horizon de la formulation et de l'utilisation des contrats

La programmation par contrats est préconisée depuis longtemps en génie logiciel, et appliquée principalement aux langages orientés objets [Mey97]. En particuler, l'approche présentée dans [AL93, Hoa69] permet le calcul du contrat d'une architecture à partir de relations simples dans le cas où les composants ne partagent pas des entrées ou des sorties.

Plan Nous présentons tout d'abord (section 1.1) deux extensions du langage Java intégrant la notion de contrat. Plus proches de nos besoins, les méthodes de spécification présentées dans la section 1.2 permettent l'expression de propriétés temporelles. Nous considérons ainsi une approche basée sur la théorie des automates, une représentation dénotationnelle fondée sur la théorie des traces, et un modèle fonctionnel.

1.1 L'approche par contrats dans les langages orientés objets

La conception par contrats présentée dans [MMM02] est intéressante pour les langages comme C++ ou Java. Dans cette approche, les contrats basés sur les propositions expriment les invariants du programme, les pré- et post-conditions, par des expressions booléennes qui doivent être vérifiées lorsque le contrat est validé.

En programmation orientée objet, l'idée de base de la conception par contrat est de considérer les services fournis par une classe comme un pacte entre la classe et son interlocuteur. Le contrat est composé de deux parties : les exigences formulées par la classe envers son interlocuteur et les promesses faites par la classe à

son interlocuteur. Les *hypothèses* spécifient les propriétés qui doivent être remplies par le contexte d'exécution afin que le composant puisse remplir ses *garanties*. Dans [BJPW99] plusieurs niveaux de contrats sont distingués : le niveau statique (par exemple : les types des paramètres et de retour des méthodes), le niveau comportemental basé sur les pré- et post-conditions, la synchronisation, la qualité de service.

1.1.1 JML

JML [LBR99] est une extension de Java intégrant les notions de pré-condition et post-condition. Le formalisme d'expression des contrats est très proche de Java. Les pré-conditions définissent les hypothèses devant être satisfaites par les paramètres pour pouvoir appeller une méthode, et les méthodes doivent satisfaire les post-conditions.

Sur la figure 1.1, la méthode `sqrt` calculant la racine carrée de son paramètre `x` ne peut être appelée que si la valeur de `x` est supérieure ou égale à 0. Dans ce cas, la méthode `sqrt` garantit que le carré du résultat est égal à `x` modulo une valeur d'approximation `eps`.

```
public class SqrtExample {
  public final static double eps = 0.0001;
    /* Pré-condition */
  //@ requires x >= 0.0;
    /* Post-condition */
  //@ ensures JMLDouble.approximatelyEqualTo(x,\result * \result,eps);
    /* Méthode */
  public static double sqrt(double x) {
    return Math.sqrt(x);
  }
}
```

FIG. 1.1 – Pré- et post- conditions JML pour une méthode écrite en Java.

JML permet de vérifier la vérification des pré-conditions et des post-conditions lors de chaque appel de la méthode. En effet, lorsque la méthode est appelée, JML vérifie la satisfaction des pré-conditions, puis la méthode est exécuté, et enfin si la méthode n'a pas généré d'exceptions, JML teste la vérification des post-conditions.

1.1.2 JASS

Le système JASS [BFMW01] est une extension de Java un peu plus proche de nos motivations. Il permet d'associer des pré/post-conditions aux méthodes d'une classe, ainsi que de définir des invariants de classes. Il offre la possibilité de faire reférence à un état précédent l'état courant dans l'expression d'une propriété. En effet, JASS repose sur la notion de *comportement*. De manière plus précise, chaque objet est associé à une séquence d'états. JASS s'oriente principalement vers le débogage et la génération de code défensif.

Comme dans JML, la notion d'agent avec des entrées/sorties n'existe pas dans JASS. Le langage est basé sur les invariants de classe, ainsi que les pré/post-conditions associées aux méthodes.

La classe `Buffer` présentée sur la figure 1.2, permet d'instancier des files d'attentes (FIFO) de tailles finies. La méthode `add` est utilisée pour ajouter un objet dans la file. Les objets placés dans la file peuvent être récupérés avec la méthode `remove`. Cette méthode retourne l'objet situé à la fin de la file d'attente, puis le supprime de la file d'attente.

```
public class Buffer implements Cloneable {
private int in,out,count;
private Object[] store;
public Buffer (int capacity) { ... }
public boolean empty() { ... }
public boolean full() { ... }
public int capacity() { ... }
public void add(Object o) { ... }
public Object remove() { ... }
public boolean contains(Object o) { ... }
...
}
```

FIG. 1.2 – La classe `Buffer`.

La figure 1.3 présente une pré-condition et une post-condition associées à la méthode `remove`. Ainsi, `remove` ne peut pas retourner un objet si la file est vide. De plus, `remove` garantit que l'objet retourné était auparavant placé dans la file. Tout comme JML, JASS intègre l'utilisation de la variable `Old` représentant l'état de la file d'attente, précédant l'appel de la méthode `remove`.

```
public Object remove() {
/* Pré-condition */
/** require !empty(); **/
...
return o;
/* Post-condition */
/** ensure Old.contains(Result); **/
}
```

FIG. 1.3 – Pré-condition et post-condition de la méthode `remove`.

1.1.3 Encapsulation de processus

Nous nous intéressons ici aux langages permettant une représentation des composants temps réel reposant sur une approche objet ou modulaire. L'encapsulation de processus dans des classes est particulièrmenent intéressante puisqu'il existe déjà des langages orientés objets intégrant le raisonnement par contrats.

Ainsi, le langage Synergy [BPS06] combine deux paradigmes :

- la modélisation orientée objets pour la conception,
- l'exécution synchrone pour une modélisation précise de comportements réactifs.

La classe `Signals` présentée sur la figure 1.4 est écrite en Synergy. Elle permet l'instanciation d'objets réactifs. Ces objets communiquent avec l'environnement grâce à l'entrée `sensor` et la sortie `actuator`. Les entrées et les sorties sont associées à une valeur, ou bien elles sont absentes. A chaque fois que `sensor` reçoit un entier, la même valeur augmentée de 1 est envoyée sur `actuator`.

```
class Signals {
    Sensor<int> sensor = new Sensor<int>(new SimInput());
    Signal<int> actuator = new Signal<int>(new SimOutput());
    public Signals() {
        active {
            if (?sensor) { emit actuator($sensor + 1); };
        };
    };
}
```

FIG. 1.4 – Une classe réactive.

Dans [KT04], une méthode d'encapsulation basée sur le paradigme orienté objets et l'héritage de comportements pour la description des modèles synchrones

est proposée. [NTGLG97] décrit un langage de modules pour la décomposition des structures complexes synchrones en modules paramétrables. Ces travaux sont destinés à définir un cadre de raisonnement formel pour encourager la réutilisabilité des composants et permettre l'emploi de processus compilés séparément à l'intérieur d'une même architecture. Il devient ainsi essentiel de pouvoir typer un processus pour pouvoir vérifier l'adéquation entre un composant et son environnement d'exécution. Le système de typage présenté dans [NTGLG97] propose de définir le type des processus à partir du graphe de flot de données. Ce système de types est défini par un environnement de typage, et un ensemble de contraintes de sous-typage. Avec cette méthode de typage, il devient alors possible de paramétrer des modules par des processus.

1.2 Modélisation de propriétés temporelles

Dans le contexte de l'ingénierie logicielle, la programmation par contrats est adaptée pour une grande variété de langages et de formalismes, mais la notion centrale de temps et/ou de trace, nécessaire à la conception de systèmes réactifs, n'est pas toujours prise en compte. Par exemple, des extensions de OCL avec l'utilisation d'une logique temporelle linéaire ont été proposées dans [ZG02, FM01], en mettant l'accent sur l'expressivité du langage de contraintes (la manière dont les contraintes expriment les propriétés des classes et des objets). D'autre part, ces deux extensions de OCL ne considèrent pas l'aspect flot de données. Toutefois, il est intéressant de remarquer que dans [FM01], les propriétés temporelles sont exprimées au moyen de CTL, ce qui offre la possibilité d'exprimer des propriétés de vivacité.

Considérons un composant `buffer` qui reçoit des données arrivant de manière périodique. La taille de cette période est de 100 unités de temps logique. Ce composant ne peut lire une donnée que dans l'état `Loading`. Ainsi l'état `Loading` doit toujours être atteignable dans les 100 prochaines unités de temps. La figure 1.5 montre l'écriture de cette propriété dans la syntaxe présentée dans [FM01].

```
context Buffer
inv: Loader@post[1,100]->forall(p:OclPath|p->includes(Loading))
```

FIG. 1.5 – Une propriété de vivacité.

D'autre part, les méthodologies de description des systèmes à flots de données ne permettent pas nécessairement l'expression de propriétés temporelles. Ainsi, l'approche fonctionnelle proposée dans [Bro97] propose une notion com-

positonnelle du raffinement afin de simplifier le traitement des langages à flots de données. Elle raisonne sur les types d'entrées-sorties et sur le graphe de causalité des entrées-sorties.

Nous nous intéressons à des techniques de spécification intégrant la notion de temps, et tenant compte de l'aspect concurrent de l'ordre d'exécution des tâches.

1.2.1 Les automates

Dans la théorie des automates d'interface [dAH01], la notion d'interface offre des avantages notamment dans la définition de l'opération de composition. Cette approche rend possible la vérification de la compatibilité des interfaces entre différents modules réactifs. Dans ce contexte, il n'est pas pertinent de séparer les hypothèses des garanties et un seul automate peut être utilisé pour modéliser un module.

Cette approche est décrite avec un exemple considérant l'implémentation d'un service de transmission de messages. Un utilisateur se sert d'un composant possédant une variable d'entrée *msg* utilisée pour envoyer les messages. Lorsqu'un message est envoyé, le composant retourne *ok* (si le message a bien été envoyé) ou $fail$ (si la transmission ne s'est pas déroulée correctement). L'interface du composant est reliée à l'interface d'un canal de communication. La variable *send* permet au composant de transmettre un message au canal de communication. Le canal peut envoyer une valeur *ack* afin d'indiquer que la transmission s'est correctement déroulée, ou bien une valeur *nack* afin d'indiquer que la transmission du message a échoué. Lorsqu'un message est émis sur la variable *msg*, le composant essaie d'envoyer ce message sur le canal de communication, et renvoie ce message si la première transmission a échoué.

L'automate d'interface du composant est illustré sur la figure 1.6. Si les deux transmissions échouent (réception de deux valeurs sur la variable *nack*), le composant indique l'échec de la transmission du message par l'envoi d'une valeur sur la variable $fail$.

Si la transmission a réussi (réception d'une valeur sur la variable *ack*), il signale la réussite pour l'envoi d'une valeur sur la variable *ok*.

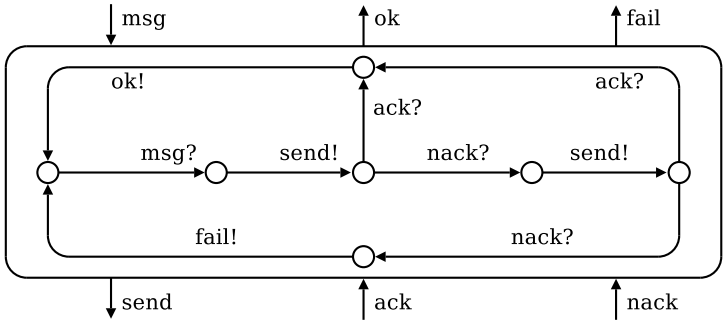


FIG. 1.6 – Automate d'interface du composant.

L'automate d'interface présenté sur la figure 1.7 définit le comportement d'un utilisateur du service de transmission de messages tel que : après avoir émis un message à envoyer, il accepte une valeur de retour sur la variable *ok*, cependant aucune valeur sur la variable $fail$ n'est acceptée. L'utilisateur fait l'hypothèse que la transmission du message n'échouera jamais.

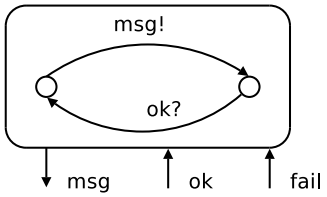


FIG. 1.7 – Automate d'interface d'un utilisateur.

Sur la figure 1.8, la composition de l'automate d'interface de l'utilisateur avec l'automate d'interface du composant est obtenue en considérant les états compatibles. En effet, aucune action n'est associée à la réception d'une valeur *fail* par l'utilisateur (voir figure 1.7), ce qui fait de l'état précédant l'émission d'une valeur sur la variable $fail$ dans l'automate d'interface du composant (voir figure 1.6), un état incompatible avec les états de l'automate de l'utilisateur.

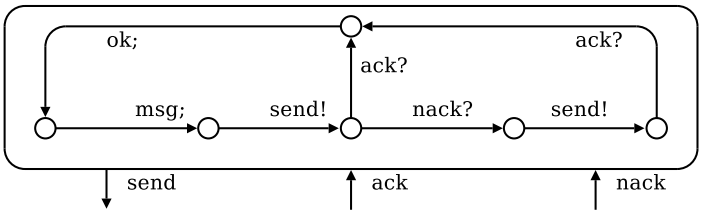


FIG. 1.8 – Composition de l'automate d'interface de l'utilisateur avec l'automate d'interface du composant.

Supposons que le canal de communication réussisse toujours à envoyer un message après deux tentatives. Ainsi, il n'émet jamais la valeur *nack*. L'automate d'interface de ce canal de communication est illustré sur la figure 1.9

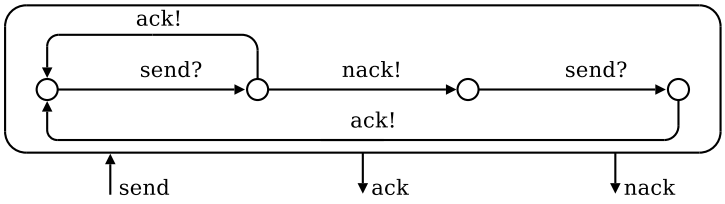


FIG. 1.9 – Canal de communication.

La règle de compatiblité définie dans [dAH01] permet d'assurer que l'utilisateur peut utiliser le composant pour transmettre un message à travers le canal de communication. Soit $User \otimes Comp$, l'automate obtenu par la composition de l'automate de l'utilisateur (voir figure 1.7) avec l'automate du composant (voir figure 1.6). Cette règle vérifie notamment que chaque entrée du produit des automates de l'utilisateur et du composant correspond à une sortie dans l'environnement, et que chaque sortie du produit des automates de l'utilisateur et du composant correspond à une entrée dans l'environnement. Ce composant utilisé dans l'environnement d'exécution décrit sur la figure 1.9, permet de garantir que tous les messages émis par l'utilisateur seront bien transmis.

Dans [LNW07], les auteurs montrent que le cadre de travail des automates d'interface peut être intégré dans celui des automates modaux d'entrée/sortie. [RBB+09] développe cette approche en tenant compte des spécifications modales. Il s'agit de l'étiquetage des transitions *pouvant éventuellement* être exécutées ainsi que de celles qui *doivent obligatoirement* être exécutées. Les spécifications modales sont équipées d'un opérateur de composition parallèle et d'une relation d'ordre impliquant l'existence d'une borne inférieure. L'élément défini par le plus grand des minorants (GLB) permet d'intégrer les notions de vues multiples et d'exigences conjonctives. En se basant sur [BCP07], les auteurs notent la difficulté de manipuler des interfaces définies sur des alphabets différents. Les modalités permettent différentes méthodes d'égalisation des alphabets selon que la composition ou la liaison parallèle est considérée. Par la suite, ils définissent un contrat comme une résiduation G/A (la résiduation est l'adjoint de la composition parallèle), où les hypothèses A et les garanties G sont définies par spécifications modales. Les notions de raffinement, composition, et de vues multiples sont développées. Ce modèle traite soigneusement l'égalisation d'alphabet (ou ensemble de variables de définition). Il est ainsi possible de tester si une implémentation définie sur un alphabet satisfait une spécification modélisée par un automate défini sur un alphabet différent. Cependant, l'utilisation de la borne inférieure des automates modaux satisfait l'approche multi-vues mais pas la notion de substituabilité dans la composition. En effet, la borne inférieure ne satisfait pas que pour quatre spécifications modales $\mathcal{S}_1$, $\mathcal{S}'_1$, $\mathcal{S}_2$, $\mathcal{S}'_2$ telles que $\mathcal{S}'_1$ raffine $\mathcal{S}_1$, et $\mathcal{S}'_2$ raffine $\mathcal{S}_2$, la composition de $\mathcal{S}'_1$ et $\mathcal{S}'_2$ raffine $\mathcal{S}_1$, $\mathcal{S}_2$. Cette propriété, appelée *"raffinement dans la composition"* dans [RBB+09] nécessite l'emploi d'une seconde opération de composition pour être satisfaite.

1.2.2 Un modèle dénotationnel

Dans [BCP07], est défini un système de contrats basés sur la notion d'hypothèses/garanties, avec une vision générique du domaine d'application du modèle. Un contrat est représenté par une paire (hypothèses/garanties) où les hypothèses

et les garanties sont représentées par un ensemble de traces. Ce modèle présente également une relation de raffinement basée sur la notion de substituabilité : si un contrat C' raffine un contrat C alors tous les processus satisfaisant C' satisfont aussi C. Considérons deux contrats ($\mathbf{A}_1$,$\mathbf{G}_1$) et ($\mathbf{A}_2$,$\mathbf{G}_2$). Le contrat ($\mathbf{A}_1$,$\mathbf{G}_1$) raffine ($\mathbf{A}_2$,$\mathbf{G}_2$), si et seulement si $\mathbf{A}_2 \subseteq \mathbf{A}_1$ et $\mathbf{G}_1 \subseteq \mathbf{G}_2$, où $\subseteq$ est l'inclusion d'ensembles. Ainsi ($\mathbf{A}_1$,$\mathbf{G}_1$) raffine ($\mathbf{A}_2$,$\mathbf{G}_2$), si et seulement si les hypothèses $\mathbf{A}_1$ sont définies par un ensemble de traces plus grand que les hypothèses $\mathbf{A}_2$, et les garanties $\mathbf{G}_1$ sont définies par un ensemble de traces moins grand que les garanties $\mathbf{G}_2$.

Une algèbre booléenne de contrats découle de cette relation de raffinement. La borne inférieure de deux contrats permet d'avoir une approche multi-vues sur les composants.

1.2.3 Un modèle fonctionnel

Dans [MM04], une notion de contrat synchrone est proposée pour le langage de programmation Lustre. Dans cette approche, les contrats sont des spécifications exécutables (observateurs synchrones) rythmées par une horloge (l'horloge de l'environnement). Dans ce modèle, les hypothèses sont modélisées par un processus observant les variables d'entrée, et les garanties sont modélisées par un autre processus observant les variables de sortie d'un composant. Cela correspond à une approche qui est satisfaisante pour vérifier la satisfaction des propriétés des différents modules (qui ont une horloge), mais ne peut guère être employée dans le monde de la modélisation des architectures asynchrones (qui possèdent des horloges multiples).

Cette approche est illustrée par l'écriture d'un contrat observant au cours du temps une variable d'entrée a et une variable de sortie b. Ce contrat stipule que : si la valeur de la variable a est toujours $vrai$ durant au moins deux instants consécutifs (sinon la valeur de a est $faux$), alors la valeur de la variable b est $vrai$ durant au moins trois instants consécutifs (sinon la valeur de b est $faux$). Les hypothèses sont modélisées par l'automate présenté sur la figure 1.10, et les garanties par l'automate présenté sur la figure 1.11.

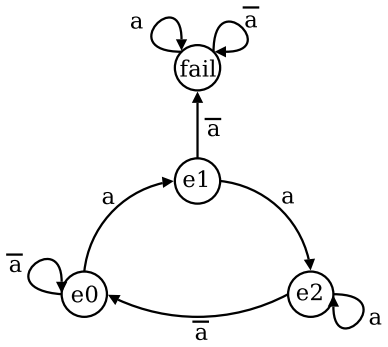


FIG. 1.10 – Hypothèses : a prend la valeur *vrai* durant au moins deux instants.

La séquence des valeurs de la variable d'entrée a satisfait les hypothèses si et seulement si l'automate n'atteint jamais l'état $fail$.

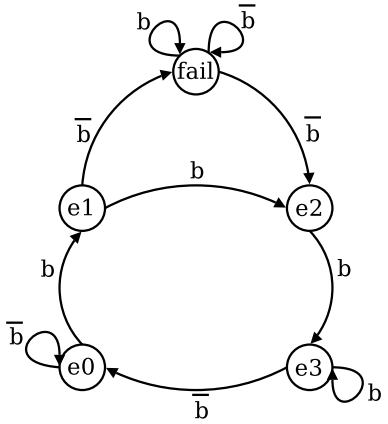


FIG. 1.11 – Garanties : b prend la valeur *vrai* durant au moins trois instants.

La séquence des valeurs de la variable de sortie b satisfait les garanties si et seulement si l'automate n'atteint jamais l'état $fail$.

Ce contrat est satisfait par un composant (voir la figure 1.12) possédant une variable d'entrée a et une variable de sortie b ayant le comportement suivant : lorsque la valeur de a est *vrai* à un instant t, la valeur de b est *vrai* durant les deux instants $t+1$ et $t+2$ suivants. En effet, les comportements du composant satisfont toujours les hypothèses et les garanties du contrat.

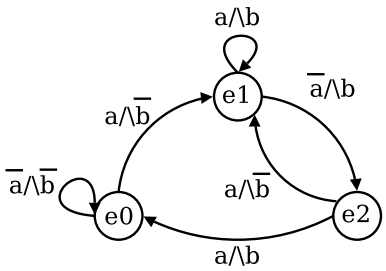


FIG. 1.12 – Comportement d'un composant.

1.3 Conclusion : enseignement à tirer

Nous souhaitons définir un cadre théorique pour le raisonnement basé sur la notion d'hypothèses/garanties. Cette approche offre la possibilité de clairement distinguer les propriétés de l'environnement d'exécution des garanties satisfaites par un composant. Ainsi la compatibilité entre un composant et son contexte d'exécution sera facilement établie. Une représentation des hypothèses et des garanties devra être choisie, ce qui conduira au développement de la notion de filtres : les garanties filtrent les processus satisfaisant une propriété donnée. A partir d'une modélisation des traces et des processus, nous souhaitons construire une algèbre de filtres et une algèbre de contrats.

En choisissant de dissocier les hypothèses faites sur le contexte d'exécution et les garanties offertes par un composant, il devient important de définir une relation définissant la classe d'équivalence des contrats satisfaits par le même ensemble de processus (relation d'équivalence de filtrage). Ainsi, dans notre algèbre de contrats comme dans les automates d'interface, un contrat peut être exprimé avec un seul filtre. La relation d'équivalence de filtrage offre la possibilité d'exprimer un contrat avec un seul filtre portant sur les garanties et les hypothèses par le filtre acceptant tous les processus (ou, inversement, avec un seul filtre pour spécifier les hypothèses, un filtre de garantie n'acceptant aucun processus).

Il en résulte une structure riche devant être :

- générique, dans la façon dont elle peut être mise en œuvre, ou à l'instanciation des modèles de calcul ;
- flexible, dans la façon dont elle peut aider à la structuration et la normalisation des expressions ;
- complète, en ce sens que toutes les combinaisons de propositions peuvent être exprimées dans le modèle.

Il est intéressant de noter que la séparation des points de vue est importante en génie logiciel. La séparation des vues permet plus de souplesse dans la recherche des relations de compatibilité entre les composants. L'aspect multi-vues procure

une meilleure isolation des différents aspects des modules, et favorise donc une approche compositionnelle de la spécification de modules.

Enfin, le modèle de spécification devra être indépendant du modèle d'expression des hypothèses ou des garanties. Ainsi, un langage tel que SIGNAL dont la sémantique est basée sur un modèle dénotationnel pourra être directement utilisé, mais aussi, une logique temporelle compatible avec notre modèle, comme par exemple ATL (Alternating-time temporal logic [AHK02]).

Chapitre 2

Une algèbre de contrats

La plupart des méthodologies de développement utilisées pour définir des architectures embarquées sont basées sur une approche itérative de validation de spécifications. En outre, le développement coopératif de systèmes nécessite l'utilisation et l'assemblage de composants développés par différents fournisseurs, de manière sûre et consistante [ELLSV97, Kop97]. Ces composants doivent fournir des conditions d'utilisation, et offrir des garanties sous ces conditions. Ceci définit une notion de *contrat*. Le concept de contrats peut être utilisé pour la validation de logiciels critiques. Nous optons pour un paradigme de contrats différent afin de définir une technique de validation des composants dans le cadre du modèle synchrone. Dans notre modèle, un composant est représenté par une vue abstraite de ces comportements. Il est défini à travers un ensemble fini de variables d'entrée/sortie communiquant avec son environnement d'exécution. Les comportements peuvent être perçus comme des ensembles de traces d'exécution sur les variables du composant. Le modèle abstrait d'un composant est donc un *processus* défini comme un ensemble de comportements.

Un *contrat* est une paire *(hypothèses, garanties)*. Les *hypothèses* décrivent les propriétés attendues par un composant qui doivent être satisfaites par le contexte (l'environnement) dans lequel le composant est utilisé. Les *garanties* décrivent les propriétés que doit satisfaire le composant lui-même lorsque le contexte satisfait les *hypothèses*. Un contrat peut être une simple documentation, cependant, lorsqu'un modèle formel le permet, les contrats peuvent être soumis à des outils de vérification formelle. Nous souhaitons fournir aux concepteurs un modèle formel, permettant des calculs efficaces sur les contrats. Nous allons présenter une version étendue du cadre algébrique proposé dans [GLTG08] et [GLTG09] permettant un raisonnement formel sur les contrats.

D'une part, les hypothèses et les garanties d'un composant sont définies par la notion de *filtre* : les hypothèses filtrent les processus (ensembles de comportements) qu'un composant peut accepter. Un *filtre* est un ensemble de processus

pour lesquels les comportements des variables d'entrée/sortie sont compatibles avec les propriétés (ou contraintes), exprimées sur les variables du composant. D'autre part, nous définissons une algèbre booléenne pour manipuler les filtres. Ce qui nous amène à définir une structure algébrique permettant de raisonner sur les contrats pour abstraire, raffiner, combiner et normaliser un composant. Ce modèle algébrique est basé sur un modèle minimaliste de traces d'exécution, permettant de s'adapter facilement à un cadre de développement particulier.

Une caractéristique de ce modèle est qu'il permet de gérer de façon précise les variables du composant et leurs comportements possibles. Ceci est un point clé. En effet, les hypothèses et les garanties sont exprimées par des propriétés contraignant les comportements de certaines variables. C'est la raison pour laquelle nous introduisons un ordre partiel sur les processus et les filtres. De plus, avoir une algèbre booléenne sur les filtres permet de formaliser sans aucune ambiguïté l'expression du complémentaire dans l'algèbre. Ce qui est un réel avantage comparé à d'autres modèles et formalismes.

Plan Ce chapitre est organisé de la manière suivante. La section 2.1 introduit une algèbre générale de processus, lequel empreinte ses notations et ses concepts à la théorie des domaines [AJ87]. Un contrat (**A**,**G**) est vu comme une paire de composants logiques filtrant les processus : les hypothèses **A** filtrent les processus à sélectionner (accepter ou inversement rejetés), qui sont autorisés (acceptés ou inversement rejetés) par les garanties **G**. Les filtres sont définis dans la section 2.2 et les contrats dans la section 2.3.

2.1 Une algèbre de processus

Nous définissons une algèbre pour les comportements et les processus. Généralement, un comportement décrit la trace d'un processus défini dans un domaine discret (une trace de Marzurkiewicz [Maz89] ou un tuple de signaux dans le modèle de marque de Lee [LSV98]). Nous avons délibérément choisi un modèle de définition abstrait pour encapsuler des variables associées à des domaines de définition divers. Ainsi les variables peuvent représenter des séquences de booléens, entiers, réels mais aussi des comportements de systèmes plus complexes tels que des systèmes hybrides. Il est également possible qu'elles représentent simplement des "comportements" caractérisés par des valeurs scalaires, afin de représenter des coûts d'exécution, des tailles de mémoires, etc. Dans cette partie, nous nous intéressons aux processus modélisés par un ensemble de comportements.

Nous faisons le choix de centrer la représentation des processus sur la théorie des comportements afin de faciliter l'application du modèle de contrats proposé à des langages eux-mêmes basés sur cette théorie. La sémantique des comporte-

ments et des processus est particulièrement proche de celle de SIGNAL.

2.1.1 Sémantique des comportements

Définition 2.1 (Comportement) *Soit $\mathcal{V}$ un ensemble infini, dénombrable de variables, et $\mathcal{D}$ un ensemble de valeurs ; pour $\mathbf{Y}$, un ensemble fini, et non vide, de variables inclus dans $\mathcal{V}$ (noté $\mathbf{Y} \subset \mathcal{V}$), un $\mathbf{Y}$-comportements défini sur un ensemble de variables $\mathbf{Y}$ est une fonction $b : \mathbf{Y} \to \mathcal{D}$.*

L'ensemble des $\mathbf{Y}$-comportements représentant l'ensemble des comportements définis sur l'ensemble de variables $\mathbf{Y}$, est noté $\mathbb{B}_{\mathbf{Y}} =_{\Delta} \mathbf{Y} \to \mathcal{D}$. La définition 2.1 est étendue aux comportements définis sur l'ensemble vide de variables : $\mathbb{B}_{\emptyset} =_{\Delta} \emptyset$ (il n'y a pas de comportements associé à l'ensemble vide de variables).

Soit $\mathbf{Y}$, un ensemble fini, et non vide, de variables inclus dans $\mathcal{V}$, $\mathbf{Y}$ non vide, c un comportement défini sur $\mathbf{Y}$, $\mathbf{X}$ un sous-ensemble (éventuellement vide) de $\mathbf{Y}$. $c_{|\mathbf{X}}$ est la restriction de c sur $\mathbf{X}$ telle que

$$c_{|\mathbf{X}} = \{(x, c(x)) \mid x \in \mathbf{X}\}, \quad \text{et} \quad c_{|\emptyset} = \emptyset$$

On en déduit alors que $c_{|\mathbf{Y}} = c$.

Sur la figure 2.1, les comportements b_1 et b_2 définis sur les variables x et y sont des fonctions de l'ensemble de variables $\{x, y\}$ vers des fonctions définissant les signaux. Le comportement b_1 est une fonction d'un ensemble d'instants discrets représentés par des entiers naturels vers l'ensemble des rationnels (utilisé ici comme domaine de valeurs). Le comportement b_2 associe x, et y à des fonctions continues du temps.

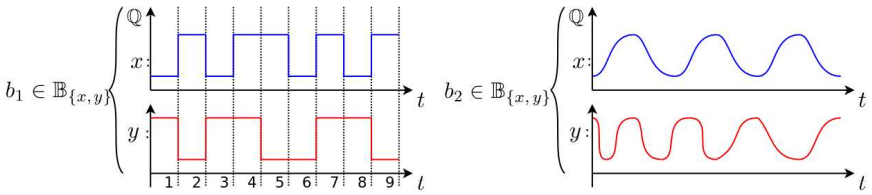


FIG. 2.1 – Exemples de comportements.

Nous définissons un *processus* comme un ensemble de comportements défini sur un ensemble de variables donné.

2.1.2 Sémantique des processus

Nous définissions un *processus* comme un ensemble de comportements sur un ensemble fini de variables. Par exemple, sur la figure 2.2, le processus représenté, possède b_1 pour comportement éventuel, mais peut également avoir b_2, b_3 comme comportements autorisés.

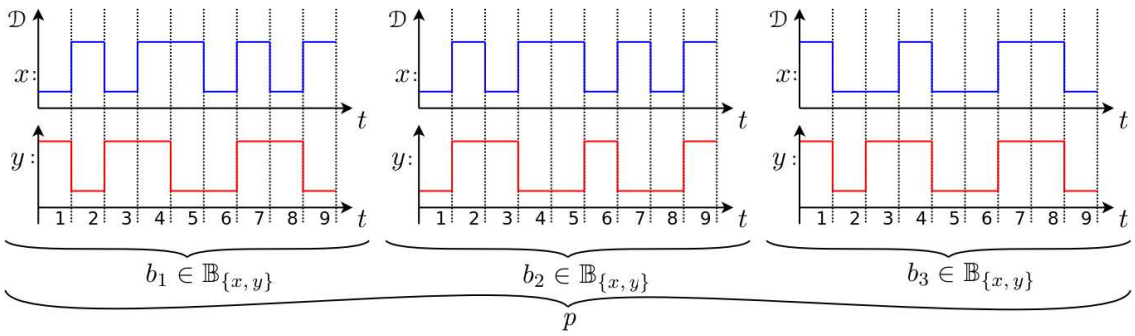


FIG. 2.2 – Exemple de processus.

Définition 2.2 (Processus) *Soit* $\mathbf{X}$ *un ensemble fini de variables (*$\mathbf{X} \subset \mathcal{V}$*). Un* $\mathbf{X}$-processus p *est un ensemble non vide de comportements définis sur* $\mathbf{X}$.

Ainsi, de la même manière que $\mathbb{B}_\emptyset =_\Delta \emptyset$, il y a un unique $\emptyset$-processus défini par $\Omega =_\Delta \{\emptyset\}$; Ω contient uniquement le comportement vide. Le *processus vide* est noté $\mho =_\Delta \emptyset$.

Ω est défini sur un ensemble vide de variables, il n'a par conséquent aucun effet lorsqu'il est composé avec un autre processus. Il peut être vu comme le processus universel, à l'inverse de $\mho$ défini par l'ensemble vide de comportements. L'utilisation de $\mho$ dans une conjonction de contraintes sera caractérisée par un ensemble vide de solutions. $\mho$ peut être vu comme le processus nul.

Soit $\mathbf{X}$ un ensemble fini de variables ($\mathbf{X} \subset \mathcal{V}$), nous notons $\mathbb{P}_\mathbf{X}$ l'ensemble des processus définis sur $\mathbf{X}$. Un processus appartenant à $\mathbb{P}_\mathbf{X}$, défini sur un ensemble fini de variables $\mathbf{X}$, est qualifié de *strict*. Par conséquent, Ω est un processus strict. $\mathbb{P}$ désigne l'ensemble de tous les processus.

$$\mathbb{P}_\mathbf{X} =_\Delta \mathcal{P}(\mathbb{B}_\mathbf{X}) \setminus \{\mho\}, \quad (\mathbb{P}_\emptyset = \{\Omega\}) \quad \mathbb{P} =_\Delta \cup_{(\mathbf{X} \subset \mathcal{V})} \mathbb{P}_\mathbf{X}$$

Le domaine de définition des comportements dans un processus p défini sur l'ensemble de variables $\mathbf{X}$ est noté $vars(p) =_\Delta \mathbf{X}$. $\mho$ est le seul processus non strict de $\mathbb{P}_\mathcal{V}$: $vars(\mho) = \mathcal{V}$. Un processus est un processus strict ou bien il s'agit de $\mho$. Ainsi, l'ensemble de tous les processus $\mathbb{P}^\star$ est défini par $\mathbb{P}^\star =_\Delta \mathbb{P} \cup \{\mho\}$ et $\forall \mathbf{X} \subset \mathcal{V}, \mathbb{P}^\star{}_\mathbf{X} =_\Delta \mathbb{P}_\mathbf{X} \cup \{\mho\}$. Pour $\mathbf{R} \subseteq \mathbb{P}^\star$, $\overline{\mathbf{R}}$ désigne le complémentaire de $\mathbf{R}$.

Les opérateurs suivants seront utilisés pour définir les filtres et les contrats : le complémentaire d'un processus p dans $\mathbb{P}_\mathbf{X}$ est un processus dans $\mathbb{P}^\star{}_\mathbf{X}$; la restriction d'un processus p dans $\mathbb{P}_\mathbf{X}$ à $\mathbf{Y} \subseteq \mathbf{X} \subset \mathcal{V}$ est l'abstraction (projection) de p sur $\mathbf{Y}$; finalement, l'extension de p défini dans $\mathbb{P}_\mathbf{X}$ à un ensemble fini de variables $\mathbf{Y} \subset \mathcal{V}$, est le processus défini sur $\mathbf{Y}$ qui représente les mêmes contraintes que p.

Définition 2.3 (Complémentaire d'un processus) *Soit* $\mathbf{X}$ *un ensemble fini de variables (*$\mathbf{X} \subset \mathcal{V}$*), le processus complémentaire* $\widetilde{p}$ *d'un processus* $p \in \mathbb{P}_{\mathbf{X}}$ *est défini par :*

$$p \in \mathbb{P}_{\mathbf{X}} \Longrightarrow \widetilde{p} =_{\Delta} (\mathbb{B}_{\mathbf{X}} \setminus p) = \{b \in \mathbb{B}_{\mathbf{X}} | b \notin p\} \quad (\widetilde{\mathbb{B}_{\mathbf{X}}} = \mho) \tag{2.1}$$

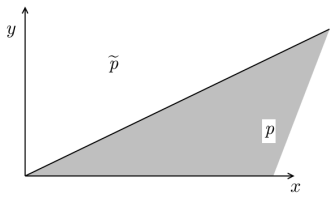


FIG. 2.3 – Complémentaire d'un processus.

Le complémentaire $\widetilde{p}$ d'un processus p défini sur les variables x et y, Figure 2.3, contient tous les comportements définis sur x, y n'appartenant pas à p.

Exemple 1 *Soit* p *un processus (avec* $vars(p) = \{x, y\}$*, et* $x, y \in \mathbb{N}$*), défini par un ensemble de comportements satisfaisant* $(x > 0) \wedge (y$ est impair*) alors* $\widetilde{p}$ *est l'ensemble des comportements tels que* $(x \leq 0) \vee (y$ est pair$)$.

Définition 2.4 (Restriction et extension de processus) *Soit* $\mathbf{X}$, $\mathbf{Y}$ *deux ensembles finis de variables tels que :* $\mathbf{X} \subseteq \mathbf{Y} \subset \mathcal{V}$*, et* $\mathbf{Y}$ *non vides. Nous définissons la restriction* $q_{|\mathbf{X}} \in \mathbb{P}_{\mathbf{X}}$ *de* $q \in \mathbb{P}_{\mathbf{Y}}$ *à* $\mathbf{X}$ *et l'extension* $p^{|\mathbf{Y}} \in \mathbb{P}_{\mathbf{Y}}$ *de* $p \in \mathbb{P}_{\mathbf{X}}$ *à l'ensemble de variables* $\mathbf{Y}$ *par :*

$$q_{|\mathbf{X}} =_{\Delta} \{c_{|\mathbf{X}} | c \in q\} \quad (alors\ q_{|\emptyset} = \Omega, q_{|vars(q)} = q) \tag{2.2}$$

$$p^{|\mathbf{Y}} =_{\Delta} \{c \in \mathbb{B}_{\mathbf{Y}} | c_{|\mathbf{X}} \in p\} \quad (alors\ \Omega^{|\mathbf{Y}} = \mathbb{B}_{\mathbf{Y}}, p^{|vars(p)} = p) \tag{2.3}$$

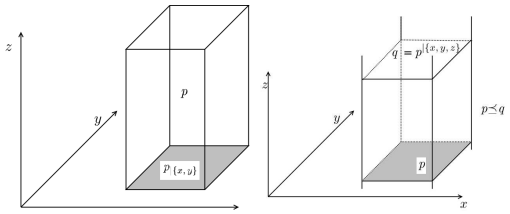


FIG. 2.4 – Restriction et extension.

La restriction $p_{|\{x,y\}}$ d'un processus p défini sur les variables x, y, z, Figure 2.4, consiste à projeter p sur un ensemble de variables restreint ; à droite, l'extension $p^{|\{x,y,z\}}$ d'un processus p défini sur x, y est le plus grand processus défini sur x, y, z tel que la restriction sur x, y est égale à p.

Exemple 2 *Soit p un processus tel que $vars(p) = \{x, y, z\}$, et $x, y, z \in \mathbb{N}$, défini par l'ensemble de comportements tels que $(x > 0) \wedge (y$ est impair$) \wedge (z < 2)$ alors $p_{|\{x,y\}}$ est l'ensemble des comportements tels que $(x > 0) \wedge (y$ est impair$)$.*

Exemple 3 *Soit p un processus tel que $vars(p) = \{x, y\}$, et $x, y, z \in \mathbb{N}$, défini par l'ensemble de comportements tels que $(x > 0) \wedge (y$ est impair$)$ alors $p^{|\{x,y,z\}}$ est l'ensemble des comportements tels que $(x > 0) \wedge (y$ est impair$) \wedge (z \in \mathbb{N})$.*

L'ensemble $\mathbb{P}^\star{}_{\mathbf{X}}$, équipé des opérations d'union, d'intersection et de la notion de complémentaire étendue avec $\widetilde{\mho} = \mathbb{B}_{\mathbf{X}}$, est une algèbre booléenne avec pour supremum $\mathbb{P}^\star{}_{\mathbf{X}}$ et pour infimum $\mho$.

La définition de la restriction est étendue à $\mho$, le processus nul, par $\mho_{|\mathbf{X}} = \{\, c_{|\mathbf{X}} \mid c \in \emptyset\} = \mho$. $\mathcal{V}$ est l'ensemble de toutes les variables, la définition de l'extension est simplement étendue à $\mho$, par $\mho^{|\mathcal{V}} = \mho$.

Propriété 2.1 *Soient $\mathbf{W}$, $\mathbf{X}$, $\mathbf{Y}$, $\mathbf{Z}$ des ensembles finis de variables, $\mathbf{Y}$, $\mathbf{Z}$ non vides, et p, q deux processus stricts :*

$$vars(p) \subseteq \mathbf{Z} \subseteq \mathbf{Y} \implies (p^{|\mathbf{Z}|\mathbf{Y}} = p^{|\mathbf{Y}}) \wedge (p^{|\mathbf{Y}}{}_{|\mathbf{Z}} = p^{|\mathbf{Z}}) \quad (2.4)$$

$$vars(p) = vars(q) \subseteq \mathbf{Y} \implies ((p \cap q)^{|\mathbf{Y}} = (p^{|\mathbf{Y}} \cap q^{|\mathbf{Y}})) \wedge ((p \cup q)^{|\mathbf{Y}} = (p^{|\mathbf{Y}} \cup q^{|\mathbf{Y}})) \quad (2.5)$$

$$vars(p) = vars(q) \subseteq \mathbf{Y} \implies ((p \subseteq q) \iff (p^{|\mathbf{Y}} \subseteq q^{|\mathbf{Y}})) \quad (2.6)$$

$$\mathbf{X} \subseteq vars(p) = vars(q) \implies ((p \subseteq q) \implies (p_{|\mathbf{X}} \subseteq q_{|\mathbf{X}})) \quad (2.7)$$

Preuve : Les preuves sont immédiates par l'équation 2.2 et l'équation 2.3.

2.1.3 Une relation d'ordre partiel sur les processus

L'opération d'extension de processus induit un ordre partiel $\preceq$, tel que $p \preceq q$ si q est une extension de p aux variables de q. La relation $\preceq$ sera ensuite utilisée pour définir les filtres. Elle est définie comme suit.

Définition 2.5 (Relation d'extension de processus) *$(\forall\ p \in \mathbb{P})$ $(\forall\ q \in \mathbb{P})$, la relation d'extension de processus $\preceq$ est définie, par*

$$(p \preceq q) \iff ((vars(p) \subseteq vars(q)) \wedge (p^{|vars(q)} = q))$$

Donc, si $(p \preceq q)$, q est défini sur plus de variables que p. Sur les variables communes à p, le processus q caractérise les mêmes contraintes que p ; ses autres variables sont libres. Cette relation est étendue à $\mathbb{P}^\star$ avec $(\mho \preceq \mho)$.

Propriété 2.2 *$(\mathbb{P}^\star, \preceq)$ est un ordre partiel.*

Preuve : La vérification de la transitivité, l'antisymétrie et la réflexivité est immédiate par l'équation 2.2 et l'équation 2.3.

2.1.4 L'ensemble des processus étendus

Dans cet ordre partiel, l'ensemble des processus étendus de p est l'ensemble de toutes les extensions de p, noté :

$$[\preceq\!\uparrow p] =_{\Delta} \{q \in \mathbb{P} | p \preceq q\} \quad ([\preceq\!\uparrow \Omega] = \{\mathbb{B}_{\mathbf{X}}\}_{\mathbf{X} \subset \mathcal{V}}) \tag{2.8}$$

L'ensemble des éléments étendus est illustré sur la figure 2.5.

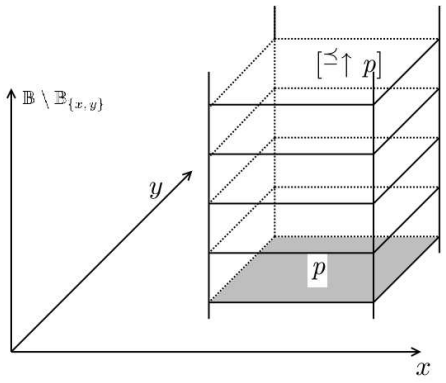


FIG. 2.5 – Ensemble des processus étendus.

2.1.5 Les variables contrôlées et les processus réduits

Afin d'étudier les propriétés des ensembles des processus étendus, nous caractérisons l'ensemble des variables contraintes par un processus donné : nous disons que $q \in \mathbb{P}$ *contrôle* une variable y, si :

- y appartient à $vars(q)$,
- il existe des comportements b dans q ayant la même restriction sur l'ensemble des variables $(vars(q) \backslash \{y\})$ qu'un comportement c dans $\mathbb{B}_{vars(q)}$ tel que c n'appartient pas à q ; donc q est strictement inclus dans $(q_{|(vars(q)\backslash\{y\})})^{|vars(q)}$.

Ceci est illustré sur la figure 2.6.

Définition 2.6 (Variable contrôlée $\rhd$) *Un processus q contrôle une variable y, noté $(q \rhd y)$, si et seulement si*

$$((y \in vars(q)) \wedge q \subsetneq ((q_{|(vars(q)\backslash\{y\})})^{|vars(q)})) \tag{2.9}$$

Un processus q contrôle un ensemble de variables X, noté $(q \rhd X)$ si et seulement si

$$(\forall\, x \in \mathbf{X})(q \rhd x) \quad (\Omega \rhd \emptyset) \tag{2.10}$$

De plus, $\rhd$ est étendu à $\mathbb{P}^\star$ avec $\mathcal{U} \rhd \mathcal{V}$.

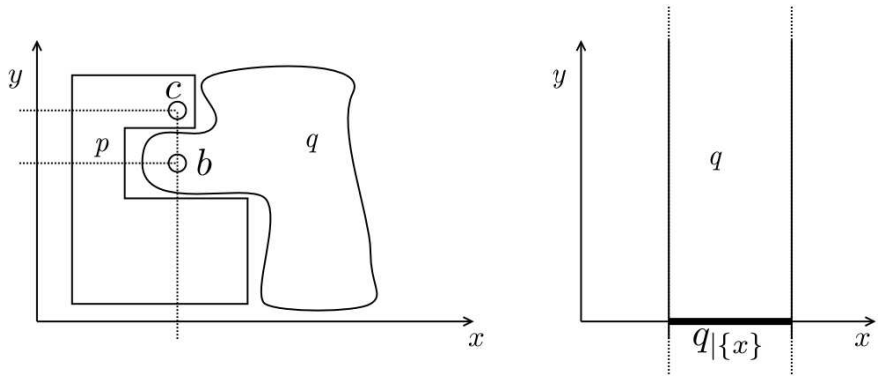


FIG. 2.6 – Variable y contrôlée (à gauche) et non contrôlée (à droite) dans un processus q.

Remarquons que si un processus q contrôle $\mathbf{X}$, cela n'implique pas que, pour tout $x \in \mathbf{X}$, et $y \in \mathbf{X}$, avec $x \neq y$, $(p_{|(\mathbf{X}\setminus\{x\})})$ contrôle y : ce qui peut être le cas lorsque x est contraint par y dans p ; alors si x est "effacée" (par la projection sur d'autres variables contrôlées), y peut être libre dans cette projection.

Nous définissons le *processus réduit* (un concept clé pour définir les filtres) comme un processus qui contrôle toutes ses variables.

Définition 2.7 (processus réduit) *Un processus strict $p \in \mathbb{P}^\star$ est* réduit *si et seulement si $p \rhd vars(p)$.*

Par exemple, Ω est un processus réduit. Par opposition à $\mathbb{B}_{\mathbf{X}}$ qui n'est pas réduit si $\mathbf{X}$ n'est pas vide. *Les processus stricts réduits* sont minimaux (au sens de la relation $\preceq$) dans $(\mathbb{P},\preceq)$. Nous notons $\overset{\triangledown}{q}$, la *réduction* de q, le processus minimal tel que $\overset{\triangledown}{q} \preceq q$ (p est réduit si et seulement si $\overset{\triangledown}{p} = p$). Pour tout $\mathbf{X}$, nous avons $\overset{\triangledown}{\mathbb{B}}_{\mathbf{X}} = \Omega$.

La figure 2.7 illustre la réduction $\overset{\triangledown}{q}$ d'un processus q et d'un processus p, dans l'ensemble des processus étendus $[\preceq\!\uparrow \overset{\triangledown}{q}]$. Posons $vars(q) = (\{x_1 \dots x_n\} \cup \{y_1 \dots y_m\})$ et q contrôle les variables $\{x_1 \dots x_n\}$, nous avons $vars(\overset{\triangledown}{q}) = \{x_1 \dots x_n\}$. Le processus p est tel que $p \in [\preceq\!\uparrow \overset{\triangledown}{q}]$ avec $vars(p) \subseteq (\{x_1 \dots x_n\} \cup \{y_1 \dots y_m\} \cup \{z_1 \dots z_l\})$; il contrôle les variables $\{x_1 \dots x_n\}$. $\{y_1 \dots x_m\} \cup \{z_1 \dots z_l\}$ est l'ensemble des variables libres, tel que $\overset{\triangledown}{q} = \overset{\triangledown}{p}$.

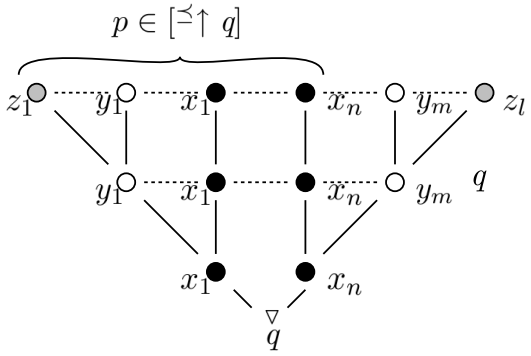


FIG. 2.7 – Réduction de processus.

Propriété 2.3 *Le complémentaire $\widetilde{p}$ d'un processus non vide p strictement inclus dans $\mathbb{B}_{vars(p)}$ est réduit si et seulement si p est réduit. Par conséquent, $\widetilde{p}$ et p contrôlent le même ensemble de variables $vars(p)$.*

Preuve : La preuve est immédiate en considérant les définitions du complémentaire (définition 2.3), des variables contrôlées (définition 2.6) ainsi que la définition d'un processus réduit (définition 2.7).

Nous en déduisons que l'ensemble des processus étendus $[\preceq\!\uparrow \overset{\triangledown}{p}]$ de la réduction de p est un filtre [AJ87] : il est non vide et chaque paire d'éléments possède une borne inférieure. Ainsi, $[\preceq\!\uparrow \overset{\triangledown}{p}]$ est composé de tous les ensembles de comportements définis sur les ensembles de variables incluant les variables contrôlées par p. Les variables qui ne sont pas contrôlées par p ne sont pas contrôlées par les processus appartenant à $[\preceq\!\uparrow \overset{\triangledown}{p}]$. Les processus de cet ensemble caractérisent les mêmes contraintes que p. Nous observons également que $vars(\overset{\triangledown}{q})$ est le plus grand sous-ensemble de variables tel que $q \rhd vars(\overset{\triangledown}{q})$. Pour un processus $q \in \mathbb{P}^\star$, nous étendons la définition de $vars()$ à l'ensemble des processus étendus de la réduction de q avec : $vars([\preceq\!\uparrow \overset{\triangledown}{q}]) =_\Delta vars(\overset{\triangledown}{q})$. Notons que $[\preceq\!\uparrow \mho] = \{\mho\}$.

Propriété 2.4 *L'ensemble des processus étendus d'un processus strict p contient un unique processus $p^{|\mathbf{Y}}$ défini sur un ensemble de variables donné $\mathbf{Y} \supseteq vars(p)$. Le processus p et ses extensions $p^{|\mathbf{Y}}$ contrôlent le même ensemble de variables,*

qui est aussi l'ensemble des variables contrôlées par la réduction de p.

$$((q \in [\preceq\uparrow p]) \wedge (r \in [\preceq\uparrow p]) \wedge (vars(q) = vars(r))) \Longrightarrow (q = r) \qquad (2.11)$$

$$(vars(p) \subseteq \mathbf{Y}) \Longrightarrow ((p_{|\mathbf{Y}}) \rhd vars(\overset{\triangledown}{p})) \qquad (2.12)$$

$$((vars(p) \cup vars(q)) \subseteq \mathbf{Y}) \Longrightarrow ((p_{|\mathbf{Y}} = q_{|\mathbf{Y}}) \Longrightarrow (\overset{\triangledown}{p} = \overset{\triangledown}{q})) \qquad (2.13)$$

Preuve : équation 2.11 :

$$\begin{array}{lll} (q \in [\preceq\uparrow p]) \wedge (r \in [\preceq\uparrow p]) & \Longleftrightarrow & (p \preceq q) \wedge (p \preceq r) \quad \text{(équation 2.8)} \\ & \Longleftrightarrow & (p_{|vars(q)} = q) \wedge (p_{|vars(r)} = r) \text{ (définition 2.5)} \\ & \Longrightarrow & ((vars(q) = vars(r)) \Longrightarrow (q = r)) \end{array}$$

Preuve : équation 2.12 :

$(\forall\, x \in vars(\overset{\triangledown}{p}))\ (p \rhd x)$; soit $q = p_{|\mathbf{Y}}$

$$\begin{array}{lll} (p \rhd x) & \Longrightarrow & (x \in vars(p)), (x \in vars(p)) \Longrightarrow (x \in \mathbf{Y}) \quad \text{(équation 2.9)} \\ & \Longrightarrow & (\exists\, a \in (p_{|(\mathbf{X}\setminus\{x\})})) \quad \text{(équation 2.9)} \\ & & \quad (\exists\, b \in p)\ (b_{|(\mathbf{X}\setminus\{x\})} = a) \\ & & \text{et} \quad (\exists\, v \in \mathcal{D})\ (\forall\, b \in p)\ ((b_{|(\mathbf{X}\setminus\{x\})} = a) \Longrightarrow (b(x) \neq v)) \\ & \Longrightarrow & (\exists\, a \in (p_{|(\mathbf{X}\setminus\{x\})})) \\ & & \quad (\exists\, e \in q)\ (e_{|(\mathbf{X}\setminus\{x\})} = a) \\ & & \text{et} \quad (\exists\, v \in \mathcal{D})\ (\forall\, e \in q)\ ((e_{|(\mathbf{X}\setminus\{x\})} = a) \Longrightarrow (e(x) \neq v)) \\ & \Longrightarrow & (\exists\, a \in (p_{|(\mathbf{X}\setminus\{x\})}))\ (\exists\, d \in (q_{|(\mathbf{Y}\setminus\{x\})})) \\ & & \quad (\exists\, e \in q)\ (e_{|(\mathbf{Y}\setminus\{x\})} = d) \wedge (d_{|(\mathbf{X}\setminus\{x\})} = (e_{|(\mathbf{X}\setminus\{x\})} = a)) \\ & & \text{et} \quad (\exists\, v \in \mathcal{D})\ (\forall\, e \in q)\ ((e_{|(\mathbf{Y}\setminus\{x\})} = d) \Longrightarrow (e(x) \neq v)) \\ & \Longrightarrow & (\exists\, d \in (q_{|(\mathbf{Y}\setminus\{x\})})) \\ & & \quad (\exists\, e \in p)\ (e_{|(\mathbf{Y}\setminus\{x\})} = d) \\ & & \text{et} \quad (\exists\, v \in \mathcal{D})\ (\forall\, e \in p)\ ((e_{|(\mathbf{Y}\setminus\{x\})} = d) \Longrightarrow (e(x) \neq v)) \\ & \Longrightarrow & (q \rhd x) \quad \text{(équation 2.9)}\square \end{array}$$

Preuve : équation 2.13 :

Nous avons $(p_{|\mathbf{Y}} = q_{|\mathbf{Y}})$. En considérant l'équation 2.12, nous en déduisons que p et q contrôlent $vars(\overset{\triangledown}{p})$ et $vars(\overset{\triangledown}{q})$. Donc p et q contrôlent $vars(\overset{\triangledown}{p}) \cup vars(\overset{\triangledown}{q})$; ce qui implique que $vars(\overset{\triangledown}{p}) = vars(\overset{\triangledown}{q})$. Par conséquent, $\overset{\triangledown}{p} = \overset{\triangledown}{q}$.

Pour le processus bloquant, nous avons $\mho \rhd \mathcal{V}$ et $[\preceq\uparrow \mho] = \{\mho\}$.

2.1.6 L'ensemble des sous-processus de processus

Nous définissons l'ensemble des sous-processus d'un processus afin de capturer tous les sous-ensembles de comportements. Soit $\mathbf{R} \subseteq \mathbb{P}^\star$, $[\mathbf{R}\downarrow_\subseteq]$ est l'ensemble des sous-comportements des processus de $\mathbf{R}$ pour $\subseteq$:

$$[\mathbf{R}\downarrow_\subseteq] =_\Delta \{p \in \mathbb{P}^\star | (\exists\ q \in \mathbf{R})(p \subseteq q)\} \tag{2.14}$$

Propriété 2.5 *A partir de ces définitions, nous déduisons que :*

$$[[\preceq\uparrow \overset{\triangledown}{\mho}]\downarrow_\subseteq] = \{\mho\} \tag{2.15}$$

$$[[\preceq\uparrow \overset{\triangledown}{\Omega}]\downarrow_\subseteq] = \mathbb{P}^\star \tag{2.16}$$

2.2 Une algèbre de filtres

Dans cette section, nous intoduisons la notion de filtre comme un ensemble de processus qui satisfont une propriété donnée. Nous proposons une relation d'ordre ($\sqsubseteq$) sur l'ensemble des filtres Φ. Puis, nous établissons que (Φ,$\sqsubseteq$) est une algèbre booléenne. Un *filtre* $\mathbf{R}$ est un sous-ensemble de $\mathbb{P}^\star$ filtrant les processus. Les filtres seront ensuite utilisés pour modéliser les hypothèses et les garanties des contrats : un contrat sera ainsi représenté par une paire de filtres. L'algèbre booléenne définie sur l'ensemble des filtres permettra de caractériser l'ensemble des contrats par une structure mathématique riche.

2.2.1 Sémantique des filtres

Un filtre $\mathbf{R}$ contient tous les processus qui sont "équivalents" , c'est-à-dire qui caractérisent la même contrainte ou propriété. Tous les processus de $\mathbf{R}$ sont "acceptés". Un filtre est construit à partir d'un unique processus *générateur* en étendant celui-ci à un ensemble de variables plus large et en incluant tous les (sous-)processus constitués d'un sous-ensemble de comportements du processus générateur.

Définition 2.8 (Filtre) *Un ensemble de processus* $\mathbf{R}$ *est un* filtre *si et seulement si* ($\exists\ r \in \mathbb{P}^\star$) *tel que,* ($(r = \overset{\triangledown}{r})\ \wedge\ (\mathbf{R} = [[\preceq\uparrow r]\downarrow_\subseteq]$)). *Le processus* r *est un* générateur *de* $\mathbf{R}$ ($\mathbf{R}$ *est généré par* r). *Nous notons* Φ *l'ensemble des filtres.*

Le filtre généré par la réduction d'un processus p est noté $\widehat{[p]} =_\Delta [[\preceq\uparrow \overset{\triangledown}{p}]\downarrow_\subseteq]$.

La figure 2.8 illustre la façon dont un filtre est généré à partir d'un processus p (représenté par la ligne épaisse) en deux opérations successives.
La première opération consiste à construire l'ensemble des processus étendus de p : cet ensemble représente tous les processus compatibles avec la contrainte caractérisée par p et définis sur un ensemble de variables plus large.
La seconde opération consiste à construire l'ensemble des sous-processus de cet ensemble de processus : tous les processus définis comme des sous-ensembles de comportements des processus contenus dans l'ensemble des processus étendus de p. En d'autres termes, tous les processus qui restent compatibles lors de l'ajout d'une contrainte supplémentaire à la contrainte caractérisée par p, puisque l'ajout de contraintes supprime des comportements.

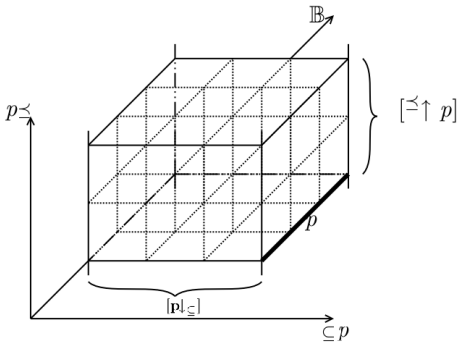


FIG. 2.8 – Génération de filtre.

Exemple 4 *Soit r un processus tel que $vars(r) = \{x, y, z\}$, et $x, y, z \in \mathbb{N}$, défini par l'ensemble de comportements satisfaisant* $(x > 10) \wedge (y$ est impair$) \wedge (z \in \mathbb{N})$ *(z est une variable libre).*

Ainsi le filtre $[[\stackrel{\preceq}{\uparrow} \stackrel{\nabla}{r}]\downarrow_{\subseteq}]$ définit l'ensemble des processus qui satisfont $(x > 10) \wedge$ (y est impair).

Soit s un processus tel que $vars(s) = \{x, y, u\}$, et $x, y, u \in \mathbb{N}$, défini par l'ensemble de comportements satisfaisant $(x > 10) \wedge (y$ est impair$) \wedge (u > 100)$. *Alors* $s \in [[\stackrel{\preceq}{\uparrow} \stackrel{\nabla}{r}]\downarrow_{\subseteq}]$.

Un filtre $\mathbf{R} = \widehat{[r]}$ satisfait les propriétés suivantes :

Propriété 2.6 *L'ensemble des variables sur lequel est défini un processus p, appartenant à un filtre généré par le processus réduit $\stackrel{\nabla}{r}$, contient l'ensemble des variables de ce processus $\stackrel{\nabla}{r}$. Le générateur d'un filtre est unique, nous le désignons par la notation $\stackrel{\nabla}{\mathbf{R}}$. Ω génère l'ensemble de tous les processus (incluant $\mho$). $\mho$ appartient à tous les filtres. Formellement ($\forall$ p,r,$s \in \mathbb{P}^\star$) :*

$$(p \in \widehat{[r]}) \quad \Longrightarrow \quad (vars(\stackrel{\nabla}{r}) \subseteq vars(p)) \tag{2.17}$$

$$\widehat{[r]} = \widehat{[s]} \quad \Longleftrightarrow \quad \stackrel{\nabla}{r} = \stackrel{\nabla}{s} \tag{2.18}$$

$$\Omega \in \widehat{[r]} \quad \Longleftrightarrow \quad \widehat{[r]} = \mathbb{P}^\star \tag{2.19}$$

$$\mho \in \mathbf{R} \tag{2.20}$$

Preuve : équation 2.17
Par les définitions données, ainsi que l'équation 2.8 et l'équation 2.14, p est inclus dans l'extension de $\overset{\triangledown}{r}$.

Preuve : équation 2.18 ($\Longleftarrow$ est trivial.)

$$\begin{aligned}
\widehat{[r]} = \widehat{[s]} \quad &\Longrightarrow \quad (\overset{\triangledown}{r} \in \widehat{[s]}) \ \wedge (\overset{\triangledown}{s} \in \widehat{[r]}) \quad \text{(équation 2.8, équation 2.14)} \\
&\Longrightarrow \quad (vars(\overset{\triangledown}{s}) \subseteq vars(\overset{\triangledown}{r})) \ \wedge (vars(\overset{\triangledown}{r}) \subseteq vars(\overset{\triangledown}{s})) \quad \text{(équation 2.17)} \\
&\Longrightarrow \quad (\overset{\triangledown}{r} \subseteq \overset{\triangledown}{s}) \ \wedge (\overset{\triangledown}{s} \subseteq \overset{\triangledown}{r}) \quad \text{(équation 2.8, équation 2.14)} \ \square
\end{aligned}$$

Preuve : équation 2.19 : ($\Longleftarrow$ est trivial)

$$\begin{aligned}
\Omega \in \widehat{[r]} \quad &\Longrightarrow \quad (vars(\overset{\triangledown}{r}) \subseteq vars(\Omega) = \emptyset) \qquad \text{(équation 2.17)} \\
&\Longrightarrow \quad \overset{\triangledown}{r} = \Omega \ \square
\end{aligned}$$

Preuve : équation 2.20
Conséquence directe de la définition 2.8 des filtres. □

Exemple 5

Soit $p \in \mathbb{P}_{\{x\}}$ un processus défini sur une variable $x \in \mathcal{V}$ dont les comportements sont une fonction d'un domaine totalement ordonné modélisant le temps $\mathbb{T}$ vers l'ensemble des nombres rationnels $\mathbb{Q}$ (figure 2.9). Soit le processus p satisfaisant la contrainte :

$$\forall b \in p, \ b(x) : \mathbb{T} \mapsto \mathbb{Q}, \text{ tel que } \forall t, t' \in \mathbb{T}, t \leq t' \Leftrightarrow b(x)(t) \leq b(x)(t')$$

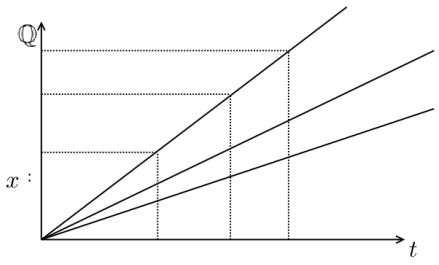


FIG. 2.9 – Exemple de contrainte.

Alors $\widehat{[p]}$ est l'ensemble de tous les processus tel que $\forall b \in \mathbb{B}$, $b \in p$, $b(x)$ est une fonction monotone croissante du domaine de temps $\mathbb{T}$ vers $\mathbb{Q}$.

Nous appellons *filtres stricts* les filtres qui sont différents de $\mathbb{P}^\star$ et $\{\mho\}$. L'ensemble des variables filtrées de $\mathbf{R}$ est $vars(\mathbf{R})$ défini par :

$$vars(\mathbf{R}) =_\Delta vars(\overset{\triangledown}{\mathbf{R}}) \tag{2.21}$$

Le théorème 2.1 exprime les propriétés devant être satisfaites entre un processus p et un processus générateur $\overset{\triangledown}{\mathbf{R}}$ afin que p appartienne au filtre $\mathbf{R}$. Ainsi un processus $p \in \mathbb{P}$ appartient à un filtre $\mathbf{R}$ si et seulement si,

- dans un premier temps, les variables $vars(\mathbf{R})$ contrôlées par le générateur de $\mathbf{R}$ sont des variables de p,
- et dans un second temps, pour un ensemble de variables $\mathbf{X}$ incluant $vars(\mathbf{R})$, et un ensemble arbitraire de variables $\mathbf{Y}$ incluant $\mathbf{X}$ et $vars(p)$, la réduction sur $\mathbf{X}$ d'une extension p à $\mathbf{Y}$, est un sous-ensemble des comportements de l'extension du processus générateur de $\mathbf{R}$ à $\mathbf{X}$.

Ce théorème est particulièrement intéressant dans le cadre d'une implémentation du modèle des filtres. En effet, il permet de vérifier l'appartenance d'un processus à un filtre en observant uniquement le processus générateur du filtre sans calculer le filtre. Formellement,

Théorème 2.1 *$(\forall\ \mathbf{X},\mathbf{Y} \subset \mathcal{V})(vars(\mathbf{R}) \subseteq \mathbf{X} \subseteq \mathbf{Y})$, un processus strict p appartient à un filtre $\mathbf{R}$, noté $(p \in \mathbf{R})$, si et seulement si :*

$$\begin{cases} (vars(\mathbf{R}) \subseteq vars(p)) \\ \wedge \\ (vars(p) \subseteq \mathbf{Y}) \implies ((p^{|\mathbf{Y}})_{|\mathbf{X}} \subseteq \overset{\triangledown}{\mathbf{R}}^{|\mathbf{X}}) \end{cases}$$

Preuve : $(\implies)$

$(p \in \mathbf{R}) \implies (vars(\mathbf{R}) \subseteq vars(p))$ (équation 2.21 et équation 2.17)

$(p \in \mathbf{R}) \iff (\exists\ s \in \mathbf{R})((p \subseteq s)\ \wedge (s \in [\preceq\uparrow \overset{\triangledown}{\mathbf{R}}]))$
(définition 2.8 et équation 2.14)

$(p \in \mathbf{R}) \iff (\exists\ s \in \mathbf{R})((p \subseteq s)\ \wedge (s = \overset{\triangledown}{\mathbf{R}}^{|vars(p)}))$ (équation 2.8)

$(p \in \mathbf{R}) \iff (p \subseteq \overset{\triangledown}{\mathbf{R}}^{|vars(p)})$

Il résulte de la propriété 2.1 (équation 2.6 et équation 2.7) que :

$(p \in \mathbf{R}) \implies (((vars(p) \subseteq \mathbf{Y}) \wedge (\mathbf{X} \subseteq \mathbf{Y})) \Longrightarrow ((p^{|\mathbf{Y}})_{|\mathbf{X}} \subseteq (\overset{\triangledown}{\mathbf{R}}^{|vars(p)})^{|\mathbf{Y}})_{|\mathbf{X}}))$

$(p \in \mathbf{R}) \implies ((p^{|\mathbf{Y}})_{|\mathbf{X}} \subseteq (\overset{\triangledown}{\mathbf{R}}^{|\mathbf{Y}})_{|\mathbf{X}} = \overset{\triangledown}{\mathbf{R}}^{|\mathbf{X}}))$ (propriété 2.1-équation 2.4) □

Preuve : $(\Longleftarrow)$

$((p^{|\mathbf{Y}})_{|\mathbf{X}} \subseteq \overset{\triangledown}{\mathbf{R}}^{|\mathbf{X}}) \implies (p_{|vars(\mathbf{R})} \subseteq \overset{\triangledown}{\mathbf{R}})$ en posant $(\mathbf{Y} = vars(p),\ \mathbf{X} = vars(\mathbf{R}))$

$\implies (p \subseteq \overset{\triangledown}{\mathbf{R}}^{|vars(p)})$ (propriété 2.1)

$\implies p \in \mathbf{R}$ □

La figure 2.10 illustre le théorème 2.1 en considérant un processus p défini sur un ensemble de variables $\mathbf{Y}$ et un filtre $\mathbf{R}$ défini sur un ensemble de variables $\mathbf{X}$

($vars(\mathbf{R})$=$\mathbf{X}$). Nous observons que p, d'abord étendu à $\mathbf{Y}$ puis réduit à $\mathbf{X}$, est contenu dans le processus générateur de $\mathbf{R}$ étendu à $\mathbf{X}$ (décrit par la ligne épaisse). Les variables contrôlées par le processus générateur $\overset{\triangledown}{\mathbf{R}}$ du filtre contenant p, sont

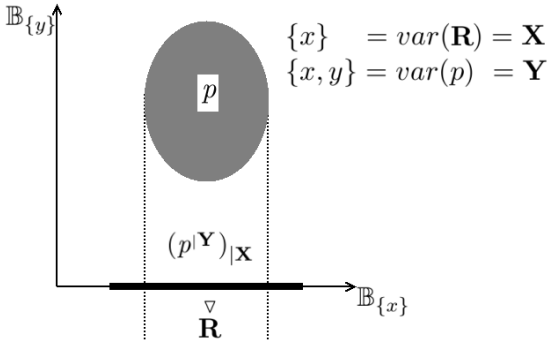


FIG. 2.10 – Le processus p appartient au filtre $\mathbf{R}$.

des variables de p. Les comportements de p sur ces variables contrôlées sont des comportements appartenant également au processus générateur. p représente les mêmes contraintes que $\overset{\triangledown}{\mathbf{R}}$ sur les variables contrôlées, cependant il peut contenir d'autres variables que celles contrôlées par $\overset{\triangledown}{\mathbf{R}}$.

Corollaire 2.1 *Ces deux propriétés équivalentes sont satisfaites :*

$$\mathbf{R} \subseteq \mathbf{S} \Longleftrightarrow ((vars(\mathbf{S}) \subseteq vars(\mathbf{R})) \wedge (\overset{\triangledown}{\mathbf{R}}_{|vars(\mathbf{S})} \subseteq \overset{\triangledown}{\mathbf{S}})) \tag{2.22}$$

$$\mathbf{R} \subseteq \mathbf{S} \Longleftrightarrow \overset{\triangledown}{\mathbf{R}} \in \mathbf{S} \tag{2.23}$$

Preuve : ($\Longrightarrow$)
Comme $\overset{\triangledown}{\mathbf{R}} \in \mathbf{S} \iff (vars(\mathbf{S}) \subseteq vars(\mathbf{R})) \wedge (\overset{\triangledown}{\mathbf{R}}_{|vars(\mathbf{S})} \subseteq \overset{\triangledown}{\mathbf{S}})$(théorème 2.1)
et $\mathbf{R} \subseteq \mathbf{S} \Longrightarrow \overset{\triangledown}{\mathbf{R}} \in \mathbf{S}$ $(\overset{\triangledown}{\mathbf{R}} \in \mathbf{R})$
donc $\mathbf{R} \subseteq \mathbf{S} \Longrightarrow (vars(\mathbf{S}) \subseteq vars(\mathbf{R})) \wedge (\overset{\triangledown}{\mathbf{R}}_{|vars(\mathbf{S})} \subseteq \overset{\triangledown}{\mathbf{S}})$

Preuve : ($\Longleftarrow$)
Comme $(\forall p \in \mathbb{P})(p \in \mathbf{R} \iff (vars(\mathbf{R}) \subseteq vars(p)) \wedge (p_{|vars(\mathbf{R})} \subseteq \overset{\triangledown}{\mathbf{R}}))$(théorème 2.1)
et $(vars(\mathbf{S}) \subseteq vars(\mathbf{R})) \wedge (\overset{\triangledown}{\mathbf{R}}_{|vars(\mathbf{S})} \subseteq \overset{\triangledown}{\mathbf{S}})$(par hypothèse)
alors $(\forall p \in \mathbb{P})(p \in \mathbf{R} \Longrightarrow (vars(\mathbf{S}) \subseteq vars(p)) \wedge (p_{|vars(\mathbf{S})} \subseteq \overset{\triangledown}{\mathbf{S}}))$
donc $(\forall p \in \mathbb{P})(p \in \mathbf{R} \Longrightarrow p \in \mathbf{S})$

Corollaire 2.2 *Ces deux propriétés équivalentes sont satisfaites :*

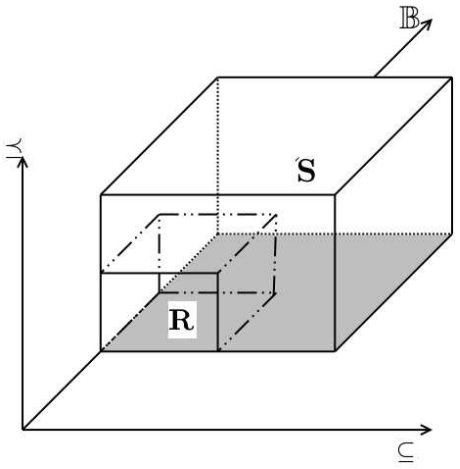
$$(\mathbf{R} \subseteq (\mathbf{S} \cap \mathbf{T})) \iff ((\mathbf{R} \subseteq \mathbf{S}) \wedge (\mathbf{R} \subseteq \mathbf{T}))(\textit{corollaire 2.1, équation 2.23}) \quad (2.24)$$
$$(\mathbf{R} \subseteq (\mathbf{S} \cup \mathbf{T})) \iff ((\mathbf{R} \subseteq \mathbf{S}) \vee (\mathbf{R} \subseteq \mathbf{T}))(\textit{corollaire 2.1, équation 2.23}) \quad (2.25)$$

2.2.2 Une relation d'ordre partiel sur les filtres

Nous définissons une relation d'ordre partiel sur l'ensemble des filtres, que nous appellerons *"relaxation"*, et noterons $\mathbf{R} \sqsubseteq \mathbf{S}$ pour exprimer que $\mathbf{R}$ est moins large que $\mathbf{S}$ (ou que $\mathbf{S}$ relaxe $\mathbf{R}$).

Définition 2.9 (Relaxation de filtre) *Soient* $\mathbf{R}$ *et* $\mathbf{S}$*, deux filtres, la relation* $\mathbf{R}$ est moins large que $\mathbf{S}$*, notée* $\mathbf{R} \sqsubseteq \mathbf{S}$ *est définie par :*

$$\{\mho\} \sqsubseteq \mathbf{S} \qquad (\mathbf{R} \sqsubseteq \{\mho\}) \iff \{\mho\} = \mathbf{R} \qquad (\mathbf{R} \sqsubseteq \mathbf{S} \iff \overset{\triangledown}{\mathbf{R}}{}^{|\mathbf{Z}} \subseteq \overset{\triangledown}{\mathbf{S}}{}^{|\mathbf{Z}}) \quad (2.26)$$
$$\textit{où } \mathbf{Z} = \textit{vars}(\mathbf{R}) \cup \textit{vars}(\mathbf{S})$$



La figure 2.11 illustre la relation de relaxation entre deux filtres $\mathbf{R}$ et $\mathbf{S}$. Les processus satisfaisant la contrainte modélisée par $\mathbf{R}$ satisfont aussi la contrainte définie par $\mathbf{S}$. Ainsi, $\mathbf{R}$ est moins large que $\mathbf{S}$ puisque l'ensemble des processus représentés par $\mathbf{R}$ traduisant une contrainte donnée, est inclus dans $\mathbf{S}$.

FIG. 2.11 – Relaxation du filtre $\mathbf{R}$.

Exemple 6 *Soient* $\mathbf{R}_1$*,* $\mathbf{R}_2$*,* $\mathbf{R}_3$ *les filtres générés par les contraintes* $(x \in \{0,1\} \wedge y = 1)$*,* $x \in \{0,1\}$*,* $(x \in \{0,1\} \vee (x = 2 \wedge z = 0))$*; Ces filtres satisfont :* $\mathbf{R}_1 \sqsubseteq \mathbf{R}_2 \sqsubseteq \mathbf{R}_3$*. Nous avons* $\mathbf{R}_1 \subseteq \mathbf{R}_2$ *et :*

- $\overset{\triangledown}{\mathbf{R}}_1 \notin \mathbf{R}_3$ *et* $\mathbf{R}_1 \nsubseteq \mathbf{R}_3$
- $\overset{\triangledown}{\mathbf{R}}_2 \notin \mathbf{R}_3$ *et* $\mathbf{R}_2 \nsubseteq \mathbf{R}_3$

Propriété 2.7 *Deux filtres stricts* $\mathbf{R}$ *et* $\mathbf{S}$ *satisfont* $(\mathbf{R} \subseteq \mathbf{S}) \iff ((vars(\mathbf{S}) \subseteq vars(\mathbf{R})) \wedge \mathbf{R} \sqsubseteq \mathbf{S})$.

Preuve : Application du corollaire 2.1 et de la définition 2.9.

2.2.3 Une algèbre de filtres

La structure de filtres définie par la relation de relaxation est un treillis. Les propriétés de l'algèbre booléenne seront ensuite utilisées pour définir l'ensemble des contrats caractérisés par une paire de filtres.

Propriété 2.8 $(\Phi, \sqsubseteq)$ *est un ordre partiel.*

La relation $\sqsubseteq$ est réflexive et antisymétrique. La transitivité est facile à montrer en utilisant l'équation 2.4 de la propriété 2.1.

Lemme 2.1 $(\Phi, \sqsubseteq)$ *est un treillis avec pour supremum* $\mathbb{P}^\star$ *et* $\{\mho\}$ *pour infimum. Le plus grand des minorants (la conjonction)* $\mathbf{R} \sqcap \mathbf{S}$*, le plus petit des majorants (la disjonction)* $\mathbf{R} \sqcup \mathbf{S}$ *sont définis par :*

$$\{\mho\} \sqcap \mathbf{R} = \mathbf{R} \sqcap \{\mho\} = \{\mho\} \tag{2.27}$$

$$(\mathbf{R} \neq \{\mho\} \wedge \mathbf{S} \neq \{\mho\}) \implies \mathbf{R} \sqcap \mathbf{S} =_\Delta [[\preceq\!\uparrow \overset{\triangledown}{p}]\!\downarrow_\subseteq] \tag{2.28}$$

$$\text{avec } p = (\mathbf{R}^{\triangledown|\mathbf{V}} \cap \mathbf{S}^{\triangledown|\mathbf{V}}),\ \mathbf{V} = vars(\mathbf{R}) \cup vars(\mathbf{S})$$

$$\{\mho\} \sqcup \mathbf{R} = \mathbf{R} \sqcup \{\mho\} = \mathbf{R} \tag{2.29}$$

$$(\mathbf{R} \neq \{\mho\} \wedge \mathbf{S} \neq \{\mho\}) \implies \mathbf{R} \sqcup \mathbf{S} =_\Delta [[\preceq\!\uparrow \overset{\triangledown}{p}]\!\downarrow_\subseteq] \tag{2.30}$$

$$\text{avec } p = (\mathbf{R}^{\triangledown|\mathbf{V}} \cup \mathbf{S}^{\triangledown|\mathbf{V}}),\ \mathbf{V} = vars(\mathbf{R}) \cup vars(\mathbf{S})$$

Preuve :
Soient $\mathbf{W}_1 = (vars(\mathbf{R}) \cup vars(\mathbf{T}))$, $\mathbf{W}_2 = (vars(\mathbf{S}) \cup vars(\mathbf{T}))$, $\mathbf{W} = (\mathbf{W}_1 \cup \mathbf{W}_2)$,

- $\mathbf{R} \sqcap \mathbf{S}$ est défini comme le plus grand des minorants de $\mathbf{R}$ et $\mathbf{S}$ (nous ignorons le cas trivial où $(\mathbf{R} = \{\mho\} \ \vee \mathbf{S} = \{\mho\})$).
  $((\mathbf{T} \sqsubseteq \mathbf{R}) \wedge (\mathbf{T} \sqsubseteq \mathbf{S})) \iff ((\mathbf{T}^{\triangledown|\mathbf{W}_1} \subseteq \mathbf{R}^{\triangledown|\mathbf{W}_1}) \wedge (\mathbf{T}^{\triangledown|\mathbf{W}_2} \subseteq \mathbf{S}^{\triangledown|\mathbf{W}_2}))$
  en utilisant le théorème 2.1 nous obtenons
  $((\mathbf{T} \sqsubseteq \mathbf{R}) \wedge (\mathbf{T} \sqsubseteq \mathbf{S})) \iff ((\mathbf{T}^{\triangledown|\mathbf{W}} \subseteq \mathbf{R}^{\triangledown|\mathbf{W}}) \wedge (\mathbf{T}^{\triangledown|\mathbf{W}} \subseteq \mathbf{S}^{\triangledown|\mathbf{W}}))$
  $((\mathbf{T} \sqsubseteq \mathbf{R}) \wedge (\mathbf{T} \sqsubseteq \mathbf{S})) \iff (\mathbf{T}^{\triangledown|\mathbf{W}} \subseteq \mathbf{R}^{\triangledown|\mathbf{W}} \cap \mathbf{S}^{\triangledown|\mathbf{W}})$
  donc $((\mathbf{T} \sqsubseteq \mathbf{R}) \wedge (\mathbf{T} \sqsubseteq \mathbf{S}))$ implique $(\mathbf{T} \sqsubseteq (\mathbf{R} \sqcap \mathbf{S}))$. En posant $\mathbf{T} = (\mathbf{R} \sqcap \mathbf{S})$, nous obtenons $(((\mathbf{R} \sqcap \mathbf{S}) \sqsubseteq \mathbf{R}) \wedge ((\mathbf{R} \sqcap \mathbf{S}) \sqsubseteq \mathbf{S}))$□

– $\mathbf{R} \sqcup \mathbf{S}$ est défini comme le plus petit des majorants de $\mathbf{R}$ et $\mathbf{S}$ (nous ignorons le cas trivial où $(\mathbf{R} = \{\mho\} \ \vee \mathbf{S} = \{\mho\})$).
$((\mathbf{R} \sqsubseteq \mathbf{T}) \ \wedge (\mathbf{S} \sqsubseteq \mathbf{T})) \iff ((\overset{\triangledown}{\mathbf{R}}^{|\mathbf{W}_1} \subseteq \overset{\triangledown}{\mathbf{T}}^{|\mathbf{W}_1}) \ \wedge (\overset{\triangledown}{\mathbf{S}}^{|\mathbf{W}_2} \subseteq \overset{\triangledown}{\mathbf{T}}^{|\mathbf{W}_2}))$
en utilisant le théorème 2.1 nous obtenons
$((\mathbf{R} \sqsubseteq \mathbf{T}) \ \wedge (\mathbf{S} \sqsubseteq \mathbf{T})) \iff ((\overset{\triangledown}{\mathbf{R}}^{|\mathbf{W}} \subseteq \overset{\triangledown}{\mathbf{T}}^{|\mathbf{W}}) \ \wedge (\overset{\triangledown}{\mathbf{S}}^{|\mathbf{W}} \subseteq \overset{\triangledown}{\mathbf{T}}^{|\mathbf{W}}))$
$((\mathbf{R} \sqsubseteq \mathbf{T}) \ \wedge (\mathbf{S} \sqsubseteq \mathbf{T})) \iff (\overset{\triangledown}{\mathbf{R}}^{|\mathbf{W}} \cup \overset{\triangledown}{\mathbf{S}}^{|\mathbf{W}} \subseteq \overset{\triangledown}{\mathbf{T}}^{|\mathbf{W}})$
donc $((\mathbf{R} \sqsubseteq \mathbf{T}) \ \wedge (\mathbf{S} \sqsubseteq \mathbf{T}))$ implique que $((\mathbf{R} \sqcup \mathbf{S}) \sqsubseteq \mathbf{T})$. En posant $\mathbf{T} = (\mathbf{R} \sqcup \mathbf{S})$, nous avons $((\mathbf{R} \sqsubseteq (\mathbf{R} \sqcup \mathbf{S})) \ \wedge (\mathbf{S} \sqsubseteq (\mathbf{R} \sqcup \mathbf{S})))$.□

La conjonction de deux filtres stricts $\mathbf{R}$ et $\mathbf{S}$ est obtenue en construisant l'ensemble des processus étendus des générateurs $\overset{\triangledown}{\mathbf{R}}$ et $\overset{\triangledown}{\mathbf{S}}$ à l'union de leurs ensembles de variables contrôlées. Nous considérons ensuite l'intersection de ces deux processus (ensembles de comportements) ; cette opération pouvant libérer certaines variables (certaines variables deviennent non-contrôlées), nous considérons la réduction de ce processus pour obtenir le générateur de la conjonction de $\mathbf{R}$ et $\mathbf{S}$. Le même mécanisme, avec l'union, est employé pour définir la disjonction de deux filtres stricts $\mathbf{R}$ et $\mathbf{S}$.

La conjonction $\mathbf{R} \sqcap \mathbf{S}$ de deux filtres $\mathbf{R}$ et $\mathbf{S}$ est le plus grand filtre $\mathbf{T} = \mathbf{R} \sqcap \mathbf{S}$ qui accepte les processus autorisés à la fois par $\mathbf{R}$ et par $\mathbf{S}$.

Exemple 7 *Soit x, une variable définie sur l'ensemble des valeurs {0,1,2,3} et u, y, v trois variables prenant leurs valeurs dans {0,1}. Soit* $r \in \mathbb{P}_{\{u,\,x,\,y\}}$, $s \in \mathbb{P}_{\{x,\,y,\,v\}}$, *deux processus réduits définis par :*

$$\begin{aligned} r =& \{b | b(u) \in \{0,1\} \wedge b(x) \in \{0,1\} \wedge b(y) \in \{0,1\}\} \\ & \cup \{(u,1),(x,2),(y,0)\} \\ s =& \{b | b(x) \in \{0,1\} \wedge b(y) \in \{0,1\} \wedge b(v) \in \{0,1\}\} \\ & \cup \{(x,3),(y,1),(v,0)\} \end{aligned}$$

Nous remarquons que $r \rhd \{u,x,y\}$ *; u et y sont libres dans r lorsque x est égal à 0 ou 1. v est libre quelle que soit la valeur de x dans r. Nous avons aussi* $s \rhd \{x,y,v\}$. *y et v sont libres dans s lorsque x est égale à 0 ou 1. Donc u est libre quelle que soit la valeur de x dans s. Nous en déduisons que :*

$$\begin{aligned} p =& r \cap s = \{b | b(u), b(x), b(y), b(v) \in \{0,1\}\} \\ \overset{\triangledown}{p} =& \{b | b(x) \in \{0,1\}\} \end{aligned}$$

La disjonction $\mathbf{R} \sqcup \mathbf{S}$ de deux filtres $\mathbf{R}$ et $\mathbf{S}$ est le plus petit filtre $\mathbf{T} = \mathbf{R} \sqcup \mathbf{S}$ qui accepte les processus autorisés par $\mathbf{R}$ ou par $\mathbf{S}$.

Exemple 8 *Soit x, une variable définie sur l'ensemble des valeurs {0,1,2,3} et u, y, v trois variables prenant leurs valeurs dans {0,1}. Soit* $r \in \mathbb{P}_{\{u,\ x,\ y\}}$, $s \in \mathbb{P}_{\{x,\ y,\ v\}}$, *deux processus réduit définis par :*

$$\begin{aligned} r = & \{b \mid b(u) \in \{0,1\} \ \wedge b(x) \in \{0,1\} \ \wedge b(y) = 0\} \\ s = & \{b \mid b(x) \in \{0,1\} \ \wedge b(y) = 1 \ \wedge b(v) \in \{0,1\}\} \end{aligned}$$

Par conséquent,

$$\begin{aligned} p = & r \cup s = \{b | b(u), b(x), b(y), b(v) \in \{0,1\}\} \\ \overset{\triangledown}{p} = & \{b | b(x) \in \{0,1\}\} \end{aligned}$$

L'opération de relaxation, ainsi que la conjonction et la disjonction sont respectivement proches des opérations d'inclusion, d'intersection, et d'union définies sur les ensembles. Nous allons exhiber leurs différences.

Lemme 2.2 *Les propriétés suivantes sont satisfaites : soient* **R, S, T** *trois filtres stricts,*

$$((\mathbf{R} \cap \mathbf{S}) \subseteq \mathbf{T}) \iff ((vars(\mathbf{T}) \subseteq (vars(\mathbf{R}) \cup vars(\mathbf{S}))) \wedge ((\mathbf{R} \sqcap \mathbf{S}) \sqsubseteq \mathbf{T})) \quad (2.31)$$

par conséquent $$((\mathbf{R} \cap \mathbf{S}) \subseteq (\mathbf{R} \sqcap \mathbf{S}))(\textit{en posant } (\mathbf{R} \sqcap \mathbf{S}) = \mathbf{T}) \quad (2.32)$$

$$((\mathbf{R} \cup \mathbf{S}) \subseteq \mathbf{T}) \iff ((vars(\mathbf{T}) \subseteq (vars(\mathbf{R}) \cap vars(\mathbf{S}))) \wedge ((\mathbf{R} \sqcup \mathbf{S}) \sqsubseteq \mathbf{T})) \quad (2.33)$$

par conséquent $$((\mathbf{R} \cup \mathbf{S}) \subseteq (\mathbf{R} \sqcup \mathbf{S}))(\textit{en posant } (\mathbf{R} \sqcup \mathbf{S}) = \mathbf{T}) \quad (2.34)$$

Preuve :
Soit $\mathbf{X} = vars(\mathbf{R})$,$\mathbf{Y} = vars(\mathbf{S})$,$\mathbf{Z} = vars(\mathbf{T})$, $\mathbf{V} = \mathbf{X} \cup \mathbf{Y} \cup \mathbf{Z}$
+équation 2.31

$$
\begin{array}{rcl}
((\mathbf{R}\cap\mathbf{S})\subseteq\mathbf{T}) & \Longleftrightarrow & (\forall\, p\in\mathbb{P})\,(p\in(\mathbf{R}\cap\mathbf{S})\implies(p\in\mathbf{T}))\\
 & & \text{Posons } \mathbf{W}=vars(p)\cup\mathbf{V}\\
 & & (p\in\mathbf{R}) \iff ((\mathbf{X}\subseteq vars(p))\ \wedge((p^{|\mathbf{W}})_{|\mathbf{V}}\subseteq\overset{\triangledown}{\mathbf{R}}^{|\mathbf{V}}))\\
 & & (p\in\mathbf{S}) \iff ((\mathbf{Y}\subseteq vars(p))\ \wedge((p^{|\mathbf{W}})_{|\mathbf{V}}\subseteq\overset{\triangledown}{\mathbf{S}}^{|\mathbf{V}}))\\
\text{théorème 2.1} & & (p\in\mathbf{T}) \iff ((\mathbf{Z}\subseteq vars(p))\ \wedge((p^{|\mathbf{W}})_{|\mathbf{V}}\subseteq\overset{\triangledown}{\mathbf{T}}^{|\mathbf{V}}))\\
 & & (p\in(\mathbf{R}\cap\mathbf{S})) \iff (((\mathbf{X}\cup\mathbf{Y})\subseteq vars(p))\\
 & & \qquad\wedge((p^{|\mathbf{W}})_{|\mathbf{V}}\subseteq(\overset{\triangledown}{\mathbf{R}}^{|\mathbf{V}}\cap\overset{\triangledown}{\mathbf{S}}^{|\mathbf{V}})))\\
((\mathbf{R}\cap\mathbf{S})\subseteq\mathbf{T}) & \Longleftrightarrow & ((vars(\mathbf{T})\subseteq\mathrm{V})\ \wedge((\overset{\triangledown}{\mathbf{R}}^{|\mathbf{V}}\cap\overset{\triangledown}{\mathbf{S}}^{|\mathbf{V}})\subseteq\overset{\triangledown}{\mathbf{T}}^{|\mathbf{V}}))\\
 & & \text{où } \mathbf{V}=(vars(\mathbf{R})\cup vars(\mathbf{S}))\\
\text{équation 2.28} & & \mathbf{R}\sqcap\mathbf{S}=[[\preceq\uparrow\overset{\triangledown}{RS}]\downarrow_{\subseteq}] \text{ où } RS=(\overset{\triangledown}{\mathbf{R}}^{|\mathbf{X}\cup\mathbf{Y}}\cap\overset{\triangledown}{\mathbf{S}}^{|\mathbf{X}\cup\mathbf{Y}})\\
\text{définition 2.9} & & \mathbf{R}\sqcap\mathbf{S}\sqsubseteq\mathbf{T}\Longleftrightarrow\overset{\triangledown}{RS}^{|\mathbf{W}}\subseteq\overset{\triangledown}{\mathbf{T}}^{|\mathbf{W}} \text{ où } \mathbf{W}=vars(RS)\cup\mathbf{Z}\\
((\mathbf{R}\sqcap\mathbf{S})\sqsubseteq\mathbf{T}) & \Longleftrightarrow & \overset{\triangledown}{RS}^{|\mathbf{V}}\subseteq\overset{\triangledown}{\mathbf{T}}^{|\mathbf{V}} \text{ (nous avons } vars(\mathbf{R}\sqcap\mathbf{S})\subseteq(\mathbf{X}\cup\mathbf{Y}))\\
 & & \text{et donc } \mathbf{W}\subseteq\mathbf{V}\\
((\mathbf{R}\sqcap\mathbf{S})\sqsubseteq\mathbf{T}) & \Longleftrightarrow & (\overset{\triangledown}{\mathbf{R}}^{|\mathbf{X}\cup\mathbf{Y}}\cap\overset{\triangledown}{\mathbf{S}}^{|\mathbf{X}\cup\mathbf{Y}})^{|\mathbf{V}}\subseteq\overset{\triangledown}{\mathbf{T}}^{|\mathbf{V}}\\
((\mathbf{R}\sqcap\mathbf{S})\sqsubseteq\mathbf{T}) & \Longleftrightarrow & (\overset{\triangledown}{\mathbf{R}}^{|\mathbf{V}}\cap\overset{\triangledown}{\mathbf{S}}^{|\mathbf{V}})\subseteq\overset{\triangledown}{\mathbf{T}}^{|\mathbf{V}} \text{ équation 2.5 et équation 2.4}\\
\text{donc} & & \\
((\mathbf{R}\cap\mathbf{S})\subseteq\mathbf{T}) & \Longleftrightarrow & ((vars(\mathbf{T})\subseteq(vars(\mathbf{R})\cup vars(\mathbf{S})))\ \wedge((\mathbf{R}\sqcap\mathbf{S})\sqsubseteq\mathbf{T}))\\
+\text{équation 2.33} & & \\
((\mathbf{R}\cup\mathbf{S})\subseteq\mathbf{T}) & \Longleftrightarrow & (\forall\, p\in\mathbb{P})\,(p\in(\mathbf{R}\cup\mathbf{S})\implies(p\in\mathbf{T}))\\
 & & \text{pour les mêmes raisons, nous avons}\\
((\mathbf{R}\cup\mathbf{S})\subseteq\mathbf{T}) & \Longleftrightarrow & ((vars(\mathbf{T})\subseteq(vars(\mathbf{R})\cap vars(\mathbf{S})))\wedge((\overset{\triangledown}{\mathbf{R}}^{|\mathbf{V}}\cup\overset{\triangledown}{\mathbf{S}}^{|\mathbf{V}})\subseteq\overset{\triangledown}{\mathbf{T}}^{|\mathbf{V}}))\\
((\mathbf{R}\sqcup\mathbf{S})\sqsubseteq\mathbf{T}) & \Longleftrightarrow & (\overset{\triangledown}{\mathbf{R}}^{|\mathbf{V}}\cup\overset{\triangledown}{\mathbf{S}}^{|\mathbf{V}})\subseteq\overset{\triangledown}{\mathbf{T}}^{|\mathbf{V}} \text{ équation 2.5 et équation 2.4}\\
\text{donc} & & \\
((\mathbf{R}\cup\mathbf{S})\subseteq\mathbf{T}) & \Longleftrightarrow & ((vars(\mathbf{T})\subseteq(vars(\mathbf{R})\cap vars(\mathbf{S})))\ \wedge((\mathbf{R}\sqcup\mathbf{S})\sqsubseteq\mathbf{T}))
\end{array}
$$

Définition 2.10 (Complémentaire d'un Filtre) *Le complémentaire* $\widetilde{\mathbf{R}}$ *du filtre* $\mathbf{R}$ *est défini par :*

$$\widetilde{\{\mho\}}=\mathbb{P}^\star,\widetilde{\mathbb{P}^\star}=\{\mho\} \tag{2.35}$$

$$(\mathbf{R}\neq\{\mho\}\wedge\mathbf{R}\neq\mathbb{P}^\star)\Longrightarrow(\widetilde{\mathbf{R}}=_\Delta[[\preceq\uparrow\overset{\widetilde{\triangledown}}{\mathbf{R}}]\downarrow_{\subseteq}]) \tag{2.36}$$

Si $\mathbf{R}\neq\{\mho\}$ *et* $\widetilde{\mathbf{R}}\neq\{\mho\}$ *alors* $\overset{\widetilde{\triangledown}}{\mathbf{R}}=(\mathbb{B}_{vars(\mathbf{R})}\setminus\overset{\triangledown}{\mathbf{R}})$ *est un processus réduit et* $vars(\mathbf{R})=vars(\widetilde{\mathbf{R}})$ *(voir équation 2.1 et propriété 2.3).*

Corollaire 2.3 *Le complémentaire d'un filtre* $\mathbf{R}$ *satisfait* $\widetilde{\mathbf{R}}\subseteq\overline{\mathbf{R}}\cup\{\mho\}$.

Le complémentaire d'un filtre strict $\mathbf{R}$ est le filtre généré par le complémentaire de son générateur $\overset{\triangledown}{\mathbf{R}}$.

Nous exposons maintenant l'un de nos principaux résultats : l'ensemble des filtres est une algèbre booléenne.

Théorème 2.2 *(Φ,$\sqsubseteq$) est une algèbre booléenne avec $\mathbb{P}^\star$ pour supremum, $\{\mho\}$ pour infimum et le complémentaire $\widetilde{\mathbf{R}}$.*

Preuve :
(Φ,$\sqsubseteq$) est un treillis, nous avons alors :

- $\mathbf{R} \sqcup (\mathbf{S} \sqcup \mathbf{T}) = (\mathbf{R} \sqcup \mathbf{S}) \sqcup \mathbf{T}$ (et la propriété duale)
- $\mathbf{R} \sqcup \mathbf{S} = \mathbf{S} \sqcup \mathbf{R}$ (et la propriété duale)
- $\mathbf{R} \sqcup (\mathbf{R} \sqcap \mathbf{S}) = \mathbf{R}$ (et la propriété duale)

Par conséquent, nous avons seulement à prouver que :

- $\widetilde{\mathbf{R}} \sqcap \mathbf{R} = \{\mho\}$; ce qui est une conséquence directe de la définition de la conjonction :
  Dans le cas où $\mathbf{R} = \{\mho\}$ alors $\mathbf{R} \sqcap \{\mho\} = \{\mho\}$ (voir $\sqcap$ équation 2.28)
  ou bien $\mathbf{R} \neq \{\mho\}$; dans ce cas $\overset{\triangledown}{\mathbf{R}} \cap \widetilde{\overset{\triangledown}{\mathbf{R}}}$ est égal à l'ensemble vide (voir la définition de l'opérateur $\sqcap$ dans le lemme 2.1) : $[[\preceq\!\uparrow \overset{\triangledown}{\mho}]\!\downarrow_{\subseteq}] = \{\mho\}$
- $\widetilde{\mathbf{R}} \sqcup \mathbf{R} = \mathbb{P}^\star$; ce qui est une conséquence directe de la définition de la disjonction :
  Dans le cas où $\mathbf{R} = \{\mho\}, \widetilde{\{\mho\}} = \mathbb{P}^\star$ (voir la définition du complémentaire $\widetilde{\mathbf{R}}$) alors $\mathbf{R} \sqcup \{\mho\} = \mathbf{R}$ (voir $\sqcup$ équation 2.30)
  ou bien $\mathbf{R} \neq \{\mho\}$; dans ce cas $\overset{\triangledown}{\mathbf{R}} \cup \widetilde{\overset{\triangledown}{\mathbf{R}}}$ est égale à $\mathbb{B}_{vars(\mathbf{R})}$ dont la réduction est Ω : $[[\preceq\!\uparrow \overset{\triangledown}{\Omega}]\!\downarrow_{\subseteq}] = \mathbb{P}^\star$
- $\mathbf{R} \sqcup (\mathbf{S} \sqcap \mathbf{T}) = (\mathbf{R} \sqcup \mathbf{S}) \sqcap (\mathbf{R} \sqcup \mathbf{T})$
  Si $\mathbf{R} = \{\mho\}$, nous avons alors $\mathbf{R} \sqcup (\mathbf{S} \sqcap \mathbf{T}) = (\mathbf{S} \sqcap \mathbf{T})$, $(\mathbf{R} \sqcup \mathbf{S}) = \mathbf{S}$, $(\mathbf{R} \sqcup \mathbf{T}) = \mathbf{T}$
  Si $\mathbf{S} = \{\mho\}$ (ou commutativement $\mathbf{T} = \{\mho\}$), nous avons alors $(\mathbf{S} \sqcap \mathbf{T}) = \{\mho\}$, alors $\mathbf{R} \sqcup (\mathbf{S} \sqcap \mathbf{T}) = \mathbf{R}$; l'autre partie de l'équation nous donne $(\mathbf{R} \sqcup \mathbf{S}) \sqcap (\mathbf{R} \sqcup \mathbf{T}) = \mathbf{R} \sqcap (\mathbf{R} \sqcup \mathbf{T})$; (Φ,$\sqsubseteq$) étant un treillis, $\mathbf{R} \sqcap (\mathbf{R} \sqcup \mathbf{T}) = \mathbf{R}$.
  Si aucun des filtres $\mathbf{R}$, $\mathbf{S}$, $\mathbf{T}$ n'est égal à $\{\mho\}$, à partir des définitions et du théorème 2.1 nous obtenons :
  $\mathbf{R} \sqcup (\mathbf{S} \sqcap \mathbf{T}) = (\mathbf{R} \sqcup \mathbf{S}) \sqcap (\mathbf{R} \sqcup \mathbf{T})$ si et seulement si
  $(\overset{\triangledown}{\mathbf{R}}^{|\mathbf{V}}) \cup ((\overset{\triangledown}{\mathbf{S}}^{|\mathbf{V}}) \cap (\overset{\triangledown}{\mathbf{T}}^{|\mathbf{V}})) = ((\overset{\triangledown}{\mathbf{R}}^{|\mathbf{V}}) \cup (\overset{\triangledown}{\mathbf{S}}^{|\mathbf{V}})) \cap ((\overset{\triangledown}{\mathbf{R}}^{|\mathbf{V}}) \cup (\overset{\triangledown}{\mathbf{T}}^{|\mathbf{V}}))$
  (où $\mathbf{V} = vars(\mathbf{R}) \cup vars(\mathbf{S}) \cup vars(\mathbf{T})$) □

2.2.4 Masquage de variables dans un filtre

Nous présentons maintenant les règles de masquage de variables dans les filtres. Le concept de masquage de variables d'un filtre sera ensuite utilisé pour définir le concept de masquage dans un contrat.

Définition 2.11 (Masquage de variables dans les filtres) *Soient x une variable, $\mathbf{R}$ un filtre, et $\mathbf{X} =_\Delta vars(\mathbf{R})$. L'élimination existentielle de x dans $\mathbf{R}$, notée $\mathbf{R}_{|\exists x}$, est définie par la projection de $\mathbf{R}$ sur les variables contrôlées différentes de x. L'élimination universelle de x dans $\mathbf{R}$, notée $\mathbf{R}_{|\forall x}$, est définie par un processus générateur caractérisé par la projection de l'ensemble des comportements de $\overset{\triangledown}{\mathbf{R}}$ sur $\mathbf{X}\backslash\{x\}$ pour lesquels x est libre dans $\overset{\triangledown}{\mathbf{R}}$.*

$$\mathbf{R}_{|\exists x} =_\Delta \begin{cases} \widehat{[(\overset{\triangledown}{\mathbf{R}})_{|\mathbf{X}\backslash\{x\}}]}, & x \in \mathbf{X} \\ \mathbf{R}, & sinon \end{cases} \qquad \mathbf{R}_{|\forall x} =_\Delta \widetilde{\widetilde{\mathbf{R}}_{|\exists x}}$$

Propriété 2.9 $\mathbf{R}_{|\forall x} \sqsubseteq \mathbf{R} \sqsubseteq \mathbf{R}_{|\exists x}$

Preuve : Par définition, le générateur de $\mathbf{R}_{|\exists x}$ est la restriction du générateur de $\mathbf{R}$, alors (théorème 2.1) tous les processus de $\mathbf{R}$ appartiennent également à $\mathbf{R}_{|\exists x}$. Donc $\widetilde{\mathbf{R}} \sqsubseteq \widetilde{\mathbf{R}}_{|\exists x}$ et (théorème 2.2) $\widetilde{\widetilde{\mathbf{R}}_{|\exists x}} \sqsubseteq \widetilde{\widetilde{\mathbf{R}}} = \mathbf{R}$

Exemple 9 *Soit $\mathbf{R}$, un filtre généré par la contrainte $((x > 0) \Rightarrow (y > 0)) \wedge ((y > 0) \Rightarrow (z > 0))$ défini sur l'ensemble de variables $\{x, y, z\}$. Alors $\mathbf{R}_{|\exists x}$ est généré par $((y > 0) \Rightarrow (z > 0))$.*

D'autre part, le complémentaire de ce même filtre est généré par la négation de la contrainte : $((x > 0) \wedge (y \leq 0)) \vee ((y > 0) \wedge (z \leq 0))$. Ainsi le complémentaire de la contrainte projetée sur l'ensemble de variables $\{y, z\}$ satisfait : $((y \leq 0)) \vee ((y > 0) \wedge (z \leq 0))$. Alors $\mathbf{R}_{|\forall x}$ est généré par $((y > 0) \wedge (z > 0))$, c'est-à-dire la négation de $((y \leq 0)) \vee ((y > 0) \wedge (z \leq 0))$.

2.3 Une algèbre de contrats

Nous définissons le concept de contrat basé sur les notions d'hypothèses/garanties. Nous proposons une relation d'équivalence sur les contrats de laquelle découle une structure mathématiquement riche sur l'ensemble des contrats.

2.3.1 Sémantique des contrats

Définition 2.12 (Contrat) *Un* contrat $\mathbf{C} = (\mathbf{A},\mathbf{G})$ *est une paire de filtres.* $vars(\mathbf{C})$ *est l'ensemble des variables contrôlées par les filtres* $\mathbf{A}$ *ou* $\mathbf{G}$*, défini par* $vars(\mathbf{C}) = vars(\mathbf{A}) \cup vars(\mathbf{G})$*.* $\mathbb{C} = \Phi \times \Phi$ *est l'ensemble de tous les contrats.*

Une hypothèse $\mathbf{A}$ est une propriété faite sur les comportements de l'environnement (couramment exprimée sur les variables d'entrée d'un processus) et qui définit ainsi l'ensemble de comportements que le processus doit prendre en considération. Les garanties définissent les propriétés qui doivent être satisfaites par un processus exécuté dans un environnement dont les comportements satisfont $\mathbf{A}$.

La figure 2.12 décrit un processus p *satisfaisant* le contrat $(\mathbf{A},\mathbf{G})$ ($\widehat{[p]}$ est le filtre généré par la réduction de p). Un processus p *satisfait* un contrat $\mathbf{C} = (\mathbf{A},\mathbf{G})$ si tous ses comportements acceptés par $\mathbf{A}$ (c'est-à-dire, les comportements de p caractérisant également les comportements de certains processus appartenant à $\mathbf{A}$), sont aussi acceptés par $\mathbf{G}$; ce qui est caractérisé de manière plus précise et plus formelle par la définition suivante.

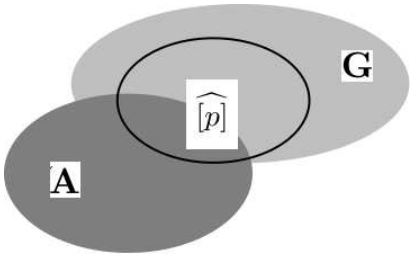


FIG. 2.12 – Un processus p satisfaisant $(\mathbf{A},\mathbf{G})$.

2.3.2 Satisfaction des contrats

Définition 2.13 (Satisfaction) *Soit* $\mathbf{C} = (\mathbf{A},\mathbf{G})$ *un contrat,* p *un processus :* $p \vDash \mathbf{C} \iff (\widehat{[p]} \sqcap \mathbf{A}) \sqsubseteq \mathbf{G}$.

Corollaire 2.4 $p \vDash \mathbf{C} \iff \widehat{[p]} \sqsubseteq (\widetilde{A} \sqcup \mathbf{G})$

Preuve : En utilisant les propriétés de l'algèbre booléenne sur les filtres : $((\widehat{[p]} \sqcap \mathbf{A}) \sqsubseteq \mathbf{G}) \iff (((\widehat{[p]} \sqcap \mathbf{A}) \sqcap \widetilde{\mathbf{G}}) = \{\mho\}) \iff (\widehat{[p]} \sqsubseteq (\widetilde{A} \sqcup \mathbf{G}))$ □

Nous définissons une relation de pré-ordre qui permet de comparer deux processus. Si $(\mathbf{A}_1,\mathbf{G}_1)$ est plus *fin* que $(\mathbf{A}_2,\mathbf{G}_2)$, alors $(\mathbf{A}_1,\mathbf{G}_1)$ renforce les propriétés modélisées par $(\mathbf{A}_2,\mathbf{G}_2)$. Ainsi, $(\mathbf{A}_2,\mathbf{G}_2)$ peut être substitué par $(\mathbf{A}_1,\mathbf{G}_1)$.

Définition 2.14 (Satisfaction du pré-ordre) *Un contrat* $(\mathbf{A}_1,\mathbf{G}_1)$ *est plus fin qu'un contrat* $(\mathbf{A}_2,\mathbf{G}_2)$*, noté* $(\mathbf{A}_1,\mathbf{G}_1) \rightsquigarrow (\mathbf{A}_2,\mathbf{G}_2)$*, si et seulement si tous les processus qui satisfont le contrat* $(\mathbf{A}_1,\mathbf{G}_1)$ *satisfont également le contrat* $(\mathbf{A}_2,\mathbf{G}_2)$ *:*

$$(\mathbf{A}_1, \mathbf{G}_1) \rightsquigarrow (\mathbf{A}_2, \mathbf{G}_2) \Longleftrightarrow (\forall\ p \in \mathbb{P})((p \vDash (\mathbf{A}_1, \mathbf{G}_1)) \Longrightarrow (p \vDash (\mathbf{A}_2, \mathbf{G}_2))) \quad (2.37)$$

Le pré-ordre vérifie la propriété suivante :

Lemme 2.3 *Soient deux contrats* $(\mathbf{A}_1,\mathbf{G}_1)$*, et* $(\mathbf{A}_2,\mathbf{G}_2)$*. La relation de pré-ordre sur ces contrats satisfait la propriété suivante :*

$$(\mathbf{A}_1, \mathbf{G}_1) \rightsquigarrow (\mathbf{A}_2, \mathbf{G}_2) \Longleftrightarrow (\widetilde{\mathbf{A}_1} \sqcup \mathbf{G}_1) \sqsubseteq (\widetilde{\mathbf{A}_2} \sqcup \mathbf{G}_2) \quad (2.38)$$

Preuve :
Nous avons (corollaire 2.4) $(\mathbf{A}_1,\mathbf{G}_1) \rightsquigarrow (\mathbf{A}_2,\mathbf{G}_2)$
$\Longleftrightarrow (\forall\ p \in \mathbb{P})\ ((\widehat{[p]} \sqsubseteq (\widetilde{\mathbf{A}_1} \sqcup \mathbf{G}_1)) \Longrightarrow (\widehat{[p]} \sqsubseteq (\widetilde{\mathbf{A}_2} \sqcup \mathbf{G}_2)))$
$\Longrightarrow$ Posons p égal au processus générateur de$(\widetilde{\mathbf{A}_1} \sqcup \mathbf{G}_1)$

$\Longleftarrow (\forall\ p \in \mathbb{P})\ ((\widehat{[p]} \sqsubseteq (\widetilde{\mathbf{A}_1} \sqcup \mathbf{G}_1)) \Longrightarrow (\widehat{[p]} \sqsubseteq (\widetilde{\mathbf{A}_1} \sqcup \mathbf{G}_1) \sqsubseteq (\widetilde{\mathbf{A}_2} \sqcup \mathbf{G}_2)))$

□
La relation suivante caractérise l'équivalence de contrats. Elle permet de définir l'ensemble des contrats qui sont satisfaits par un même ensemble de processus.

Définition 2.15 (Equivalence de filtrage) *Deux contrats* $\mathbf{C}_1 = (\mathbf{A}_1,\mathbf{G}_1)$ *et* $\mathbf{C}_2 = (\mathbf{A}_2,\mathbf{G}_2)$ *satisfont la relation* d'équivalence de filtrage*, notée* $(\mathbf{A}_1,\mathbf{G}_1) \leftrightsquigarrow (\mathbf{A}_2,\mathbf{G}_2)$ *si et seulement si :*
$((\mathbf{A}_1,\mathbf{G}_1) \rightsquigarrow (\mathbf{A}_2,\mathbf{G}_2)) \ \wedge\ ((\mathbf{A}_2,\mathbf{G}_2) \rightsquigarrow (\mathbf{A}_1,\mathbf{G}_1))$.

Corollaire 2.5 *Deux contrats* $\mathbf{C}_1 = (\mathbf{A}_1,\mathbf{G}_1)$ *et* $\mathbf{C}_2 = (\mathbf{A}_2,\mathbf{G}_2)$ *satisfont la relation* d'équivalence de filtrage *si et seulement si* $(\widetilde{\mathbf{A}_1} \sqcup \mathbf{G}_1) = (\widetilde{\mathbf{A}_2} \sqcup \mathbf{G}_2)$.

Preuve : Découle directement du corollaire 2.4 □

2.3.3 Une relation d'ordre partiel sur les contrats

Le raffinement de contrats équivaut à relâcher les hypothèses et renforcer les garanties sous les hypothèses initiales. Intuitivement, pour un processus p qui satisfait un contrat $\mathbf{C}$, si $\mathbf{C}$ raffine $\mathbf{D}$, alors p satisfait $\mathbf{D}$. Notre relation de raffinement formalise la substituabilité d'un contrat par un autre.

Définition 2.16 (Raffinement de contrats) *Soient* $\mathbf{C}_1 = (\mathbf{A}_1,\mathbf{G}_1)$ *et* $\mathbf{C}_2 = (\mathbf{A}_2,\mathbf{G}_2)$ *deux contrats. Le contrat* $\mathbf{C}_1$ raffine *le contrat* $\mathbf{C}_2$*, noté* $\mathbf{C}_1 \preccurlyeq \mathbf{C}_2$*, si et seulement si les trois propriétés suivantes sont satisfaites :*

(i) $(\mathbf{A}_1,\mathbf{G}_1) \rightsquigarrow (\mathbf{A}_2,\mathbf{G}_2)$,
(a) $(\mathbf{A}_2 \sqsubseteq \mathbf{A}_1)$,
(c) $\mathbf{G}_1 \sqsubseteq \mathbf{A}_1 \sqcup \mathbf{G}_2$.

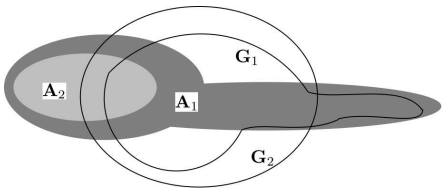


FIG. 2.13 – Raffinement de contrats.

La figure 2.13 décrit un contrat $(\mathbf{A}_1,\mathbf{G}_1)$ qui raffine un contrat $(\mathbf{A}_2,\mathbf{G}_2)$. Parmi les contrats équivalents en terme de filtrage, pouvant être utilisés pour raffiner un contrat $(\mathbf{A}_2,\mathbf{G}_2)$, nous choisisons les contrats $(\mathbf{A}_1,\mathbf{G}_1)$ qui admettent des hypothèses d'exécution plus larges que $(\mathbf{A}_2,\mathbf{G}_2)$ $(\mathbf{A}_2 \sqsubseteq \mathbf{A}_1)$ et dont les garanties acceptent moins de processus que $\mathbf{A}_1 \sqcup \mathbf{G}_2$. Cependant, d'autres choix auraient pu être faits.

Par définition de la relation de pré-ordre, nous pouvons exprimer la relation de raffinement de contrats dans l'algèbre de filtres de la manière suivante :

Lemme 2.4 $(\mathbf{A}_1,\mathbf{G}_1) \preccurlyeq (\mathbf{A}_2,\mathbf{G}_2)$ *si et seulement si les trois propiétés suivantes sont satisfaites :*

(a) $\mathbf{A}_2 \sqsubseteq \mathbf{A}_1$,
(b) $(\mathbf{A}_2 \sqcap \mathbf{G}_1) \sqsubseteq \mathbf{G}_2$,
(c) $\mathbf{G}_1 \sqsubseteq \mathbf{A}_1 \sqcup \mathbf{G}_2$.

Preuve :
L'item (i) (dans la définition 2.16) est équivalent à $(\mathbf{A}_2 \sqsubseteq (\mathbf{A}_1 \sqcup \mathbf{G}_2)) \wedge ((\mathbf{A}_2 \sqcap \mathbf{G}_1) \sqsubseteq \mathbf{G}_2)$ (lemme 2.3 et propriétés de l'algèbre booléenne sur les filtres) ; alors (i) et (a) sont équivalents à (a) et (b). □

Propriété 2.10 $(\mathbb{C},\preccurlyeq)$ *est un ordre partiel.*

Preuve : $\preccurlyeq$ est clairement réflexive ; il ne reste plus qu'à montrer la transitivité et l'antisymétrie.

– Transitivité :

(a) $(\mathbf{A}_3 \sqsubseteq \mathbf{A}_1)$: avec (a) dans le lemme 2.4 nous avons $\mathbf{A}_3 \sqsubseteq \mathbf{A}_2 \sqsubseteq \mathbf{A}_1$

(b) $((\mathbf{A}_3 \sqcap \mathbf{G}_1) \sqsubseteq \mathbf{G}_3)$:

$$\begin{array}{rll} ((\mathbf{A}_2 \sqcap \mathbf{G}_1) \sqsubseteq \mathbf{G}_2) & \Longrightarrow & ((\mathbf{A}_3 \sqcap \mathbf{A}_2 \sqcap \mathbf{G}_1) \sqsubseteq (\mathbf{A}_3 \sqcap \mathbf{G}_2)) \text{ (treillis)} \\ & \Longrightarrow & ((\mathbf{A}_3 \sqcap \mathbf{G}_1) \sqsubseteq ((\mathbf{A}_3 \sqcap \mathbf{G}_2) \sqsubseteq \mathbf{G}_3)) \\ & & (\mathbf{A}_3 \sqsubseteq \mathbf{A}_2 \text{ et (b) dans le lemme 2.4}) \end{array}$$

(c) $(\mathbf{G}_1 \sqsubseteq \mathbf{A}_1 \sqcup \mathbf{G}_3)$:

$$\mathbf{G}_1 \sqsubseteq (\mathbf{A}_1 \sqcup \mathbf{G}_2) \sqsubseteq (\mathbf{A}_1 \sqcup (\mathbf{A}_2 \sqcup \mathbf{G}_3)) = (\mathbf{A}_1 \sqcup \mathbf{G}_3)$$

– Antisymétrie : avec (a) dans le lemme 2.4 nous avons $\mathbf{A}_1 = \mathbf{A}_2$. En appliquant cette égalité et les propriétés de l'algèbre booléenne sur les filtres à (b) et (c) dans le lemme 2.4, nous obtenons :

$$\begin{array}{rll} ((\mathbf{G}_1 \sqsubseteq (\mathbf{A}_1 \sqcup \mathbf{G}_2)) \wedge (\mathbf{G}_2 \sqsubseteq (\mathbf{A}_1 \sqcup \mathbf{G}_1))) & \Longrightarrow & ((\mathbf{A}_1 \sqcup \mathbf{G}_1) = (\mathbf{A}_1 \sqcup \mathbf{G}_2)) \\ (((\mathbf{A}_1 \sqcap \mathbf{G}_2) \sqsubseteq \mathbf{G}_1) \wedge ((\mathbf{A}_1 \sqcap \mathbf{G}_1) \sqsubseteq \mathbf{G}_2)) & \Longrightarrow & ((\mathbf{A}_1 \sqcap \mathbf{G}_1) = (\mathbf{A}_1 \sqcap \mathbf{G}_2)) \\ & \Longrightarrow & (\mathbf{G}_1 = \mathbf{G}_2) \end{array}$$

et donc $\mathbf{C}_1 = \mathbf{C}_2$

□

2.3.4 Une structure de treillis pour l'ensemble des contrats

La relation de raffinement ($\preccurlyeq$) définie sur l'ensemble des contrats, définira un treillis sur cet ensemble. Dans ce treillis, l'union (ou disjonction) de deux contrats est définie par le plus petit contrat raffiné par chacun de ces deux contrats (la borne supérieure). L'intersection (ou conjonction) de deux contrats est définie par le plus grand contrat minorant chacun de deux contrats (la borne inférieure).

Lemme 2.5-a. Un contrat $\mathbf{D} = (\mathbf{B},\mathbf{H})$ est un minorant de deux contrats $\mathbf{C}_1 = (\mathbf{A}_1,\mathbf{G}_1)$ et $\mathbf{C}_2 = (\mathbf{A}_2,\mathbf{G}_2)$ si et seulement si il satisfait les propriétés suivantes :

$$\mathbf{B} = \mathbf{A}_1 \sqcup \mathbf{A}_2 \sqcup \mathbf{B} \tag{2.39}$$

$$\begin{aligned} \mathbf{H} = \mathbf{H} \sqcap ((\mathbf{A}_1 \sqcap \widetilde{\mathbf{A}_2} \sqcap \mathbf{G}_1) \sqcup (\widetilde{\mathbf{A}_1} \sqcap \mathbf{A}_2 \sqcap \mathbf{G}_2) \sqcup (\mathbf{G}_1 \sqcap \mathbf{G}_2) \\ \sqcup (\mathbf{B} \sqcap (\widetilde{\mathbf{A}_1} \sqcup \mathbf{G}_1) \sqcap (\widetilde{\mathbf{A}_2} \sqcup \mathbf{G}_2))) \end{aligned} \tag{2.40}$$

Preuve : $(\mathbf{D} \preccurlyeq \mathbf{C}_1) \wedge (\mathbf{D} \preccurlyeq \mathbf{C}_2) \Longleftrightarrow$ (lemme 2.4)

$$\begin{array}{lcl} \mathbf{A}_1 \sqsubseteq \mathbf{B} & \wedge & \mathbf{A}_2 \sqsubseteq \mathbf{B} \\ ((\mathbf{H} \sqcap \mathbf{A}_1) \sqsubseteq \mathbf{G}_1) & \wedge & ((\mathbf{H} \sqcap \mathbf{A}_2) \sqsubseteq \mathbf{G}_2) \\ \mathbf{H} \sqsubseteq (\mathbf{B} \sqcup \mathbf{G}_1) & \wedge & \mathbf{H} \sqsubseteq (\mathbf{B} \sqcup \mathbf{G}_2) \end{array}$$

$\Longleftrightarrow$ (en utilisant les propriétés du treillis et de l'algèbre booléenne)
$\mathbf{B} = \mathbf{A}_1 \sqcup \mathbf{A}_2 \sqcup \mathbf{B}$
$\mathbf{H} = \mathbf{H} \sqcap ((\mathbf{B} \sqcap (\widetilde{\mathbf{A}_1} \sqcup \mathbf{G}_1) \sqcap (\widetilde{\mathbf{A}_2} \sqcup \mathbf{G}_2)) \sqcup (\mathbf{G}_1 \sqcap \mathbf{G}_2))$

$\Longleftrightarrow$ (en introduisant la première relation caractérisant $\mathbf{B}$ dans la relation définissant $\mathbf{H}$)
$\mathbf{B} = \mathbf{A}_1 \sqcup \mathbf{A}_2 \sqcup \mathbf{B}$
$\mathbf{H} = \mathbf{H} \sqcap (((\mathbf{A}_1 \sqcup \mathbf{A}_2 \sqcup \mathbf{B}) \sqcap (\widetilde{\mathbf{A}_1} \sqcup \mathbf{G}_1) \sqcap (\widetilde{\mathbf{A}_2} \sqcup \mathbf{G}_2)) \sqcup (\mathbf{G}_1 \sqcap \mathbf{G}_2))$
Nous obtenons avec les règles de l'algèbre booléenne sur les filtres :
$\mathbf{H} = \mathbf{H} \sqcap ((\mathbf{A}_1 \sqcap \widetilde{\mathbf{A}_2} \sqcap \mathbf{G}_1) \sqcup (\widetilde{\mathbf{A}_1} \sqcap \mathbf{A}_2 \sqcap \mathbf{G}_2) \sqcup (\mathbf{G}_1 \sqcap \mathbf{G}_2)$
. $\sqcup (\mathbf{B} \sqcap (\widetilde{\mathbf{A}_1} \sqcup \mathbf{G}_1) \sqcap (\widetilde{\mathbf{A}_2} \sqcup \mathbf{G}_2)))$
□

Lemme 2.6-a. Un contrat $\mathbf{D} = (\mathbf{B},\mathbf{H})$ est un majorant de deux contrats $\mathbf{C}_1 = (\mathbf{A}_1,\mathbf{G}_1)$ et $\mathbf{C}_2 = (\mathbf{A}_2,\mathbf{G}_2)$ si et seulement si il satisfait les propriétés suivantes :

$$\mathbf{B} = \mathbf{A}_1 \sqcap \mathbf{A}_2 \sqcap \mathbf{B} \quad (2.41)$$

$$\mathbf{H} = (\widetilde{\mathbf{A}_1} \sqcap \mathbf{G}_1) \sqcup (\widetilde{\mathbf{A}_2} \sqcap \mathbf{G}_2) \sqcup (\mathbf{A}_1 \sqcap \mathbf{A}_2 \sqcap (\mathbf{G}_1 \sqcup \mathbf{G}_2) \sqcap \mathbf{B}) \sqcup \mathbf{H} \quad (2.42)$$

Preuve : $(\mathbf{C}_1 \preccurlyeq \mathbf{D}) \wedge (\mathbf{C}_2 \preccurlyeq \mathbf{D}) \Longleftrightarrow$ (lemme 2.4)

$$\begin{array}{lcl} \mathbf{B} \sqsubseteq \mathbf{A}_1 & \wedge & \mathbf{B} \sqsubseteq \mathbf{A}_2 \\ (\mathbf{B} \sqcap \mathbf{G}_1) \sqsubseteq \mathbf{H} & \wedge & (\mathbf{B} \sqcap \mathbf{G}_2) \sqsubseteq \mathbf{H} \\ \mathbf{G}_1 \sqsubseteq \mathbf{A}_1 \sqcup \mathbf{H} & \wedge & \mathbf{G}_2 \sqsubseteq \mathbf{A}_2 \sqcup \mathbf{H} \end{array}$$

$\Longleftrightarrow$ (en utilisant les propriétés du treillis et de l'algèbre booléenne)
$\mathbf{B} = \mathbf{A}_1 \sqcap \mathbf{A}_2 \sqcap \mathbf{B}$
$\mathbf{H} = (\widetilde{\mathbf{A}_1} \sqcap \mathbf{G}_1) \sqcup (\widetilde{\mathbf{A}_2} \sqcap \mathbf{G}_2) \sqcup (\mathbf{B} \sqcap (\mathbf{G}_1 \sqcup \mathbf{G}_2)) \sqcup \mathbf{H}$

$\Longleftrightarrow$ (en introduisant la première relation caractérisant $\mathbf{B}$ dans la relation définissant $\mathbf{H}$)
$\mathbf{B} = \mathbf{A}_1 \sqcap \mathbf{A}_2 \sqcap \mathbf{B}$
$\mathbf{H} = (\widetilde{\mathbf{A}_1} \sqcap \mathbf{G}_1) \sqcup (\widetilde{\mathbf{A}_2} \sqcap \mathbf{G}_2) \sqcup (\mathbf{A}_1 \sqcap \mathbf{A}_2 \sqcap \mathbf{B} \sqcap (\mathbf{G}_1 \sqcup \mathbf{G}_2)) \sqcup \mathbf{H}$
□

Lemme 2.5 (Borne inférieure de contrats) *Deux contrats* $\mathbf{C}_1 = (\mathbf{A}_1,\mathbf{G}_1)$ *et* $\mathbf{C}_2 = (\mathbf{A}_2,\mathbf{G}_2)$ *ont une borne inférieure* $\mathbf{C} = (\mathbf{A},\mathbf{G})$*, notée* $(\mathbf{C}_1 \Downarrow \mathbf{C}_2)$*, définie*

par :

$$\mathbf{A} = \mathbf{A}_1 \sqcup \mathbf{A}_2 \tag{2.43}$$

$$\mathbf{G} = ((\mathbf{A}_1 \sqcap \widetilde{\mathbf{A}_2} \sqcap \mathbf{G}_1) \sqcup (\widetilde{\mathbf{A}_1} \sqcap \mathbf{A}_2 \sqcap \mathbf{G}_2) \sqcup (\mathbf{G}_1 \sqcap \mathbf{G}_2)) \tag{2.44}$$

Preuve :

- $\mathbf{C} = (\mathbf{A},\mathbf{G})$ est une borne inférieure des contrats $(\mathbf{A}_1,\mathbf{G}_1)$ et $(\mathbf{A}_2,\mathbf{G}_2)$.
  Alors $\mathbf{C}$ est également un minorant de $(\mathbf{A}_1,\mathbf{G}_1)$ et $(\mathbf{A}_2,\mathbf{G}_2)$.
  $\mathbf{C}$ est un minorant si et seulement si (lemme 2.5-a)
  $\mathbf{A} = \mathbf{A} \sqcup \mathbf{A}_1 \sqcup \mathbf{A}_2$ (satisfait par l'équation 2.43)
  $\mathbf{G} = \mathbf{G} \sqcap ((\mathbf{A}_1 \sqcap \widetilde{\mathbf{A}_2} \sqcap \mathbf{G}_1) \sqcup (\widetilde{\mathbf{A}_1} \sqcap \mathbf{A}_2 \sqcap \mathbf{G}_2) \sqcup (\mathbf{G}_1 \sqcap \mathbf{G}_2)$
  $\sqcup (\mathbf{A} \sqcap (\widetilde{\mathbf{A}_1} \sqcup \mathbf{G}_1) \sqcap (\widetilde{\mathbf{A}_2} \sqcup \mathbf{G}_2)))$
  Or $\mathbf{G} = ((\mathbf{A}_1 \sqcap \widetilde{\mathbf{A}_2} \sqcap \mathbf{G}_1) \sqcup (\widetilde{\mathbf{A}_1} \sqcap \mathbf{A}_2 \sqcap \mathbf{G}_2) \sqcup (\mathbf{G}_1 \sqcap \mathbf{G}_2))$ (équation 2.44), nous obtenons alors par substitution : $\mathbf{C}$ est une borne inférieure si et seulement si :
  $\mathbf{G} = \mathbf{G} \sqcap (\mathbf{G} \sqcup (\mathbf{A} \sqcap (\widetilde{\mathbf{A}_1} \sqcup \mathbf{G}_1) \sqcap (\widetilde{\mathbf{A}_2} \sqcup \mathbf{G}_2)))$
  Ce qui satisfait l'équation 2.40.
  Donc $\mathbf{C}$ est un minorant de $(\mathbf{A}_1,\mathbf{G}_1)$ et $(\mathbf{A}_2,\mathbf{G}_2)$.
  □

- si $\mathbf{D} = (\mathbf{B},\mathbf{H})$ est un minorant de $\mathbf{C}_1$ et $\mathbf{C}_2$ alors $\mathbf{D}$ raffine $\mathbf{C}$:
  $\mathbf{D}$ est un minorant de $\mathbf{C}_1$ et $\mathbf{C}_2$ si et seulement si (lemme 2.5-a) :
  $\mathbf{B} = \mathbf{A}_1 \sqcup \mathbf{A}_2 \sqcup \mathbf{B}$
  $\mathbf{H} = \mathbf{H} \sqcap ((\mathbf{A}_1 \sqcap \widetilde{\mathbf{A}_2} \sqcap \mathbf{G}_1) \sqcup (\widetilde{\mathbf{A}_1} \sqcap \mathbf{A}_2 \sqcap \mathbf{G}_2) \sqcup (\mathbf{G}_1 \sqcap \mathbf{G}_2)$
  $\sqcup (\mathbf{B} \sqcap (\widetilde{\mathbf{A}_1} \sqcup \mathbf{G}_1) \sqcap (\widetilde{\mathbf{A}_2} \sqcup \mathbf{G}_2)))$
  $\mathbf{D}$ est un minorant de $\mathbf{C}_1$ et $\mathbf{C}_2$ si et seulement si (en remplacant $\mathbf{A}_1 \sqcup \mathbf{A}_2$ par $\mathbf{A}$ et $((\mathbf{A}_1 \sqcap \widetilde{\mathbf{A}_2} \sqcap \mathbf{G}_1) \sqcup (\widetilde{\mathbf{A}_1} \sqcap \mathbf{A}_2 \sqcap \mathbf{G}_2) \sqcup (\mathbf{G}_1 \sqcap \mathbf{G}_2))$ par $\mathbf{G}$)
  $\mathbf{B} = \mathbf{A} \sqcup \mathbf{B}$
  $\mathbf{H} = \mathbf{H} \sqcap (\mathbf{G} \sqcup (\mathbf{B} \sqcap (\widetilde{\mathbf{A}_1} \sqcup \mathbf{G}_1) \sqcap (\widetilde{\mathbf{A}_2} \sqcup \mathbf{G}_2)))$
  $\mathbf{D}$ raffine $\mathbf{C}$ si et seulement si (lemme 2.4) :

  (a) $\mathbf{A} \sqsubseteq \mathbf{B}$ (cette propriété est satisfaite)
  (b) $(\mathbf{A} \sqcap \mathbf{H}) \sqsubseteq \mathbf{G}$. Nous avons :

  $$\begin{array}{lcl} (\mathbf{A} \sqcap \mathbf{H}) & = & \mathbf{A} \sqcap (\mathbf{G} \sqcup (\mathbf{B} \sqcap (\widetilde{\mathbf{A}_1} \sqcup \mathbf{G}_1) \sqcap (\widetilde{\mathbf{A}_2} \sqcup \mathbf{G}_2))) \\ & & (\text{or } \mathbf{A} \sqsubseteq \mathbf{B}) \\ (\mathbf{A} \sqcap \mathbf{H}) & = & \mathbf{A} \sqcap (\mathbf{G} \sqcup (\mathbf{A} \sqcap (\widetilde{\mathbf{A}_1} \sqcup \mathbf{G}_1) \sqcap (\widetilde{\mathbf{A}_2} \sqcup \mathbf{G}_2))) \\ & & (\text{or } (\mathbf{A} \sqcap (\widetilde{\mathbf{A}_1} \sqcup \mathbf{G}_1) \sqcap (\widetilde{\mathbf{A}_2} \sqcup \mathbf{G}_2)) \sqsubseteq \mathbf{G}) \\ (\mathbf{A} \sqcap \mathbf{H}) & = & \mathbf{A} \sqcap \mathbf{G} \sqsubseteq \mathbf{G} \\ \text{Donc} & & \mathbf{A} \sqcap \mathbf{H} \sqsubseteq \mathbf{G} \end{array}$$

□

(c) $\mathbf{H} \sqsubseteq \mathbf{B} \sqcup \mathbf{G}$. Nous avons bien :
$\mathbf{H} \sqsubseteq (\mathbf{G} \sqcup (\mathbf{B} \sqcap (\widetilde{\mathbf{A}_1} \sqcup \mathbf{G}_1) \sqcap (\widetilde{\mathbf{A}_2} \sqcup \mathbf{G}_2)))$

□

Lemme 2.6 (Borne supérieure de contrats) *Deux contrats* $\mathbf{C}_1 = (\mathbf{A}_1,\mathbf{G}_1)$ *et* $\mathbf{C}_2 = (\mathbf{A}_2,\mathbf{G}_2)$ *ont une borne supérieure* $\mathbf{C} = (\mathbf{A},\mathbf{G})$*, notée* $(\mathbf{C}_1 \Uparrow \mathbf{C}_2)$*, definie par :*

$$\mathbf{A} = \mathbf{A}_1 \sqcap \mathbf{A}_2 \tag{2.45}$$

$$\mathbf{G} = (\widetilde{\mathbf{A}_1} \sqcap \mathbf{G}_1) \sqcup (\widetilde{\mathbf{A}_2} \sqcap \mathbf{G}_2) \sqcup (\mathbf{A}_1 \sqcap \mathbf{G}_2) \sqcup (\mathbf{A}_2 \sqcap \mathbf{G}_1) \tag{2.46}$$

Preuve :

- $\mathbf{C} = (\mathbf{A},\mathbf{G})$ est une borne supérieure des contrats $(\mathbf{A}_1,\mathbf{G}_1)$ et $(\mathbf{A}_2,\mathbf{G}_2)$:
  Alors $\mathbf{C}$ est également un majorant des contrats $(\mathbf{A}_1,\mathbf{G}_1)$ et $(\mathbf{A}_2,\mathbf{G}_2)$.
  $\mathbf{C}$ est une borne supérieure si et seulement si (lemme 2.6-a) :
  $\mathbf{A} = \mathbf{A}_1 \sqcap \mathbf{A}_2 \sqcap \mathbf{A}$, (satisfait par l'équation 2.45)
  $\mathbf{G} = (\widetilde{\mathbf{A}_1} \sqcap \mathbf{G}_1) \sqcup (\widetilde{\mathbf{A}_2} \sqcap \mathbf{G}_2) \sqcup (\mathbf{A}_1 \sqcap \mathbf{A}_2 \sqcap (\mathbf{G}_1 \sqcup \mathbf{G}_2) \sqcap \mathbf{A}) \sqcup \mathbf{G}$
  Nous avons :
  $(\widetilde{\mathbf{A}_1} \sqcap \mathbf{G}_1) \sqcup (\widetilde{\mathbf{A}_2} \sqcap \mathbf{G}_2) \sqcup (\mathbf{A}_1 \sqcap \mathbf{A}_2 \sqcap (\mathbf{G}_1 \sqcup \mathbf{G}_2) \sqcap \mathbf{A}) \sqsubseteq \mathbf{G}$ □
- si $\mathbf{D} = (\mathbf{B},\mathbf{H})$ est un majorant de $\mathbf{C}_1$ et $\mathbf{C}_2$ alors $\mathbf{C}$ raffine $\mathbf{D}$:
  $\mathbf{D}$ est un majorant de $\mathbf{C}_1$ et $\mathbf{C}_2$ si et seulement si (lemme 2.6-a)
  $\mathbf{B} = \mathbf{A}_1 \sqcap \mathbf{A}_2 \sqcap \mathbf{B}$
  $\mathbf{H} = (\widetilde{\mathbf{A}_1} \sqcap \mathbf{G}_1) \sqcup (\widetilde{\mathbf{A}_2} \sqcap \mathbf{G}_2) \sqcup (\mathbf{A}_1 \sqcap \mathbf{A}_2 \sqcap (\mathbf{G}_1 \sqcup \mathbf{G}_2) \sqcap \mathbf{B}) \sqcup \mathbf{H}$
  $\mathbf{C}$ raffine $\mathbf{D}$ si et seulement si (lemme 2.4) :

  (a) $\mathbf{B} \sqsubseteq \mathbf{A}$ (cette propriété est satisfaite)
  (b) $(\mathbf{B} \sqcap \mathbf{G}) \sqsubseteq \mathbf{H}$. Ce qui est vérifié puisque nous avons :
  $\mathbf{H} = (\widetilde{\mathbf{A}_1} \sqcap \mathbf{G}_1) \sqcup (\widetilde{\mathbf{A}_2} \sqcap \mathbf{G}_2) \sqcup (\mathbf{G} \sqcap \mathbf{B}) \sqcup \mathbf{H}$
  (c) $\mathbf{G} \sqsubseteq \mathbf{A} \sqcup \mathbf{H}$. Ce qui est vérifié puisque nous avons :
  $\mathbf{A} \sqcup \mathbf{H} = \mathbf{A} \sqcup (\widetilde{\mathbf{A}_1} \sqcap \mathbf{G}_1) \sqcup (\widetilde{\mathbf{A}_2} \sqcap \mathbf{G}_2) \sqcup \mathbf{H}$
  et $\mathbf{G} \sqsubseteq \mathbf{A} \sqcup (\widetilde{\mathbf{A}_1} \sqcap \mathbf{G}_1) \sqcup (\widetilde{\mathbf{A}_2} \sqcap \mathbf{G}_2)$

□

Propriété 2.11 *Un contrat* $\mathbf{C}_1 = (\mathbf{A}_1,\mathbf{G}_1)$ *possède un complémentaire* $\widetilde{\mathbf{C}_1} = (\mathbf{A}_2,\mathbf{G}_2)$ *si et seulement si* $\mathbf{A}_1 = \widetilde{\mathbf{G}_1}$*. Ce complémentaire est alors* $\widetilde{\mathbf{C}_1} = (\mathbf{G}_1,\widetilde{\mathbf{G}_1})$*.*

Preuve :

Deux contrats $\mathbf{C}_1 = (\mathbf{A}_1,\mathbf{G}_1)$ et $\mathbf{C}_2 = (\mathbf{A}_2,\mathbf{G}_2)$ sont complémentaires si et seulement si leur borne inférieure est égale au contrat $(\mathbb{P}^\star,\{\mho\})$, et leur borne supérieure est égale au contrat $(\{\mho\},\mathbb{P}^\star)$. Donc $\mathbf{C}_1 = (\mathbf{A}_1,\mathbf{G}_1)$ et $\mathbf{C}_2 = (\mathbf{A}_2,\mathbf{G}_2)$ sont complémentaires si et seulement si ils satisfont les propriétés suivantes :

1 $\mathbf{A}_1 \sqcup \mathbf{A}_2 = \mathbb{P}^\star$
2 $((\mathbf{A}_1 \sqcap \widetilde{\mathbf{A}_2} \sqcap \mathbf{G}_1) \sqcup (\widetilde{\mathbf{A}_1} \sqcap \mathbf{A}_2 \sqcap \mathbf{G}_2) \sqcup (\mathbf{G}_1 \sqcap \mathbf{G}_2)) = \{\mho\}$
3 $\mathbf{A}_1 \sqcap \mathbf{A}_2 = \{\mho\}$
4 $(\widetilde{\mathbf{A}_1} \sqcap \mathbf{G}_1) \sqcup (\widetilde{\mathbf{A}_2} \sqcap \mathbf{G}_2) \sqcup (\mathbf{A}_1 \sqcap \mathbf{G}_2) \sqcup (\mathbf{A}_2 \sqcap \mathbf{G}_1) = \mathbb{P}^\star$

Ces propriétés sont satisfaites si et seulement si :

13 $\mathbf{A}_2 = \widetilde{\mathbf{A}_1}$ (équations 1 et 3)
2-1 $(\mathbf{A}_1 \sqcap \mathbf{G}_1) = \{\mho\}$
2-2 $(\widetilde{\mathbf{A}_1} \sqcap \mathbf{G}_2) = \{\mho\}$
2-3 $(\mathbf{G}_1 \sqcap \mathbf{G}_2) = \{\mho\}$
4 $(\widetilde{\mathbf{A}_1} \sqcap \mathbf{G}_1) \sqcup (\mathbf{A}_1 \sqcap \mathbf{G}_2) = \mathbb{P}^\star$

Ces propriétés sont satisfaites si et seulement si :

13 $\mathbf{A}_2 = \widetilde{\mathbf{A}_1}$ (équations 1 et 3)
2-1 $(\mathbf{A}_1 \sqcap \mathbf{G}_1) = \{\mho\}$
2-2 $(\widetilde{\mathbf{A}_1} \sqcap \mathbf{G}_2) = \{\mho\}$
2-3 $(\mathbf{G}_1 \sqcap \mathbf{G}_2) = \{\mho\}$
4 $(\mathbf{A}_1 \sqcup \widetilde{\mathbf{G}_1}) \sqcap (\widetilde{\mathbf{A}_1} \sqcup \widetilde{\mathbf{G}_2}) = \{\mho\}$

Ces propriétés sont satisfaites si et seulement si :

13 $\mathbf{A}_2 = \widetilde{\mathbf{A}_1}$ (équations 1 et 3)
2-1 $(\mathbf{A}_1 \sqcap \mathbf{G}_1) = \{\mho\}$
2-2 $(\widetilde{\mathbf{A}_1} \sqcap \mathbf{G}_2) = \{\mho\}$
2-3 $(\mathbf{G}_1 \sqcap \mathbf{G}_2) = \{\mho\}$
4-2 $\mathbf{A}_1 \sqcap \widetilde{\mathbf{G}_2} = \{\mho\}$
4-3 $\widetilde{\mathbf{A}_1} \sqcap \widetilde{\mathbf{G}_1} = \{\mho\}$
4-4 $\widetilde{\mathbf{G}_1} \sqcap \widetilde{\mathbf{G}_2} = \{\mho\}$

Ces propriétés sont satisfaites si et seulement si :

- $\mathbf{A}_2 = \widetilde{\mathbf{A}_1}$ (équations 1 et 3)
- $\mathbf{G}_1 = \widetilde{\mathbf{A}_1}$ (équations 2-1 et 4-3)
- $\mathbf{G}_2 = \mathbf{A}_1$ (équations 2-2 et 4-2)

□

Dans cet ordre partiel défini sur l'ensemble des contrats, il est impossible de caractériser le complémentaire de chaque contrat. Par conséquent, $(\mathbb{C},\preccurlyeq)$ n'est pas une algèbre booléenne.

Le treillis des contrats satisfaisant la relation *d'équivalence de filtrage* avec $(\mathbf{A},\mathbf{G})$ (acceptant les mêmes processus que $(\mathbf{A},\mathbf{G})$) forme un cube présenté sur la figure 2.14. Tous les contrats présentés sur le cube sont satisfaits par le même ensemble de processus. Ainsi un contrat peut être raffiné par un autre contrat acceptant

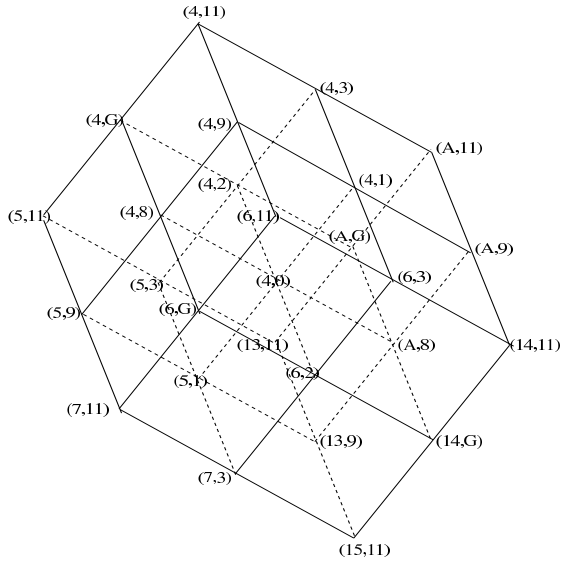


FIG. 2.14 – Treillis des contrats satisfaisant la relation d'équivalence de filtrage.

le même ensemble de processus mais dont l'expression des hypothèses et des garanties est différente. Les notations suivantes sont utilisées pour désigner les filtres :

0	$\{\mho\}$	5	$\widetilde{\mathbf{G}}$	10	$\mathbf{G}$
1	$\widetilde{\mathbf{A}} \sqcap \widetilde{\mathbf{G}}$	6	$(\mathbf{A} \sqcap \widetilde{\mathbf{G}}) \sqcup (\widetilde{\mathbf{A}} \sqcap \mathrm{G})$	11	$\widetilde{\mathbf{A}} \sqcup \mathbf{G}$
2	$\widetilde{\mathbf{A}} \sqcap \mathbf{G}$	7	$\widetilde{\mathbf{A}} \sqcup \widetilde{\mathbf{G}}$	12	$\mathbf{A}$
3	$\widetilde{\mathbf{A}}$	8	$\mathbf{A} \sqcap \mathbf{G}$	13	$\mathbf{A} \sqcup \widetilde{\mathbf{G}}$
4	$\mathbf{A} \sqcap \widetilde{\mathbf{G}}$	9	$(\mathbf{A} \sqcap \mathbf{G}) \sqcup (\widetilde{\mathbf{A}} \sqcap \widetilde{\mathbf{G}})$	14	$\mathbf{A} \sqcup \mathbf{G}$
				15	$\mathbb{P}^\star$

2.3.5 Une algèbre de contrats

Une algèbre de Heyting H est un treillis borné tel que pour tous éléments a et b dans H, il existe un plus grand élément x dans H, tel que la borne inférieure de a et x raffinent b.

Nous en déduisons que notre algèbre de contrats est une algèbre de Heyting [Bel99]. Cette structure possède la propriété intéressante d'être distributive.

Théorème 2.3 *($\mathbb{C}$, $\preccurlyeq$) est une algèbre de Heyting avec un supremum ($\{\mho\}$,$\mathbb{P}^\star$) et un infimum ($\mathbb{P}^\star$,$\{\mho\}$).*
Pour tous contrats $\mathbf{C}_1 = (\mathbf{A}_1,\mathbf{G}_1)$, $\mathbf{C}_2 = (\mathbf{A}_2,\mathbf{G}_2)$, il y a un plus grand élément $\mathbf{X} = (\mathbf{I},\mathbf{J})$ dans $\mathbb{C}$ tel que la borne inférieure de $\mathbf{C}_1$ et $\mathbf{X}$ raffine $\mathbf{C}_2$:

$\mathbf{I} = (\widetilde{\mathbf{A}_1} \sqcap \mathbf{A}_2) \sqcup (\mathbf{A}_1 \sqcap \mathbf{A}_2 \sqcap \mathbf{G}_1 \sqcap \widetilde{\mathbf{G}_2})$

$\mathbf{J} = \mathbf{G}_2 \sqcup (\mathbf{A}_1 \sqcap \widetilde{\mathbf{A}_2}) \sqcup (\mathbf{A}_1 \sqcap \widetilde{\mathbf{G}_1}) \sqcup (\widetilde{\mathbf{A}_2} \sqcap \widetilde{\mathbf{G}_1})$

Preuve :
Soit un contrat $\mathbf{D} = (\mathbf{B},\mathbf{H})$, tel que $(\mathbf{C}_1 \Downarrow \mathbf{D}) \preccurlyeq \mathbf{C}_2$.
$(\mathbf{C}_1 \Downarrow \mathbf{D}) \preccurlyeq \mathbf{C}_2 \iff$ (lemme 2.4)
$\mathbf{A}_2 \sqsubseteq (\mathbf{A}_1 \sqcap \mathbf{B})$
$(\mathbf{A}_2 \sqcap (\ (\mathbf{G}_1 \sqcap \mathbf{H}) \sqcup (\widetilde{\mathbf{A}_1} \sqcap\ \mathbf{B} \sqcap \mathbf{H}) \sqcup (\widetilde{\mathbf{B}} \sqcap \mathbf{A}_1 \sqcap \mathbf{G}_1)\ )) \sqsubseteq \mathbf{G}_2$
$(\mathbf{G}_1 \sqcap \mathbf{H}) \sqcup (\widetilde{\mathbf{A}_1} \sqcap \mathbf{B} \sqcap \mathbf{H}) \sqcup (\widetilde{\mathbf{B}} \sqcap \mathbf{A}_1 \sqcap \mathbf{G}_1) \sqsubseteq \mathbf{A}_1 \sqcup \mathbf{B} \sqcup \mathbf{G}_2$
$\iff$
(par les propriétés de l'algèbre de filtres)
$\mathbf{B} = ((\widetilde{\mathbf{A}_1} \sqcap \mathbf{A}_2) \sqcup \mathbf{B})$
$((\mathbf{G}_1 \sqcap \mathbf{A}_2 \sqcap \widetilde{\mathbf{G}_2} \sqcap \mathbf{H}) \sqcup (\widetilde{\mathbf{A}_1} \sqcap \mathbf{A}_2 \sqcap \widetilde{\mathbf{G}_2} \sqcap \mathbf{B} \sqcap \mathbf{H}) \sqcup (\widetilde{\mathbf{B}} \sqcap \mathbf{A}_2 \sqcap \widetilde{\mathbf{G}_2} \sqcap \mathbf{A}_1 \sqcap \mathbf{G}_1)) \sqsubseteq \{\mho\}$
$(\widetilde{\mathbf{G}_1} \sqcap \widetilde{\mathbf{B}} \sqcap \widetilde{\mathbf{G}_2} \sqcap \widetilde{\mathbf{G}_1} \sqcap \widetilde{\mathbf{H}}) \sqsubseteq \{\mho\}$
$\iff$
(par les propriétés de l'algèbre de filtres)
$\mathbf{B} = ((\widetilde{\mathbf{A}_1} \sqcap \mathbf{A}_2) \sqcup \mathbf{B})$
$\mathbf{B} = (((\mathbf{A}_2 \sqcap \widetilde{\mathbf{G}_2}) \sqcap (\mathbf{A}_1 \sqcap \mathbf{G}_1)) \sqcup \mathbf{B})$
$\mathbf{H} = ((\widetilde{\mathbf{G}_1} \sqcup \widetilde{\mathbf{A}_2} \sqcup \mathbf{G}_2) \sqcap \mathbf{H})$
$\mathbf{H} = ((\mathbf{A}_1 \sqcup \widetilde{\mathbf{A}_2} \sqcup \mathbf{G}_2 \sqcup \widetilde{\mathbf{B}}) \sqcap \mathbf{H})$
$\mathbf{H} = ((\mathbf{A}_1 \sqcup \mathbf{B} \sqcup \mathbf{G}_2 \sqcup \widetilde{\mathbf{G}_1}) \sqcap \mathbf{H})$
$\iff$
(par les propriétés de l'algèbre de filtres)
$\mathbf{B} = (\widetilde{\mathbf{A}_1} \sqcap \mathbf{A}_2) \sqcup (\mathbf{A}_2 \sqcap \widetilde{\mathbf{G}_2} \sqcap \mathbf{A}_1 \sqcap \mathbf{G}_1) \sqcup \mathbf{B}$
$\mathbf{H} = (\mathbf{G}_2 \sqcup (\widetilde{\mathbf{A}_1} \sqcap \mathbf{A}_2) \sqcup (\widetilde{\mathbf{A}_2} \sqcap \mathbf{B}) \sqcup (\widetilde{\mathbf{A}_2} \sqcap \widetilde{\mathbf{G}_1}) \sqcup (\mathbf{A}_1 \sqcap \widetilde{\mathbf{G}_1}) \sqcup (\widetilde{\mathbf{G}_1} \sqcap \widetilde{\mathbf{B}})\ ) \sqcap \mathbf{H}$
Les propriétés de l'algèbre booléenne sur les filtres nous permettent d'exprimer le complémentaire de $\mathbf{B}$.
$\widetilde{\mathbf{B}} = (\mathbf{A}_1 \sqcap \widetilde{\mathbf{A}_2}) \sqcup (\mathbf{A}_1 \sqcap \mathbf{G}_2) \sqcup (\mathbf{A}_1 \sqcap \widetilde{\mathbf{G}_1}) \sqcup \widetilde{\mathbf{A}_2} \sqcap \widetilde{\mathbf{B}}$
$\iff$
Nous réintroduisons les définitions de $\mathbf{B}$ et $\widetilde{\mathbf{B}}$ dans l'expression de $\mathbf{H}$.
$\mathbf{B} = (\widetilde{\mathbf{A}_1} \sqcap \mathbf{A}_2) \sqcup (\mathbf{A}_2 \sqcap \widetilde{\mathbf{G}_2} \sqcap \mathbf{A}_1 \sqcap \mathbf{G}_1) \sqcup \mathbf{B}$
$\mathbf{H} = (\ \mathbf{G}_2 \sqcup (\mathbf{A}_1 \sqcap \widetilde{\mathbf{A}_2}) \sqcup (\widetilde{\mathbf{A}_2} \sqcap \widetilde{\mathbf{G}_1}) \sqcup (\mathbf{A}_1 \sqcap \widetilde{\mathbf{G}_1})\ ) \sqcap \mathbf{H}$
Alors $\mathbf{X} = (\mathbf{I},\mathbf{J})$ est un contrat tel que : $(\mathbf{C}_1 \Downarrow \mathbf{X}) \preccurlyeq \mathbf{C}_2$.
Si $\mathbf{X}$ est le plus grand contrat tel que $(\mathbf{C}_1 \Downarrow \mathbf{X}) \preccurlyeq \mathbf{C}_2$, alors $\mathbf{D} \preccurlyeq$ X, tel que (lemme 2.4) :

- $((\widetilde{\mathbf{A}_1} \sqcap \mathbf{A}_2) \sqcup (\mathbf{A}_1 \sqcap \mathbf{A}_2 \sqcap \mathbf{G}_1 \sqcap \widetilde{\mathbf{G}_2})\ \sqsubseteq (\widetilde{\mathbf{A}_1} \sqcap \mathbf{A}_2) \sqcup (\mathbf{A}_2 \sqcap \widetilde{\mathbf{G}_2} \sqcap \mathbf{A}_1 \sqcap \mathbf{G}_1) \sqcup \mathbf{B})$
- $(((\widetilde{\mathbf{A}_1} \sqcap \mathbf{A}_2) \sqcup (\mathbf{A}_1 \sqcap \mathbf{A}_2 \sqcap \mathbf{G}_1 \sqcap \widetilde{\mathbf{G}_2})\ \sqcap (\ \mathbf{G}_2 \sqcup (\mathbf{A}_1 \sqcap \widetilde{\mathbf{A}_2}) \sqcup (\widetilde{\mathbf{A}_2} \sqcap \widetilde{\mathbf{G}_1}) \sqcup (\mathbf{A}_1 \sqcap \widetilde{\mathbf{G}_1})\ ) \sqcap \mathbf{H}) \sqsubseteq (\mathbf{G}_2 \sqcup (\mathbf{A}_1 \sqcap \widetilde{\mathbf{A}_2}) \sqcup (\mathbf{A}_1 \sqcap \widetilde{\mathbf{G}_1}) \sqcup (\widetilde{\mathbf{A}_2} \sqcap \widetilde{\mathbf{G}_1})\ )\ )$

– $((\mathbf{G}_2 \sqcup (\mathbf{A}_1 \sqcap \widetilde{\mathbf{A}_2}) \sqcup (\widetilde{\mathbf{A}_2} \sqcap \widetilde{\mathbf{G}_1}) \sqcup (\mathbf{A}_1 \sqcap \widetilde{\mathbf{G}_1})) \sqcap \mathbf{H} \sqsubseteq (((\widetilde{\mathbf{A}_1} \sqcap \mathbf{A}_2) \sqcup (\mathbf{A}_2 \sqcap \widetilde{\mathbf{G}_2} \sqcap \mathbf{A}_1 \sqcap \mathbf{G}_1) \sqcup \mathbf{B}) \sqcup (\mathbf{G}_2 \sqcup (\mathbf{A}_1 \sqcap \widetilde{\mathbf{A}_2}) \sqcup (\mathbf{A}_1 \sqcap \widetilde{\mathbf{G}_1}) \sqcup (\widetilde{\mathbf{A}_2} \sqcap \widetilde{\mathbf{G}_1}))))$

Ces propriétés sont clairement satisfaites. □

Dans une algèbre de Heyting H, le pseudo-complément $\neg a$ est le plus grand élément de H tel que la borne inférieure de a et $\neg a$ est égale à l'infimum.

Lemme 2.7 (Pseudo-complément) *Dans notre algèbre de contrats le pseudo-complément* $\neg\mathbf{C}$ *de* $\mathbf{C} = (\mathbf{A},\mathbf{G})$ *est égal à* $(\widetilde{\mathbf{A}} \sqcup \mathbf{G},\mathbf{A} \sqcap \widetilde{\mathbf{G}})$.

Preuve : Soit deux contrats $\mathbf{C} = (\mathbf{A},\mathbf{G})$ et $\mathbf{D} = (\mathbf{B},\mathbf{H})$, tels que $(\mathbf{C} \Downarrow \mathbf{D}) = (\mathbb{P}^\star,\{\mho\})$.

$$
\begin{array}{l}
(\mathbf{C} \Downarrow \mathbf{D}) = (\mathbb{P}^\star,\{\mho\}) \\
\Longleftrightarrow \\
\mathbf{A} \sqcup \mathbf{B} = \mathbb{P}^\star \\
((\mathbf{A} \sqcap \widetilde{\mathbf{B}} \sqcap \mathbf{G}) \sqcup (\widetilde{\mathbf{A}} \sqcap \mathbf{B} \sqcap \mathbf{H}) \sqcup (\mathbf{G} \sqcap \mathbf{H})) = \{\mho\} \\
\Longleftrightarrow \\
\mathbf{B} = \widetilde{\mathbf{A}} \sqcup \mathbf{B} \\
\mathbf{B} = (\mathbf{A} \sqcap \mathbf{G}) \sqcup \mathbf{B} \\
\mathbf{H} = (\mathbf{A} \sqcap \widetilde{\mathbf{G}} \sqcap \mathbf{H})
\end{array}
$$

Ce qui équivaut à dire que $\mathbf{D}$ est un contrat de la forme :

$\mathbf{B} = \widetilde{\mathbf{A}} \sqcup \mathbf{G} \sqcup \mathbf{B}$

$\mathbf{H} = (\mathbf{A} \sqcap \widetilde{\mathbf{G}} \sqcap \mathbf{H})$

Montrons que $(\widetilde{\mathbf{A}} \sqcup \mathbf{G},\mathbf{A} \sqcap \widetilde{\mathbf{G}})$ est le plus grand contrat tel que

$$(\mathbf{C} \Downarrow (\widetilde{\mathbf{A}} \sqcup \mathbf{G},\mathbf{A} \sqcap \widetilde{\mathbf{G}})) = (\mathbb{P}^\star, \{\mho\})$$

Ce qui équivaut à montrer que $\mathbf{D} \preccurlyeq (\widetilde{\mathbf{A}} \sqcup \mathbf{G},\mathbf{A} \sqcap \widetilde{\mathbf{G}})$, c'est-à-dire (lemme 2.4) :

– $(\widetilde{\mathbf{A}} \sqcup \mathbf{G}) \sqsubseteq (\widetilde{\mathbf{A}} \sqcup \mathbf{G} \sqcup \mathbf{B})$

– $(\widetilde{\mathbf{A}} \sqcup \mathbf{G}) \sqcap (\mathbf{A} \sqcap \widetilde{\mathbf{G}} \sqcap \mathbf{H}) \sqsubseteq \mathbf{A} \sqcap \widetilde{\mathbf{G}}$

– $(\mathbf{A} \sqcap \widetilde{\mathbf{G}} \sqcap \mathbf{H}) \sqsubseteq ((\widetilde{\mathbf{A}} \sqcup \mathbf{G} \sqcup \mathbf{B}) \sqcup (\mathbf{A} \sqcap \widetilde{\mathbf{G}}))$

Ces propriétés sont clairement satisfaites. □

2.3.6 Masquage de variables dans un contrat

Définition 2.17 (Elimination de variables dans un contrat) *Soient x une variable, et* $\mathbf{C} = (\mathbf{A},\mathbf{G})$ *un contrat, l'élimination de x dans* $\mathbf{C}$ *est le contrat* $\mathbf{C}_{\backslash x}$ *défini par :* $\mathbf{C}_{\backslash x} =_\Delta (\mathbf{A}_{|\forall x},\mathbf{G}_{|\exists x})$

Propriété 2.12 *Un contrat* $\mathbf{C}$ *raffine le contrat* $\mathbf{C}$ *dans lequel des variables ont été éliminées :* $\mathbf{C} \preccurlyeq \mathbf{C}_{\setminus x}$

Preuve :
Soit un contrat $\mathbf{C} = (\mathbf{A},\mathbf{G})$. A partir du lemme 2.4 et de la définition 2.17, $\mathbf{C} \preccurlyeq \mathbf{C}_{\setminus x}$ si et seulement si les propriétés suivantes sont satisfaites :

(a) $\mathbf{A}_{|\forall x} \sqsubseteq \mathbf{A}$

(b) $(\mathbf{A}_{|\forall x} \sqcap \mathbf{G}) \sqsubseteq \mathbf{G}_{|\exists x}$

(c) $\mathbf{G} \sqsubseteq \mathbf{A} \sqcup \mathbf{G}_{|\exists x}$

La satisfaction de ces propriétés est une conséquence triviale de la propriété 2.9 et du théorème 2.2 □

2.4 Conclusion

En considérant le choix d'une abstraction des comportements par des fonctions d'un ensemble de noms vers des domaines de valeurs (booléens, entiers, séries, représentation discrète du temps par un ensemble de marques, fonctions continues), nous avons introduit la notion de filtre afin de caractériser formellement les composants logiques filtrant un processus à travers les hypothèses et les garanties d'un contrat. Dans notre modèle, un processus p remplit ses exigences (ou satisfait) $(\mathbf{A},\mathbf{G})$, s'il est rejeté par $\mathbf{A}$ (il est alors hors de portée du contrat $(\mathbf{A},\mathbf{G})$), ou bien s'il est accepté par $\mathbf{G}$.

Nos principaux résultats sont que :

– la structure de filtres définit une algèbre booléenne,

– et également que la structure de contrats définit une algèbre de Heyting.

Cette structure mathématiquement riche permet de raisonner sur les contrats avec une grande souplesse au niveau du raffinement, de l'abstraction, de la composition. En outre, la négation d'un contrat peut être formellement exprimée dans le modèle. De plus, les contrats ne sont pas limités à l'expression de propriétés de sûreté, comme dans la plupart des modèles d'expression de propriétés, mais ils englobent l'expression de propriétés de vivacité. Tout cela est dû à la notion centrale de filtres.

Dans le but d'évaluer la généricité et l'évolutivité de notre approche, nous avons mis au point un langage de modules basé sur le paradigme des contrats et nous l'avons utilisé pour la spécification de composants basés sur la notion de processus. Le paradigme que nous proposons est de considérer le contrat comme le type de comportement d'un module ou d'un composant. Il sera ensuite utilisé pour l'élaboration de l'architecture fonctionnelle d'un système en considérant une obligation de preuve validant la satisfaction des hypothèses par l'environnement et des garanties par le composant lors de la construction de cette architecture.

Deuxième partie

Typage par contrat et encapsulation de processus

Chapitre 3

Le langage Signal

3.1 Une introduction à SIGNAL

Dans cette partie, nous présentons la syntaxe abstraite du langage de programmation synchrone SIGNAL [LTL03], ainsi que la syntaxe nécessaire à la lecture et la compréhension des exemples de processus spécificiés avec ce langage.

SIGNAL est un langage déclaratif utilisé pour la description d'applications temps réel. Un programme SIGNAL est un processus qui satisfait un système d'équations portant sur des variables de flots de données. Dans le langage SIGNAL, une variable de flots de données est appelée un *signal*. Un signal est une suite non bornée d'instants logiques associés à une valeur typée. *L'horloge* d'un signal désigne les instants durant lesquels le signal est associé à une valeur. Lorsqu'un signal a une valeur, on dit que ce signal est *présent*. Un processus SIGNAL est caractérisé par l'ensemble de tous les comportements satisfaisant le système d'équations associé au processus.

3.1.1 Syntaxe abstraite

Dans le langage de programmation synchrone SIGNAL, un processus P est la composition d'équations portant sur des signaux. Un signal x est un flot de valeurs associées à une représentation discrète du temps déterminée par l'horloge du signal. Ainsi un signal n'est présent qu'à certains instants durant lesquels une valeur lui est affectée. Nous introduisons la syntaxe abstraite du langage ainsi que les notions essentielles pour comprendre les exemples écrits en SIGNAL.

En SIGNAL, la syntaxe d'un processus P est définie par la grammaire :

$$\begin{array}{rcll} P & ::= & x = y\ f\ z & \text{(équation)} \\ & | & P \,|\, Q & \text{(composition)} \\ & | & P/x & \text{(restriction)} \end{array}$$

SIGNAL utilise trois processus primitifs : pre (désigne la précédente valeur d'un signal à un instant donné), when (l'échantillonnage de signal) et default (la valeur par défaut d'un signal). Il utilise également trois fonctions booléennes : la négation not, l'égalité eq, et l'identité id.

Equation fonctionnelle

Une équation $(x_1, x_2, ..., x_m) = f\,(y_1, y_2, ..., y_n)$ décrit une relation logique (or, and, not ...), relationnelle (=, <,...), ou arithmétique (+, -,...), entre un ensemble d'opérandes $(y_1, y_2, ..., y_n)$ et un résultat désigné par un ensemble de valeurs noté $(x_1, x_2, ..., x_m)$, mis en relation par un processus f. Nous notons $v \in \mathcal{D}$ pour désigner une valeur associée aux variables, x, y, $z \in \mathcal{V}$. tt et ff représenteront les valeurs booléennes *vrai* et *faux*.

Retard

L'équation x = pre v y initialise la variable x avec la valeur v et affecte la valeur précédente de y à x. L'opérateur pre impose que x et y soient deux variables synchrones (x et y sont présentes en même temps).

$y \mapsto$ (t_1, true) (t_2, false) (t_3, false) (t_4, true) (t_5, true) (t_6, false)...
$x \mapsto$ (t_1, false) (t_2, true) (t_3, false) (t_4, false) (t_5, true) (t_6, true)...

FIG. 3.1 – Un comportement qui satisfait l'équation x = pre false y.

La figure 3.1 présente un comportement satisfaisant l'équation x = pre v y où $\{t_1, ..., t_6\}$ représentent un ensemble d'instants discrets.

Echantillonnage

L'équation $x = y$ when z affecte la valeur de la variable y à x, lorsque que le signal booléen z est présent et vrai. Cet opérateur impose que les instants où x est présent soient inclus dans l'horloge de z et l'horloge de y. Le signal z est du type booléen. when impose que les signaux x et y soient tous les deux du même type.

$y \mapsto$ (t_1, true)		(t_3, false)	(t_4, false)		(t_5, true)...
$z \mapsto$	(t_2, false)	(t_3, true)	(t_4, false)	(t_5, true)	(t_5, true)...
$x \mapsto$		(t_3, false)			(t_5, true)...

FIG. 3.2 – Un comportement qui satisfait l'équation $x = y$ when z.

Choix

L'équation $x \ = y$ default z associe la valeur de y à x lorsque y est présent, sinon x prend la valeur de z. Le signal x est présent lorsque les signaux y ou z sont présents. L'horloge de x est l'union de l'horloge de y et z. L'opérateur default impose que les signaux x, y, et z soient tous du même type.

$y \mapsto$	$(t_1,$ true$)$		$(t_3,$ false$)$		$(t_5,$ false$)$	$(t_6,$ true$)$...
$z \mapsto$		$(t_2,$ false$)$	$(t_3,$ true$)$	$(t_4,$ true$)$		$(t_6,$ false$)$...
$x \mapsto$	$(t_1,$ true$)$	$(t_2,$ false$)$	$(t_3,$ false$)$	$(t_4,$ true$)$	$(t_5,$ false$)$	$(t_6,$ true$)$...

FIG. 3.3 – Un comportement qui satisfait l'équation $x \ = y$ default z.

Composition

La composition | est un opérateur qui désigne la composition synchrone telle qu'elle est définie dans le modèle de SIGNAL. La composition synchrone $P|Q$ de deux processus P et Q est un processus équivalent à l'intersection des ensembles de comportements associés aux processus P et Q. Dans la sémantique de SIGNAL, $P|Q$ est le plus grand ensemble de comportements satisfaisant en même temps le système d'équations associé à P ainsi que le système d'équations associé à Q.

Masquage

Le masquage / est un opérateur utilisé pour réduire la portée d'un signal à l'intérieur d'un processus. Ainsi P/x réduit la portée du signal x à l'intérieur du processus P, empêchant toute lecture ou modification du signal x depuis l'extérieur. De cette façon, le signal x est défini de manière locale au processus P. Par conséquent, il est possible de définir des signaux locaux de même nom dans des processus différents.

3.1.2 Syntaxe concrète

Nous donnons un aperçu de la syntaxe concrète de SIGNAL afin de suggérer quelle est la forme des programmes, ainsi que la structure des processus écrits en SIGNAL.

Si `P` est la syntaxe concrète d'un processus SIGNAL, nous écrirons P pour désigner sa syntaxe abstraite. L'opérateur d'affectation := est utilisé pour associer une contrainte à un signal.

- `(x1, ..., xn) := f(y1, ..., yn)` est la syntaxe concrète associée à l'équation $x = f(y)$,

- les expressions `not c`, `x when c`, et `x default y` ont la même signification que dans la syntaxe abstraite,
- dans la syntaxe abstraite, l'opérateur de retard est noté $x = \mathsf{pre}\ v\ y$, tandis que dans la syntaxe concrète, il est noté `x := y$1 init v`. L'opérateur de retard peut être utilisé pour accéder à des données antérieures de n instants en utilisant une expression de la forme `x := y$n init v`,
- la déclaration `P where t x` définit le type `t` du SIGNAL `x` de manière locale au processus `P`, ce qui est équivalent dans la syntaxe abstraite à P/x,
- l'équation `x := v` (pour une constante `v`) est équivalente à `x := x$1 init v` et `x := when c` est équivalent à `x := true when c`.

3.1.3 Exemple de processus SIGNAL

Nous utilisons ici les primitives `where`, `default`, et `$` pour définir un processus rudimentaire augmentant la valeur de la variable `counter`, celle-ci pouvant être remise à 0 par le signal `reset`. Le processus `Count` est un programme très simple qui va permettre d'illustrer l'écriture de processus dans le langage SIGNAL.

L'entête du programme est constituée du nom du processus ainsi que de l'interface spécifiant les variables d'entrée marquées par un "`?`", et les variables de sortie marquées par un "`!`". Le processus `Count` possède une variable d'entrée `reset` et une variable de sortie `val`.

Le corps du processus est caractérisé par un ensemble d'équations séparées par "`|`". Rappelons que ces équations doivent toutes être satisfaites simultanément. A chaque occurrence du signal `reset`, le signal `val` est remis à 0 (`0 when reset`). Sinon, le signal `val` incrémente la variable `counter` (`counter + 1`).

La variable locale `counter` est initialisée à 0 et enregistre les précédentes valeurs du signal `val` (`val := (0 when reset) default (counter + 1)`). Dans la dernière partie du programme, la primitive `where` permet de déclarer les variables locales et d'expliciter leur type.

```
 process Count = (? event reset; ! integer val ; )
(| counter := val$1 init 0
 | val     := (0 when reset) default (counter + 1)
 |) where integer counter;
end;
```

Le processus `Count` est dirigé par l'horloge du signal de sortie `val` laquelle est différente de l'horloge de `reset`. Lorsque `val` est sollicité par l'environnement et que `reset` est absent, alors `Count` incrémente la valeur affectée à `val`, et si `reset` est présent alors la valeur 0 est affectée à `val`.

La figure 3.4 présente un comportement admis par le processus `Count`.

event	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}
reset		true					true				true	true		
val	1	0	1	2	3	4	0	1	2	3	0	0	1	2
counter	0	1	0	1	2	3	4	0	1	2	3	0	0	1

FIG. 3.4 – Trace d'exécution.

3.2 Un modèle pour les systèmes polychrones

Dans le langage SIGNAL, le temps est modélisé par un système de marques, permettant une représentation discrète. Un instant discret est associé à une marque. Cette structure est particulièrement adaptée à la représentation des systèmes polychrones. Ces systèmes sont caractérisés par le fait qu'il n'est pas nécessairement possible d'exprimer toutes les horloges des signaux en fonction d'une unique horloge principale. Nous présentons ce modèle ainsi que la sémantique des processus SIGNAL et les propriétés algébriques engendrées par cette représentation.

3.2.1 Un modèle de marques

Une marque t est un élément d'un ensemble dense $\mathbb{T}$ équipé d'une relation d'ordre partiel $\leq$.

Définition 3.1 (Ordre partiel sur les marques) *L'ordre partiel $(\mathcal{T}, \leq)$ modélisant le temps dans un processus* SIGNAL *est un sous-ensemble $\mathcal{T} \subset \mathbb{T}$ satisfaisant les propriétés suivantes :*

- *$\mathcal{T}$ est dénombrable,*
- *$\mathcal{T}$ est minoré par l'instant 0 pour la relation $\leq$,*
- *$\mathcal{T}$ est bien fondé : il n'existe pas de suite infinie (t_n) tel que $\forall n \in \mathbb{N}$, $t_{n+1} \leq t_n$.*

Définition 3.2 (Evénement) *Un événement $e \in \mathbb{E} = \mathcal{T} \times \mathbb{V}$ est une relation entre une marque et une valeur.*

Une chaîne $C \subseteq \mathcal{T}$ est un sous-ensemble de $\mathcal{T}$ totalement ordonné. Soit $\mathcal{C}$ l'ensemble des chaînes C. Pour toute marque $t \in C$, nous écrirons $min(C)$, et $pred_C(t)$ pour respectivement désigner le minimum et le prédécesseur immédiat de t dans C.

Définition 3.3 (Signal) *Un signal $s \in \mathcal{S} = \mathcal{T} \rightarrow \mathbb{V}$ est une fonction définie sur une chaîne de marques vers un ensemble de valeurs. Nous noterons $tags(s)$ pour désigner le domaine de définition de s.*

Définition 3.4 (Comportements) *Un comportement* $b \in \mathcal{B} = \mathcal{X} \rightarrow \mathcal{S}$ *est une fonction associant un nom* $x \in \mathcal{X}$ *à un signal* $s \in \mathcal{S}$. *Nous notons* $vars(b)$ *pour désigner l'ensemble des variables sur lequel le comportement* b *est défini.* $tags(b) = \cup_{x \in vars(b)} tags(b(x))$ *désigne l'ensemble des marques associé à* b, *c'est-à-dire l'ensemble des instants où l'une des variables de* b *est présente. L'expression informelle "la variable* x *est présente à l'instant* t *dans le comportement* b*" est formellement caractérisée par* $t \in tags(b(x))$.

La projection d'un comportement b sur un ensemble de noms de signaux $X \in \mathcal{X}$ est notée $b_{|\mathrm{X}}$. La projection vérifie : $vars(b_{|\mathrm{X}}) = X$ et $\forall x \in X, b_{|\mathrm{X}}(x) = b(x)$.

Définition 3.5 (Processus) *Dans le langage* SIGNAL, *un processus est un ensemble de comportements définis sur un même ensemble de signaux (noté* $vars(p)$*).*

La *composition synchrone* de deux processus p et q, est l'ensemble des comportements b vérifiant $vars(b) = vars(p) \cup vars(q)$, tel que $b_{|\mathrm{vars(p)}} \in p$ et $b_{|\mathrm{vars(q)}} \in q$. Ainsi, $p \,|\, q = \{\ b \cup c \mid (b, c) \in p \times q \wedge b_{|vars(p) \cap vars(q)} = c_{|vars(p) \cap vars(q)}\ \}$.

$$p \ni b = \left(\begin{array}{cccccccc} & t_1 & t_2 & t_3 & t_4 & t_5 & t_6 & \ldots \\ r \mapsto & \mathsf{true} & \mathsf{true} & & \mathsf{true} & \mathsf{false} & \mathsf{true} & \ldots \\ x \mapsto & & \mathsf{false} & \mathsf{true} & \mathsf{true} & & \mathsf{true} & \ldots \\ y \mapsto & \mathsf{true} & & \mathsf{false} & & \mathsf{false} & \mathsf{true} & \ldots \end{array}\right)$$

$$q \ni c = \left(\begin{array}{cccccccc} & t_1 & t_2 & t_3 & t_4 & t_5 & t_6 & \ldots \\ x \mapsto & & \mathsf{false} & \mathsf{true} & \mathsf{true} & & \mathsf{true} & \ldots \\ y \mapsto & \mathsf{true} & & \mathsf{false} & & \mathsf{false} & \mathsf{true} & \ldots \\ s \mapsto & \mathsf{false} & \mathsf{true} & \mathsf{false} & & & \mathsf{true} & \ldots \end{array}\right) \Bigg\} \; vars(p) \cap vars(q)$$

FIG. 3.5 – La composition synchrone $p \,|\, q$.

La composition synchrone des processus p et q est un ensemble de comportements définis sur $var(p) \cup var(q)$ tel que la restriction de ces comportements à $var(p) \cap var(q)$ appartient à $p_{|vars(p) \cap vars(q)}$ et $q_{|vars(p) \cap vars(q)}$. Sur la figure 3.5, la projection du comportement b sur $\{x,\ y\}$ est égale à la projection de comportement c sur $\{x,\ y\}$.

La composition synchrone de deux processus SIGNAL est une opération commutative et associative.

3.2.2 Sémantique dénotationnelle de SIGNAL

La dénotation $[\![\mathrm{P}]\!]$ d'un processus P représente le plus grand ensemble de comportements acceptés par le système d'équations associé à P. $[\![\mathrm{P}]\!]$ est défini

par induction sur ces équations. Pour chaque équation, $x = f\ y$, la fonction $[\![\ ]\!]$ caractérise la relation entre les signaux impliqués dans l'équation considérée et les chaînes de marques associées à ces signaux.

Fonction de dénotation $[\![\ ]\!]$ L'équation $x = \mathsf{pre}\ v\ y$ initialise la variable x avec la valeur v, et aux autres instants où x est présent la précédente valeur de y lui est attribuée. Elle définit les signaux x et y sur une même chaîne de marques $C \in \mathcal{C}$ et définit la valeur de x par la valeur du signal y à l'instant prédécesseur immédiat de t dans C. Cette équation impose que x et y soient deux signaux synchrones (x et y sont présents durant les mêmes marques C).

$$[\![x = \mathsf{pre}\ v\ y\ ]\!] = \left\{ b \in \mathcal{B}_{|x,y} \;\middle|\; \begin{array}{l} tags(b(x)) = tags(b(y)) = C \in \mathcal{C}, \\ b(x)(min(C)) = v, \\ \forall t \in C \backslash min(C),\ b(x)(t) = b(y)(t)(pred_C(t)) \end{array} \right\}$$

L'équation $x = y\ \mathsf{when}\ z$ définit x par y lorsque z est vrai. Soit une marque t, considérons les quatre cas suivants :

- Si $t \notin tags(y)$, alors $t \notin tags(x)$,
- Si $t \notin tags(z)$, alors $t \notin tags(x)$,
- Si $t \in tags(y)$, $t \in tags(z)$, et $z(t) = ff$, alors $t \notin tags(x)$,
- Si $t \in tags(y)$, $t \in tags(z)$, et $z(t) = tt$, alors $t \in tags(x)$ et $x(t) = y(t)$.

$$[\![x = y\ \mathsf{when}\ z\ ]\!] = \left\{ b \in \mathcal{B}_{|x,y,z} \;\middle|\; \begin{array}{l} tags(b(x)) = \left\{ \begin{array}{l} t \in tags(b(y)) \cap tags(b(z)), \\ \text{tel que } b(z)(t) = tt \end{array} \right\} \\ \forall t \in tags(b(x)),\ b(x)(t) = b(y)(t) \end{array} \right\}$$

L'équation $x = y\ \mathsf{default}\ z$ associe la valeur de y à x lorsque y est présent, sinon la valeur de z est associée à x. A un instant donné, il y a trois cas à considérer :

- Si $t \in tags(y)$ alors $t \in tags(x)$ et $x(t) = y(t)$,
- Si $t \notin tags(y)$ et $t \in tags(z)$, alors $t \in tags(x)$, et $x(t) = z(t)$
- Si $t \notin tags(y)$ et $t \notin tags(z)$ alors $t \notin tags(x)$,

$$[\![x = y\ \mathsf{default}\ z\ ]\!] = \left\{ b \in \mathcal{B}_{|x,y,z} \;\middle|\; \begin{array}{l} tags(b(y)) \cup tags(b(z)) = tags(b(x)) = C \in \mathcal{C} \\ \forall t \in C, b(x)(t) = \left\{ \begin{array}{l} b(y)(t),\ t \in tags(b(y)) \\ b(z)(t),\ t \notin tags(b(y)) \end{array} \right. \end{array} \right\}$$

$$[\![x \;=\; \mathsf{pre}\ true\ y\ ]\!] \ni \begin{pmatrix} & t_1 & t_2 & t_3 & t_4 & t_5 & \dots \\ y \mapsto & \mathsf{false} & \mathsf{true} & \mathsf{true} & \mathsf{false} & \mathsf{true} & \dots \\ x \mapsto & \mathsf{true} & \mathsf{false} & \mathsf{true} & \mathsf{true} & \mathsf{false} & \dots \end{pmatrix}$$

$$[\![x \;=\; y\ \mathsf{when}\ z]\!] \ni \begin{pmatrix} & t_1 & t_2 & t_3 & t_4 & t_5 & \dots \\ y \mapsto & \mathsf{true} & \mathsf{true} & & & \mathsf{true} & \dots \\ z \mapsto & \mathsf{false} & & \mathsf{false} & \mathsf{true} & \mathsf{true} & \dots \\ x \mapsto & & & & & \mathsf{true} & \dots \end{pmatrix}$$

$$[\![x \;=\; y\ \mathsf{default}\ z]\!] \ni \begin{pmatrix} & t_1 & t_2 & t_3 & t_4 & t_5 & \dots \\ y \mapsto & \mathsf{false} & & \mathsf{false} & \mathsf{true} & & \dots \\ z \mapsto & \mathsf{true} & \mathsf{false} & & \mathsf{false} & \mathsf{true} & \dots \\ x \mapsto & \mathsf{false} & \mathsf{false} & \mathsf{false} & \mathsf{true} & \mathsf{true} & \dots \end{pmatrix}$$

FIG. 3.6 – Valeur dénotationnelle des opérateurs pre , when , default

La figure 3.6 illustre la valeur dénotationnelle :
- d'un comportement défini sur les variables x, et y, satisfaisant l'équation de retard $x \;=\; \mathsf{pre}\ v\ y$.
- d'un comportement défini sur les variables x, y et z, satisfaisant l'équation d'échantillonnage $x \;=\; y\ \mathsf{when}\ z$.
- d'un comportement défini sur les variables x, y et z, satisfaisant l'équation de choix $x \;=\; y\ \mathsf{default}\ z$.

La valeur dans la sémantique dénotationnelle de la composition synchrone $P \mid Q$ est la composition synchrone $[\![P]\!] \mid [\![Q]\!]$ des valeurs dénotationnelles $[\![P]\!]$ et $[\![Q]\!]$.

$$[\![P \mid Q]\!] = [\![P]\!] \mid [\![Q]\!]$$

3.2.3 Propriétés algébriques

La définition des processus SIGNAL et de la composition synchrone permet de définir une structure algébrique riche sur l'ensemble des processus écrits en SIGNAL. En effet, un processus SIGNAL est un ensemble de comportements définis sur un même ensemble de signaux, et la composition synchrone de deux processus P et Q est un ensemble de comportements obtenus par l'union de l'ensemble des comportements de P avec l'ensemble des comportements de Q. La composition synchrone de deux processus P et Q satisfait $vars(P \mid Q) = vars(P) \cup vars(Q)$.

Ainsi, un processus P est équivalent à la projection de $P \mid Q$ sur $vars(P)$, si et seulement si la projection de Q sur les signaux $vars(P)$ est contenue dans la projection de P sur $vars(Q)$.

Propriété 3.1 *Pour tous processus P et Q,*
- $(\llbracket P \rrbracket \,|\, \llbracket Q \rrbracket)_{|vars(P)} \subseteq \llbracket P \rrbracket$
- $(\llbracket P \rrbracket \,|\, \llbracket Q \rrbracket)_{|vars(Q)} \subseteq \llbracket Q \rrbracket$
- $\llbracket P \rrbracket = (\llbracket P \rrbracket \,|\, \llbracket Q \rrbracket)$*, si et seulement si* $\llbracket P \rrbracket \subseteq \llbracket Q \rrbracket$

Corollaire 3.1 *Pour tous processus* SIGNAL *P, Q, R, nous avons :*
- $\llbracket P \rrbracket \,|\, \llbracket P \rrbracket = \llbracket P \rrbracket$
- $\llbracket P \rrbracket \subseteq \llbracket Q \rrbracket \Rightarrow (\llbracket P \rrbracket \,|\, \llbracket R \rrbracket) \subseteq (\llbracket Q \rrbracket \,|\, \llbracket R \rrbracket)$

3.3 Conclusion

Un programme SIGNAL permet de définir un processus. Ce processus est composé d'une interface modélisant les variables d'entrée/sortie, et d'un ensemble d'équations devant être résolues simultanément. Chaque équation associée au corps d'un processus définit un ensemble de comportements autorisés. La sémantique dénotationnelle d'un processus est caractérisée par la composition synchrone de ces ensembles de comportements. En effet, SIGNAL est un langage dans lequel la sémantique des processus repose sur la théorie des comportements.

Un processus SIGNAL est un ensemble de comportements définis sur un même ensemble de variables tout comme dans le modèle de contrats présenté dans le chapitre 2. De plus, dans le langage SIGNAL la définition des comportements est équivalente à celle proposée dans la section 2.1 du chapitre 2, en fixant le domaine de définition des variables $\mathcal{D}$ égale à $\mathcal{S}$. En outre, nous pouvons constater que la définition de la projection d'un comportement en SIGNAL est équivalente à la restriction d'un comportement dans le modèle de contrats. Ainsi, les opérations de restriction, d'extension, et la théorie sur les filtres peut être aisément appliquées aux processus SIGNAL. Par exemple, le processus générateur d'un filtre peut être un processus spécifié avec SIGNAL.

Chapitre 4

Un langage de modules basé sur le typage par contrats

Nous adoptons le paradigme de *contrat* pour définir une méthodologie de validation de composants basée sur la notion de processus dans le cadre de la modélisation de logiciels embarqués. Elle se compose d'un cadre algébrique, basé sur deux concepts très simples, permettant de raisonner sur les contrats présentés dans le chapitre 2. Tout d'abord, les hypothèses et les garanties d'un composant sont définies comme des dispositifs de filtres : les hypothèses filtrent les comportements du contexte d'exécution acceptés par le composant et les garanties filtrent les comportements fournis par le composant. L'ensemble des filtres constitue une algèbre booléenne et l'ensemble des contrats est caractérisé par une algèbre de Heyting. Il en résulte un cadre de raisonnement permettant d'abstraire, de raffiner, de combiner et de normaliser les contrats.

Dans cette partie, nous présentons une version étendue des travaux publiés dans [GTLG09]. Nous allons utiliser l'algèbre de contrats dans l'objectif de concevoir un système de modules dont le système de types est fondé sur la notion de contrat. Le type d'un module est un contrat modélisant les hypothèses faites sur l'environnement d'exécution et les garanties offertes par ses comportements. Il permet d'associer un module à une interface qui peut être utilisée dans des variétés de scénarios tels que le contrôle de la composabilité de modules ou le support efficace de la compilation modulaire.

Plan La contribution principale de ce chapitre est d'utiliser l'algèbre de contrats (chapitre [2]) pour la définition d'un système de modules fortement typés : les contrats sont utilisés pour typer les composants satisfaisant les propriétés comportementales attendues. Ainsi, nous définissons un *langage de modules* mettant en œuvre notre algèbre de contrats afin de l'appliquer à la validation des systèmes à base de composants. La section 4.1.4 présente le principe de base du langage.

Les concepts de spécification et d'implémentation sont présentés ainsi qu'un mécanisme générique d'encapsulation permettant de représenter une classe de spécifications ou d'implémentations partageant un modèle commun de comportement. La relation de sous-typage basée sur le raffinement de contrats est présentée dans la section 4.2. Notre objectif est de vérifier l'exactitude de la construction d'un programme. Le système d'inférence de types permet d'attribuer un contrat aux noms de processus dans le langage de modules et de générer une obligation de preuve sous la forme d'une fonction observatrice dans le langage de programmation cible. Le théorème de correction présenté dans la section 4.3 formalise l'exactitude d'un programme écrit dans un langage de modules en vérifiant que l'ensemble des processus décrits satisfont les contrats qui leur sont associés.

4.1 Les points clés du langage

Dans cette section, nous présentons un langage dont le principe de base est la séparation de l'interface, qui déclare les propriétés d'un programme à l'aide de contrats, de l'implémentation, qui définit une spécification exécutable. Nous définissons ainsi une syntaxe et un système de typage mettant en œuvre notre algèbre de contrats dans le but de l'appliquer à la validation des systèmes à base de composants.

4.1.1 Un exemple

Nous illustrons notre approche en considérant une application automobile simple présentée dans [AMPF07b]. Nous définissons des contrats caractérisant les propriétés spécifiques d'un moteur à quatre temps. Dans cette spécification illustrée par la figure 4.1, le comportement cyclique du moteur est caractérisé par quatre phases successives : *Intake*, *Combustion*, *Compression*, et *Exhaust*. Ces phases sont associées à une position de l'arbre à cames dont la position angulaire est mesurée en degrés. Ce procédé de mesure modélise un système de temps discret et symbolique. L'information concernant la position de l'arbre à cames est transmise à chaque tic de l'horloge de la variable *cam*, correspondant à un échantillonnage en fonction du temps de l'information émise par des capteurs.

A chaque émission de cette information une réaction est déclenchée. Par exemple, à 90° la valve d'admission est fermée et une transition vers la phase de compression est déclenchée.

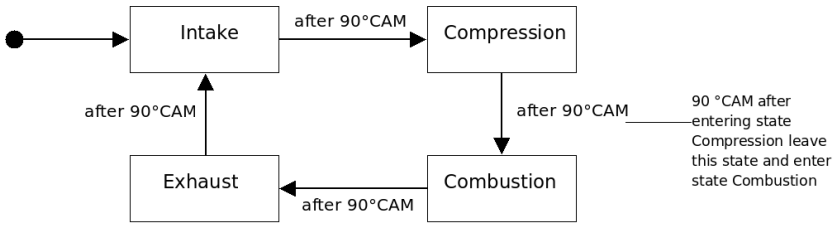


FIG. 4.1 – Cycle d'un moteur à 4 temps.

Dans le langage de modules, une spécification est désignée par le mot clé **contract**. Elle définit un ensemble de variables d'entrée/sortie soumis à un contrat. L'interface définit la manière dont le composant interagit avec son environnement d'exécution à travers ses variables d'entrée/sortie. En outre, une spécification intègre des propriétés qui sont modélisées par une composition de contrats. Par exemple, la spécification du mode d'admission du contrôleur du moteur peut être définie par l'hypothèse `0 <= (cam mod 360) <90` et la garantie `intake` (voir figure 4.2). Une mise en œuvre de l'interface désignée par le mot-clé **process** contient une implémentation compatible avec le contrat associant la valeur *true* au signal `intake` lorsque `0 <= (cam mod 360) <90` (voir figure 4.3).

```
module type IntakeType =
  contract
    input integer cam;
    output event intake;
    assume 0<=(cam mod 360)<90
    guarantee intake
  end;
```

FIG. 4.2 – Une spécification.

```
module IntakeMode : IntakeType =
  process
    input integer cam;
    output event intake;
    intake:=true when
                  (cam mod 360)<90
  end;
```

FIG. 4.3 – Une implémentation.

Un composant est donc considéré comme une paire $M : I$, consistant en une implémentation M, typée (ou représentée) par une interface I, satisfaisant le contrat associé à I. La sémantique de la spécification I, notée $[\![\mathrm{I}]\!]$, est un ensemble de processus (au sens de la section 2.1) dont les traces satisfont les contrats liés à I. La sémantique $[\![\mathrm{M}]\!]$ du module M est le singleton contenant le processus associé à M.

4.1.2 Encapsulation par foncteurs

La spécification des propriétés considérées pour le moteur se compose de quatre contrats. Chaque contrat spécifie une phase du moteur. Chaque phase est associée à la position de l'arbre à cames et déclenche un événement. Afin d'éviter de spécifier quatre contrats distincts, nous définissons quatre instances d'un contrat générique. Dans ce but, nous définissons un mécanisme d'encapsulation afin de représenter de manière générique les classes d'interfaces ou d'implémentations partageant un modèle paramétrable commun de comportements. Dans l'exemple du moteur, nous avons utilisé un *foncteur* paramétré par la valeur de l'angle de l'arbre à cames associé à chaque phase (voir figure 4.4).

```
module type phase =
 functor(integer min, integer max)
  contract
    input integer cam ;
    output event trigger;
    assume min<=(cam mod 360)<max
    guarantee trigger
 end;
```

FIG. 4.4 – Un foncteur.

```
module type engine = contract
input integer cam ;
output event intake, compression,
             combustion, exhaust;
phase(0, 90)(cam, intake)
and phase(90, 180)(cam, compression)
and phase(180, 270)(cam, combustion)
and phase(270, 360)(cam, exhaust)
end;
```

FIG. 4.5 – Utilisation de foncteur.

Le contrat générique, appelé `Phase` (voir figure 4.5), est paramétré par le nom de la phase à déclencher et par la position limite de l'arbre à cames. Lorsque la position angulaire de l'arbre à cames est dans la position attendue, le contrôleur du moteur active la phase spécifiée par le paramètre `trigger`. Le contrat modélisant le moteur est défini par la composition "**and**" de quatre applications de la forme générique du contrat associé à chaque phase. Chaque application définit une phase particulière du moteur avec ses déclencheurs et ses valeurs angulaires appropriées. La composition est définie par la borne inférieure des quatre contrats (le plus grand des contrats raffinant les quatre contrats). Chaque application du foncteur `Phase` produit un contrat qui est ensuite composé avec les autres afin de produire le contrat spécifiant les propriétés associées à chacune des quatre phases.

4.1.3 Syntaxe du langage de modules

Nous définissons la syntaxe formelle de notre langage de modules présentée sur la figure 4.6. La grammaire est paramétrée par la syntaxe des programmes, notés p ou q, lesquels sont écrits dans un langage de programmation spécifique. Les noms sont notés x ou y. Les types t sont utilisés pour déclarer les paramètres et les variables d'entrée/sortie dans les interfaces. Les hypothèses et les garanties sont décrites par des expressions p et q dans un langage cible ou un langage spécifique au contexte d'utilisation du langage de modules. Une expression *exp* manipule des contrats, des modules, des foncteurs, et des instances de foncteurs ou encore des références.

x, y		noms
p, q		processus
b, c	$::=$ **event** $\mid$ **boolean** $\mid$ **short** $\mid$ **integer** $\mid \ldots$	types de données
t	$::= b \mid$ **input** $b \mid$ **output** $b \mid x \mid t \times t$	types
dec	$::= t\ x\ [, dec]$	déclaration
ag	$::=$ [**assume** p] **guarantee** $q; \mid ag$ **and** ag	contrat
exp	$::=$ **contract** $dec;\ ag$ **end**	contrat
	$\mid$ **process** $dec;\ p$ **end**	processus
	$\mid$ **functor** $(dec)\ exp$	foncteur
	$\mid\ exp$ **and** exp	composition
	$\mid\ x\ (exp^*)$	application
	$\mid$ **let** def **in** exp	définition locale
def	$::=$ **module** [**type**] $x\ =\ exp$	définition d'une spécification
	$\mid$ **module** x [: t] $=\ exp$	définition d'un module
	$\mid\ def; def$	succession de définitions

FIG. 4.6 – Grammaire du langage de modules.

4.1.4 Spécification, implémentation et typage de modules

Dans le langage de modules, les spécifications définissent les propriétés devant être satisfaites par l'environnement, ainsi que la manière dont un composant interagit par le biais de ses variables avec le contexte d'exécution. De plus, les spécifications sont utilisées pour définir les propriétés satisfaites par un composant dans un contexte donné. L'implémentation d'une spécification contient un ensemble compatible de variables d'entrée/sortie, et satisfait le contrat spécifié.

Nous définissons un système de types pour les contrats et les processus dans le langage de modules fondé sur notre algèbre de contrats. Dans la syntaxe du langage, les contrats et les processus sont associés à des noms x. Ces noms peuvent être utilisés pour déclarer et typer les paramètres formels d'un foncteur. Par conséquent, dans le système de types, les noms des contrats ou des processus sont associés à un type de module T.

Le type de base d'un module est de la forme $\tau(I, C)$. La balise τ a pour valeur π, si elle désigne le type d'un processus ($\pi(I, C)$ est le type d'un processus), ou bien γ dans le cas où la balise désigne le type d'un contrat ($\gamma(I, C)$ est le type d'un contrat). L'ensemble I est constitué de paires $x : t$ afin de déclarer que les variables (d'entrée/sortie) x sont de type t. Le contrat de C est une paire de prédicats (p, q), qui représentent les hypothèses p et les garanties q. Le type d'un foncteur $\Lambda(x : S).T$ est constitué du nom des paramètres x et des types S et T, S désignant le type des paramètres formels du foncteur et T le type du résultat de l'instanciation.

$$S, T ::= t \mid \tau(I, C) \mid S \times T \mid \Lambda(x : S).T \quad \tau ::= \gamma \mid \pi$$

Le rôle de l'environnement de typage Γ est de définir l'association $x : T$ des noms x et de leur type T (nous notons $\Gamma(x)$ le type de x dans l'environnement Γ). Le rôle de l'ensemble des contraintes de typage Σ est de construire l'ensemble des relations de raffinement de la forme $C \preceq D$.

$$\Gamma ::= \emptyset \mid \Gamma \cup x : T \qquad \Sigma ::= \emptyset \mid \Sigma \cup C \preceq D$$

4.2 Sous-typage et raffinement

4.2.1 Raffinement de modules

Nous souhaitons définir une relation de sous-typage sur les types t en étendant la relation de raffinement de contrats définie dans la section 2.3 à l'algèbre de types du langage de modules. Dans ce but, nous souhaitons appliquer le principe de sous-typage $S \leq T$ afin d'exprimer que l'ensemble des objets sémantiques représentés par S est inclus dans l'ensemble des objets sémantiques représentés par T.

Par exemple, soient deux processus P et Q, tels que P raffine Q (figure 4.7). Un module M encapsulant P avec une variable d'entrée x de type *long*, et deux variables de sortie a, b de type *short* est un sous-type d'un module N encapsulant Q avec deux variables d'entrée x, y et une variable de sortie a : ainsi M peut être remplacé par N.

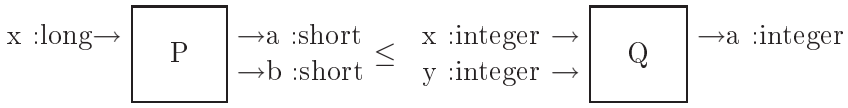


FIG. 4.7 – Raffinement de modules.

Sous les contraintes de typage Σ, s est un sous-type de t , noté $\Sigma \supset s \leq t$ si et seulement si Σ contient (ou implique) la relation $s \leq t$. La définition inductive de la relation de sous-typage $\leq$ débute avec les axiomes de sous-typage relatifs aux types de données. A partir de cette relation de sous-typage, des règles algébriques sont définies, qui sont ensuite utilisées pour définir les règles de déclarations de types de modules.

En particulier, le type d'un module $S = \tau(I, C)$ est un sous-type de $T = \tau(J, D)$, noté $\Sigma \supset S \leq T$, si et seulement si :

- l'ensemble des variables d'entrée de I est inclus dans l'ensemble des variables d'entrée de J,
- l'ensemble des variables de sortie de J est inclus dans l'ensemble des variables de sortie de I,
- les variables d'entrée de J sont des sous-types des variables d'entrée correspondantes dans I,
- les variables de sortie de I sont des sous-types des variables de sortie correspondantes dans J,
- le contrat C raffine D.

Ainsi, du point de vue de l'interface, un module S raffine un module T, si S apporte plus de précision au niveau du typage des variables de sortie à partir d'un typage moins précis des variables d'entrée. Cette approche se retrouve dans la relation de raffinement de contrats : si C raffine D, alors

- les garanties de C sont plus fortes que les garanties de D,
- les hypothèses de C définissent un contexte d'exécution moins précis que le contexte d'exécution modélisé par les hypothèses de D.

Dans la règle relative aux foncteurs, nous écrivons $V[y/x]$ pour désigner la substitution de x par y dans V (en posant que y n'apparaît pas dans V). La figure 4.8 explicite l'ensemble des règles de sous-typage.

- la règle (1) exprime la réflexivité de la relation de raffinement,
- la règle (2) exprime la relation de sous-typage sur les types simples,
- la règle (3) exprime la relation de raffinement des variables d'entrée,
- la rèlge (4) exprime la relation de raffinement des variables de sortie,
- la règle (5) exprime la transitivité de la relation de raffinement,
- la règle (6) exprime la relation de raffinement des ensembles de couples (*variable* : *type*) : si le type S d'une variable x appartenant à l'ensemble I raffine le type T de la variable x dans l'ensemble J, et que l'ensemble $I \backslash (x : S)$ raffine $J \backslash (x : T)$, alors I raffine J,

$$(1)\ T \leq T \quad (2)\ \textbf{event} \leq \textbf{boolean} \quad \textbf{short} \leq \textbf{integer} \leq \textbf{long}$$

$$(3)\ \frac{\Sigma \supset b \leq c}{\Sigma \supset \textbf{input}\ c \leq \textbf{input}\ b} \quad (4)\ \frac{\Sigma \supset b \leq c}{\Sigma \supset \textbf{output}\ b \leq \textbf{output}\ c}$$

$$(5)\ \frac{\Sigma \supset S \leq T \quad \Sigma \supset T \leq U}{\Sigma \supset S \leq U} \quad (6)\ \frac{\Sigma \supset I \leq J \quad \Sigma \supset S \leq T}{\Sigma \supset (x : S) \cup I \leq (x : T) \cup J}$$

$$(7)\ \frac{\Sigma \supset I \leq J \quad x \notin \mathit{vars}(I)}{\Sigma \supset I \leq (x : \textbf{input}\ b) \cup J} \quad (8)\ \frac{\Sigma \supset I \leq J \quad x \notin \mathit{vars}(J)}{\Sigma \supset (x : \textbf{output}\ b) \cup I \leq J}$$

$$(9)\ \begin{array}{c}\tau \leq \gamma \\ \pi \leq \tau\end{array} \quad (10)\ \frac{\Sigma \supset I \leq J \quad (C \preceq D) \in \Sigma \quad \tau \leq \tau'}{\Sigma \supset \tau(I, C) \leq \tau'(J, D)}$$

$$(11)\ \frac{\Sigma \supset S \leq U \quad T \leq V}{\Sigma \supset S \times T \leq U \times V} \quad (12)\ \frac{\Sigma \supset U \leq S \quad \Sigma \supset T \leq V[y/x]}{\Sigma \supset \Lambda(x : S).T \leq \Lambda(y : U).V}$$

FIG. 4.8 – Les règles de sous-typage

- selon la règle (7), si un ensemble de variables d'entrée I contient moins de variables que J, alors I raffine J,
- selon la règle (8), si un ensemble de variables de sortie I contient plus de variables que J, alors I raffine J,
- la règle (9) exprime une relation d'ordre sur les types de modules : le type d'un processus est un sous-type du type associé à un contrat,
- la règle (10) exprime la relation de raffinement de modules : un module M raffine un module M' si et seulement si l'ensemble des types des variables d'entrée/sortie de M raffine celui de M', et si le contrat associé à M raffine celui associé à M',
- la règle (11) exprime le raffinement compositionnel de modules,
- la règle (12) exprime la relation de raffinement de foncteurs : un foncteur F raffine un foncteur F' si et seulement si l'ensemble des types de paramètres de F' raffine celui de F, et si le type du module produit par l'application du foncteur F raffine celui de F'.

4.2.2 Raffinement de contrats

Nous pouvons interpréter la relation $\Sigma \supset C \preceq D$ comme l'expression de l'enregistrement de la contrainte du raffinement de C par D dans Σ. Cette relation de raffinement correspond à une obligation de preuve dans le langage cible, dont le sens est défini par la relation sémantique $[\![C]\!] \preceq [\![D]\!]$ dans l'algèbre de contrats (où $[\![\mathrm{C}]\!]$ et $[\![\mathrm{D}]\!]$ désignent la valeur sémantique de C et D dans l'algèbre de contrats). La validité de cette contrainte peut par exemple être prouvée par un prouveur.

L'ensemble Σ des relations de raffinement $C \preceq D$ est obtenu par la décomposition structurelle des relations de sous-typage de la forme $s \leq t$ en axiomes élémentaires (par exemple : **event** $\leq$ **boolean**), ou en obligations de preuves $C \preceq D$.

4.2.3 Sous-typage et types simples

Le type d'un module étant une paire $\tau(I, C)$, et l'ensemble des contrats $\mathbb{C}$ étant une algèbre de Heyting, nous souhaitons définir une relation de sous-typage qui définit une algèbre de Heyting sur l'ensemble des types des variables de flots, afin de pouvoir caractériser l'ensemble des modules (l'ensemble des paires : ensemble de variables d'entrée/sortie et contrat) par une algèbre de Heyting.

Les types simples sont utilisés pour typer certains paramètres des modules et typer les données transportées par les variables de flots. Soit $\mathcal{T}$, l'ensemble des types simples, où un type $t \in \mathcal{T}$ caractérise l'ensemble des objets sémantiques appartenant à ce type. La relation $\leq$ est une relation d'ordre partiel caractérisant le sous-typage sur $\mathcal{T}$, tel que $\forall t_1, t_2 \in \mathcal{T}$, $t_1 \leq t_2$ si l'ensemble des objets sémantiques appartenant au type t_1 est inclus dans l'ensemble des objets sémantiques du type t_2 (voir la règle (2) de la section 4.2.1). La borne inférieure de t_1 et t_2, notée $t_1 \sqcap t_2$, est l'ensemble des objets appartenant à t_1 et t_2. La borne supérieure de t_1 et t_2, notée $t_1 \sqcup t_2$, est l'ensemble des objets appartenant à t_1 ou t_2. $(\mathcal{T}, \leq)$ est un treillis ayant pour infimum *void* (équivalent à l'ensemble vide d'objets), et pour supremum *null* (équivalent à l'ensemble de tous les objets). Le complémentaire $\neg t$ d'un type $t \in \mathcal{T}$ est l'ensemble des objets sémantiques n'appartenant pas à t tel que $\neg t = null - t$. $(\mathcal{T}, \leq)$ est une algèbre booléenne, et par conséquent une algèbre de Heyting.

Le type des variables de flots est caractérisé par le type des données du flot, mais aussi par le fait qu'il s'agit de variables d'entrées ou de variables de sorties. Les balises **input** et **output** permettent de distinguer les variables d'entrée (**input**) et les variables de sortie (**output**).

Soit $\mathcal{OF}$, l'ensemble des types des variables de flots de sortie. Considérons la relation d'ordre partiel, $\leq$, tel que $\forall x, y \in \mathcal{T}$, **output** $x \leq$ **output** $y \iff x \leq y$ (règle (3) de la section 4.2.1). $(\mathcal{T}, \leq)$ étant une algèbre booléenne, il en découle que $(\mathcal{OF}, \leq)$ est aussi une algèbre booléenne avec pour supremum **output** *null*, et pour infimum **output** *void*. La borne inférieure notée $\sqcap$ vérifie : $\forall$(**output** x), (**output** y) $\in \mathcal{OF}$, (**output** x) $\sqcap$ (**output** y) $\iff$ **output** ($x \sqcap y$), et la borne supérieure notée $\sqcup$: $\forall$(**output** x), (**output** y) $\in \mathcal{OF}$, (**output** $x \sqcup$ **output** y) $\iff$ **output** ($x \sqcup y$). Pour tout **output** $x \in \mathcal{OF}$, le complémentaire de **output** x noté $\neg$(**output** x) vérifie $\neg$**output** $x \iff$ **output** $\neg x$. La figure 4.9 illustre une structure de types des variables de sortie en considérant les types simples *boolean*, *event*, *long*, *integer*, *short*, tels qu'ils sont décrits dans le manuel de référence de SIGNAL [GLD94].

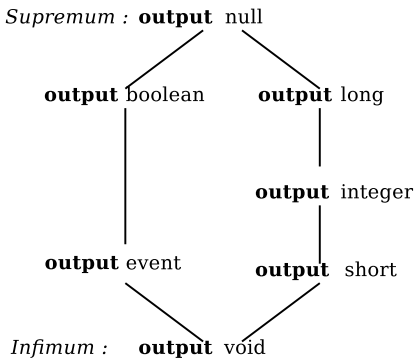


FIG. 4.9 – Un treillis des types des variables de sortie.

Soit $\mathcal{IF}$, l'ensemble des types des variables de flots d'entrée. Considérons la relation d'ordre partiel, $\leq$, tel que $\forall x, y \in \mathcal{T}$, **input** $x \leq$ **input** $y \iff y \leq x$ (règle (3) de la section 4.2.1). $(\mathcal{T}, \leq)$ étant une algèbre booléenne, il en découle que $(\mathcal{IF}, \leq)$ est aussi une algèbre booléenne avec pour supremum **input** *void*, et pour infimum **input** *null*. La borne inférieure notée $\sqcap$ vérifie : $\forall$(**input** x), (**input** y) $\in \mathcal{IF}$, (**input** $x \sqcap$ **input** y) $\iff$ **input** $(x \sqcup y)$. La borne supérieure notée $\sqcup$: $\forall$(**input** x), (**input** y) $\in \mathcal{IF}$, (**input** $x \sqcup$ **input** y) $\iff$ **input** $(x \sqcap y)$. Pour tout $x \in \mathcal{IF}$, le complémentaire de **input** x noté $\neg$(**input** x) vérifie $\neg$**input** x $\iff$ **input** $\neg x$. La figure 4.10 illustre une structure de types des variables d'entrée en considérant les types simples *boolean*, *event*, *long*, *integer*, *short*, tels qu'ils sont décrits dans le manuel de référence de SIGNAL [GLD94].

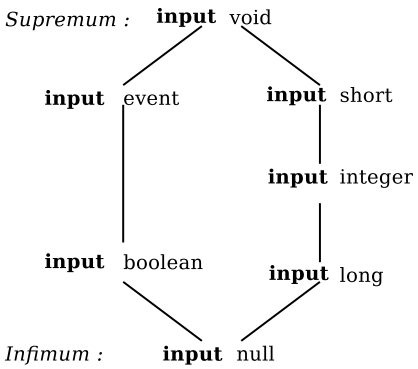


FIG. 4.10 – Un treillis des types des variables d'entrée.

Les algèbres booléennes $(\mathcal{IF}, \leq)$ et $(\mathcal{OF}, \leq)$ modélisant respectivement l'ensemble des types des variables de flots d'entrée et l'ensemble des types des variables de flots de sortie, vont maintenant permettre de définir une algèbre booléenne sur les ensembles de variables d'entrée/sortie des modules.

4.2.4 Une algèbre booléenne sur les ensembles de types de variables de sortie

Soit $\mathcal{OE}$ l'ensemble des ensembles des couples associant une variable de sortie à son type. Soit la relation d'ordre partiel $\leq$ décrite par les règles (6) et (8) de la section 4.2.1. Rappelons ces règles :

$$(6)\ \frac{\Sigma \supset I \leq J \quad \Sigma \supset S \leq T}{\Sigma \supset (x : S) \cup I \leq (x : T) \cup J} \qquad (8)\ \frac{\Sigma \supset I \leq J \quad x \notin \mathit{vars}(J)}{\Sigma \supset (x : \mathbf{output}\ b) \cup I \leq J}$$

Nous avons $I \leq J$ si et seulement si, I contient plus de variables que J, et que le type des variables communes à I et J soit plus fin dans I. Ce qui est équivalent à : $I \leq J \ \Leftrightarrow\ \forall (x : \mathbf{output}\ t_1) \in J, \exists (x : \mathbf{output}\ t_2) \in I$ tel que $t_2 \leq t_1$.

Preuve : Montrons que $\leq$ est une relation d'ordre sur $\mathcal{OE}$.

- $\forall I \in \mathcal{OE}$, on a toujours $I \leq I$. La relation $\leq$ est clairement réflexive.
- Soient $I,\ J,\ K \in \mathcal{OE}$ tels que $I \leq J$ et $J \leq K$. Nous avons alors :
  $I \leq J \ \Leftrightarrow\ \forall (x : \mathbf{output}\ t_J) \in J, \exists (x : \mathbf{output}\ t_I) \in I$ tel que $t_I \leq t_J$
  $J \leq K \ \Leftrightarrow\ \forall (x : \mathbf{output}\ t_K) \in K, \exists (x : \mathbf{output}\ t_J) \in J$ tel que $t_J \leq t_K$
  Or,
  $$\left.\begin{array}{r} \forall (x \in \mathbf{output}\ t_J) \in J, \exists (x : \mathbf{output}\ t_I) \in I \text{ tel que } t_I \leq t_J \\ \wedge \\ \forall (x \in \mathbf{output}\ t_K) \in K, \exists (x : \mathbf{output}\ t_J) \in J \text{ tel que } t_J \leq t_K \end{array}\right\} \Rightarrow$$
  $$\left.\forall (x \in \mathbf{output}\ t_K) \in K, \exists (x : \mathbf{output}\ t_I) \in I \text{ tel que } t_J \leq t_K \right\} \Rightarrow \quad J \leq K$$
  Par conséquent, la relation $\leq$ est transitive.
- Soient $I,\ J \in \mathcal{OE}$ tels que $I \leq J$ et $J \leq I$. Nous avons alors :
  $I \leq J \ \Leftrightarrow\ \forall (x \in \mathbf{output}\ t_1) \in J, \exists (x : \mathbf{output}\ t_2) \in I$ tel que $t_2 \leq t_1$
  $J \leq I \ \Leftrightarrow\ \forall (x \in \mathbf{output}\ t_1) \in I, \exists (x : \mathbf{output}\ t_2) \in J$ tel que $t_2 \leq t_1$
  Or,
  $$\left.\begin{array}{r} \forall (x \in \mathbf{output}\ t_1) \in J, \exists (x : \mathbf{output}\ t_2) \in I \text{ tel que } t_2 \leq t_1 \\ \wedge \\ \forall (x \in \mathbf{output}\ t_1) \in I, \exists (x : \mathbf{output}\ t_2) \in J \text{ tel que } t_2 \leq t_1 \end{array}\right\} \Rightarrow \quad I = J$$
  Par conséquent, la relation $\leq$ est antisymétrique.

$(\mathcal{OE},\ \leq)$ est bien un ordre partiel. □

La borne inférieure de $I \in \mathcal{OE}$ et $J \in \mathcal{OE}$, notée $I \sqcap J$, est définie de la manière suivante :

$$\begin{array}{ll} \text{Si } \nexists t \in \mathcal{T} \text{ tel que } (x : \mathbf{output}\ t) \in J & \\ \text{alors } (I \cup (x : \mathbf{output}\ b)) \sqcap J & = (I \sqcap J) \cup (x : \mathbf{output}\ b) \\ \text{Si } \nexists t \in \mathcal{T} \text{ tel que } (x : \mathbf{output}\ t) \in I & \\ \text{alors } I \sqcap (J \cup (x : \mathbf{output}\ b)) & = (I \sqcap J) \cup (x : \mathbf{output}\ b) \\ \text{Sinon } (I \cup (x : S)) \sqcap (J \cup (x : T)) & = (I \sqcap J) \cup (x : S \sqcap T) \end{array}$$

$\forall I, J \in \mathcal{OE}$, la borne inférieure est l'union des ensembles de variables de I et J, en considérant la borne inférieure des types des variables communes. Ce qui est équivalent à :

$$I \sqcap J = \left(\begin{array}{ll} & \left\{ \begin{array}{lll} (x : \textbf{output}\ (t_I \sqcap\ t_J)) & | & (x : \textbf{output}\ t_I \in I) \\ & & (x : \textbf{output}\ t_J) \in J) \end{array} \right\} \\ \bigcup & \left\{ \begin{array}{lll} (x : \textbf{output}\ t_J) & | & \forall t_I \in \mathcal{T}, (x : \textbf{output}\ t_I \notin I) \\ & & (x : \textbf{output}\ t_J) \in J) \end{array} \right\} \\ \bigcup & \left\{ \begin{array}{lll} (x : \textbf{output}\ t_I) & | & (x : \textbf{output}\ t_I \in I), \\ & & \forall t_J \in ST, (x : \textbf{output}\ t_J) \notin J) \end{array} \right\} \end{array} \right)$$

Preuve : Soit $I,\ J,\ T \in \mathcal{OE}$ tels que $T \leq I$ et $T \leq J$. Alors,

$$(T \leq I) \wedge (T \leq J) \Leftrightarrow \left\{ \begin{array}{l} \forall (x : \textbf{output}\ t_1) \in I,\ \exists (x : \textbf{output}\ t_2) \in T \text{ tel que } t_2 \leq t_1 \\ \wedge \\ \forall (x : \textbf{output}\ t_1) \in J,\ \exists (x : \textbf{output}\ t_2) \in T \text{ tel que } t_2 \leq t_1 \end{array} \right.$$

Nous en déduisons la forme générale de T :

$$T = \left(\begin{array}{ll} & \left\{ \begin{array}{lll} (x : \textbf{output}\ t_3) & | & ((x : \textbf{output}\ t_1) \in I), \\ & & ((x : \textbf{output}\ t_2) \in J), \\ & & t_3 \in \mathcal{T}, t_3 \leq t_1,\ t_3 \leq t_2 \end{array} \right\} \\ \bigcup & \left\{ \begin{array}{lll} (x : \textbf{output}\ t_3) & | & (x : \textbf{output}\ t_1 \in I), \\ & & \forall t_2 \in \mathcal{T},\ ((x : \textbf{output}\ t_2) \notin J), \\ & & t_3 \in \mathcal{T}, t_3 \leq t_1 \end{array} \right\} \\ \bigcup & \left\{ \begin{array}{lll} (x : \textbf{output}\ t_3) & | & \forall t_1 \in \mathcal{T},\ ((x : \textbf{output}\ t_1) \notin I), \\ & & ((x : \textbf{output}\ t_2) \in J), \\ & & t_3 \in \mathcal{T}, t_3 \leq t_2 \end{array} \right\} \\ \bigcup & \left(\begin{array}{lll} H \text{ tel que } H \in \mathcal{OE}, & & \forall t_2, t_3 \in \mathcal{T}, \\ & & ((x : \textbf{output}\ t_2) \notin I), \\ & & ((x : \textbf{output}\ t_3) \notin J), \\ & & \exists t_1 \in \mathcal{T}, ((x : \textbf{output}\ t_1) \in H) \end{array} \right) \end{array} \right)$$

Nous en déduisons l'expression du plus grand des minorants de I et J :

$$I \sqcap J = \left(\begin{array}{ll} & \left\{ \begin{array}{lll} (x : \textbf{output}\ (t_I \sqcap\ t_J)) & | & ((x : \textbf{output}\ t_I) \in I), \\ & & ((x : \textbf{output}\ t_J) \in J) \end{array} \right\} \\ \bigcup & \left\{ \begin{array}{lll} (x : \textbf{output}\ t_J) & | & \forall t_J \in \mathcal{T}, ((x : \textbf{output}\ t_I) \notin I), \\ & & ((x : \textbf{output}\ t_J) \in J) \end{array} \right\} \\ \bigcup & \left\{ \begin{array}{lll} (x : \textbf{output}\ t_I) & | & ((x : \textbf{output}\ t_I) \in I), \\ & & \forall t_J \in \mathcal{T}, ((x : \textbf{output}\ t_J) \notin J) \end{array} \right\} \end{array} \right)$$

Il est trivial que $T \leq I \sqcap J$. □

La borne supérieure de $I \in \mathcal{OE}$ et $J \in \mathcal{OE}$, notée $I \sqcup J$, est définie de la manière suivante :

$$\begin{array}{ll}
\text{Si } \nexists t \in \mathcal{T} \text{ tel que } (x : \textbf{output } t) \in J & \\
\text{ alors } (I \cup (x : \textbf{output } b)) \sqcup J & = (I \sqcup J) \\
\text{Si } \nexists t \in \mathcal{T} \text{ tel que } (x : \textbf{output } t) \in I & \\
\text{ alors } I \sqcup (J \cup (x : \textbf{output } b)) & = (I \sqcup J) \\
\text{Sinon } (I \cup (x : S)) \sqcup (J \cup (x : T)) & = (I \sqcup J) \cup (x : S \sqcup T)
\end{array}$$

$\forall I, J \in \mathcal{OE}$, la borne supérieure est l'intersection des ensembles de variables de I et J, en considérant la borne supérieure des types des variables communes. Ce qui est équivalent à :

$$I \sqcup J \quad = \quad \left\{ \begin{array}{lll} (x : \textbf{output } (t_I \sqcup\ t_J)) & | & ((x : \textbf{output } t_I) \in I) \\ & & ((x : \textbf{output } t_J) \in J) \end{array} \right\}$$

Preuve : Soit $I,\ J,\ T \in OEF$ tels que $I \leq T$ et $J \leq T$. Or,

$$(I \leq T) \wedge (J \leq T) \Leftrightarrow \left\{ \begin{array}{l} \forall (x : \textbf{output } t_1) \in T,\ \exists (x : \textbf{output } t_2) \in I \text{ tel que } t_2 \leq t_1 \\ \wedge \\ \forall (x : \textbf{output } t_1) \in T,\ \exists (x : \textbf{output } t_2) \in J \text{ tel que } t_2 \leq t_1 \end{array} \right.$$

Nous en déduisons la forme générale de T :

$$T \quad = \quad \left(\begin{array}{ll} & \left\{ \begin{array}{lll} (x : \textbf{output } t_3) & | & ((x : \textbf{output } t_1) \in I), \\ & & ((x : \textbf{output } t_2) \in J), \\ & & t_3 \in \mathcal{T}, t_1 \leq t_3,\ t_2 \leq t_3 \end{array} \right\} \\ \bigcap & H \text{ avec } H \in \mathcal{OE} \end{array} \right)$$

Nous en déduisons l'expression du plus petit des majorants de I et J :

$$I \sqcup J \quad = \quad \left\{ \begin{array}{lll} (x : \textbf{output } (t_I \sqcup\ t_J)) & | & ((x : \textbf{output } t_I) \in I), \\ & & ((x : \textbf{output } t_J) \in J) \end{array} \right\}$$

Il est trivial que $I \sqcup J \leq T$. □

Dans ce treillis, le complémentaire d'un ensemble I de variables d'entrée/sortie, noté $\neg I$, est défini par :

Si $\nexists t \in \mathcal{T}$ tel que $(x : \textbf{output } t) \in I$ alors $\neg(I \cup (x : \textbf{output } null)) = \neg I$
Sinon $\neg(I \cup (x : \textbf{output } b)) = (\neg I \cup (x : \textbf{output } \neg b))$

Ce qui est équivalent à :

$$\neg I = \left(\begin{array}{ll} & \left\{ \begin{array}{lll} (x : \textbf{output } (\neg t_I) & | & ((x : \textbf{output } t_I) \in I) \end{array} \right\} \\ \bigcup & \left\{ \begin{array}{lll} (x : \textbf{output } null) & | & \nexists t \in \mathcal{T} \text{ tel que } ((x : \textbf{output } t_I) \in I) \end{array} \right\} \end{array} \right)$$

$(\mathcal{OF}, \leq)$ étant une algèbre booléenne, il en découle que $(\mathcal{OE}, \leq)$ est aussi une algèbre booléenne avec pour supremum $\emptyset$, et pour infimum $\{(x : \textbf{output } null) \mid x \in \mathcal{V}\}$.

Preuve : $(\mathcal{OE}, \sqsubseteq)$ est un treillis, nous avons alors :
- $\mathbf{R} \sqcup (\mathbf{S} \sqcup \mathbf{T}) = (\mathbf{R} \sqcup \mathbf{S}) \sqcup \mathbf{T}$ (et la propriété duale)
- $\mathbf{R} \sqcup \mathbf{S} = \mathbf{S} \sqcup \mathbf{R}$ (et la propriété duale)

– $\mathbf{R} \sqcup (\mathbf{R} \sqcap \mathbf{S}) = \mathbf{R}$ (et la propriété duale)

Par conséquent, nous avons seulement à prouver que :

– $\widetilde{\mathbf{R}} \sqcap \mathbf{R} = \{\mho\}$; ce qui est une conséquence directe de la définition de la borne inférieure et du complémentaire.

– $\widetilde{\mathbf{R}} \sqcup \mathbf{R} = \mathbb{P}^\star$; ce qui est une conséquence directe de la définition de la borne supérieure et du complémentaire.

– $\mathbf{R} \sqcup (\mathbf{S} \sqcap \mathbf{T}) = (\mathbf{R} \sqcup \mathbf{S}) \sqcap (\mathbf{R} \sqcup \mathbf{T})$ (ou la propriété duale).
Soit $R, S, T \in \mathcal{OE}$,

$$
\begin{aligned}
&R \sqcap (S \sqcup T) = \\
&\left(\begin{array}{ll}
 & \left\{ (x : \mathbf{output}\ (t_R \sqcap\ t_{S \sqcup T})) \ \middle|\ \begin{array}{l} (x : \mathbf{output}\ t_R \in R), \\ (x : \mathbf{output}\ t_{S \sqcap T}) \in (S \sqcup T)) \end{array} \right\} \\
\bigcup & \left\{ (x : \mathbf{output}\ t_R) \ \middle|\ \begin{array}{l} (x : \mathbf{output}\ t_R \in R), \\ \forall t_{S \sqcup T} \in ST,\ (x : \mathbf{output}\ t_{S \sqcup T}) \notin (S \sqcup T)) \end{array} \right\} \\
\bigcup & \left\{ (x : \mathbf{output}\ t_{S \sqcup T}) \ \middle|\ \begin{array}{l} \forall t_R \in ST,\ (x : \mathbf{output}\ t_R \notin R), \\ (x : \mathbf{output}\ t_{S \sqcup T}) \in (S \sqcup T)) \end{array} \right\}
\end{array} \right) = \\
&\left(\begin{array}{ll}
 & \left\{ (x : \mathbf{output}\ (t_R \sqcap (t_S \sqcup t_T))) \ \middle|\ \begin{array}{l} (x : \mathbf{output}\ t_R \in R), \\ (x : \mathbf{output}\ t_S) \in S), \\ (x : \mathbf{input}\ t_T) \in T) \end{array} \right\} \\
\bigcup & \left\{ (x : \mathbf{output}\ t_R) \ \middle|\ \begin{array}{l} (x : \mathbf{output}\ t_R \in R), \\ \forall t_S \in ST,\ (x : \mathbf{output}\ t_S) \notin S), \\ \forall t_T \in ST,\ (x : \mathbf{output}\ t_T) \notin T) \end{array} \right\} \\
\bigcup & \left\{ (x : \mathbf{output}\ (t_S \sqcup t_T)) \ \middle|\ \begin{array}{l} \forall t_R \in ST,\ (x : \mathbf{output}\ t_R \notin R), \\ (x : \mathbf{output}\ t_S) \in S), \\ (x : \mathbf{output}\ t_T) \in T) \end{array} \right\}
\end{array} \right) = \\
&\left(\begin{array}{ll}
 & \left\{ (x : \mathbf{output}\ ((t_R \sqcap t_S) \sqcup (t_R \sqcap t_T))) \ \middle|\ \begin{array}{l} (x : \mathbf{output}\ t_R \in R), \\ (x : \mathbf{output}\ t_S) \in S), \\ (x : \mathbf{output}\ t_T) \in T) \end{array} \right\} \\
\bigcup & \left\{ (x : \mathbf{output}\ t_R) \ \middle|\ \begin{array}{l} (x : \mathbf{output}\ t_R \in R), \\ \forall t_S \in ST,\ (x : \mathbf{output}\ t_S) \notin S), \\ \forall t_T \in ST,\ (x : \mathbf{output}\ t_T) \notin T) \end{array} \right\} \\
\bigcup & \left\{ (x : \mathbf{output}\ (t_S \sqcup t_T)) \ \middle|\ \begin{array}{l} \forall t_R \in ST,\ (x : \mathbf{output}\ t_R \notin R), \\ (x : \mathbf{output}\ t_S) \in S), \\ (x : \mathbf{output}\ t_T) \in T) \end{array} \right\}
\end{array} \right) = \\
&\left(\begin{array}{ll}
 & \left\{ (x : \mathbf{output}\ (t_{R \sqcap S} \sqcup t_{R \sqcap T})) \ \middle|\ \begin{array}{l} (x : \mathbf{output}\ t_{R \sqcap S}) \in (R \sqcap S), \\ (x : \mathbf{output}\ t_{R \sqcap T}) \in (R \sqcap T)) \end{array} \right\} \\
\bigcup & \left\{ (x : \mathbf{output}\ t_{R \sqcap S} \ \middle|\ \begin{array}{l} (x : \mathbf{output}\ t_{R \sqcap S} \in (R \sqcap S)), \\ \forall t_{R \sqcap T} \in ST,\ (x : \mathbf{output}\ t_{R \sqcap T}) \notin (R \sqcap T)) \end{array} \right\} \\
\bigcup & \left\{ (x : \mathbf{output}\ t_{R \sqcap T}) \ \middle|\ \begin{array}{l} (x : \mathbf{output}\ t_{R \sqcap S} \in (R \sqcap S)), \\ \forall t_{R \sqcap T} \in ST,\ (x : \mathbf{output}\ t_{R \sqcap T}) \notin (R \sqcap T)) \end{array} \right\}
\end{array} \right) \\
&= (R \sqcap S) \sqcup (R \sqcap T)
\end{aligned}
$$

Le treillis est distributif.

Donc $(\mathcal{OE}, \leq)$ est une algèbre booléenne.

□

4.2.5 Une algèbre booléenne sur les ensembles de types de variables d'entrée

Soit $\mathcal{IE}$ l'ensemble des ensembles des couples associant une variable d'entrée à son type. Soit la relation d'ordre partiel $\leq$ décrite par les règles (6) et (7) de la section 4.2.1. Rappelons ces règles :

$$(6)\ \frac{\Sigma \supset I \leq J \quad \Sigma \supset S \leq T}{\Sigma \supset (x : S) \cup I \leq (x : T) \cup J} \qquad (7)\ \frac{\Sigma \supset I \leq J \quad x \notin \mathit{vars}(J)}{\Sigma \supset I \leq (x : \mathbf{input}\ b) \cup J}$$

Nous avons $I \leq J$ si et seulement si, I contient moins de variables que J, et que le type des variables communes à I et J soit plus fin dans J. Ce qui est équivalent à : $I \leq J \ \Leftrightarrow\ \forall (x \in \mathbf{input}\ t_1) \in I, \exists (x : \mathbf{input}\ t_2) \in J$ tel que $t_1 \leq t_2$.

De la même manière que $(\mathcal{OE}, \leq)$ est une algèbre booléenne (voir la sous-section 4.2.4 précédente), il découle de l'algèbre booléenne $(\mathcal{IF}, \leq)$, une algèbre booléenne $(\mathcal{IE}, \leq)$ ayant pour supremum $\{(x : \mathbf{input}\ \mathit{null}) \mid x \in \mathcal{V}\}$, et pour infimum $\emptyset$. La borne inférieure de $I \in \mathcal{IE}$ et $J \in \mathcal{IE}$, notée $I \sqcap J$, est définie de la manière suivante :

$$\begin{array}{ll}
\text{Si } \nexists t \in \mathcal{T} \text{ tel que } (x : \mathbf{input}\ t) \in J & \\
\text{ alors } (I \cup (x : \mathbf{input}\ b)) \sqcap J & = (I \sqcap J) \\
\text{Si } \nexists t \in \mathcal{T} \text{ tel que } (x : \mathbf{input}\ t) \in I & \\
\text{ alors } I \sqcap (J \cup (x : \mathbf{input}\ b)) & = (I \sqcap J) \\
\text{Sinon } (I \cup (x : S)) \sqcap (J \cup (x : T)) & = (I \sqcap J) \cup (x : S \sqcap T)
\end{array}$$

$\forall I, J \in \mathcal{IE}$, la borne inférieure est l'intersection des ensembles de variables de I et J, en considérant la borne supérieure des types des variables communes.

La borne supérieure de $I \in \mathcal{OE}$ et $J \in \mathcal{OE}$, notée $I \sqcup J$, est définie de la manière suivante :

$$\begin{array}{ll}
\text{Si } \nexists t \in \mathcal{T} \text{ tel que } (x : \mathbf{input}\ t) \in J & \\
\text{ alors } (I \cup (x : \mathbf{input}\ b)) \sqcup J & = (I \sqcup J) \cup (x : \mathbf{input}\ b) \\
\text{Si } \nexists t \in \mathcal{T} \text{ tel que } (x : \mathbf{input}\ t) \in I & \\
\text{ alors } I \sqcup (J \cup (x : \mathbf{input}\ b)) & = (I \sqcup J) \cup (x : \mathbf{input}\ b) \\
\text{Sinon } (I \cup (x : S)) \sqcup (J \cup (x : T)) & = (I \sqcup J) \cup (x : S \sqcup T)
\end{array}$$

$\forall I, J \in \mathcal{IE}$, la borne supérieure est l'union des ensembles de variables de I et J,

en considérant la borne inférieure des types des variables communes.
Dans cette algèbre, le complémentaire d'un ensemble I, noté $\neg I$, est défini par :

Si $\nexists t \in \mathcal{T}$ tel que $(x : \textbf{input}\ t) \in I$ alors $\neg(I \cup (x : \textbf{input}\ null))=\neg I$
Sinon $\neg(I \cup (x : \textbf{input}\ b))=\neg I \cup (x : \textbf{input}\ \neg b)$

Les algèbres booléennes $(\mathcal{IE}, \leq)$ et $(\mathcal{OE}, \leq)$ modélisant respectivement les ensembles d'ensembles des types de variables de flots d'entrée et l'ensemble d'ensembles des types des variables de flots de sortie, vont maintenant permettre de définir une algèbre booléenne sur les ensembles d'ensembles des types de variables d'entrée/sortie des modules.

4.2.6 Ensembles de types de variables d'entrée/sortie

Les opérateurs d'intersection (noté $\sqcap$) et d'union (noté $\sqcup$) sont étendus afin de combiner l'ensemble des variables d'entrée et l'ensemble des variables de sortie de deux modules.

L'opérateur d'intersection est défini de la manière suivante :

Si $\nexists t \in \mathcal{T}$ tel que $(x : \textbf{input}\ t) \in J$
alors $(I \cup (x : \textbf{input}\ b)) \sqcap J \quad =(I \sqcap J)$
Si $\nexists t \in \mathcal{T}$ tel que $(x : \textbf{input}\ t) \in I$
alors $I \sqcap (J \cup (x : \textbf{input}\ b)) \quad =(I \sqcap J)$
Si $\nexists t \in \mathcal{T}$ tel que $(x : \textbf{output}\ t) \in J$
alors $(I \cup (x : \textbf{output}\ b)) \sqcap J \quad =(I \sqcap J) \cup (x : \textbf{output}\ b)$
Si $\nexists t \in \mathcal{T}$ tel que $(x : \textbf{output}\ t) \in I$
alors $I \sqcap (J \cup (x : \textbf{output}\ b)) \quad =(I \sqcap J) \cup (x : \textbf{output}\ b)$
Sinon $(I \cup (x : S)) \sqcap (J \cup (x : T)) \quad =(I \sqcap J) \cup (x : S \sqcap T)$

Par exemple, soient deux contrats $\mathbf{C}_1$ et $\mathbf{C}_2$. La borne inférieure (figure 4.11) d'un module M satisfaisant $\mathbf{C}_1$ avec une variable d'entrée x de type *long*, et deux variables de sortie a, b de type *short*, et d'un module N satisfaisant $\mathbf{C}_2$ avec deux variables d'entrée x, y et une variable de sortie a est un module satisfaisant ($\mathbf{C}_1 \Downarrow \mathbf{C}_2$) avec une variable d'entrée x de type *long*, et deux variables de sortie a, b de type *short*.

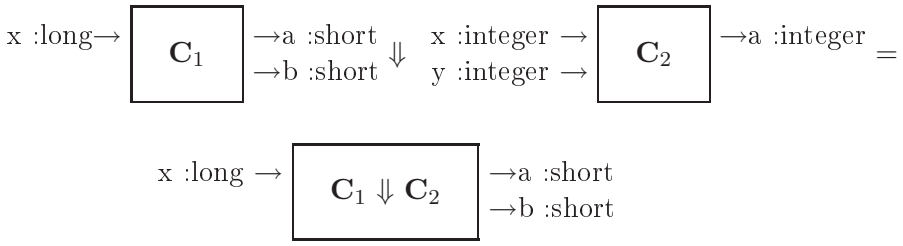


FIG. 4.11 – Borne inférieure de modules.

L'opérateur d'union est défini de la manière suivante :

$$
\begin{array}{ll}
\text{Si } \nexists t \in \mathcal{T} \text{ tel que } (x : \textbf{input}\ t) \in J & \\
\text{ alors } (I \cup (x : \textbf{input}\ b)) \sqcup J & = (I \sqcup J) \cup (x : \textbf{input}\ b) \\
\text{Si } \nexists t \in \mathcal{T} \text{ tel que } (x : \textbf{input}\ t) \in I & \\
\text{ alors } I \sqcup (J \cup (x : \textbf{input}\ b)) & = (I \sqcup J) \cup (x : \textbf{input}\ b) \\
\text{Si } \nexists t \in \mathcal{T} \text{ tel que } (x : \textbf{output}\ t) \in J & \\
\text{ alors } (I \cup (x : \textbf{output}\ b)) \sqcup J & = (I \sqcup J) \\
\text{Si } \nexists t \in \mathcal{T} \text{ tel que } (x : \textbf{output}\ t) \in I & \\
\text{ alors } I \sqcup (J \cup (x : \textbf{output}\ b)) & = (I \sqcup J) \\
\text{Sinon } (I \cup (x : S)) \sqcup (J \cup (x : T)) & = (I \sqcup J) \cup (x : S \sqcup T)
\end{array}
$$

Par exemple, soient deux contrats $\mathbf{C}_1$ et $\mathbf{C}_2$. La borne supérieure (figure 4.12) d'un module M satisfaisant $\mathbf{C}_1$ avec une variable d'entrée x de type *long*, et deux variables de sortie a, b de type *short*, et d'un module N satisfaisant $\mathbf{C}_2$ avec deux variables d'entrée x, y et une variable de sortie a est un module satisfaisant ($\mathbf{C}_1 \Uparrow \mathbf{C}_2$) avec deux variables d'entrées x, y de type *integer*, et une variable de sortie a de type *integer*.

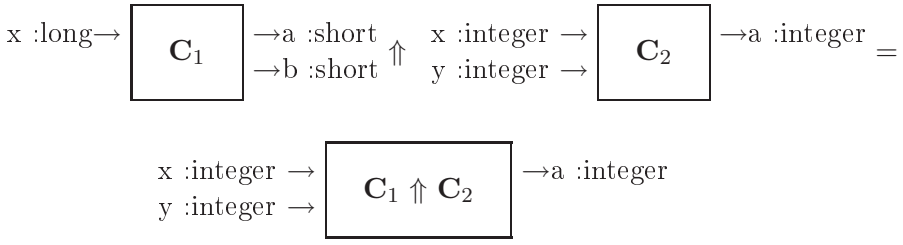


FIG. 4.12 – Borne supérieure de modules.

Ces opérations d'union et d'intersection sur l'ensemble des variables d'entrée/sortie sont respectivement la borne inférieure et la borne supérieure des ensembles de variables d'entrée/sortie. En considérant la relation d'ordre partiel exprimée par les règles (6), (7), et (8) dans la partie 4.2.1, l'ensemble des

ensembles de couples : nom et type de variables d'entrée/sortie, forme une algèbre booléenne, avec pour infimum $\{(x : \textbf{output}\ null) \mid x \in \mathcal{V}\}$ et supremum $\{(x : \textbf{input}\ null) \mid x \in \mathcal{V}\}$. Dans cette algèbre, le complémentaire d'un ensemble I de variables d'entrée/sortie, noté $\neg I$, est défini par :

$$\begin{array}{ll}
\text{Si } \nexists t \in \mathcal{T} \text{ tel que } (x : \textbf{input}\ t) \in I & \text{alors } \neg(I \cup (x : \textbf{input}\ null)) = \neg I \\
\text{Si } \nexists t \in \mathcal{T} \text{ tel que } (x : \textbf{output}\ t) \in I & \text{alors } \neg(I \cup (x : \textbf{output}\ null)) = \neg I \\
\text{Sinon} & \neg(I \cup (x : \textbf{input}\ b)) = \neg I \cup (x : \textbf{input}\ \neg b) \\
 & \neg(I \cup (x : \textbf{output}\ b)) = \neg I \cup (x : \textbf{output}\ \neg b)
\end{array}$$

Cette représentation définit une algèbre booléenne sur les ensembles de variables d'entrée/sortie, et par conséquent une algèbre de Heyting sur ces ensembles.

A partir de la relation de sous-typage, qui met en œuvre et étend la relation de raffinement de l'algèbre de contrats dans l'algèbre de typage des modules, il est facile d'exprimer les relations qui caractérisent la borne inférieure (le plus grand des minorants) et la borne supérieure (le plus petit des majorants) de deux modules, de deux foncteurs, ou encore de deux n-uplets de modules, par induction sur la structure de types. Il est possible d'utiliser ces opérateurs (en particulier la borne inférieure) afin de définir la sémantique d'opérateurs de composition de deux modules (figure 4.13).

$$\begin{array}{ll}
\text{Borne inférieure :} & \\
\text{- de modules :} & \tau(I, C) \sqcap \tau(J, D) = \tau(I \sqcap J, C \Downarrow D) \\
\text{- de produits :} & S \times T \sqcap U \times V = (S \sqcap U) \times (T \sqcap V) \\
\text{- de foncteurs :} & \Lambda(x : S).T \sqcap \Lambda(y : U).V = \Lambda(x : (S \sqcup U)).(T \sqcap V[y/x]) \\
\text{Borne supérieure :} & \\
\text{- de modules :} & \tau(I, C) \sqcup \tau(J, D) = \tau(I \sqcup J, C \Uparrow D) \\
\text{- de produits :} & S \times T \sqcup U \times V = (S \sqcup U) \times (T \sqcup V) \\
\text{- de foncteurs :} & \Lambda(x : S).T \sqcup \Lambda(y : U).V = \Lambda(x : (S \sqcap U)).(T \sqcup V[y/x])
\end{array}$$

FIG. 4.13 – Bornes inférieures et supérieures.

A partir de ces définitions sur l'ensemble des modules, nous observons que pour tous modules S et T, nous avons les relations de raffinement suivantes : $S \sqcap T \leq S, T \leq S \sqcup T$. Ces propriétés sont essentielles pour une approche multi-vues. La borne inférieure de deux modules raffine chacun de ces deux modules. Ceci est la conséquence directe de la structure d'algèbre de Heyting. Ainsi différentes propriétés peuvent être développées séparément puis composées avec l'opération du calcul de la borne inférieure, afin d'obtenir un composant satisfaisant toutes les propriétés désirées.

En représentant les ensembles de variables d'entrée/sortie par une algèbre de Heyting, et compte tenu de la structure d'algèbre de Heyting de l'ensemble de contrats défini au chapitre [2], il en découle sur l'ensemble des modules $\mathcal{M}$ une algèbre de Heyting $(\mathcal{M}, \leq)$. Il en va de même pour l'ensemble des foncteurs, ou encore l'ensemble des $n-$uplets de modules. L'affectation de types et la relation de sous-typage peuvent être vérifiées par un prouveur.

4.2.7 Inférence de types dans le langage de modules

Notre objectif est de vérifier l'exactitude de la construction d'un programme. Par conséquent, le système d'inférence de types est tenu de produire une assignation de types cohérente. Cette assignation est caractérisée par l'association de contrats à des noms de processus dans le langage de modules. Le système d'inférence de types doit également générer une obligation de preuve sous la forme d'une fonction observatrice écrite dans le langage de programmation cible. L'obligation de preuve traduit la satisfaction des contrats par les processus auxquels ils sont associés.

Ce système d'inférence (voir la figure 4.14) est défini par le séquent $\Gamma/\Sigma \vdash exp$ où Γ est l'environnement de typage, Σ l'ensemble des contraintes de typage (et de raffinement de contrats) et exp est une expression écrite dans le langage de modules. Il intègre le système de types du langage cible : nous supposons que la relation $\Gamma \vdash p$ dit que p est bien typé dans le langage cible, sous l'hypothèse Γ.

Le séquent établit une correspondance structurelle entre les expressions et les types. Elle est définie par induction sur la structure des expressions de la même manière que celle proposée pour Standard ML dans [Ler00].

$$(1)\frac{\Gamma(x) = T}{\Gamma/\Sigma \vdash x : T}$$

$$(2)\frac{\Gamma/\Sigma \vdash s : S \quad \Gamma/\Sigma \vdash t : T}{\Gamma/\Sigma \vdash s \times t : S \times T}$$

$$(3)\frac{\Gamma/\Sigma \vdash s : S \quad \Gamma/\Sigma \vdash t : T}{\Gamma/\Sigma \vdash \Lambda(x : s).t : \Lambda(x : S).T}$$

$$(4)\frac{\Gamma/\Sigma \vdash t : T}{\Gamma/\Sigma \vdash t\, x : (x : T)}$$

$$(5)\frac{\Gamma/\Sigma \vdash dec : I \quad \Gamma/\Sigma \vdash dec' : J}{\Gamma/\Sigma \vdash dec, dec' : I \cup J}$$

$$(6)\frac{\Gamma/\Sigma \vdash p \quad \Gamma/\Sigma \vdash q}{\Gamma/\Sigma \vdash \textbf{assume}\, p\, \textbf{guarantee}\, q}$$

$$(7)\frac{\Gamma/\Sigma \vdash ag \quad \Gamma/\Sigma \vdash ag'}{\Gamma/\Sigma \vdash (ag\ \textbf{and}\ ag')}$$

$$(8)\frac{\Gamma/\Sigma \vdash exp : S \quad \Gamma/\Sigma \vdash exp' : T}{\Gamma/\Sigma \vdash (exp\ \textbf{and}\ exp') : S \sqcap T}$$

$$(9)\frac{\Gamma/\Sigma \vdash dec : I \quad \Gamma \cup I \vdash p}{\Gamma/\Sigma \vdash \textbf{process}\, dec; p\, \textbf{end} : \pi(I, ((), p))}$$

$$(10)\frac{\Gamma/\Sigma \vdash dec : I \quad \Gamma \cup I \vdash ag}{\Gamma/\Sigma \vdash \textbf{contract}\, dec; ag\, \textbf{end} : \gamma(I, ag)}$$

$$(11)\frac{\Gamma/\Sigma \vdash dec : I \quad \Gamma \cup I/\Sigma \vdash exp : T}{\Gamma/\Sigma \vdash \textbf{functor}\ (dec)\ exp\ \textbf{end} : \Lambda I.T}$$

$$(12)\frac{\Gamma/\Sigma \vdash x : \Lambda(z : S).T \quad \Gamma/\Sigma \vdash y : U \quad \Sigma \subset U \leq S}{\Gamma/\Sigma \vdash x(y) : T[y/z]}$$

$$(13)\frac{\Gamma/\Sigma \vdash exp : T \quad \gamma \cdot T \leq T}{\Gamma/\Sigma \vdash \textbf{module type}\, x = exp : (x : T)}$$

$$(14)\frac{\Gamma/\Sigma \vdash exp : S \quad \Gamma/\Sigma \vdash t : T \quad \Sigma \subset S \leq T}{\Gamma/\Sigma \vdash \textbf{module}\, x : t = exp : (x : \pi \cdot T)}$$

$$(15)\frac{\Gamma/\Sigma \vdash def : I \quad \Gamma \cup I/\Sigma \vdash def' : J}{\Gamma/\Sigma \vdash def;\ def' : I \cup J}$$

$$(16)\frac{\Gamma/\Sigma \vdash def : I \quad \Gamma \cup I/\Sigma \vdash exp : T}{\Gamma/\Sigma \vdash \mathbf{let}\, def\, \mathbf{in}\, exp : T}$$

FIG. 4.14 – Règles d'inférence de type.

La règle (1) associe une expression syntaxique à un type dans l'environnement Γ.

La règle (2) exprime le typage d'un produit ou d'une paire.

La règle (3) exprime le typage d'une lambda-fonction.

La rèlge (4) exprime le typage d'une expression syntaxique déclarant un paramètre.

La règle (5) exprime le typage d'une succession de déclarations de paramètres.

La règle (6) exprime le typage d'une expression syntaxique exprimant une hypothèse et une garantie.

La règle (7) exprime le typage de la composition de deux expressions syntaxiques définissant chacune un contrat.

La règle (8) exprime le typage de la composition d'expressions syntaxiques par la borne inférieure des types associés à ces expressions.

La règle (9) exprime le typage d'un processus par un couple exprimant le type de ses paramètres, et le contrat garantissant l'exécution du processus dans tous les contextes d'exécution.

La règle (10) exprime l'association d'une expression syntaxique définissant un contrat à un type de module.

La règle (11) exprime le typage d'un foncteur par une lambda-fonction.

La règle (12) exprime la contrainte de typage associée à l'application d'un foncteur x à un paramètre y : le type est la valeur retournée par l'application de la lambda-fonction caractérisant le type du foncteur au type de y.

La règle (13) exprime la contrainte de typage associée à la définition d'un type x de modules.

La règle (14) exprime la contrainte de typage d'un module x associé au type t.

La règle (15) exprime le typage d'une succession de définitions.

La règle (16) exprime le typage d'une définition locale.

L'opérateur $\tau \cdot T$ transforme le type T en type de processus si τ vaut π, ou bien en type contrat, si τ vaut γ. Il est défini par $\tau \cdot (\tau'(I, C)) = \tau \cdot (I, C)$ et $\tau \cdot (\Lambda(x : S).T) = \Lambda(x : S).(\tau \cdot T)$. Cet opérateur est utilisé pour vérifier que la définition du type d'un module est un contrat, et afin de transformer le contrat en type d'implémentation.

4.3 Correction de programme

4.3.1 Sémantique dénotationnelle

La sémantique $[\![exp]\!]_\rho$ d'un terme *exp* dans le système de modules est définie par induction sur la structure de *exp*. Il s'agit d'un ensemble de processus du modèle de contrats satisfaisant leurs spécifications. La fonction ρ associe une expression syntaxique à une valeur dénotationnelle. Un terme p désigne un terme du langage de programmation cible étendu par le langage de modules.

La figure 4.15 explicite l'ensemble des règles de définition de la sémantique dénotationnelle. L'opérateur $\sqcap$ est la transposition de la relation d'intersection entre les types de modules (définie à la sous-section 4.2.5) dans le domaine de la sémantique dénotationnelle. La valeur dénotationnelle d'un processus est la valeur dénotationnelle de son contrat nominal, c'est-à-dire l'ensemble des processus qui satisfont le contrat nominal.

Définition 4.1 (Contrat nominal) *Le contrat nominal* $p_{\succcurlyeq} = (\mathbb{P}^\star, \widehat{[p]})$ *d'un processus garantit tous les comportements du processus* p *dans tous les contextes d'exécution.*

$$[\![x]\!]_\rho = \rho(x) \tag{1}$$

$$[\![dec, dec']\!]_\rho = [\![dec]\!]_\rho \times [\![dec']\!]_\rho \tag{2}$$

$$[\![\textbf{contract}\ dec; ag\ \textbf{end}]\!]_\rho = ([\![dec]\!]_\rho, ([\![ag]\!]_\rho)) \tag{3}$$

$$[\![\textbf{process}\ dec; p\ \textbf{end}]\!]_\rho = ([\![dec]\!]_\rho, ([\![p]\!]_\rho)_\succeq) \tag{4}$$

$$[\![exp\ \textbf{and}\ exp']\!]_\rho = [\![exp]\!]_\rho \sqcap [\![exp']\!]_\rho \tag{5}$$

$$[\![\textbf{functor}\ (dec)\ exp\ \textbf{end}]\!]_\rho = let\ (x : C) = [\![dec]\!]_\rho\ in\ \lambda(x).[\![exp]\!]_{\rho \uplus (x \mapsto C)} \tag{6}$$

$$[\![x(y)]\!]_\rho = ([\![x]\!]_\rho)(y) \tag{7}$$

$$[\![\textbf{module}\,[\textbf{type}]\,x = exp]\!] = (x \mapsto [\![exp]\!]_\rho) \tag{8}$$

$$[\![def;\, def']\!] = [\![def]\!]_\rho \uplus [\![def']\!]_{\rho \uplus [\![def]\!]_\rho} \tag{9}$$

$$[\![\textbf{let}\, def\, \textbf{in}\, exp]\!] = [\![exp]\!]_{\rho \uplus [\![def]\!]_\rho} \tag{10}$$

FIG. 4.15 – Sémantique dénotationnelle

La règle (1) associe une expression syntaxique à une valeur dénotationnelle.
La règle (2) exprime la valeur dénotationnelle d'une succession de déclarations par le produit des valeurs dénotationnelles de ces déclarations.
La règle (3) exprime la valeur dénotationnelle d'un contrat.
La règle (4) exprime la valeur dénotationnelle d'un processus par son contrat nominal.
La règle (5) exprime la valeur dénotationnelle de la composition de deux modules par la borne inférieure des valeurs dénotationnelles de ces deux expressions.
La règle (6) exprime la valeur dénotationnelle d'un contrat par une fonction prenant en argument la valeur dénotationnelle des arguments du foncteur.
La règle (7) exprime la valeur dénotionnelle de l'application d'un foncteur.
La règle (8) exprime la valeur dénotationnelle de la définition d'un module ou d'un type de module.
La règle (9) exprime la valeur dénotationnelle d'une succession de définitions.
La règle (10) exprime la valeur dénotationnelle d'une définition locale.

4.3.2 Théorème de correction

La correction d'un système de modules repose sur le fait qu'un programme est bien typé si les contraintes impliquées par Γ/Σ sont consistantes. Nous disons que l'environnement ρ est bien typé avec Γ/Σ, noté $\rho : \Gamma/\Sigma$, si et seulement si toutes les définitions de Γ sont consistantes sous les contraintes Σ. Notons que dans une implémentation de ce système de types, les contraintes générées dans Σ définissent des obligations de preuve utiles pour vérifier qu'une spécification *exp* est bien typée. Dans le but d'établir un théorème de correction, la sémantique $[\![exp]\!]_\rho$ d'un terme *exp* dans le système de modules est définie par induction sur la structure de *exp* (voir la section 4.3.1). Il s'agit d'un ensemble de processus du modèle de contrats qui satisfont leur spécification.

Théorème 4.1 *Si* $\rho : \Gamma/\Sigma$ *pour un ensemble de contraintes satisfaisables* Σ *et* $\Gamma/\Sigma \vdash exp : T$ *alors* $[\![exp]\!]_\rho \subseteq [\![T]\!]_\rho$.

Intuition de la preuve. La preuve du théorème de correction est définie par induction sur la structure des expressions (exp). Pour chaque catégorie syntaxique dans la grammaire de exp, l'ensemble de contraintes Σ est considéré satisfaisable, $\rho : \Gamma/\Sigma$ est un environnement bien typé, et $\Gamma/\Sigma \vdash exp : T$. Les hypothèses d'induction portent sur le fait que les sous-expressions $exp_{i\in 1..n}$ de exp satisfont les sous-propriétés $[\![exp_i]\!]_\rho \subseteq [\![T_i]\!]_\rho$ attendues pour la satisfaction du théorème afin d'en déduire que $[\![exp]\!]_\rho \subseteq [\![T]\!]_\rho$ par définition de la sémantique dénotationnelle.

4.4 Conclusion

A partir du modèle de contrats présenté dans le chapitre 2, nous avons introduit un système de modules basé sur le paradigme des contrats. La structure de l'ensemble des contrats étant une algèbre de Heyting, il en découle une algèbre de Heyting sur l'ensemble des modules permettant ainsi de raisonner sur les modules avec suffisamment de flexibilité au niveau du raffinement, de la composition ainsi que de l'abstraction.

Le paradigme que nous proposons est de considérer les contrats comme étant le type de comportement d'un composant, et de l'utiliser pour l'élaboration de l'architecture fonctionnelle d'un système avec une obligation de preuve qui valide l'exactitude des hypothèses et des garanties lors de la construction de cette architecture. Ce modèle peut être très simplement utilisé pour vérifier des systèmes synchrones multi-horloges spécifiés en SIGNAL.

Chapitre 5

Application au langage SIGNAL

Dans cette partie, nous présentons une application du modèle de contrats et du langage de modules à SIGNAL. Le langage de modules est utilisé dans le but de spécifier les générateurs des contrats et l'implémentation des processus devant satisfaire ces contrats. Dans cette application, les générateurs des filtres ainsi que les processus sont écrits en SIGNAL. La proximité des définitions des comportements et des processus dans le modèle de contrats et dans SIGNALrend l'utilisation de ce modèle de contrats très simple.

5.1 Un modèle de contrats pour SIGNAL

Dans le modèle multi-horloges de SIGNAL [3], un processus p est défini par la composition synchrone d'équations sur des signaux (notées $x = y\ f\ z$). Un signal x est un flot infini de valeurs présentes à des instants symboliques, selon le rythme de l'horloge de x. Un ensemble de marques t désigne une période symbolique du temps durant laquelle un événement se produit. Les marques permettent d'échantillonner les valeurs d'un signal sur une série d'instants dénombrables. Les événements, les signaux, les comportements et les processus sont définis de la manière suivante :

- un *événement* e est une paire associant une *marque* t et une valeur v,
- un *signal* est une fonction d'une chaîne de marques vers un ensemble de valeurs,
- un comportement b est une fonction associant un nom de variable à un signal,
- un processus $p \in \mathbb{P}$ est un ensemble de comportements définis sur le même ensemble de variables.

Nous souhaitons maintenant utiliser le modèle de contrats développé dans le chapitre 2 pour la spécification de contrats en SIGNAL et la vérification de processus écrits en SIGNAL. Par conséquent, nous donnons une nouvelle définition

des comportements dans ce modèle spécifiant que le domaine de définition $\mathcal{D}$ des variables est l'ensemble des signaux $\mathcal{S}$.

Définition 5.1 (Comportement SIGNAL dans le modèle de contrats) *Soit $\mathcal{V}$ un ensemble infini, dénombrable de variables (ce qui est équivalent à un ensemble infini, dénombrable de noms de signaux), et $\mathcal{S}$ un ensemble de signaux (qui est un ensemble particulier de valeurs défini à la section 3.2.1 du chapitre 3) ; pour $\mathcal{X}$ un ensemble fini, et non vide, de variables inclus dans $\mathcal{V}$, un comportement défini sur un ensemble de variables $\mathcal{X}$ est une fonction $b : \mathcal{X} \to \mathcal{S}$.*

En considérant le modèle de contrats avec cette nouvelle définition des comportements, l'opération de restriction des comportements SIGNAL dans le modèle de contrats est équivalente à l'opération de projection dans la sémantique du langage SIGNAL. Il en va de même pour la définition des processus : un processus est un ensemble de comportements défini sur un même ensemble de signaux. Ainsi, les filtres ont pour processus générateurs des processus SIGNAL. Toutes les propriétés développées sur l'algèbre de contrats sont valides lors de l'application du modèle à SIGNAL.

Dans le contexte du développement basé sur la notion de composants ou sur le concept de contrats, la substituabilité et le raffinement de contrats sont des concepts fondamentaux indispensables [DHJP08]. Le raffinement permet de remplacer un composant par une version plus aboutie de celui-ci. La notion de substituabilité permet d'implémenter un contrat (ou un sous-système) indépendamment de son contexte d'utilisation. Ces propriétés sont essentielles pour considérer une implémentation comme un succession de raffinements, jusqu'à l'implémentation finale. Comme il est mentionné dans [RBB+09], d'autres aspects peuvent être considérés dans la méthodologie de développement. En particulier, les concepts de multi-vues pour un composant donné, et de partage d'une implémentation pour plusieurs spécifications.

En considérant la composition synchrone de processus SIGNAL et la borne inférieure des contrats comme opération de composition, nous avons les propriétés suivantes :

Propriété 5.1 *Soit quatre processus $p,q \in \mathbb{P}$, et deux contrats $\mathbf{C}_1$, $\mathbf{C}_2$, $\mathbf{C'}_1$, $\mathbf{C'}_2$ $\in \mathbb{C}$.*

$$\mathbf{C}_1 \preccurlyeq \mathbf{C}_2 \implies ((p \vDash \mathbf{C}_1) \Longrightarrow (p \vDash \mathbf{C}_2)) \tag{5.1}$$

$$\mathbf{C}_1 \rightsquigarrow \mathbf{C}_2 \iff ((p \vDash \mathbf{C}_1) \Longrightarrow (p \vDash \mathbf{C}_2)) \tag{5.2}$$

$$\mathbf{C'}_1 \preccurlyeq \mathbf{C}_1 \wedge \mathbf{C'}_2 \preccurlyeq \mathbf{C}_2 \implies \mathbf{C'}_1 \Downarrow \mathbf{C'}_2 \preccurlyeq \mathbf{C}_1 \Downarrow \mathbf{C}_2 \tag{5.3}$$

$$((p \vDash \mathbf{C}_1) \wedge (q \vDash \mathbf{C}_2)) \implies (p \,|\, q \vDash (\mathbf{C}_1 \Downarrow \mathbf{C}_2)) \tag{5.4}$$

$$((p \vDash \mathbf{C}_1) \wedge (p \vDash \mathbf{C}_2)) \iff (p \vDash (\mathbf{C}_1 \Downarrow \mathbf{C}_2)) \tag{5.5}$$

- (5.1) et (5.2) montrent la substituabilité engendrée par le raffinement. Plus particulièrement, par la relation (5.1), si un contrat $\mathbf{C}_1$ raffine un contrat $\mathbf{C}_2$ alors tout processus p satisfaisant $\mathbf{C}_1$ satisfait aussi $\mathbf{C}_2$. Par conséquent, l'ensemble des processus admis par $\mathbf{C}_1$ étant inclus dans l'ensemble des processus admis par $\mathbf{C}_2$, tout composant satisfaisant $\mathbf{C}_2$ peut être remplacé par un composant satisfaisant $\mathbf{C}_1$. Avec la relation (5.2), un contrat $\mathbf{C}_1$ est plus fin qu'un contrat $\mathbf{C}_2$ si et seulement si les processus satisfaisant $\mathbf{C}_1$ satisfont aussi $\mathbf{C}_2$.
- (5.3) et (5.4) autorisent la substituabilité dans la composition. Plus particulièrement, par la relation (5.3), si un contrat $\mathbf{C'}_1$ raffine un contrat $\mathbf{C}_1$, et si un contrat $\mathbf{C'}_2$ raffine un contrat $\mathbf{C'}_2$ alors la borne inférieure de $\mathbf{C'}_1$ et $\mathbf{C'}_2$ raffine la borne inférieure de $\mathbf{C}_1$ et $\mathbf{C}_2$. Avec la relation (5.4), si un processus SIGNAL p satisfait un contrat $\mathbf{C}_1$, et si un processus SIGNAL q satisfait un contrat $\mathbf{C}_2$ alors la composition synchrone de p et q satisfait la borne inférieure des contrats $\mathbf{C}_1$ et $\mathbf{C}_2$. Ainsi un sous-système peut être développé de manière isolée. Une fois mis au point de façon indépendante, les sous-systèmes doivent se substituer à leur spécification et se composer comme prévu.
- (5.5) traduit la notion de vues multiples sur un même composant : un processus p satisfait un contrat $\mathbf{C}_1$ et un contrat $\mathbf{C}_2$ si et seulement si p satisfait la borne inférieure des contrats $\mathbf{C}_1$ et $\mathbf{C}_2$. Cette propriété apporte une solution au besoin de modularité posé par le développement simultané de systèmes par des équipes différentes utilisant différents outils. Un exemple est le développement simultané des propriétés de sécurité ou de fiabilité ainsi que des aspects fonctionnels. D'autre part, la nécessité d'une bonne gestion des accès concurrents à la mémoire par des sous-systèmes est un autre aspect. Enfin, l'énergie est aujourd'hui de plus en plus considérée comme une ressource critique, ce qui conduit à l'examen d'un autre aspect de la conception du système. Chacun de ces aspects nécessite des cadres et des outils pour leur analyse et leur conception. Pourtant, ils ne sont pas totalement indépendants. La possibilité de développer de multiples aspects, ou plusieurs points de vue de façon simultanée est donc essentielle.

Preuve : équation 5.1 :
Conséquence directe de la définition 2.16. □

Preuve : équation 5.2 :
Conséquence directe de la définition 2.14. □

Preuve : équation 5.3 :
Soit deux contrats $\mathbf{C}_1 = (\mathbf{A}_1,\mathbf{G}_1)$, et $\mathbf{C'}_1 = (\mathbf{A'}_1,\mathbf{G'}_1)$, tels que $\mathbf{C'}_1 \preccurlyeq \mathbf{C}_1$.

Soit deux contrats $\mathbf{C}_2 = (\mathbf{A}_2,\mathbf{G}_2)$, et $\mathbf{C'}_2 = (\mathbf{A'}_2,\mathbf{G'}_2)$, tels que $\mathbf{C'}_2 \preccurlyeq \mathbf{C}_2$.

Rappelons que :

$$\mathbf{C'}_1 = (\mathbf{A'}_1, \mathbf{G'}_1) \preccurlyeq \mathbf{C}_1 = (\mathbf{A}_1, \mathbf{G}_1) \iff \begin{cases} \mathbf{A}_1 \sqsubseteq \mathbf{A'}_1 \\ (\mathbf{A}_1 \sqcap \mathbf{G'}_1) \sqsubseteq \mathbf{G}_1 \\ \mathbf{G'}_1 \sqsubseteq \mathbf{A'}_1 \sqcup \mathbf{G}_1 \end{cases}$$

et

$$\mathbf{C'}_2 = (\mathbf{A'}_2, \mathbf{G'}_2) \preccurlyeq \mathbf{C}_2 = (\mathbf{A}_2, \mathbf{G}_2) \iff \begin{cases} \mathbf{A}_2 \sqsubseteq \mathbf{A'}_2 \\ (\mathbf{A}_2 \sqcap \mathbf{G'}_2) \sqsubseteq \mathbf{G}_2 \\ \mathbf{G'}_2 \sqsubseteq \mathbf{A'}_2 \sqcup \mathbf{G}_2 \end{cases}$$

Ce qui implique :

$\mathbf{A}_1 \sqsubseteq \mathbf{A}_{1'}$
$\mathbf{A}_2 \sqsubseteq \mathbf{A}_{2'}$
$(\mathbf{A}_1 \sqcap \mathbf{G'}_1 \sqcap \mathbf{A}_2 \sqcap \mathbf{G'}_2) \sqsubseteq (\mathbf{G}_1 \sqcap \mathbf{G}_2)$
$(\mathbf{G'}_1 \sqcap \mathbf{G'}_2) \sqsubseteq \mathbf{A'}_1 \sqcup \mathbf{A'}_2 \sqcup \mathbf{G}_1 \sqcap \mathbf{G}_2$

$\Longrightarrow$

(par les propriétés de l'algèbre de filtres)

$\mathbf{A}_1 \sqsubseteq \mathbf{A}_{1'}$
$\mathbf{A}_2 \sqsubseteq \mathbf{A}_{2'}$
$(\mathbf{A}_1 \sqcup \mathbf{A}_2) \sqsubseteq (\mathbf{A'}_1 \sqcup \mathbf{A'}_2)$
$((\mathbf{A}_1 \sqcap \mathbf{G'}_1 \sqcap \mathbf{G'}_2) \sqcup (\mathbf{G}_1 \sqcap \mathbf{A}_1 \sqcap \widetilde{\mathbf{A'}_2}) \sqcup (\mathbf{A}_2 \sqcap \mathbf{G'}_1 \sqcap \mathbf{G'}_2) \sqcup (\mathbf{G'}_2 \sqcap \mathbf{A}_2 \sqcap \widetilde{\mathbf{A'}_1}))$
$\quad \sqsubseteq ((\mathbf{G}_1 \sqcap \mathbf{G}_2) \sqcup (\mathbf{G}_1 \sqcap \mathbf{A}_1 \sqcap \widetilde{\mathbf{A}_2}) \sqcup (\mathbf{G}_2 \sqcap \mathbf{A}_2 \sqcap \widetilde{\mathbf{A}_1}))$
$((\mathbf{G'}_1 \sqcap \mathbf{G'}_2) \sqcup (\mathbf{G'}_1 \sqcap \mathbf{A'}_1 \sqcap \widetilde{\mathbf{A'}_2}) \sqcup (\mathbf{G'}_2 \sqcap \mathbf{A'}_2 \sqcap \widetilde{\mathbf{A'}_1}))$
$\quad \sqsubseteq (\mathbf{A'}_1 \sqcup \mathbf{A'}_2 \sqcup (\mathbf{G}_1 \sqcap \mathbf{G}_2) \sqcup (\mathbf{G}_1 \sqcap \mathbf{A}_1 \sqcap \widetilde{\mathbf{A}_2}) \sqcup (\mathbf{G}_2 \sqcap \mathbf{A}_2 \sqcap \widetilde{\mathbf{A}_1}))$
$\Longrightarrow$

(par les propriétés de l'algèbre de filtres)

$(\mathbf{A}_1 \sqcup \mathbf{A}_2) \sqsubseteq (\mathbf{A'}_1 \sqcup \mathbf{A'}_2)$
$((\mathbf{A}_1 \sqcup \mathbf{A}_2) \sqcap ((\mathbf{G'}_1 \sqcap \mathbf{G'}_2) \sqcup (\mathbf{G'}_1 \sqcap \mathbf{A'}_1 \sqcap \widetilde{\mathbf{A'}_2}) \sqcup (\mathbf{G'}_2 \sqcap \mathbf{A'}_2 \sqcap \widetilde{\mathbf{A'}_1})))$
$\quad \sqsubseteq ((\mathbf{G}_1 \sqcap \mathbf{G}_2) \sqcup (\mathbf{G}_1 \sqcap \mathbf{A}_1 \sqcap \widetilde{\mathbf{A}_2}) \sqcup (\mathbf{G}_2 \sqcap \mathbf{A}_2 \sqcap \widetilde{\mathbf{A}_1}))$
$((\mathbf{G'}_1 \sqcap \mathbf{G'}_2) \sqcup (\mathbf{G'}_1 \sqcap \mathbf{A'}_1 \sqcap \widetilde{\mathbf{A'}_2}) \sqcup (\mathbf{G'}_2 \sqcap \mathbf{A'}_2 \sqcap \widetilde{\mathbf{A'}_1}))$
$\quad \sqsubseteq ((\mathbf{A'}_1 \sqcup \mathbf{A'}_2) \sqcup ((\mathbf{G}_1 \sqcap \mathbf{G}_2) \sqcup (\mathbf{G}_1 \sqcap \mathbf{A}_1 \sqcap \widetilde{\mathbf{A}_2}) \sqcup (\mathbf{G}_2 \sqcap \mathbf{A}_2 \sqcap \widetilde{\mathbf{A}_1})))$
$\Longrightarrow$

(par les propriétés de l'algèbre de filtres)

$(\mathbf{C'}_1 \Downarrow \mathbf{C'}_2) \preccurlyeq (\mathbf{C}_1 \Downarrow \mathbf{C}_2)$

□

Preuve : équation 5.4 :

Soient p et q deux processus SIGNAL. Par la propriété propriété 3.1 :

$$((p \,|\, q)_{|vars(p)} \subseteq p) \tag{5.6}$$

Or, $(vars(p) \subseteq vars(p) \cup vars(p))$, et $vars(p \,|\, q) = vars(p) \cup vars(p)$, alors,
$\Longrightarrow ((\,(p \,|\, q)_{|vars(p)})^{|vars(p) \cup vars(q)} \subseteq p^{|\mathrm{vars}(p) \cup \mathrm{vars(q)}})$(équation 2.2 et équation 2.3)
$\Longrightarrow ((p \,|\, q)^{|vars(p) \cup vars(q)} \subseteq p^{|\mathrm{vars}p \cup vars(\mathrm{q})})$
Or, $(vars(\overset{\triangledown}{p} \,|\, q) \subseteq vars(p \,|\, q))$, alors, $(\overset{\triangledown}{(p \,|\, q)}^{|vars(p) \cup vars(p)} \subseteq \overset{\triangledown}{p}^{|vars(p) \cup vars(p)})$.
Donc, $\widehat{[p \,|\, q]} \sqsubseteq \widehat{[p]}$ (définition 2.9).

Si $(p \vDash \mathbf{C}_1)$ alors $(p \,|\, q \vDash \mathbf{C}_1)$, nous en déduisons :

$$(((p \vDash \mathbf{C}_1) \wedge (q \vDash \mathbf{C}_2)) \implies ((p \,|\, q \vDash \mathbf{C}_1) \wedge (p \,|\, q \vDash \mathbf{C}_2)))$$

En utilisant la définition de la borne inférieure de deux contrats (lemme 2.5), on obtient :

$$(((p \,|\, q \vDash \mathbf{C}_1) \wedge (p \,|\, q \vDash \mathbf{C}_2)) \implies (p \,|\, q \vDash (\mathbf{C}_1 \Downarrow \mathbf{C}_2)))$$

□

Preuve : équation 5.5 : Conséquence de la définition de la borne inférieure de deux contrats (voir le lemme 2.5). □

5.2 Extension du langage SIGNAL

Nous définissons la syntaxe formelle de notre langage de modules pour son application à SIGNAL. Dans la grammaire présentée sur la figure 5.1, les programmes notés p ou q sont écrits dans le langage de programmation SIGNAL. Les implémentations ainsi que les processus générateurs des contrats sont écrits dans le même langage SIGNAL.

De la même manière que dans la section 4.1, les noms sont notés x ou y. Les types t sont utilisés pour déclarer les types des paramètres et les variables d'entrée/sortie dans les interfaces. Les hypothèses et les garanties sont décrites par des expressions p et q dans un langage cible ou un langage spécifique au contexte d'utilisation du langage de modules. Dans le cas d'application à SIGNAL, le langage cible est SIGNAL. Une expression *exp* manipule des contrats, des modules, des foncteurs, et des instances de foncteurs ou encore des références.

x, y		noms
p, q		processus spécifiés en SIGNAL
b, c	$::=$ **event** $\mid$ **boolean** $\mid$ **short** $\mid$ **integer** $\mid \ldots$	types de données
t	$::= b \mid$ **input** $b \mid$ **output** $b \mid x \mid t \times t$	types
dec	$::= t\ x\ [, dec]$	déclaration
ag	$::=$ [**assume** p] **guarantee** $q; \mid ag$ **and** ag	contrat
exp	$::=$ **contract** $dec;\ ag$ **end**	contrat
	$\mid$ **process** $dec;\ p$ **end**	processus
	$\mid$ **functor** $(dec)\ exp$	foncteur
	$\mid$ exp **and** exp	composition
	$\mid$ $x\ (exp^*)$	application
	$\mid$ **let** def **in** exp	définition locale
def	$::=$ **module** [**type**] $x\ =\ exp$	définition d'une spécification
	$\mid$ **module** $x\ [: t]\ =\ exp$	définition d'un module
	$\mid$ $def; def$	succession de définitions

FIG. 5.1 – Grammaire du langage de modules appliqué à SIGNAL

5.3 Un cas d'utilisation

Nous illustrons les différents aspects de notre algèbre de contrats en reconsidérant la spécification du moteur à quatre temps et la traduction en processus observateurs dans le langage de notre choix : le langage de programmation multi-horloges SIGNAL.

La figure 5.2 représente la machine à états représentant les quatre phases successives du moteur à 4 temps : admission (*Intake*), compression (*Compression*), combustion (*Combustion*), et échappement (*Exhaust*). Ces différentes phases sont associées à la position de l'arbre à cames mesurée en degrés.

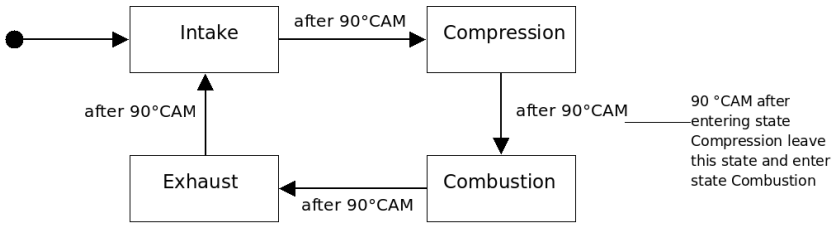


FIG. 5.2 – Cycle d'un moteur à 4 temps

La position angulaire de l'arbre à cames définit un "temps" de référence dis-

cret. L'horloge du signal *cam* représente la position de l'arbre à cames mesurée en degrés CAM°, et initialisée à 0. Les transitions de la machine à états sont dirigées par la position angulaire de l'arbre à cames. Les variables *cam*, *Intake*, *Compression*, *Combustion*, *Exhaust* modélisent le comportement du moteur. Nous souhaitons définir un contrat stipulant que la phase d'admission (*Intake*) est effective lorsque la position de l'arbre à cames se situe dans le premier quart de tour de sa révolution. Pour cela, nous définissons le processus générateur d'un filtre pour modéliser les hypothèses. Ce processus générateur mesure la valeur de la variable d'environnement *cam*. La valeur de *cam* doit être comprise entre 0 et 90 degrés. En considérant cette hypothèse, la machine à états doit garantir que le moteur à 4 temps est dans la phase d'admission (*Intake*). Les processus générateurs des filtres des hypothèses et des garanties sont :

$$\mathbf{A}_{Intake}{=}_{\Delta}(cam\ mod\ 360° < 90) \quad \mathbf{G}_{Intake}{=}_{\Delta} Intake$$

Un aspect bénéfique de notre algèbre est que la séparation des hypothèses réalisées sur l'environnement et des garanties du système est facilitée par la possibilité d'exprimer naturellement le complémentaire d'un filtre et plus particulièrement du générateur du filtre. Ici, le complémentaire de $\mathbf{A}_{Intake}$ est simplement défini par :

$$\widetilde{\mathbf{A}_{Intake}}{=}_{\Delta}(cam\ mod\ 360° \geq 90)$$

En revanche, il semble beaucoup plus difficile d'écrire le complémentaire d'un automate.

La structure générique des processus trouve une instance directe dans le modèle de calcul multi-horloges de SIGNAL. Spécifions les générateurs de $\mathbf{A}_{Intake}$ et $\mathbf{G}_{Intake}$ au moyen d'équations SIGNAL :

$$\mathbf{A}_{Intake} = true\ when\ (cam\ modulo\ 360 < 90)$$
$$\mathbf{G}_{Intake} = true\ when\ intake\ default\ false$$

Une subtilité du langage SIGNAL est que le contrat ne traite pas uniquement des valeurs *vrai* ou *faux*, des signaux, mais également des statuts des signaux, *présent* ou *absent*. Par conséquent, le complémentaire de l'hypothèse $\mathbf{A}_{Intake}$ est simplement défini par $\widetilde{\mathbf{A}_{Intake}} = false\ when\ \mathbf{A}_{Intake}\ default\ true$ signifiant que $\widetilde{\mathbf{A}_{Intake}}$ est satisfait si et seulement si *cam* est absent, ou bien *cam* est présent avec une valeur supérieure à 90. Notons que, pour une trace de $\mathbf{A}_{Intake}$ donnée, l'ensemble des traces possibles correspondantes est infini (et dense), puisque les traces de $\widetilde{\mathbf{A}_{Intake}}$ ne sont pas nécessairement associées à la même horloge que $\mathbf{A}_{Intake}$.

Exemple 10 *Considérons le comportement suivant comme étant le comportement admis par le processus générateur de* $\mathbf{A}_{Intake}$ *:*

$$\mathbf{A}_{Intake} = 1\_\ 1\_\ 1\_\ 1\_\ 1\_\ 1\_\ 1\_\ 1\_\ 1\_$$

Ainsi $\widetilde{\mathbf{A}_{Intake}}$ *acceptera le comportement :*

$$\widetilde{\mathbf{A}_{Intake}} = 0\_\ \_\ \_\ 0\_\ \_\ \_\ 0\_\ \_\ \_\ 0\_\ \_\ \_\ 0\_$$

Ou encore :

$$\widetilde{\mathbf{A}_{Intake}} = 0111011\_\ 01\_\ 10\_\ \_\ 101\ldots$$

Il est également intéressant de noter que l'horloge de $\widetilde{\mathbf{A}_{Intake}}$ (sa référence dans le temps) n'est pas explicitement exprimée en fonction de $\mathbf{A}_{Intake}$ ou $\mathbf{G}_{Intake}$: elle est implicitement et partiellement liée à l'horloge de *cam*.

Notons que notre modèle de contrats est "agnostique" vis-à-vis d'un modèle de calcul particulier et autorise un domaine de définition générique des comportements. Si nous avions considéré des spécifications exécutables, telles que des observateurs synchrones [MM04], il aurait été nécessaire de connaître l'horloge de l'environnement dans le but de déterminer l'horloge du complémentaire. Ceci n'est pas nécessaire avec l'exemple d'application du modèle de contrats à SIGNAL. La structure booléenne de notre algèbre supporte la capacité de raffiner de manière compositionnelle (sans altérer ou abstraire les propriétés individuelles).

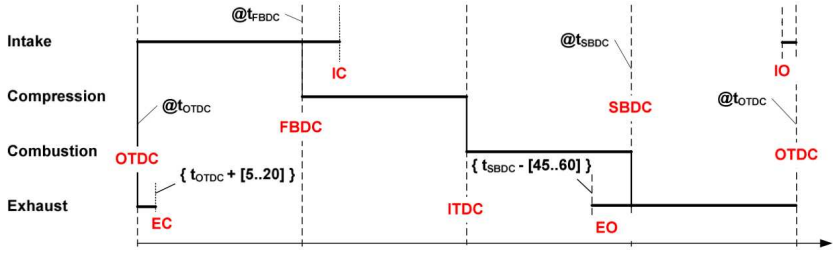


FIG. 5.3 – Modèle d'un moteur à quatre temps.

Par exemple, considérons un modèle plus précis du moteur à 4 temps présenté dans [AMPF07a] (Figure 5.3). Ce modèle requiert que le moteur doit atteindre l'état EC (Exhaust closes) entre 5 et 20 degrés, tant qu'il est dans la phase d'admission (*Intake*). La version raffinée prenant en compte cette propriété du moteur est simplement la composition (la borne inférieure sur les contrats) du contrat initial avec les nouveaux filtres :

$$A_{EC} = true\ when (4 < cam\ modulo\ 360 < 21)$$
$$G_{EC} = true\ when\ EC\ default\ false$$

Rappelons le foncteur de spécification de module `phase` défini dans la sous-section 4.1.2 définissant l'expression générique d'un contrat qui associe une phase à une position de l'arbre à cames :

```
module type phase =
 functor(integer min, integer max)
   contract
     input integer cam ;
     output event trigger;
     assume min<=(cam mod 360)<max
     guarantee trigger
   end;
```

De manière similaire à l'événement *Intake*, $OTDC$ (Overlap Top Dead Center) se produit au début du cycle. L'instant t_{OTDC} est l'instant où cet événement est observé, et l'occurrence de l'événement EC est contraint par $\{t_{OTDC}+[5..20]\}$, par conséquent $t_{OTDC}+5 \leq t_{EC} \leq t_{OTDC}+20$. Nous allons affiner la spécification du moteur en incorporant ces nouvelles contraintes. Ce qui nous donne :

```
module type better_engine = engine
  and contract
        input integer CAM ;
        output event EC, IC, EO, IO;
            phase(5,20)(CAM,EC)
        and phase(130,150)(CAM,IC)
        and phase(210,225)(CAM,EO)
        and phase(344,350)(CAM,IO)
  end;
```

Il est inutile de dire qu'un prouveur sophistiqué, basé par exemple sur l'arithmétique de Pressburger, nous aiderait à vérifier la compatibilité de la cartographie du moteur présentée précédemment. Néanmoins, dans le but d'effectuer une simulation, une implémentation de la spécification du moteur peut être obtenue simplement. Il est également possible de définir un processus observateur pouvant être utilisé comme une obligation de preuve devant être satisfaite par une implémentation de la cartographie du moteur. Ce langage de modules peut également être un moyen de synthétiser un contrôleur afin d'imposer à la mise en œuvre du modèle, la satisfaction de la propriété spécifiée.

```
process phase =
 {integer min, max;}
 (? integer cam; ! event trigger;)
 (| cam ^= trigger
  | trigger := when min<= (cam mod 360)< max
  |);
```

FIG. 5.4 – Processus SIGNAL spécifiant une phase générique de la cartographie du moteur.

La figure 5.4 présente une implémentation en SIGNAL d'une phase générique de la cartographie du moteur. Le processus `phase` se compose de l'équation qui est exécutée lorsque sa sortie `trigger` est nécessaire (lorsque l'horloge de `trigger` est "active"). Ce processus est paramétré par `min` et `max` désignant chacun une position angulaire de l'arbre à cames. Si le signal `cam` est présent, avec une valeur comprise dans [`min` ; `max`], alors `trigger` est présent. Sinon, `cam` est absent, ou bien hors des valeurs de l'intervalle [`min` ; `max`], dans ce cas, `trigger` est absent. Les signaux `cam` et `trigger` représentent respectivement l'entrée et la sortie du processus où `min` et `max` désignent les paramètres du foncteur permettant l'instanciation du processus.

```
process engine =
 (? integer CAM; ! event OTD, FBD, ITD, SBD;)
 (| OTD := phase {  0,  90} (CAM)
  | FBD := phase { 90, 180} (CAM)
  | ITD := phase {180, 270} (CAM)
  | SBD := phase {270, 360} (CAM)
  |);
```

FIG. 5.5 – Implémentation du moteur à quatre temps

Dans le processus `engine` (voir la figure 5.5), les quatre instances des équations de `phase` spécifient les signaux de sortie qui sont associés à une phase du moteur. Notons *qu'a priori*, ces signaux ne sont pas synchronisés : ils sont concurrents. Ceci est fait pour favoriser l'aspect compositionnel de la spécification du système. Le raffinement itératif permettra de construire une spécification exécutable séquentiellement, par exemple, en prenant en compte la synchronisation des signaux `OTD`, `FBD`, `ITD` et `SBD`.

Ce choix est en faveur du modèle de composition (polychrone), au détriment de l'exécutabilité (synchronie), permettant de traiter les spécifications additionnelles du moteur de manière compositionnelle, ce qui montre que le langage SIGNAL et

notre système de modules partagent la même philosophie de conception concurrente et compositionnelle. La figure 5.6 présente une implémentation prenant en considération les propriétés portants sur les signaux `OTD`, `FBD`, `ITD` et `SBD` raffinant la spécification initiale du moteur. Cette implémentation est obtenue par composition des équations relatives aux signaux `OTD`, `FBD`, `ITD`, et `SBD` avec la spécification initiale `engine`.

```
process betterengine =
 (? integer CAM; ! event OTD, FBD, ITD, SBD,
                        EC, IC, EO, IO;)
 (| (OTD, FBD, ITD, SBD) := engine (CAM)
  | EC := phase {  5,  20} (CAM)
  | IC := phase {130, 150} (CAM)
  | EO := phase {210, 225} (CAM)
  | IO := phase {344, 350} (CAM)
  |);
```

FIG. 5.6 – Version raffinée de l'implémentation du moteur à quatre temps.

Les contrats peuvent être utilisés pour exprimer des propriétés d'exclusion. Par exemple, lorsque le moteur est dans le mode d'admission (*Intake*), il ne faut pas commencer la phase de compression (*Compression*). Ce qui se traduit par le contrat : $\mathbf{A}_{excl}=_{\Delta}OTDC$ et $\mathbf{G}_{excl}=_{\Delta}\neg FBDC$. La figure 5.7 présente l'écriture de cette propriété d'exclusion écrite dans le langage de modules appliqué à SIGNAL.

```
module type exclude =
  contract
    output event intake, compression;
    assume intake
    guarantee (not compression default intake)
end;
```

FIG. 5.7 – Une propriété d'exclusion.

En plus des propriétés de sûreté, les contrats peuvent également être utilisés pour exprimer des propriétés de vivacité. Par exemple, considérons le protocole de mise en marche du moteur. Un démarreur est utilisé pour lancer sa rotation. Lorsque le moteur est démarré, le démarreur peut être désactivé.

$$\mathbf{A}_{count} =_{\Delta} Exhaust \quad \mathbf{G}_{count} =_{\Delta} cycle' = cycle + 1$$
$$\mathbf{A}_{live} =_{\Delta} (cycle > 0) \quad \mathbf{G}_{live} =_{\Delta} F(\neg starter)$$

FIG. 5.8 – Une propriété de vivacité.

Sur la figure 5.8, nous présentons le contrat ($\mathbf{A}_{count}$,$\mathbf{G}_{count}$) garantissant que les cycles du moteur sont correctement comptabilisés. Le contrat ($\mathbf{A}_{live}$,$\mathbf{G}_{live}$) garantit que l'utilisation du démarreur (*starter*) s'achèvera au bout de quelques cycles. Nous écrivons *cycle*$'$ pour désigner la prochaine valeur de la variable du *cycle*, et l'opérateur $F()$, de la logique LTL, sera utilisé pour désigner le futur d'une propriété.

Nous utilisons les équations présentées à la figure 5.8 pour écrire ces propriétés dans le langage de modules :

```
module type counter =
  functor (output event trigger,
           integer count)
    contract
      input  event trigger;
      output integer count;
      assume trigger
      guarantee count = (count$1 init 0) + 1
    end;
```

FIG. 5.9 – Comptage des cycles

Sur la figure 5.9, l'expression `count$1 init 0` est initialisée à 0. Les autres valeurs, qui suivront l'état initial, sont les valeurs précédentes de `count`. Les signaux `count$1 init 0` et `count` sont présents en même temps, il s'agit de deux signaux synchrones dans la sémantique de SIGNAL. La figure 5.10 présente l'écriture de cette propriété de survie écrite dans le langage de modules appliqué à SIGNAL. Nous écrirons `eventually` pour désigner l'état futur d'une propriété LTL.

```
module type starter =
  counter (intake, count)
  and contract
    output boolean starterOff;
        assume count > 0
        guarantee eventually starterOff
      end;
```

FIG. 5.10 – Le démarreur est éventuellement coupé.

5.4 Implémentation

Dans la section 4, nous avons développé un système de modules basés sur le paradigme des contrats présenté dans le chapitre 2. Dans la section 5.2, ce système de modules a été employé pour définir des contrats dont les générateurs sont écrits en SIGNAL pour la spécification de processus SIGNAL. Ce système de modules encapsulant des équations sur des flots de données écrites dans la syntaxe SIGNAL a été implémenté en OCaml. Ce prototype produit un arbre de preuves qui consiste à :

- l'élaboration d'un programme SIGNAL décrivant la structure d'un système au moyen de la notion de modules,
- l'affectation de types respectant le système d'inférence du langage de modules,
- la génération d'obligations de preuves de raffinement ou de satisfaction de contrats exprimés au moyen de processus observateurs ou de propriétés temporelles spécifiées en SIGNAL.

Nous allons maintenant présenter le prouveur utilisé pour la résolution des preuves générées par le prototype.

5.4.1 SIGALI, le vérificateur associé à SIGNAL

L'outil Sigali [MRLBS01] permet de prouver la correction de propriétés dynamiques des comportements, il se limite cependant aux propriétés booléennes. La nature des équations du langage SIGNAL permet d'utiliser une méthode basée sur les systèmes dynamiques d'équations polynomiales sur $\mathbb{Z}/3\mathbb{Z}$ (c'est-à-dire les entiers modulo 3 : $\{0,\ 1,\ -1\}$) comme modèle formel de comportements. La technique consiste à manipuler un système d'équations à la place d'ensembles de solutions. Plus précisement, un ensemble d'états peut être représenté par un unique système polynomial. En effet, les trois états possibles d'un signal booléen x sont *présent et vrai* ($x = 1$), *présent et faux* ($x = -1$), et *absent* ($x = 0$). De la même manière, chaque primitive d'un processus SIGNAL peut être codée par une

équation polynomiale. Par exemple, `c = a when b` signifie : "Si `b` = 1 alors `c` = `a` sinon `c` = 0", ce qui se traduit par $c = a(-b - b^2)$: les solutions de cette équation sont l'ensemble des comportements satisfaisant l'équation `c = a when b`. Sigali implémente un ensemble d'opérateurs basiques tels que les quantifieurs, et les opérateurs temporels de la logique CTL.

5.4.2 Une obligation de preuve générée par le prototype

Nous présentons une preuve générée avec le prototype, associée à la satisfaction du contrat ($\mathbf{A}_{Intake}$,$\mathbf{G}_{Intake}$) par l'implémentation SIGNAL `engine`.

Rappelons la relation de satisfaction d'un contrat : Soient $\mathbf{C} = (\mathbf{A},\mathbf{G})$ un contrat, p un processus, $p \vDash \mathbf{C} \iff (\widehat{[p]} \sqcap \mathbf{A}) \sqsubseteq \mathbf{G}$.

L'ensemble des filtres étant une algèbre booléenne, il est trivial de déduire l'équivalence suivante : $p \vDash \mathbf{C} \iff (\widehat{[p]} \sqcap \mathbf{A}) = \widehat{[p]} \sqcap \mathbf{A} \sqcap \mathbf{G}$. La preuve générée en SIGNAL consiste à vérifier que le processus générateur de $(\widehat{[p]} \sqcap \mathbf{A})$ est défini par le même ensemble de traces que le processus générateur de $(\widehat{[p]} \sqcap \mathbf{A} \sqcap \mathbf{G})$.

La propriété sera ensuite exportée vers le prouveur de SIGNAL, Sigali, afin de prouver ou non que le programme généré satisfait la propriété. La satisfaction implique que l'affectation de types et le programme SIGNAL produit soient en accord avec la spécification attendue.

Afin de se limiter à des signaux booléens, la position angulaire de l'arbre à cames est codée par deux signaux `c125` et `c226` définis sur l'ensemble des valeurs booléennes. Les positions sont associées aux valeurs suivantes, exprimée en tant que programme SIGNAL incluant un appel au vérificateur Sigali.

- si `c125`= $faux$ et `c226`= $faux$, alors $0 \leq (cam\ mod\ 360) < 90$,
- si `c125`= $faux$ et `c226`= $vrai$, alors $90 \leq (cam\ mod\ 360) < 180$,
- si `c125`= $vrai$ et `c226`= $faux$, alors $180 \leq (cam\ mod\ 360) < 270$,
- si `c125`= $vrai$ et `c226`= $vrai$, alors $270 \leq (cam\ mod\ 360) < 360$.

Le prototype fournit l'obligation de preuve suivante :

```
1   process e_proof = ( ? boolean c125 ; boolean c226 ; )
2   (|
```

Appel de `engine6engine_assume7`, le générateur de $(\widehat{[p]} \sqcap \mathbf{A}_{Intake})$.

```
3     (out_def_intake1520, out_def_combustion1621,
4        out_def_compression1722, out_def_exhaust1823,
5        out_def_aux1924) := engine6engine_assume7(c125, c226)
```

Appel de `engine6engine_assume7engine_garantee8`, le générateur de $(\widehat{[p]} \sqcap \mathbf{A}_{Intake} \sqcap \mathbf{G}_{Intake})$.

```
6    |(out_def_intake3141, out_def_combustion3242,
7        out_def_compression3343, out_def_exhaust3444, out_def_aux3545)
8         := engine6engine_assume7engine_garantee8(c125, c226)
```

Le signal `obs_proc_49` **observe la satisfaction des contraintes de l'implémentation et de** $\mathbf{A}_{Intake}$.
Le signal `obs_proc_50` **observe la satisfaction des contraintes de l'implémentation ainsi que de** $\mathbf{A}_{Intake}$ **et** $\mathbf{G}_{Intake}$.

```
9    | obs_proc_49 := out_def_aux1924 default not(out_def_aux3545)
10                    default (obs_proc_49$1 init true)
11   | obs_proc_50 := out_def_aux3545 default not(out_def_aux1924)
12                    default (obs_proc_50$1 init true)
```

Sigali doit vérifier l'égalité des signaux observateurs `obs_proc_49` et `obs_proc_50` sur l'ensemble des états atteignables du système :

```
13   |Sigali(AG(B_Or(B_And(B_True(obs_proc_49),B_True(obs_proc_50)),
14               B_And(B_False(obs_proc_49),B_False(obs_proc_50)))))
15  |)
16  where
17   use SIGALI;                        %Appel de la librairie Sigali.%
18   boolean obs_proc_49, obs_proc_50 ;
19   boolean out_def_intake1520 ; boolean out_def_combustion1621 ;
20           boolean out_def_compression1722 ;
21           boolean out_def_exhaust1823 ;
22           boolean out_def_aux1924 ; boolean out_def_intake3141 ;
23           boolean out_def_combustion3242 ;
24           boolean out_def_compression3343 ;
25           boolean out_def_exhaust3444 ; boolean out_def_aux3545 ;
```

Définition du générateur `engine6engine_assume7` **de** $(\widehat{[p]} \sqcap \mathbf{A}_{Intake})$ **(lignes 26 à 73) :**

```
26  process engine6engine_assume7 =  (? boolean c19 ; boolean c210 ;
27                                    ! boolean out_def_intake1520 ;
28                                      boolean out_def_combustion1621 ;
29                                      boolean out_def_compression1722 ;
30                                      boolean out_def_exhaust1823 ;
31                                      boolean out_def_aux1924 ; )
32  (| (intake11, combustion12, compression13, exhaust14)
33          := engine6(c19, c210)
34   | (intake15, combustion16, compression17, exhaust18, aux19)
35          := engine_assume7(c19, c210)
```

Composition des sorties de l'implémentation avec les sorties du générateur de $\mathbf{A}_{Intake}$ (lignes 36 à 55) :

```
36    | out_def_aux1924 ^= aux19
```

Le signal booléen `out_def_aux1924` est présent et vrai si les hypothèses sont satisfaites :

```
37    | out_def_aux1924:=aux19 when ^aux19
38    | out_def_intake1520 ^= intake11
39    | out_def_intake1520:=(intake15 when (^0)  default intake11) when ^intake11
40    | out_def_combustion1621  ^= combustion12
41    | out_def_combustion1621  := combustion12
42    | out_def_compression1722 ^= compression13
43    | out_def_compression1722 := compression13
44    | out_def_exhaust1823     ^= exhaust14
45    | out_def_exhaust1823     := exhaust14
46   |)
47   where boolean intake11 ;
48         boolean combustion12 ;
49         boolean compression13 ;
50         boolean exhaust14 ;
51         boolean intake15 ;
52         boolean combustion16 ;
53         boolean compression17 ;
54         boolean exhaust18 ;
55         boolean aux19 ;
```

`engine6` désigne l'implémentation du moteur (lignes 56 à 64) :

```
56   process engine6 = (? boolean c1 ; boolean c2 ;
57                       ! boolean intake ; boolean combustion ;
58                         boolean compression ; boolean exhaust ; )
59   (| c1 ^= c2 ^= intake ^= compression ^= combustion ^= exhaust
60    | intake := true when (not(c1) and not (c2)) default false
61    | combustion := true when (not (c1) and c2) default false
62    | compression := true when (c1 and not(c2)) default false
63    | exhaust := true when (c1 and c2 ) default false |)
64   ;
65
```

`engine6_assume7` désigne le générateur de $\mathbf{A}_{Intake}$ (lignes 66 à 73) :

```
66   process engine_assume7 = (? boolean c1 ; boolean c2 ;
67                             ! boolean intake ; boolean combustion ;
68                               boolean compression ; boolean exhaust ;
```

```
69                                boolean aux ; )
70   (aux := (true when (not (c1) and not (c2))) default false |)
71   ;
72
73   end;
```

Définition du générateur `engine6engine_assume7engine_garantee8` **de** $(\widehat{[p]} \sqcap \mathbf{A}_{Intake} \sqcap \mathbf{G}_{Intake})$ **(lignes 74 à 134) :**

```
74   process engine6engine_assume7engine_garantee8 =
75           (? boolean c125 ; boolean c226 ;
76           ! boolean out_def_intake3141 ; boolean out_def_combustion3242 ;
77           boolean out_def_compression3343 ; boolean out_def_exhaust3444 ;
78           boolean out_def_aux3545 ; )
79   (| (intake27, combustion28, compression29, exhaust30)
80         := engine6(c125, c226)
81    | (intake31, combustion32, compression33, exhaust34, aux35)
82         := engine_assume7(c125, c226)
83    | (intake36, combustion37, compression38, exhaust39, aux40)
84         := engine_garantee8(c125, c226)
```

Composition des sorties de l'implémentation avec les sorties des générateurs de $\mathbf{A}_{Intake}$ **et de** $\mathbf{G}_{Intake}$ **(lignes 85 à 111) :**

```
85    | aux35 ^= out_def_aux3545 default aux35
```

Le signal booléen `out_def_aux3545` **est présent et vrai si les hypothèses et les garanties sont satisfaites :**

```
86    | out_def_aux3545:=(aux40 default aux35)
87         when ^((when ((aux35 when (aux35 ^* aux40))
88                        = (aux40 when (aux35 ^* aux40))))
89    | out_def_intake3141      ^= intake27
90    | out_def_intake3141      := intake27
91    | out_def_combustion3242  ^= combustion28
92    | out_def_combustion3242  := combustion28
93    | out_def_compression3343 ^= combustion29
94    | out_def_compression3343 := compression29
95    | out_def_exhaust3444     ^= exhaust30
96    | out_def_exhaust3444     := exhaust30
97    |)
98   where boolean intake27 ;
99         boolean combustion28 ;
100        boolean compression29 ;
101        boolean exhaust30 ;
102        boolean intake31 ;
```

```
103         boolean combustion32 ;
104         boolean compression33 ;
105         boolean exhaust34 ;
106         boolean aux35 ;
107         boolean intake36 ;
108         boolean combustion37 ;
109         boolean compression38 ;
110         boolean exhaust39 ;
111         boolean aux40 ;
```

`engine6` **désigne l'implémentation du moteur (lignes 112 à 120) :**

```
112   process engine6 = (? boolean c1 ; boolean c2 ;
113                       ! boolean intake ; boolean combustion ;
114                         boolean compression ; boolean exhaust ; )
115   (| c1 ^= c2 ^= intake ^= compression ^= combustion ^= exhaust
116    | intake := true when (not(c1) and not ( c2 ) ) default false
117    | combustion := true when (not(c1 ) and c2 ) default false
118    | compression := true when (c1 and not(c2)) default false
119    | exhaust := true when (c1 and c2) default false |)
120   ;
```

`engine6_assume7` **désigne le générateur de** $\mathbf{A}_{Intake}$ **(lignes 121 à 126) :**

```
121   process engine_assume7 = (? boolean c1 ; boolean c2 ;
122                             ! boolean intake ; boolean combustion ;
123                               boolean compression ; boolean exhaust ;
124                               boolean aux ; )
125   (aux := ( true when (not(c1) and not(c2))) default false |)
126   ;
```

`engine6_assume7` **désigne le générateur de** $\mathbf{G}_{Intake}$ **(lignes 127 à 134) :**

```
127   process engine_garantee8 = (? boolean c1 ; boolean c2 ;
128                               ! boolean intake ; boolean combustion ;
129                                 boolean compression ; boolean exhaust ;
130                                 boolean aux ; )
131   (aux := ( true when out_def_intake3141 ) default false |)
132   ;
133
134   end;
135
136   end;
```

Soit le fichier `engine.sig` désignant l'obligation de preuve ci-dessus, l'exportation vers Sigali se fait par la commande suivante :

```
%signal -z3z engine.sig
```

Le fichier `engine_proof.z3z` est généré après la compilation du programme SIGNAL `engine.sig`. Ce fichier est la traduction du fichier de l'obligation de preuve `engine.sig` dans le formalisme de Sigali. Le fichier `engine_proof_CMD.z3z` est également généré. Ce fichier contient les commandes pour charger le fichier `engine_proof.z3z` :

```
read("e_proof.z3z");
read("Creat_SDP.lib");
read("Bibli.lib");
PROP_306: B_True(S,obs_proc_49);
PROP_303: B_True(S,obs_proc_50);
PROP_301: B_And(PROP_306, PROP_303);
PROP_284: B_False(S,obs_proc_49);
PROP_279: B_False(S,obs_proc_50);
PROP_277: B_And(PROP_284, PROP_279);
PROP: B_Or(PROP_301, PROP_277);
RESULT : AG(PROP);
```

`RESULT` est l'ensemble des états du système pour lesquels tous les successeurs directs ou indirects vérifient la propriété `PROP`. En effet, `PROP` exprime le fait que les observateurs `obs_proc_49` et `obs_proc_50` sont tous les deux vrais ou bien tous les deux faux.

Il ne nous reste plus qu'à lancer une session de Sigali, et exécuter les commandes du fichier `e_proof_CMD.z3z` :

```
%sigali
 *--------------------------------------*
 * Sigali - version 2.3 (Dec 2005)     *
 *--------------------------------------*

Sigali : read("e_proof.z3z");
read("Creat_SDP.lib");
read("Bibli.lib");
PROP_306: B_True(S,obs_proc_49);
PROP_303: B_True(S,obs_proc_50);
PROP_301: B_And(PROP_306, PROP_303);
PROP_284: B_False(S,obs_proc_49);
PROP_279: B_False(S,obs_proc_50);
PROP_277: B_And(PROP_284, PROP_279);
PROP: B_Or(PROP_301, PROP_277);
```

```
RESULT : AG(PROP);
----------------------------------------

----------------------------------------
Sigali :
----------------------------------------
Polynomial Dynamical System Building
----------------------------------------
```

`PROP` est une propriété traduisant le fait que `obs_proc_49` et `obs_proc_50` sont affectées aux mêmes valeurs booléennes dans un même état du système. `AG` est un opération issu de la logique CTL. Dans Sigali, `AG(PROP)` retourne l'ensemble des états du système pour lesquels les états successeurs directs ou indirects satisfont la propriété `PROP`. Sur la figure 5.11, les états initiaux du système S1 satisfont la propriété `AG(PROP)`, tandis que `AG(PROP)` n'est pas satisfaite par les états initiaux du système S2.

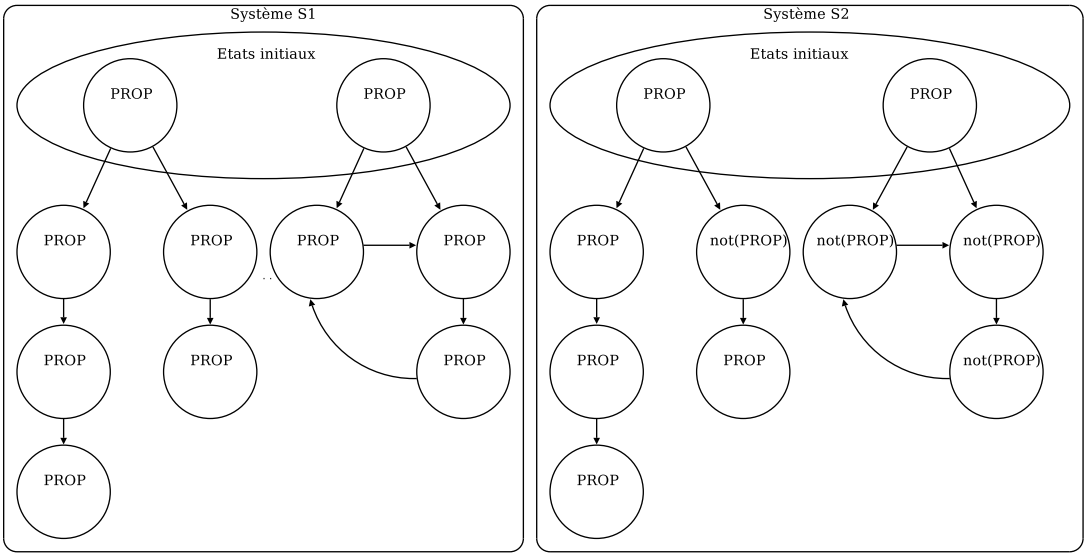


FIG. 5.11 – A gauche, les états initiaux du système S1 satisfont la propriété AG(PROP). A droite, AG(PROP) n'est pas satisfaite par les états initiaux du système S2.

Il ne reste plus qu'à demander à Sigali si tous les états successeurs (de manière directe ou indirecte) des états initiaux satisfont la propriété `RESULT` :

```
Sigali : subset(initial(S),RESULT);
----------------------------------------
True
----------------------------------------
```

Les propriétés générées peuvent cependant être utilisées à d'autres fins. Comme dans [MBLBLG00], il est possible d'utiliser le service de synthèse de contrôleur de Sigali pour générer automatiquement un programme SIGNAL qui applique la propriété sur le programme généré. Dans le cadre d'un système à nombre infini d'états (avec des propriétés sur des nombres réels), une autre possibilité d'utilisation serait de générer du code défensif lorsque la propriété est violée, par exemple, afin de produire une trace.

5.5 Un second cas d'utilisation

Nous reprenons l'exemple du composant de transmission de messages modélisé au moyen d'automates d'interface dans la sous-section 1.2.1. Nous utilisons ici le langage SIGNAL pour définir le processus générateur des hypothèses et le processus générateur des garanties du contrat spécifiant qu'un composant utilisé avec le canal de transmission (modélisé par l'automate de la figure 5.13) qui ne renvoie jamais de valeur sur la variable *nack*, ne retournera jamais de valeur sur la variable $fail$ à l'utilisateur (modélisé par l'automate de la figure 5.12).

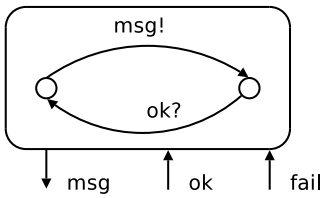


FIG. 5.12 – Automate d'interface d'un utilisateur.

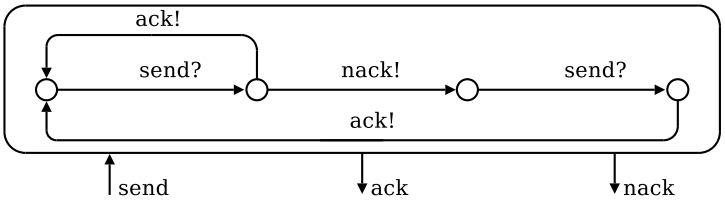


FIG. 5.13 – Canal de communication.

Nous utilisons un processus intermédiaire : le processus `interleave` impose la contrainte que chaque valeur reçue sur sa variable d'entrée `a` précède une valeur reçue sur sa variable d'entrée `b`, et que chaque valeur reçue sur sa variable d'entrée `b` précède une valeur reçue sur sa variable d'entrée `a`.

L'écriture de ce processus en SIGNAL est la suivante :

```
process interleave =
        (? boolean a, b ; )
(| a ^= (when x)
 | x := (when not(x))
 | x := not(x$1 init true)
 |) where boolean x end;
```

Le processus `channel` spécifié ci-dessous en SIGNAL décrit un canal de communication acheminant toujours un message jusqu'à sa destination, c'est-à-dire ne retournant jamais de valeurs sur la variable *nack*. Ce canal de communication est le contexte d'utilisation d'un composant de transmission de messages. Ainsi le processus `channel` est le processus générateur du filtre spécifiant les hypothèses du contrat. La valeur de `obs` est toujours *true*.

```
process channel(? boolean msg, ack, nack ;
                ! boolean send, fail, obs ; )
(| interleave(msg, ack)
 | interleave(msg, nack)
 | interleave((nack default ack), ack)
 | obs := true
 |);
```

Le processus `user` spécifié en SIGNAL décrit le comportement d'un utilisateur n'acceptant pas que son message ne soit pas transmis, c'est-à-dire n'acceptant pas la lecture d'une valeur sur la variable $fail$. Cet utilisateur représente les garanties du composant de transmission de messages. Ainsi le processus `user` est le processus générateur du filtre spécifiant les garanties du contrat. La valeur de `obs` est *true* tant que le processus `user` ne reçoit pas de valeur sur la variable $fail$.

```
process user(? boolean msg, ack, nack ;
             ! boolean send, fail, obs ; )
(| interleave(msg, ok)
 | obs := true when ok
          default false when fail|);
```

Considérons le composant caractérisé par l'automate de la figure 5.14 et défini par l'implémentation suivante :

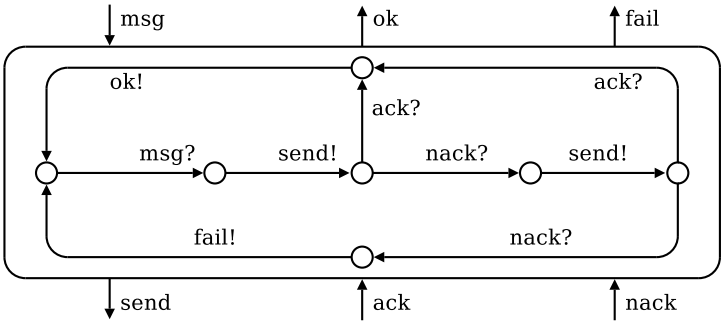


FIG. 5.14 – Automate d'interface du composant.

```
process comp(? boolean msg, ack, nack ; ! boolean send, fail ; )
(| cycle ^= (ack default nack)
 | interleave(send, (ack default nack))
 | cycle := (((cycle$1 init 0)+1) when nack) default (0 when ack)
 | tmp := nack when cycle>1
 | interleave(tmp,fail)
 | interleave(ack,ok)
 | interleave(send,(ok default fail))
 |) where int cycle ; event tmp ;
    end;
```

Ce composant retourne une valeur sur la variable *fail* après deux réceptions consécutives de valeurs sur la variable *nack*. Le canal n'émettant jamais deux valeurs consécutives sur la variable *nack*, le composant satisfait le contrat.

Le contrat ($[\widehat{\texttt{channel}}]$,$[\widehat{\texttt{user}}]$) est satisfait si la relation de satisfaction est vérifiée :

$$(\widehat{[\texttt{comp}]} \sqcap \widehat{[\texttt{channel}]}) \sqsubseteq \widehat{[\texttt{user}]}$$

Rappellons que dans le prototype cela consiste à vérifier que le processus générateur de $(\widehat{[p]} \sqcap \mathbf{A})$ est défini par le même ensemble de traces que le processus générateur de $(\widehat{[p]} \sqcap \mathbf{A} \sqcap \mathbf{G})$. Le processus `user` ne contraignant aucune variable hormis la variable `obs`, la vérification se fait par l'observation des valeurs des variables `obs` définies dans les processus `user` et `channel`. Ainsi, cette vérification consiste à vérifier que `obs` est définie par le même ensemble de traces dans $(\widehat{[p]} \sqcap \mathbf{A})$ et dans $(\widehat{[p]} \sqcap \mathbf{A} \sqcap \mathbf{G})$.

Conclusion

Les travaux de recherche menés dans le cadre de cette thèse ont eu pour objectifs de définir une méthode de spécification formelle par composants de systèmes temps réel modélisés selon l'approche synchrone.

Nous avons souhaité définir une méthodologie de description de ces systèmes centrée autour de concepts mathématiques forts. La méthode développée offre ainsi la possiblité de pouvoir certifier que l'implémentation obtenue par raffinement itératif à partir d'une spécification de haut niveau, est conforme aux propriétés attendues dans le cahier des charges du système.

Contribution

Nous avons développé une méthode de modélisation générique de contrats dans laquelle un contrat est un couple hypothèses/garanties. Les garanties associées à un composant ne sont satisfaites que si les hypothèses sont satisfaites par l'environnement d'exécution.

Approche par contrats des systèmes temps réel. La structure de base du modèle de contrats est celle des comportements. Un comportement est une trace d'exécution définie sur un ensemble de variables, ainsi un processus est un ensemble de comportements définis sur un même ensemble de variables. La généricité de la méthode repose sur le fait que le domaine des variables n'est pas déterminé par le modèle. La définition du domaine des variables par un ensemble de fonctions d'un domaine de temps discret vers un domaine de valeurs, permet l'utilisation direct du modèle pour la synthèse d'architectures temps réel spécifiées avec un langage tel que SIGNAL, dont la sémantique est également centrée sur la notion de comportements.

Gestion des ensembles de variables. Les filtres permettent de modéliser les hypothèses et les garanties par un ensemble de processus modélisant une même contrainte et définis sur des ensembles de variables différents. Ainsi le développeur

décrit une propriété sans se soucier du fait que le processus devant la satisfaire sera défini sur un nombre plus ou moins important de variables.

Multi-vues et substituabilité. L'ensemble des filtres définit une algèbre booléenne facilitant les opérations de composition, la définition d'une borne inférieure et d'une borne supérieure, ainsi que la définition d'un filtre complémentaire. De cette structure mathématique découle l'ensemble des contrats modélisés par un ensemble de paires (hypothèses/garanties) de filtres. L'ensemble de contrats constitue une algèbre de Heyting. La relation de composition obtenue par le calcul de la borne inférieure de deux contrats offre de multiples points de vue sur un même composant. Il est ainsi possible de représenter un composant par un ensemble de contrats devant tous être satisfaits, ainsi le composant peut être considéré comme la composition des différentes vues, ou plus particulièrement comme la borne inférieure des différents contrats de l'ensemble. De la même manière, la relation d'ordre partiel sur les contrats axée sur la notion de substituabilité, permet aisément de vérifier la substituabilité de deux composants dans une architecture.

Langage de modules et typage par contrats. Le langage de modules permet d'intégrer la méthodologie de conception par contrats dans de nombreux langages, tels que SIGNAL, et par conséquent, l'intégration des concepts de vues multiples et de substituabilité de l'algèbre de contrats. Ces concepts permettent entre autres une nouvelle répartition du travail d'implémentation au sein d'une équipe de développement. Le développement d'un composant ou d'une architecture peut ainsi être divisé selon les différentes vues. Par exemple, un développeur se chargera de la satisfaction des contraintes d'ordonnancement, tandis qu'un autre abordera les contraintes de dépendance de données : le langage de modules se chargera de composer les implémentations de chaque vue. La tâche du développeur est alors simplifiée, ce qui contribue à une réduction significative du coût global de conception, sans négliger la fiabilité accrue du système produit.

Nous venons de voir que l'utilisation de l'algèbre de contrats remplit la propriété de substituabilité attendue par les motivations de l'introduction. De par leur nature, les contrats répondent aux autres attentes développées dans l'introduction. La relation de satisfaction d'un contrat par un processus remplit le besoin de modularité : la capacité de tester la conformité entre la mise en œuvre d'un composant et son implémentation. Cette relation est transcrite dans le langage de modules par la contrainte d'affectation d'un module à un type de module, dans le système d'inférence de types du langage (règle (13) sur la figure 4.14 de la sous-section 4.2.7, chapitre 4).

La compositionnalité est la capacité de vérifier l'adéquation entre le modèle d'une application et son architecture d'exécution. De manière itérative ou modulaire, cette propriété se traduit dans l'algèbre de contrats par la possiblité de tester le raffinement du contrat spécifiant le type de composant attendu dans une architecture, par le contrat associé au composant que l'on souhaite utiliser.

Intégration dans POLYCHRONY

Dans le but de profiter des avantages de la conception par contrats dans les systèmes temps réel, et de répondre aux attentes développées dans l'introduction, il est possible d'intégrer ces travaux dans POLYCHRONY. Rappelons que POLYCHRONY est une plateforme de développement pour la définition de systèmes temps réel spécifiés en SIGNAL.

Le prototype développé réalise la preuve de la satisfaction de contrats spécifiés en SIGNAL par la synthèse d'observateurs, eux aussi décrits en SIGNAL. Ainsi, nous avons pu montrer que cette approche permettait la validation d'un système en démontrant la satisfaction des contrats associés aux composants d'une architecture.

Une application immédiate est la possibilité d'intégrer des composants compilés séparément, dont les contraintes d'exécution requises et les propriétés garanties sont inscrites dans le contrat. Par conséquent, cette approche est applicable à la génération de code modulaire.

Le travail d'intégration dans POLYCHRONY est avant tout un travail de réalisation d'interface afin de rendre l'écriture des contrats la plus simple et la plus expressive possible. D'autre part, il est intéressant de ne pas se limiter à SIGNAL pour l'écriture des contrats. Un langage expressif permettra d'exprimer davantage de propriétés, ainsi l'utilisation d'expressions CTL permettra de spécifier des propriétés de vivacité. Notons que le model-checker Sigali permet de prouver la satisfaction de propriétés exprimées avec des expressions CTL, dans des programmes SIGNAL, ouvrant la possibilité de spécifier les propriétés de vivacité pour les processus SIGNAL.

Perspectives

Le développement de ce modèle de contrats offre de nombreuses perspectives. D'abord du point de vue de l'expressivité des contrats, deux questions évidentes vont se poser :

- Quels sont les langages les mieux adaptés pour l'écriture des contrats et pour quels types de propriétés ? Y-a-t-il un moyen de choisir "le bon modèle" d'expression ?

- Comment développer une interface homme/machine pertinente pour la définition de contrats par un utilisateur ? Il pourrait être judicieux de fournir un éditeur spécifique pour différents types de propriétés (propriétés de sûreté, de vivacité).

Dans un second temps, il semble intéressant de tester ce modèle pour différents types de problèmes :

- des problèmes à dimension industrielle,
- des problèmes restreints à des propriétés de synchronisation ou d'ordonnancement.

Ceci afin d'étudier les dimensions spatiales des preuves générées, et par conséquent tester la capacité des prouveurs à effectuer de telles preuves.

Un autre champ d'application est le calcul des propriétés minimales [EGP08] que doit satisfaire l'environnement pour que le composant satisfasse les propriétés attendues. Ceci a pour but d'élaborer le contexte d'exécution nécessaire au bon fonctionnement d'un composant.

Bibliographie

[AHK02] R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49(5) :672–713, 2002.

[AJ87] S. Abramsky and A. Jung. Domain theory. In H*andbook of logic in computer science (vol. 3) : semantic structures*, pages 1–168. Oxford University Press, 1987.

[AL93] M. Abadi and L. Lamport. Composing specifications. *ACM Trans. Program. Lang. Syst.*, 15(1) :73–132, 1993.

[AMPF07a] C. André, F. Mallet, and M.-A. Peraldi-Frati. A multi-form time approach to real-time system modeling. Research Report RR 2007-14-FR, I3S, 2007.

[AMPF07b] C. André, F. Mallet, and M-A. Peraldi-Frati. A multiform time approach to real-time system modeling : application to an automotive system. In *International Symposium on Industrial Embedded Systems*, pages 234–241, 2007.

[BCP07] A. Benveniste, B. Caillaud, and R. Passerone. A generic model of contracts for embedded systems. Technical Report 6214, INRIA Rennes, June 2007.

[Bel99] J. L. Bell. Boolean algebras and distributive lattices treated constructively. *Math. Logic Quarterly*, 45 :135–143, 1999.

[Ber89] G. Berry. Real time programming : special purpose or general purpose languages. Research Report RR-1065, INRIA, 1989.

[BFMW01] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass - Java with assertions. *Electronic Notes in Theoretical Computer Science*, 55(2) :1–15, 2001.

[BJPW99] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *IEEE Computer*, 13(7), July 1999.

[BPS06] R. Budde, A. Poigné, and K.-H. Sylla. Synergy - an object-oriented synchronous language. I*n Havelund, K., Rosu, G. eds : Runtime*

Verification, Volume 153 of Electronic Notes in Theoretical Computer Science(1) :99–115, 2006.

[Bro97] M. Broy. Compositional refinement of interactive systems. *Journal of the ACM*, 44(6) :850–891, 1997.

[dAH01] L. de Alfaro and T. A. Henzinger. Interface automata. *SIGSOFT Softw. Eng. Notes*, 26(5) :109–120, 2001.

[DHJP08] L. Doyen, T. A. Henzinger, B. Jobstmann, and T. Petrov. Interface theories with component reuse. In *EMSOFT '08 : Proceedings of the 8th ACM international conference on Embedded software*, pages 79–88. ACM, 2008.

[DI99] Maley D. and Spence I. Emulating design by contract in c++. In *Emulating Design by Contract in C++*. IEEE Computer Society, 1999.

[EGP08] M. Emmi, D. Giannakopoulou, and C.S. Pasareanu. Assume-guarantee verification for interface automata. In 15*th International Symposium on Formal Methods*, Turku, Finland, May 2008.

[ELLSV97] S. Edwards, L. Lavagno, E. A. Lee, and A. Sangiovanni-Vincentelli. Design of embedded systems: formal models, validation, and synthesis. *Proceedings of the IEEE*, 85(3) :366–390, 1997.

[FM01] S. Flake and W. Mueller. An OCL extension for realtime constraints. In *Lecture Notes in Computer Science 2263*, pages 150–171, 2001.

[GLD94] T. Gautier, Paul Le Guernic, and F. Dupont. Signal v4 : manuel de reference. Technical Report 832, IRISA, Juin 1994.

[GLTG08] Y. Glouche, P. Le Guernic, J.-P. Talpin, and T. Gautier. A boolean algebra of contracts for logical assume-guarantee reasoning. Technical Report 6570, INRIA Rennes, July 2008.

[GLTG09] Y. Glouche, P. Le Guernic, J.-P Talpin, and T. Gautier. A boolean algebra of contracts for logical assume-guarantee reasoning. In *6th International Workshop on Formal Aspects of Component Software (FACS 2009)*, November 2009.

[GTLG09] Y. Glouche, J.-P. Talpin, P. Le Guernic, and T. Gautier. A module language for typing by contracts. In E. Denney, D. Giannakopoulou, and C. S. Păsăreanu, editors, *Proceedings of the First NASA Formal Methods Symposium*, pages 86–95. NASA Ames Research Center, Moffett Field, CA, USA, April 2009.

[Hal98] N. Halbwachs. Synchronous programming of reactive systems, a tutorial and commented bibliography. In *Tenth International*

Conference on Computer-Aided Verification, CAV'98, Vancouver (B.C.), june 1998. LNCS 1427, Springer Verlag.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10) :576–580, October 1969.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. *Logics and models of concurrent systems*, pages 477–498, 1985.

[Kop97] H. Kopetz. Component-based design of large distributed real-time systems. *Control Engineering Practice*, 6(1) :53–60, January 1997.

[KT04] M. Kerbœuf and J.-P. Talpin. Encapsulation and behavioural inheritance in a synchronous model of computation for embedded system services adaptation. In *Special issue on process algebra and system architectures*, volume volume (c) Elsevier, 2004.

[LBR99] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. *Behavioral Specifications of Businesses and Systems*, pages 175–188, 1999.

[Ler00] X. Leroy. A modular module system. *Journal of Functional Programming*, v. 10(3), 2000.

[LNW07] K. G. Larsen, U. Nyman, and A. Wasowski. Modal I/O automata for interface and product line theories. In R. De Nicola, editor, *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 64–79. Springer, 2007.

[LSV98] E.A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(12) :1217–1229, 1998.

[LTL03] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. Polychrony for System Design. *Journal for Circuits, Systems and Computers*, 12(3) :261–304, April 2003.

[Maz89] A. Mazurkiewicz. Basic notions of trace theory. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop*, pages 285–363, London, UK, 1989. Springer-Verlag.

[MBLBLG00] H. Marchand, P. Bournai, M. Le Borgne, and P. Le Guernic. Synthesis of discrete-event controllers based on the signal environment. *Discrete Event Dynamic System : Theory and Applications*, 10(4) :325–346, October 2000.

[Mey97] B. Meyer. *Object-oriented software construction (2nd ed.)*. Prentice-Hall, Inc., 1997.

[MM04] F. Maraninchi and L. Morel. Logical-time contracts for reactive embedded components. In *EUROMICRO*, pages 48–55. IEEE Computer Society, 2004.

[MMM02] R. Mitchell, J. McKim, and B. Meyer. *Design by contract, by example*. Addison Wesley Longman Publishing Co., Inc., 2002.

[MRLBS01] H. Marchand, E. Rutten, M. Le Borgne, and M. Samaan. Formal verification of programs specified with Signal : Application to a power transformer station controller. *Science of Computer Programming*, 41(1) :85–104, 2001.

[NTGLG97] D. Nowak, J.-P. Talpin, T. Gautier, and P. Le Guernic. An ML-like module system for the synchronous language SIGNAL. In *European Conference on Parallel Processing (EUROPAR'97)*, August 1997.

[RBB+09] J.-B. Raclet, E. Badouel, A. Benveniste, B. Caillaud, and R. Passerone. Why are modalities good for interface theories ? In *Proc. of the 9th International Conference on Application of Concurrency to System Design (ACSD'09)*, pages 119–127. IEEE Computer Society Press, 2009.

[ZG02] P. Ziemann and M. Gogolla. An extension of OCL with temporal logic. In *Critical Systems Development with UML-Proceedings of the UML'02 workshop, Technische Universität München, Institut für Informatik*, pages 53–62, 2002.

Table des figures

Résumé

Les contrats basés sur les notions d'hypothèses/garanties constituent un paradigme expressif pour une conception modulaire et compositionnelle de spécification de programmes. Ils sont devenus un concept fondamental dans les procédés employés par les outils de conception assistée par ordinateur, pour la conception de systèmes informatiques. Dans cette thèse, nous élaborons des fondements pour la mise en œuvre de systèmes embarqués basée sur la notion de contrats. Nous proposons ainsi une algèbre de contrats basée sur deux concepts simples : les hypothèses et les garanties des composants sont définies par des filtres, les filtres sont caractérisés par une structure d'algèbre booléenne. Les choix effectués pour définir la structure des filtres permettent de définir une algèbre de Heyting sur l'ensemble des contrats. Un cadre de travail est ainsi défini, dans lequel les contrats sont utilisés pour vérifier la correction des hypothèses faites sur le contexte d'utilisation d'un composant, et pour fournir à l'environnement les garanties qui lui sont demandées. Nous utilisons cette algèbre pour définir un système de modules dont le paradigme de typage est basé sur la notion de contrats. Le type d'un module est un contrat caractérisé par les hypothèses faites par l'environnement et les garanties offertes par les comportements du module. Nous illustrons cette présentation avec la spécification d'un moteur à quatre temps.

Abstract

Contract-based design is an expressive paradigm for a modular and compositional specification of programs. It is in turn becoming a fundamental concept in mainstream industrial computer-aided design tools for embedded system design. In this thesis, we elaborate new foundations for contract-based embedded system design by proposing a general-purpose algebra of assume/guarantee contracts based on two simple concepts : first, the assumption or guarantee of a component is defined as a filter and, second, filters enjoy the structure of a Boolean algebra. This yields a structure of contracts that is a Heyting algebra. In this framework, contracts are used to negotiate the correctness of assumptions made on the definition of a component at the point where it is used and provides guarantees to its environment. We put this algebra to work for the definition of a general purpose module system whose typing paradigm is based on the notion of contract. The type of a module is a contract holding assumptions made and guarantees offered by its behaviors. We illustrate this presentation with the specification of a simplified 4-stroke engine model.