Cohésion et couplage de théories d'actions

Andreas Herzig Ivan José Varzinczak*

IRIT – Université Paul Sabatier
118 route de Narbonne – 31062 Toulouse France

e-mail : {herzig,ivan}@irit.fr

Résumé

Dans ce travail on récupère quelques principes de projet utilisés en génie du logiciel et on les adapte à l'analyse et à la conception de descriptions de domaine en raisonnement sur les actions. On propose un découpage de la théorie en des modules, en montrant comment les concepts de cohésion et couplage peuvent être changés en des tests de consistance de différentes configurations de ceux-ci. Cela nous donne de nouveaux critères pour l'évaluation de descriptions de domaine. On montre que des théories modulaires possèdent des propriétés théoriques intéréssantes.

Mots Clef

Raisonnement sur les actions, cohésion, couplage.

Abstract

Here we recast some principles used in software engineering and adapt them to the design and analysis of domain descriptions in reasoning about actions. We propose splitting a theory in modules and show how the requirements of cohesion and coupling can be turned into tests of consistency of different arrangements of them. This gives us new criteria for domain description evaluation. We show that modular theories have interesting theoretical properties.

Keywords

Reasoning about actions, cohesion, coupling.

1 Introduction

Il semble y avoir un consensus selon lequel des principes de la programmation par objet peuvent être aussi utilisés pour la représentation de la connaissance. Un de tels concepts est la *modularité*, c'est-à-dire partager le logiciel en de plusieurs modules, basés sur leurs fonctionnalités ou la similarité de l'information dont chacun s'occupe. Cela veut dire qu'à la place d'avoir un programme qui ressemble à un couteau suisse, mieux vaut le décomposer en plusieurs sous-modules, chacun responsable d'une tâche précise. Par exemple, un programme composé d'un module pour lancer des requêtes dans une base de données et un module pour vérifier l'intégrité de celle-ci est plus modulaire qu'un seul module qui fait ces deux tâches en même temps.

Les plus grands bénéfices des systèmes modulaires sont : réutilisabilité, facilité de modification et meilleur contrôle de la complexité. Parmi les critères les plus utilisés pour évaluer à quel point un logiciel est modulaire se situent les notions informelles de cohésion et couplage [10, 9].

La *cohésion* traduit à quel point les pièces d'un seul composant sont en relation les unes avec les autres. Un module est cohésif lorsqu'au haut niveau d'abstraction il ne fait qu'une seule et précise tâche. Plus un module est centré sur un seul but, plus cohésif il est. Il est difficile de comprendre et réutiliser un module multifonction, pendant qu'un module avec haute cohésion est plus simple de réutiliser et d'étendre ultérieurement.

Le *couplage* s'agit de l'interdépendance entre un module en particulier et son entourage. Un bas couplage veut dire maintenir les dépendances (partage d'information) entre des composants à un niveau minimum. Un projet qui a un bas couplage est plus facile d'entretenir, puisque la probabilité d'y avoir des effets collatéraux et d'affecter une partie importante du système est considérablement réduite.

A l'unanimité en projet de logiciel, la meilleur façon de concevoir un programme est en y garantissant un bas couplage et une haute cohésion.

2 Modules naturels en descriptions

Comme directive pour achever modularité on utilise le principe d'*occultation d'information* : des entités différentes doivent être séparées en des modules différents, et chaque module ne doit pas avoir d'accès direct aux contenus des autres. En raisonnement sur les actions, accéder à un module veut dire performer des tâches de raisonnement telles que prédiction, postdiction, planification, entre autres. Cela consiste à utiliser ses formules logiques dans des inférences. Dans cette section nous présentons l'ontologie des descriptions de domaine et la façon dont on partage en des modules leurs ensembles d'axiomes.

Toute description de domaine contient une représentation des effets des actions. On appelle *lois d'effets* les formules mettant en relation une action et ses effets. Les énoncés des conditions qui rendent une action pas exécutable sont

*Financé par le gouvernement de la RÉPUBLIQUE FÉDÉRATIVE DU BRÉSIL. Bourse d'études CAPES BEX 1389/01-7.

appelés *lois d'inexécutabilité*. Les *lois d'exécutabilité* stipulent le contexte où l'exécution d'une action est garantie. Finalement, les *lois statiques* sont des formules ne mentionnant pas d'action et qui expriment les contraintes qui doivent être respectées dans chaque état possible du monde.

En regardant une description de domaine comme un logiciel, on peut imaginer son organisation d'un point de vue orienté-objet et en faire une sorte de diagramme de classes. Ceci est illustré par la figure 1 qui montre les relations entre les différents types d'entités.

Une description de domaine consiste d'une description des effets des actions, leurs non-effets, exécutabilités, inexécutabilités et aussi des lois statiques du domaine. Parmi les effets des actions, on peut distinguer les *effets directs* et les *effets indirects* (ramifications). Selon la terminologie orienté-objet, un effet direct *est un* effet, le même étant vrai pour les effets indirects. Les effets, les non-effets, les exécutabilités, les inexécutabilités et lois statiques sont des types particuliers de descriptions de domaine.

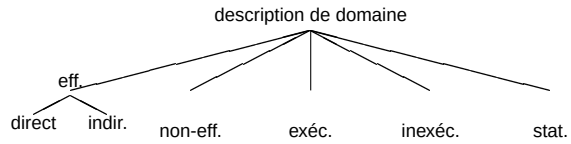

Figure 1: « Diagramme de classes » des modules en projet de descriptions de domaine. Les liens représentent des relations *est-un*.

Les non-effets des actions sont liés au *problème du décor* [7], et les effets indirects au *problème de la ramification* [1]. Ici nous faisons abstraction de ces problèmes-là et supposons que nous avons une relation de conséquence logique assez puissante pour en dériver les conclusions attendues. On suppose donné une relation de conséquence « dopée » $\mid\!\approx$ avec laquelle tous les axiomes de décor et effets indirects peuvent être dérivés, et on l'utilise désormais. (Pour une approche différente, regarder [3].) Par exemple nous avons $\mid\!\approx \textit{chargé}(s) \rightarrow \textit{chargé}(\textit{do}(\textit{attendre}, s))$ (c'est-à-dire attendre ne change pas le statu de *chargé*), et

$$\left\{ \begin{array}{c} \textit{enmarche}(S_0), \\ \textit{enmarche}(s) \rightarrow \textit{vivant}(s), \\ \neg\textit{vivant}(\textit{do}(\textit{tirer}, s)) \end{array} \right\} \mid\!\approx \neg\textit{enmarche}(\textit{do}(\textit{tirer}, S_0))$$

(tirer a l'effet indirect de que la victime ne marche plus.)

On s'intéresse, donc, aux effets directs (désormais effets), inexécutabilités, exécutabilités et lois statiques. Cela nous introduisons formellement dans ce qui suit.

Lois d'effet Les approches logiques pour le raisonnement sur les actions contiennent des expressions qui mettent en relation chaque action et ses effets. On suppose que tels effets peuvent aussi être conditionnels. Donc, une *loi d'effet* pour l'action a est de la forme

$$\textit{Poss}(a, s) \rightarrow (\Phi(s) \rightarrow \Phi(\textit{do}(a, s)))$$

où $\Phi(s)$ est une *formule simple d'état sur la situation* s, et $\Phi(\textit{do}(a, s))$ une *formule simple d'état sur* $\textit{do}(a, s)$ [6].

Un exemple de loi d'effet est

$\textit{Poss}(\textit{tirer}, s) \rightarrow (\textit{chargé}(s) \rightarrow \neg\textit{vivant}(\textit{do}(\textit{tirer}, s)))$,

en disant que toujours que *tirer* est exécutable et que l'arme est chargé, alors après tirer la dinde est morte. Un autre exemple est $\textit{Poss}(\textit{attirer}, s) \rightarrow \textit{enmarche}(\textit{do}(\textit{attirer}, s))$: le résultat d'attirer la dinde est qu'elle se met en marche.

Lois d'inexécutabilité Le projet de descriptions de domaine doit aussi prendre en compte l'expression des qualifications des actions, c'est-à-dire les conditions sous lesquelles une certaine action ne peut pas être exécutée du tout. Une *loi d'exécutabilité* pour l'action a est de la forme

$$\Phi(s) \rightarrow \neg\textit{Poss}(a, s)$$

où $\Phi(s)$ est une formule simple d'état sur la situation s. Par exemple, $\neg\textit{arme}(s) \rightarrow \neg\textit{Poss}(\textit{tirer}, s)$ établie que *tirer* ne peut pas être exécuté si on n'a pas d'arme.

Lois statiques Les approches prenant en compte les effets indirects des actions utilisent des formules caractérisant les propositions invariables à propos du monde et qui déterminent l'ensemble d'états possibles. Une *loi statique* est une formule simple d'état sur un terme de situation s. Comme exemple, $\textit{enmarche}(s) \rightarrow \textit{vivant}(s)$ dit que si une dinde marche, alors elle est vivante [11].

Lois d'exécutabilités Qu'avec des lois statiques et d'effets on ne peut pas garantir que *tirer* est exécutable si on a une arme. Une *loi d'exécutabilité* pour l'action a est de la forme $\Phi(s) \rightarrow \textit{Poss}(a, s)$, où $\Phi(s)$ est une formule simple d'état sur s. Par exemple $\textit{arme}(s) \rightarrow \textit{Poss}(\textit{tirer}, s)$ établie que l'on peut tirer pourvu que l'on aie une arme, et $\textit{Poss}(\textit{attirer}, s)$ que la dinde peut toujours être attirée.

Descriptions de domaine Nous regroupons les quatre types d'entités définies ci-dessus de la façon suivante : pour une action a, $\mathbf{Eff}_a$ est l'ensemble de ses lois d'effet, $\mathbf{Inex}_a$ celui de ses lois d'inexécutabilité, et $\mathbf{Exe}_a$ est l'ensemble de ses lois d'exécutabilité. $\mathbf{Stat}$ dénote l'ensemble des lois statiques du domaine. Donc $\mathbf{Eff}_a$, $\mathbf{Exe}_a$ et $\mathbf{Inex}_a$, pour chaque a, et $\mathbf{Stat}$ sont les modules naturels que l'on considère dans la description d'un domaine.

Pour simplifier la notation, on définie $\mathbf{Eff} = \bigcup \mathbf{Eff}_a$, $\mathbf{Inex} = \bigcup \mathbf{Inex}_a$, et $\mathbf{Exe} = \bigcup \mathbf{Exe}_a$. Nous supposons que tous ces ensembles sont consistants.

En rappelant notre modélisation orienté-objet d'une description de domaine, on peut voir $\mathbf{Eff}$, $\mathbf{Exe}$, $\mathbf{Inex}$ et $\mathbf{Stat}$, respectivement, comme des « instances » des classes *effets directs*, *inexécutabilités*, *exécutabilités* et *lois statiques* que l'on a présentées au début de cette section. La figure 2 illustre cette caractérisation.

Une *description de domaine* $\mathbf{D}$ est un tuplet de la forme $\langle \mathbf{Eff}, \mathbf{Inex}, \mathbf{Exe}, \mathbf{Stat} \rangle$.

3 Description de domaine modulaire

Si l'information d'un module n'est pas mélangée avec celle des autres, on peut espérer que des effets collatéraux indésirables dûs à des modifications ultérieures sont moins

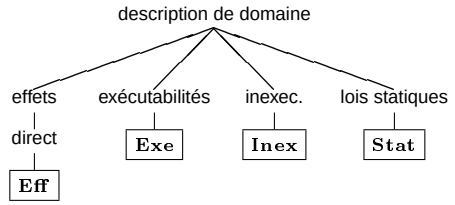

Figure 2: « Instances » des modules d'une description de domaine.

probables de se propager vers d'autres parties de la description de domaine. Le même on peut dire sur le test de consistance si au delà d'être séparés les modules sont aussi projetés de façon que leur interaction soit minimisée.
Comme on a vu, en génie du logiciel on évalue la modularité à travers deux critères qualitatifs : la cohésion, qui indique la « force interne » d'un module, et le couplage, qui montre l'interdépendance relative entre des modules différents. Ces notions-là sont assez informelles et floues, même en génie du logiciel, et ne peuvent pas être mesurées de façon objective. Néanmoins, nous les exploitons ici lorsqu'on les applique dans l'analyse des descriptions de domaine, et montrons que ces concepts informels du génie du logiciel peuvent s'avérer utiles pour le test de consistance de plusieurs configurations différentes des modules.

3.1 Cohésion

En général la cohésion est conséquence directe de la modularisation, et son évaluation dépend surtout des entités que l'on prend en compte lorsqu'on formalise un domaine.
Quand on parle d'ensembles de formules logiques on prend par cohésion à quel point un module logique est simple, bien défini, tout en considérant les différents types de formules que l'on peut en dériver. *Le moins de types de lois un modules entraîne, le plus cohésif il est.*
Comme un exemple, soit le module suivant :

$$\left\{ \; \mathit{arme}(s) \leftrightarrow \mathit{Poss}(\mathit{tirer}, s) \; \right\}$$

De tel ensemble tout seul on peut dériver $\neg \mathit{arme}(s) \rightarrow \neg \mathit{Poss}(\mathit{tirer}, s)$ et $\mathit{arme}(s) \rightarrow \mathit{Poss}(\mathit{tirer}, s)$, des formules de deux types différents. Alors, on dit que cet ensemble a une basse cohésion, puisque tout seul il sert à dériver des exécutabilités et des inexécutabilités. Une meilleur approche serait de le décomposer dans les deux suivants :

$$\mathbf{Inex}_{\mathit{tirer}} = \{\neg \mathit{arme}(s) \rightarrow \neg \mathit{Poss}(\mathit{tirer}, s)\}$$

$$\mathbf{Exe}_{\mathit{tirer}} = \{\mathit{arme}(s) \rightarrow \mathit{Poss}(\mathit{tirer}, s)\}$$

Cohésion totale n'est pas toujours facile à obtenir. Supposons, par exemple, une situation où on raisonne à propos des effets de boire une tasse de café :

$$\mathbf{Eff}_{\mathit{boire}} = \left\{ \begin{array}{l} \mathit{Poss}(\mathit{boire}, s) \rightarrow \\ (\mathit{sucre}(s) \rightarrow \mathit{content}(\mathit{do}(\mathit{boire}, s)), \\ \mathit{Poss}(\mathit{boire}, s) \rightarrow \\ (\mathit{sel}(s) \rightarrow \neg \mathit{content}(\mathit{do}(\mathit{boire}, s)) \end{array} \right\}$$

$\mathbf{Eff}_{\mathit{boire}}$ entraîne $(\mathit{sucre}(s) \wedge \mathit{sel}(s)) \rightarrow \neg \mathit{Poss}(\mathit{boire}, s)$. Cela veut dire que de $\mathbf{Eff}_{\mathit{boire}}$ on n'obtient pas que des lois d'effets, mais aussi des inexécutabilités. Donc, $\mathbf{Eff}_{\mathit{boire}}$ n'est pas si cohésif que l'on croyait.
Une façon d'augmenter la cohésion d'un module de lois d'effet c'est spécifier complètement les préconditions des effets de chaque action. Par exemple, si on considère la version affaiblie de $\mathbf{Eff}_{\mathit{boire}}$ suivante

$$\mathbf{Eff}'_{\mathit{boire}} = \left\{ \begin{array}{l} \mathit{Poss}(\mathit{boire}, s) \rightarrow \\ (\mathit{sucre}(s) \wedge \neg \mathit{sel}(s)) \rightarrow \mathit{content}(\mathit{do}(\mathit{boire}, s)), \\ \mathit{Poss}(\mathit{boire}, s) \rightarrow \\ (\mathit{sel}(s) \wedge \neg \mathit{sucre}(s)) \rightarrow \neg \mathit{content}(\mathit{do}(\mathit{boire}, s)) \end{array} \right\}$$

on garantie une cohésion plus grande que celle de départ.
On résume cela dans le principe de projet suivant :

P1. **Cohésion maximale** : *Chaque module d'une description de domaine doit être conçu de façon à maximiser sa cohésion.*

3.2 Couplage

Le concept de couplage évalue à quel point un module est attaché ou dépendant d'autres modules.
On définie couplage d'un ensemble de formules comme le tant d'interaction qui est nécessaire entre le module en question et les autres pour en dériver une formule d'un type en particulier. Interaction, dans notre sens, veut dire partage de formules logiques. *Le moins d'interaction entre des modules, le moins couplés ils sont.* Comme exemple, considérons la description de domaine $\mathbf{D}_1$ qui suit :

$$\mathbf{Eff}_1 = \left\{ \begin{array}{l} \mathit{Poss}(\mathit{attirer}, s) \rightarrow \mathit{enmarche}(\mathit{do}(\mathit{attirer}, s)), \\ \mathit{Poss}(\mathit{tirer}, s) \rightarrow \\ (\mathit{chargé}(s) \rightarrow \neg \mathit{vivant}(\mathit{do}(\mathit{tirer}, s))) \end{array} \right\}$$

$$\mathbf{Inex}_1 = \{\neg \mathit{vivant}(s) \rightarrow \neg \mathit{Poss}(\mathit{attirer}, s)\}, \mathbf{Exe}_1 = \emptyset$$

$$\mathbf{Stat}_1 = \left\{ \begin{array}{c} \mathit{enmarche}(s) \rightarrow \mathit{vivant}(s), \\ \mathit{mort}(s) \leftrightarrow \neg \mathit{vivant}(s) \end{array} \right\}$$

Pour en dériver la loi statique $\mathit{enmarche}(s) \rightarrow \neg \mathit{mort}(s)$, on a besoin que de $\mathbf{Stat}_1$, c'est-à-dire aucun autre module est strictement nécessaire pour que cela se produise. Par contre, pour en conclure $\mathit{mort}(s) \rightarrow \neg \mathit{Poss}(\mathit{attirer}, s)$ on a besoin de $\mathbf{Stat}_1$ et $\mathbf{Inex}_1$ tous les deux.
Des descriptions totalement découplées ne sont pas communes dans des applications pratiques. Pour l'exemple ci-dessus, il semble être impossible de diminuer l'interaction entre $\mathbf{Stat}_1$ et $\mathbf{Inex}_1$ sans que cela ne rende la description presque inutile. Cependant, si $\mathbf{Exe}_1$ dans l'exemple contenait $\mathit{Poss}(\mathit{attirer}, s)$, alors cela changerait tout : dans ce cas-là, avec $\mathbf{Inex}_1$ on dériverait la loi statique $\mathit{vivant}(s)$, mais celle-ci n'est pas dérivable de $\mathbf{Stat}_1$ tout seul. C'est-à-dire qu'un niveau plus grand d'interaction entre ce module et les autres est nécessaire pour aboutir à cela. Donc on dirait qu'il y a un haut couplage entre les composants de $\mathbf{D}_1$. On résume cela dans un second principe de projet :

P2. **Couplage minimal** : *Les modules d'une description de domaine doivent être conçus de façon à minimiser leur couplage*.

3.3 Vers des critères formels

Au même temps que des descriptions de domaine comportant des modules cohésifs sont plus intelligibles, en ayant un bas couplage entre tels modules diminue la complexité des effets indirects qui peuvent y parvenir lorsqu'on y procède à des changements. Dans le reste de ce papier on donne quelques justificatifs pour cela en présentant des critères plus raffinés. (Pour plus de détails du point de vue formel cf. [4]).
Notre hypothèse centrale est de que les différents types d'axiomes doivent être nettement sépares et n'intervenir que dans un sens, c'est-à-dire les lois statiques avec les lois d'effets peuvent avoir des conséquences qui ne suivent pas des lois d'effets toute seules, mais par contre les lois d'effets ne doivent pas permettre du tout qu'on en dérive des lois statiques, des exécutabilités, etc.
En général, les principes P1 et P2 sont violés par des descriptions de domaine formalisées dans tous les approches connus jusqu'à présent dans la littérature qui permettent la représentation des quatre types de lois que l'on considère ici. Nous discutons, donc, chacun de ces principes à travers des exemples.

4 Cohésion maximale

D'après la définition de $\mathbf{Stat}$, c'est facile à voir que de tel module on peut dériver n'importe quel type de formule, ce qui fait que $\mathbf{Stat}$ est par sa propre définition un module peu cohésif.
$\mathbf{Inex}$ et $\mathbf{Exe}$, par contre, sont maximalement cohésifs : ni $\mathbf{Inex}$ ni $\mathbf{Exe}$ entraîne tout seul des lois statiques. En plus, on a les résultats suivants :

Théorème 4.1
Si $\mathbf{Inex} \not\approx \Phi(s) \rightarrow \mathit{Poss}(a,s)$, alors $\mathbf{Inex} \not\approx \neg\Phi(s)$.

Théorème 4.2
Si $\mathbf{Inex} \not\approx \mathit{Poss}(a,s) \rightarrow (\Phi(s) \rightarrow \Phi(\mathit{do}(a,s)))$,
alors $\mathbf{Inex} \not\approx \mathit{Poss}(a,s) \rightarrow \neg\Phi(s)$.

c'est-à-dire si $\mathbf{Inex}$ entraîne une exécutabilité, alors elle est un théorème de la logique de base ; si il entraîne une loi d'effet, celle-ci est superflue. Des versions pour $\mathbf{Exe}$ peuvent aussi être énoncées.
Les choses deviennent plus compliquées lorsqu'il s'agit de l'ensemble de lois d'effets. D'un côté on a

Théorème 4.3 Si $\mathbf{Eff} \not\approx \Phi(s) \rightarrow \mathit{Poss}(a,s)$, alors $\mathbf{Eff} \not\approx \neg\Phi(s)$.

(c'est-à-dire les exécutabilités dérivés de $\mathbf{Eff}$ sont des théorèmes de la logique).

Théorème 4.4 $\mathbf{Eff} \not\approx \Phi(s)$, pour toute formule simple d'état $\Phi(s)$.

Cependant, comme on a vu dans la section 3.1, il est possible qu'on en dérive des inexécutabilités de $\mathbf{Eff}$. Etablir cohésion de $\mathbf{Eff}$ dans ce cas-là demande une révision des préconditions des effets des actions. De toutes les façons, concevoir un algorithme pour accomplir cette tâche n'est pas difficile (en fonction de la limite de pages, on omet la présentation de l'algorithme ici).

5 Couplage minimal

Le principe de couplage minimal P2 peut être raffiné en deux principes plus spécifiques :

P2-1. **Pas d'inexécutabilité implicite**:
si $\mathbf{D} \not\approx \Phi(s) \rightarrow \neg\mathit{Poss}(a,s)$, alors
$\mathbf{Inex}, \mathbf{Stat} \not\approx \Phi(s) \rightarrow \neg\mathit{Poss}(a,s)$

Si une loi d'inexécutabilité peut être inférée de la description de domaine, alors elle doit être dérivable que des ensembles de lois statiques et d'inexécutabilités.

P2-2. **Pas de loi statique implicite**:
si $\mathbf{D} \not\approx \Phi(s)$, alors $\mathbf{Stat} \not\approx \Phi(s)$.

Si une loi statique est conséquence logique de la description de domaine, alors elle doit l'être aussi de l'ensemble de lois statiques tout seul.

5.1 Pas de loi d'inexécutabilité implicite

Considérons la description de domaine $\mathbf{D}_2$ suivante :

$$\mathbf{Eff}_2 = \left\{ \begin{array}{l} \mathit{Poss}(\mathit{attirer}, s) \rightarrow \mathit{enmarche}(\mathit{do}(\mathit{attirer}, s)), \\ \mathit{Poss}(\mathit{tirer}, s) \rightarrow \\ (\mathit{chargé}(s) \rightarrow \neg\mathit{vivant}(\mathit{do}(\mathit{tirer}, s))) \end{array} \right\}$$

$$\mathbf{Inex}_2 = \mathbf{Exe}_2 = \emptyset,\ \mathbf{Stat}_2 = \{\mathit{enmarche}(s) \rightarrow \mathit{vivant}(s)\}$$

De $\mathit{Poss}(\mathit{attirer}, s) \rightarrow \mathit{enmarche}(\mathit{do}(\mathit{attirer}, s))$ il suit avec $\mathbf{Stat}_2$ que $\mathit{Poss}(\mathit{attirer}, s) \rightarrow \mathit{vivant}(\mathit{do}(\mathit{attirer}, s))$, c'est-à-dire dans toutes les situations, après attirer la dinde elle est vivante :

$$\mathbf{Eff}_2, \mathbf{Stat}_2 \not\approx \mathit{Poss}(\mathit{attirer}, s) \rightarrow \mathit{vivant}(\mathit{do}(\mathit{attirer}, s))$$

Maintenant, comme nous avons que $\mathbf{D}_2 \not\approx \neg\mathit{vivant}(s) \rightarrow \neg\mathit{vivant}(\mathit{do}(\mathit{attirer}, s))$, le statu du fluent *vivant* n'est pas modifié par l'action *attirer*, ce qui fait que l'on aie $\mathbf{Eff}_2, \mathbf{Stat}_2 \not\approx (\mathit{Poss}(\mathit{attirer}, s) \wedge \neg\mathit{vivant}(s)) \rightarrow (\mathit{vivant}(\mathit{do}(\mathit{attirer}, s)) \wedge \neg\mathit{vivant}(\mathit{do}(\mathit{attirer}, s)))$. A partir de cela il suit que $\mathbf{D}_2 \not\approx \neg\mathit{vivant}(s) \rightarrow \neg\mathit{Poss}(\mathit{attirer}, s)$, c'est-à-dire la dinde ne peut pas être attirée si elle morte. Mais $\mathbf{Inex}_2, \mathbf{Stat}_2 \not\not\approx \neg\mathit{vivant}(s) \rightarrow \neg\mathit{Poss}(\mathit{attirer}, s)$, donc le principe P2-1 est violé. La formule $\neg\mathit{vivant}(s) \rightarrow \neg\mathit{Poss}(\mathit{attirer}, s)$ est un exemple de ce que l'on appelle une *inexécutabilité implicite*.

5.2 Pas de loi statique implicite

Les lois d'exécutabilités augmentent le pouvoir expressif du langage, mais par contre peuvent rentrer en conflit avec les inexécutabilités. Par exemple, soit $\mathbf{D}_3$ tel que $\mathbf{Eff}_3 = \mathbf{Eff}_2$, $\mathbf{Inex}_3 = \{\neg \mathit{vivant}(s) \rightarrow \neg \mathit{Poss}(\mathit{attirer}, s)\}$, $\mathbf{Exe}_3 = \{\mathit{Poss}(\mathit{attirer}, s)\}$, et $\mathbf{Stat}_3 = \mathbf{Stat}_2$. (Observez que le principe P2-1 est satisfait.) On en conclue $\mathbf{Inex}_3, \mathbf{Exe}_3 \not\models \mathit{vivant}(s)$: la dinde est immortelle ! Cela est une *loi statique implicite*, car $\mathit{vivant}(s)$ n'est pas une conséquence de $\mathbf{Stat}_3$ tout seul : P2-2 est violé.
L'existence de lois statiques implicites est donc un indice de que les exécutabilités sont trop fortes : dans notre exemple, on a eu tort d'avoir supposé que *attirer* est exécutable dans n'importe quel contexte. Sinon, il se peut aussi que de leurs côté les inexécutabilités sont trop fortes, ou bien que les lois statiques elles-mêmes sont trop faibles.

6 Conséquences de la modularité

Vérifier si une description de domaine satisfait les principes P2-1–P2-2 peut se faire avec une adaptation du matériel sur le sujet présent dans la littérature [3, 5]. Cependant nous ne plongeons pas là-dedans maintenant et juste présentons les principaux résultats que l'on obtient lorsque l'on considère des descriptions de domaine satisfaisant les principes de projet que l'on a proposé.
Etre modulaire s'avère une propriété utile des descriptions de domaine : au delà d'être une directive de projet qui aide à éviter des erreurs, elle clairement sert à restreindre l'espace de recherche, ce qui rend le processus de raisonnement plus simple. Pour voir, si $\mathbf{D}$ est une description modulaire, alors, sa consistance se résume à celle de $\mathbf{Stat}$:

Théorème 6.1 Si $\mathbf{D}$ n'a pas de loi statique implicite, alors $\mathbf{D} \not\models \bot$ ssi $\mathbf{Stat} \models \bot$.

Théorème 6.2 Si $\mathbf{D}$ n'a pas de loi statique implicite, alors $\mathbf{D} \not\models \mathit{Poss}(a, s) \rightarrow (\Phi(s) \rightarrow \Phi(\mathit{do}(a, s)))$ ssi $\mathbf{Eff}_a, \mathbf{Inex}_a, \mathbf{Stat} \not\models \mathit{Poss}(a, s) \rightarrow (\Phi(s) \rightarrow \Phi(\mathit{do}(a, s)))$.

Cela veut dire que sous P2-2 on a modularité à l'intérieur de $\mathbf{Eff}$ aussi : pour déduire les effets d'une action a on n'a pas besoin de prendre en compte les lois pour les autres actions. Des versions pour exécutabilités et inexécutabilités peuvent aussi être énoncées.

7 Conclusion

On a établie un lien entre génie des connaissances et génie du logiciel en montrant que des concepts et techniques développés pour celui-ci sont aussi utiles dans la conception de descriptions de domaine. En particulier, avec les concepts de cohésion et couplage, on a obtenu de meilleurs critères pour l'évaluation de descriptions de domaine.
Sous un premier regard, à cause de l'interaction entre $\mathbf{Stat}$ et les autres modules, on pourrait dire que des descriptions de domaines formalisées de tel façon ne minimisent pas le couplage. Toutefois, étant donné la structure intrinsèque de $\mathbf{Stat}$, il faut bien noter que l'on ne pourrait pas faire autrement : de la même façon que c'est pas possible d'écrire des programmes complètement découplés, on ne peut pas avoir de descriptions de domaines totalement découplées (à moins que l'on se restreigne à des domaines sans ramifications, comme dans [8]).
On peut argumenter que des conséquences non-intuitives dans des descriptions de domaine sont dues plutôt à des axiomes mal écrits qu'au manque de modularité. Certes, mais ce que l'on a présenté ici c'est le cas où rendre une description de domaine modulaire nous donne un outil pour détecter au moins quelques de tels problèmes et les corriger. (On ne se propose pas à corriger des axiomes mal écrits d'une fois pour toutes. Pour cela, dans [2] on présente une méthode de mise-à-jour de théories d'action.) Au delà de ça, avoir des entités séparées et contrôler leur interaction permet de mieux localiser des éventuels problèmes, ce qui peut être crucial pour des applications d'intérêt pratique.

References

[1] J. J. Finger. *Exploiting constraints in design synthesis*. Thèse de Doctorat, Stanford University, 1987.

[2] A. Herzig, L. Perrussel, et I. Varzinczak. Elaborating domain descriptions. *Proc. ECAI'06*, Riva del Garda, 2006. IOS Press.

[3] A. Herzig et I. Varzinczak. Domain descriptions should be modular. *Proc. ECAI'04*, pages 348–352, Valencia, 2004. IOS Press.

[4] A. Herzig et I. Varzinczak. Cohesion, coupling and the meta-theory of actions. *Proc. IJCAI'05*, pages 442–447, Edinburgh, 2005. Morgan Kaufmann.

[5] A. Herzig et I. Varzinczak. On the modularity of theories. *Advances in Modal Logic*, volume 5, pages 93–109. King's College Publications, 2005. Papiers selectionés de AiML 2004.

[6] F. Lin. Embracing causality in specifying the indirect effects of actions. *Proc. IJCAI'95*, pages 1985–1991, 1995.

[7] J. McCarthy et P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, volume 4, pages 463–502. 1969.

[8] F. Pirri et R. Reiter. Some contributions to the metatheory of the situation calculus. *Journal of the ACM*, 46(3):325–361, 1999.

[9] R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 1992.

[10] I. Sommerville. *Software Engineering*. Addison Wesley, 1985.

[11] M. Thielscher. Computing ramifications by postprocessing. *Proc. IJCAI'95*, pages 1994–2000, 1995.