



# Formalisation en Coq des systèmes de transitions modales

Formalisation en Coq des systèmes de transitions modales.

Mémoire de stage.

Établissement mère : Université de Bordeaux.

Étudiant : Joas Yannick Kinouani, Université de Bordeaux.

Responsable pédagogique : Mme Anca Muscholl, professeur d'université, équipe Méthodes formelles, Laboratoire bordelais de recherche en informatique (La-BRI), Université de Bordeaux.

Établissement d'accueil : Université Toulouse 3 Paul Sabatier.

Maîtres de stage : M. Jean-Paul Bodeveix, professeur d'université, et M. Mamoun Filali, chargé de recherche au Centre national de la recherche scientifique (CNRS), équipe Assistance à la certification d'applications distribuées et embarquées (ACADIE), Institut de recherche en informatique de Toulouse (IRIT), Université Toulouse 3 Paul Sabatier.

Photos de couverture :

© Semmick Photo / Shutterstock.com (photographe), 2014. *Toulouse, France-July 21, 2014: Tourists and locals pay a visit to the Capitole de Toulouse, built in 1190* (photo). <http://www.shutterstock.com> (en ligne), téléchargée le 12 mars 2015.

© David Pegzlz / Shutterstock.com (photographe), 2014. *Bordeaux, France-July 31, 2014: People cooling in front of Place de la Bourse* (photo). <http://www.shutterstock.com> (en ligne), téléchargée le 6 février 2015.

# Avant-propos

Il est intuitivement évident que le logiciel zéro défaut n'existe pas ; rien n'est parfait dans ce monde. On peut toutefois s'en approcher [Por02]. L'Université de Bordeaux propose à ses étudiants un parcours en vérification logicielle axé autour de cette problématique. Ils y apprennent deux façons de réduire la présence de bogues : avant la programmation par la modélisation, et pendant la programmation par le typage statique. Les chapitres 1 et 2 survolent respectivement ces deux méthodes dites *formelles* car fondées sur les mathématiques.

La modélisation (cf. section 1.1) consiste à construire un objet mathématique  $M$  — un *modèle* — qui modélise le futur logiciel, c.-à-d. qui se comporte comme le logiciel que l'on prévoit de développer, et sur lequel on peut raisonner, du fait de sa nature mathématique, pour en déduire les propriétés. Si celles-ci coïncident avec le cahier des charges établi pour le projet, alors on écrit, ou génère si possible, le code source du programme de manière conforme à  $M$ .

Le typage statique (cf. section 2.1) permet d'annoter les données et les opérations que le logiciel effectue sur celles-ci afin de certifier dès la compilation, proportionnellement au degré d'expressivité qu'il autorise, l'impossibilité d'une partie ou de la totalité des comportements indésirables lors de l'exécution.

Ce stage portait sur un croisement des deux approches : l'implémentation de modèles avec le typage statique du langage Coq.

## Remerciements

La bonne fin d'une chose est plus agréable que son commencement. L'École de la République n'a pas toujours été facile ; on n'y réussit pas tout seul. Et c'est avec soulagement que j'exprime ma gratitude à toutes les personnes qui m'ont encouragé, enseigné, et encadré durant cette maîtrise, et à toutes celles et ceux qui, au long de ma scolarité, m'ont assisté, motivé, et poussé.

Je remercie chaleureusement M. Jean-Paul Bodeveix et M. Mamoun Filali, qui m'ont accueilli au sein de leur équipe de recherche dans une ambiance de

travail très agréable. Je remercie également la formidable équipe pédagogique du parcours Vérification logicielle de l'Université de Bordeaux : M. Philippe Narbel, M. Pierre Castéran, M. Alain Griffault, M. Marc Zeitoun, M. Frédéric Herbretau, Mme Anca Muscholl, M. Hugo Gimbert, M. Aymeric Vincent, M. Grégoire Sutre, M. Jérôme Leroux, et M. Paul Dorbec.

Un merci spécial à ma tendre maman, et au bon Dieu, pour les nombreux coups de chance qui m'ont amené là où je suis.

## Notes de production

Ce rapport a été produit avec PDFL<sup>A</sup>T<sub>E</sub>X et les extensions L<sup>A</sup>T<sub>E</sub>X de T<sub>E</sub>Xlive 2014. Il utilise Romande ADF comme police à empattement, Latin Modern Sans comme police sans empattement, Computer Concrete pour les codes sources, et Euler VM pour les mathématiques.

# Table des matières

<b>Avant-propos</b>	<b>i</b>
<b>1 Problématique et état de l'art</b>	<b>1</b>
1.1 Généralités sur la modélisation logicielle . . . . .	2
1.1.1 Institution . . . . .	3
1.1.2 Systèmes de transitions . . . . .	7
1.2 Développement à base de composants . . . . .	10
1.2.1 Assemblage et remplacement de composants . . . . .	10
1.2.2 Métathéorie des contrats . . . . .	13
1.2.3 Systèmes de transitions modales à évènements structurés .	17
1.3 Objectifs du stage . . . . .	20
<b>2 Formalisation en Coq des systèmes de transitions modales</b>	<b>21</b>
2.1 Termes et types en Coq . . . . .	21
2.1.1 Des types comme valeurs de première classe . . . . .	22
2.1.2 Ordre supérieur . . . . .	23
2.1.3 Types dépendants . . . . .	23
2.1.4 Correspondance de Curry-Howard . . . . .	25
2.2 Architecture de la bibliothèque . . . . .	26
2.3 Systèmes de transitions modales en Coq . . . . .	27
2.4 Résultat et conclusion . . . . .	31
<b>A Ensembles ordonnés</b>	<b>33</b>



# 1

## Problématique et état de l'art

Dans ce chapitre introductif, nous donnons les notions préliminaires. La section 1.1 resitue le stage dans le cadre de la maîtrise Vérification logicielle de l'Université de Bordeaux en expliquant la problématique de la modélisation logicielle<sup>1</sup>. La section 1.2 se focalise sur le thème du stage, à savoir l'assemblage et le raffinement de modèles, et leur formalisation mathématique par les systèmes de transitions modales à évènements structurés [Bau+12]. Le contexte scientifique étant ainsi clarifié, la section 1.3 présente l'objectif du stage — l'implémentation en Coq des systèmes de transitions modales à évènements structurés.

Le laboratoire d'accueil, l'Institut de recherche en informatique de Toulouse (IRIT), est une unité mixte — c.-à-d. multitutelle — de recherche qui regroupe<sup>2</sup> 241 enseignants-chercheurs, 27 chercheurs, 44 ingénieurs, techniciens, et administratifs, 220 doctorants, et 155 chercheurs postdoctoraux, invités, ou contractuels. 6 % d'entre eux ont pour tutelle l'Université Toulouse 1 Capitole, 7 % l'Université Toulouse 2 Jean Jaurès, 46 % l'Université Toulouse 3 Paul Sabatier, 19 % l'Institut national polytechnique (INP) de Toulouse, et 7 % le Centre national de la recherche scientifique (CNRS). Il y a sept thèmes pour 20 équipes de recherche : (1) l'analyse et la synthèse d'information (quatre équipes); (2) l'indexation et la recherche d'information (deux équipes); (3) l'interaction, la coopération, et l'adaptation par l'expérimentation (deux équipes); (4) le raisonnement et la décision (trois équipes); (5) la modélisation, les algorithmes, et le calcul haute performance (une équipe); (6) l'architecture, les systèmes, et les réseaux (cinq équipes); (7) la sûreté de développement du logiciel et la certification (trois équipes). J'ai été accueilli au sein de l'équipe Assistance à la certification d'applications distribuées et embarquées (ACADIE, thème 7) par Pr. Jean-Paul Bodeveix, et M. Mamoun Filali, chargé de recherche au CNRS.

---

1. Les techniques de vérification enseignées dans le parcours Vérification logicielle de l'Université de Bordeaux sont la modélisation logicielle et le typage statique. Ce chapitre parle de la modélisation logicielle, et le chapitre 2 du typage statique.

2. Chiffres du 26 mai 2015. Voir [www.irit.fr](http://www.irit.fr).

## 1.1. Généralités sur la modélisation logicielle

Dans un bureau d'étude, l'ingénieur civil ou aérospatial raisonne, selon des théories propres à son domaine d'expertise, sur un modèle, par exemple un dessin, une maquette en trois dimensions, ou une simulation informatique, du projet à construire, par exemple du futur bâtiment, pont, ou avion. Il ne fait pas du test, car la construction n'a pas encore commencée. Il fait de la modélisation : il dispose d'un cahier des charges, c.-à-d. d'une liste de propriétés que le bâtiment, le pont, ou l'avion devra respecter, et il cherche un modèle qui les satisfait toutes. L'idée est que si les constructeurs construisent de façon conforme au modèle, le résultat final remplira lui aussi toutes les exigences. La modélisation logicielle est la mise en pratique de cette approche dans le génie logiciel [Por02]. À l'Université de Bordeaux, les étudiants en modélisation logicielle apprennent :

- des langages mathématiques, comme  $\text{Lang}_{\text{LTL}}$  — le langage de la logique LTL<sup>3</sup> du temps linéaire — et  $\text{Lang}_{\text{CTL}}$  — celui de la logique CTL<sup>4</sup> du temps arborescent —, permettant d'exprimer mathématiquement les exigences que le logiciel à construire devra satisfaire. Du point de vue mathématique, un langage n'est ni plus ni moins qu'un ensemble *Lang d'expressions bien formées*. Par exemple, la langue française  $\text{Lang}_{\text{Fr}}$  est, à l'écrit, l'ensemble des expressions françaises bien écrites, chacune d'elles étant une suite finie et grammaticalement correcte de lettres et de signes de ponctuation. Ce document est une expression française bien écrite — si on pardonne à l'auteur ses fautes grammaticales. L'exigence “le logiciel ne bogue jamais” est aussi une expression française bien écrite, et sa traduction en  $\text{Lang}_{\text{CTL}}$  est “ $\text{AG}\neg\text{bogue}$ ”;
- des classes de modèles, chacune adéquate pour modéliser tel ou tel aspect d'un logiciel ou d'un protocole réseau — vu comme un logiciel distribué exécuté simultanément par des processeurs ou processus connectés entre eux. Par exemple, les systèmes de transitions, les automates Altarica, les machines B évènementielles, les  $\omega$ -automates, les automates probabilistes, les automates de contrôle (*control flow automata* en anglais), les calculs locaux, les canaux PAPS (*FIFO channels* en anglais) — gérés selon la politique « premier arrivé, premier servi » —, les protocoles pour réseaux ad hoc, et les réseaux de Pétri.

Pour plus de souplesse, les langages sont généralement paramétrés par une *signature*, c.-à-d. un choix libre, modifiable, et arbitraire de notations, au sens où chacun peut opter pour telle ou telle signature sans changer la signification des expressions. Les signatures disponibles sont restreintes par une collection  $\text{Sig}$  de signatures. Une signature  $\Sigma \in \text{Sig}$  ayant été adoptée, un  $\Sigma$ -langage  $\text{Lang}_\Sigma$  est un ensemble de  $\Sigma$ -expressions bien formées, c.-à-d. un ensemble d'expressions bien formées et construites sur la base de la signature  $\Sigma$ . Par exemple, pour  $\text{Lang}_{\text{CTL}}$ , une signature est un ensemble de symboles, comme  $\Sigma = \{\text{bogue}\}$  ou  $\Sigma' = \{\text{crash}\}$ , où chaque symbole représente une propriété. Le fait est que le symbole choisi pour une propriété importe peu. La propriété « boguer » peut être symbolisée par les symboles *bogue*, *crash*, *p*, ou *x*. L'expression “ $\text{AG}\neg\text{bogue}$ ” est une  $\Sigma$ -expression bien formée de  $\text{Lang}_{\text{CTL}, \Sigma}$ , tandis que l'expression “ $\text{AG}\neg\text{crash}$ ” est une

3. Abréviation de l'anglais *linear-time logic*.

4. Abréviation de l'anglais *computation tree logic*.

$\Sigma'$ -expression bien formée de  $\text{Lang}_{\text{CTL}, \Sigma'}$ . Toute fonction  $\sigma \in \Sigma \rightarrow \Sigma'$  permettant, à l'instar de la fonction  $\sigma = \{ \text{bogue} \mapsto \text{crash} \}$ , de passer d'une signature  $\Sigma$  à une signature  $\Sigma'$ , est un *morphisme de signatures*. Les modèles doivent eux aussi tenir compte de la signature fixée pour la modélisation. Un  $\Sigma$ -modèle est un modèle construit sur la base de la signature  $\Sigma$ .

Une signature  $\Sigma \in \text{Sig}$ , un  $\Sigma$ -langage  $\text{Lang}_\Sigma$ , un ensemble d'exigences  $\Phi \subseteq \text{Lang}_\Sigma$ , et une classe de  $\Sigma$ -modèles  $\text{Mod}_\Sigma$  étant donnés, on cherche un  $\Sigma$ -modèle  $M \in \text{Mod}_\Sigma$  du futur logiciel tel que  $M$  satisfasse toutes les exigences dans  $\Phi$ , ce que l'on note “ $M \models \Phi$ ” (lire «  $M$  satisfait  $\Phi$  »). Une fois qu'un tel  $M$  est obtenu, il est préférable, afin d'éviter les erreurs de programmation, de générer ensuite automatiquement le code source du logiciel conforme à  $M$ . Pour cela, il faut que  $M$  soit suffisamment détaillé. Le processus d'ajouts successifs de détails et de précisions à  $M$  s'appelle le *raffinement de modèles*. Ce stage porte sur l'article [Bau+12], abordé à la section 1.2.2, qui en propose une formalisation mathématique via les systèmes de transitions modales à événements structurés. Toutes les classes de modèles mentionnées ci-dessus sont, en fait, des systèmes de transitions spécialisés. D'où l'importance de cette notion. La section 1.1.2 lui est consacrée.

### 1.1.1. Institution

La théorie des institutions [ST12] unifie dans un seul formalisme les idées que nous venons de présenter.

**Définition 1.1 — Institution.** Une *institution*

$$\langle \text{Sig}, \text{Lang}, \text{Mod}, \models \rangle$$

est la donnée :

1. d'une collection  $\text{Sig}$  de *signatures* munie, pour chaque couple de signatures  $\Sigma, \Sigma' \in \text{Sig}$ , d'une collection  $\text{Sig}(\Sigma, \Sigma')$  de *morphismes de signatures*;
2. pour chaque signature  $\Sigma \in \text{Sig}$ , d'un  $\Sigma$ -langage  $\text{Lang}_\Sigma$ , c.-à-d. d'un ensemble de  $\Sigma$ -expressions bien formées;
3. pour chaque morphisme de signatures  $\sigma \in \text{Sig}(\Sigma, \Sigma')$ , d'une fonction totale  $\sigma^T \in \text{Lang}_\Sigma \rightarrow \text{Lang}_{\Sigma'}$ , appelée  $\sigma$ -traducteur, qui traduit toute  $\Sigma$ -expression bien formée  $\varphi \in \text{Lang}_\Sigma$  en une  $\Sigma'$ -expression bien formée  $\sigma^T(\varphi) \in \text{Lang}_{\Sigma'}$ , que l'on nomme la  $\sigma$ -traduction de  $\varphi$ ;
4. pour chaque signature  $\Sigma \in \text{Sig}$ , d'une collection  $\text{Mod}_\Sigma$  de  $\Sigma$ -modèles;
5. pour chaque morphisme de signatures  $\sigma \in \text{Sig}(\Sigma, \Sigma')$ , d'une fonction totale  $\sigma^R \in \text{Mod}_{\Sigma'} \rightarrow \text{Mod}_\Sigma$ , appelée  $\sigma$ -réducteur, qui ramène tout  $\Sigma'$ -modèle  $M' \in \text{Mod}_{\Sigma'}$  à un  $\Sigma$ -modèle  $\sigma^R(M') \in \text{Mod}_\Sigma$  que l'on nomme la  $\sigma$ -réduction de  $M'$ ;
6. pour chaque signature  $\Sigma \in \text{Sig}$ , d'une relation  $\models_\Sigma \subseteq \text{Mod}_\Sigma \times \text{Lang}_\Sigma$ , dite de *satisfaction*, spécifiant, pour chaque  $\Sigma$ -expression bien formée  $\varphi \in \text{Lang}_\Sigma$ , quels sont les  $\Sigma$ -modèles  $M \in \text{Mod}_\Sigma$  qui *satisfont*  $\varphi$ . On écrit “ $M \models_\Sigma \varphi$ ”, ou “ $M \models \varphi$ ” en l'absence d'ambiguïté, plutôt que “ $(M, \varphi) \in \models_\Sigma$ ”, et on dit que  $M$  *satisfait*  $\varphi$ . Par extension, si  $\Phi \subseteq \text{Lang}_\Sigma$  est un ensemble de  $\Sigma$ -expressions bien formées, “ $M \models_\Sigma \Phi$ ”, ou “ $M \models \Phi$ ” en l'absence d'ambiguïté, signifie que pour tout  $\varphi \in \Phi$ ,  $M \models_\Sigma \varphi$ ;
7. d'une preuve que tout changement de notation préserve la relation de satisfaction — propriété qui porte le nom de *condition de satisfaction*. Cela

signifie qu'il faut que pour tout morphisme de signatures  $\sigma \in \text{Sig}(\Sigma, \Sigma')$ ,

$$\sigma^R(M') \models_{\Sigma} \varphi \text{ si et seulement si } M' \models_{\Sigma'} \sigma^T(\varphi),$$

où  $\varphi \in \text{Lang}_{\Sigma}$  est une  $\Sigma$ -expression bien formée, et  $M' \in \text{Mod}_{\Sigma'}$  est un  $\Sigma'$ -modèle. ■

Les deux problèmes majeurs de la modélisation logicielle sont la vérification de modèles et la satisfiabilité.

**Définition 1.2 — Problème de vérification de modèles.** Soit  $I$  une institution. Le problème consistant à trouver un algorithme prenant en entrée une signature  $\Sigma \in \text{Sig}_I$ , un  $\Sigma$ -modèle  $M \in \text{Mod}_{I,\Sigma}$ , et un ensemble  $\Phi \subseteq \text{Lang}_{I,\Sigma}$  de  $\Sigma$ -expressions bien formées, et capable de retourner “vrai” si  $M \models_{I,\Sigma} \Phi$ , et “faux” sinon, est le *problème de vérification de modèles*. Un tel algorithme, s'il existe, est appelé un *vérificateur de modèles*. ■

**Définition 1.3 — Problème de satisfiabilité.** Soit  $I$  une institution. Le problème consistant à trouver un algorithme prenant en entrée une signature  $\Sigma \in \text{Sig}_I$ , et un ensemble  $\Phi \subseteq \text{Lang}_{I,\Sigma}$  de  $\Sigma$ -expressions bien formées, et capable de retourner, s'il existe, un  $\Sigma$ -modèle  $M \in \text{Mod}_{I,\Sigma}$  tel que  $M \models_{I,\Sigma} \Phi$ , et “faux” sinon, est le *problème de satisfiabilité*. Un tel algorithme, s'il existe, est appelé un *solveur SAT*. ■

Avant de se lancer dans la recherche d'un modèle  $M$  satisfaisant le cahier des charges  $\Phi$  — recherche qui peut être automatisée si le problème de satisfiabilité est *calculable*, c.-à-d. résoluble — il faut d'abord analyser les implications logiques de  $\Phi$  pour y trouver peut-être des incohérences ou de la redondance. Pour cela, on adopte la terminologie suivante :

**Définition 1.4 — Implémentation.** Soient  $I$  une institution,  $\Sigma \in \text{Sig}_I$  une signature, et  $\Phi \subseteq \text{Lang}_{I,\Sigma}$  un ensemble de  $\Sigma$ -expressions bien formées. On dit qu'un  $\Sigma$ -modèle  $M \in \text{Mod}_{I,\Sigma}$  est une *implémentation* de  $\Phi$ , ou que  $M$  *implémente*  $\Phi$ , si et seulement si  $M$  satisfait  $\Phi$ . L'ensemble

$$\text{Imp}_{\Sigma} \Phi \triangleq \{ M \in \text{Mod}_{I,\Sigma} \mid M \models_{I,\Sigma} \Phi \} \subseteq \text{Mod}_{I,\Sigma}$$

est l'ensemble des implémentations de  $\Phi$ . ■

**Remarque 1.5.**  $\text{Imp}_{\Sigma} \emptyset = \text{Mod}_{I,\Sigma}$ . En effet,

$$\begin{aligned} \text{Imp}_{\Sigma} \emptyset &\triangleq \{ M \in \text{Mod}_{I,\Sigma} \mid M \models_{I,\Sigma} \emptyset \} \\ &= \{ M \in \text{Mod}_{I,\Sigma} \mid \forall \varphi \in \emptyset, M \models_{I,\Sigma} \varphi \} \\ &= \{ M \in \text{Mod}_{I,\Sigma} \mid \text{vrai} \} = \text{Mod}_{I,\Sigma}. \end{aligned}$$

**Définition 1.6 — Caractérisation.** Soient  $I$  une institution,  $\Sigma \in \text{Sig}_I$  une signature, et  $\Delta \subseteq \text{Mod}_{I,\Sigma}$  un ensemble de  $\Sigma$ -modèles. On dit qu'une  $\Sigma$ -expression bien formée  $\varphi \in \text{Lang}_{I,\Sigma}$  est une *caractérisation* de  $\Delta$ , ou que  $\varphi$  *caractérise*  $\Delta$ , si et seulement si  $\varphi$  est satisfaite par tous les  $\Sigma$ -modèles de  $\Delta$ . L'ensemble

$$\text{Car}_{\Sigma} \Delta \triangleq \{ \varphi \in \text{Lang}_{I,\Sigma} \mid \forall M \in \Delta, M \models_{I,\Sigma} \varphi \} \subseteq \text{Lang}_{I,\Sigma}$$

est l'ensemble des caractérisations de  $\Delta$ . ■

**Remarque 1.7.**  $\text{Car}_\Sigma \emptyset = \text{Lang}_{I,\Sigma}$ . En effet,

$$\begin{aligned}\text{Car}_\Sigma \emptyset &\triangleq \{ \varphi \in \text{Lang}_{I,\Sigma} \mid \forall M \in \emptyset, M \models_{I,\Sigma} \varphi \} \\ &= \{ \varphi \in \text{Lang}_{I,\Sigma} \mid \text{vrai} \} \\ &= \text{Lang}_{I,\Sigma}.\end{aligned}$$

**Définition 1.8 — Fermeture.** Soient  $I$  une institution,  $\Sigma \in \text{Sig}_I$  une signature,  $\Phi \subseteq \text{Lang}_{I,\Sigma}$  un ensemble de  $\Sigma$ -expressions bien formées, et  $\Delta \subseteq \text{Mod}_{I,\Sigma}$  un ensemble de  $\Sigma$ -modèles. La *fermeture* de  $\Phi$  est l'ensemble

$$\text{Ferm}_\Sigma \Phi \triangleq \text{Car}_\Sigma(\text{Imp}_\Sigma \Phi) \subseteq \text{Lang}_{I,\Sigma}$$

des caractérisations des implémentations de  $\Phi$ , c.-à-d. l'ensemble des  $\Sigma$ -expressions bien formées satisfaites par toutes les implémentations de  $\Phi$ . La *fermeture* de  $\Delta$  est l'ensemble

$$\text{Ferm}_\Sigma \Delta \triangleq \text{Imp}_\Sigma(\text{Car}_\Sigma \Delta) \subseteq \text{Mod}_{I,\Sigma}$$

des implémentations des caractérisations de  $\Delta$ , c.-à-d. l'ensemble des  $\Sigma$ -modèles qui satisfont toutes les caractérisations de  $\Delta$ .  $\Phi$  et  $\Delta$  sont respectivement dit *clos* si et seulement si  $\Phi = \text{Ferm}_\Sigma \Phi$  et  $\Delta = \text{Ferm}_\Sigma \Delta$ . ■

**Définition 1.9 — Conséquence sémantique.** Soient  $I$  une institution,  $\Sigma \in \text{Sig}_I$  une signature, et  $\Phi \subseteq \text{Lang}_{I,\Sigma}$  un ensemble de  $\Sigma$ -expressions bien formées. On dit qu'une  $\Sigma$ -expression bien formée  $\varphi \in \text{Lang}_{I,\Sigma}$  est une *conséquence sémantique* de  $\Phi$ , ou que  $\Phi$  *implique sémantiquement*  $\varphi$ , et on écrit " $\Phi \models_{I,\Sigma} \varphi$ ", ou " $\Phi \models \varphi$ " en l'absence d'ambiguïté, si et seulement si  $\varphi \in \text{Ferm}_\Sigma \Phi$ , c.-à-d. si et seulement si toute implémentation de  $\Phi$  satisfait  $\varphi$ , ou, de façon synonyme, si et seulement si tout  $\Sigma$ -modèle satisfaisant  $\Phi$  satisfait aussi  $\varphi$ , c.-à-d., en termes ensemblistes, si et seulement si  $\text{Imp}_\Sigma \Phi \subseteq \text{Imp}_\Sigma \{ \varphi \}$ . ■

La notion de conséquence sémantique est vraiment très importante en modélisation logicielle. Imaginons que  $\Phi$  soit le cahier des charges. La fermeture de  $\Phi$  est l'ensemble de toutes les propriétés du futur produit déductibles de  $\Phi$ . En effet, après avoir démontré que  $\Phi \models_{I,\Sigma} \varphi$ , c.-à-d. que  $\varphi \in \text{Ferm}_\Sigma \Phi$ , on est sûr que  $\varphi$  sera une caractéristique supplémentaire du logiciel avant même de l'avoir construit.

**Théorème 1.10.** Soient  $I$  une institution, et  $\Sigma \in \text{Sig}_I$  une signature. Pour tout ensemble  $\Phi \subseteq \text{Lang}_{I,\Sigma}$  de  $\Sigma$ -expressions bien formées, et toute  $\Sigma$ -expression bien formée  $\varphi \in \text{Lang}_{I,\Sigma}$ , les propriétés suivantes sont vraies :

1. réflexivité :  $\{ \varphi \} \models_{I,\Sigma} \varphi$ ;
2. affaiblissement : pour tout ensemble  $\Psi \subseteq \text{Lang}_{I,\Sigma}$  de  $\Sigma$ -expressions bien formées, si  $\Phi \models_{I,\Sigma} \varphi$  alors  $\Phi \cup \Psi \models_{I,\Sigma} \varphi$ ;
3. supposition : pour tout ensemble  $\Psi \subseteq \text{Lang}_{I,\Sigma}$  de  $\Sigma$ -expressions bien formées,  $\{ \varphi \} \cup \Psi \models_{I,\Sigma} \varphi$ ;
4. transitivité : pour toute famille  $(\Psi_i)_{i \in \Phi}$  d'ensembles de  $\Sigma$ -expressions bien formées indexée par  $\Phi$ , si  $\Psi_i \models_{I,\Sigma} i$  pour tout  $i \in \Phi$ , et si  $\Phi \models_{I,\Sigma} \varphi$ , alors  $\bigcup_{i \in \Phi} \Psi_i \models_{I,\Sigma} \varphi$ . ■

**Démonstration.** Facile. ■

Un problème classique est de chercher jusqu'à quel point on peut laisser l'ordinateur décider automatiquement si  $\Phi \models_{I,\Sigma} \varphi$ .

**Définition 1.11 — Problème de démonstration automatique.** Soit  $I$  une institution. Le problème consistant à trouver un algorithme prenant en entrée une signature  $\Sigma \in \text{Sig}_I$ , un ensemble  $\Phi \subseteq \text{Lang}_{I,\Sigma}$  de  $\Sigma$ -expressions bien formées, et une  $\Sigma$ -expression bien formée  $\varphi \in \text{Lang}_{I,\Sigma}$ , et capable de retourner une démonstration que  $\Phi \models_{I,\Sigma} \varphi$  si  $\Phi \models_{I,\Sigma} \varphi$ , et “faux” sinon, est le *problème de démonstration automatique*. Un tel algorithme, s’il existe, est appelé un *démonstrateur automatique*. ■

La  $\sigma$ -traduction des  $\Sigma$ -expressions bien formées, lors d’un changement de notation  $\sigma : \Sigma \rightarrow \Sigma'$ , préserve la conséquence sémantique.

**Théorème 1.12 —  $\sigma$ -invariance de  $\models$ .** Soient  $I$  une institution,  $\sigma \in \text{Sig}_I(\Sigma, \Sigma')$  un morphisme de signatures,  $\Phi \subseteq \text{Lang}_{I,\Sigma}$  un ensemble de  $\Sigma$ -expressions bien formées, et  $\varphi \in \text{Lang}_{I,\Sigma}$  une  $\Sigma$ -expression bien formée. Si  $\Phi \models_{I,\Sigma} \varphi$  alors  $\sigma^T(\Phi) \models_{I,\Sigma'} \sigma^T(\varphi)$ . ■

**Démonstration.** Supposons que  $\Phi \models_{I,\Sigma} \varphi$ . Montrons que  $\sigma^T(\Phi) \models_{I,\Sigma'} \sigma^T(\varphi)$ , c.-à-d. que toute implémentation de  $\sigma^T(\Phi)$  satisfait  $\sigma^T(\varphi)$ .

Soit  $M' \in \text{Imp}_{\Sigma'}(\sigma^T(\Phi))$  une implémentation de  $\sigma^T(\Phi)$ . Pour tout  $\phi \in \Phi$ ,  $M' \models_{I,\Sigma'} \sigma^T(\phi)$ . D’après la condition de satisfaction, cela implique que pour tout  $\phi \in \Phi$ ,  $\sigma^R(M') \models_{I,\Sigma} \phi$ , c.-à-d. que  $\sigma^R(M') \models_{I,\Sigma} \Phi$ . Or, par hypothèse,  $\Phi \models_{I,\Sigma} \varphi$ . Il s’ensuit, par la définition de la conséquence sémantique, que  $\sigma^R(M') \models_{I,\Sigma} \varphi$ . Et la condition de satisfaction nous amène à conclure que  $M' \models_{I,\Sigma'} \sigma^T(\varphi)$ . ■

**Définition 1.13 — Validité.** Soient  $I$  une institution, et  $\Sigma \in \text{Sig}_I$  une signature. On dit qu’une  $\Sigma$ -expression bien formée  $\varphi \in \text{Lang}_{I,\Sigma}$  est *valide*, et on écrit “ $\models_{I,\Sigma} \varphi$ ”, ou “ $\models \varphi$ ” en l’absence d’ambiguïté, si et seulement si  $\emptyset \models_{I,\Sigma} \varphi$ , c.-à-d. si et seulement si  $\varphi \in \text{Ferm}_\Sigma \emptyset \triangleq \text{Car}_\Sigma(\text{Imp}_\Sigma \emptyset) = \text{Car}_\Sigma \text{Mod}_{I,\Sigma}$ , ou, de façon synonyme, si et seulement si tout  $\Sigma$ -modèle  $M \in \text{Mod}_{I,\Sigma}$  satisfait  $\varphi$ . ■

**Définition 1.14 — Équivalence sémantique.** Soient  $I$  une institution, et  $\Sigma \in \text{Sig}_I$  une signature. On dit que deux  $\Sigma$ -expressions bien formées  $\varphi, \psi \in \text{Lang}_{I,\Sigma}$  sont *sémantiquement équivalentes*, ou que  $\varphi$  *équivaut sémantiquement* à  $\psi$ , et on écrit “ $\varphi \equiv_{I,\Sigma} \psi$ ”, ou “ $\varphi \equiv \psi$ ” en l’absence d’ambiguïté, si et seulement si  $\{\varphi\} \models_{I,\Sigma} \psi$  et  $\{\psi\} \models_{I,\Sigma} \varphi$ , ou, en termes ensemblistes, si et seulement si  $\text{Imp}_\Sigma \{\varphi\} \subseteq \text{Imp}_\Sigma \{\psi\}$  et  $\text{Imp}_\Sigma \{\psi\} \subseteq \text{Imp}_\Sigma \{\varphi\}$ , c.-à-d. si et seulement si  $\text{Imp}_\Sigma \{\varphi\} = \text{Imp}_\Sigma \{\psi\}$ , c.-à-d. si et seulement si  $\varphi$  et  $\psi$  ont exactement les mêmes implémentations. ■

Enfin, il est bon d’avoir un cahier des charges  $\Phi$  sans redondance, où aucune exigence n’est la conséquence sémantique des autres, et consistant, cohérent, sans contradiction.

**Définition 1.15 — Redondance.** Soient  $I$  une institution, et  $\Sigma \in \text{Sig}_I$  une signature. Un ensemble  $\Phi \subseteq \text{Lang}_{I,\Sigma}$  de  $\Sigma$ -expressions bien formées est *redondant* si et seulement s’il existe  $\varphi \in \Phi$  tel que  $\Phi - \{\varphi\} \models_{I,\Sigma} \varphi$ . ■

**Définition 1.16 — Consistance, cohérence.** Soient  $I$  une institution, et  $\Sigma \in \text{Sig}_I$  une signature. Un ensemble  $\Phi \subseteq \text{Lang}_{I,\Sigma}$  de  $\Sigma$ -expressions bien formées est *consistant* ou *cohérent* si et seulement si  $\text{Imp}_\Sigma \varphi \neq \emptyset$ , c.-à-d. si et seulement s’il est possible de construire un modèle qui le satisfait. Dans le cas contraire,  $\Phi$  est *inconsistant* ou *incohérent*. ■

Dans la suite, nous nous focaliserons sur une classe de modèles importante : celle des systèmes de transitions.

### 1.1.2. Systèmes de transitions

La théorie des systèmes de transitions est devenue un outil standard pour modéliser les comportements d'un logiciel.

**Définition 1.17 — Système de transitions.** Un *système de transitions*

$$\langle \mathcal{Q}, \mathcal{I}, \mathcal{E}, \mathcal{T}, \alpha, \beta, \lambda \rangle$$

est un objet dont l'*état* est modifié par des *événements* :

1.  $\mathcal{Q}$  est un ensemble d'*états* ;
2.  $\mathcal{I} \subseteq \mathcal{Q}$  est le sous-ensemble des états *initiaux*, c.-à-d. desquels le système peut démarrer ;
3.  $\mathcal{E}$  est un ensemble d'*événements* ;
4.  $\mathcal{T}$  est un ensemble de *transitions* ;
5.  $\alpha, \beta \in \mathcal{T} \rightarrow \mathcal{Q}$  et  $\lambda \in \mathcal{T} \rightarrow \mathcal{E}$  sont des fonctions totales qui associent à chaque transition  $t \in \mathcal{T}$ , une *source*  $\alpha(t) \in \mathcal{Q}$ , une *cible*  $\beta(t) \in \mathcal{Q}$ , et un événement  $\lambda(t) \in \mathcal{E}$ , respectivement.

Une transition  $t \in \mathcal{T}$  indique que l'événement  $\lambda(t) \in \mathcal{E}$  fait passer le système à l'état  $\beta(t) \in \mathcal{Q}$  s'il survient tandis que celui-ci est dans l'état  $\alpha(t) \in \mathcal{Q}$ . ■

Cette définition, tirée de [Arn92], est plus adaptée à la programmation que celle qu'on présente souvent aux étudiants, et qui consiste à prendre  $\mathcal{T}$  tel que  $\mathcal{T} \subseteq \mathcal{Q} \times \mathcal{E} \times \mathcal{Q}$ , et à poser, pour toute transition  $(q, e, q') \in \mathcal{T}$ ,  $\alpha(q, e, q') \triangleq q$ ,  $\beta(q, e, q') \triangleq q'$ , et  $\lambda(q, e, q') \triangleq e$ . En effet, dans cette définition, l'ensemble  $\mathcal{T}$  est, en quelque sorte, un type abstrait qui a pour interface les fonctions  $\alpha$ ,  $\beta$ , et  $\lambda$ . Nous la choisissons en vue de son implémentation en Coq. Toutefois, pour les exemples ci-dessous, nous prendrons  $\mathcal{T}$  tel que  $\mathcal{T} \subseteq \mathcal{Q} \times \mathcal{E} \times \mathcal{Q}$ .

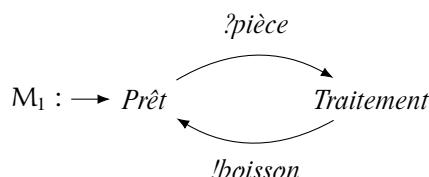


FIGURE 1.1 — Modélisation d'un distributeur automatique de boissons.

**Exemple 1.18 — Distributeur automatique de boissons.** Le système de transitions  $M_1$  de la figure 1.1 modélise le comportement d'un logiciel contrôlant un distributeur automatique de boissons. Il fait que la machine délivre une boisson toutes les fois qu'une pièce est introduite. Les éléments de  $\mathcal{Q}_{M_1}$  sont les états *Prêt* et *Traitement*. L'unique état initial de  $\mathcal{I}_{M_1}$ , *Prêt*, est indiqué par une flèche entrante sans source. Les éléments de  $\mathcal{E}_{M_1}$  sont les événements *?pièce* et *!boisson*. Les flèches étiquetées par les événements représentent les transitions de  $\mathcal{T}_{M_1}$ . La transition (*Prêt*, *?pièce*, *Traitement*) survient au moment où le logiciel détecte l'introduction d'une pièce, et la transition (*Traitement*, *!boisson*, *Prêt*) au moment où la boisson est délivrée. Par convention, les événements initiés par le logiciel commencent par un point d'exclamation, et ceux qu'il subit par un point d'interrogation. ■

Souvent, les systèmes de transitions sont annotés par des propriétés rédigées dans le langage d'une institution.

**Définition 1.19 — Système de transitions annoté.** Soient  $I$  une institution, et  $\Sigma \in \text{Sig}_I$  une signature. Un *système de transitions annoté sur*  $(I, \Sigma)$ , ou *système de transitions*  $(I, \Sigma)$ -*annoté*,

$$\langle Q, I, E, T, \alpha, \beta, \lambda, \langle - \rangle \rangle,$$

est un système de transition  $\langle Q, I, E, T, \alpha, \beta, \lambda \rangle$  muni d'une *fonction d'annotation totale*  $\langle - \rangle \in Q \rightarrow g(\text{Lang}_{I, \Sigma})$  qui associe à chaque état  $q \in Q$  une annotation  $\langle q \rangle \subseteq \text{Lang}_{I, \Sigma}$ , c.-à-d. un ensemble de  $\Sigma$ -expressions bien formées. ■

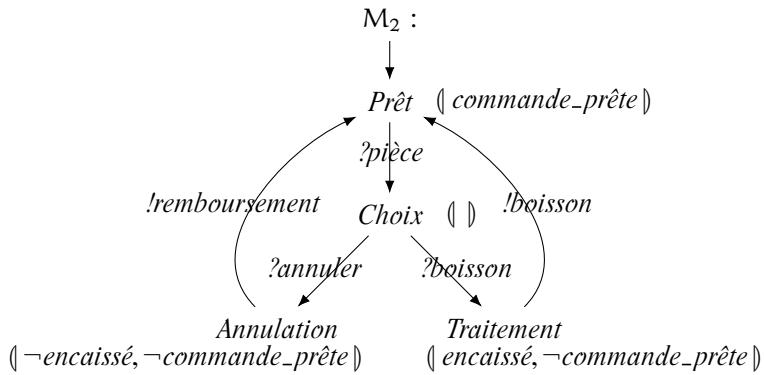


FIGURE 1.2 — Système de transitions annoté d'un distributeur automatique de boissons.

**Exemple 1.20 — Distributeur automatique de boissons (continuation).** Sur la figure 1.2, nous avons annoté sur  $(\text{Prop}, \Sigma)$ , où  $\text{Prop}$  est l'institution de la logique propositionnelle, et  $\Sigma = \{ \text{encaissé}, \text{commande-prête} \}$  est la signature, un système de transitions  $M_2$  pour un nouveau distributeur automatique de boissons. Les noms des états, des événements, et des transitions sont plutôt clairs. Nous commenterons seulement les annotations, lesquelles sont rédigées dans le  $\Sigma$ -langage propositionnel  $\text{Lang}_{\text{Prop}, \Sigma}$ . On écrit “ $\langle \Phi \rangle$ ” à côté de l'état  $q$  — en omettant les accolades — pour indiquer que  $\langle q \rangle_{M_2} = \Phi$ . Par exemple,

$$\begin{aligned} \langle \text{Prêt} \rangle_{M_2} &= \{ \text{commande-prête} \}, \\ \langle \text{Choix} \rangle_{M_2} &= \emptyset, \\ \langle \text{Annulation} \rangle_{M_2} &= \{ \neg \text{encaissé}, \neg \text{commande-prête} \}, \\ \text{et } \langle \text{Treatment} \rangle_{M_2} &= \{ \text{encaissé}, \neg \text{commande-prête} \}. \end{aligned}$$

Intuitivement, *encaissé* dénote la proposition : « La machine a encaissé la pièce introduite par l'utilisateur » (au sens où cet argent ne lui sera jamais rendu), et *boisson-prête* la proposition : « La machine a délivré la boisson commandée par l'utilisateur » (pour faire simple, on considère, même si c'est un peu limite, qu'une machine neuve a délivré une commande). Nous faisons l'hypothèse que le lecteur connaît la logique propositionnelle, ses conséquences et équivalences sémantiques, et ses  $\Sigma$ -expressions bien formées valides. ■

**Définition 1.21 — Satisfaction d'état.** Soient  $I$  une institution,  $\Sigma \in \text{Sig}_I$  une signature,  $M$  un système de transitions  $(I, \Sigma)$ -annoté, et  $\varphi \in \text{Lang}_{I, \Sigma}$  une  $\Sigma$ -expression bien formée. On dit qu'un état  $q \in Q_M$  *satisfait*  $\varphi$ , et on écrit “ $q \models_M \varphi$ ” — veuillez noter l'indice “ $M$ ” pour bien faire la différence avec la relation de satisfaction de l'institution  $I$  —, ou “ $q \models \varphi$ ” en l'absence d'ambiguïté, si et seulement si  $(q)_{\mathcal{M}} \models_{I, \Sigma} \varphi$ . ■

**Exemple 1.22 — Distributeur automatique de boissons (continuation).** Dans le système de transitions  $(\text{Prop}, \Sigma)$ -annoté  $M_2$  de la figure 1.2, on a

*Traitemen*t  $\models_{M_2}$  *encaissé*,  
et *Annulation*  $\models_{M_2} \neg \text{encaissé} \wedge \neg \text{commande\_prête}$ .

De plus, puisque  $(Choix)_{M_2} = \emptyset$ , l'état *Choix* satisfait uniquement les  $\Sigma$ -expressions propositionnelles bien formées  $\varphi$  qui sont valides. Par exemple *encaissé*  $\vee \neg \text{encaissé}$ . En effet,  $\emptyset \models_{\text{Prop}, \Sigma} \varphi$  signifie que  $\models_{\text{Prop}, \Sigma} \varphi$ . ■

Il y a plusieurs langages mathématiques pour exprimer les exigences que doivent satisfaire un système de transitions annoté. Par exemple, ceux de la logique du temps linéaire LTL<sup>5</sup>, de la logique d'Hennessy-Milner HML<sup>6</sup>, et de la logique du temps arborescent CTL<sup>7</sup>. Nous suivons [Bau+12] et présentons seulement HML.

**Définition 1.23 — Logique d'Hennessy-Milner HML.** Pour toute institution  $I$  — procurant un langage pour annoter les états des systèmes de transitions —, l'institution  $HML(I)$  de la logique d'Hennessy-Milner est définie ainsi :

1.  $\text{Sig}_{HML(I)} \triangleq \text{Ens} \times \text{Sig}_I$  est la classe des couples  $(E, \Sigma)$ , où  $E \in \text{Ens}$  est un ensemble dont les éléments sont appelés *événements*, et  $\Sigma \in \text{Sig}_I$  est une signature de l'institution  $I$ . Pour chaque couple de signatures  $(E, \Sigma), (E', \Sigma') \in \text{Sig}_{HML(I)}$ , les morphismes de signatures  $\sigma \in \text{Sig}_{HML(I)}((E, \Sigma), (E', \Sigma'))$  sont les couples de la forme  $\sigma = (\sigma_E, \sigma_\Sigma)$ , où  $\sigma_E \in E \rightarrow E'$  est une fonction totale, et  $\sigma_\Sigma \in \Sigma \rightarrow \Sigma'$  un morphisme de signatures de l'institution  $I$ ;
2. pour chaque signature  $(E, \Sigma) \in \text{Sig}_{HML(I)}$ , le langage  $\text{Lang}_{HML(I), (E, \Sigma)}$  est le plus petit ensemble contenant toutes les  $\Sigma$ -expressions bien formées du  $\Sigma$ -langage  $\text{Lang}_{I, \Sigma}$  de l'institution  $I$  — pour annoter les états des systèmes de transitions —, et qui est tel que :
  - si  $\varphi, \psi \in \text{Lang}_{HML(I), (E, \Sigma)}$  sont des  $(E, \Sigma)$ -expressions bien formées, alors  $\varphi \wedge \psi, \varphi \vee \psi \in \text{Lang}_{HML(I), (E, \Sigma)}$  (lire respectivement «  $\varphi$  et  $\psi$  » et «  $\varphi$  ou  $\psi$  ») le sont aussi;
  - si  $e \in E$  est un évènement, et  $\varphi \in \text{Lang}_{HML(I), (E, \Sigma)}$  une  $(E, \Sigma)$ -expression bien formée, alors  $\langle e \rangle \varphi \in \text{Lang}_{HML(I), (E, \Sigma)}$  l'est aussi;
  - si  $e \in E$  est un évènement, et  $\varphi \in \text{Lang}_{HML(I), (E, \Sigma)}$  une  $(E, \Sigma)$ -expression bien formée, alors  $[e] \varphi \in \text{Lang}_{HML(I), (E, \Sigma)}$  l'est aussi;
3. pour chaque signature  $(E, \Sigma) \in \text{Sig}_{HML(I)}$ , les  $(E, \Sigma)$ -modèles de  $\text{Mod}_{HML(I), (E, \Sigma)}$  sont les systèmes de transitions  $(I, \Sigma)$ -annotés  $M$  dont l'ensemble des évènements est  $\mathcal{E}_M = E$ ;

5. Abréviation de l'anglais *linear-time logic*.

6. Abréviation de l'anglais *Hennessy-Milner logic*.

7. Abréviation de l'anglais *computation tree logic*.

4. nous étendons la relation de satisfaction d'état de la définition 1.21 à toutes les  $(E, \Sigma)$ -expressions bien formées de la logique d'Hennessy-Milner. Nous avons déjà, pour toute  $\Sigma$ -expression bien formée  $\varphi$  du  $\Sigma$ -langage  $\text{Lang}_{I, \Sigma}$  de l'institution I, que  $q \models_M \varphi$  si et seulement si  $(\| q \|)_M \models_{I, \Sigma} \varphi$ . Par extension, nous posons que :

- pour toutes  $(E, \Sigma)$ -expressions bien formées  $\varphi, \psi \in \text{Lang}_{\text{HML}(I), (E, \Sigma)}$ ,  $q \models_M \varphi \wedge \psi$  si et seulement si  $q \models_M \varphi$  et  $q \models_M \psi$ ;
- pour toutes  $(E, \Sigma)$ -expressions bien formées  $\varphi, \psi \in \text{Lang}_{\text{HML}(I), (E, \Sigma)}$ ,  $q \models_M \varphi \vee \psi$  si et seulement si  $q \models_M \varphi$  ou  $q \models_M \psi$ ;
- pour tout évènement  $e \in E$  et toute  $(E, \Sigma)$ -expression bien formée  $\varphi \in \text{Lang}_{\text{HML}(I), (E, \Sigma)}$ ,  $q \models_M \langle e \rangle \varphi$  si et seulement s'il existe une transition  $(q, e, q') \in \rightarrow_M$  telle que  $q' \models_M \varphi$ ;
- pour tout évènement  $e \in E$  et toute  $(E, \Sigma)$ -expression bien formée  $\varphi \in \text{Lang}_{\text{HML}(I), (E, \Sigma)}$ ,  $q \models_M [e] \varphi$  si et seulement si quelle que soit la transition  $(q, e, q') \in \rightarrow_M$ , on a  $q' \models_M \varphi$ .

Un  $(E, \Sigma)$ -modèle  $M \in \text{Mod}_{\text{HML}(I), (E, \Sigma)}$  satisfait ou implémente une  $(E, \Sigma)$ -expression bien formée  $\varphi \in \text{Lang}_{\text{HML}(I), (E, \Sigma)}$  si et seulement si tous ses états initiaux satisfont  $\varphi : M \models_{\text{HML}(I), (E, \Sigma)} \varphi$  si et seulement si pour tout  $q_0 \in I_M$ ,  $q_0 \models_M \varphi$ . ■

**Exemple 1.24 — Distributeur automatique de boissons (continuation).** Revenons sur le système de transitions annoté  $M_2$  de la figure 1.2. On se place dans l'institution HML(Prop). On vérifie sans peine que

$$M_2 \models_{\text{HML}(\text{Prop}), (E, \Sigma)} \langle ?pièce \rangle [?annuler] \neg encaissé. ■$$

Nous venons de voir l'essentiel de ce que les étudiants de l'Université de Bordeaux apprennent en maîtrise de vérification logicielle. Parlons maintenant, si le lecteur le veut bien, des nouvelles connaissances acquises pendant le stage.

## 1.2. Développement à base de composants

La problématique générale qui nous préoccupait durant le stage était le développement à base de composants certifiés. Il s'agit d'une approche formalisée mathématiquement dans [Ben+12a] par la métathéorie des contrats. La théorie des systèmes de transitions modales à évènements structurés, que je devais implémenter en Coq, est une instance de cette théorie, d'où son nom de *métathéorie*.

### 1.2.1. Assemblage et remplacement de composants

Un composant est un objet qui procure des services — ses *garanties* — aussi longtemps que ses *prérequis* sont respectés. La figure 1.3 représente, en utilisant la notation UML<sup>8</sup>, un composant  $M$  avec ses prérequis  $P_1$  et  $P_2$ , et ses garanties  $G_1$  et  $G_2$ . En général, le concepteur du composant cache le comment, c.-à-d. son fonctionnement interne, et, par là même, le pourquoi, c.-à-d. la raison pour laquelle le respect des prérequis permet toujours de jouir des garanties. Pour l'utilisateur, le quoi est suffisant, c.-à-d. la donnée des prérequis et des garanties.

8. Abréviation de l'anglais *Unified Modeling Language*.

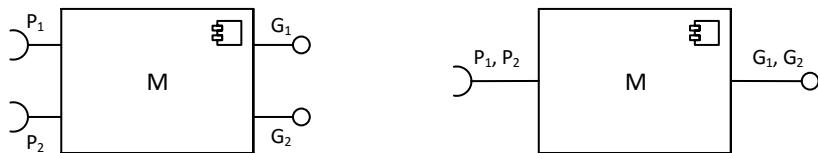


FIGURE 1.3 — Représentation d'un composant  $M$  en UML qui garantit  $G_1$  et  $G_2$  sous réserve des prérequis  $P_1$  et  $P_2$ .

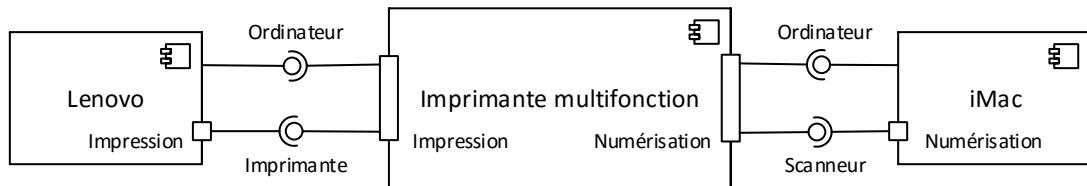


FIGURE 1.4 — Imprimante multifonction.

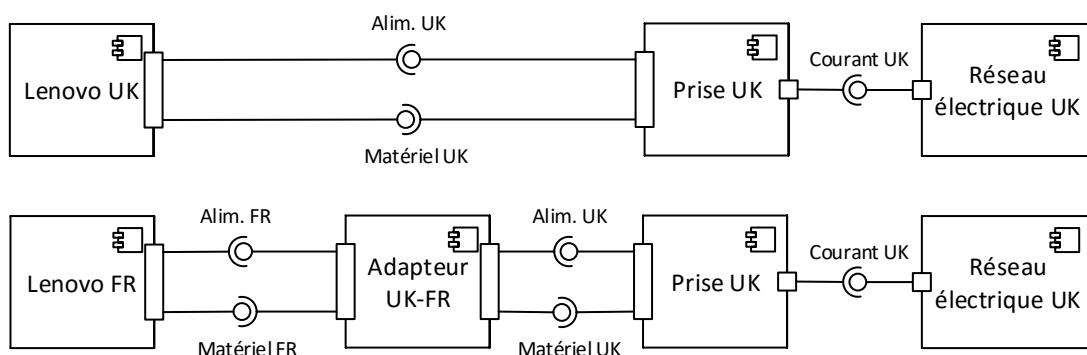


FIGURE 1.5 — Remplacement d'un ordinateur.

Un composant peut très bien proposer plusieurs services, chacun avec ses propres prérequis et garanties. La figure 1.4 montre, par exemple, une imprimante multifonction qui propose les services d'impression et de numérisation. Chaque service est symbolisé par un rectangle sur le bord du composant. Tout composant qui satisfait le prérequis d'être un ordinateur peut bénéficier, s'il a été conçu pour, des services d'impression et de numérisation de l'imprimante. Un composant peut remplacer, pour un service donné, un autre composant s'ils satisfont, pour ce service, les mêmes prérequis et les mêmes garanties. Sur la figure 1.5, un ordinateur britannique est remplacé, au Royaume-Uni, par un ordinateur français grâce à un adaptateur électrique.

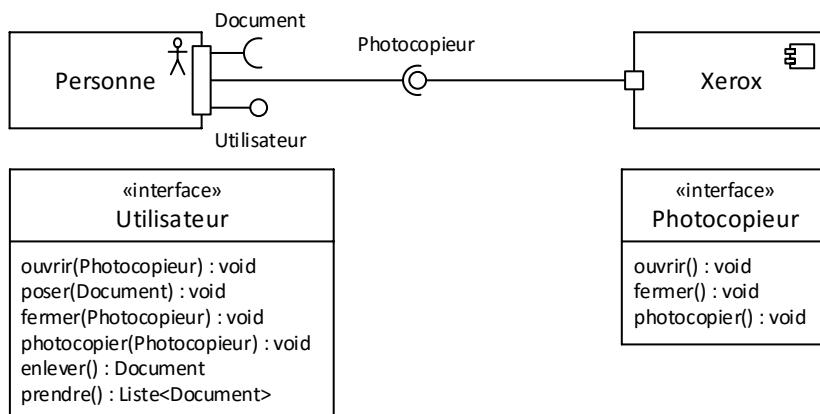


FIGURE 1.6 — Spécification statique des prérequis/garanties.

Les prérequis et garanties d'un service peuvent être donnés statiquement sous forme de listes d'opérations avec, éventuellement, des préconditions et des postconditions. Les opérations prérequisites sont alors celles dont le fournisseur du service a besoin pour implémenter les opérations garanties. Par exemple, sur la figure 1.6, pour qu'une personne puisse utiliser un photocopieur, il faut que celui-ci lui permette d'ouvrir le capot, le fermer, et faire des photocopies. De son côté, en tant qu'utilisateur, il pose un document sous le capot et, après l'avoir photocopier, l'enlève, et prend ses photocopies. Une spécification dynamique sous forme de système de transitions montre comment les opérations doivent s'enchaîner. Sur la figure 1.7, on peut voir deux systèmes de transitions  $U$  et  $P$ , le premier décrivant le comportement d'un utilisateur, et le second le mode d'emploi du photocopieur. Sur la figure 1.8, le système de transitions  $U \circledast P$  dépeint le comportement du photocopieur en présence de l'utilisateur. Dans les couples qui apparaissent au niveau des états et des transitions, la première composante correspond à l'utilisateur, et la seconde au photocopieur. L'événement  $e$  marque l'inaction. On peut constater que, mis à part les noms des états et des événements,  $U$  et  $U \circledast P$  sont les mêmes systèmes de transitions : l'utilisateur maîtrise le photocopieur.

Ce mémoire serait incomplet s'il ne mentionnait pas la belle et prometteuse métathéorie des contrats [Ben+12b ; Ben+12a] qui m'a été présentée par Jean-Baptiste Raclet, maître de conférences à l'Institut de recherche en informatique de Toulouse (IRIT), et coauteur de [Ben+12a]. Elle est une formalisation ma-

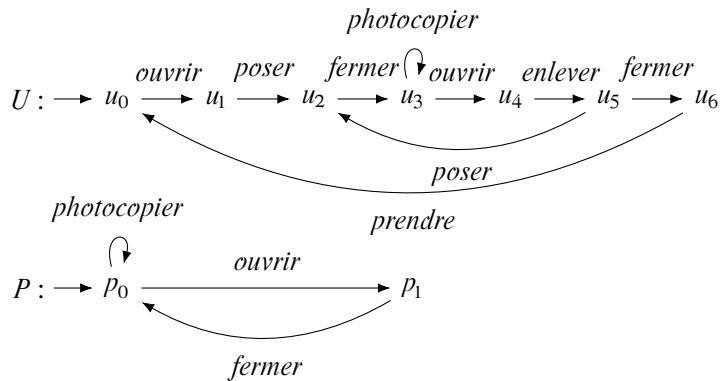


FIGURE 1.7 — Spécification dynamique des prérequis/garanties.

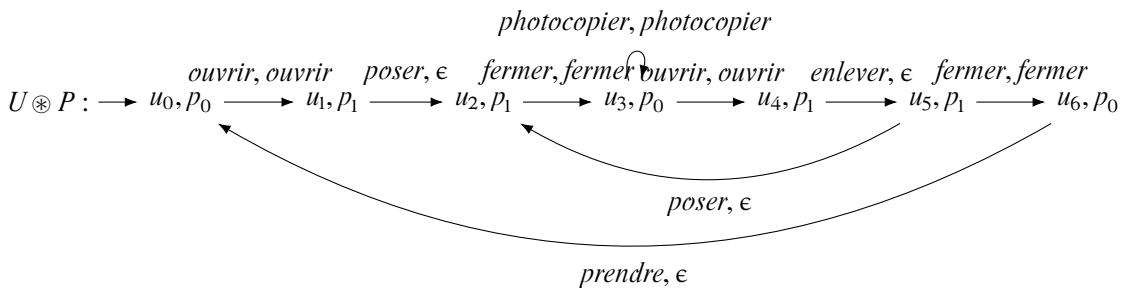


FIGURE 1.8 — Comportement du photocopieur en présence de l'utilisateur.

thématique du développement à base de composants. Les bases de la notion d'ensemble ordonné, qui revient régulièrement dans la suite du texte, sont rappelées dans l'appendice A; laquelle fixe aussi les notations employées ci-après.

### 1.2.2. Métathéorie des contrats

La métathéorie des contrats spécifie ce que doit contenir une théorie des contrats. Intuitivement, une théorie des contrats doit définir le concept de contrat, indiquer la nature des possibles réalisations d'un contrat, dire comment produire de nouvelles réalisations à partir de réalisations existantes, et préciser sous quelles conditions une réalisation satisfait ou implémenté un contrat.

**Définition 1.25 — Théorie des contrats.** Une *théorie* ou *algèbre des contrats*

$$\langle \mathcal{C}, \mathcal{R}, \equiv, \mathcal{L}, *, \models \rangle$$

est la donnée :

1. d'un ensemble  $\mathcal{C}$  de *contrats*;
2. d'un ensemble partitionné ou setoïde  $\langle \mathcal{R}, \equiv \rangle$  de *réalisations*;
3. d'un sous-ensemble  $\mathcal{L} \subseteq \mathcal{R}$  de réalisations dites *légales*;
4. d'une opération partielle  $* \in \mathcal{R} \times \mathcal{R} \rightarrow \mathcal{R}$  de *composition* de réalisations,  $\equiv$ -compatible — c.-à-d. telle que pour tous  $M_1, M_2, M \in \mathcal{R}$ ,  $M_1 \equiv M_2$  implique

que  $M_1 * M$  est défini si et seulement si  $M_2 * M$  est défini, et s'ils sont définis alors  $M_1 * M \equiv M_2 * M$  —,  $\equiv$ -commutative — c.-à-d. telle que pour tous  $M_1, M_2 \in \mathcal{R}$ ,  $M_1 * M_2$  est défini si et seulement si  $M_2 * M_1$  est défini, et s'ils sont définis alors  $M_1 * M_2 \equiv M_2 * M_1$  —, et  $\equiv$ -associative — c.-à-d. telle que pour tous  $M_1, M_2, M_3 \in \mathcal{R}$ ,  $M_1 * (M_2 * M_3)$  est défini si et seulement si  $(M_1 * M_2) * M_3$  est défini, et s'ils sont définis alors  $M_1 * (M_2 * M_3) \equiv (M_1 * M_2) * M_3$  —, pour construire, quand cela peut se faire, de nouvelles réalisations;

5. d'une relation  $\models \subseteq \mathcal{R} \times \mathcal{C}$ , dite *de satisfaction* des contrats par les réalisations,  $\equiv$ -compatible — c.-à-d. telle que pour tous  $M_1, M_2 \in \mathcal{R}$  et  $\varphi \in \mathcal{C}$ ,  $M_1 \equiv M_2$  et  $M_1 \models \varphi$  impliquent  $M_2 \models \varphi$ . On écrit " $M \models \varphi$ " plutôt que " $(M, \varphi) \in \models$ ", et on dit que la réalisation  $M \in \mathcal{R}$  *satisfait* le contrat  $\varphi \in \mathcal{C}$ . ■

Les deux avantages de la métathéorie des contrats par rapport à la théorie des institutions sont, d'une part, l'absence des complications dues aux morphismes de signatures et à la condition de satisfaction, et, d'autre part, la présence explicite d'une opération permettant de produire de nouvelles réalisations à partir de réalisations connues. On retrouve la notion d'implémentation :

**Définition 1.26 — Implémentation.** Soient  $T$  une théorie des contrats, et  $\varphi \in \mathcal{C}_T$  un contrat. On dit qu'une réalisation  $M \in \mathcal{R}_T$  est une *implémentation* de  $\varphi$ , ou que  $M$  *implémente*  $\varphi$ , si et seulement si  $M \models_T \varphi$ . L'ensemble

$$\text{Imp } \varphi \triangleq \{ M \in \mathcal{R}_T \mid M \models_T \varphi \}$$

est l'ensemble des implémentations de  $\varphi$ . ■

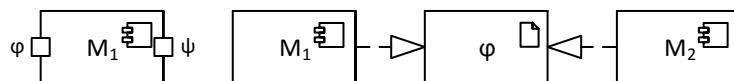


FIGURE 1.9 — Contrats et composants. La flèche en pointillés à tête triangulaire symbolise la relation d'implémentation.

Un composant est une réalisation, et ses services sont les contrats qu'il implémente. À chaque contrat correspond un ensemble d'implémentations remplaçables les unes par les autres. Sur la figure 1.9, la réalisation  $M_1$  implémente les contrats  $\varphi$  et  $\psi$ . Ces contrats décrivent les services rendus par  $M_1$  avec leurs prérequis et garanties. Comme  $M_1$  et  $M_2$  implémentent tous les deux  $\varphi$ , l'un peut se substituer à l'autre pour le service représenté par  $\varphi$ .

En suivant l'intuition que suggère les appellations *contrat*, *réalisation*, et *implémentation*, nous pouvons déjà identifier des concepts importants : la consistance, le raffinement, la conjonction, la composition, le quotient, et la compatibilité de contrats.

**Définition 1.27 — Consistance, cohérence.** Soit  $T$  une théorie des contrats. Un contrat  $\varphi \in \mathcal{C}_T$  est *consistant* ou *cohérent* si et seulement si  $\text{Imp } \varphi \neq \emptyset$ , c.-à-d. si et seulement s'il est possible de réaliser une implémentation qui le satisfait. Dans le cas contraire, le contrat est *inconsistant* ou *incohérent*. ■

**Définition 1.28 — Raffinement.** Soient  $T$  une théorie des contrats, et  $\varphi, \psi \in \mathcal{C}_T$  deux contrats. On dit que  $\varphi$  est un *raffinement* de  $\psi$ , ou que  $\varphi$  *raffine*  $\psi$ , et on écrit “ $\varphi \leq \psi$ ”, si et seulement si  $\text{Imp } \varphi \subseteq \text{Imp } \psi$ . Cela signifie que  $\varphi$  est un contrat plus précis ou plus restrictif que  $\psi$ , au sens où il élimine une partie des implémentations de  $\psi$ . ■

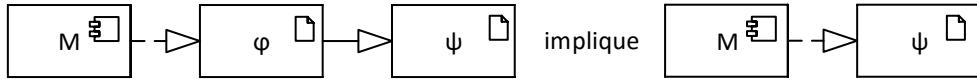


FIGURE 1.10 — Raffinement de contrats. La flèche en trait plein à tête triangulaire symbolise la relation de raffinement.

La relation de raffinement  $\leq$  hérrite de la relation d'inclusion  $\subseteq$  son caractère d'ordre partiel sur les contrats.

**Définition 1.29 — Conjonction.** Soient  $T$  une théorie des contrats, et  $\varphi, \psi \in \mathcal{C}_T$  deux contrats. S'il existe, le contrat

$$\varphi \wedge \psi \triangleq \inf\{\varphi, \psi\}$$

est la *conjonction* de  $\varphi$  et  $\psi$ . ■

D'après la définition de la borne inférieure,

$$\varphi \wedge \psi = \{\gamma \in \mathcal{C}_T \mid \gamma \leq \varphi \text{ et } \gamma \leq \psi\}^\top.$$

Autrement dit, la conjonction de  $\varphi$  et  $\psi$  est, s'il existe, le contrat qui, parmi tous ceux qui raffinent à la fois  $\varphi$  et  $\psi$ , est le moins restrictif, c.-à-d. celui qui a le plus grand nombre d'implémentations qui satisfont à la fois  $\varphi$  et  $\psi$ . La conjonction formalise la pratique qui consiste à renforcer un contrat par de nouvelles exigences, ou à permettre à ses implémentations de fournir plusieurs services.



FIGURE 1.11 — Conjonction de contrats.

**Définition 1.30 — Composition de contrats.** Soient  $T$  une théorie des contrats, et  $\varphi, \psi \in \mathcal{C}_T$  deux contrats. S'il existe, le contrat

$$\varphi \circledast \psi \triangleq \{\gamma \in \mathcal{C}_T \mid \forall M \in \text{Imp } \gamma, \exists M_\varphi \in \text{Imp } \varphi, \exists M_\psi \in \text{Imp } \psi, M = M_\varphi * M_\psi\}^\top$$

est la *composition* de  $\varphi$  et  $\psi$ . ■



FIGURE 1.12 — Composition de contrats.

En d'autres termes, la composition de  $\varphi$  et  $\psi$  est, s'il existe, le contrat qui a le plus grand nombre d'implémentations qui résultent d'une composition de deux réalisations  $M_\varphi \in \text{Imp } \varphi$  et  $M_\psi \in \text{Imp } \psi$ . La composition formalise l'assemblage de composants.

**Définition 1.31 — Quotient.** Soient  $T$  une théorie des contrats, et  $\varphi, \psi \in \mathcal{C}_T$  deux contrats. S'il existe, le contrat

$$\varphi \div \psi \triangleq \{\gamma \in \mathcal{C}_T \mid \gamma \circledast \psi \leqslant \varphi\}^\top$$

est le *quotient* de  $\varphi$  et  $\psi$ . ■

Le quotient de  $\varphi$  et  $\psi$  est le plus grand ensemble d'implémentations qui, lorsqu'elles sont composées avec une implémentation de  $\psi$ , produisent une implémentation de  $\varphi$  (figure 1.13). Le quotient formalise la pratique bien connue de décomposer le problème décrit par  $\varphi$  en deux sous-problèmes plus simples  $\psi$  et  $\varphi \div \psi$ , de les résoudre séparément, puis de produire, à partir des deux sous-solutions obtenues, une solution globale pour  $\varphi$ .



FIGURE 1.13 — Quotient de contrats.

Toute théorie des contrats distingue les réalisations légales à ses yeux de celles qu'elle juge illégales. Cette séparation est utile pour définir la compatibilité.

**Définition 1.32 — Compatibilité.** Soient  $T$  une théorie des contrats, et  $M, E \in \mathcal{R}_T$  deux réalisations.  $E$  est un *environnement compatible* pour  $M$  si et seulement si  $M * E \in \mathcal{L}_T$ , c.-à-d. si et seulement si  $M * E$  est légale. De même, si  $\varphi, \psi \in \mathcal{C}_T$  sont des contrats,  $\psi$  est *compatible* avec  $\varphi$  si et seulement si pour toute implémentation  $M_\varphi \in \text{Imp } \varphi$  de  $\varphi$ , il existe un environnement compatible  $E_\psi \in \text{Imp } \psi$ . ■

Il faut peut-être clarifier le lien entre la modélisation logicielle, le développement à base de composants, et la métathéorie des contrats. Nous avons vu que le but de la modélisation logicielle est de guider les constructeurs pour qu'ils construisent un logiciel conforme aux exigences du cahier des charges. Le logiciel est le composant, et le cahier des charges son contrat. L'idée est la suivante. Au lieu de passer par un langage mathématique, on peut donner le contrat directement sous forme de modèle. Au départ, il est général et abstrait.

Puis, suite à des raffinements successifs, il devient de plus en plus détaillé. L'objectif est d'arriver à un modèle qui précise tous les choix d'implémentation, afin de pouvoir générer automatiquement, si possible, le code source du composant. Une classe de modèles qui s'inscrit dans cette approche est celle des systèmes de transitions modales à évènements structurés.

### 1.2.3. Systèmes de transitions modales à évènements structurés

La théorie des systèmes de transitions modales à évènements structurés [Bau+12] est une théorie des contrats. Nous ne parlerons ici que des aspects liés au raffinement. La conjonction, le produit, et le quotient nous ammenerait beaucoup trop loin pour un rapport de stage. Le lecteur intéressé par ces sujets pourra consulter [Bau+12]. Commençons par expliquer ce que sont des évènements structurés.

**Définition 1.33 — Ensemble structuré d'étiquettes.** Un *ensemble structuré d'étiquettes*

$$\langle \mathcal{E}, \preccurlyeq, \perp \rangle$$

est la donnée :

1. d'un ensemble partiellement ordonné d'étiquettes  $\langle \mathcal{E}, \preccurlyeq \rangle$ ;
2. d'une plus petite étiquette  $\perp \in \mathcal{E}$  telle que  $\perp \preccurlyeq e$  pour tout  $e \in \mathcal{E}$ . ■

Pour les systèmes de transitions, les étiquettes sont les évènements, et l'ordre partiel la relation de raffinement.

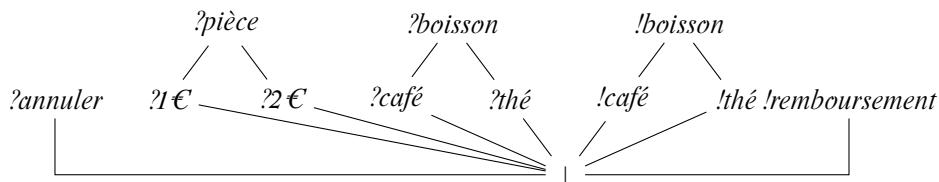


FIGURE 1.14 — Ensemble structuré d'évènements pour un distributeur automatique de boissons.

**Exemple 1.34 — Distributeur automatique de boissons (continuation).** Revenons sur le système de transitions  $M_2$  de l'exemple 1.20 page 8. Le diagramme de Hasse de la figure 1.14 est celui d'un ensemble structuré d'évènements où les évènements  $?pièce$ ,  $?boisson$ , et  $!boisson$  sont raffinés par les évènements  $?1€$ ,  $?2€$ ,  $?café$ ,  $?thé$ ,  $!café$ , et  $!thé$ , correspondant respectivement à l'introduction d'une pièce de 1€ ou de 2€, et au choix par l'utilisateur, ou la délivrance par la machine, d'un café ou d'un thé. ■

**Définition 1.35 — Étiquette d'implémentation.** Soit  $E$  un ensemble structuré d'étiquettes. Une étiquette  $i \in \mathcal{E}_E - \{\perp_E\}$  distincte de  $\perp_E$  est une *étiquette d'implémentation* si et seulement si pour toute étiquette  $e \in \mathcal{E}_E - \{\perp_E\}$ ,  $e \preccurlyeq_E i$  implique  $e = i$ . L'ensemble des étiquettes d'implémentation de  $E$  est noté “ $\text{Imp } E$ ”. L'ensemble  $\{i \in \text{Imp } E \mid i \preccurlyeq_E e\}$  des étiquettes d'implémentation de  $E$  qui précèdent une étiquette  $e$  est noté “[ $e$ ]”. ■

Les seuls évènements qui raffinent un évènement d'implémentation  $i$  sont  $\perp_E$  et  $i$  lui-même. Sur un diagramme de Hasse, les évènements d'implémentation sont ceux qui se trouvent juste au-dessus de  $\perp_E$ .

**Exemple 1.36 — Distributeur automatique de boissons (continuation).** Dans l'exemple précédent,  $?annuler$ ,  $?1€$ ,  $?2€$ ,  $?café$ ,  $?thé$ ,  $!café$ ,  $!thé$ , et  $!remboursement$  sont les évènements d'implémentation. ■

**Proposition 1.37.** Les étiquettes d'implémentation d'un ensemble structuré d'étiquettes sont deux à deux incomparables. ■

**Démonstration.** Soient  $i, j \in \text{Imp } E$  deux étiquettes d'implémentation distinctes. Il n'est pas vrai que  $i \preccurlyeq_E j$  ou  $j \preccurlyeq_E i$  (incomparabilité). Car la définition de la notion d'étiquette d'implémentation impliquerait dans les deux cas que  $i = j$ . Ce qui contredirait l'hypothèse que  $i \neq j$ . ■

**Définition 1.38 — Ensemble bien structuré d'évènements.** Un *ensemble bien structuré d'étiquettes* est un ensemble structuré d'étiquettes  $E$  tel que pour tout étiquette  $e \in \mathcal{E}_E - \{\perp_E\}$ ,  $[e] \neq \emptyset$ , de sorte qu'il existe toujours une étiquette d'implémentation qui précède  $e$ . ■

**Exemple 1.39 — Distributeur automatique de boissons (continuation).** L'ensemble structuré d'évènements de la figure 1.14 est bien structuré. ■

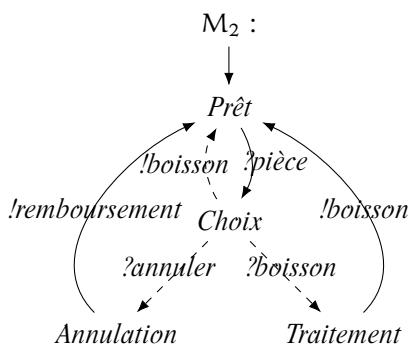


FIGURE 1.15 — Système de transitions modales à évènements structurés.

Un contrat est vu comme un système de transitions, c.-à-d. une spécification dynamique du comportement de ses implantations, où les transitions ont des modalités (figure 1.15). La modalité *may*, représentée par une flèche en pointillés, indique que la transition est permise par le contrat : elle sera ou ne sera pas implantée. La modalité *must*, représentée par une flèche en trait plein, signale que la transition est obligatoire : elle sera implantée. Toute transition obligatoire est évidemment permise : l'attribution à une transition de la modalité *must* implique celle de la modalité *may*.

**Définition 1.40 — Systèmes de transitions modales à évènements structurés.** Un *système de transitions modales à évènements structurés*

$$\langle \mathcal{Q}, i, \mathcal{E}, \preccurlyeq, \perp, \mathcal{T}, \alpha, \beta, \lambda, \mu \rangle$$

est la donnée :

1. d'un système de transitions  $\langle \mathcal{Q}, \{i\}, \mathcal{E}, \mathcal{T}, \alpha, \beta, \lambda \rangle$  ;
2. d'un ensemble bien structuré d'étiquettes  $\langle \mathcal{E}, \preceq, \perp \rangle$  ;
3. d'une fonction totale  $\mu : \mathcal{T} \rightarrow \wp\{may, must\}$  qui associe à chaque transition  $t \in \mathcal{T}$  un ensemble de modalités  $\mu(t) \subseteq \{may, must\}$  tel que :
  - $\mu(t)$  est non vide ;
  - $must \in \mu(t)$  implique  $may \in \mu(t)$ . ■

**Exemple 1.41 — Distributeur automatique de boissons (continuation).** Le système de transitions modales à évènements structurés de la figure 1.15 utilise l'ensemble structuré d'étiquettes de la figure 1.14. ■

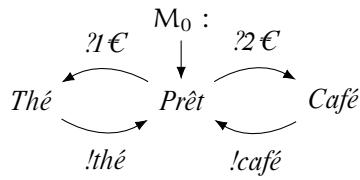


FIGURE 1.16 — Implémentation.

**Définition 1.42 — Implémentation.** Une *implémentation* est un système de transitions modales à évènements structurés  $M$  où tous les évènements sont des évènements d'implémentation, et où toutes les transitions portent la modalité *must* : pour tout  $t \in \mathcal{T}_M$ ,  $\lambda_M(t) \in \text{Imp}\langle \mathcal{E}_M, \preceq_M, \perp_M \rangle$  et  $must \in \mu_M(t)$ . ■

**Exemple 1.43 — Distributeur automatique de boissons (continuation).** Le système de transitions modales sur la figure 1.16 est une implémentation. ■

**Définition 1.44 — Raffinement.** Soient  $M_1$  et  $M_2$  deux systèmes de transitions modales partageant le même ensemble bien structuré d'évènements. On dit que  $M_1$  raffine  $M_2$ , et on écrit " $M_1 \leq M_2$ ", si et seulement s'il existe une relation entre leurs états  $\mathcal{R} \subseteq \mathcal{Q}_{M_1} \times \mathcal{Q}_{M_2}$  telle que  $i_{M_1} \mathcal{R} i_{M_2}$ , et pour tous  $q_1 \in \mathcal{Q}_{M_1}$  et  $q_2 \in \mathcal{Q}_{M_2}$  vérifiant  $q_1 \mathcal{R} q_2$  :

- aucune nouvelle transition non prévue par  $M_2$  n'apparaît dans  $M_1$  : toute transition de source  $q_1$  permise par  $M_1$  — cela comprend évidemment celles qui sont obligatoires — raffine une transition de source  $q_2$  permise par  $M_2$  : pour tout  $t_1 \in \mathcal{T}_{M_1}$  tel que  $\alpha_{M_1}(t_1) = q_1$  et  $may \in \mu_{M_1}(t_1)$ , il existe  $t_2 \in \mathcal{T}_{M_2}$  tel que  $\alpha_{M_2}(t_2) = q_2$ ,  $may \in \mu_{M_2}(t_2)$ ,  $\lambda_{M_1}(t_1) \preceq \lambda_{M_2}(t_2)$ , et  $\beta_{M_1}(t_1) \mathcal{R} \beta_{M_2}(t_2)$  ;
- $M_1$  respecte les obligations prescrites par  $M_2$  : toute transition de source  $q_2$ , spécifiée comme étant obligatoire par  $M_2$ , est raffinée dans  $M_1$  par une transition obligatoire de source  $q_1$  : pour tout  $t_2 \in \mathcal{T}_{M_2}$  tel que  $\alpha_{M_2}(t_2) = q_2$  et  $must \in \mu_{M_2}(t_2)$ , il existe  $t_1 \in \mathcal{T}_{M_1}$  tel que  $\alpha_{M_1}(t_1) = q_1$ ,  $must \in \mu_{M_1}(t_1)$ ,  $\lambda_{M_1}(t_1) \preceq \lambda_{M_2}(t_2)$ , et  $\beta_{M_1}(t_1) \mathcal{R} \beta_{M_2}(t_2)$ . ■

**Exemple 1.45 — Distributeur automatique de boissons (continuation).** Sur la figure 1.17, on a la chaîne de raffinement  $M_0 \leq M_1 \leq M_2$ . Les états de même couleur sont en relation. ■

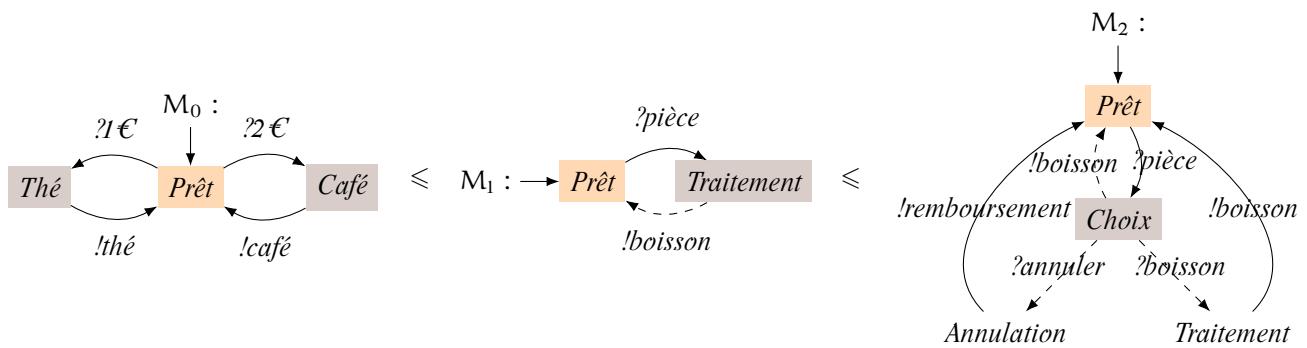


FIGURE 1.17 — Raffinement.

Voilà pour cette longue mais nécessaire introduction. Nous avons maintenant tous les éléments pour préciser les objectifs du stage.

### 1.3. Objectifs du stage

Coq [BC04; Chl13] est un assistant de preuve dont le typage est si expressif, qu'on peut y implémenter des théories mathématiques pour certifier leurs théorèmes. Nous verrons au chapitre 2 les techniques qui permettent cela. Sa rigueur implacable produit des démonstrations bien plus fiables que celles écrites sur du papier. Le but du stage était de réviser en Coq la théorie des systèmes de transitions modales à événements structurés de [Bau+12], en la généralisant à un nombre quelconque, voire une infinité, de modalités.

# 2

## Formalisation en Coq des systèmes de transitions modales

Coq [BC04; Chl13] est un assistant de preuve qui offre une garantie supplémentaire non négligeable par rapport aux preuves faites à la main. Il est utile pour implémenter des théories mathématiques existantes, et réviser leurs théorèmes — ce que nous devions faire dans ce stage pour la théorie des systèmes de transitions modales à événements structurés. Coq est aussi un langage de programmation fonctionnelle qui permet de définir des types de données, et des programmes opérant sur eux. De sorte qu'avec lui, on peut démontrer qu'un programme est conforme à sa spécification, ou extraire un programme d'une démonstration. Ce chapitre revient brièvement sur ce qui rend tout cela possible : les types comme valeurs de première classe, l'ordre supérieur, le produit dépendant, et la correspondance de Curry-Howard. Nous présenterons ensuite nos travaux. Nous supposons que le lecteur a déjà pratiqué la programmation fonctionnelle.

### 2.1. Termes et types en Coq

Le typage statique est une approximation du comportement qu'aura un logiciel au moment de son exécution. Son but est d'assurer, dès la compilation, que celui-ci ne va pas se comporter de certaines façons indésirables. Il y parvient en étiquetant les termes du langage de programmation par des *types* [Pie02]. Par exemple, en Coq, l'écriture “ $t: T$ ” signifie que le terme  $t$  est de type  $T$ . Imaginons qu'on cherche à écrire un programme *quotient* qui doit avoir pour comportement celui de calculer le quotient de la division euclidienne de deux entiers naturels. Dans la plupart des langages de programmation, on peut exprimer quelque chose qui ressemble plus ou moins à ceci :  $\text{quotient} : \text{nat} * \text{nat} \rightarrow \text{nat}$ . Ce typage est une approximation du comportement souhaité pour *quotient* — à savoir celui de calculer le quotient de la division euclidienne de deux entiers naturels — qui dit, en substance, que *quotient* prend en entrée deux entiers naturels — le dividende et le diviseur —, et retourne un entier naturel

— le quotient. Grâce à ce typage, un vérificateur de type intégré au compilateur peut facilement contrôler, sans le concours du programmeur, que le code source proposé pour *quotient*, sensé calculer le quotient de la division euclidienne de deux entiers naturels, ne va pas retourner un booléen ou un entier négatif. On voit donc que certains comportements indésirables sont écartés, mais pas tous. Par exemple, le programme *quotient*, tout en s'appelant « quotient », pourrait se contenter de retourner son premier argument, et le vérificateur de type ne dira rien. La solution de ce problème est d'augmenter la précision — c.-à-d. l'expressivité — du typage, afin que le comportement de *quotient* à l'exécution soit entièrement décrit par son type. Voyons succinctement l'approche de Coq pour y parvenir.

### 2.1.1. Des types comme valeurs de première classe

En programmation, une valeur est dite *de première classe* lorsqu'on peut : (1) la lier à un identificateur, c.-à-d. lui attribuer un nom; (2) la stocker dans une structure de données, par exemple une liste; et (3) l'utiliser comme argument ou valeur de retour d'une fonction<sup>1</sup>.

En Coq, les types sont des valeurs de première classe. Une difficulté se présente. Toute valeur de première classe est typée. Par exemple, 14 est de type *nat* — le type des entiers naturel —, et *true* de type *bool* — le type des booléens. Mais, puisque *nat* et *bool* sont, tout comme 14 et *true*, des valeurs de première classe, quels sont donc leurs types? Quand nous interrogeons Coq, voici sa réponse :

```
Check nat.
Check bool.
```

```
nat: Set
bool: Set
```

La commande Check affiche le type d'une expression. Dans la terminologie de Coq, le type d'un type est une *sorte*. Le type des types *nat* et *bool* est la sorte *Set*. *Set* signifie *ensemble* en français. Il est tout à fait intuitif de regarder les types *nat* et *bool* comme des ensembles. Seulement voilà : *Set* étant une valeur de première classe, il faut aussi lui attribuer un type, auquel il faudra encore attribuer un type, et ainsi de suite, à l'infini. Les concepteurs de Coq ont eu l'idée suivante.

```
Set Printing Universes.
Check Set.
Check Type.
```

```
Set: Type (* (Set)+1 *)
Type (* Top.1 *): Type (* (Top.1)+1 *)
```

En Coq, “(\*)” et “(\*)” délimitent un bloc de commentaires. Lorsqu'on demande à Coq le type du type *Set*, il répond : *Type*. Et si l'on précède notre requête par la commande Set Printing Universes, il précise dans un commentaire : *(Set)+1*. En fait, cette réponse signifie que le type du type *Set* est la sorte *Type<sub>1</sub>*. De même,

1. Source : <https://msdn.microsoft.com/fr-fr/library/Dd233158.aspx>.

quand on demande à Coq le type du type *Type*, il répond *Type*. Cela ne veut pas dire que *Type* est son propre type. Car Coq serait alors inconsistent. Avec l'option Set Printing Universes active, Coq révèle son secret. Le *Top.1* qui apparaît en commentaire dénote une variable interne que nous appellerons *k*. Coq dit, en fait, que le type du type  $Type_k$  est la sorte  $Type_{k+1}$ . Ainsi, il y a un univers infini de sortes  $(Type_k)_{k \in \mathbb{N}}$ , avec  $Type_0 = Set$ . L'utilisateur n'a pas la possibilité de jouer avec l'indice *k*. Lorsqu'il tape “*Type*”, le système détermine tout seul la valeur de *k*, ou signale une inconsistence. Du point de vue de l'utilisateur, *Type*: *Type*.

### 2.1.2. Ordre supérieur

L'*ordre supérieur* signifie que les fonctions sont, elles-aussi, des valeurs de première classe. Par exemple,  $f: (nat \rightarrow nat) \rightarrow bool$  est une fonction prenant en argument une fonction  $g: nat \rightarrow nat$ , et retournant un booléen  $f(g): bool$ . Par contre,  $f: nat \rightarrow (nat \rightarrow bool)$  est une fonction prenant en argument un entier naturel  $n: nat$ , et retournant une fonction  $f(n): nat \rightarrow bool$ ; de sorte que si  $p: nat$ , alors  $f(n)(p): bool$  est un booléen. Une telle fonction est, pour ainsi dire, une fonction à deux arguments. Pour réduire l'usage des parenthèses, l'application d'une fonction *f* à un terme *t* est notée “*f t*”. La flèche “ $\rightarrow$ ” est associative à droite, et l'application d'une fonction associative à gauche :  $nat \rightarrow nat \rightarrow bool = nat \rightarrow (nat \rightarrow bool)$ , et  $f n p = (f n) p$ .

### 2.1.3. Types dépendants

Soient  $T_1: Type, \dots, T_n: Type$ ,  $n$  termes de type *Type*. Toute fonction

$$F: T_1 \rightarrow \dots \rightarrow T_n \rightarrow Type,$$

tenant  $n$  arguments  $x_1: T_1, \dots, x_n: T_n$ , et retournant un type  $F x_1 \dots x_n: Type$ , est un *type dépendant*; car le type  $F x_1 \dots x_n$  dépend des valeurs données aux arguments  $x_1, \dots, x_n$ . Les *types génériques*, par exemple, sont des types dépendants pour lesquels l'un des  $T_k = Type$ . C'est le cas du type *list T* des listes d'éléments de type *T*:

Check *list*.

*list: Type → Type*

Nous savons que *nat* et *bool* sont de type  $Set = Type_0$ , c.-à-d. de type *Type* du point de vue de l'utilisateur. Par conséquent, *list nat* est le type des listes d'entiers naturels, et *list bool* celui des listes de booléens. Grâce à la commande *Print*, nous pouvons voir comment Coq a prédéfini le type *list T*:

Print *list*.

```
Inductive list (T: Type): Type :=
  nil: list T
  cons: T → list T → list T
```

La réponse qui s'affiche se lit comme un roman : quel que soit le type  $T$ , la liste vide  $\text{nil}: \text{list } T$  est une liste d'éléments de type  $T$ , et si  $x: T$  et  $l: \text{list } T$ , alors  $\text{cons } x \ l: \text{list } T$  — l'ajout de  $x$  en tête de la liste  $l$  — est une liste d'éléments de type  $T$ . Étant donné un type dépendant  $F$ , on appelle *produit dépendant* le type des fonctions prenant  $n$  arguments  $x_1: T_1, \dots, x_n: T_n$ , et dont le type de retour est de la forme  $F \ t_1 \dots t_n$ , où les  $t_k: T_k$  sont des termes qui dépendent des valeurs données aux arguments  $x_1, \dots, x_n$ . En Coq, ce type est noté :

“ $\forall (x_1: T_1) \dots (x_n: T_n), F \ t_1 \dots t_n$ ”.

Le produit dépendant est la première clé pour gagner en précision dans le typage. Nous pouvons, par exemple, définir le type  $n\text{-list } T n$  des listes d'éléments de type  $T$  et de longueur  $n$  :

```
Inductive n_list (T: Type): nat → Type :=
  [] n_nil: list T 0
  [] n_cons: ∀ (x: T) (n: nat) (l: n_list T n), n_list T (n + 1).
```

*n\_list* is defined

On commence par indiquer que  $n\text{-list}$  est un type générique, c.-à-d. dépendant, à deux arguments. On peut d'ailleurs le vérifier :

Check *n\_list*.

*n\_list*: Type → nat → Type

Les deux lignes suivantes expriment que la liste vide  $n\text{-nil}$  est une liste d'éléments de type  $T$  de longueur nulle, et que si  $x: T$ ,  $n: \text{nat}$ , et  $l: n\text{-list } T n$ , alors alors  $n\text{-cons } x \ n \ l$  — l'ajout de  $x$  en tête de la liste  $l$  de longueur  $n$  — est une liste d'éléments de type  $T$  de longueur  $n + 1$ . Le type prédéfini des listes peut s'écrire ainsi :

```
Inductive list (T: Type): Type :=
  [] nil: list T
  [] cons: ∀ (x: T) (l: list T), list T.
```

Toutefois, comme dans “ $\forall (x: T) (l: \text{list } T), \text{list } T$ ” aucun des deux arguments  $x$  et  $l$  n'est utilisé après la virgule délimiteur, il est plus simple d'écrire “ $T \rightarrow \text{list } T \rightarrow \text{list } T$ ”. Pour terminer cette section, voici une syntaxe alternative et élégante autorisée par Coq pour le produit dépendant :

```
Inductive list (T: Type): Type :=
  [] nil: list T
  [] cons (x: T) (l: list T): list T.

Inductive n_list (T: Type): nat → Type :=
  [] n_nil: list T 0
  [] n_cons (x: T) (n: nat) (l: n_list T n): n_list T (n + 1).
```

La seconde clé pour augmenter l'expressivité du typage est la correspondance de Curry-Howard.

### 2.1.4. Correspondance de Curry-Howard

La correspondance de Curry-Howard regarde une proposition  $\varphi$  comme un type, et tout terme  $t: \varphi$  de type  $\varphi$  comme une démonstration de sa véracité [Mon00]. Chaque sorte de termes — les variables, les fonctions, les  $n$ -uplets, les unions, le pattern-matching, etc. — est alors l'analogue d'une construction logique. Une variable  $x: \varphi$  est donc une démonstration inconnue de  $\varphi$  : elle représente l'hypothèse que  $\varphi$  est vraie. Une fonction  $f: \varphi \rightarrow \psi$ , de la forme  $f = \text{fun } (x: \varphi) \Rightarrow t$  — syntaxe de Coq —, où  $t: \psi$  est un terme de type  $\psi$ , associe à toute hypothèse  $x: \varphi$  de  $\varphi$  une preuve  $t: \psi$  de  $\psi$ . Cela signifie que  $f: \varphi \rightarrow \psi$  est une démonstration de  $\psi$  à partir de  $\varphi$ , c.-à-d. une démonstration que  $\varphi$  implique  $\psi$ . Ainsi, l'implication et le type des fonctions sont une seule et même chose. Un  $n$ -uplets  $(t_1, \dots, t_n): \varphi_1^* \dots \varphi_n^*$ , où les  $t_k: \varphi_k$  sont des termes de type  $\varphi_k$ , est la donnée d'une démonstration de  $\varphi_1$ , d'une démonstration de  $\varphi_2$ , ..., et d'une démonstration de  $\varphi_n$ , c.-à-d. d'une démonstration de  $\varphi_1 \wedge \dots \wedge \varphi_n$ . Le type produit coïncide donc avec la conjonction. Une union  $t: \{\varphi_1\} + \dots + \{\varphi_n\}$  — pensez aux unions du langage C — est soit un terme de type  $\varphi_1$ , soit un terme de type  $\varphi_2$ , ..., soit un terme de type  $\varphi_n$ . Elle prouve une des propositions  $\varphi_k$ , et est donc une démonstration de  $\varphi_1 \vee \dots \vee \varphi_n$ . De sorte que le type union correspond à la disjonction. Un pattern-matching sur  $t$  est un raisonnement par cas. Un type dépendant  $F: T_1 \rightarrow \dots \rightarrow T_n \rightarrow \text{Type}$  est une proposition paramétrée, c.-à-d. un prédictat. Lorsque  $F: T \rightarrow \text{Type}$  n'a qu'un argument, tout couple  $(t, d): T^* F t$  est une preuve que le prédictat  $F$  est vrai pour le terme  $t: T$ , c.-à-d. une démonstration de la proposition quantifiée existentiellement  $\exists(x: T), F x$  : la quantification existentielle est un cas particulier du type produit. Enfin, si  $F: T_1 \rightarrow \dots \rightarrow T_n \rightarrow \text{Type}$  possède  $n$  arguments, le produit dépendant,

$$\forall (x_1: T_1) \dots (x_n: T_n), F t_1 \dots t_n,$$

est le type des fonctions qui prouvent  $F t_1 \dots t_n$  à partir de  $x_1: T_1, \dots, x_n: T_n$ , les  $x_k$  pouvant être aussi bien des hypothèses que des variables de données. Ce sont des démonstrations d'une proposition quantifiée universellement : la quantification universelle est le produit dépendant. Par exemple,

$$\forall (n: \text{nat}) (d: \text{nat}) (H: d \neq 0), \exists (q: \text{nat}), n \leq d^* q \wedge n < d^*(q+1),$$

est le type des fonctions ou des démonstrations qui, à partir de deux entiers naturels  $n$  et  $d$ , et de l'hypothèse que  $d$  est non nul, prouvent l'existence d'un entier naturel  $q$  vérifiant la propriété caractéristique du quotient. Les démonstrations du genre : « Si  $q$  n'existe pas on aurait telle contradiction ; donc  $q$  existe », ne sont pas des fonctions, car elles ne donnent aucun témoin pour  $q$ . Pour être des fonctions, les démonstrations doivent être *constructives*, au sens où elles doivent donner un algorithme pour construire  $q$  à partir de  $n$  et  $d$ . On trouve dans cette catégorie les démonstrations par récurrence structurelle. Le constructivisme permet l'extraction de programmes.

Coq possède la sorte  $\text{Set}: \text{Type}_1$  pour les ensembles, et la sorte  $\text{Prop}: \text{Type}_1$  pour les propositions. Les constructions pour la sorte  $\text{Set}$  — le type produit, type union, etc. — ont leur analogue dans la sorte  $\text{Prop}$  — la conjonction, la disjonction, etc. Elles sont définies deux fois : une fois pour  $\text{Set}$ , une autre fois pour  $\text{Prop}$ . Toutefois, la séparation entre  $\text{Set}$  et  $\text{Prop}$  est justifiée par le fait qu'au moment d'extraire les programmes, il faut se débarrasser de tous les termes de

sorte *Prop*. Car un programme efficace doit calculer les résultats et non perdre du temps à les prouver, surtout si les preuves sont déjà faites.

Nous voulons maintenant finir ce rapport de stage en donnant un aperçu de nos travaux.

## 2.2. Architecture de la bibliothèque

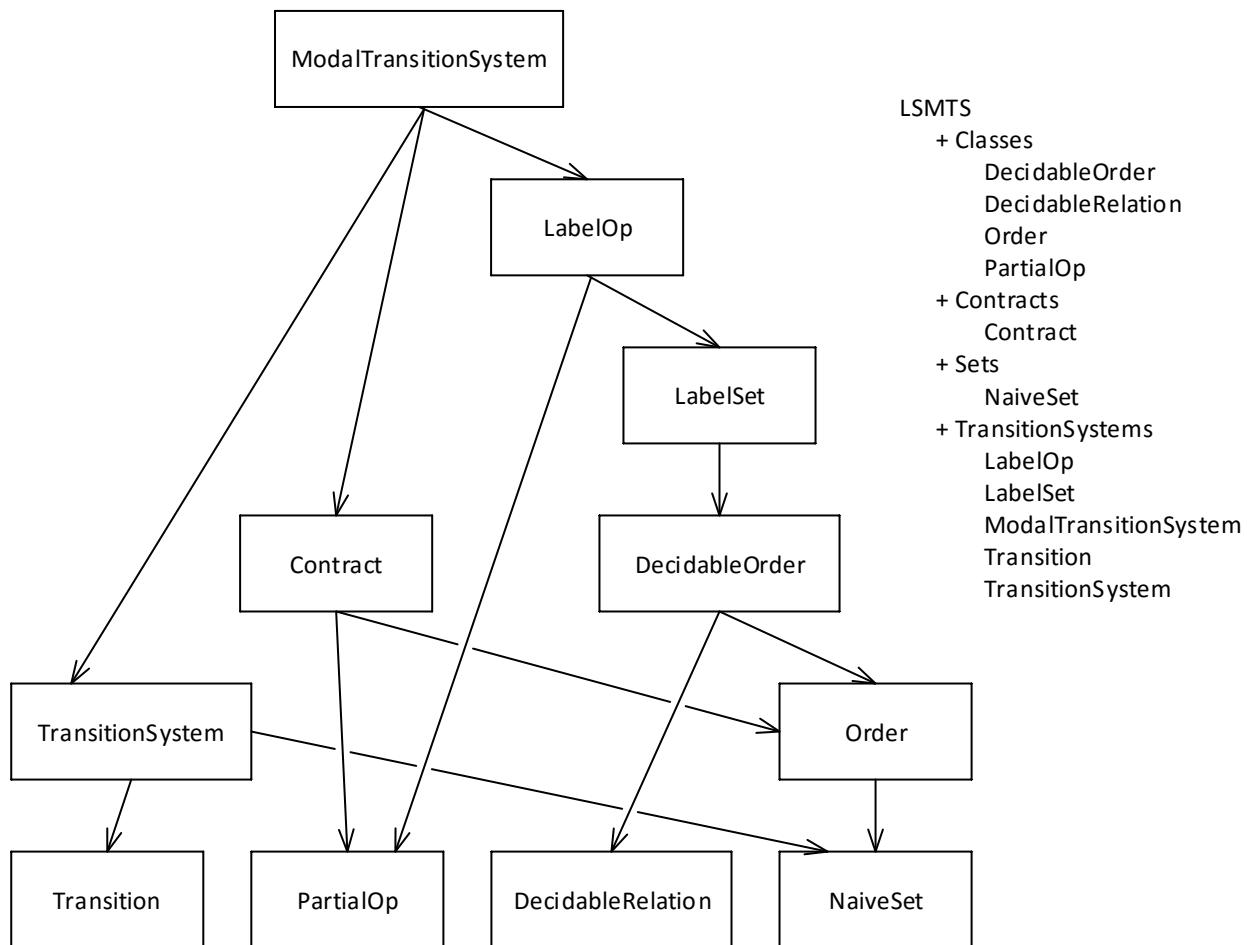


FIGURE 2.1 — Architecture de la bibliothèque. Le module pointé par un flèche est utilisé par le module à la source de cette flèche. À droite : organisation des fichiers sources dans les répertoires Classes, Contracts, Sets, et TransitionSystems.

La figure 2.1 montre les modules Coq que nous avons développés. Voici le résumé du contenu de chacun d'eux :

*NaiveSet* : reprise de l'implémentation de la théorie naïve des ensembles disponible dans la bibliothèque standard. Ce module tire profit de l'en-codage Unicode des caractères pour permettre l'emploi des notations conventionnelles :  $\in$ ,  $\subseteq$ , etc.

*DecidableRelation* : implémentation des relations dites *décidables*, c.-à-d. des relations binaires  $\mathcal{R} \subseteq \mathcal{E} \times \mathcal{E}$  sur un ensemble  $\mathcal{E}$ , pour lesquelles il existe un algorithme de décision permettant de savoir, pour tous  $x, y \in \mathcal{E}$ , si  $x \mathcal{R} y$ .

*PartialOp* : implémentation des opérations binaires  $\mathcal{O} \in \mathcal{E} \times \mathcal{E} \rightarrow \mathcal{E}$  sur un ensemble  $\mathcal{E}$ , qui sont partielles, c.-à-d. définies uniquement sur un domaine  $\mathcal{D} \subseteq \mathcal{E} \times \mathcal{E}$ .

*Transition* : implémentation des transitions par des triplets de la forme source-événement-cible.

*TransitionSystem* : implémentation des systèmes de transition de la définition 1.17.

*Order* : reprise de l'implémentation des quasi-ordres, ordres stricts, et ordres partiels de la bibliothèque standard, et définition des notions d'élément minimal/maximal, de plus petit/grand élément, et de borne inférieure/supérieure.

*DecidableOrder* : implémentation des ordres décidables, c.-à-d. munis d'un algorithme de décision.

*Contract* : implementation de la métathéorie des contrats.

*LabelSet* : implémentation des ensembles structurés d'étiquettes.

*LabelOp* : implémentation des opérations sur les ensembles structurés d'étiquettes.

*ModalTransitionSystem* : généralisation et implémentation des systèmes de transitions modales à événements structurés, en passant de deux modalités à un ensemble arbitrairement grand de modalités.

Nous nous proposons de présenter à titre d'exemple quelques fragments de code, puis nous concluerons.

### 2.3. Systèmes de transitions modales en Coq

Il existe une technique standard pour formaliser en Coq les structures mathématique : les classes [CS14]. Les classes se présente comme des enregistrements avec des paramètres et des champs dont les valeurs sont fixées au moment de l'instanciation. Les instances d'une classe ne partagent entre elles que les valeurs des paramètres ; celles des champs sont enfermées dans leur intérieur, de sorte que leur lecture n'est possible que par des accesseurs. La pratique recommandée dans [CS14] est de placer toutes les composantes d'une structure mathématique dans les paramètres, et de ne garder dans les champs que les propriétés à démontrer au moment de l'instanciation. Cela afin de pouvoir facilement comparer les instances entre elles sans passer par des accesseurs. Nous avons suivi cette méthode par exemple pour l'implémentation des ensembles structurés d'étiquettes (*label-set* en anglais) :

```
Require Import Coq.Relations.Relation_Definitions.

Class LSet_LabelSet {label: Type} (precedes: relation label) (bottom: label): Prop :=
LSet_buildLabelSet {
  (* Propriétés à démontrer au moment de l'instantiation *)
}.
```

Les trois composantes  $\langle \mathcal{E}, \preceq, \perp \rangle$  d'un ensemble structuré d'étiquettes sont respectivement formalisées par le type *label*, la relation binaire *precedes* sur *label*,

et le plus petit élément *bottom* de *label*. Le type générique *relation* est prédéfini dans la bibliothèque standard comme étant un prédictat à deux arguments :

```
Definition relation (T: Type) : Type := T → T → Prop.
```

Pour que  $\langle \mathcal{E}, \preccurlyeq, \perp \rangle$  soit un ensemble structuré d'étiquettes, il faut que  $\langle \mathcal{E}, \preccurlyeq \rangle$  soit un ensemble partiellement ordonné, et que  $\perp$  soit le plus petit élément de  $\mathcal{E}$  :

```
Add LoadPath ".." as LSMTS.
Require Import Coq.Relations.Relation_Definitions.
Require Import LSMTS.Sets.NaiveSet.
Require Import LSMTS.Classes.Order.

Class LSet_LabelSet {label: Type} (precedes: relation label) (bottom: label): Prop :=
LSet_buildLabelSet {
  LSet_QuasiOrder: QO_QuasiOrder precedes;
  LSet_eq: relation label := QO_eq LSet_QuasiOrder;
  LSet_Equivalence: Equivalence LSet_eq := QO_Equivalence LSet_QuasiOrder;
  LSet_PartialOrder: PO_PartialOrder LSet_eq precedes :=
    QO_PartialOrder LSet_QuasiOrder;
  LSet_bottom_least: PO_least LSet_PartialOrder (Nu_full label) bottom
}.
```

Expliquons pas à pas ce que ce bout de code traduit. Tout d'abord, nous chargeons nos modules *NaiveSet* et *Order*, lesquels se trouvent dans les répertoires *..Sets/* et *..Classes/* respectivement :

```
Add LoadPath ".." as LSMTS.
Require Import LSMTS.Sets.NaiveSet.
Require Import LSMTS.Classes.Order.
```

Les accolades autour du paramètre *label* indique qu'il est implicite, car il se déduit aussi bien à partir de *precedes* que de *bottom*. Le typage en *Prop* exprime que les champs ne contiennent que des propositions qui seront instanciées par des démonstrations :

```
Class LSet_LabelSet {label: Type} (precedes: relation label) (bottom: label): Prop :=
```

Pour démontrer qu'une relation binaire est un ordre partiel, il faut d'abord montrer qu'elle est un quasi-ordre. Cette preuve est stockée dans le champs *LSet\_QuasiOrder* :

```
LSet_buildLabelSet {
  LSet_QuasiOrder: QO_QuasiOrder precedes;
```

En Coq, l'antisymétrie, qui fait qu'un quasi-ordre  $\rightsquigarrow$  devienne un ordre partiel, se prouve modulo une relation d'équivalence  $\equiv$ . Nous prenons celle qui est induite par  $\rightsquigarrow$ , c.-à-d. définie telle que  $x \equiv_{\rightsquigarrow} y$  si et seulement si  $x \rightsquigarrow y$  et  $y \rightsquigarrow x$ . Cela donne immédiatement un ordre partiel :

```
LSet_eq: relation label := QO_eq LSet_QuasiOrder;
LSet_Equivalence: Equivalence LSet_eq := QO_Equivalence LSet_QuasiOrder;
LSet_PartialOrder: PO_PartialOrder LSet_eq precedes :=
```

```
QO_PartialOrder LSet_QuasiOrder;
```

Enfin, le champs *LSet\_bottom\_least* contient la preuve que *bottom* est la plus petite de toutes les étiquettes :

```
LSet_bottom_least: PO_least LSet_PartialOrder (Nu_full_label) bottom
}.
```

Nous pouvons définir des méthodes, c.-à-d. des fonctions qui s'appliquent aux instances de la classe :

```
Class LSet_LabelSet {label: Type} (precedes: relation label) (bottom: label): Prop :=
LSet_buildLabelSet {
  (* Code précédent omis *)

  LSet_Implementation: Nu_naiveSet label :=
  fun (i: label) => ¬(LSet_eq i bottom) ∧
    (∀ l: label, ¬(LSet_eq l bottom) → precedes l i → LSet_eq l i);

  LSet_PrecedingImplementations (l: label): Nu_naiveSet label :=
  fun (i: label) => precedes i l ∧ LSet_Implementation i
}.
```

Par exemple, la méthode *LSet\_Implementation*, appliquée à une instance de *LSet\_LabelSet*, retourne son ensemble d'implémentations. On peut voir que les ensembles employés ici ne sont pas des ensembles de la théorie ZFC, mais plutôt des ensembles naïfs, c.-à-d. définis sur la base du principe de compréhension, qui suppose qu'à tout prédicat correspond l'ensemble des objets qui le satisfont :

```
Definition Nu_naiveSet (T: Type): Type := T → Prop.
```

Bien sûr, c'est à manipuler avec précaution, car nous savons que certains prédicats, comme celui du paradoxe de Russel, ne définissent aucun ensemble. Il faut toujours penser à définir des méthodes pour accéder aux valeurs fixées aux paramètres :

```
Class LSet_LabelSet {label: Type} (precedes: relation label) (bottom: label): Prop :=
LSet_buildLabelSet {
  (* Code précédent omis *)

  LSet_label := label;
  LSet_precedes := precedes;
  LSet_bottom := bottom
}.
```

Coq permet d'utiliser Unicode pour les symboles mathématiques familiers :

```
Notation "l1 ⪻ l2 '[LSet' L ]'" := (LSet_precedes L l1 l2) (at level 70, no associativity).
Notation "l1 == l2 '[LSet' L ]'" := (LSet_eq L l1 l2) (at level 70, no associativity).
Notation "l1 <> l2 '[LSet' L ]'" := (¬(LSet_eq L l1 l2)) (at level 70, no associativity).
Notation "[[l]]" := (LSet_PrecedingImplementations _ l) (at level 35, right associativity).
```

On écrira par exemple “ $l_1 \preceq l_2$  [LSet  $L$ ]” lorsque l’étiquette  $l_1$  est plus petite que l’étiquette  $l_2$  dans l’ensemble structuré d’étiquettes  $L$ .

La classe *LSet\_WellFormed* implémente les ensembles bien structurés d’étiquettes. Elle est un prédictat sur les *LSet\_LabelSet* :

```
Class LSet_WellFormed '(L: LSet_LabelSet): Prop :=
LSet_buildWellFormed {
  LSet_WellFormed_LabelSet := L;
  LSet_WellFormed_intro:
    ∀ l: L, l <> LSet_bottom L [LSet_] → ∃ i: L, i ∈ ⟦l⟧
}.
```

Le backquote qui précède le paramètre  $L$  génère automatiquement comme implicites les paramètres *label*, *precedes*, et *bottom* de la classe *LSet\_LabelSet*. On active cette fonctionnalité par l’instruction :

```
Generalizable Variables label precedes bottom.
```

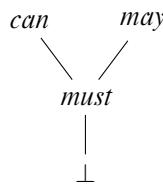


FIGURE 2.2 — Ensemble structuré de modalités.

Pour implémenter les systèmes de transitions modales, il a fallu mettre une partie des composantes dans les champs. Nous avons dans un premier temps généralisé [Bau+12] en passant de deux modalités à un ensemble arbitrairement grand de modalités. Donnons un exemple qui montre l’intérêt d’une telle extension. On pourrait, au lieu de se limiter aux modalités *may* et *must*, signalant respectivement que la transition est permise et obligatoire, ajouter la modalité *can* pour indiquer que la transition est implémentable. On aurait alors (1) que *must* implique *may*, c.-à-d. que toute transition obligatoire est permise, et (2) que *must* implique *can*, c.-à-d. que toute transition obligatoire est implémentable. L’idée est de passer pour cela par un ensemble structuré d’étiquettes comme le montre la figure 2.2 : *must* est l’unique modalité d’implémentation, et on a  $must \preceq may$  et  $must \preceq can$ ,  $m \preceq m'$  signifiant que la modalité  $m$  implique la modalité  $m'$ . Voici donc la définition généralisée des systèmes de transitions modales :

**Définition 2.1 — Systèmes de transitions modales à évènements structurés.**  
Un *système de transitions modales à évènements structurés*

$$\langle Q, i, \mathcal{E}, \preceq_{\mathcal{E}}, \perp_{\mathcal{E}}, \mathcal{T}, \alpha, \beta, \lambda, \mathcal{M}, \preceq_{\mathcal{M}}, \perp_{\mathcal{M}}, \mu \rangle$$

est la donnée :

1. d’un système de transitions  $\langle Q, \{i\}, \mathcal{E}, \mathcal{T}, \alpha, \beta, \lambda \rangle$  ;

2. d'un ensemble bien structuré d'évènements  $\langle \mathcal{E}, \preceq_{\mathcal{E}}, \perp_{\mathcal{E}} \rangle$ ;
3. d'un ensemble structuré de modalités  $\langle \mathcal{M}, \preceq_{\mathcal{M}}, \perp_{\mathcal{M}} \rangle$ ;
4. d'une fonction totale  $\mu : \mathcal{T} \rightarrow \wp(\mathcal{M})$  qui associe à chaque transition  $t \in \mathcal{T}$  un ensemble de modalités  $\mu(t) \subseteq \mathcal{M}$  tel que :
  - $m \preceq_{\mathcal{M}} m'$  et  $m \in \mu(t)$  impliquent  $m' \in \mu(t)$ . ■

**Remarque 2.2.** Nous avons choisi dans l'équipe ACADIE de ne pas imposer que  $\mu(t)$  soit non vide. Il n'est pas dérangeant d'avoir  $\mu(t) = \emptyset$  sur certaines transitions  $t$ . ■

Du côté du code Coq, nous voulons avoir le type des systèmes de transitions modales pour des ensembles structurés d'évènements et de modalités fixés, et ce, afin de pouvoir définir sur lui des relations. C'est pourquoi, nous ne gardons que les évènements et les modalités dans les paramètres :

```
Add LoadPath ".." as LSMTS.
Require Import LSMTS.Sets.NaiveSet.
Require Import LSMTS.TransitionSystems.LabelSet.
Require Import LSMTS.TransitionSystems.TransitionSystem.

Class MTS_TransitionSystem
  '(Event: LSet_WellFormed) '(Modal: LSet_LabelSet) : Type :=
  MTS_buildTransitionSystem {
    MTS_Event := Event;
    MTS_Modal := Modal;
    MTS_state: Type;
    MTS_initial: MTS_state;
    MTS_System: TS_TransitionSystem (Nu_singleton MTS_initial) Event;
    MTS_Modality: TS_Transition MTS_TransitionSystem → Nu_naiveSet Modal;
    MTS_inheritance:
      ∀ t m m', m ∈ (MTS_Modality t) → m ≺ m' [LSet _] →
      m' ∈ (MTS_Modality t)
  }.
```

## 2.4. Résultat et conclusion

Le temps est venu de faire le bilan entre ce qui a été fait et ce qui était prévu.

On voulait premièrement certifier en Coq que la théorie des systèmes de transitions modales est une théorie des contrats. Cela demandait : (1) de définir le type des systèmes de transitions modales comme type des contrats — ce qui a été fait —, (2) de définir le type des implémentations comme type des réalisations — ce qui a été fait —, (3) définir la relation d'équivalence entre les implémentations — cela n'a pas été fait —, (4) définir l'ensemble des implémentations légales — cela n'a pas été fait —, (5) définir la composition de systèmes de transitions modales comme opération de composition — cela a été fait, (6) définir la relation de raffinement comme relation de satisfaction — ce qui a été fait.

Dans un deuxième temps, on prévoyait de certifier l'article [Bau+12]. Sur la période du stage, nous en avons réviser en Coq le quart. L'objectif était de

certifier au moins la monotonie de la composition de systèmes de transitions modales par rapport au raffinement, c.-à-d. que  $M_1 \leq M_2$  et  $M'_1 \leq M'_2$  impliquent  $M_1 \otimes M'_1 \leq M_2 \otimes M'_2$ . Cet objectif n'a, hélas, pas été atteint. Nous sommes parvenus au moins à certifier que si  $M_1 \leq M_2$  alors  $\text{Imp } M_1 \subseteq \text{Imp } M_2$ .

Ce stage a été pour moi l'occasion de découvrir une nouvelle utilisation de Coq. Je suis impressionné par les possibilités qu'offre cet outil.

# Ensembles ordonnés

Passons rapidement en revue les concepts de relation binaire, de quasi-ordre, d'ordre strict, de relation d'équivalence, d'ordre partiel, et de minimilité/maximalité.

**Définition A.1 — Relation binaire.** Soit  $\mathcal{E}$  un ensemble. Toute partie  $\mathcal{R} \subseteq \mathcal{E} \times \mathcal{E}$  est une *relation binaire* sur  $\mathcal{E}$ . On écrit “ $x \mathcal{R} y$ ” plutôt que “ $(x, y) \in \mathcal{R}$ ” pour signifier que  $x, y \in \mathcal{E}$  sont reliés par  $\mathcal{R}$ . ■

En Coq, l'ensemble  $\mathcal{E}$  est représenté par un type, et la relation binaire  $\mathcal{R}$  par un prédicat binaire  $R$  tel que  $R x y$  est vrai si et seulement si  $x \mathcal{R} y$ :

```
Definition relation (E: Type) : Type := E → E → Prop.
```

**Définition A.2 — Quasi-ordre.** Soit  $\mathcal{E}$  un ensemble. Un *quasi-ordre sur  $\mathcal{E}$*  est une relation binaire  $\rightsquigarrow$  sur  $\mathcal{E}$  réflexive — c.-à-d. telle que  $x \rightsquigarrow x$  pour tout  $x \in \mathcal{E}$  —, et transitive — c.-à-d. telle que  $x \rightsquigarrow y$  et  $y \rightsquigarrow z$  impliquent  $x \rightsquigarrow z$  pour tout  $x, y, z \in \mathcal{E}$ . On dit que  $\langle \mathcal{E}, \rightsquigarrow \rangle$  est un *ensemble quasi-ordonné*. ■

```
Class Reflexive {E} (R: relation E) : Prop :=
  reflexivity: ∀ x, R x x.
```

```
Class Transitive {E} (R: relation E) : Prop :=
  transitivity: ∀ x y z, R x y → R y z → R x z.
```

```
Class QO_QuasiOrder {E} (R: relation E) : Prop := {
  QO_set := E;
  QO_precedes := R;
  QO_Reflexive: Reflexive R;
  QO_Transitive: Transitive R
}.
```

Notation " $x \prec y$  "[QO' Q']" := ( $QO\_precedes Q x y$ ) (at level 70, no associativity).

**Définition A.3 — Ordre strict.** Soit  $\mathcal{E}$  un ensemble. Un *ordre strict sur  $\mathcal{E}$*  est une relation binaire  $\prec$  sur  $\mathcal{E}$  irréflexive — c.-à-d. telle que pour tout  $x \in \mathcal{E}$ , il n'est pas vrai que  $x \prec x$  —, et transitive. On dit que  $\langle \mathcal{E}, \prec \rangle$  est un *ensemble strictement ordonné*. ■

Definition complement {E} (R: relation E): relation E := fun x y =>  $\neg(R x y)$ .

Class Irreflexive {E} (R: relation E): Prop :=  
irreflexivity: Reflexive (complement R).

Class SO\_StrictOrder {E} (R: relation E): Prop := {  
SO\_set := E;  
SO\_precedes := R;  
SO\_Irreflexive: Irreflexive R;  
SO\_Transitive: Transitive R  
}.

Notation " $x \prec y$  "[SO' S']" := ( $SO\_precedes S x y$ ) (at level 70, no associativity).

Tout ensemble quasi-ordonné  $\langle \mathcal{E}, \sim \rangle$  induit un ensemble strictement ordonné  $\langle \mathcal{E}, \prec_\sim \rangle$  dont l'ordre strict  $\prec_\sim$  est, par définition, tel que  $x \prec_\sim y$  si et seulement si on a  $x \sim y$  mais pas  $y \sim x$ .

**Définition A.4 — Relation d'équivalence.** Soit  $\mathcal{E}$  un ensemble. Une *relation d'équivalence sur  $\mathcal{E}$*  est un quasi-ordre  $\equiv$  sur  $\mathcal{E}$  qui, en plus d'être réflexif et transitif, est *symétrique* — c.-à-d. tel que  $x \equiv y$  implique  $y \equiv x$  pour tout  $x, y \in \mathcal{E}$ . On dit que  $\langle \mathcal{E}, \equiv \rangle$  est un *setoïde* ou *ensemble partitionné* car les *classes d'équivalence* de  $\equiv$  forment une *partition* de  $\mathcal{E}$ , et, inversement, toute *partition* de  $\mathcal{E}$  induit une relation d'équivalence sur  $\mathcal{E}$ . Ces résultats sont supposés connus. ■

Class Symmetric {E} (R: relation E): Prop :=  
symmetry:  $\forall x y, R x y \rightarrow R y x$ .

Class Equivalence {E} (R: relation E): Prop := {  
Equivalence\_Reflexive: Reflexive R;  
Equivalence\_Symmetric: Symmetric R;  
Equivalence\_Transitive: Transitive R  
}.

**Définition A.5 — Ordre partiel.** Soit  $\langle \mathcal{E}, \equiv \rangle$  un ensemble partitionné. Un *ordre partiel sur  $\langle \mathcal{E}, \equiv \rangle$*  est un quasi-ordre  $\preccurlyeq$  sur  $\mathcal{E}$  qui, en plus d'être réflexif et transitif, est *antisymétrique* pour  $\equiv$  — c.-à-d. tel que  $x \preccurlyeq y$  et  $y \preccurlyeq x$  impliquent  $x \equiv y$  pour tout  $x, y \in \mathcal{E}$ . On dit que  $\langle \mathcal{E}, \equiv, \preccurlyeq \rangle$  est un *ensemble partiellement ordonné*. ■

Class Antisymmetric {E} eq {Eq: Equivalence eq} (R: relation E) :=  
antisymmetry:  $\forall \{x y\}, R x y \rightarrow R y x \rightarrow eq x y$ .

Class PO\_PartialOrder {E} eq {Eq: Equivalence eq} (R: relation E) :=  
PO\_set := E;

```

PO_eq := eq;
PO_precedes := R;
PO_QuasiOrder: QO_QuasiOrder R;
PO_Antisymmetric: Antisymmetric eq R
}.

```

Notation " $x == y$  "[PO' P']" := ( $PO\_eq P x y$ ) (at level 70, no associativity).

Notation " $x \preccurlyeq y$  "[PO' P']" := ( $PO\_precedes P x y$ ) (at level 70, no associativity).

Lorsque la relation d'équivalence  $\equiv$  n'est pas précisée, alors il s'agit, sauf mention contraire, de l'égalité  $=$ . L'antisymétrie fait que sur le graphe de l'ordre  $\preccurlyeq$ , tout circuit est endigué par une classe d'équivalence, c.-à-d. qu'il n'y a pas de circuit à cheval sur deux classes d'équivalence.

Tout ensemble quasi-ordonné  $\langle \mathcal{E}, \sim \rangle$  induit un ensemble partiellement ordonné  $\langle \mathcal{E}, \equiv_{\sim}, \sim \rangle$  dont la relation d'équivalence  $\equiv_{\sim}$  est, par définition, telle que  $x \equiv_{\sim} y$  si et seulement si  $x \sim y$  et  $y \sim x$ . De façon analogue, tout ensemble partiellement ordonné  $\langle \mathcal{E}, \equiv, \preccurlyeq \rangle$  induit un ensemble strictement ordonné  $\langle \mathcal{E}, \prec_{\equiv, \preccurlyeq} \rangle$  dont l'ordre strict  $\prec_{\equiv, \preccurlyeq}$  est, par définition, tel que  $x \prec_{\equiv, \preccurlyeq} y$  si et seulement si on a  $x \preccurlyeq y$  mais pas  $x \equiv y$ .

**Définition A.6 — Comparabilité.** Soit  $\langle \mathcal{E}, \equiv, \preccurlyeq \rangle$  un ensemble partiellement ordonné. Deux éléments  $x, y \in \mathcal{E}$  sont *comparables* si et seulement si  $x \preccurlyeq y$  ou  $y \preccurlyeq x$ . ■

Coercion  $PO\_set: PO\_PartialOrder \rightarrow Sortclass$ .

```

Definition PO_comparable '(P: PO_PartialOrder): relation P :=
  fun x y: P => x \preccurlyeq y [PO P] \vee y \preccurlyeq x [PO P].

```

La coercion sur  $PO\_set$  permet d'interpréter tout ordre partiel comme un type. En fait, la coercion utilise la méthode  $PO\_set$  pour convertir automatiquement toute instance de  $PO\_PartialOrder$  en *Type*.

**Définition A.7 — Élément minimal, élément maximal.** Soit  $\langle \mathcal{E}, \equiv, \preccurlyeq \rangle$  un ensemble partiellement ordonné, et  $\mathcal{X} \subseteq \mathcal{E}$  une partie de  $\mathcal{E}$ . Un élément  $m \in \mathcal{X}$  est *minimal* (resp. *maximal*) dans  $\mathcal{X}$  si et seulement si pour tout  $x \in \mathcal{X}$ ,  $x \preccurlyeq m$  (resp.  $m \preccurlyeq x$ ) implique  $x \equiv m$ . ■

```

Definition PO_Minimal '(P: PO_PartialOrder) (S: Nu_naiveSet P): Nu_naiveSet P :=
  fun m: P => m \in S \wedge (\forall x: P, x \in S \rightarrow x \preccurlyeq m [PO P] \rightarrow x == m [PO P]).

```

```

Definition PO_Maximal '(P: PO_PartialOrder) (S: Nu_naiveSet P): Nu_naiveSet P :=
  fun m: P => m \in S \wedge (\forall x: P, x \in S \rightarrow m \preccurlyeq x [PO P] \rightarrow m == x [PO P]).

```

Les éléments minimaux et maximaux de  $\mathcal{X}$  sont clairement deux à deux incomparables.

**Définition A.8 — Plus petit/grand élément.** Soit  $\langle \mathcal{E}, \equiv, \preccurlyeq \rangle$  un ensemble partiellement ordonné, et  $\mathcal{X} \subseteq \mathcal{E}$  une partie de  $\mathcal{E}$ . Le *plus petit* (resp. *grand*) élément de  $\mathcal{X}$ , noté " $\mathcal{X}^\perp$ " (resp. " $\mathcal{X}^\top$ ") est, s'il existe, l'unique objet tel que  $\mathcal{X}^\perp \in \mathcal{X}$  (resp.  $\mathcal{X}^\top \in \mathcal{X}$ ) et  $\mathcal{X}^\perp \preccurlyeq x$  (resp.  $x \preccurlyeq \mathcal{X}^\top$ ) pour tout  $x \in \mathcal{X}$ . ■

```
Definition PO_least '(P: PO_PartialOrder) (S: Nu_naiveSet P) (l: P): Prop :=
  l ∈ S ∧ (∀ x: P, x ∈ S → l ≲ x [PO P]).
```

```
Definition PO_greatest '(P: PO_PartialOrder) (S: Nu_naiveSet P) (g: P): Prop :=
  g ∈ S ∧ (∀ x: P, x ∈ S → x ≲ g [PO P]).
```

**Définition A.9 — Borne inférieure.** Soit  $\langle \mathcal{E}, \equiv, \preccurlyeq \rangle$  un ensemble partiellement ordonné, et  $\mathcal{X} \subseteq \mathcal{E}$  une partie de  $\mathcal{E}$ . La  *borne inférieure* de  $\mathcal{X}$ ,

$$\inf \mathcal{X} \triangleq \{ m \in \mathcal{E} \mid \text{pour tout } x \in \mathcal{X}, m \preccurlyeq x \}^\top$$

est, s'il existe, le plus grand élément de l'ensemble des *minorants* de  $\mathcal{X}$ . ■

La traduction anglaise de *minorant* est *lower bound*, et celle de *borne inférieure* est *greatest lower bound* :

```
Definition PO_LowerBound
  '(P: PO_PartialOrder) (S: Nu_naiveSet P): Nu_naiveSet P :=
    fun l: P ⇒ ∀ x: P, x ∈ S → l ≲ x [PO P].
```

```
Definition PO_glb
  '(P: PO_PartialOrder) (S: Nu_naiveSet P) (glb: P): Prop :=
  let L := (PO_LowerBound P S) in glb ∈ L ∧ PO_greatest P L glb.
```

**Définition A.10 — Borne supérieure.** Soit  $\langle \mathcal{E}, \equiv, \preccurlyeq \rangle$  un ensemble partiellement ordonné, et  $\mathcal{X} \subseteq \mathcal{E}$  une partie de  $\mathcal{E}$ . La  *borne supérieure* de  $\mathcal{X}$ ,

$$\sup \mathcal{X} \triangleq \{ m \in \mathcal{E} \mid \text{pour tout } x \in \mathcal{X}, x \preccurlyeq m \}^\perp$$

est, s'il existe, le plus petit élément de l'ensemble des *majorants* de  $\mathcal{X}$ . ■

La traduction anglaise de *majorant* est *upper bound*, et celle de *borne supérieure* est *least upper bound* :

```
Definition PO_UpperBound
  '(P: PO_PartialOrder) (S: Nu_naiveSet P): Nu_naiveSet P :=
    fun u: P ⇒ ∀ x: P, x ∈ S → x ≲ u [PO P].
```

```
Definition PO_lub
  '(P: PO_PartialOrder) (S: Nu_naiveSet P) (lub: P): Prop :=
  let U := (PO_UpperBound P S) in lub ∈ U ∧ PO_least P U lub.
```

- [Arn92] André ARNOLD. *Systèmes de transitions finis et sémantique des processus communicants*. Études et recherches en informatique. Masson, 1992. ISBN : 2-225-82746-X.
- [Bau+12] Sebastian S. BAUER et al. « Extending Modal Transition Systems with Structured Labels ». In : *Mathematical Structures in Computer Science* 22.4 (2012), p. 581-617.
- [BC04] Yves BERTOT et Pierre CASTÉRAN. *Interactive Theorem Proving and Program Development. Coq'Art : The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science : An EATCS Series. Springer Berlin Heidelberg, 2004. ISBN : 978-3-642-05880-6.
- [Ben+12a] Albert BENVENISTE et al. *Contracts for the Design of Embedded Systems. Part II : Theory*. HAL. 2012.
- [Ben+12b] Albert BENVENISTE et al. *Contracts for the Design of Embedded Systems. Part I : Methodology and Use Cases*. HAL. 2012.
- [Chl13] Adam CHLIPALA. *Certified Programming with Dependent Types. A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013. ISBN : 978-0-262-02665-9.
- [CS14] Pierre CASTÉRAN et Matthieu SOZEAU. *A Gentle Introduction to Type Classes and Relations in Coq*. HAL. 2014. URL : <http://www.labri.fr/perso/casteran/CoqArt/TypeClassesTut/typeclassestut.pdf>.
- [Mon00] Jean-François MONIN. « Introduction aux méthodes formelles ». In : 2<sup>e</sup> éd. Collection technique et scientifique des télécommunications. Paris : Hermès Science Publications, 2000. ISBN : 2-7462-0140-2.
- [Pie02] Benjamin C. PIERCE. *Types and Programming Languages*. The MIT Press, 2002. ISBN : 0-262-16209-1.

- [Por02] Jean-Marie PORTAL. « Le zéro défaut n'existe pas mais on peut tout de même s'en approcher ». In : *01 Informatique* 1661 (jan. 2002), p. 24. URL : <http://www.atelierb.eu/pdf/presse/01-Informatique-JRA.pdf> (visité le 16/04/2015).
- [ST12] Donald SANNELLA et Andrzej TARLECKI. « Working within an arbitrary logical system ». In : *Foundations of Algebraic Specification and Formal Software Development*. Monographs in Theoretical Computer Science : An EATCS Series. Springer Berlin Heidelberg, 2012, p. 155-228. ISBN : 978-3-642-17335-6.