MASTER EN SCIENCES INFORMATIQUES

Conception d'un outil d'aide à l'analyse par la méthode des Problem frames de Jackson

Gonze, Michel; Martinez, Federico

Award date:
2005

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'Informatique
Année académique 2004 - 2005

Conception d'un outil d'aide à l'analyse par la méthode des Problem Frames de Jackson

Michel Gonze et Federico Martinez

Mémoire présenté en vue de l'obtention du grade de Licencié en Informatique.

Résumé

La méthode des problem frames de Jackson est utilisée dans l'analyse et la structuration des problèmes de développement logiciel. Les problem frames constituent un moyen pour appréhender la complexité inhérente aux problèmes informatiques.

L'objet de notre travail consiste à définir les éléments nécessaires à l'implémentation d'un outil d'aide à l'analyse par la méthode des problem frames de Jackson.

Nous définissons un langage formel basé sur la notation graphique de Jackson. Nous décrivons ensuite les fonctionnalités de l'éditeur en mettant l'accent sur les fonctionnalités plus spécifiques aux problem frames.

Nous fournissons un cadre théorique pour l'intégration de notations externes qui s'appuie sur un modèle de sémantique fonctionnelle qui permet de vérifier la cohérence entre un problème et ses sous-problèmes. Nous terminons en prenant pour exemple d'intégration de notation les statecharts de Harel.

Mots clés

Problem Frame, Outil, Analyse, Sémantique, Statechart

Abstract

Jackson's Problem frames method is used in analysing and structuring software development problems. It offers the possibility of representing the complexity that is inherent to many computer based problems.

The principal objective of our work is to establish the requirements for a Computer Aided Analysis tool based on Jackson's Problem Frame method.

We define a formal language based on Jackson's graphical notation. We describe the editor's functionalities, emphasizing functionalities that are specific for tackling Problem Frames.

We provide a theoretical framework for integrating external notations that use a functional semantic model in order to validate the coherence between a problem and its constituent sub-problems. Finally, we illustrate this integration with the Harel's statecharts.

Keywords

Problem Frame, Tool, Requirements Engineering, Sémantics, Statechart

Remerciements

Nous tenons à remercier tout particulièrement Monsieur Schobbens, notre promoteur qui nous a proposé ce sujet, pour sa disponibilité et ses précieux conseils qui nous ont orienté tout au long de notre travail.

Nous remercions également G. Delannay qui a accepté de suivre le déroulement de notre travail et nous a permis d'insèrer son métamodèle dans les annexes de notre travail.

Enfin, nous tenons à remercier nos compagnes qui nous ont soutenu en permanence pendant l'élaboration de ce travail et qui nous ont permis de surmonter les difficultés avec tant de bienveillance, et nos amis qui se sont continuellement montrés à notre écoute.

Table des matières

Introduction

Les problem frames identifient et décrivent des situations récurrentes dans les problèmes informatiques. A la manière des design patterns, qui s'appliquent aux solutions informatiques, les problem frames sont des classes de problèmes qui fournissent une taxinomie dans laquelle chaque nouvelle expérience et connaissance acquise est assignée à l'endroit adéquat et peut être partagée de manière appropriée.

Les problèmes informatiques sont complexes et variés et les praticiens qui connaissent en même temps le domaine d'application et les techniques d'analyse sont rares. Partant de ce constat, Jackson a défini sa méthode d'analyse en se basant sur deux principes :

- Les problèmes complexes peuvent être décomposés en sous-problèmes plus simples correspondant à des classes de problèmes connus : les problem frames.
- Les descriptions d'analyse faites dans des langages appropriés peuvent être réunies par un dénominateur commun.

Le but de notre travail est de définir les concepts essentiels nécessaires à la conception d'un outil d'aide à l'analyse par la méthode des problem frames de Jackson.

Un tel outil devra permettre notamment de spécifier des problèmes et leur décomposition en sous-problèmes de classes particulières, de vérifier la cohérence entre le problème et les sous-problèmes et d'intégrer des descriptions réalisées à l'aide de notations externes.

Notre travail est organisé comme suit :

- Les problem frames sont décrits par Jackson dans [1] et [2]. Nous en présentons les principes essentiels dans le premier chapitre.
- Le métamodèle constitue la base d'un outil sur laquelle sont construites les fonctionnalités. Nous reprenons le métamodèle défini par G. Delannay. Il définit un langage pour problem frames. Nous complétons, dans le deuxième chapitre, les points laissés en suspens.
- Nous définissons ensuite dans le troisième chapitre les fonctionnalités. Outre les fonctionnalités classiques d'un éditeur graphique, que nous

présentons de manière formelle dans les annexes B et C, nous définissons des fonctionnalités spécifiques aux problem frames.

- Dans le quatrième chapitre, nous définissons un cadre général pour l'intégration d'une notation externe.
- L'intégration d'une notation est illustrée dans le cinquième chapitre par un exemple. Notre choix s'est porté sur le statecharts.
- Nous avons ajouté un dernier chapitre suite à la parution d'un article [HRJ] définissant une sémantique pour les problem frames. Nous effectuons une comparaison à posteriori avec notre travail.

Chapitre 1

Introduction à la méthode des Problem Frames

La méthode des problem frames est utilisée dans l'analyse et la structuration des problèmes de développement de logiciels.

Nous allons tout d'abord décrire les concepts clés utilisés par Jackson. Ils seront introduits l'un après l'autre, chacun d'eux ne se basant que sur des concepts présentés auparavant.

Ensuite nous décrirons les besoins qui ont conduit Jackson à élaborer une telle méthode. Le fil conducteur est la mise en place d'un système de raisonnement permettant à l'analyste d'utiliser les propriétés de l'environnement pour répondre aux exigences du client :

- L'analyste doit comprendre l'environnement le plus précisément possible et détecter les leviers sur lesquels il peut s'appuyer pour montrer les effets désirés.
- L'analyste doit délimiter son champ d'investigation et ne décrire que les éléments qui lui serviront dans le raisonnement.
- L'analyste ne connaît pas tous les problèmes qu'il doit décrire ; il faut qu'il puisse se reposer sur une taxinomie des problèmes pour qu'il puisse associer un problème donné avec un type de problème, chaque type de problème étant accompagné de "recettes" de résolution.
- Parallèlement, les problèmes complexes peuvent être vu sous plusieurs angles, chaque angle de vision du problème complexe donnant lieu à une projection sur un type de problème simple.

Enfin, nous décrirons les différentes étapes de la méthode de Jackson.

1.1 Vocabulaire de Jackson

"*The key idea (...) is the decomposition of complex and realistic problems into structures of simple subproblems of kinds that fit recognized problem frames.*" [2, p. xvii].

Les problèmes considérés sont les problèmes informatiques. Les aspects sociaux, politiques, éthiques et économiques sont ignorés [2, p. xiii].

Les **problèmes informatiques** sont de nature très diverses et peuvent survenir dans des contextes très différents. Presque toutes les parties de notre environnement physique et humain peuvent fournir la matière première et le contexte pour un problème informatique [2, p. xi].

L'analyse d'un problème est composé de **descriptions**. Celles-ci décrivent des parties du problème à l'aide de phénomènes remarquables (les parties informelles de la description) et de construction logique sur ces phénomènes (les parties formelles)[1, p.58]. Les **phénomènes** sont des faits, de simples vérités dans le monde, les plus petites unités observables et significatives [1, p. 143].

Les descriptions sont soient **descriptives** (description de ce qui est), soient **prescriptives** (description de ce qui devrait être) [Kovitz, p.42][1]

Un projet informatique destiné à résoudre un problème est un projet de construction d'une ou plusieurs **machines**, la description précise du comportement et des propriétés qui transformera un ordinateur en une machine qui sera utile dans un environnement particulier [1, p.156]. L'ordinateur est capable de se transformer en cette machine car il possède trois propriétés : il est programmable, il peut faire des calculs et il est équipé d'interfaces standards [1, p. 110].

La partie de l'environnement qui affectera la machine et où celle-ci agira est le **domaine d'application**. Cet environnement est spécifique à un problème donné ; c'est lui qui contient les exigences du client. Le domaine d'application est la matière que la machine doit façonner pour satisfaire les exigences du client [1, p.10].

La partie du monde où la machine est installée et où les effets de celle-ci seront ressentis et évalués est le **contexte du problème**.

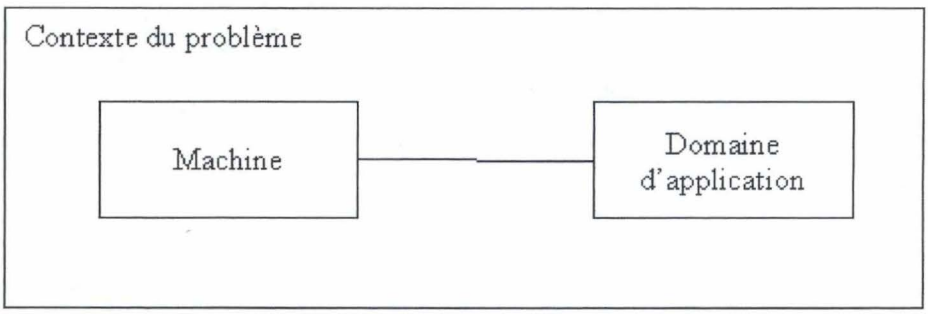
Les domaines sont les différentes parties qui constituent le domaine d'application. Un **domaine** est une partie particulière du monde qui peut être distinguée car elle forme un tout relativement cohésif, séparé du reste du monde [1, p.73].

Les **exigences** concernent les phénomènes du domaine d'application.

[1] Jackson utilise les qualificatifs "descriptif" et "optatif" ; nous préférons la terminologie plus familière utilisée par Ben Kovitz

Pour les décrire correctement, on décrit les relations entre ces phénomènes [1, p.169].

La machine peut satisfaire les exigences car elle partage des phénomènes avec le domaine d'application [1, p.156].



Les problèmes informatiques sont **complexes**. Ils peuvent être décomposés en parties plus simples qu'il est plus facile de décrire.

Les **sous-problèmes** peuvent être apparentés à des classes de problèmes connus, les problem frames.

Un **problem frame** (aussi abrégé PF) définit la forme d'un problème par :

- La capture des caractéristiques et des interconnexions des parties de l'environnement qui le concerne.
- Les aspects importants et les difficultés qui sont susceptibles d'apparaître.

A la manière des design patterns, les problem frames identifient et décrivent des situations récurrentes dans l'environnement d'un problème. Ils fournissent une taxinomie dans laquelle chaque nouvelle expérience et connaissance acquise est assignée à l'endroit adéquat et peut être partagée de manière appropriée [2, p.xii].

Lorsqu'un sous-problème correspond (**fits**) à un problem frame donné, il est susceptible d'être résolu. Idéalement, chaque problem frame devrait être associé à une méthode systématique reconnue pour son efficacité dans l'analyse et la résolution des problèmes de cette classe [2, p.140].

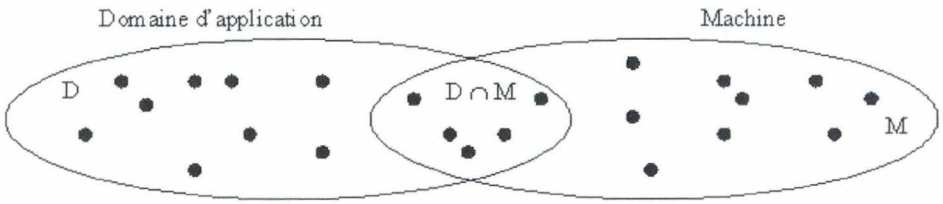
Jackson associe à chaque classe de problèmes un "frame concern" (FC) caractéristique. Ce FC identifie les critères fondamentaux d'une analyse réussie d'un problème qui appartient à la classe de problème concernée. Il spécifie les descriptions qui sont nécessaires et comment elles doivent être mises en commun pour pouvoir donner un argument convaincant que le problème a été complètement compris et analysé.

1.2 Besoins de l'analyste

1.2.1 Compréhension du problème

Le travail de l'analyste

Le but d'un projet informatique est de répondre aux exigences du client. Ces exigences sont formulées en terme des phénomènes du domaine d'application et non pas en terme de phénomènes de la machine. La machine peut garantir la satisfaction des exigences car elle partage des phénomènes avec le domaine d'application [1, p.170].



Tous les phénomènes du domaine d'application ne sont pas connus de la machine. Cela implique qu'il y a un écart entre les exigences du client et ce que la machine peut accomplir directement, car les exigences du client ne sont pas limitées aux phénomènes partagés avec la machine. Le travail de l'analyste consiste à combler ce fossé, autrement dit, montrer que des liens de causalité existent dans le domaine d'application entre les phénomènes que la machine peut contrôler et les phénomènes que le client désire voir apparaître dans le domaine d'application [1, p.170].

Soient,

- R les exigences exprimées en terme du domaine d'application.
- D, les propriétés de ce domaine.
- S, une spécification dérivée de R exprimée en terme de $D \cap M$ (l'interface commune à D et M).
- P, un programme dérivé de la spécification S
- C, les propriétés de l'ordinateur et de la sémantique du langage de programmation.

Pour prouver qu'un programme satisfait les exigences du client, il faut vérifier les deux expressions suivantes :

$C, P \vdash S$

$D, S \vdash R$

Si l'ordinateur a les propriétés C et que s'y ajoutent les propriétés que celui-ci acquiert par le programme P, alors il aura les propriétés S, autrement dit il satisfera la spécification S [1, p.171].

Si le domaine a les propriétés D et que s'y ajoutent les propriétés S, alors il aura les propriétés R, autrement dit, il satisfera les exigences [1, p.171].

Le travail d'analyse du problème doit être le plus formel possible, tout en tenant compte du caractère intrinsèquement informel de l'environnement. Il est aussi important d'être le plus précis possible que de reconnaître qu'une grande partie du monde physique ne peut être décrit fidèlement ou adéquatement en termes formels purs [2, p.xv].

Les descriptions formelles de domaines se font selon une procédure en trois étapes [2, p.162] :

- Formalisation des propriétés du domaine.
- Raisonnements basés sur la formalisation.
- Interprétation des résultats dans le domaine.

Cette procédure est basée sur l'hypothèse forte que le domaine est suffisamment formalisable.

Un domaine est informel si [2, p.163] :

- Toute formalisation et abstraction de phénomènes du domaine et de leurs relations est au mieux une approximation.
- On ne peut pas limiter les considérations qui affectent les propriétés et le comportement du domaine, autrement dit, il est difficile de faire des déclarations universelles qui s'appliquent dans tous les cas.

Il est néanmoins possible de construire des descriptions fiables de domaines informels en utilisant des termes fondamentaux, appelés **désignations**. Une désignation distingue un certain type de phénomène particulier. Elle indique en langage naturel comment il peut être reconnu et lui donne un nom [1, p.58].

Une désignation bien écrite permet d'éviter deux types d'erreurs [2, p.164] :

- Confondre deux phénomènes qui sont intimement liés.
- Appliquer un certain terme à un cas particulier de manière incertaine. Il faut tester la règle de reconnaissance le mieux possible.

Afin d'alléger les descriptions, on peut définir des termes sur base de désignations et d'autres définitions. Ces définitions n'introduisent pas de nouvelles observations sur le domaine ou de nouveaux phénomènes [2, p.165]. Ce sont des termes formels qui ne sont ni vrai ni faux, mais bien formés ou mal formés [2, p.167].

Une description d'un domaine informel peut dès lors être constituée de constructions logiques reposant sur quelques termes fondamentaux informels.

La nature du problème

Il faut définir convenablement les interactions entre l'ordinateur et son environnement de même qu'il faut porter la compréhension du problème au-delà de cette interface : des propriétés et des comportements de parties de l'environnement qui ne sont pas directement visibles de l'interface peuvent ainsi être explorés. Il peut être pertinent de noter les aspects de leur comportement qui ne jouent jamais un rôle direct dans l'interface (un domaine est un sous-ensemble pertinent de l'environnement). Le problème se trouve dans l'environnement et il n'y a pas de raisons de le réduire à ce qui peut être uniquement visible de l'interface entre cet environnement et l'ordinateur. Il se trouve plus en profondeur dans le monde, plus éloigné de l'ordinateur [2, p.7]. Il ne faut cependant pas s'éloigner de trop et risquer de décrire des phénomènes et les relations entre ces phénomènes qui n'ont aucune influence dans le problème étudié.

Nous posons donc les hypothèses suivantes [2, p.16] :

- Il faut faire la description de l'interaction entre la machine et l'environnement et non pas la description de l'état interne de la machine.
- D'autre part, il ne sera procédé à une description du monde que pour autant que cela concerne le problème du client, et non au-delà.

Ce n'est pas parce que l'environnement est continuellement en changement et que le système à construire est continuellement en évolution qu'il faut considérer inutile une analyse approfondie du problème. Dans le cas d'une réponse variée du système en fonction des propriétés changeantes de l'environnement, il peut se révéler nécessaire de se placer à un niveau de généralité plus élevé et introduire la notion de description de domaine paramètrable. On y décrit explicitement les différentes fonctionnalités requises par le système que celui-ci interprétera lors de l'exécution [2, p.xv].

1.2.2 Taxinomie des problèmes

Un problème est principalement caractérisé par la structure et les propriétés du domaine d'application et par les exigences exprimées en terme de ce domaine [1, p.10].

Un problem frame est une sorte de type qui représente et définit une classe communément reconnue de sous-problèmes simples. Ses éléments constitutifs sont l'exigence du problème, des domaines d'un certain type, une machine et des phénomènes typés partagés par les différents domaines. Un PF se distingue donc par ces éléments et par les difficultés et aspects importants qui doivent être adressés dans l'analyse du problème.

Il y a six types de phénomènes, à savoir trois sortes d'individus (choses qui peuvent être nommées et distinguées les unes des autres) et trois sortes

de relations (ensemble d'associations entre les individus) [2, p.79].

Les individus :

- Les événements : ils sont indivisibles et instantanés. Deux événements ne peuvent survenir en même temps.
- Les entités : ce sont des individus qui persistent dans le temps. Ils peuvent changer leur état au cours du temps. Certaines entités produisent des événements. Certaines entités peuvent opérer des changements spontanés sur leur propre état, d'autres sont passives.
- Les valeurs : ce sont des individus intangibles qui existent en dehors du temps et de l'espace. Elles ne changent pas.

Les relations :

- Les états : ce sont des relations entre les entités et les valeurs. Ils peuvent changer au cours du temps.
- Les vérités : ce sont des relations entre des valeurs qui ne peuvent changer dans le temps.
- Les rôles : ce sont des relations entre un événement et des individus qui y participent d'une certaine manière. Ils ne changent pas dans le temps. Chaque rôle exprime dans un événement ce que l'on attribuerait à l'un des arguments de l'événement. Il est utile d'utiliser des rôles car il peut arriver que tous les participants à un événement ne soient pas partagés. Il peut aussi arriver que le contrôle du rôle soit désolidarisé du contrôle de l'événement.

Il y a trois types de domaines [2, p.83] :

- Domaines causaux : domaines dont les propriétés incluent des associations causales prévisibles parmi ses phénomènes causaux. Ces associations causales nous permettent de calculer l'effet du comportement de la machine au niveau de l'interface avec le domaine. Un domaine causal peut contrôler une partie, le tout ou aucun phénomène partagé avec un autre domaine.
- Domaines conciliants[2] : Il s'agit principalement des utilisateurs. Ce sont des domaines physiques mais il n'y a pas de causalité interne prévisible sûre. Autrement dit, dans la plupart des cas, on ne peut contraindre une personne à initier un événement. Le plus que l'on peut faire est de fournir une procédure à suivre.
- Domaines lexicaux : c'est la représentation physique de phénomènes symboliques (de données). C'est aussi un domaine causal dans la mesure où il faut manipuler ces données pour les écrire et les lire. Bien souvent, pour ces domaines, les exigences sont exprimées en terme de ses phénomènes symboliques, mais la machine doit interagir avec le

[2] Jackson utilise le mot "biddable" pour souligner que le domaine fait preuve de bonne volonté pour satisfaire l'exigence. Nous considérons "biddable" et "conciliant" comme équivalent. Nous utilisons le mot français dans le texte et le mot anglais pour identifier le type

domaine en terme de ses phénomènes causaux.

Nous étudierons plus en détail ces typages lorsque nous compléterons le métamodèle (cf. chapitre suivant).

A titre d'exemple, voici quatre PF de base représentant des aspects particuliers de problèmes plus complexes [2, ch.4 et 5] :

- Comportement exigé : une partie de l'environnement doit être contrôlé afin qu'il satisfasse certaines conditions. Le problème est de construire une machine qui imposera ce contrôle.
- Affichage d'information : de l'information sur les états et comportements d'une partie de l'environnement est continuellement demandée. Le problème est de construire une machine qui obtiendra cette information de l'environnement et qui la présentera à l'endroit requis dans la forme requise.
- Manipulation d'objets : le problème est de créer un outil qui permette à l'utilisateur de créer et éditer une certaine classe de textes ou d'objets graphiques manipulable par ordinateur, ou des structures similaires, de telle sorte qu'ils pourront être copiés, imprimés, analysés ou utilisés.
- Transformation : Il s'agit de transformer des fichiers inputs donnés en des fichiers outputs particuliers. Les données en sortie doivent respecter un certain format et doivent être dérivées des données en entrée en respectant certaines règles. Le problème est de construire une machine qui produira les outputs requis à partir des inputs donnés.

Les PF de base sont des abstractions de certains problèmes plus complexes, ne tenant compte que de certains aspects du problème original. Il arrive que dans une analyse, il soit nécessaire de considérer des variantes des PF de base. A titre d'exemple, en voici trois :

- Un modèle est un domaine à part entière correspondant par analogie au monde réel. Il peut être envisagé comme étant l'ensemble des phénomènes privés de la machine qui servent à représenter le monde réel [2, p.178]. Le modèle est un domaine qui n'existe pas dans le problème original, c'est un domaine à concevoir, il fait partie de la solution [2, p.183]. Ce domaine est totalement distinct du monde réel. Le modèle est un domaine lexical, le monde réel est un domaine causal. Ils ne partagent aucun phénomène entre eux. Il est donc indispensable d'établir des relations de correspondance entre les phénomènes des deux domaines. Lorsque l'on utilise un modèle, on sépare le problème en deux sous-problèmes : dans le premier problème, on analyse la construction du modèle à partir du monde réel, dans le second, on utilise ce modèle.
- L'introduction d'un domaine conciliant dans un PF nécessite de reconsidérer les exigences du problème. Par exemple, si l'on introduit un domaine conciliant dans le PF de base "Comportement exigé", nous obtenons un PF "Comportement commandé". Dans ce cas, une partie de l'environnement doit être contrôlé conformément à des commandes

données par un opérateur. Le problème est de construire une machine qui acceptera les commandes de l'opérateur et qui imposera le contrôle en conséquence. La nuance avec le PF "Comportement exigé" est qu'il faut tenir compte du degré d'obéissance de l'opérateur. Celui-ci ne se prêtera pas toujours aux instructions.

- Un domaine de connexion est nécessaire lorsque l'interface entre le domaine principal et la machine n'est pas parfaitement fiable et synchronisé. Autrement, si l'observation d'un phénomène par la machine et dans le domaine n'est pas identique ou s'il y a un délai significatif entre la production d'un phénomène et sa prise en compte, il est nécessaire d'introduire un domaine de connexion qui tiendra compte de ces comportements et propriétés. Le degré de fiabilité ou de synchronisation dépend bien sûr de l'exigence [1, p.31].

1.2.3 Complexité des problèmes

Les problèmes complexes doivent être ramenés à des sous-problèmes plus simples. Ils doivent être décomposés convenablement : les sous-problèmes doivent être plus faciles à résoudre et leurs solutions doivent pouvoir être rassemblées sans heurt par la suite [2, p.57].

La décomposition du problème, de la machine et des domaines se fera en sous-problèmes de classes reconnaissables et familières. La clé d'une telle démarche de décomposition est la connaissance préalable [2, p.59] :

- Des classes de problèmes.
- De ce qui est impliqué lorsqu'on résout ces classes de problèmes.
- Des difficultés et des aspects importants liés à chaque classe de problème.

La connaissance préalable idéale est la connaissance approfondie de problèmes et solutions liés à un même domaine d'application. Si celle-ci fait défaut, une connaissance générale des classes de sous-problèmes qui apparaissent dans la plupart des domaines d'application est nécessaire. Ces classes de sous-problèmes familiers sont définies par les problem frames.

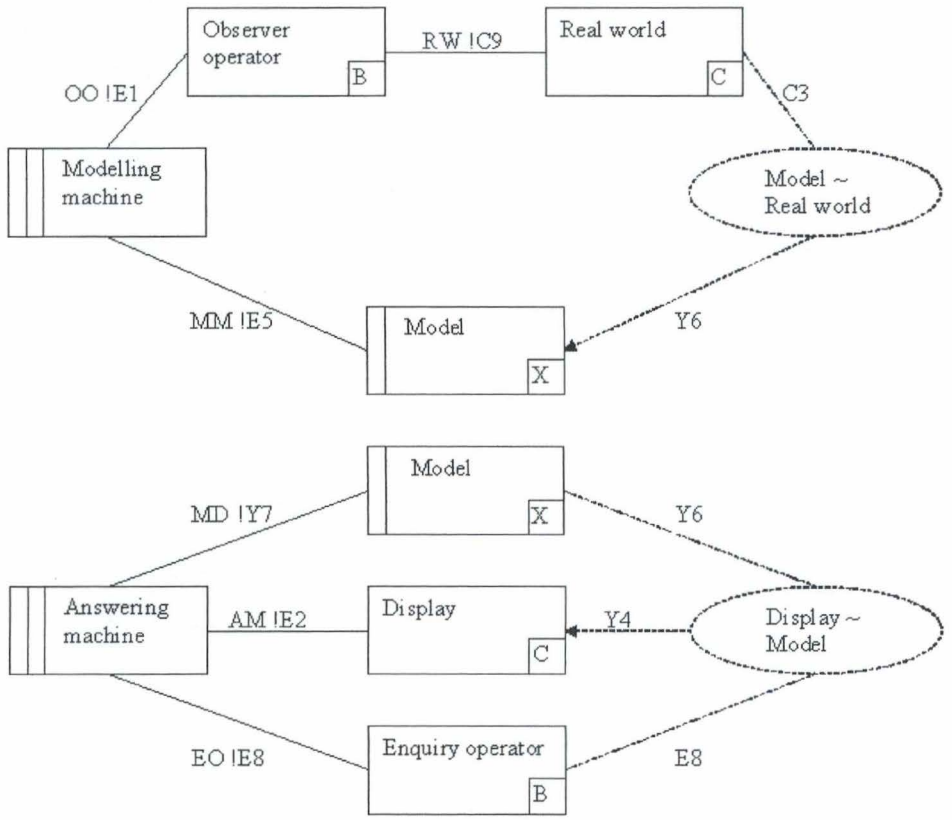
Principes généraux à respecter lors de la décomposition [2, p.60] :

- Les sous-problèmes sont complets. Ils sont des problèmes à part entière. Lors de l'analyse, il faut faire l'hypothèse que les autres sous-problèmes sont résolus. On considère que les descriptions prescriptives de ces autres sous-problèmes sont en fait des descriptions descriptives.
- Les sous-problèmes forment entre eux une structure parallèle plutôt qu'une structure hiérarchique. Chaque sous-problème est une projection du problème. Ces projections ne sont pas nécessairement disjointes.

- Les sous-problèmes peuvent entrer en concurrence les uns avec les autres. Étant donné la structure parallèle, il faut être attentif aux relations entre les sous-problèmes.

1.2.4 Diagrammes de problem frames

Les PF sont représentés par des diagrammes. Nous allons présenter de manière informelle la syntaxe des PF à l'aide d'un exemple. L'exemple considéré prend en compte plusieurs des aspects présentés jusqu'ici. Il s'agit d'un PF de base (Affichage d'information) auquel nous avons ajouté un modèle, un domaine de connexion (Observer operator) et un domaine conciliant (Enquiry operator) [2, p.93, p.183, p.215 et p.226].



Explication succincte des éléments du diagramme :

- Les rectangles représentent des domaines.
- Les lignes représentent les interfaces entre domaines.
- Un rectangle avec deux lignes verticales représente la machine (il n'y en a qu'un par sous-problème).
- Les rectangles avec une ligne verticale représentent les domaines à

concevoir. Le domaine à concevoir est la représentation physique d'une certaine information. La distinction dans la représentation graphique par rapport à un domaine donné est faite pour signaler que l'on est libre de concevoir et spécifier la structure de ses données et dans une moindre mesure le contenu de ses données.

- Les rectangles sans ligne sont les domaines donnés par le problème. Un domaine donné est un domaine dont les propriétés sont fixées ; on n'est pas libre de le concevoir. Dans certains cas on peut changer son comportement par une conception appropriée de la machine. Dans le deuxième sous-problème, le modèle est à considérer comme un domaine donné plutôt que comme un domaine à concevoir. Nous avons cependant gardé cette notation pour montrer que le modèle était le lien entre les deux sous-problèmes.
- Les marques dans les rectangles donnent le type du domaine, C pour causal, B pour conciliant et X pour lexical.
- Les identificateurs d'ensembles de phénomènes partagés sont numérotés de manière unique à l'intérieur d'un même diagramme. Ils sont préfixés en fonction du type des phénomènes, C pour causal, E pour événements et Y pour lexical.
- Les identificateurs d'ensemble de phénomènes sont complétées par un préfixe (deux lettres par convention suivi d'un point d'exclamation) indiquant quel domaine contrôle (ou produit) l'ensemble de phénomènes partagés.
- Les ovales à traits intermittents représentent les exigences.
- Un arc à traits intermittents (éventuellement orienté) reliant l'ovale à un ou plusieurs domaines représente une référence d'exigence. Celle-ci indique que l'exigence se réfère à certains phénomènes du ou des domaines qu'elle touche. L'arc peut être orienté. Dans ce cas, il indique que la référence est contraignante, c'est-à-dire que l'exigence stipule une relation désirée entre les phénomènes concernés.

Ensemble de phénomènes du PF présenté :

- E1 : Commandes d'entrée de données.
- E2 : Information de sortie produites à l'affichage.
- C3 : Phénomènes du monde réel.
- Y4 : Réponses aux questions posées par l'opérateur.
- E5 : Opérations du modèle du monde réel.
- Y6 : Phénomènes du modèle.
- Y7 : Phénomènes du modèle exposés à la machine.
- E8 : Questions posées par l'opérateur.
- C9 : Phénomènes du monde réel vus par l'opérateur observateur.

Le problème est composé de deux sous-problèmes :

- La machine de modélisation doit faire en sorte que les phénomènes du modèle (Y6) correspondent aux phénomènes du monde réel (C3). Elle

doit produire les bonnes opérations du modèle du monde réel. Elle se repose sur les commandes entrées par l'opérateur observateur (E1) qui doit entrer les commandes appropriées basées sur ce qu'il observe du monde réel (C9).

- La machine qui répond doit faire en sorte que les réponses (Y4) correspondent aux phénomènes du modèle (Y6) et aux questions entrées par l'opérateur interrogateur (E8).

Ces deux PF peuvent correspondre par exemple au problème suivant : une course cycliste durant laquelle des opérateurs entrent des données pendant la course et où d'autres opérateurs interrogent le système sur les données compilées.

1.3 Méthode d'analyse

Dans cette section, nous décrivons les 3 grands principes de la méthode de Jackson :

- la structuration itérative du problème,
- les descriptions d'analyse,
- la composition des descriptions.

Nous nous appuyons sur un exemple de problème pour illustrer la méthode : le contrôle d'un distributeur de colis [1,p.138] et [2, p.270].

Un distributeur de colis (package router) est une machine mécanique utilisée par la poste ou par d'autres sociétés de livraison pour distribuer les colis dans des paniers (bin) en fonction de leur destination. Les colis portent des code barres. Ils sont acheminés par un tapis roulant (package conveyor) vers un poste de lecture (reading station) qui lit l'identifiant des colis (id) et leur destination. Ensuite, les colis glissent dans des tuyaux (pipe) équipés de détecteurs de mouvement (sensor) placés aux deux extrémités de ces tuyaux. Les tuyaux sont connectés à des embranchements (switch) à deux sorties qui peuvent être actionnés par l'ordinateur de contrôle (lorsqu'il n'y a pas de colis qui se trouve entre le tuyau d'arrivée et les deux tuyaux de sortie). Cet assemblage forme un arbre binaire. Des paniers de destination correspondants aux destinations des colis terminent les extrémités de l'arbre. Un colis ne peut pas en dépasser un autre, ni dans les tuyaux, ni dans les embranchements. Il y a un coude dans les tuyaux à l'endroit où se trouvent les détecteurs de telle sorte que les colis puissent être identifiés séparément. Toutefois, les colis glissent à des vitesses différentes et imprévisibles et peuvent parfois être trop proche l'un de l'autre pour que les embranchements puissent être actionnés à temps. Un message d'erreur est affiché lorsqu'un colis est dirigé vers un mauvais panier. Un opérateur peut démarrer ou interrompre le tapis roulant par l'intermédiaire de l'ordinateur de contrôle. Le problème est

de construire l'ordinateur de contrôle de telle sorte qu'il obéisse aux ordres de l'opérateur, qu'il dirige les colis vers les paniers de destination correspondante en positionnant les embranchements convenablement et signaler lorsque le colis sont mal acheminés.

1.3.1 Structuration itérative du problème

Dans cette étape, il s'agit de délimiter le problème, d'établir des structures parallèles (projections du problème complexe en sous-problèmes simples) et de faire la correspondance avec des PF connus.

Lorsque l'on entreprend l'analyse et la structuration d'un problème, il est fondamental de déterminer la localisation du problème et les parties de l'environnement qui sont concernées. Ces sous-ensembles pertinents, appelés domaines, sont ceux où le client vérifiera les effets observables et ceux nécessaires à la description.

L'analyse se fait itérativement au fur et à mesure que l'on prend connaissance du problème. L'analyse d'un aspect du problème peut induire la prise en considération d'un nouveau domaine. Cet ajout suscite de nouvelles réflexions concernant les nouveaux aspects du problème qui n'avaient pas encore été envisagés jusque là.

Tous les domaines mentionnés doivent être décrits et seulement ceux-ci : il faut inclure dans l'analyse tous les domaines qui sont susceptibles d'influencer directement ou indirectement les exigences relatives au problème. Cela permet d'étudier ces domaines et de prendre en considération les éléments qui joueront un rôle dans ces exigences [2, p.32].

Dans un premier temps, le diagramme du problème de contrôle du distributeur de colis comprend quatre domaines et une machine : le tapis roulant (package conveyor), l'unité d'affichage (display unit), le distributeur de colis (package router) et l'opérateur (router operator).

Une première décomposition révèle trois sous-problèmes :

- Sous-problème 1 : Obéir aux commandes de l'opérateur.
- Sous-problème 2 : Distribution correcte des colis.
- Sous-problème 3 : Afficher les colis mal acheminés.

L'analyse du premier sous-problème ne révèle pas de nouveaux problèmes. Par contre, celle des deux autres sous-problèmes soulève de nouvelles interrogations :

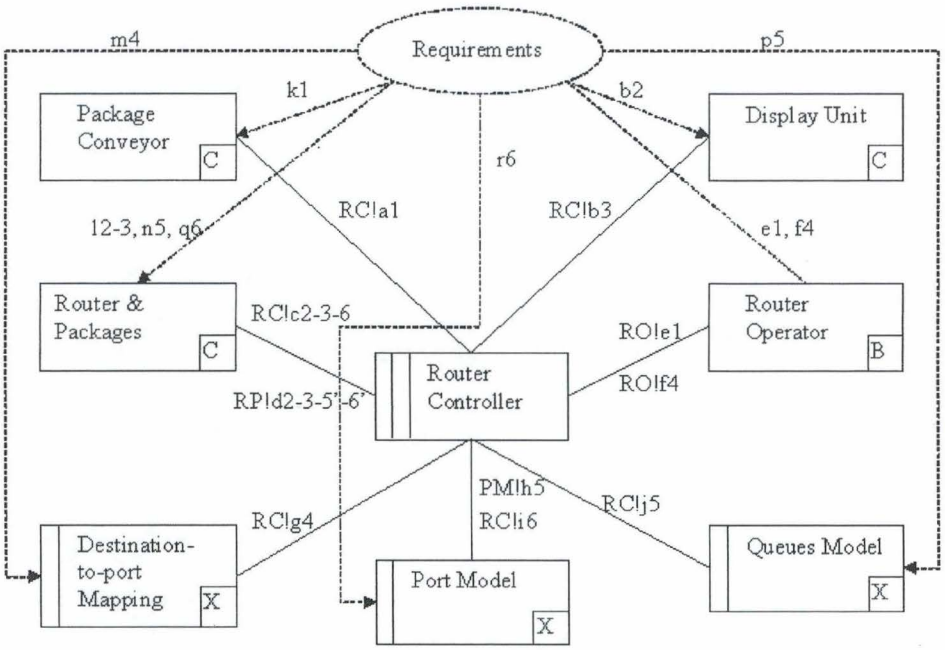
- Quelle est la correspondance entre les paniers et les destinations ?
- Comment la machine peut-elle orienter les colis vers les bons paniers et savoir que ceux-ci sont arrivés ?

Pour répondre à ces interrogations, nous distinguons trois autres sous-problèmes :

- Sous-problème 4 : Éditeur des destinations.
- Sous-problème 5 : Modèle dynamique des queues.
- Sous-problème 6 : Modèle dynamique de la structure du distributeur.

Le premier sous-problème permet de faire les associations entre les paniers et les destinations. Le deuxième sous-problème modélise le mouvement des colis dans le distributeur de colis. Le troisième sous-problème fournit à la machine une représentation de la structure du distributeur de colis (agencement des tuyaux et embranchements).

Le diagramme global du problème de contrôle du distributeur de colis est le suivant [3] :



Ensemble de phénomènes :

- a :
  - Event : OnC : Démarrer le tapis roulant
  - Event : OffC : Arrêter le tapis roulant
- b :
  - Event : ShowPkgId : Afficher identifiant du colis
  - Event : ShowBin : Afficher panier

[3] Nous représentons les ensembles de phénomènes de manière différente de celle de Jackson tout en respectant le métamodèle. Les interfaces peuvent représenter plusieurs ensembles de phénomènes. Ces ensembles sont identifiés par des lettres et des chiffres. Ces derniers représentent les sous-problèmes où ils apparaissent. Par exemple, l'identifiant f2-3 signale que le même ensemble de phénomènes f apparaît dans les sous-problèmes 2 et 3 pour la même interface

- Event : ShowDest : Afficher destination

- c :
  - Event[] : Lsw(i) : Positionner l'embranchement i à gauche
  - Event[] : Rsw(i) : Positionner l'embranchement i à droite
- d :
  - Event : SendLabel(p,l) : Envoyer label l du colis p
  - CausalState : LId(l,i) : Identifiant i du label l (valeur spécifiée dans le label)
  - CausalState : LDest(l,d) : Destination d du colis l (valeur spécifiée dans le label)
  - CausalState[] : SwPos(I, pos) : Position de l'embranchement i : LEFT/RIGHT
  - CausalState[] : SensOn(I, active) : Détecteur i activé ssi colis devant : TRUE, sinon FALSE
- e :
  - Event : OnBut : Appuyer sur bouton On
  - Event : OffBut : Appuyer sur bouton Off
- f :
  - Event : Edit commands : Commandes d'édition
- g :
  - Event : Mapping operations : Opérations d'associations
- h :
  - SymbolicState : Layout Model States : Les états du modèle de disposition du distributeur
- i :
  - Port Model Operations : Opérations de modélisation des portes
- j :
  - Event : EnQ(r, q, s) : Mettre dans la queue q associée au détecteur s l'enregistrement r
  - Event : DeQ(r, q, s) : Enlever de la queue q associée au détecteur s l'enregistrement r
  - SymbolicState : RecPkg(r, p, d) : Enregistrement r contenant l'identifiant du colis p et du panier d
- k :
  - CausalState : Running : Tapis roulant en marche
  - CausalState : Stopped : Tapis roulant à l'arrêt
- l :
  - Event : PkgArr(p, b) : Le colis p arrive au panier b
  - SymbolicState : Assoc(d, b) : Association de la destination d avec le panier b
  - SymbolicState : PDest(p, d) : Destination d du colis p
- m :
  - SymbolicState : isD(d) : Est la destination d
  - SymbolicState : isB(b) : Est le panier b
  - SymbolicState : assoc(d, b) : Association du panier b avec la desti-

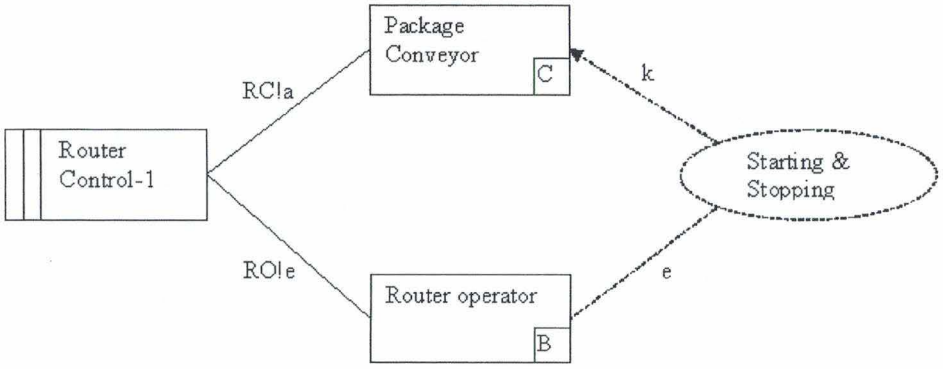
nation d

- n :
  - Event : PkgArr(p, s) : Le colis p est détecté par le détecteur s
  - CausalState : PId(p, i) : Identifiant I du colis p
  - CausalState : PDest(p, d) : Destination d du colis p
- p :
  - SymbolicState : LastArr(s, q, p, d) : Un colis p avec une destination d est arrivé au détecteur s associé avec la queue q
  - Empty(s, q) : La queue q correspondant au détecteur s est vide
- q :
  - RouterLayout : Disposition du distributeur
- r :
  - Port Layout : Disposition des portes

Correspondance avec des PF

Sous-problème 1 : Obéir aux commandes de l'opérateur (PF Commanded behaviour)

La machine démarre et arrête le tapis roulant sur base des commandes lancées par l'opérateur.



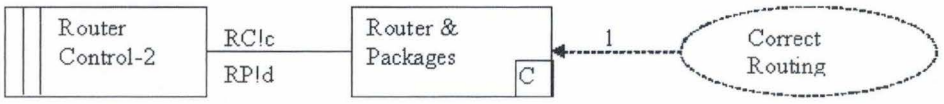
Ensemble de phénomènes :

- a :
  - Event : OnC : Démarrer le tapis roulant
  - Event : OffC : Arrêter le tapis roulant
- e :
  - Event : OnBut : Appuyer sur bouton On
  - Event : OffBut : Appuyer sur bouton Off
- k :
  - CausalState : Running : Tapis roulant en marche

- CausalState : Stopped : Tapis roulant à l'arrêt

Sous-problème 2 : Distribution correcte des colis (PF Required behaviour)

La machine positionne les différents embranchements pour que le parcours du colis dans le distributeur aboutisse au bon panier. Elle positionne ces embranchements lorsque le colis se présente devant le détecteur associé.

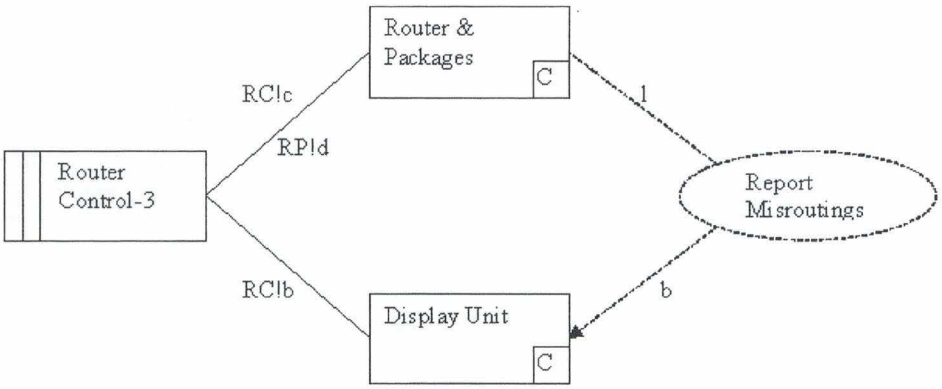


Ensemble de phénomènes :

- c :
  - Event[] : Lsw(i) : Positionner l'embranchement i à gauche
  - Event[] : Rsw(i) : Positionner l'embranchement i à droite
- d :
  - Event : SendLabel(p,l) : Envoyer label l du colis p
  - CausalState : LId(l,i) : Identifiant i du label l (valeur spécifiée dans le label)
  - CausalState : LDest(l,d) : Destination d du colis l (valeur spécifiée dans le label)
  - CausalState[] : SwPos(I, pos) : Position de l'embranchement i : LEFT/RIGHT
  - CausalState[] : SensOn(I, active) : Détecteur i activé ssi colis devant : TRUE, sinon FALSE
- l :
  - Event : PkgArr(p, b) : Le colis p arrive au panier b
  - SymbolicState : Assoc(d, b) : Association de la destination d avec le panier b
  - SymbolicState : PDest(p, d) : Destination d du colis p

Sous-problème 3 : Afficher les colis mal acheminés (PF Information display)

La machine détecte les colis mal acheminés et rapporte les erreurs dans l'unité d'affichage.

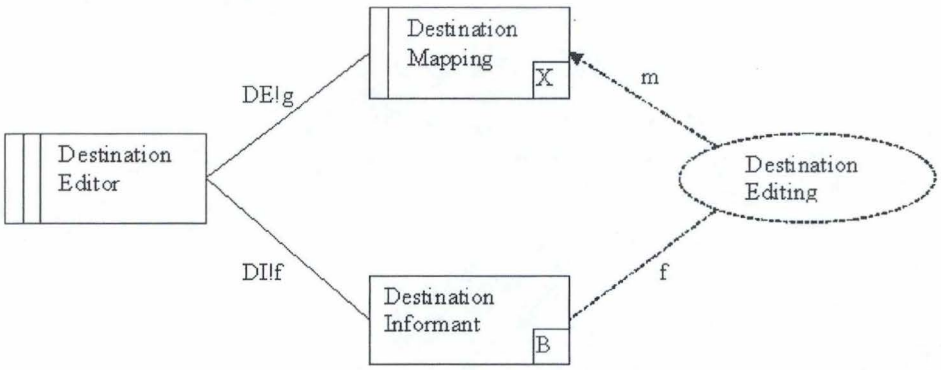


Ensemble de phénomènes :

- b :
  - Event : ShowPkgId : Afficher identifiant du colis
  - Event : ShowBin : Afficher panier
  - Event : ShowDest : Afficher destination
- c :
  - Event[] : Lsw(i) : Positionner l'embranchement i à gauche
  - Event[] : Rsw(i) : Positionner l'embranchement i à droite
- d :
  - Event : SendLabel(p,l) : Envoyer label l du colis p
  - CausalState : LId(l,i) : Identifiant i du label l (valeur spécifiée dans le label)
  - CausalState : LDest(l,d) : Destination d du colis l (valeur spécifiée dans le label)
  - CausalState[] : SwPos(I, pos) : Position de l'embranchement i : LEFT/RIGHT
  - CausalState[] : SensOn(I, active) : Détecteur i activé ssi colis devant : TRUE, sinon FALSE
- l :
  - Event : PkgArr(p, b) : Le colis p arrive au panier b
  - SymbolicState : Assoc(d, b) : Association de la destination d avec le panier b
  - SymbolicState : PDest(p, d) : Destination d du colis p

Sous-problème 4 : Editeur des destinations (PF Simple Workpieces)

Un opérateur édite les associations entre destination et panier : ajout et suppression des associations.



Ensemble de phénomènes :

- f :
  - Event : Edit commands : Commandes d'édition
- g :
  - Event : Mapping operations : Opérations d'associations
- m :
  - SymbolicState : isD(d) : Est la destination d
  - SymbolicState : isB(b) : Est le panier b
  - SymbolicState : assoc(d, b) : Association du panier b avec la destination d

Les commandes d'édition peuvent être les suivantes :

add D d+ ; Ajout d'une ou plusieurs destinations d

add B b+ ; Ajout d'un ou plusieurs paniers b

add A b d+ ; Ajout d'une association entre un panier b et une ou plusieurs destinations d

del D d+ ; Suppression d'une ou plusieurs destinations d

del B b+ ; Suppression d'un ou plusieurs paniers b

del A b d+ Suppression d'une association entre un panier b et une ou plusieurs destinations d

Les opérations d'associations équivalentes lancées par la machine peuvent être les suivantes :

addD(d)

addB(b)

addA(d, b)

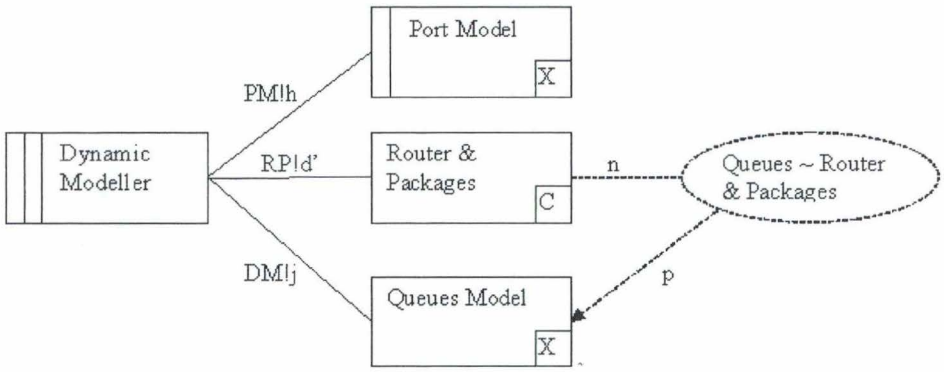
delD(d)

delB(b)

delA(d,b)

Sous-problème 5 : Modèle dynamique des queues (PF Model domain)

La machine modélise le mouvement des colis dans les tuyaux par des queues FIFO en se basant sur un modèle de la structure du distributeur.



Ensemble de phénomènes :

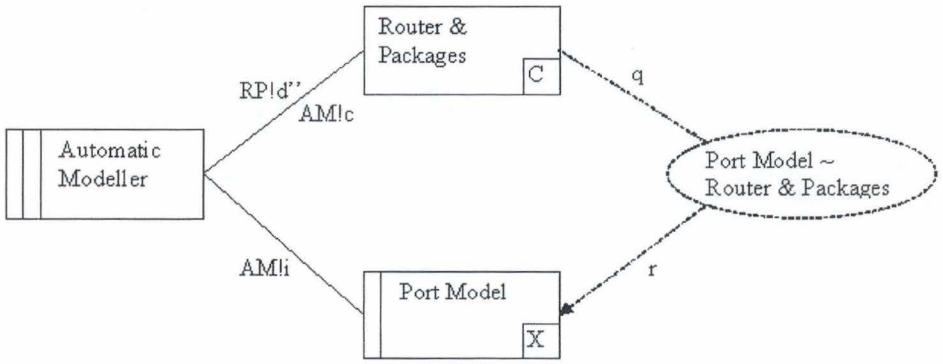
- d' : [4]
  - Event : SendLabel(p,l) : Envoyer label l du colis p
  - CausalState : LId(l,i) : Identifiant i du label l (valeur spécifiée dans le label)
  - CausalState : LDest(l,d) : Destination d du colis l (valeur spécifiée dans le label)
  - CausalState[] : SensOn(I, active) : Détecteur i activé ssi colis devant : TRUE, sinon FALSE
- h :
  - SymbolicState : Layout Model States : Les états du modèle de disposition du distributeur
- j :
  - Event : EnQ(r, q, s) : Mettre dans la queue q associée au détecteur s l'enregistrement r
  - Event : DeQ(r, q, s) : Enlever de la queue q associée au détecteur s l'enregistrement r
  - SymbolicState : RecPkg(r, p, d) : Enregistrement r contenant l'identifiant du colis p et du panier d
- n :
  - Event : PkgArr(p, s) : Le colis p est détecté par le détecteur s
  - CausalState : PId(p, i) : Identifiant I du colis p
  - CausalState : PDest(p, d) : Destination d du colis p

[4] il s'agit d'un sous-ensemble de l'ensemble d du diagramme global ; celui-ci ne contient pas de CausalState SwPos(l, pos).

- p :
  - SymbolicState : LastArr(s, q, p, d) : Un colis p avec une destination d est arrivé au détecteur s associé avec la queue q
  - Empty(s, q) : La queue q correspondant au détecteur s est vide

Sous-problème 6 : Modèle dynamique de la structure du distributeur (PF Model domain)

Construction dynamique d'un modèle de la structure du distributeur.



Ensemble de phénomènes :

- c :
  - Event[] : Lsw(i) : Positionner l'embranchement i à gauche
  - Event[] : Rsw(i) : Positionner l'embranchement i à droite
- d" : [5]
  - Event : SendLabel(p,l) : Envoyer label l du colis p
  - CausalState[] : SwPos(I, pos) : Position de l'embranchement i : LEFT/RIGHT
  - CausalState[] : SensOn(I, active) : Détecteur i activé ssi colis devant : TRUE, sinon FALSE
- i :
  - Port Model Operations : Opérations de modélisation des portes
- q :
  - RouterLayout : Disposition du distributeur
- r :
  - Port Layout : Disposition des portes

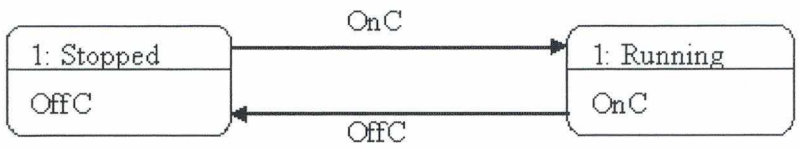
1.3.2 Descriptions d'analyse

Chaque sous-problème doit faire l'objet d'une analyse de chacun de ses domaines.

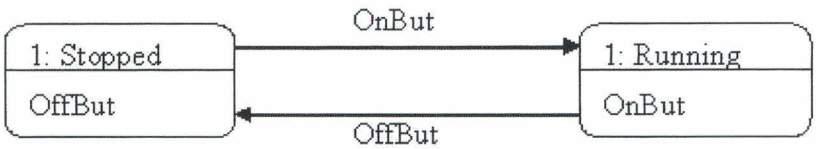
[5] il s'agit d'un sous-ensemble de l'ensemble d du diagramme global ; celui-ci ne contient pas les CausalState LId(l,i) et LDest(l,d).

Par exemple, le sous-problème 1 (obéir aux commandes de l'opérateur) comporte le tapis roulant, la machine et l'exigence. Le tapis roulant démarre lorsque l'on appuie sur le bouton On. Il s'arrête lorsqu'on appuie sur le bouton Off. Lorsqu'il tourne et que l'on appuie sur le bouton On, il ne se passe rien (le tapis continue de tourner). Lorsqu'il est à l'arrêt et que l'on appuie sur le bouton Off, il ne se passe rien (le tapis reste à l'arrêt). Il faut combler le fossé entre ce sur quoi la machine peut agir et ce que l'on désire voir satisfait dans le domaine. Il faut donc trouver le comportement de la machine qui permettra d'atteindre ce but. Ci-dessous, les trois descriptions à l'aide de la notation des statecharts :

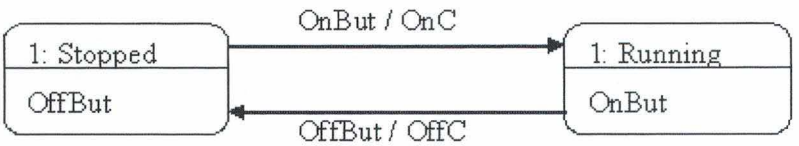
Description du tapis roulant



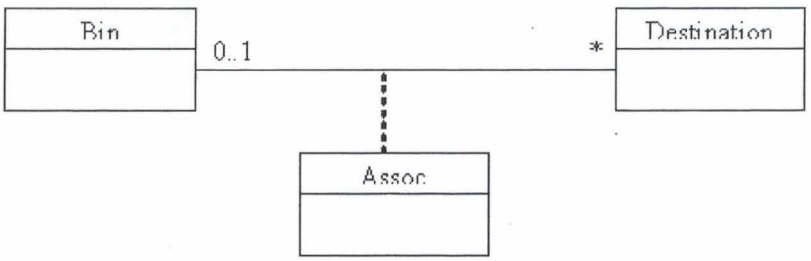
Description de l'exigence



Description de la machine



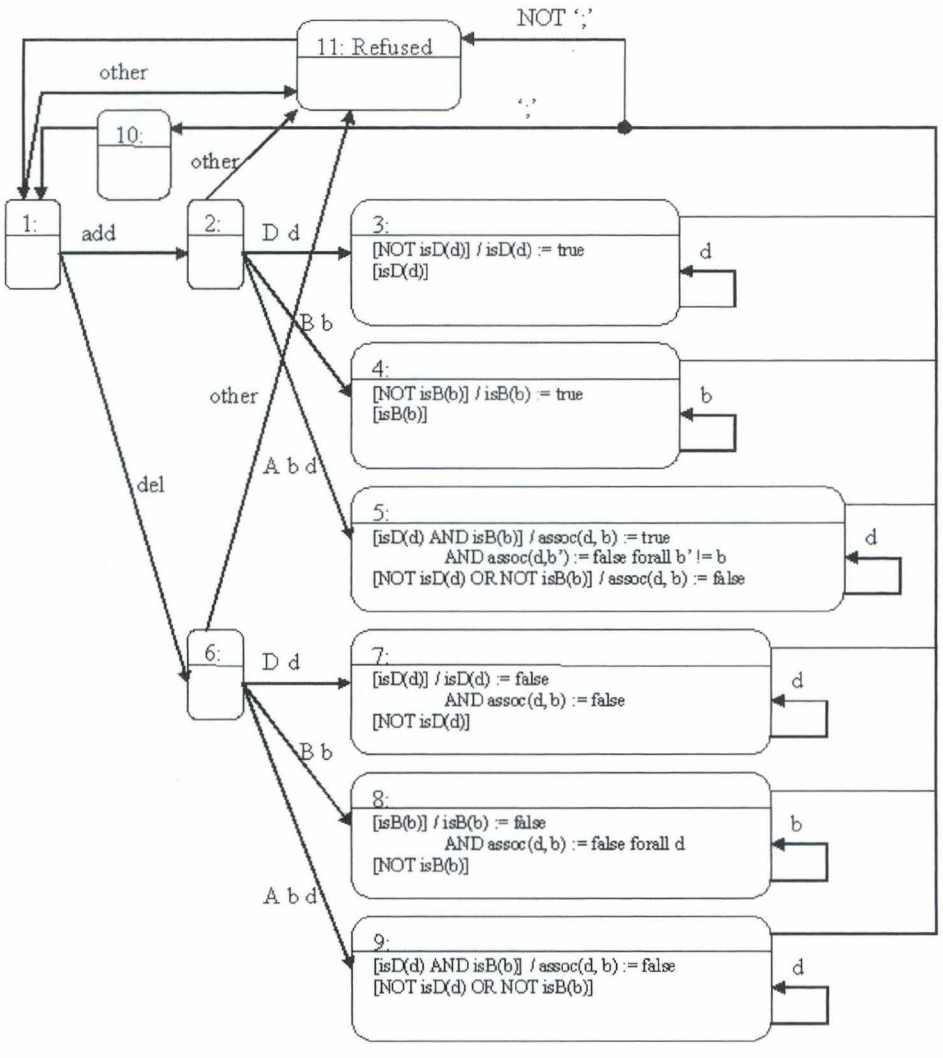
Autre exemple, le sous-problème 4 (éditeur des destinations). L'étude du domaine nous informe que plusieurs destinations peuvent être associées à un panier :



Les domaines peuvent également être décrits à l'aide de statecharts, en tenant compte des commandes d'édition et des opérations d'associations présentées plus haut :

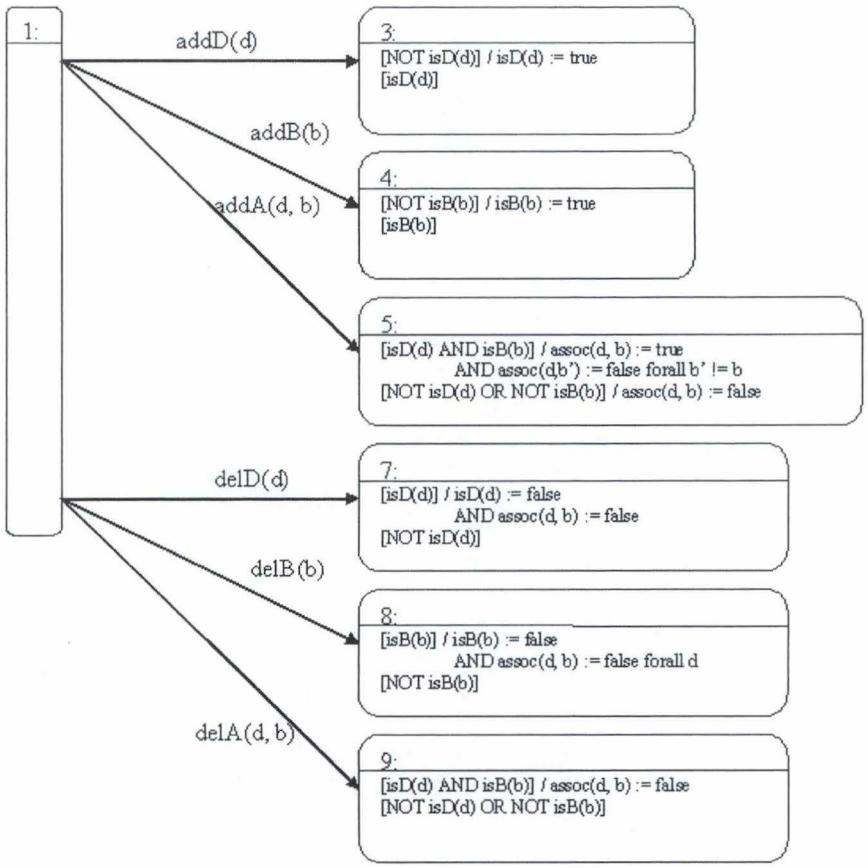
Description de l'exigence

L'exigence est décrite en terme des phénomènes de l'ensemble m partagés avec le domaine lexical Destination Mapping et des phénomènes de l'ensemble f partagés avec l'opérateur (les commandes d'édition). Par exemple, les transitions de l'état 1 à l'état 2 et 3 décrivent que si l'on a les événements add D d, alors d est une destination. Les transitions de l'état 1 à l'état 2 et 5 expriment que si l'on a les événements add A b d, alors si d est une destination et b un panier alors le panier b est associé à la destination d. De plus, aucun autre panier n'est plus associé à cette destination.



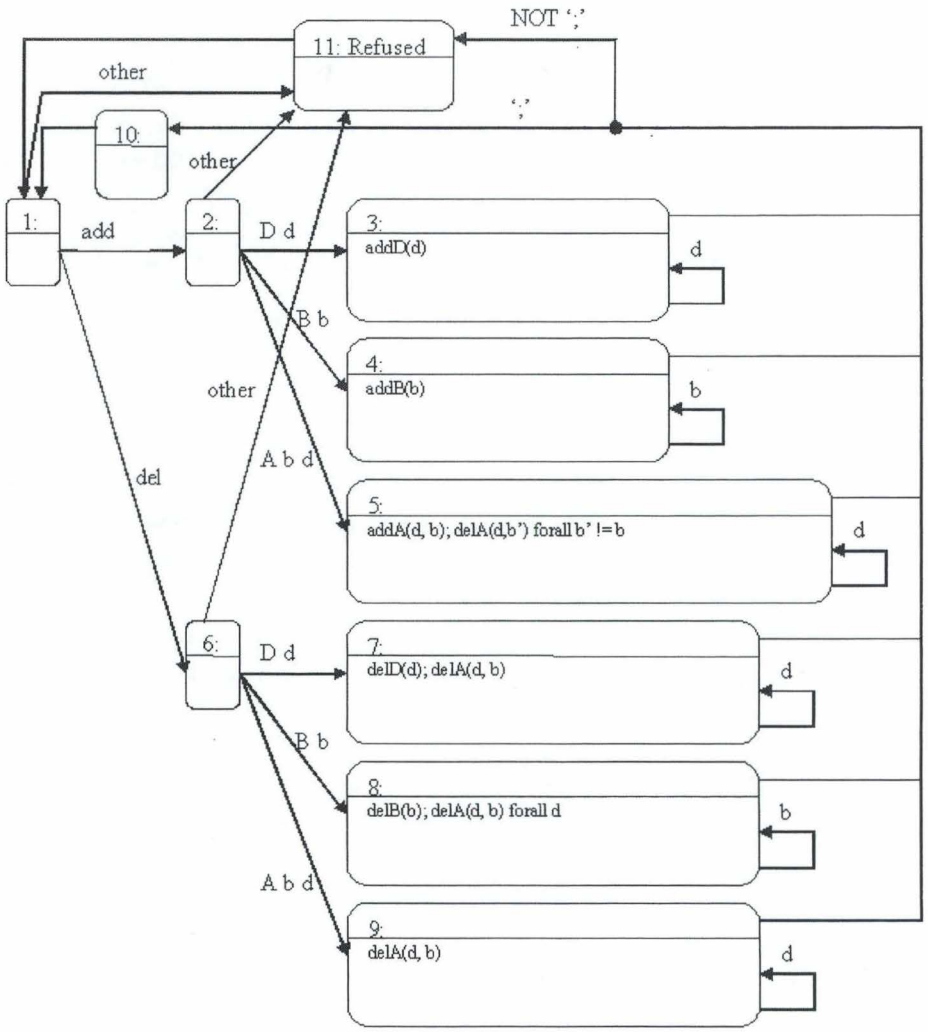
Description du domaine des correspondances

C'est dans ce domaine qu'il faut combler le fossé entre les phénomènes voulut et ceux contrôlables par la machine. Le Destination Mapping est décrit en terme des phénomènes de l'ensemble m partagés avec l'exigence et des phénomènes de l'ensemble g partagés avec la machine (les opérations d'associations). Par exemple, la transition de l'état 1 à l'état 3 décrit que si l'on a l'opération addD(d), alors d est une destination. La transition de l'état 1 à l'état 5 exprime que si l'on a l'opération addA(b,d), alors si d est une destination et b un panier alors le panier b est associé à la destination d. De plus, aucun autre panier n'est plus associé à cette destination.



Description de la machine

sLa machine est décrite en terme des phénomènes de l'ensemble g partagés avec le domaine lexical Destination Mapping (les opérations d'associations) et des phénomènes de l'ensemble f partagés avec l'opérateur (les commandes d'édition). Par exemple, les transitions de l'état 1 à l'état 2 et 3 décrivent que si l'on a les événements add D d, alors la machine produit l'opération addD(d). Les transitions de l'état 1 à l'état 2 et 5 expriment que si l'on a les événements add A b d, alors la machine produit les opérations addA(b,d) et les delA(d,b') pour tout b' != b.



1.3.3 Composition des descriptions

"*Separate where you can, and compose where you must.*"[1, p.30].

Des problèmes peuvent survenir lorsque l'on tente de composer les différentes descriptions des sous-problèmes. Ces problèmes arrivent lorsque deux sous-problèmes comportent des projections différentes d'un domaine commun [2, p304]. Les deux projections sont-elles cohérentes entre elles ? Les exigences peuvent-elles être satisfaites en même temps ? Le domaine possède-t-il toutes les propriétés décrites dans les deux projections ? Un domaine partageant des phénomènes avec les machines des deux sous-problèmes peut-il

répondre simultanément aux deux machines ? [6]

La composition de sous-problèmes se fait par paires de sous-problèmes. Si l'on a par exemple trois sous-problèmes qui présentent des problèmes de composition, l'on compose deux de ces sous-problèmes et ensuite le sous-problème résultant est composé avec le troisième sous-problème.

Par exemple, dans le problème du distributeur de colis, le domaine "Port model" est décrit dans les sous-problèmes 5 et 6. Il est généré dans le sous-problème 6 et utilisé dans le sous-problème 5. Il y a un problème de composition car il ne faut pas que l'on puisse générer le domaine alors qu'il est en train d'être utilisé.

Deux sous-problèmes qui n'ont pas de domaines et pas de phénomènes en commun n'ont pas de problèmes de composition, autrement dit, ils peuvent être composés sans problème.

Deux sous-problèmes qui partagent un domaine n'interagissent pas entre eux lorsque le domaine partagé est autonome [7]. Si le domaine n'est pas autonome, il faut tenir compte de plusieurs aspects :

- Les descriptions doivent être comparables : lorsque le même phénomène d'un domaine est décrit suivant un niveau d'abstraction différent d'une description à l'autre, il faut relier les deux abstractions de manière unique et consistante à travers toutes les descriptions.
- Les descriptions doivent être cohérentes : lorsque l'incohérence n'est pas décelable directement, il peut être utile de combiner les descriptions et d'examiner le résultat attentivement pour vérifier s'il donne satisfaction aux exigences des deux sous-problèmes[8]. Si cela n'est pas le cas, il faut donner la priorité à l'une d'entre elles.
- Les aspects de priorité d'une description sur l'autre : lorsque des problèmes d'incohérence sont insolubles, les différents comportements sont ordonnés par ordre de priorité pour chaque état du domaine. Ensuite, c'est le comportement de plus haute priorité qui est adopté. Il faut alors vérifier les aspects d'interférence.
- Des aspects d'interférence et de synchronisation d'une description avec l'autre : lorsque deux sous-problèmes ont un domaine en commun, il

[6] La composition des différentes machines n'est pas un problème de composition de descriptions d'analyse mais de composition de solutions [2, p307].

[7] Un domaine autonome est un domaine actif : il cause spontanément des événements et des changements d'états. Ils contrôlent tous les phénomènes à l'interface. Rien ne peut affecter son comportement. cf. chapitre Métamodèle, Domaines actifs

[8] Jackson décrit la procédure pour combiner deux statecharts [2, p316], l'un contenant m états et l'autre n états, le statechart résultant comporte m x n états. Si les deux statecharts ne comportent pas beaucoup d'états cela est envisageable, comme dans l'exemple qu'il donne, par contre si le nombre d'états est élevé, comme le signale Harel dans [HarelPoliti, p62] le nombre total d'états du statechart résultant peut être très vite très grand et ingérable.

peut arriver que l'exécution d'un sous-problème interfère avec l'exécution de l'autre sous-problème, lorsqu'ils interagissent avec le domaine au même moment. Cela arrive souvent lorsque dans un sous-problème on considère un domaine comme dynamique (domaine contrôlé, ou domaine à construire) et dans l'autre sous-problème on le considère comme autonome ou même statique. Cet aspect d'interférence influence la découpe du problème en sous-problèmes et est une illustration de la nature itérative de la structuration du problème. Par ailleurs, même si deux sous-problèmes partagent un domaine et qu'ils n'ont pas de problèmes d'interférence, il peut arriver que la composition des deux sous-problèmes empêche l'exigence du problème globale d'être satisfaite, Jackson parle alors de problème de synchronisation entre les descriptions [2,p.327].

Chapitre 2

Métamodèle

2.1 Présentation du métamodèle

Notre travail est basé sur le métamodèle développé par G. Delannay [MM] dont vous pouvez trouver une copie dans l'annexe A. Dans ce document, les concepts définis par Jackson dans [1] et [2] ont été repris non sans y avoir apporter quelques modifications. Certains concepts ont été laissés en suspens. Nous nous proposons de compléter les aspects en suspens afin de disposer d'une base adéquate pour l'élaboration de l'outil.

La principale différence entre la vue de Jackson et celle du métamodèle réside dans la distinction entre les diagrammes de contexte, les diagrammes de problème et les diagramme de PF : dans le métamodèle, il n'y a qu'un seul type de diagramme, le diagramme de problème [MM, p4] :

- Le diagramme de contexte est un diagramme de problème dans lequel des éléments ont été cachés, principalement l'exigence.
- Le diagramme de PF, lorsqu'il ne représente pas un problème, est un diagramme de problème dont les ensembles de phénomènes sont vides [MM, p13].

Les concepts que nous allons préciser dans ce chapitre sont les suivants :

- type d'un phénomène ;
- type d'un ensemble de phénomènes ;
- type et dimensions d'un domaine ;
- correspondance entre problème et PF.

Les PF sont une sorte d'agent fédérateur des descriptions faites dans des notations plus spécialisées des divers domaines impliqués dans un problème. Afin de permettre l'intégration de ces notations dans les PF, il faut passer par ce typage.

Pour exprimer des contraintes précises sur les diagrammes de classe, nous

reprenons le langage "OCL-like" utilisé dans [MM].

2.2 Typologie des phénomènes

Une étape importante lors de l'intégration d'une notation est de rapprocher les concepts trouvés dans la notation des concepts des PF. Nous caractérisons les phénomènes du métamodèle afin de faciliter la spécialisation des différents types de phénomènes avec les concepts d'une notation spécifique [MM, p.9].

Nous complétons ensuite le métamodèle en définissant les relations entre phénomènes.

2.2.1 Caractérisation des phénomènes du métamodèle

Nous pouvons relever dans le texte de Jackson certains aspects et dimensions des phénomènes. Ces éléments nous permettrons d'établir de caractériser les phénomènes.

Dimension temporelle :

Les phénomènes ont-ils une persistance dans le temps ?

Valeurs clés :

- Instantané : le phénomène se déroule à un instant donné.
- Persistant : il existe pendant une certaine période de temps.
- Intemporel : il est hors du temps.

Unicité dans le temps : seulement un seul événement peut se dérouler à un moment donné. Mais, deux entités peuvent coexister.

Dimension spatiale :

Les phénomènes sont-ils tangibles ou intangibles ?

Dimension ensembliste :

Les liens entre phénomènes du même type sont-ils pertinents (notion ensembliste d'ordre, d'unicité, ...) ? cfr. Liens entre phénomènes.

Identité :

Les phénomènes peuvent être distingués les uns des autres.

Atomicité / relation de composition :

Une fois que l'on a choisi de regarder le monde d'une certaine manière et que l'on a choisi les phénomènes, ceux-ci sont-ils indivisibles ?

Altérabilité :

Le phénomène change au cours du temps.

Causal ou symbolique :

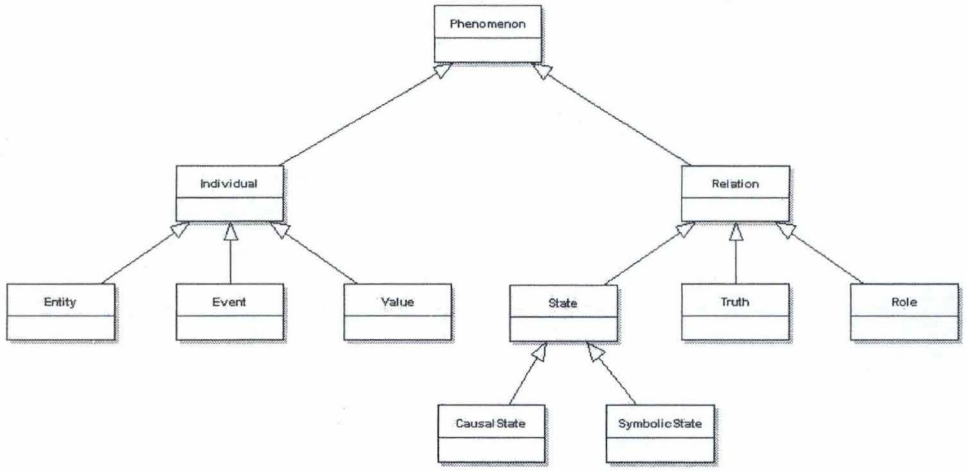
- Les phénomènes causals sont initiateurs d'autres phénomènes : une entité peut initier un événement.
- Les phénomènes symboliques sont utilisés pour symboliser d'autres phénomènes et relations entre eux.

Tableau des phénomènes :

Si on applique les concepts précédents à la classification du métamodèle, nous obtenons le tableau suivant :

	Event	Entity	Value	State	Truth	Role
Dimension temporelle	instantané	persistent	intemporel	persistent	intemporel	intemporel
Dimension spatiale	tangible	tangible	intangible	tangible	intangible	tangible
Dimension ensembliste	ensemble ordonné dans le temps	une classe d'entité est un amas non ordonné	une classe de valeurs est un ensemble ordonné	non significatif	non significatif	non significatif
Identité	oui	oui	oui	non	non	non
Composition	non	non	non	?	?	?
Altérabilité	non	oui	non	oui / non	non	non
Causal ou symbolique	causal	causal	symbolique	causal/symb.	symbolique	causal

Les Relations State entre Entity et Value sont causal et les Relations State entre Value sont symboliques. Nous prenons donc l'option de sous-typer la classe des State en CausalState et SymbolicState.



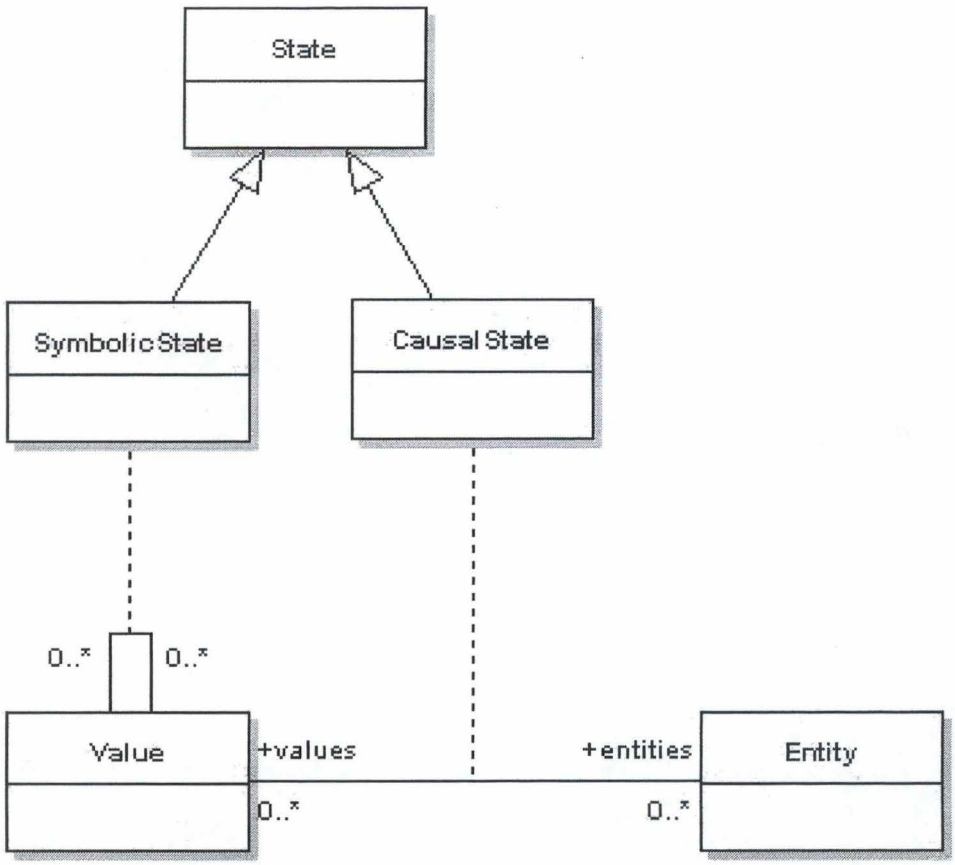
2.2.2 Liens entre les phénomènes

Dans [2,p80], Jackson définit une taxinomie des phénomènes et distingue deux grandes catégories de phénomènes : les Individuals et les Relations. Il décrit également les liens qui peuvent exister entre ces deux catégories. Il définit les phénomènes de type Relation comme une association entre Individuals et décrit quels types d'Inviduals participent aux trois types de Relation. D'une manière générale, on peut représenter la relation entre les phénomènes de type Individual et Relation comme suit :

Chaque Relation implique un ou plusieurs Individual et chaque type Relation impose des contraintes concernant le ou les type(s) Individual pouvant participer à l'association.

Relation State

Un état est une relation entre Individual de type Entity ou Value qui peut changer dans le temps. Pour le typage des ensembles de phénomènes, il est nécessaire de déterminer si le State est un SymbolicState ou un CausalState. Cela se fait à travers la relation entre Individual. Un CausalState est défini en terme de relation entre entity et value. Un SymbolicState est défini uniquement en terme de relation entre Value.



Jackson [2,81] semble accepter le fait qu'un état ne soit pas lié à un Individual (ex : Locked() ou Locked). Nous le considérons alors implicite dans la notation. Le diagramme est donc cohérent. Il est toutefois possible que l'information ne soit pas disponible notamment lors de l'intégration d'autres notations.

Contraintes sur les CausalState :

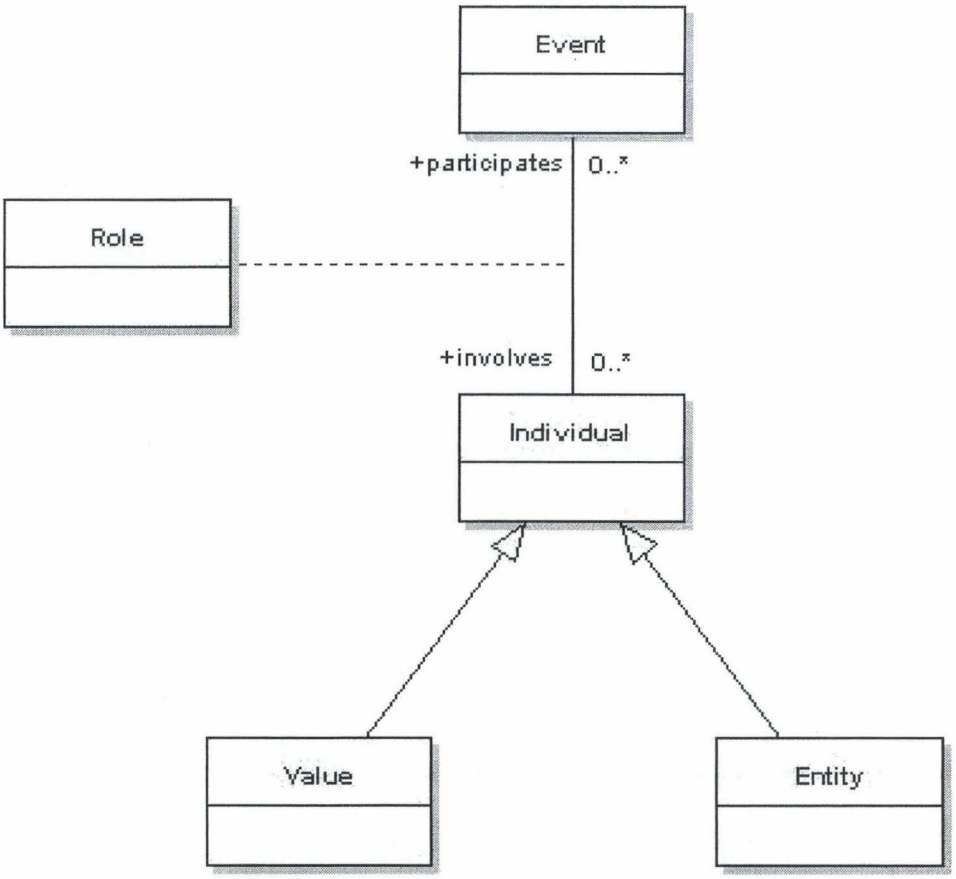
Si le State est de type CausalState cela implique qu'il existe au moins un Individual de type Entity impliqué dans la relation.

Contraintes sur les SymbolicState :

Si le State est de type SymbolicState cela implique que tous les Individuals impliqués dans le relation sont du type Value.

Relation Role

Le rôle est la relation entre un événement et d'autres Individual qui participent à cet événement. Role devient une classe d'association entre Event et Entity/Value.



Un rôle ne fait pas intervenir des événements qui participent à d'autres événements. En effet, il ne peut pas y avoir de relations entre des événements puisque les événements ne surviennent pas en même temps (unicité dans le temps).

Un événement est souvent accompagné de paramètres. Le lien entre l'événement et les différents paramètres peut être exprimé par des rôles.

Relation Truth

Une vérité est une relation entre Value qui ne change pas dans le temps.

2.3 Typage des ensembles de phénomènes

Dans les PF présenté dans [2], Jackson introduit la notion de type de phénomènes liés à une interface ou référencés par l'exigence. Il définit trois valeurs utilisées dans les noms des ensembles de phénomènes :

- C pour causal
- Y pour symbolique
- E pour event.

Pour pouvoir vérifier si un problème particulier correspond à un PF, l'outil devra être capable de déduire le type des ensembles de phénomènes du problème et de les comparer avec les types des éléments correspondants du PF. L'étape suivante sera d'essayer de déduire de ces types le type du domaine. Ces opérations de typage sont essentielles pour effectuer la correspondance entre le PF et le problème.

La déduction du type se fait à partir des phénomènes.

Valeurs possibles :

- Causal : tous les phénomènes sont de type causal.
- Symbolique : tous les phénomènes sont de type symbolique.
- Event : tous le phénomènes sont des événements
- Undef : l'ensemble de phénomène n'est pas typé.

Le type Event est évident. Ce typage des ensembles de phénomènes nécessite de typer les phénomènes du point de vue de la causalité.

Jackson définit en [2,83] les deux catégories de phénomènes :

- Phénomène de type causal : Evénements ou rôles ou états reliant des entities. Causal parce qu'ils sont directement causés ou contrôlés par certains domaines et parce qu'ils peuvent causer d'autres phénomènes à leur tour.
- Phénomène de type symbolique : Valeurs ou vérités ou états reliant seulement des valeurs. Symbolique parce qu'ils sont utilisés pour symboliser d'autres phénomènes et relations entre eux.

Définition

Ces catégories nous permettent de définir les différents types d'ensembles de phénomènes comme suit :

Type Causal : ensemble de phénomènes constitué d'Event, de Role et/ou de Causal State.

PhenomenaSet.type = Causal implies PhenomenaSet.phens->forall(ph : Phenomenon | ph.ocllsType(Event) or ph.ocllsType(Role) or ph.ocllsType(CausalState))

Type Event : ensemble de phénomènes constitué d'Event ; Event est un sous-type de Causal.

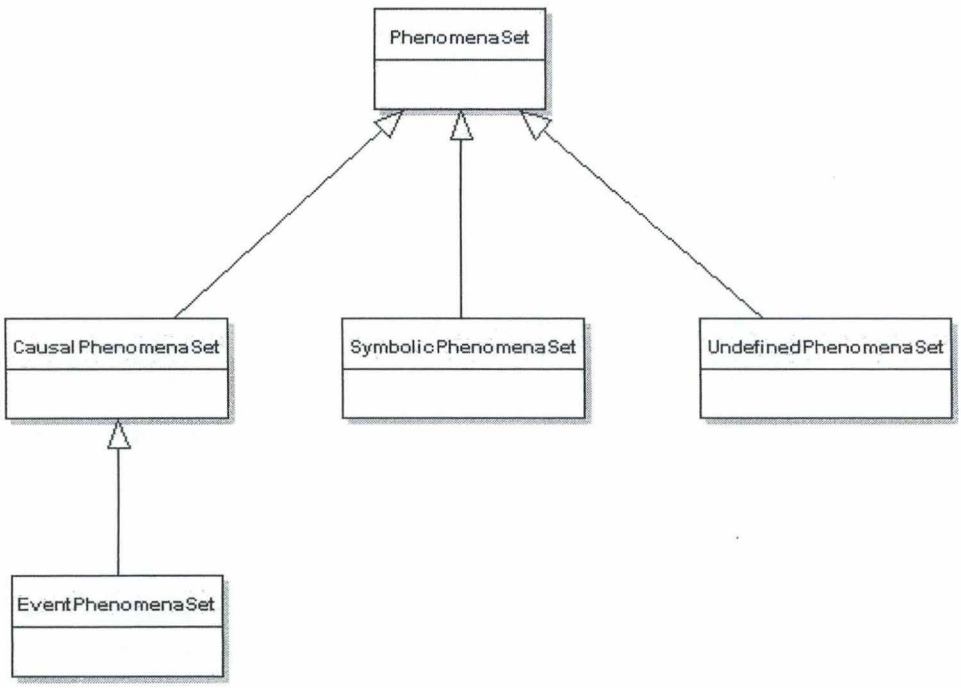
PhenomenaSet.type = Event implies PhenomenaSet.phens->forall(ph : Phenomenon | ph.oclIsType(Event))

Type Symbolic : ensemble de phénomènes constitué de Value, de Truth et/ou de Symbolic State.

PhenomenaSet.type = Symbolic implies PhenomenaSet.phens->forall(ph : Phenomenon | ph.oclIsType(Value) or ph.oclIsType(Truth) or ph.oclIsType(SymbolicState))

Type Undefined : ensembles de phénomènes constitué de phénomènes de type causal et de phénomènes de type symbolic.

PhenomenaSet.type = Undefined implies PhenomenaSet.phens->exists(ph : Phenomenon | ph.oclIsType(Event) or ph.oclIsType(Role) or ph.oclIsType(CausalState)) and PhenomenaSet.phens->exists(ph : Phenomenon | ph.oclIsType(Value) or ph.oclIsType(Truth) or ph.oclIsType(SymbolicState))



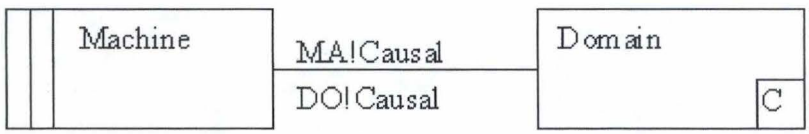
Un phénomène est global. S'il fait partie d'un ensemble de phénomènes, par exemple, de type causal, les autres PhenomenaSet dont il fera partie seront de type Causal ou Undefined.

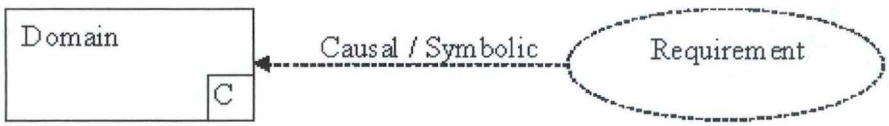
2.4 Typage des domaines

Jackson distingue trois types de domaine : Causal, Biddable et Lexical. Chacun de ses types possède des caractéristiques différentes concernant les types de phénomènes. Le but est de pouvoir déterminer le type d'un domaine à partir du typage des ensembles de phénomènes associés ou, à défaut, de contrôler la cohérence entre le type de domaine et le type des ensembles de phénomènes.

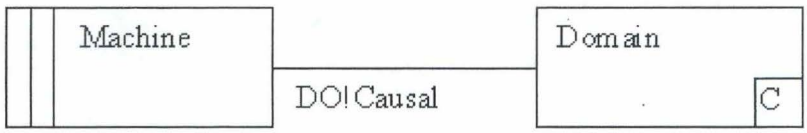
2.4.1 Type Causal

Jackson définit un domaine causal comme un domaine dont les propriétés permettent de prédire le comportement à travers ses phénomènes causaux. Il existe une relation de causalité qui permet de mesurer l'effet du comportement de la machine à l'interface avec le domaine. Souvent les domaines causaux vont présenter à leur interface des phénomènes causaux contrôlés par la machine. La machine pourra mesurer l'effet sur le comportement du domaine à travers d'autres phénomènes causaux du domaine. L'exigence sera exprimée généralement en terme de phénomènes du domaine. Ces domaines peuvent être rangés dans la catégorie inerte ou réactif (voir dimensions) suivant que l'on a un feedback ou non (DO !Causal optionnel). On aura donc le profil suivant :





D'autres domaines causaux sont les domaines actifs. Ils causent spontanément des événements et des changements d'états. Ils contrôlent tous leurs phénomènes à l'interface avec la machine. L'exigence n'exerce pas de contraintes sur eux. Rien en dehors du domaine ne peut affecter leur comportement. Ils présentent le profil suivant :



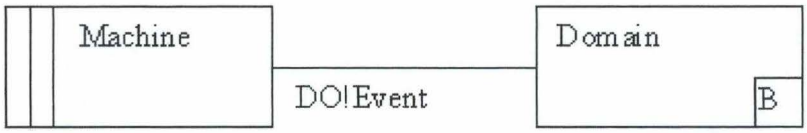
Contraintes :

Dans le contexte du domaine Causal, tous les ensembles de phénomènes partagés sont de type Causal.

Domain.type = Causal implies Domain.iLinks.phenSets.type=Causal

2.4.2 Type Biddable

La plupart du temps ces domaines représentent des êtres humains. Ce type de domaine cause spontanément des événements et des changements d'états. Il interprète les phénomènes partagés avec d'autres domaines et agit selon son bon vouloir. Le terme "conciliant" est utilisé dans une perspective optimiste de l'accomplissement de son rôle dans le problème. Il agit pour que l'exigence puisse être vérifiée. On va retrouver typiquement des événements à son interface avec la machine. L'exigence ne va logiquement pas exercer de contrainte sur son comportement vu son caractère conciliant (on ne peut pas le contraindre à initier un événement). On aura donc le profil suivant :



Contraintes :

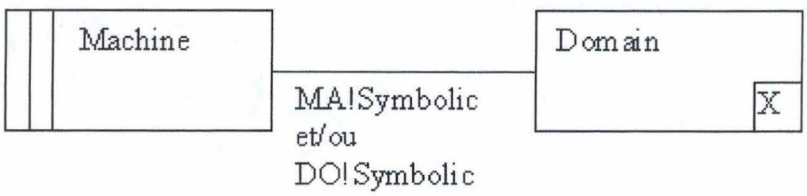
Exprimé en terme de typage des ensembles de phénomènes, un domaine de type Biddable signifie que, à l'interface, les ensembles de phénomènes seront de type Event contrôlés par ce domaine, et que l'exigence s'exprimera en terme de ces mêmes événements (commandes).

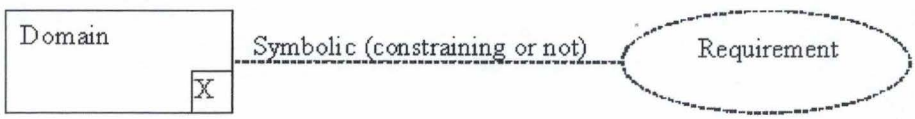
Domain.type = Biddable implies Domain.iLinks.phenSets.type=Event and Domain.Controls' includesAll(Domain.iLinks.phenSets) and Domain.rlinks.phenSets' includesAll(Domain.iLinks.phenSets)

2.4.3 Type Lexical

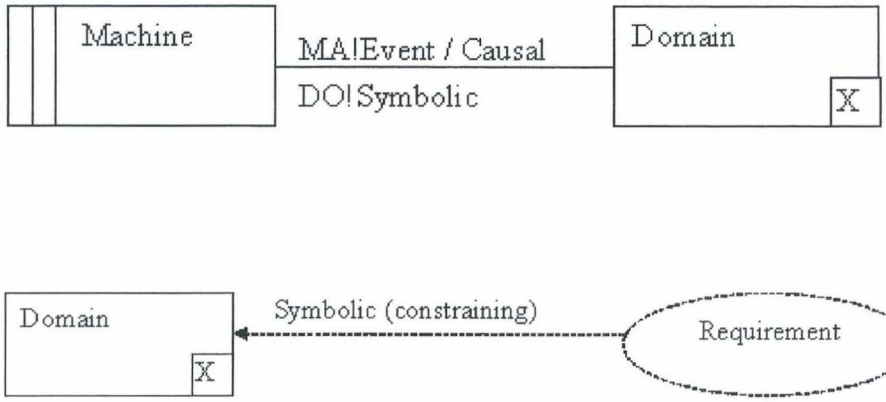
Un domaine lexical est une représentation physique de données (phénomènes symboliques). Les phénomènes causaux fournissent la représentation physique des phénomènes symboliques. Dans le cas où l'intérêt n'est porté qu'à l'aspect statique, les phénomènes symboliques peuvent être déterminés à l'interface. On aura les profils suivants :

Statique





Dynamique (inerte)



Les phénomènes privés sont uniquement des phénomènes symboliques.

Contraintes :

Tous les ensembles de phénomènes contrôlés par le domaine lexical sont de type Symbolic. Le domaine lexical va toujours présenter un ensemble de phénomènes observables. Les ensembles de phénomènes référencés par l'exigence sont également de type Symbolic.

La notion de contrôle d'ensembles de phénomènes s'exprime différemment selon que les phénomènes sont des événements, auquel cas on peut la traduire par "produit par" ou, par exemple, des états pour lesquels cette notion peut se traduire par "observable dans".

2.5 Dimensions des domaines

Les caractéristiques d'un domaine déterminent comment il va interagir avec d'autres domaines et avec la machine. [1, p67].

Jackson caractérise les domaines suivant plusieurs dimensions [1, pp69-70], [2, pp 143-174] et [MM, pp6-7] :

- La dimension de réactivité : une dimension qui mesure le changement du domaine dans le temps et sa réactivité face aux stimuli extérieurs. Dans cette dimension, on distingue les domaines statiques qui ne changent pas dans le temps et les domaines dynamiques qui changent dans le temps : changement d'états et producteur d'événements.
  Parmi les domaines dynamiques, on peut distinguer trois types :
  - Inerte : il a des états et change uniquement à cause de stimuli exté-

rieur : typiquement, le domaine Workpiece d'un Workpiece problem frame est inerte.
- Réactif : il a des états, il change et agit uniquement sur base de stimuli extérieurs.
- Actif : il a des états, change et agit de manière spontanée.

Parmi les domaines dynamiques actifs, on peut distinguer trois types qui correspondent à la dimension de servitude :
- Autonome : on ne peut l'influencer.
- Biddable : on peut le faire coopérer mais on n'est pas sûr du résultat (contrairement au domaine réactif).
- Programmable : on peut l'influencer totalement.

- La dimension de tolérance : domaine fragile, inhibiteur ou robuste.
- La dimension formelle : domaine formalisé ou non.
- La dimension de continuité : domaine discret ou continu.
- La dimension structurelle : le domaine à une structure arborescente, séquentielle, relationnelle ou en graphe acyclique dirigé.

Un domaine peut avoir une certaine dimension dans une description et une autre dimension dans une autre description : par exemple, un domaine lexical généré dans un problème a une dimension dynamique et ce même domaine utilisé dans un autre problème a une dimension statique.

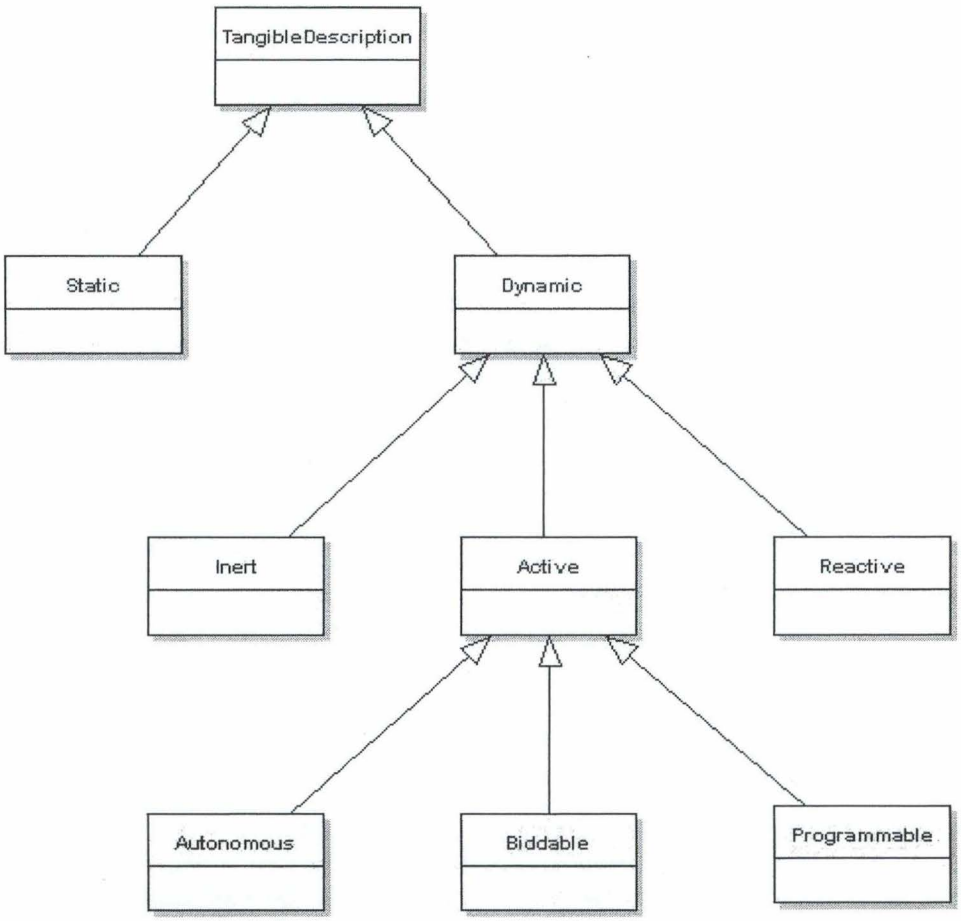
Dans [2], dans le cadre de l'étude des PF, Jackson décrit les domaines du point de vue des dimensions évoquées ci-dessus. Ces dimensions vont déterminer le comportement des domaines et, donc, préciser les caractéristiques du domaine du point de vue de ses phénomènes.

Par exemple [2,p93], dans Information display frame, RealWorld est un domaine complètement autonome. Cette caractéristique du domaine est importante dans le cadre du PF et apporte des contraintes plus précises au niveau des ensembles de phénomènes associés au domaine. Elle est aussi importante dans le cadre de la composition des différents sous-problèmes qui ont des domaines en commun.

Les caractéristiques du domaine dépendront bien évidemment de la sélection des phénomènes importants pour chaque domaine.

2.5.1 Caractérisation par la dimension de réactivité

La dimension de réactivité permet d'établir la hiérarchie suivante :

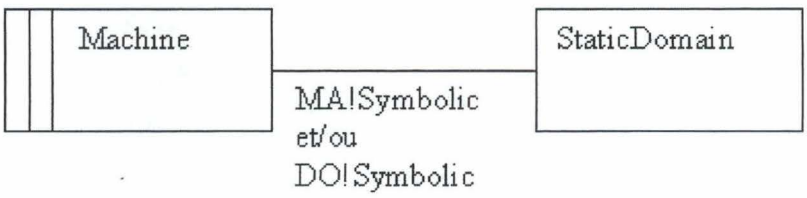


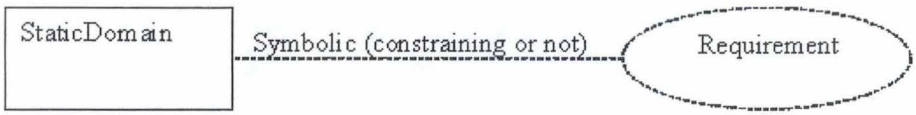
Pour chaque type de domaine suivant la classification par dimension, nous définissons les différents typages d'ensembles des phénomènes.

Static

Un domaine statique est un domaine qui n'a pas de dimension de temps. Il n'y a pas d'événements ; rien ne se passe et rien ne change. Le domaine est toujours dans le même état (ou il n'y a pas d'état).

Description en terme d'interface :





Description en terme de scope :

Il y a uniquement des phénomènes symboliques : Value, Truth ou SymbolicState.

Contraintes :

Domain.dimension = Static implies Domain.scope.phens->forall(ph : Phenomenon | ph.oclIsType(Value) or ph.oclIsType(Truth) or ph.oclIsType(SymbolicState))

Si le "scope" contient uniquement des phénomènes symboliques, les ensembles de phénomènes référencés par les interfaces et le requirementReference ne seront constitués, par déduction [MM,contraintes 14 et 20], que de phénomènes symboliques.

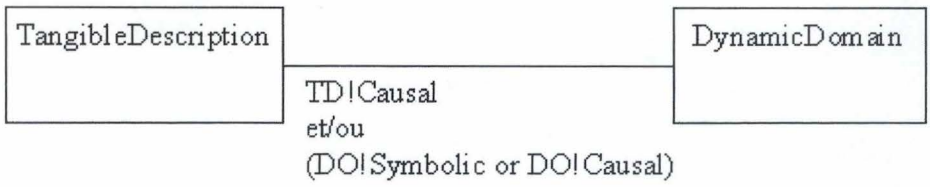
Domain.dimension = Static implies Domain.iLinks.phenSets->forall(phSet : PhenomenaSet | phSet.type = Symbolic) Domain.dimension = Static implies Domain.rLinks.phenSets->forall(phSet : PhenomenaSet | phSet.type = Symbolic)

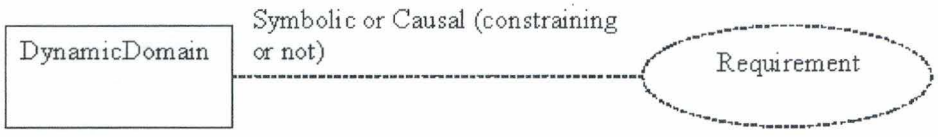
La relation 'implies' ($\Rightarrow$) pourrait être remplacé par une équivalence ($\Leftrightarrow$). Cela veut dire qu'on peut déduire la dimension du domaine à partir de ses phénomènes.

Dynamic

Contrairement au domaine statique, la notion de temps est importante pour les domaines dynamiques. Il y a des choses qui se passent et des choses qui changent. On va retrouver dans la description du domaine des phénomènes permettant d'exprimer ce caractère dynamique : des phénomènes causaux.

Description en terme d'interface :





Description en terme de scope :

Présence de phénomènes causaux : Event, Role ou CausalState.

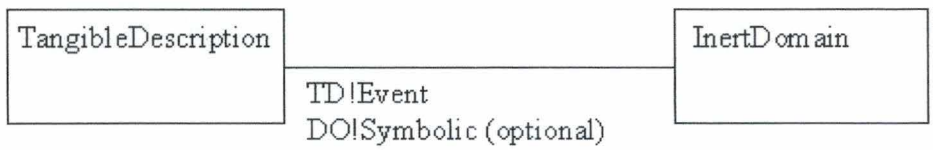
Contraintes :

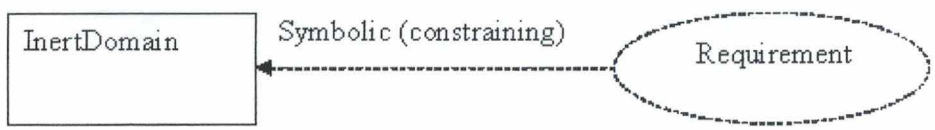
Domain.dimension = Dynamic implies Domain.scope.phens->exists(ph : Phenomenon | ph.oclIsType(Event) or ph.oclIsType(Role) or ph.oclIsType(CausalState))

Inert

Le domaine inerte peut changer ses états en réponse à des événements contrôlés extérieurement mais il ne peut initier de changements d'états et d'événements.

Description en terme d'interface :





Description en terme de scope :

Il n'y a pas de contrainte supplémentaire au niveau du scope. Les caractéristiques sont exprimées en terme d'interface.

Contraintes :

Domain.dimension = Inert implies Domain.iLinks.phenSets->forall(phSet : PhenomenaSet | (phSet.type=Event and Domain.Controls->excludes(phSet))

or (phSet.type=Symbolic and Domain.Controls->includes(phSet)) and Domain.iLinks.phenSets->exists(phSet : PhenomenaSet | phSet.type=Event)

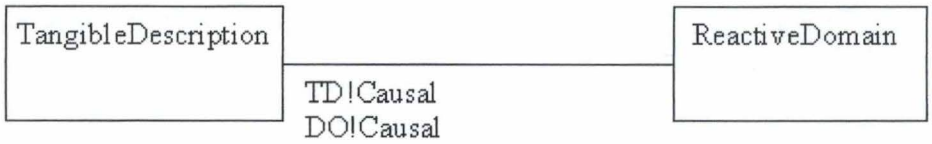
Soit le phSet est de type Event et le domaine ne le contrôle pas ou soit il est de type symbolic et il est contrôlé par le domaine Il existe au moins un phSet de type Event.

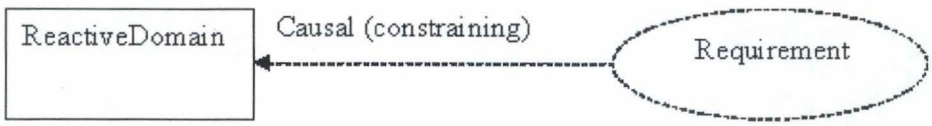
Domain.dimension = Inert implies Domain.rLinks.Constraining= true and Domain.rLinks.phenSets→forall(phSet : phSet.type=Symbolic)

Reactive

Le domaine est réactif s'il effectue certaines actions en réponse à des stimuli extérieurs. Les stimuli extérieurs vont être concrétisés par des ensembles de phénomènes causaux contrôlés par les TangibleDescription à l'interface. La " réactivité " du domaine pourra être observable à travers des ensembles de phénomènes causaux contrôlés par le domaine (le feedback). Le requirement sera contraignant en terme de phénomènes causaux également.

Description en terme d'interface :





Contraintes :

Domain.dimension = Reactive implies Domain.iLinks.phenSets.type=Causal

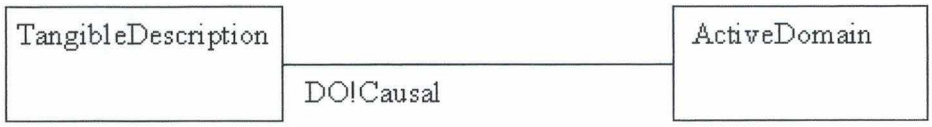
Domain.dimension = Reactive implies Domain.rLinks.Constraining= true and Domain.rLinks.phenSets → forall(phSet : phSet.type=Causal)

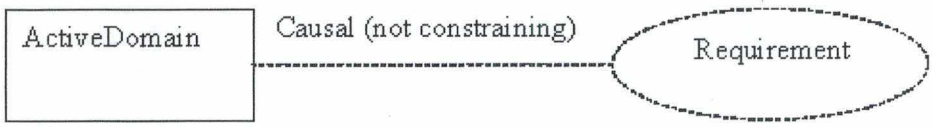
Active

Un domaine actif est un domaine qui effectue spontanément certaines actions sans qu'il y ait de stimuli extérieurs. Il n'y aura donc pas d'ensembles

de phénomènes causaux contrôlés par les TangibleDescription à l'interface. L' " activité " du domaine pourra être observable à travers des ensembles de phénomènes causaux contrôlés (produits) par le domaine. L'exigence ne sera évidemment pas contraignante et sera exprimée en terme de phénomènes causaux[1].

Description en terme d'interface :





Contraintes :

Domain.dimension = Active implies Domain.iLinks.phenSets.type=Causal and Domain.Controls→includesAll(Domain.iLinks.phenSets)

Domain.dimension = Reactive implies Domain.rLinks.Constraining= false and Domain.rLinks.phenSets→forall(phSet : phSet.type=Causal)

Jackson apporte une distinction au niveau des domaines actifs dans [1, p70] concernant la "servitude" du domaine :

- S'il est autonome, il est complètement indépendant.
- S'il est conciliant, il est indépendant mais influençable.
- S'il est programmable, il est complètement "asservi" : nous contrôlons entièrement, en tant que concepteur, son comportement ; comportement qui est actif. Typiquement, il s'agit d'un ordinateur (general-purpose computer).

Autonomous

Les domaines autonomes sont des domaines actifs : ils causent spontanément des événements et des changements d'états. Ils contrôlent tous leurs

[1]Les domaines programmables sont à considérer comme donnés pour un sous-problème même s'ils constituent une machine dans un autre sous-problème

phénomènes à l'interface. Les autres domaines ne peuvent affecter leur comportement. L'exigence n'exerce pas de contrainte sur eux. Il n'y a pas de caractéristiques supplémentaires, en terme de phénomènes et d'ensembles de phénomènes, qui permet de les différencier des domaines actifs.

Biddable

Les domaines conciliants sont des domaines actifs : ils causent spontanément des événements et des changements d'états. Ils contrôlent tous leurs phénomènes à l'interface. Rien ne peut affecter leur comportement. L'exigence n'exerce pas de contrainte sur eux. Ils se comportent donc comme les domaines Autonomous. Bien qu'on ne puisse mettre une exigence sur un domaine conciliant, le comportement idéal du système requiert que ce domaine respecte les demandes de l'analyste.

Programmable

Les domaines programmables se comportent comme les domaines Autonomous.

Nous ne voyons pas l'intérêt de caractériser les domaines suivant la dimension de servitude en terme de typage de phénomènes et ensembles de phénomènes. Ils présentent tous les trois les mêmes caractéristiques qui sont celles d'un domaine dynamique actif. La correspondance du point de vue typage doit se faire au niveau dynamique actif et pas en considérant la dimension de servitude.

Active ou Reactive

Nous considérons que le comportement important dans le cadre d'un PF sera toujours soit actif, soit réactif. Un type de problème se focalisera toujours sur l'aspect actif plutôt que sur l'aspect réactif ou vice versa.

2.5.2 Caractérisation par la dimension de tolérance

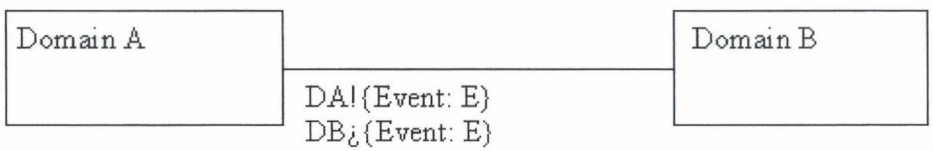
Une caractéristique importante des domaines causaux dynamiques est leur capacité de tolérer des événements contrôlés extérieurement en permettant ou non que seuls certains d'entre eux ne causent des changements d'états du domaine. Une classification simple distingue trois niveaux de tolérance [2, pp152-153] :

- Comportement robuste : le domaine permet que certains événements se passent sans que cela ne produise de changement d'état.

- Comportement inhibiteur : le domaine ne permet pas que certains événements se passent dans certains états du domaine. Le domaine empêche que ces événements aient lieu. Par exemple, les propriétés mécaniques d'un interrupteur permettent uniquement deux positions, allumé ou éteint : il n'y a pas de position intermédiaire et lorsque l'interrupteur est dans une position, on ne peut le bouger que dans l'autre position.
- Comportement fragile : le domaine permet que certains événements se passent mais risque de se mettre dans un état inconnu.

Cette dimension de tolérance affecte la description du domaine mais ne semble pas avoir d'incidence sur les types de phénomènes ou d'ensemble de phénomènes ni sur les profils d'interface.

Elle n'intervient pas non plus dans le contrôle d'un ensemble de phénomènes. Dans le cas du comportement inhibiteur, des commandes lancées par un domaine sont inhibées ou non par le domaine qui les reçoit, mais c'est toujours le premier domaine qui contrôle ces phénomènes. Jackson suggère d'indiquer cette notion d'inhibition au niveau de l'interface comme une nuance de contrôle des phénomènes [2, pp153-154]. Si un domaine A contrôle des événements qu'un domaine B inhibe, on peut le représenter de la manière suivante :



Cela permet de factoriser la notion d'inhibition pour un ensemble de phénomènes et donc de simplifier la description du domaine inhibiteur.

Nous pensons qu'il ne faut pas reprendre cette notion d'inhibition dans le métamodèle au niveau de l'interface. Par contre, il peut être utile de représenter la notion de tolérance comme attribut d'un domaine lorsque celui-ci est fragile. En effet, l'analyste doit en tenir compte pour s'assurer que l'exigence sera satisfaite. Cette notion peut être représentée comme attribut du domaine, du même type que Designed [MM, p5] : Fragile : BoolUndef

2.5.3 Caractérisation par la dimension formelle

Afin de pouvoir décrire un domaine informel, il convient de construire des désignations de phénomènes qui ne sont rien d'autre que des formalisations. Ces formalisations sont des abstractions plus ou moins grossières en fonction des besoins de l'analyse [2, pp313-314]. Par exemple, dans le problème du

distributeur de colis, dans le sous-problème 2 (distribution correcte des colis) il convient peut-être de distinguer plusieurs étapes lorsque le colis passe devant le détecteur i :

- (a) Le CausalState SensOn(i) est false avant que le colis ne passe devant le détecteur i.
- (b) Le CausalState SensOn(i) est true au moment où l'avant du colis passe devant le détecteur i.
- (c) Le CausalState SensOn(i) est true aussi longtemps que le colis se trouve devant le détecteur i.
- (d) Le CausalState SensOn(i) est false lorsque l'arrière du colis n'est plus devant le détecteur i.

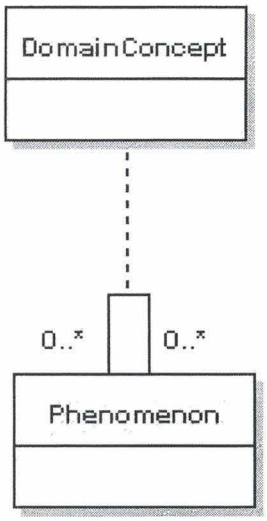
En effet, si l'on peut actionner l'embranchement correspondant au détecteur i suffisamment vite au moment où l'arrière du colis passe le détecteur ou si l'on peut actionner l'embranchement pour le prochain paquet lorsque l'avant du colis passe le détecteur, alors il est utile de faire cette abstraction fine.

Par contre, dans un autre sous-problème, le sous-problème 3 (afficher les colis mal acheminés), on peut considérer les étapes (b) à (d) comme formant un seul événement. Il n'est alors pas nécessaire d'utiliser l'abstraction fine. Une détection simple du passage du colis devant le détecteur i suffit.

Ces granularités différentes de l'abstraction d'un même phénomène doivent être mises en correspondance lorsqu'il faut comparer les deux sous-problèmes. Cette étape de comparaison est indispensable lors de la composition des sous-problèmes [2, p315].

Dans l'exemple, le même phénomène est mis en correspondance en considérant par exemple que le passage du colis dans le sous-problème 3 correspond à l'étape (b) dans le sous-problème 2. Si cette identification est consistante pour toute la description, alors les descriptions sont comparables.

Nous considérons que le concept de phénomène dans le métamodèle peut être complété par une classe d'association entre phénomènes. Cette classe d'association représente les concepts communs du domaine partagés par plusieurs phénomènes.



Cela permet à l'analyste de lier les phénomènes pour lesquels il devra chercher des points de comparaison.

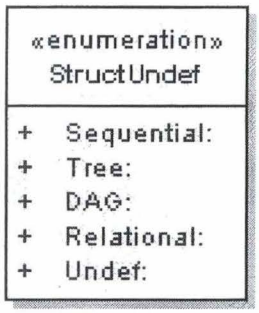
2.5.4 Caractérisation par la dimension de continuité

De manière générale, dans un domaine causal, il y a des phénomènes discrets et des phénomènes continus. Il peut être utile de former des approximations discrètes de phénomènes continus. Ces approximations sont des désignations comme pour tout autre phénomène. Il peut cependant s'avérer hasardeux de faire ces approximations discrètes trop prématurément [2, p156].

Nous ne tenons pas compte de cette dimension pour caractériser un domaine.

2.5.5 Caractérisation par la dimension structurelle

Il peut être très utile de donner une indication sur la structure des domaines statiques et des domaines lexicaux. Cette structure peut être représentée par un attribut de domaine pour donner une indication sur la notation à utiliser : Structure : StrucUndef. Le type StructUndef est un type énuméré dont les valeurs possibles sont : sequential, tree, DAG, relational, Undef [MM, p7].



2.6 Cohérence entre typage et correspondance

Tout le développement effectué dans les points précédents nous permet de définir de nouvelles règles de cohérence entre type de domaine et type d'ensembles de phénomènes, de nouvelles contraintes de correspondance.

2.6.1 Typage de domaine et typage d'ensembles de phénomènes

Un domaine est caractérisé par son type : Causal, Biddable ou Lexical. Pour chaque type, nous avons énoncé les règles à respecter en termes de type d'ensembles de phénomènes et de contrôle de ces ensembles.

Un domaine peut être caractérisé par sa dimension de réactivité : Static, Inert, Active ou Reactive. La dimension impose également certaines caractéristiques, en général plus contraignantes, en terme de type d'ensembles de phénomènes et de contrôle de ces ensembles.

Le type d'un ensemble de phénomènes (Symbolic, Causal, Event, Undefined) est déterminé à partir du type des phénomènes (Event, Value, Entity, Truth, CausalState, SymbolicState, Role) constituant cet ensemble. Les ensembles de phénomènes d'un PF sont aussi typés.

On peut en conclure que le type de domaine ou la dimension de réactivité d'un domaine doit rester cohérents avec le type des phénomènes constituant le scope du domaine et, plus particulièrement, avec le type des événements partagés aux interfaces ou référencés par l'exigence. Le type et la dimension représentent un typage non orthogonal des domaines.

Les domaines dans le cadre d'un PF sont obligatoirement typés et peuvent être caractérisés par leur dimension.

2.6.2 Correspondance

Le métamodèle spécifie un certain nombre de contraintes concernant la correspondance entre un problème et un PF [MM, p21-22], essentiellement :

- correspondance au niveau des concepts (topologie du problème),
- correspondance pour les types de domaine.

Dans les sections précédentes, nous complétons le métamodèle avec la notion de type d'ensemble de phénomènes et de dimensions de domaine.

Nous intégrons ces différentes notions dans les contraintes permettant de vérifier la correspondance.

Ensembles de phénomènes

Un ensemble qui correspond à un autre ensemble de phénomène doit être du même type.

Domaines

Si la dimension du domaine est spécifiée au niveau du PF, les domaines qui correspondent et les domaines à faire correspondre doivent avoir la même dimension.

Chapitre 3

Spécification des fonctions de l'outil

Si les chapitres précédents présentaient une introduction à la méthode des problem frames et une formalisation des concepts à travers un métamodèle, ce chapitre-ci rentre au coeur de l'outil en décrivant ce que l'analyste peut espérer d'un outil d'aide à l'analyse par cette méthode.

Nous parcourons succinctement toutes les fonctionnalités classiques d'un éditeur graphique, les fonctionnalités disponibles dans tous les éditeurs.

Nous nous penchons plus en détail sur les fonctionnalités plus spécifiques aux PF.

Nous présentons les fonctionnalités de manière informelle. Dans l'annexe B, le lecteur pourra trouver des exemples de description plus formelle qui utilisent les cas d'utilisation et, dans l'annexe C, une approche utilisant la méthode des PF.

Le but de ce chapitre n'est pas de définir un cahier des charges d'une application. Nous ne parlons pas d'exigences non fonctionnelles, de modèle d'interface graphique ou encore de spécifications techniques. Le but est de donner une vue d'ensemble de ce qu'un outil peut offrir pour aider l'analyste à définir un problème avec la méthode des PF.

3.1 Fonctionnalités classiques d'un éditeur graphique

Parmi les fonctions classiques, un ensemble de services que l'outil offrira comprend la création, l'édition, la suppression et la sauvegarde (persistance) des différents diagrammes. Cela reprend évidemment aussi bien les diagrammes de problème, les liens entre les problèmes et les sous-problèmes, les diagrammes de PF, les liens entre les PF et les problèmes et les objets

figurant dans les diagrammes.

Toute la gestion des objets manipulés s'appuie sur le métamodèle. Il faut donc greffer, à ces fonctions de "gestion", des fonctions de validation qui permettent de vérifier la cohérence avec le métamodèle (contraintes imposées par le métamodèle).

L'essentiel des fonctionnalités suivantes sont décrites dans les cas d'utilisation (annexe B) et dans les diagrammes de PF (annexe C) :

- Édition des Problèmes/PF ;
- Validation des Problèmes/PF ;
- Persistance des Problèmes/PF ;
- Création d'un problème à partir d'un PF.

3.2 Fonctionnalités spécifiques aux PF

D'autres fonctions plus spécifiques, intervenant dans le processus d'analyse d'un problème par la méthode des PF, seront nécessaires.

Chacune de ces fonctions adresse un problème particulier de la méthode d'analyse présentée au chapitre 1. Pour chacune d'elles, nous décrivons le problème dans le contexte général des problem frames. Ensuite nous décrivons ce que peut apporter un outil pour aider à résoudre ce problème.

3.2.1 Vérification de la décomposition d'un problème par projection

La décomposition d'un problème complexe en sous-problèmes plus simples, plus faciles à résoudre, est le principe conducteur de la méthode des PF. Une des caractéristiques importantes de la décomposition dans l'approche de Jackson, est qu'un sous-problème est une projection d'un problème. Ces sous-problèmes ne forment pas nécessairement des partitions. [2,p.65].

Le sujet est abordé dans [MM,pp.16-21]. Nous reprenons ci-dessous les principes essentiels :

- La projection est définie en terme de projections de partie du problème.
- Un requirement peut être une projection d'un seul autre requirement. Une description tangible peut être une projection de plusieurs autres descriptions tangibles. Cela implique qu'un domaine peut être projection d'une spécification. Il est important de considérer dans l'analyse d'un sous-problème particulier que les autres sous-problèmes sont résolus et, donc, de pouvoir considérer les sous-spécifications comme partie du domaine d'application dans le sous-problème considéré [2,p.60].
- Les projections de description sont définies en terme de scope et de

phénomènes.

- "Une description est une projection d'une ou plusieurs autres descriptions" signifie que son scope doit être inclus dans le scope de l'autre description ou dans l'union des scopes des autres descriptions.
- Chaque phénomène mentionné dans un problème doit également être mentionné dans au moins un de ses sous-problèmes ('complétude').

Chacun de ces principes fait l'objet de contrainte(s) précise(s) décrite(s) dans [MM,p.18].

L'outil va vérifier que la cohérence est préservée entre le problème et les sous-problèmes résultant de la projection. Il va le faire sur base du respect des contraintes énoncées plus haut et sur base de nouveaux paramètres que nous décrivons ci-dessous.

Fonctionnement

L'opération de vérification de la cohérence de la projection entre le problème principal et un sous-problème s'effectue quand le sous-problème est défini.

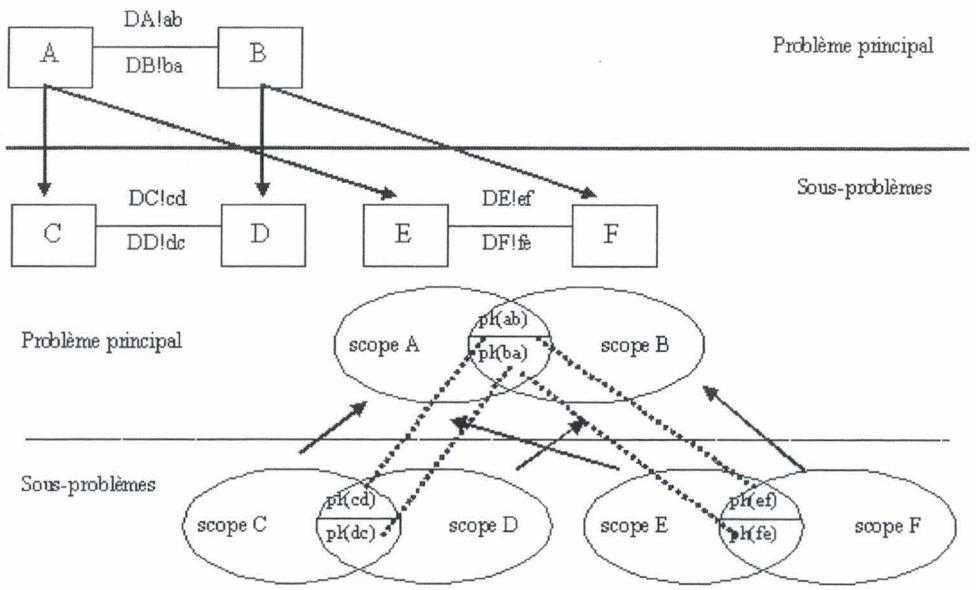
Dans le processus d'itération de la méthode d'analyse, la résolution des premiers sous-problèmes peut soulever de nouvelles interrogations et conduire à définir de nouveaux sous-problèmes (sous-problèmes 4, 5 et 6 dans l'exemple du distributeur de colis ; cf.chapitre 1). Nous considérons que l'incorporation d'éventuels nouveaux éléments se fait d'abord dans le problème principal et, par projection, dans les sous-problèmes.

La définition de la projection en terme d'inclusion de scope est très générale. Nous tentons d'étendre la notion de projection aux notions de contrôle d'ensembles de phénomènes et à la notion de type.

Projection et contrôle

La relation de contrôle doit être conservée à travers la relation de projection. Un phénomène faisant partie d'un ensemble de phénomènes contrôlés par un domaine dans le problème principal devra être inclus, au niveau du sous-problème, dans un ensemble contrôlé par un domaine, projection du domaine du problème principal ; et vice-versa.

Graphiquement on peut poser le problème comme suit :



$ph(ab) = ph(cd) \cup ph(ef)$

Nous introduisons la notion de scope partiel. Le scope partiel d'un domaine A_i dans un domaine A' projection de $(A_0, \ldots, A_i, \ldots, A_n)$ domaines, noté $scope_{A_i}(A')$, est l'ensemble des phénomènes qui font partie du scope de A' et du scope de A_i.

On a aussi :

$scope_A(A') = scope(A')$ si A' projection de A uniquement.

$scope(A') = scope_A(A') \cup scope_B(A')$ si A' projection de A et de B.

En généralisant :

$scope(A') = \bigcup_{i:0->n} scope_i(A')$

où A' projection de $(A_0, \ldots, A_i, \ldots, A_n)$.

La notion de scope partiel peut s'étendre aux ensembles de phénomènes contrôlés par un domaine.

Soit $ph_A(scope(A) \cap scope(B))$: ensemble des phénomènes de l'interface entre A et B contrôlé par A.

On a :

$ph_A(scope(A) \cap scope(B)) = \bigcup_x ph_A^x(scope_A(A^x) \cap scope_B(B^x))$

L'égalité vérifie la cohérence au niveau de l'information de contrôle. Elle suppose que le scope partiel est connu. Si ce n'est pas le cas, il faudra le déterminer (voir détermination du scope partiel).

Projection et typage

Il faut déterminer les caractéristiques de typage à conserver à travers la relation de projection et étudier les éventuelles contraintes limitant cette relation de projection. Nous étudions ces caractéristiques pour les types d'ensembles de phénomènes, les types des domaines et les dimensions associées aux domaines. Par définition, les phénomènes sont globaux et conservent donc leur type à travers la projection.

Type d'ensembles de phénomènes

Le type d'un ensemble de phénomènes dépend du type des phénomènes qui le composent et, comme les ensembles constitués au niveau du problème principal par rapport aux sous-problèmes peuvent être différents, le type de l'ensemble pourra être différent.

Type de domaine

Le type du domaine est conservé à travers la relation de projection. En effet, un type Biddable ne pourra se transformer en type Causal ou Lexical. Un type Causal ou Lexical ne pourra se transformer en Biddable. Reste à savoir si un type Lexical peut se transformer en Causal et inversement ? Jackson [2,p.84] considère que l'on peut voir un domaine Lexical de deux points de vue : soit comme une structure de phénomènes symboliques (type Lexical), soit comme un domaine Causal où des événements causent des changements d'états. Dans le chapitre 2 : Type Lexical, dans les deux cas, le type du domaine est pour nous un domaine de type Lexical. Nous considérons qu'un domaine de type Causal ne peut être projeté en domaine de type Lexical ; nous n'avons pas trouvé d'exemple dans la littérature pour l'infirmer.

La conséquence de la conservation du type d'un domaine à travers la relation de projection définit une nouvelle contrainte :

un domaine d'un sous-problème ne peut être projection que de domaine(s) du même type.

Dimensions des domaines

La dimension de réactivité peut changer suite à une projection. Par exemple, un domaine peut être vu comme statique dans un sous-problème et dynamique dans un autre sous-problème. Dans l'exemple du distributeur de colis (chapitre 1), le domaine Port Model est un domaine dynamique dans le sous-problème 6 et statique dans le sous-problème 5. Le domaine lexical est construit dans le sous-problème 6 et il est utilisé dans le sous-problème 5.

Nous n'avons pas pu dégager de contraintes à partir de la dimension de tolérance et de la dimension structurelle.

3.2.2 Vérification de la correspondance entre un problème et un PF (fitting)

Lors de la décomposition en sous-problèmes, le PF constitue un guide dans la constitution du sous-problème. Lorsque le sous-problème est complètement décrit, il faut vérifier qu'il correspond au PF choisi. Nous appelons cette opération, l'opération de vérification de la correspondance, ou de "fitting".

Un frame concern d'un PF (les aspects importants qui doivent être adressés par l'analyste pour résoudre ce PF) ne peut être correctement traité que si l'analyste choisit le PF approprié au problème. Si ce n'est pas le cas, le développement résultant sera certainement difficile à mener et ne sera probablement pas un succès.

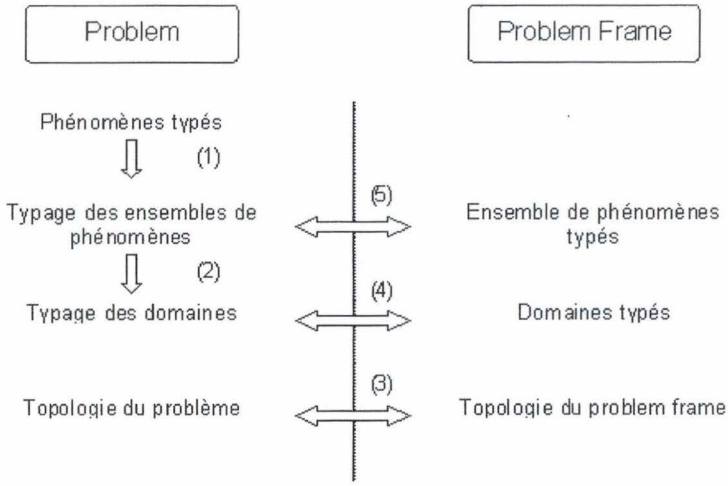
Une condition indispensable pour qu'un problème corresponde à un PF approprié est que toutes les propriétés du diagramme de problème correspondent aux propriétés de la classe de problème choisie.

Cette vérification se fait sur la topologie du problème mais également sur les caractéristiques des différents domaines décrits.

Fonctionnement

L'analyste a effectué la description des différentes composantes du problème : la machine, les domaines et le requirement. Ces descriptions ont été faites dans une notation appropriée et importées dans l'outil grâce à la fonction d'importation des notations externes décrite plus loin (3.2.3).

Le processus de vérification de la correspondance va se dérouler conformément au schéma ci-dessous.



(1) Pour chaque ensemble de phénomènes, en nous basant sur les phénomènes typés [1] constituant cet ensemble, nous déterminons son type.

(2) Nous estimons ensuite le type du domaine à partir des ensembles de phénomènes contrôlés ou référencés par le domaine au niveau des interfaces.

(3) Les topologies du problème et du PF doivent être semblables.

(4) Les propriétés des domaines spécifiées dans le PF doivent être identiques aux propriétés des domaines dans le problème.

(5) Le type des ensembles de phénomènes correspond.

Etape 1 : Typage d'ensemble de phénomènes

Le type d'un ensemble de phénomènes (Symbolic, Causal, Event, Undefined) est déterminé à partir du type des phénomènes (Event, Value, Entity, Truth, CausalState, SymbolicState, Role) constituant cet ensemble.

Cette opération est effectuée pour chaque ensemble de phénomènes faisant partie des interfaces avec d'autres domaines ou la machine et pour chacun des ensembles de phénomènes du domaine référencés ou contraints par le requirement.

Nous énonçons les règles permettant de déduire le type d'un ensemble de phénomènes au chapitre 2 : Typage des ensembles de phénomènes.

[1] Les phénomènes extraits d'une description réalisée avec une notation externe sont obligatoirement typés (3.2.3).

Etape 2 : Typage de domaine

Pour chaque type de domaine, nous avons énoncé les règles à respecter en termes de type d'ensembles de phénomènes et de contrôle de ces ensembles dans le chapitre 2 : Typage des domaines. C'est ce que nous appelons les profils associés aux types de domaine.

Dans cette étape, nous déterminons le profil du domaine. Ce profil doit correspondre au type du domaine, si cette information est déjà fournie. Si ce n'est pas le cas, nous ne pouvons pas déduire de manière univoque le type du domaine à partir de la seule information de profil. Le profil sera cependant utilisé dans l'étape suivante de correspondance.

Dans la section 2.5 (Dimension des domaines), nous introduisons la notion de dimension de réactivité qui permet de caractériser plus finement les domaines et de déterminer comment ils vont interagir avec leur environnement.

Des profils sont aussi associés à cette dimension. Nous les utilisons de façon similaire au typage du domaine.

On peut en conclure que le type de domaine ou la dimension de réactivité d'un domaine doit rester cohérents avec le type des phénomènes constituant le scope du domaine et, plus particulièrement, avec le type des événements partagés au niveau des interfaces ou référencés par le requirement. Le type et la dimension représentent un typage non orthogonal des domaines.

Etapes 3, 4 et 5 : Correspondance

Les règles de correspondance sont énoncées à la section 2.6.2.

3.2.3 Description dans une notation appropriée

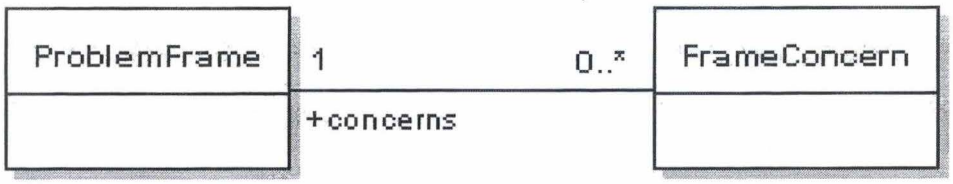
Un analyste va identifier dans un problème donné, les sous-problèmes qui correspondent à des PF et puis décrire ces problèmes/sous-problèmes à l'aide de notations qu'il jugera adéquates. L'outil va traduire ces descriptions dans le langage des PF. La traduction des descriptions en PFL (Problem Frame Language) est étudiée dans les 2 chapitres suivants.

Il s'agit de l'étape préliminaire à la réalisation de la fonction précédente.

3.2.4 Frame Concern

Cette fonctionnalité permet d'associer un FC à un PF. Ce FC est une liste des aspects importants dont l'analyste doit tenir compte.

Nous associons à chaque frame un ensemble de concerns.



Cette notion n'est pas applicable aux problèmes. Une nouvelle contrainte restreint la relation aux problem frames qui ne sont pas des problèmes :

forall pf IN ProblemFrame : pf IN Problem implies pf.concerns = EMPTY

Cette fonctionnalité permet de capitaliser la connaissance nécessaire à la résolution d'une classe de problèmes.

Cela comprend :

- Les notations à utiliser dans les descriptions des différents domaines.
- La méthode de développement à utiliser.
- Les aspects importants dont il faut tenir compte pour parvenir à combler le fossé qu'il y a entre les exigences et ce que la machine peut accomplir directement.

Une perspective intéressante serait d'explorer les possibilités de formalisation des FC et les possibilités de vérifier qu'un FC a été bien adressé.

3.2.5 Détection des problèmes potentiels de composition

Comme nous le décrivons dans la section 1.3.3, les problèmes de composition arrivent lorsque deux sous-problèmes comportent des projections différentes d'un domaine commun [2, p304]. Tous les cas sont potentiellement problématiques sauf si le domaine commun est autonome et que les scopes sont disjoints.

La fonctionnalité va déterminer les domaines qui sont susceptibles de poser problème et les phénomènes de ces domaines repris dans différents sous-problèmes (intersection des scopes partiels).

Il est possible d'avoir des problèmes de composition même si les scopes partiels dans les différents sous-problèmes sont disjoints. Cela arrive quand des liens de causalité existent entre des phénomènes qui se trouvent dans des scopes partiels différents (pour un même domaine ou des domaines différents du problème principal).

Exemple du chapitre 1 : le Package Conveyor et le Package Router sont des domaines différents dans le problème de distribution de colis et pourtant, l'arrêt du tapis roulant a une incidence sur le distributeur de colis.

Une possibilité de mettre en évidence ce type de problème, quand les scopes partiels sont disjoints, serait d'étendre la notion de scope d'un domaine aux phénomènes intervenant dans cette relation de causalité.

3.2.6 Lier des abstractions différentes d'un même phénomène

Lorsque l'analyste étudie les problèmes de composition, la première chose à faire est de s'assurer que les descriptions sont comparables (cf. point 2.5.3 : Caractérisation par la dimension formelle). Certains phénomènes sont liés entre eux car ils désignent le même concept du domaine avec des degrés d'abstraction différents.

La fonctionnalité permet de relier un ou plusieurs phénomènes d'une projection d'un domaine dans un sous-problème avec un ou plusieurs phénomènes d'une projection d'un même domaine dans un sous-problème différent.

Cette fonctionnalité a pour but de mettre en évidence les phénomènes liés afin que l'analyste puisse définir des points de comparaison. Une fois que ces points sont définis et qu'ils sont consistants pour l'ensemble du problème, les descriptions qui contiennent ces phénomènes liés sont comparables.

3.2.7 Entrepôt des problèmes classés par PF

Le but de cette fonctionnalité est de favoriser la connaissance générale des classes de sous-problèmes conformément à l'esprit de Jackson [2,p.60] en fournissant des exemples d'instanciation de PF.

Comme nous associons à un PF un ensemble de FC, nous associons à un PF un ensemble de problèmes correspondant à ce PF. L'entrepôt sera alimenté par l'outil au fur et à mesure par le travail de l'analyste.

3.2.8 Incorporation d'un sous-problème dans le problème principal

Dans la méthode itérative, l'analyste peut être amené à définir de nouveaux sous-problèmes lors de la résolution de sous-problèmes existants. Cette fonctionnalité permet de compléter automatiquement le diagramme global du problème avec les informations contenues dans le sous-problème, à savoir :

- les nouveaux domaines,
- les informations de contrôle sur les phénomènes,
- les phénomènes.

On peut calculer le scope d'un domaine ou de la machine du problème principal :

$scope(A) = \bigcup_{i:0->n} scope_A(A^i)$

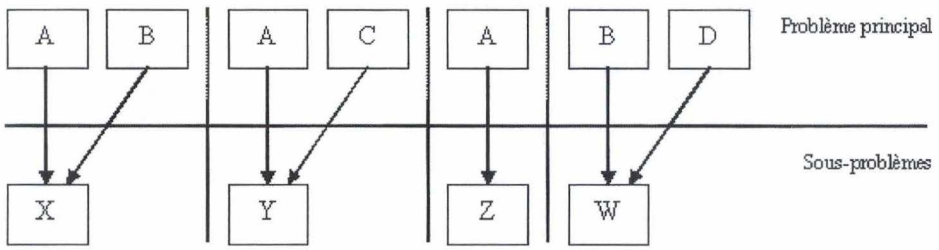
où A^i projection de A.

Le scope d'un domaine est maintenant défini en terme de scopes partiels au niveau des sous-problèmes. La détermination des scopes partiels suffit pour reconstruire le scope des domaines ou de la machine du problème principal. Comme nous l'avons vu, le scope partiel est le scope du domaine du sous-problème si le domaine est projection d'un seul domaine. Si le domaine est projection de plusieurs domaines, pour chaque phénomène de ce domaine, il faut déterminer à quel scope partiel du sous-problème il appartient. S'il fait partie du scope du domaine principal, il fera automatiquement partie du scope partiel de ce domaine.

Nous explorons ci-après un certain nombre de moyens de pouvoir déduire automatiquement cette information ou, en tout cas, nous donnons dans la mesure du possible suffisamment d'options pour calculer ces informations.

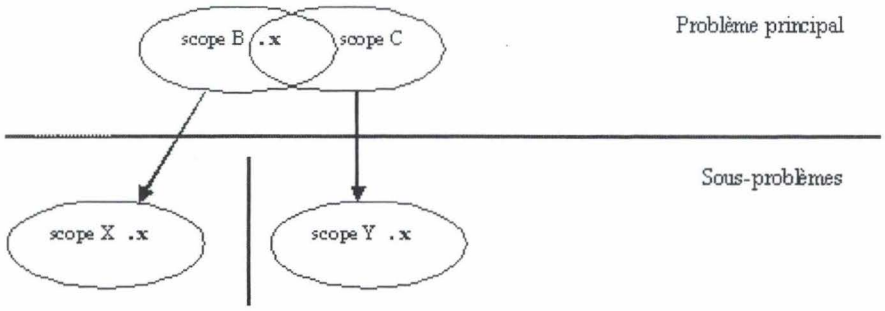
Analyse des autres sous-problèmes

Il est possible de déduire l'information d'appartenance à un scope partiel particulier en prenant en compte l'information disponible dans les autres sous-problème. Prenons les projections suivantes (X,Y,Z et W font partie de 4 sous-problèmes différents) :



Considérons un phénomène $x \in scope(X)$ et :

- $x \in scope(Z)$: on peut en déduire que $x \in scope_A(X)$
- $x \in scope(Y)$: on peut en déduire que :
  - soit $x \in scope_A(X)$
  - soit $x \in scope_B(X)$ et $x \in scope_C(Y)$ (élément de l'interface entre B et C)
  - soit $x \in scope_A(X), scope_B(X), scope_C(Y)$ (élément d'une interface reliant A, B et C).



- $x \notin scope(Y)$ et $x \in scope(W)$: on peut en déduire que :
  - soit $x \notin scope_A(X)$ et donc $x \in scope_B(X)$ (on peut penser raisonnablement que ce sera le cas)
  - soit $x \in scope_A(X)$ (il peut faire partie d'une interface entre A et B ou A et D...)

Contrôle

Une fois le scope partiel connu, il est alors possible de reconstituer les ensembles de phénomènes contrôlés par le domaine et évidement par différence les ensembles référencés par d'autres domaines (voir section 3.2.1 : projection et contrôle).

Chapitre 4

Intégration d'une notation

4.1 Introduction

La description détaillée d'un problème nécessite souvent une formalisation adaptée que le langage des PF ne permet pas. Jackson, dans [2] utilise fréquemment les diagrammes de machines à états, des arbres ou, encore, des symboles logiques pour décrire les différents domaines, l'exigence ou la spécification.

Une fonctionnalité indispensable d'un éditeur de PF est de pouvoir importer des descriptions réalisées dans une notation particulière et, cela, suivant une méthodologie bien fondée.

L'objectif de ce chapitre est de définir un cadre théorique pour l'intégration d'une notation et d'élaborer un cahier des charges reprenant les différents éléments permettant son intégration dans l'outil.

Nous présentons un modèle de sémantique pour les notations correspondant à l'esprit de Jackson.

Nous introduisons ensuite le concept de couche et plus particulièrement de couche de liaison. Quand un système doit être décrit à l'aide d'une notation, il faut que celle-ci puisse être compréhensible par un outil manipulant des concepts liés aux PF (ce que nous appelons, dans le métamodèle, les 'PFLConcept') et, notamment, les phénomènes. Il faut donc que l'outil puisse extraire ces concepts à partir des éléments spécifiques de la notation. C'est le rôle de la couche de liaison.

Ensuite nous appliquons le modèle à la fonction de projection de façon à pouvoir valider la cohérence sémantique de toute l'arborescence du problème (problème et sous-problèmes).

4.2 Canevas de sémantique

Nous voulons qu'une notation puisse exprimer le comportement d'un domaine ou de la machine confronté à son environnement. Intuitivement un système qui fonctionne est un système qui reçoit des signaux (phénomènes) de son environnement et auxquels il va réagir en **fonction** de ce qu'il reçoit. Le choix d'une sémantique fonctionnelle en découle tout naturellement.

Une sémantique fonctionnelle a déjà été utilisée par d'autres auteurs. La sémantique dénotationnelle classique [Stoy77], prévue pour des systèmes transformationnels, donne les sorties du programme comme une fonction de ses entrées. Kahn [Kahn77] a utilisé la sémantique fonctionnelle pour des réseaux connectés de façon asynchrone, une approche approfondie par Broy [Broy89,98] dans sa méthode FOCUS.

L'idée est d'exprimer un canevas de sémantique commun à toutes les notations ; ces notations s'appliquent à toutes les descriptions [1].

L'élaboration d'un canevas répond à un certain nombre d'exigences par rapport à une notation.

Un langage de description devra être suffisamment précis pour capturer les caractéristiques intéressantes du domaine.

Il permettra d'exprimer le comportement observable en réponse aux modifications de l'environnement et, le cas échéant, vérifier le comportement attendu.

Nous utilisons, dans ce but, la notion d'**instantané** de la description. Cet instantané (*snapshot*) représente l'état de la description en terme de phénomènes, englobant tous les phénomènes figurant dans les interfaces (PFLInterface, PFLRequirementReference). L'instantané ne reprend pas l'information de contrôle, de contrainte ou de référence liée à une description. Il correspond à une image de toutes les interfaces à un instant donné. Cette image reprend toute l'information contenue dans les phénomènes qui concernent une description et son environnement, par exemple, les valeurs des variables, les événements générés,etc.

Chaque description D possède son scope, noté $scope(D)$, qui représente l'ensemble des phénomènes qui sont contrôlés, référencés ou contraints par la description.

$scope : PFLDescription \rightarrow 2^{PFLPhenomenon}$

Un instantané s_D associe à chaque phénomène faisant partie de $scope(D)$ un booléen qui indique si le phénomène est présent (se produit) ou pas. L'ensemble des instantanés possibles d'une description D est noté Σ_D.

[1] Nous préfixons tous les éléments, faisant référence aux concepts du langage des problem frames décrits dans le métamodèle, par PFL (Problem Frames Language).

Nous appelons la fonction permettant de construire un instantané pour une description, *snapshot*, définie par :

$$snapshot : PFLPhenomenon \rightarrow Boolean$$

$$s_D \in \Sigma_D \Leftrightarrow dom(s_D) = scope(D) \wedge (\forall ph \in scope(D)\ ,\ \exists\, 1! b \in Boolean : snapshot(ph) = b)$$

Pour calculer le comportement d'une description face aux modifications de son environnement, il sera nécessaire de conserver un historique des instantanés. En effet, nous observons une description à travers son interface et nous ne connaissons pas les phénomènes internes qui vont également influencer la réaction. Ces phénomènes internes vont varier en fonction des différentes "réactions" qui se sont déjà produites.

Un instantané sera toujours un état courant obtenu à partir d'un état initial, noté s_D^0, et une série d'autres états. Nous appelerons "trace" la suite de ces états.

$$trace : snapshot^+$$

Soit T_D l'ensemble de toutes les traces pour la description D. Nous définissons une trace t_D comme :

$\forall t_D \in T_D : t_D = (s_D^0 \ldots s_D^n)$ avec $s_D^0, \ldots, s_D^n \in \Sigma_D$

Définition

Une notation spécifie le comportement d'un domaine ou de la machine. Ce comportement dépend de l'état de la description et de modifications de son environnement. Le comportement se traduit par une modification de l'état de la description observable à travers son interface avec l'environnement. Le requirement spécifie le comportement attendu qui va dépendre, de la même façon, de l'état et de modifications du domaine d'application et se traduit par une modification des propriétés du domaine d'application.

Le canevas de sémantique commun à toutes les descriptions est défini par :

$$[\![DL_D]\!]_{Jackson} = \{f_{Jackson} \mid \forall t \in T_D, \exists s \in \Sigma_D : f_{Jackson}(t) = s\}$$

La sémantique des notations utilisées pour modéliser les descriptions devra pouvoir être exprimée en respectant le canevas décrit ci-dessus.

La notation ne connaît pas le concept de phénomène et la sémantique d'une notation sera exprimée en terme d'éléments de la notation. Il faut donc faire le lien entre ce que nous appellerons la couche notation et la couche Jackson. Cette couche intermédiaire représente la "colle" permettant d'intégrer la notation à l'outil.

4.3 Couche de liaison

Dans la section précédente, nous évoquions l'intégration d'une notation à l'outil du point de vue de la sémantique et cette sémantique est définie à partir des phénomènes. Un langage de description (DL) devra donc donner la possibilité à l'outil d'extraire la liste des phénomènes qui constitueront le 'scope' de la description.

Cette information n'est pas suffisante. Pour pouvoir assurer les fonctions présentées au chapitre 3, l'outil aura besoin de connaître les propriétés des phénomènes nécessaires pour le typage, mais aussi les caractéristiques des ensembles de phénomènes. Les phénomènes sont partitionnés en ensemble de phénomènes (PFLPhenomenaSet) qui sont soit contrôlés, soit "observés" pour chaque domaine ou machine. De même, certains phénomènes peuvent être "contraints" ou "référencés" par le requirement. Ces informations doivent également être alimentées par les notations.

La correspondance entre les éléments de la notation et les phénomènes se fait par l'intermédiaire d'une fonction que nous appellerons adaptDLLayer. Cette fonction traduit les concepts de la notation en concepts PF conduisant à l'intégration dans le métamodèle de l'information contenue dans la description.

L'information contenue dans la notation ne sera pas nécessairement suffisante pour effectuer cette correspondance. Alors une information supplémentaire (G) devra être fournie.

Soit, la forme générale :

$adaptDLLayer : DLConcept, G \rightarrow PFLConcept$

avec $DLConcept$: ensemble des éléments de la notation (figurant dans la sémantique),
G : information complémentaire.

G est un type abstrait qui doit être défini lors de l'intégration de la notation dans l'outil. G définit l'information supplémentaire nécessaire pour qu'un diagramme de la notation puisse être exploité au niveau de la couche de Jackson.

Une autre optique aurait été d'ajouter à la notation toute l'information

nécessaire pour qu'elle puisse être directement traduite en terme de phénomènes de Jackson et de relations entre ces phénomènes. Dans ce cas, la notation doit contenir toute l'information pour que chacun de ses éléments puisse être directement transposable en phénomènes. La couche de liaison est alors limitée à une relation entre éléments de DL et phénomènes. G est vide.

Cette deuxième optique n'est jamais qu'un cas particulier de celle retenue, tout en étant beaucoup plus contraignante. De plus, il se pourrait que pour certaines notations, l'information ne puisse être intégrée dans la notation elle-même. Nous nous sommes dirigés vers une solution plus générique et plus souple.

La façon de procéder sera évaluée au cas par cas. Une partie de l'information peut être ajoutée simplement en imposant un formalisme particulier dans la notation.

Pour chaque élément de la notation, il faut déterminer les règles de transformation en phénomènes ainsi que le type de phénomènes (cf. chapitre 2). La couche de liaison est évidemment spécifique à la notation.

Seuls les éléments repris au niveau de la sémantique doivent être analysés.

Les éléments cibles de la correspondance sont des spécialisations des types de phénomènes choisis.

4.4 Cahier des charges

Pour permettre l'intégration d'une notation dans l'outil, un certain nombre de spécifications sont nécessaires. Outre le canevas de sémantique que le langage de description doit respecter, il faut, comme évoqué plus haut, faire le lien entre les concepts DL et PFL. Le souci principal est de pouvoir intégrer l'information extraite dans le métamodèle.

Les informations nécessaires sont :

1. Etablir la liste des phénomènes qui constitueront le scope de la description et donner leur type (voir typage des phénomènes). Les éléments du DL pourront être vus comme des spécialisations de type de phénomènes. Exemple : $StatechartEvent\ is-a\ PFLEvent$.

2. Organiser les phénomènes en ensembles ; chaque ensemble étant contrôlé ou référencé par une PFLTangibleDescription, contraint ou référencé par un PFLRequirement.

   Les relations suivantes doivent être spécifiées :

   - $PFLscope : PFLDescription \rightarrow PFLPhenomenaSet$
   - $PFLphens : PFLPhenomenaSet \rightarrow PFLPhenomenon$

- $PFLcontrols : PFLTangibleDescription \rightarrow PFLPhenomenaSet$
- $PFLreferences : PFLDescription \rightarrow PFLPhenomenaSet$
- $PFLconstraints : PFLIntangibleRequirement \rightarrow PFLPhenomenaSet$

4.5 Projection

La décomposition du problème principal en sous-problèmes est un élément essentiel de la méthode des problem frames. Cette décomposition est basée sur une relation de projection entre le problème principal et les différents sous-problèmes. La projection est définie en terme de projections entre parties du problème. Un IntangibleRequirement (IR) peut être une projection d'un autre IR. Une TangibleDescription (TD) peut être une projection d'une ou plusieurs autres TD. Une analyse du problème de la projection en terme de cohérence dans le métamodèle est effectué dans [MM,pp.16-20] et est complétée dans le chapitre 2 de ce document.

L'objectif de cette section est d'intégrer la fonction de projection dans le cadre théorique.

Nous définissons la relation de projection au niveau de la couche de Jackson. Nous considérons une description D définie par scope(D) et une description SD, projection de D, par scope(SD) et définissons la fonction de projection comme l'inclusion des scopes modulo renomination.

Soit $D : PFLDescription$ et $s \in \Sigma_D$ et $t \in T_D$

Considérons la sous-description associée à (projection de) D, soit $SD : PFLDescription$ et $s_p \in \Sigma_{SD}$ et $t_p \in T_{SD}$

On a :

$SD = projectionOf(D) \Leftrightarrow scope(SD)[x/y] \subseteq scope(D)$

où x correspond au phénomène y dans le problème principal.

La sémantique est exprimée en terme d'instantanés. Nous redéfinissons la fonction de projection par rapport aux instantanés. Prendre un sous-ensemble du scope de la description principale revient à réduire le domaine de la fonction snapshot de la même façon.

$SD = projectionOf(D) \Leftrightarrow (\forall s_p \in \Sigma_{SD}, \forall x \in dom(s_p), \exists s \in \Sigma_D : s_p[x/y] \subseteq s)$

où x correspond au phénomène y dans le problème principal.

La fonction de projection, notée $|_p$, pour les instantanés est définie par :

$|_p : \Sigma_D \rightarrow \Sigma_{SD}$

$\forall s \in \Sigma_D, \exists s_p \in \Sigma_{SD} : s|_p = s_p \Rightarrow s_p \subseteq s.$

De la même façon, pour les traces :

$|_p : \Sigma_D^+ \rightarrow \Sigma_{SD}^+$

tel que : $t|_p = t_p \Rightarrow \forall i, s^i \in t, s_p^i \in t_p : s^i|_p = s_p^i$

La projection est définie au niveau sémantique comme étant :

$Projection : ensemble\,de\,fonctions(t \rightarrow s) \rightarrow ensemble\,de\,fonctions(t_p \rightarrow s_p)$

La sémantique d'un langage de description pour une description SD, projection d'une description D est défini par :

Soit $|_p$ la fonction de projection ,

$[\![DL_{SD}]\!]_{Jackson}|_p = \{f_p \,|\, \exists f \in [\![DL_{SD}]\!]_{Jackson}, \forall s \in T_D, \forall s \in \Sigma_D : f_p(t|_p) = s|_p \wedge f(t) = s\}$

Le métamodèle fournit une idée précise des relations entre le problème principal et les sous-problèmes. Le modèle sémantique pour la projection détermine ce que doit décrire une description d'un composant d'un sous-problème par rapport à la description effectuée dans le problème principal.

L'approche sémantique de la projection est compatible avec les définitions et les contraintes de cohérence entre le problème et les sous-problèmes du métamodèle : toutes les deux se basent sur l'inclusion des scopes.

Chapitre 5

Exemple d'intégration d'une notation : les Statecharts

5.1 Introduction

Jackson utilise souvent la notation des statecharts dans ses exemples de description de domaine. Il nous a semblé approprié d'étudier cette notation comme exemple d'intégration.

Une première étape sera de proposer une sémantique s'intégrant dans le canevas de sémantique proposé. Ensuite, conformément au cahier des charges établi au chapitre précédent, nous spécifions toutes les informations nécessaires à l'intégration de la notation dans les PF.

5.1.1 Choix entre Harel et UML 1.5

Nous avons choisi d'intégrer les statecharts de Harel plutôt que ceux de UML version 1.5 en raison du caractère partiellement formel et des imprécisions au niveau de la sémantique UML. De plus, les sémantiques pour les statecharts de Harel sont plus nombreuses dans la littérature.

Sans être exhaustif, voici quelques points dans la définition des statecharts UML qui nous semblent difficiles à formaliser :

- Le statechart est intégré à un ensemble plus vaste : les éléments du statechart ont des types particuliers auxquels sont attachés des rôles, ils appartiennent à un package.
- Les événements sont de quatre types différents :
  - Les ChangeEvents : un événement survient lorsqu'une condition devient vraie.
  - Les SignalEvents : un signal émis par un autre objet et reçu par

l'objet que l'on décrit.
- Les CallEvents : la réception d'un appel d'opération.
- Les TimeEvents : un délai où le passage d'un temps donné.

Les SignalEvents et les CallEvents ne sont pas distingués syntaxiquement [UML1.5, p3-146]. Les ChangeEvents sont similaires aux conditions de garde, bien qu'il est stipulé qu'ils sont à distinguer. La raison invoquée est qu'une condition de garde n'est testée que lorsqu'il y a un événement qui survient, alors que le ChangeEvent est testé en permanence [UML1.5, p3-146]. Or, il est stipulé par ailleurs qu'une transition peut ne pas avoir d'événement lié [UML1.5, p2-153] et donc, une condition de garde sera évaluée en permanence.

- Les gardes et les actions peuvent avoir des attributs et des liens avec l'objet que l'on décrit dans les transitions [UML1.5, p3-143], mais cela n'est pas stipulé pour les transitions internes.
- L'ordre d'exécution des actions liées à une transition n'est pas stipulé, ni les effets d'une action sur l'action suivante (deux actions dont la première initialise des données utilisées par la deuxième).
- Les transitions conflictuelles sont choisies arbitrairement.

La description des statecharts de Harel est plus précise et va nous permettre de construire une sémantique de manière plus fidèle à l'esprit de Harel. A titre d'exemple, lorsque nous nous demanderons quelle est la durée de vie d'un événement et comment il faut distinguer les événements internes des événements externes, nous trouverons la réponse dans le texte de Harel [HarelNaamad, p325 et p326]. Lorsque nous nous demanderons dans quels cas une transition peut ne pas avoir d'événements liés mais seulement une condition (une garde dans la terminologie UML 1.5), nous trouverons la réponse dans [HarelPoliti, p56]. Lorsque nous nous demanderons comment doit être exécutée une séquence d'action, nous trouverons une réponse précise [HarelNaamad, p301]. Autant d'éléments pour lesquels nous devons choisir arbitrairement une interprétation avec les statecharts UML version 1.5.

5.2 Une sémantique fonctionnelle pour les statecharts

Nous définissons une sémantique formelle des statecharts basée sur la sémantique de STATEMATE [HarelNaamad].

Un certain nombre de choix ont été faits en définissant la sémantique conduisant à considérer un sous-ensemble des statecharts. Nous les décrivons brièvement ci-dessous.

- Nous choisissons le mode asynchrone de communication décrit par Ha-

rel : le système démarre le traitement de ses inputs aussitôt qu'il les reçoit.

- La priorité sur les transitions (ordre) est celui de Statemate : une transition de plus haut niveau a priorité sur une transition de plus bas niveau (contrairement aux state diagrams UML).
- Nous ne permettons pas de déclencheurs composés, d'événements négatifs (non-événements). Les actions se limitent à la génération d'événements internes et externes et à la mise à jour de variables.
- Il n'y a pas de synchronisme dans les états AND. Autrement dit, une transition peut être prise dans un des deux sous-états de l'état AND sans qu'il y ait obligatoirement une transition dans l'autre sous-état.

Le modèle de temps asynchrone suppose que le système réagit lorsque se déroule un changement externe. Il permet que plusieurs changements se passent simultanément et surtout que plusieurs micro-étapes répondant à ces changements se déroulent en un point donné du temps jusqu'à ce que le système arrive à un état stable, c'est-à-dire que plus aucune transition n'est possible sans un nouveau changement externe. Ce n'est qu'à ce moment, que le système est prêt à réagir à d'autres changements externes. Une telle collection de micro-étapes est appelée une macro-étape.

Intuitivement une macro-étape consiste en une chaîne de réactions (micro-étapes) du système à un stimuli extérieur. Au niveau de la micro-étape, un ensemble de transitions non conflictuelles sont sélectionnées en fonction des événements générés dans les micro-étapes précédentes et d'une évaluation de variables et de conditions, conduisant à une nouvelle évaluation et un nouvel ensemble d'événements générés.

Du point de vue de l'environnement, l'exécution proprement dite d'une macro-étape ne prend pas de temps. C'est comme si le temps s'arrêtait pendant la durée de l'exécution.

L'exécution répétée de micro-étapes jusqu'à ce que le système soit dans un état stable, n'incrémente pas l'horloge. Elle constitue la macro-étape, à savoir, la série de micro-étapes qui est exécutée en un point du temps sans que des changements externes ne se déroulent entre les micro-étapes. Les événements qui sont produits par les micro-étapes successives peuvent s'accumuler comme signaux de sortie.

Notre sémantique fait la distinction entre les événements générés par l'environnement et les événements générés localement. Les événements externes sont consultés à la première micro-étape. Les événements internes sont consommés dans la micro-étape qui suit la micro-étape où ils ont été générés. Les événements externes générés par le système ne sont pas consommés par les micro-étapes suivantes. Ils sont communiqués à l'environnement à la fin de la macro-étape.

Nous appellerons dorénavant événements input, les événements externes générés par l'environnement, événements output, les événements générés à destination de l'environnement (communiqués à l'environnement à la fin de la macro-étape) et événements internes, les événements générés localement et utilisés dans les différentes micro-étapes. Pour les variables, nous parlerons de variables locales si elles sont internes et de variables partagées si elles sont externes.

Dans les paragraphes suivants, nous fournissons un ensemble de définitions abstraites de la syntaxe pour le sous-ensemble des statecharts choisis. Ces définitions seront utilisées lors de la définition de la sémantique. Ensuite nous traitons de la construction et de l'exécution d'une micro-étape et finalement nous l'étendons à la macro-étape. La plupart des définitions sont empruntées de [EW2001] et [DJHP97].

5.2.1 Syntaxe des statecharts

Dans cette partie, nous décrivons la syntaxe abstraite pour le sous-ensemble choisi des statecharts. Nous décrivons premièrement les concepts des statecharts basés sur les hiérarchies d'états. Nous introduisons ensuite les concepts de données incluant les événements et nous terminons par les concepts d'exécution.

Concepts basés sur les hiérarchies d'états

La modélisation des statecharts basée sur les hiérarchies d'états se base sur la définition d'un ensemble structuré hiérarchiquement d'états, incluant une relation fils entre certains états. Les états peuvent être des états de type BASIC, OR ou AND. Si un état n'a pas de fils, il est de type BASIC sinon il est soit de type OR ou AND. Quand le système est dans un état OR, cela signifie qu'il est dans exactement un de ses fils. Quand il est dans un état AND, cela signifie qu'il est dans tous ses fils en même temps. Nous notons l'ensemble des états d'un statechart SC par *states(SC)*. A cet ensemble, nous associons une fonction *child*

$child : states(SC) \rightarrow 2^{states(SC)}$,

une fonction *type*

$type : states(SC) \rightarrow \{BASIC, OR, AND\}$,

et une fonction désignant pour chaque état OR un de ses fils comme le fils par *défaut* qui est atteint quand l'état OR est atteint :

Soit s, s' $\in states(SC)$, on a :

$default : \{ s \mid type(s) = OR\} \rightarrow states(SC)$,

$\forall s : type(s) = OR \Leftrightarrow (\exists! s' : default(s) = s' \wedge s' \in child(s))$.

Pour être bien défini, un tel ensemble hiérarchisé d'état, *states(SC)*, doit être un arbre fini avec un état racine particulier appelé *root* [1].

La fonction *child* fournit l'ensemble des successeurs d'un noeud de l'arbre.

Si un état *s'* est un sous-état de s ($s' \in child(s)$), l'état s est aussi appelé l'état père de *s'*, noté *father(s')*.

$father : states(SC) \rightarrow states(SC)$,

défini par

$\forall s, s' \in states(SC) : s' \in child(s) \Leftrightarrow \exists 1! s : father(s') = s$.

La profondeur d'un état est définie par

$depth(s) := 0$ si $s = root$

$depth(father(s) + 1)$ sinon,

et la profondeur d'un statechart SC est donné par

$depth(SC) = \max\{depth(s) | s \in states(SC)\}$.

La fonction *child* définit un ordre partiel sur les états :

$s \preccurlyeq s'$

$s' \in child(s) \Rightarrow s \preccurlyeq s'$

$s \preccurlyeq s' \wedge s' \preccurlyeq s'' \Rightarrow s \preccurlyeq s''$

$s \prec s'$ ssi $s \neq s' \wedge s \preccurlyeq s'$.

On dit que si $s \preccurlyeq s'$, s est plus haut que s' dans la hiérarchie des états.

Le comportement dynamique d'un statechart est assuré par un ensemble de transitions et de réactions statiques. Nous ne considérons pas les réactions statiques dans le cadre de ce travail.

Transitions

Nous associons à un statechart SC un ensemble de noms de transitions *trans(SC)*. Nous définissons un ensemble de fonctions associées à une transition t :

$source : trans(SC) \rightarrow 2^{states(SC)}$

définit pour t un ensemble non-vide d'états source,

$target : trans(SC) \rightarrow 2^{states(SC)}$

définit pour t un ensemble non-vide d'états cible.

Une transition est une relation d'un ensemble d'états (source) vers un autre ensemble d'états (cible).

[1] le *root* doit être un état OR.

D'autres fonctions seront utilisées pour définir les conditions d'un statechart bien formé et d'exécutions de transitions. Nous les définissons plus loin. Pour les mêmes raisons, nous introduisons le concept de configuration décrivant le sous-ensemble maximal d'états pouvant être actifs en même temps.

Le *least common ancestor lca(S)* d'un sous-ensemble d'états S définit l'état le plus rapproché dans la hiérarchie qui incorpore tous les états de S. Autrement dit, tous les états qui englobent S sont plus haut ou au même niveau dans la hiérarchie des états que le *lca(S)*. Comme le *root* est un *ancestor* pour chaque état, *lca(S)* existe pour chaque sous-ensemble S d'états. Il est défini par

$lca(S) \preceq S$ (lca(S) est un *ancestor* pour chaque état de S)

$\forall \hat{s} \in states(SC) : \hat{s} \preceq S \Rightarrow \hat{s} \preceq lca(S)$ (lca(S) minimal) .

Le *least common or-ancestor* $lca^+(S)$ d'un sous-ensemble d'états S définit l'état OR le plus rapproché dans la hiérarchie des états qui contient tous les états de S et qui n'est pas contenu dans S. Comme le *root* doit être un état OR, le lca^+ existe pour chaque sous-ensemble d'états qui ne contient pas le *root*. Si le *root* est contenu dans S, il sera aussi le lca^+ par définition :

$$lca^+(S) := root \text{ si } root \in S$$
$$s \quad \text{si } root \notin S \ \wedge s \prec S \wedge type(s) = OR$$
$$\wedge (\forall \hat{s} \in states(SC) : type(\hat{s}) = OR \wedge \hat{s} \prec S \Rightarrow \hat{s} \preceq s)$$

Deux états s, s' sont ancestralement relié, si $s \preceq s' \vee s' \preceq s$.

Deux états s, s' sont orthogonaux, noté $s \perp s'$, si s et s' ne sont pas ancestralement relié.

$s \perp s' \Leftrightarrow s \not\preceq s' \wedge s' \not\preceq s \wedge type(lca(\{s, s'\}) = AND$.

La relation d'orthogonalité décrit les pairs d'états qui peuvent être simultanément actifs.

Un sous-ensemble $S \subseteq states(SC)$ est consistent , noté $\downarrow (S)$ si , pour chaque paire s, s' de S, soit s et s' sont reliés ancestralement, soit s et s' sont orthogonaux.

Une configuration est un ensemble consistent maximal d'états.

Soit $config(SC) \subseteq 2^{states(SC)}$ l'ensemble de toutes les configurations de SC.

$$c \in config(SC) \Leftrightarrow root \in c$$
$$\wedge (\forall s \in c : \ (type(s) = OR \ \wedge \ (\exists! s' \in c : s' = child(s)))$$
$$\vee (type(s) = AND \ \wedge \ (\forall s' = child(s) : s' \in c)))$$

Le scope d'une transition, noté *scope(t)*, exprime l'idée d'aire d'influence. Il est l'état de type OR le plus bas dans la hiérarchie qui est un ancestor pour tous les états source et cible d'une transition. Il est l'état le plus bas dans lequel le système reste sans sortir ni rentrer quand la transition est prise.

$scope(t) := lca^{+}(source(t) \cup target(t))$

Pour un ensemble consistent $S \subseteq states(SC)$, l'ensemble complet par défaut, noté *dcompl(S)*, est le plus petit ensemble C tel que :

- $S \subseteq C$
- $s \in C \ \wedge \ s \neq root \Rightarrow father(s) \in C$
- $s \in C \wedge type(s) = OR \wedge child(s) \cap S = \emptyset \Rightarrow default(s) \in C$
- $s \in C \wedge type(s) = AND \wedge child(s) \subseteq C$

Un ensemble de transitions T est consistent, noté $\downarrow T$, si chaque paire de transitions $t_1, t_2 \in T$ est consistente, noté $\downarrow$*(t1,t2)*. Deux transitions sont consistentes si elles sont actives dans 2 régions orthogonales, autrement dit, si leurs scopes sont orthogonaux

$\downarrow (t_1, t_2) \Leftrightarrow \text{scope}(t_1) \perp \text{scope}(t_2)$.

Si un ensemble de transition exécutable n'est pas consistent, nous utilisons une relation de priorité pour sélectionner un sous-ensemble consistent. La priorité d'une transition est donnée par la distance de son scope par rapport au *root*.

$prio : T \rightarrow N$

$t \mapsto depth(SC) - depth(scope(t))$.

Statechart bien formé.

Un statechart SC est bien formé, noté *wff(SC)*, si, pour toutes les transitions t, on a :

- $\downarrow source(t) \wedge \downarrow target(t)$
- $\forall s \in source(t) : type(s) = OR \Rightarrow \forall s' : s \prec s' \Rightarrow s' \notin source(t)$
- $\forall s \in target(t) : type(s) = OR \Rightarrow \forall s' : s \prec s' \Rightarrow s' \notin target(t)$
- $root \notin source(t) \cup target(t)$

Concepts de données

L'espace de données associé à un statechart peut contenir des événements et des variables. Dans un espace donné, nous faisons la distinction entre ce qui est contrôlé par l'environnement, ce qui est local et ce qui est observable par l'environnement.

Si on considère l'ensemble *Events* de tous les événements, on a :

$Events = I \uplus O_{LOC} \uplus O_{OUT}$ avec I : événements générés par l'environnement

O_{LOC} : événements locaux

O_{OUT} : événements générés à destination de l'environnement.

De la même façon, pour l'ensemble *Var* des variables, nous distinguons :

$Var = var_{IN} \uplus var_{LOC} \uplus var_{OUT}$

avec :

- var_{IN} : variables contrôlées par l'environnement,
- var_{LOC} : variables locales,
- var_{OUT} : variables observables par l'environnement.

Les événements et variables associés à une statechart SC sont respectivement notés *events(SC)* et *var(SC)*.

Concepts d'exécution.

En plus des concepts élémentaires de transition évoqués plus haut, les transitions sont aussi étiquetées avec des *events* qui déclenchent leur exécution, avec des *guards* pour contrôler leur exécution et des *actions* à effectuer lors de l'exécution de la transition.

Guards

Une transition peut seulement être exécutée si son *guard* est évalué à *true*. Un *guard* est simplement une expression booléenne sur les variables. Il ne peut contenir d'occurrence des variables out (elles peuvent seulement être mises à jour mais jamais testées). L'ensemble des expressions typées sur *var(SC)* est noté *Exp(var(SC))*. Nous utilisons Exp^{TYPE}pour désigner des expressions de type *type*. Nous utilisons Bexp pour Exp^{bool}(les expressions booléennes).

L'ensemble des *guards* sur les variables *var(SC)* et les états *states(SC)*, noté $Guards(var(SC), states(SC))$, est défini par

exp où $exp \in Bexp(var(SC) \cup in(states(SC)))$

Actions

Si une transition est exécutée, son expression *action* provoque la génération d'événements et/ou la modification de certaines variables.

Les assignations sont fournies comme actions de base pour mettre à jour des variables.

Les actions de base peuvent être combinées par l'opérateur ; .

L'ensemble des actions sur les événements *events(SC)* et les variables $var(SC)$, noté *Actions(events(SC),var(SC))*, est défini par

e	où $e \in events(SC) \setminus I$,
v := exp	où exp $\in Exp(var(SC))^{type(v)} \wedge v \in var(SC) \setminus var_{IN}$,
a ; a'	où $a, a' \in Actions$.

L'ensemble des événements générés par une action est noté *genEvent(a)*.

Etiquettes de transition

L'ensemble des noms de transition *trans(SC)* est associé avec 3 fonctions supplémentaires définissant les *events, guards* et *actions* pour chaque transition :

- $event : trans(SC) \rightarrow events(SC)$
- $guard : trans(SC) \rightarrow Guards(SC)$
  où $Guards(SC) = Guards(var(SC), states(SC))$,
- $action : trans(SC) \rightarrow Actions(SC)$
  où $Actions(SC) = Actions(events(SC), var(SC))$.

Les statecharts permettent aussi d'associer un comportement aux états en utilisant le concept de *static reactions*. Ces réactions se retrouvent le plus souvent comme un ensemble de paires *guard/action*. Nous modélisons cela comme :

$$sr : states(SC) \rightarrow 2^{Guards(SC) \times Actions(SC)} .$$

Toutefois nous ne considérons pas les réactions statiques dans les paragraphes suivants pour ne pas alourdir les différentes formules ; l'impact se limitant à des actions supplémentaires à prendre en compte dans le calcul d'une micro-étape.

5.2.2 Définition d'une micro-étape

Une micro-étape, partant d'une configuration donnée, d'une valorisation de variables et d'événements, exécute un ensemble maximal de transitions disponibles non conflictuelles et aboutit à une nouvelle configuration, une nouvelle valorisation de variables et de nouveaux événements.

Valuation

Nous définissons une *valuation* σ comme la représentation d'un état du système en terme de configuration et de variables. La *valuation* assigne des valeurs aux variables du système et détermine quels sont les états actifs du statechart. Ceux-ci doivent former une configuration.

Pour éviter toute confusion avec l'état d'un statechart, nous appelerons dorénavant l'état d'un système décrit par un statechart une *valuation.*

Nous définissons 2 sous-fonctions σ_{states} et σ_{var} qui assignent respectivement un booléen aux états et une valeur à chaque variable.

Considérons un ensemble typé de variables $\mathcal{V} \in 2^{Var}$, un domaine typé de données (valeurs possibles) $\mathcal{D}$ [2] et un ensemble d'états $\mathcal{S} \in 2^{States}$:

- σ_{var} est une fonction totale, préservant le type, assignant à chaque variable de $\mathcal{V}$ une valeur de $\mathcal{D}$.
  $\sigma_{var} : \mathcal{V} \rightarrow \mathcal{D}$
  avec $\forall v \in \mathcal{V} : \sigma(v) = x \wedge x \in \mathcal{D}^{type(X)}$.
- σ_{states} est une fonction totale qui assigne à chaque état de $\mathcal{S}$ un booléen pour déterminer si l'état est actif ($true$) ou non ($false$).
  $\sigma_{states} : \mathcal{S} \rightarrow \mathcal{D}^{boolean}$
  avec $\forall s \in \mathcal{S} : \sigma(s) = b \wedge b \in \mathcal{D}^{boolean}$.

La valuation correspond à l'union des 2 sous-fonctions :

$\sigma : \mathcal{V} \cup \mathcal{S} \rightarrow \mathcal{D}$

et $\sigma = \sigma_{states} \cup \sigma_{var}$.

Pour un statechart donné SC, $\mathcal{V} = var(SC)$ et $\mathcal{S} = states(SC)$.

L'ensemble de toutes les valuations d'un statechart SC est noté Σ_{SC}.

Nous utilisons Σ en place de Σ_{SC}, pour représenter l'ensemble des valuations, quand le contexte est évident.

Les 2 sous-fonctions σ_{states} et σ_{var} déterminent 2 ensembles :

- $conf(\sigma)$ l'ensemble des états actifs de SC,
  $conf(\sigma) = \{s \in states(SC) \mid \sigma_{states}(s) = true\}$
- $var(\sigma)$ l'ensemble des variables manipulées par le statechart qui correspond au domaine de σ_{var}.
  $var(\sigma) = dom(\sigma_{var})$

L'ensemble des états actifs d'une valuation doit correspondre à une configuration. On a :

$\sigma \in \Sigma(SC) \Rightarrow conf(\sigma) \in conf(SC)$

Valuation initiale

La valuation initiale, notée σ_0, est une valuation qui satisfait aux conditions suivantes :

- $conf(\sigma) = dcompl(root)$: la configuration par défaut.
- Nous définissons une valeur par défaut, notée $d_{init}^{type} \in \mathcal{D}^{type}$, pour chaque type de variable. A chaque variable x de $var_{LOC} \cup var_{OUT}$, nous affectons la valeur $d_{init}^{type(x)}$. La valeur des variables var_{IN} est fournie par l'environement.

[2] Nous notons $\mathcal{D}^X$ le sous-ensemble des valeurs de type X.

Sémantique des expressions guards et actions

Brièvement, nous décrivons comment les guards et les actions sont évalués.

Si nous considérons une valuation σ et une action a, $[\![a]\!]_\sigma$est la valuation résultant de l'évaluation de l'expression a. Par exemple,

$[\![action; actionsequence]\!]_\sigma = [\![actionsequence]\!]_\sigma \circ [\![action]\!]_\sigma$

$[\![x := \exp]\!]_\sigma = \sigma\,[x/\,[\![\exp]\!]_\sigma]$.

On a également :

$[\![in(s)]\!]_\sigma = s \in conf(\sigma)$

Sélection des transitions et leur effet sur les configurations

Une transition t est permise pour σ et les événements inputs $\mathcal{I}$[3], noté $enabled(\sigma, \mathcal{I}, t)$ ssi toutes ses sources sont contenues dans la configuration courante, les événements déclencheurs en input et ses guards évalués à *true* :

$enabled(\sigma, \mathcal{I}, t) \Leftrightarrow source(t) \subseteq conf(\sigma) \wedge event(t) \subseteq \mathcal{I} \wedge [\![guard(t)]\!]_\sigma = true.$

L'algorithme de calcul des transitions pouvant être exécutées lors d'une micro-étape utilise un mode top-down. Prenant en compte la priorité, on peut stopper l'algorithme dans une "branche" quand une transition est trouvée dans cette branche, comme les transitions de plus bas niveau ont une priorité plus basse. Soit $En(\sigma, \mathcal{I})$, l'ensemble de tous les sous-ensembles consistents maximum de transitions possibles. L'algorithme part du *root* et parcourt la hiérarchie suivant la configuration $conf(\sigma)$. Nous définissons par récurrence un ensemble d'ensembles de transitions relatif à un état donné de la configuration $s \in conf(\sigma)$. On calcule $en(\sigma, \mathcal{I}, s) \in 2^{2^T}$ satisfaisant

$st \in en(\sigma, \mathcal{I}, s) \Rightarrow \downarrow st$

par induction sur la profondeur de s.

1. $type(s) = BASIC : en(\sigma, \mathcal{I}, s) = \emptyset$
2. $type(s) = AND : en(\sigma, \mathcal{I}, s) = \{st_1 \cup \ldots. \cup st_n | (st_i \in en(\sigma, \mathcal{I}, s_i)) \vee (en(\sigma, \mathcal{I}, s_i) = \emptyset \wedge st_i = \emptyset)\}$ avec $\{s_1, \ldots., s_n\} = child(s)$
3. $type(s) = OR :$ soit $T = \{t_1, \ldots, t_k\} = \{t | scope(t) = s \wedge enabled(\sigma, \mathcal{I}, t)\}$
   (a) $T = \emptyset$:
   soit s' unique fils de s dans conf (σ) :
   $en(\sigma, \mathcal{I}, s) = en(\sigma, \mathcal{I}, s')$

[3] $\mathcal{I}$ représente les événements inputs d'une micro-étape mais ne correspond pas aux événements générés par l'environnement, noté I. $\mathcal{I}$ peut contenir aussi bien des événements externes qu'internes. La même remarque est d'application pour $\mathcal{O}$.

(b) $T \neq \emptyset$:

toutes les transitions en T ont la même priorité et sont donc en conflit avec chacune des autres.

$en(\sigma, \mathcal{I}, s) = \{\{t_1\}, \ldots, \{t_k\}\}$

Les transitions prises à partir d'une configuration donnée sont alors calculées par :

$En(\sigma, \mathcal{I}) = en(\sigma, \mathcal{I}, root)$.

Pour un ensemble de transitions donné $st \in En(\sigma, \mathcal{I})$, nous calculons les ensembles $exited(st)$, $entered(st)$ et $active(st)$ par :

$exited(st) = \{s \mid \exists t \in st : scope(t) \prec s\} \cap conf(\sigma)$,

$active(st) = conf(\sigma) \setminus exited(st)$,

$entered(st) = dcompl\,(\, active(st) \cup \bigcup_{t \in st} target(t)) \setminus active(st)$.

Plus généralement :

$exited(\sigma, \mathcal{I}) = \bigcup_{st \in En(\sigma, \mathcal{I})} exited(st)$

$entered(\sigma, \mathcal{I}) = \bigcup_{st \in En(\sigma, \mathcal{I})} entered(st)$.

Définition

Nous représentons la transition du système de σ à σ' en réaction à un input $\mathcal{I}$ et en produisant un output $\mathcal{O}$ par :

$\sigma^{\mathcal{I}} \longrightarrow^{\mathcal{O}} \sigma'$ avec :

- $conf(\sigma') = conf(\sigma) \cup entered(\sigma, \mathcal{I}) \setminus exited(\sigma, \mathcal{I})$
- $\mathcal{O} = \bigcup_{st \in En(\sigma, \mathcal{I})} genEvent(action(st))$

Les événements générés peuvent être locaux ($\mathcal{O}_{LOC}$) ou destinés à l'environnement ($\mathcal{O}_{OUT}$). On aura :

$\mathcal{O} = \mathcal{O}_{LOC} \uplus \mathcal{O}_{OUT}$.

Les transitions contenues dans les différents st peuvent assigner des valeurs aux variables partagées conflictuellement entre elles. Nous choisissons de manière non déterministe une sérialisation de toutes les actions des transitions dans st.

5.2.3 Exécution d'une macro-étape

Une macro-étape est une chaîne de micro-étapes partant d'un état du système stable et aboutissant à un état stable. Elle se termine lorsqu'il n'y a plus de transitions possibles. La chaîne peut être infinie.

Les événements externes (I) sont consultés à la première micro-étape. Les

événements internes générés lors d'une micro-étape ($O_{LOC_{i+1}}$) sont consommés dans la micro-étape suivante ($i+1$). Les événements externes générés lors d'une micro-étape ($O_{OUT_{i+1}}$) ne sont pas consommés par la micro-étape suivante mais sont communiqués à l'environnement à la fin de la macro-étape.

Considérons la macro-étape partant de la valuation σ et d'un ensemble d'événements inputs I, aboutissant à une valuation σ' et générant un ensemble d'événements O à destination de l'environnement, noté :

$$\sigma \overset{I}{\overset{O}{\Longrightarrow}} \sigma'$$

La macro-étape s'obtient en exécutant les n micro-étapes différentes comme suit :

$$\sigma \overset{I}{\overset{O}{\Longrightarrow}} \sigma' \Leftrightarrow \sigma \overset{I}{\longrightarrow}^{O_1} \sigma_1 \overset{I_1}{\longrightarrow}^{O_2} \sigma_2 \ldots \sigma_i \overset{I_i}{\longrightarrow}^{O_{i+1}} \sigma_{i+1} \ldots \sigma_{n-1} \overset{I_{n-1}}{\longrightarrow}^{O} \sigma'$$

La i^{me} micro-étape, notée $\sigma_i \overset{I_i}{\longrightarrow}^{O_{i+1}} \sigma_{i+1}$, est calculée par :

$$I_i = O_{LOC_i} \qquad \forall i : 0 < i \leq n \ (I_0 = I)$$
$$O_{i+1} = I_{i+1} \uplus O_{OUT_i} \quad \forall i : 0 \leq i \leq n$$

La macro-étape se termine quand le système est stable, autrement dit, quand il n'y a plus de transition pouvant être exécutée.

$stable(\sigma_n, I_n) \Leftrightarrow En(\sigma_n, I_n) = \emptyset$.

Les événements externes générés par la macro-étape sont obtenus par :

$O = \bigcup_{0<i\leq n} O_{OUT_i}$

5.2.4 Sémantique

On définit la sémantique d'un statechart comme étant la réaction d'un système à un stimuli externe ; la réaction est définie en terme d'événements communiqués à l'environnement, de changements de valeurs de variables et d'états du système.

La sémantique est définie par un ensemble de fonctions de réaction d'un système dans un état σ à des événements externes (i) conduisant à un état du système σ' et générant des événements à destination de l'environnement (o) ; chaque état étant le résultat d'un ensemble de réaction (un historique).

Un historique est la suite des macro-étapes qui ont été exécutées pour atteindre une valuation σ en partant de la valuation initiale, notée σ_0. Soit $\mathcal{H}$, l'ensemble des historiques pour un statechart.

Nous définissons un historique, noté h,par :

$$h = (\sigma_0 \overset{i_0}{\Longrightarrow}^{o_1} \sigma_1 \overset{i_1}{\Longrightarrow}^{o_2} \sigma_2 \ldots \sigma_{n-1} \overset{i_{n-1}}{\Longrightarrow}^{o_n} \sigma)$$

La sémantique est définie par :

$[\![.]\!] : SC \rightarrow 2^{(\mathcal{H} \times 2^I \rightarrow \Sigma \times 2^{O_{OUT}})}$

$[\![SC]\!] = \{f \mid \forall h, \sigma', i, o : f(h,i) = (\sigma', o) \wedge (\sigma \overset{i}{\Longrightarrow}^{o} \sigma')\}$

5.3 Couche de liaison pour les Statecharts.

Le but de la couche de liaison est de faire correspondre les concepts au niveau de la couche Jackson et de la couche statechart, pour donner les règles permettant d'extraire l'information contenue dans un statechart au niveau de la couche de Jackson.

Les différents concepts utilisés dans la sémantique sont :

- la valuation : exprimée en termes de **valeurs** attribuées aux **variables** et en terme de configurations qui représentent un ensemble d'**états** (statechart) dans lesquels la description se trouve ;
- les **événements**.

Pour un statechart donné SC, les éléments qui permettent d'extraire les phénomènes seront donc les suivants :

- $var(SC) \rightarrow \mathcal{D}^{type(var(SC))}$,
- $states(SC) \rightarrow Boolean$,
- $events(SC)$.

Pour chacuns des ces 3 éléments, nous étudions les modalités de conversion en phénomènes.

5.3.1 Evénements

La traduction des événements en concepts PFL est évidente.

Nous pouvons spécialiser le type PFLEvent avec le type StatechartEvent.

On a : adaptDLLayer(StatechartEvent) = PFLEvent.

Pour un statechart donné SC, nous obtenons très facilement la liste des événements : events(SC). Nous ne nous intéressons qu'à une partie des événements : les événements générés par l'environnement (I) et les événements générés à destination de l'environnement (O_{OUT}). Les événements locaux sont émis et consommés lors des micro-étapes. Ils ne figurent pas dans l'input, ni l'ouput lié à une macro-étape.

Nous avons cependant besoin de plus d'information au niveau de la couche de Jackson. Il faut, en effet, déterminer quel domaine contrôle quel événement. Les événements output sont contrôlés par la description. Les événements inputs sont contrôlés par l'environnement.

Deux possibilités pour spécifier le contrôle des événements Inputs :

- utiliser un formalisme particulier en préfixant le nom de l'événement avec le nom de la description qui le contrôle ;
- demander à l'utilisateur de fournir cette information. Nous utilisons un sous-type de G :
  $G_{InputEventControls} : PFLTangibleDescription \rightarrow StatechartEvent$

Le même raisonnement peut être appliqué pour déterminer vers qui sont destinés les événements (messages) contrôlés par la description.

$G_{OutputEventReferences} : PFLTangibleDescription \rightarrow StatechartEvents$

Pour le requirement, nous aurons :

$G_{OutputEventReferences} : PFLIntangibleRequirement \rightarrow StatechartEvents$

et

$G_{OutputEventConstraints} : PFLIntangibleRequirement \rightarrow StatechartEvents.$

5.3.2 Etats

Chaque valuation contient l'ensemble des états du statechart (states(SC)) auxquels on associe un booléen permettant de déterminer si l'état fait partie de la configuration ou non. Nous pouvons donc extraire très facilement la liste des états de la description.

Nous pouvons spécialiser le type PFLState avec le type StatechartEvent.

On a : adaptDLLayer(StatechartState) = PFLState.

Jackson [2,p.80] définit une phénomène de type State comme étant une relation entre une entité et une valeur. Dans ce cas-ci, l'entité est la description et la valeur le nom de l'état du statechart.

Nous devons déterminer ensuite quels seront les états observables par l'environnement ainsi que déterminer la description référençant ses états ; s'agissant du requirement, la relation pourra être aussi la relation *constraints*.

Cela se fait par l'intermédiaire des fonctions suivantes (information fournie par l'analyste) :

- $G_{StateReferences} : PFLDescription \rightarrow StatechartState$
- $G_{StateConstraints} : PFLIntangibleRequirement \rightarrow StatechartState$

Les états observables, notés $states(SC)_e$, sont les états référencés par $G_{StateReferences}$ ou 'contraints' par $G_{StateConstraints}$:

$states(SC)_e = \{s \mid s \in states(SC) \wedge (\exists D : PFLDescription, \exists R : PFLIntangibleRequirement$ tel que $G_{StateReferences}(D) = s \vee G_{StateConstraints}(R) = s)\}$

5.3.3 Variables et valeurs de variables

La représentation d'une variable en tant que phénomènes n'est pas explicitement définie par Jackson. Dès lors, nous prenons la convention de définir la conversion d'une variable x en phénomènes par :

$\forall x \in var(SC), \forall y \in \mathcal{D}^{type(x)} : (x = y)\ instanceOf(PFLState).$

Pour chaque variable, nous reprenons l'ensemble des valeurs possibles et nous considérons chaque attribution d'une valeur à une variable comme étant un phénomène, plus précisément une instanciation de PFLState ; l'entité étant la variable et la valeur (PFLValue) une valeur possible.

Les variables peuvent être rangées en 2 catégories : locales et globales. Nous ne nous intéressons qu'aux variables globales que l'environnement pourra 'observer' et que le requirement pourra 'contraindre' ou 'observer'. L'information de visibilité des variables devra être disponible dans les propriétés de la variable. A la définition d'une variable, 2 propriétés seront obligatoires : le type et le 'scope' (local/global). Nous préférons cette solution à une information supplémentaire (G) à donner par l'analyste car, lors de la déclaration d'une variable dans un outil CASE, il semble logique et minimal de devoir déterminer ces attributs.

Nous notons $var(SC)_e$ l'ensemble des variables globales d'un statechart SC $(var(SC) \cap (Var_{IN} \cup Var_{OUT}))$ et $stateVar(SC)_e$ l'ensemble des phénomènes assignant une valeur à une variable.

$stateVar(SC)_e : SC \rightarrow Var \times \mathcal{D}$

défini par :

$stateVar(SC)_e = \{(var, val) |\ var \in var(SC)_e \wedge val \in \mathcal{D}^{type(val)}\}.$

Les variables input $(var(SC)_e \cap Var_{IN})$ sont référencées par la description et sont contrôlées par l'environnement. Cette dernière information est donnée par :

$G_{VarControls} : PFLTangibleDescription \rightarrow PFLState$

Les variables ouptut $(var(SC)_e \cap Var_{OUT})$, inversement, sont référencées par l'environnement et/ou le requirement et sont contrôlées par la description. Donné par :

$G_{VarReferences} : PFLDescription \rightarrow PFLState$

Les variables peuvent être contraintes par le requirement :

$G_{VarConstraints} : PFLIntangibleRequirement \rightarrow PFLState$

5.4 Cahier des charges

Comme spécifié au point [4.5], les informations suivantes sont nécessaires :

- la liste des phénomènes et leur type,
- l'organisation des phénomènes en ensemble et l'information de contrôle (ou de contrainte) sur ces ensembles.

5.4.1 Liste des phénomènes

Si nous considérons la description D modélisée par un statechart SC, l'ensemble des phénomènes qui vont constituer le scope de la description est déterminé par l'union des différents éléments suivants :

- Les événements de $events(SC)$ limités à I et O_{OUT}. Ces phénomènes sont de type Event.
- Les états de $states(SC)$ limités aux états observables (information fournie par $G_{StateReferences}$ et $G_{StateConstraints}$) : $states(SC)_e$. Ces phénomènes sont de type CausalState. L'entité correspondante (voir 2.2.2) (phénomène de type Entity) est la description et les valeurs associées (phénomène de type Value) sont les noms des états dans le statechart.
- Les variables de $var(SC)$ limitées aux variables partagées (scope de la variable = global) : $var(SC)_e$. Pour chaque variable concernée, nous définissons un phénomène de type CausalState par valeur possible (infinité de valeurs) : $stateVar(SC)_e$. Un phénomène de type Entity correspondant au nom de la variable est associé à un phénomène de type Value correspondant à une valeur possible de la variable [4] .

5.4.2 Ensembles de phénomènes

Les phénomènes doivent être regroupés en ensembles de phénomènes.

Chaque ensemble est soit contrôlé par la description, soit référencé par la description, soit référencé par une description externe, soit contraint par le requirement. Les phénomènes sont regroupés indépendamment de leur type. Chaque phénomène est associé à une (ou plusieurs) description(s) et le regroupement est effectué à partir de cette information. L'information est fournie :

- pour les événements :

[4]Cette solution est théorique et ne peut être implémentée dans un outil. Une solution pourrait être d'importer tous les phénomènes Entity correspondant aux variables et d'importer les valeurs uniquement quand elles sont explicitement spécifiées ; par exemple x = 1 dans un 'guard'.

- par $G_{OutputEventReferences}$: les phénomènes sont contrôlés par la description et sont référencés par une description externe,
- par $G_{OutputEventConstraints}$: les phénomènes sont contrôlés par la description et sont contraints par le requirement,
- par $G_{InputEventControls}$: les phénomènes sont contrôlés par une description externe et sont référencés par la description ;

- pour les états, par $G_{StateReferences}$: les phénomènes sont contrôlés par la description et sont référencés par une description externe ou contranits par le requirement ($G_{StateConstraints}$) ;
- pour les variables :
  - par $G_{VarReferences}$: les phénomènes sont contrôlés par la description et sont référencés par une description externe et/ou le requirement,
  - par $G_{VarControls}$: les phénomènes sont contrôlés par une description externe et sont référencés par la description,
  - par $G_{VarConstraints}$: les phénomènes sont contraints par le requirement.

Les informations extraites vont permettre d'effectuer le typage des ensembles de phénomènes et, ensuite, le typage des domaines. Nous pourrons alors contrôler la cohérence de la description avec les autres descriptions et la correspondance éventuelle avec un PF.

5.5 Cadre théorique

Pour établir que notre sémantique des statecharts s'intègre dans le canevas général des notations, nous devons d'abord définir comment traduire le concept d'instantané en partant des concepts Statecharts et en utilisant la couche de liaison.

L'instantané représente l'état d'une description placée dans son environnement. Il englobe tous les phénomènes intervenant dans la spécification du comportement de la description. Nous devons définir le scope de la description et déterminer quelle information nous permet de savoir si les phénomènes sont présents (se produisent) ou pas (occurrence des phénomènes).

Le scope est défini au point 5.4.1.

Pour un instantané donné, l'occurrence des phénomènes est déterminée par :

- pour les variables et les états : une valuation σ nous donne l'information sur les états actifs et sur la valeur de toutes les variables. Nous associons *true* à tous les états de $conf(\sigma) \cap states(SC)_e$ et nous associons *true* à toutes les valorisations de variables de $\sigma_{var} \cap stateVar(SC)_e$;
- pour les événements : nous associons *true* aux événements de I ou aux événements de O_{OUT}.

Une macro-étape peut être vue comme une fonction de $\Sigma \times I$ vers $\Sigma \times O_{OUT}$ (d'un instantané vers une autre instantané). Comme $I \subseteq Events$ et $O_{OUT} \subseteq Events$, nous généralisons.

L'instantané au niveau de la couche notation sera un élément de $\Sigma \times Events$.

La fonction de réaction détermine le nouvel état d'une description en fonction d'une *trace* ; cette trace représentant les différents états pris par la description depuis l'état initial.

La *trace* au niveau Jackson est comparable à l'historique au niveau Jackson. Les différents instantanés sont, pour une trace :

$\sigma_0 \overset{i_0}{\Longrightarrow}^{o_1} \sigma_1 \overset{i_1}{\Longrightarrow}^{o_2} \sigma_2 \ldots \sigma_{n-1} \overset{i_{n-1}}{\Longrightarrow}^{o_n} \sigma$

les suivants :

(σ_0, i_0), (σ_1, o_1), (σ_1, i_1), $(\sigma_2, o_2), \ldots, (\sigma_{n-1}, i_{n-1})$, (σ, o_n), (σ, I).

Nous avons pu établir la correspondance entre les concepts d'instantané et de trace (couche de Jackson) en partant, respectivement, du concept de valuation et d'événements et du concept d'historique (couche statechart). La sémantique des statecharts est une fonction qui, à partir d'un historique, nous renvoie un couple (valuation, événements). Elle est en cela tout à fait cohérente avec le canevas de sémantique défini dans le chapitre 4.

$f_{Jackson}(trace) = instantane \Leftrightarrow f_{Statechart}(historique) = (valuation, events)$

Nous pouvons en conclure que notre sémantique respecte le cadre théorique.

Chapitre 6

Sémantique de Jackson

Hall, Rapanotti et Jackson ont publié une sémantique pour les Problem Frames dans [HRJ]. Il nous a semblé important de comparer leur sémantique avec notre travail même si nous avions pris connaissance de cette publication alors que notre travail était presque terminé.

Dans ce chapitre, nous parcourons les différents points abordés dans l'article et mettons en évidence les similitudes et différences par rapport à ce que nous avons développé. Par souci de simplification, nous appellerons les auteurs de l'article HRJ.

6.1 Présentation de la sémantique

La sémantique de HRJ se focalise sur la relation entre les PF et les expressions de la forme :

$K, S \vdash R$

(où K est une description du domaine du problème, S la spécification et R le requirement).

Cette relation permet de relier le "framework" des PF avec le modèle de référence pour les exigences et spécifications. HRJ décrivent succinctement les deux dans l'article et renvoient à d'autres publications pour une approche plus complète.

Une des caractéristiques essentielles de la sémantique est qu'elle "représente et manipule seulement la structure au-dessus du niveau des descriptions de domaine". Elle peut, par exemple, s'accommoder d'une description (tout à fait informelle) du comportement d'un domaine de type opérateur, comme : l'opérateur fume.

Dans la description du framework, HRJ précisent qu'une partie essentielle

de la définition d'un problème, absente du le diagramme, sont les descriptions des domaines, les détails du requirement et de la machine. Ces descriptions seront enregistrées ailleurs. Le framework ne définit pas la notation à utiliser pour ces descriptions ; descriptions formelles ou informelles sont tout aussi acceptables pourvu qu'elles soient suffisamment précises pour capturer les caractéristiques des domaines respectifs.

La satisfaction d'une description, peut être réduite à une proposition. C'est ce qui détermine la nature propositionnelle de leur sémantique. Bien sûr, HRJ indique, qu'en principe, leur sémantique est extensible dans le cas de descriptions de domaines plus formelles.

Le Modèle de référence (MR) en quelques mots

Soit :

- la connaissance du domaine K : ce que nous savons de l'environnement ;
- l'exigence R : ce que le client requiert d'un système travaillant dans cet environnement ;
- la spécification S : la description du système qui peut être utilisée pour son implémentation.

Le MR formalise la relation entre le monde réel, les exigences et la machine comme :

$K, S \vdash R$

ce qui correspond à : "une preuve peut être donnée qu'une implémentation de S, introduite dans un domaine (celui du problème) satisfaisant K, garantira la satisfaction de l'exigence R".

Les désignations fournissent des noms pour décrire l'environnement, le système et leurs concepts associés. Le vocabulaire est utilisé pour nommer les phénomènes qui sont groupés en :

- ceux, notés e, qui appartiennent ou sont contrôlés par l'environnement ; e est partitionné entre les phénomènes visibles (observables) par la machine (e_v) et ceux qui lui sont cachés (e_h) ;
- ceux, notés s, qui appartiennent ou sont contrôlés par le système ; s est partitionné entre les phénomènes visibles (observables) par l'environnement (s_v) et ceux qui lui sont cachés (s_h).

$e_v \cup s_v$ est le vocabulaire de l'interface entre l'environnement et le système.

Le MR assure que S est écrit dans le vocabulaire de l'interface et, donc, peut uniquement utiliser des termes contenant les phénomènes de $e_v \cup s_v$. D'un autre coté, K et R peuvent utiliser des termes contenant des phénomènes de $e \cup s_v$.

Syntaxe formelle pour les PF.

HRJ définissent une syntaxe formelle pour les PF. Ils définissent une notation textuelle le "Problem Frames Description Language" (PFDL). Le but de cette notation est de fournir une couche syntaxique sur base de laquelle il est possible de définir la fonction sémantique.

Comme évoqué plus haut, certains langages sont choisis pour décrire le domaine, l'exigence et la spécification. Ce sont les propositions de leur sémantique. Ils sont référencés comme "Domain and Requirements Description Language" (DRDL). Le DRDL doit permettre la représentation de phénomènes et des relations entre eux. Il peut distinguer le contrôle de l'observation des phénomènes et peut marquer les phénomènes et les domaines. Le marquage correspond plus ou moins à ce que nous appelons le typage. Le DRDL est accompagné par une certaine notion de raisonnement, notée $\vdash_{DRDL}$.

Pour être capable d'exprimer un diagramme de problème en PFDL, HRJ font la correspondance entre les éléments graphiques et leurs contreparties dans le langage.

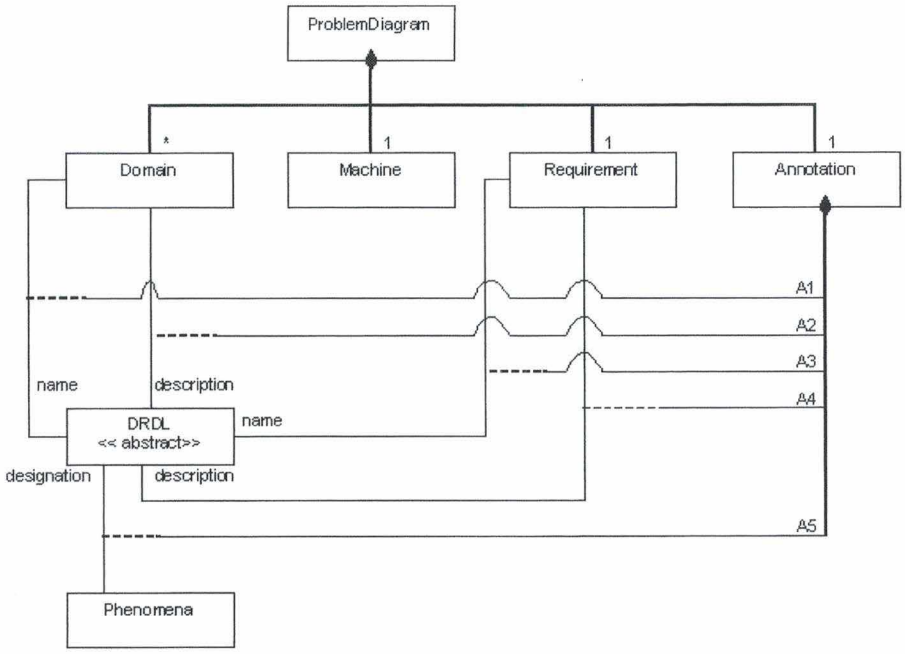
Un problème est représenté par $(D_1, ..., D_n, M, R, A)$

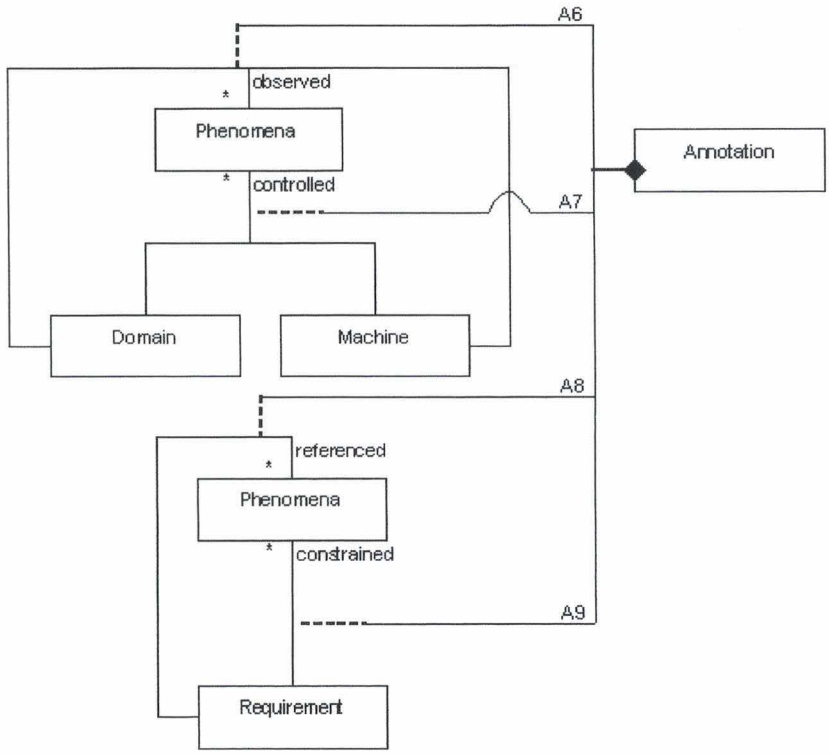
où

- $D_1, ..., D_n$ sont les différents domaines du problème ;
- M : la machine ;
- R : le requirement ;
- A : l'annotation qui réfère aux 9 fonctions d'annotation $(A_1, ..., A_9)$:
  - Domaines et Requirement doivent avoir des noms (A1,A3) et des descriptions (A2,A4) dans le DRDL.
  - Les phénomènes doivent également être nommés et décrits dans le DRDL (A5).
  - Les phénomènes sont partitionnés en ensembles de phénomènes qui peuvent contrôlés (A6) ou observés (A7) par les domaines et M.
  - De la même façon, certains phénomènes sont contraints (A8) ou bien référencés (A9) par R.

  Toutes ces fonctions sont englobées dans l'annotation d'un diagramme de problème.

En traduisant toutes ces notions en diagramme de classe, nous pourrions obtenir les diagrammes suivants (une certaine liberté a été prise par rapport aux diagrammes de classe pour faciliter la lecture) :


ProblemDiagram
*
1
1
1
Domain
Machine
Requirement
Annotation
A1
A2
A3
A4
A5
name
description
DRDL
<< abstract>>
name
designation
description
Phenomena

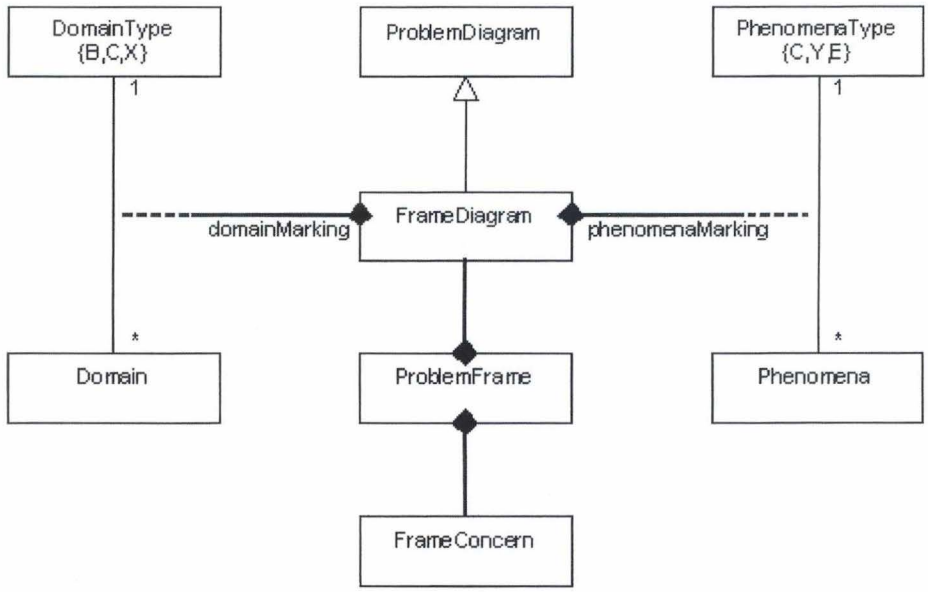

A6
*
observed
Phenomena
*
controlled
A7
Annotation
Domain
Machine
A8
*
referenced
Phenomena
*
constrained
A9
Requirement

Problem Frames

Un PF est un template pour des classes de problèmes. Le langage est enrichi par l'introduction du typage (marking) des domaines et des phénomènes. Chaque domaine peut être marqué comme biddable (B), lexical (X) ou causal (C). Chaque (ensemble de) phénomène peut être marqué comme causal (C), symbolique (Y) ou événement (E).

Le diagramme d'un PF est appelé un frame diagram et est un diagramme de problème augmenté du typage des domaines et des phénomènes.

Un PF fournit un frame concern commun à tous les problèmes de la classe. Le PF sera constitué d'un frame diagram et d'un frame concern.



Sémantique

Un diagramme de problème, à travers son environnement et son exigence, définit un "challenge" pour développer une machine permettant de le résoudre. Le challenge est la base de la sémantique de HRJ.

Étant donné une description du monde K, un requirement R et les ensembles de phénomènes partagés c (contrôlés par la machine) et o (observés par la machine) à travers certaines DRDL, HRJ définissent un challenge pour trouver une spécification satisfaisante comme :

$$c, o : [K, R] = \{S \mid S \; controls \; c \wedge S \; observes \; o \wedge \; K, S \vdash_{DRDL} R\}$$

$c, o : [K, R]$ représente l'ensemble de toutes les spécifications qui contrôlent les phénomènes de c (s_v dans le MR), observent les phénomènes de o (e_v dans le MR) et sont capables, dans le contexte de K, d'adresser R. Les détails de $\vdash_{DRDL}$ sont fournis par le DRDL.

Pour dériver un challenge pour un diagramme de problème particulier, il faut considérer sa description PFDL. Étant donné un diagramme de problème $(\{D_1, ..., D_n\}, M, R, A)$ avec $A = (A_1, ..., A_9)$, dans la sémantique, HRJ définissent un challenge comme :

$c, o : [DD_1 \wedge ... \wedge DD_n, DR]$

avec :

- $DD_i = A_2(D_i)$: les descriptions des différents domaines ;
- $DR = A_4(R)$: la description du requirement ;
- $c = A_6(M)$: les phénomènes contrôlés par la machine ;
- $o = A_7(M)$: les phénomènes observés par la machine.

Un PF est un template pour une classe de problèmes. En appliquant un PF à un diagramme de problème, on peut dériver le diagramme d'un (sous-)problème pour lequel la sémantique est un challenge.

Sémantiquement, l'application d'un PF à un diagrame de problème peut être vu comme la correspondance des domaines et des connexions (topologie des diagrammes) ainsi que des types de domaines et des phénomènes.

Un PF = (D,M,R,A,DM,PM) correspond à un diagramme de problème PD = (D',M',R',A') quand il y a une correspondance injective $\iota : D \rightarrow D'$ qui respecte la topologie et le type des domaines et phénomènes.

Une instanciation d'un PF est donc l'application de ι à (D,M,R,A,DM,PM).

6.2 Comparaison à postériori

Mettons en évidence les similitudes et différences par rapport à notre travail.

PFDL

Le langage PFDL, défini par HRJ, permet d'exprimer textuellement les diagrammes de problèmes.

Nous avons suivi la même approche à travers le métamodèle. C'est évidemment le point de départ obligé pour la construction d'un outil graphique pour les PF.

Le langage PDFL se limite à pouvoir retranscrire un diagramme de problème. HRJ évoque certaines contraintes comme le partitionnement des phénomènes aux interfaces mais la définition d'un métamodèle détaillé ne fait

pas partie du "scope" de l'article. Il est donc logique que lors de l'élaboration d'un outil nous soyons amenés à définir un modèle plus précis, plus contraignant.

Diagrammes

Dans [MM], nous faisons référence à des classes Problem, Problem Frames,... HRJ considèrent les diagrammes :

- Problem Diagram (PD)
- Frame Diagram (FD) : Problem Diagram accompagné du typage des domaines et phénomènes
- Problem Frame : Frame Diagram accompagné d'un Frame Concern.

Ces diagrammes sont les représentations graphiques des objets Problem et Problem Frame.

Nous considérons un problème comme étant un PF pour lequel les ensembles de phénomènes sont non vides. La relation entre un problème et un PF doit être vue plutôt comme de l'héritage par construction que comme une spécialisation [MM, p.3]. HRJ considèrent un FD comme étant un PD avec le typage. Ils ne prennent pas en compte l'aspect phénomènes. De plus, le typage des domaines et des ensembles de domaines d'un problème est nécessaire pour vérifier la correspondance entre un problème et un PF.

Fitting

Sémantiquement le fitting est vu de la même manière : une correspondance entre le PF et le problème des points de vue topologie du problème et typage.

Sémantique

Notre approche a été différente dans le sens où nous avons envisagé la sémantique au niveau "description de domaine" ; le but étant d'intégrer une notation dans un outil de manière bien fondée.

La question importante est de savoir si les deux approches restent cohérentes.

Relation de satisfaction

La sémantique de HRJ se base sur la relation de satisfaction :

$$D, S \vdash R$$

Cette formule a été empruntée à la logique et considérée comme une primitive que nous définissons comme suit :

$\forall m$, si $m \models D \wedge m \models S$ alors $m \models R$

Pour tout modèle, si le modèle satisfait D et S alors le modèle satisfait

R.

Nous avons défini la sémantique en disant : étant donné R, on va donner l'ensemble des modèles qui satisfont le requirement.

$$[\![R]\!] = \{m | m \models R\}$$

Voyons maintenant comment passer de l'une à l'autre.

Cadre théorique

Nous avons défini un canevas de sémantique générique pour toutes les descriptions. La sémantique est définie comme un ensemble de fonctions d'un état de la description (appelé instantané) vers un autre état.

Pour spécifier le comportement d'un système dans un domaine d'application, nous définissons un opérateur de composition des deux sémantiques, noté :

$$[\![D]\!] \,||\, [\![S]\!]$$

Fonctionnement de la composition

La sémantique pour D est exprimée en terme de fonctions de réaction passant d'un instantané de s_D (état courant dela trace t_D) vers un instantané s'_D.

Soit f_D tel que $f_D(t_D) = s'_D$

où $t_D = (s^0_D \ldots s_D)$.

Ce passage d'un instantané vers un autre (par exemple, un événement généré par le domaine d'application à destination du système) va provoquer une réaction du système, définie par f_S tel que $f_S(t_S) = s'_S$.

Cette modification de l'interface va à son tour induire une réaction du domaine d'application.

Le domaine d'application et le système communique à travers les phénomènes partagés (ce que HRJ appellent le vocabulaire de l'interface).

La notion d'instantané peut également être appliquée à l'interface entre le système et le domaine d'application.

On a :

$s_{D\cap S} \in \Sigma_{D\cap S} \Leftrightarrow dom(s_{D\cap S}) = scope(D) \cap scope(S) \wedge (\forall ph \in scope(D \cap S), \exists 1! b \in Boolean : snapshot(ph) = b)$

Pour une trace, au niveau de l'interface, on a :

$t_{D\cap S} = (s^0_{D\cap S} \ldots s_{D\cap S})$.

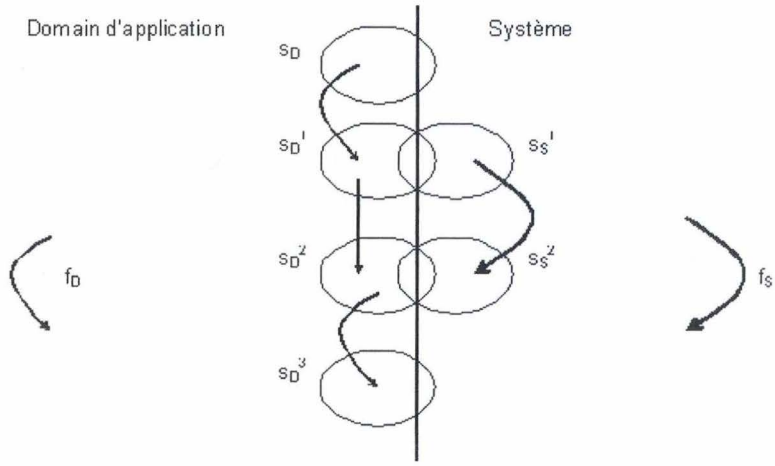
La composition est définie par :

$[\![D]\!] \,||\, [\![S]\!] \Leftrightarrow$

$\{f_D \circ f_S \circ f_D \,|(f_D(t_D) = s_D^1) \wedge (f_S(t_S^1) = s_S^2) \wedge (f_D(t_D^2) = s_D^3)\}$

avec s_x^i représentant un instantané de x à l'instant i.

Graphiquement cela pourrait se représenter comme suit :



Requirement

Le requirement est exprimé exclusivement en terme de phénomènes du domaine d'application.

Avec la fonction sémantique au niveau du requirement, nous passons également d'un instantané à un autre. Un instantané de départ qui représente des modifications du domaine d'application (par exemple un événement généré) et le résultat de la fonction qui représente les modifications attendues (un ensemble de phénomènes qui doivent être présents ou absents) .

En terme de phénomènes, si nous reprenons la figure précédente, s_R^1 ($\subseteq s_D^1$) aura comme image s_R^3 ($\subseteq s_D^3$).

Nous définissons la relation $|_R$ qui va nous permettre de comparer un instantané du domaine d'application à un instantané du requirement, simplement en réduisant l'instantané du domaine d'application aux phénomènes référencés ou contraints par le requirement. On a : $\forall s_D \in \Sigma_D$, $\forall s_R \in \Sigma_R$: $dom(s_D|_R) = dom(s_R)$

Equivalence

Nous pouvons maintenant exprimer la sémantique proposée par Jackson par la sémantique définie dans le canevas général.

On a :

$$D, S \vdash R \Leftrightarrow [\![D]\!] \,||\, [\![S]\!] \subseteq [\![R]\!]$$

ce qui signifie que :

$\forall s_D, s_D^1, s_D^3, s_R^1, s_R^3 : (f_D \circ f_S \circ f_D(t_D) = s_D^3) \wedge (f_R(t_R^1) = s_R^3) \wedge (f_D(t_D) = s_D^1) \Rightarrow (s_D^1|_R = s_R^1) \wedge (s_D^3|_R = s_R^3)$

Nous pouvons passer d'une sémantique à l'autre. Nous pouvons dès lors dire que notre canevas de sémantique reste cohérent avec l'approche proposée par HRJ dans leur article.

Chapitre 7

Conclusion

Notre travail présente les bases essentielles pour la conception d'un outil d'aide à l'analyse par la méthode de Jackson.

La conception d'un tel outil passe par une bonne compréhension des concepts établis par Jackson. Nous exposons dans le chapitre 1 les concepts et les principes essentiels de la méthode. Nous définissons d'abord tous les concepts en fonction des besoins de l'analyste et nous exposons ensuite la méthode d'analyse. Un exemple nous servira de fil conducteur dans la présentation de la méthode.

Les concepts que l'on trouve dans les problem frames ont été formalisés en grande partie par le métamodèle de G. Delannay, repris dans l'annexe A. Dans le chapitre 2, nous complétons les points laissés en suspens, notamment, le typage des phénomènes, des ensembles de phénomènes et des domaines. Le langage présenté dans le métamodèle définit précisément, en terme de phénomènes, des concepts comme un domaine, un problème, un sous-problème, un frame, etc.

Dans la méthode de Jackson, la projection d'un problème en un sous-problème permet d'isoler une partie des éléments du problème et des relations entre ceux-ci ; elle constitue une vue sur ce problème. Le sous-problème obtenu devrait correspondre à un problem frame connu, afin que l'on puisse appliquer au sous-problème l'ensemble des "recettes" de résolution qui accompagne ce problem frame. La caractérisation des phénomènes et des domaines est un élément crucial dans la réussite de cette correspondance. Elle nous permet de finaliser dans le métamodèle les concepts de cohérence au niveau des relations de projection et de correspondance avec les problem frames.

Nous relevons ensuite dans le chapitre 3 les fonctionnalités remarquables de l'outil. Nous séparons les fonctionnalités classiques d'édition des fonctionnalités plus spécifiques aux problem frames. Les premières sont présentées de

manière formelle dans l'annexe B par des cas d'utilisation et, dans l'annexe C par une approche, plus originale, utilisant la méthode des problem frames. Les secondes adressent chacune un problème particulier dans la méthode d'analyse de Jackson. Nous avons décrit d'abord chaque problème dans le contexte général des problem frames et ensuite ce que peut apporter un tel outil pour aider à résoudre un problème.

Le métamodèle que nous avons complété définit un langage (PFL) qui est cependant trop général pour pouvoir spécifier un aspect particulier d'une description. L'importation d'une description réalisée dans une notation externe est une fonctionnalité indispensable à un outil. Une grande partie de notre travail a été consacrée à ce sujet. Une notation doit être suffisamment précise pour capturer les caractéristiques intéressantes du domaine, de la spécification ou du requirement qu'elle décrit. Elle doit pouvoir exprimer un comportement observable en réponse aux modifications de l'environnement et, le cas échéant, un comportement attendu. Dans le chapitre 4, nous avons défini pour cette raison un canevas de sémantique commun à toutes les notations. Lors de l'intégration dans l'outil, il faut vérifier que la sémantique de la notation est cohérente avec notre canevas. La sémantique se base sur la notion d'instantané qui représente l'état de la description à un moment donné, en terme de phénomènes. Elle est représentée par une fonction de réaction d'un instantané vers un autre.

Une notation doit également pouvoir représenter des phénomènes et distinguer ce qui est contrôlé de ce qui est observé. Pour parvenir à traduire les concepts pertinents de la notation en phénomènes, nous avons introduit le concept de couche de liaison. Pour chaque élément de la notation, nous établissons des règles de correspondance permettant d'extraire, d'une description, l'information nécessaire à ce que nous appelons la couche de Jackson. L'ensemble des règles de correspondance constitue cette couche de liaison.

Nous définissons un cahier des charges qui reprend les éléments à prendre en compte ou à développer lors de l'intégration d'une notation particulière.

Nous appliquons ensuite la fonction de projection au modèle sémantique pour nous donner les moyens de vérifier la cohérence sémantique sur l'ensemble problème/sous-problèmes.

Pour illustrer ces concepts par un exemple, nous présentons dans le chapitre 5 l'intégration de la notation des statecharts de Harel, notation fréquemment utilisée dans les exemples fournis par Jackson. Nous construisons une sémantique fonctionnelle pour les statecharts. Conformément au cahier des charges, nous définissons ensuite les règles de correspondance entre les concepts des statecharts et les concepts de Jackson. Nous concluons l'exemple en montrant que notre sémantique respecte le cadre théorique.

Hall, Rapanotti et Jackson (HRJ) ont écrit récemment une sémantique

pour les problem frames. Dans le dernier chapitre, nous effectuons une comparaison à posteriori avec notre travail. Nous montrons que notre approche est cohérente avec celle de HRJ et, en particulier, pour les sémantiques.

Nous avons défini les éléments principaux nécessaires à la conception d'un outil. Pour l'implémentation d'un outil fonctionnel, un certain nombre de problèmes doivent encore être adressés.

Nous avons construit un cadre théorique pour l'intégration de notations externes. Nous avons effectué le travail d'intégration pour les statecharts. Il devra être complété par d'autres notations pour que l'outil soit réellement fonctionnel. Nous pensons principalement aux diagrammes de classes ou encore aux diagrammes de séquence. D'un point de vue technique, il serait judicieux d'utiliser un format standard d'importation d'une description, comme XMI, afin de laisser la possibilité à l'analyste de réaliser les descriptions avec un outil dédié à la notation externe.

Choppy et Reggio dans [CR] explore une voie très intéressante à notre sens. Elle montre comment utiliser les différentes notations UML dans la résolution de problèmes correspondant à un problem frame particulier et ce avec une approche bien fondée.

D'autres concepts pourraient également être liés aux problem frames comme la génération d'éléments de solution ou de squelette d'architecture.

En résumé, nous pensons que les problem frames ouvrent des perspectives intéressantes de développement pour appréhender la complexité inhérente aux problèmes informatiques en permettant la réutilisation des notations existantes et spécialiséess.

Bibliographie

[1] Michael Jackson, Software Requirements and Specification : a lexicon of practice, principles and prejudices, Addison-Wesley, 1995

[2] Michael Jackson, Problem Frames : analyzing and structuring software development problems, Addison-Wesley, 2001

[Broy89] Manfred Broy, Functional Specification of Communicating Systems., IFIP Congress, 1989, pp.851-856, DBLP, http ://dblp.uni-trier.de

[Broy98] Manfred Broy, Franz Huber, Barbara Paech, Bernhard Rumpe, Katharina Spies, Software and System Modeling Based on a Unified Formal Semantics In : Requirements Targeting Software and Systems Engineering, International Workshop RTSE'97, Benried, Germany, Manfred Broy, Bernhard Rumpe (ed.), pp. 43 - 68, Springer Verlag, 1998

[Cockburn] Alistair Cockburn, Writing Effective Use Cases, Addison-Wesley Professional, 2000

[DJHP97] W. Damm, B. Josko, H. Hungar, and A. Pnueli, A compositional real-time semantics of STATEMATE designs, COMPOS 97, Lecture Notes in Computer Science 1536, 1997, W.-P. de Roever, H. Langmaack, and A. Pnueli, Springer-Verlag

[EW2001] Rik Eshuis and Roel Wieringa, Reuiqrement-Level semantics for UML Statecharts, 2001

[HarelNaamad] D. Harel and A. Naamad, The STATEMATE semantics of statecharts, ACM Transactions on Software Engineering and Methodology, 1996, 5, 4, pp.293–333

[HarelPoliti] D. Harel and M. Politi, Modeling Reactive Systems with Statecharts : The STATEMATE approach, McGraw-Hill, 1996

[HRJ] Jon G. Hall, Lucia Rapanotti and Michael Jackson, Problem frame semantics for software development, Software and Systems Modeling, 2004

[Kahn77] Gilles Kahn and David B. MacQueen, Coroutines and Networks of Parallel Processes, IFIP Congress, 1977, pp.993-998

[Kovitz] Ben Kovitz, Practical Software Requirements : A Manual of Content and Style, Manning Publications Co., 1998

[MM] Gaetan Delannay, A metamodel of Jackson's problem frames, Not published, 2002, see annexe A

[Stoy77] Joseph E. Stoy, Denotational Semantics : The Scott-Strachey Approach to Programming Language Theory, 1977, IT Press, Cambridge, USA

[UML1.5] UML 1.5 Reference manual, www.omg.org

Annexes

Annexe A

Métamodèle

A metamodel of Jackson's problem frames

G. Delannay

4-03-2002

1 Introduction

This document presents a metamodel of *problem frames* as they were defined by Michael Jackson in [1] and [2]. The objective is to give a precise semantic of it, what should be useful if, for example, one wants to build a tool that supports problem frames. This metamodel is written in UML; an OCL-like language is used for defining precise constraints on UML diagrams.

The metamodel is made of eight diagrams, that correspond to the eight next sections of this document. They are all structured in the same way:

1. A 'comments' subsection gives explanations about the featured diagram and its potentially associated constraints.
2. Some aspects of Jackson's problem frames are not (or not yet) taken into account in the metamodel. Moreover, a few divergences emerged between the former and the latter. All this is included in a subsection called 'differences with Jackson'.
3. Some questions are left unanswered; some exploration paths and perspectives arose from the exercise of metamodelling. They are consigned in a subsection called 'pending questions & perspectives'.

The first subsection will always be present. The second and third ones may be missing from time to time.

In order to achieve the best possible traceability between the metamodel and Jackson's problem frames, in addition to the mentioned subsections, Jackson will be cited each time one of its concepts or ideas is introduced.

The first three diagrams present all the concepts of the metamodel, and the specialization links between them. The five next diagrams focus on the association links.

The last section concludes on the work done and presents the opportunities of further investigation.

1.1 Typographic conventions

Each time we want to emphasize an element from a diagram, we will write it in `Courier` style. Sentences written in *italics* are Jackson's citations.

2 Main concepts

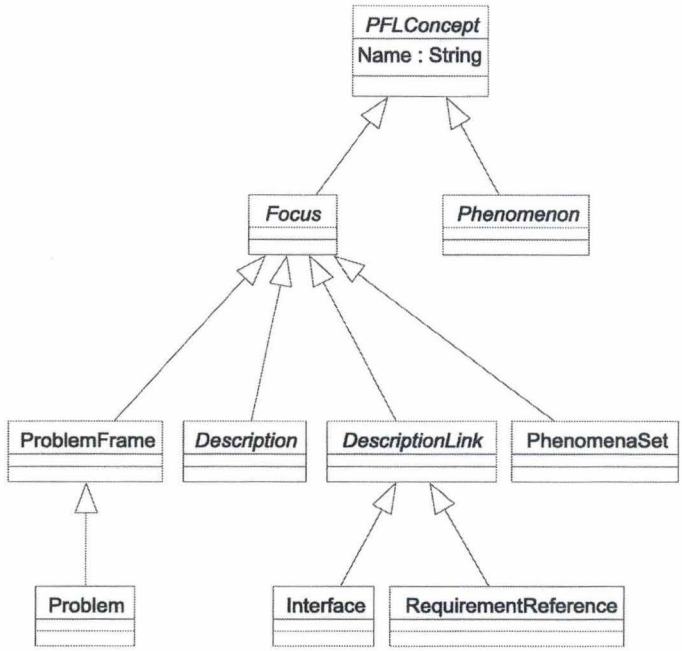


Figure 1: Main concepts

2.1 Comments

As shown by figure 1, all the concepts of the metamodel are organized in a single-rooted hierarchy, the root class being `PFLConcept`[1]. The `Phenomenon` and `Focus` classes represent the two major kinds of concepts that are present in the model :

- According to Jackson [1, p. 143], *phenomena are what* appear *to exist, or to be present, or to be the case, when you observe the world or some part of it [...].* Phenomena are of greatest importance in requirements engineering (RE), because RE's core business is to describe reality as it is and as one want it to become through the introduction of a software system. Instances of `Phenomenon`, in our model, will then be the 'representatives' of phenomena in the world that we want to talk about. We will look at the different kinds of phenomena in section 4.
- All phenomena of interest cannot be apprehended at once. The `Focus` concept enables us to turn our attention to some phenomena, in function of the particular aspect of the reality we want to emphasize.

[1] 'PFL' means 'Problem Frames Language'.

Jackson's problem frames technique belongs to the field of requirements engineering : the objective is to define *what* is the problem to solve, not *how* to define the software solution to the problem [1, pp. 206–208]. So it's rather natural to find the concept of `Problem` (or sub-`Problem`[2]) among the focus concepts.

A problem depicts a part of the reality —the application domain [1, pp. 9–11], AD— that eventually will have some needed properties thanks to the introduction of a software system, or machine [1, pp. 109–112](M). M and AD form together the problem context [1, pp. 156–158]. The needed properties are called the problem's requirement [1, pp. 169–172]. AD can be composed of several domains [1, pp. 73–76] that each have their own characteristics [1, pp. 66–70]. The machine, the domains and the requirement are the different `Descriptions`[3] —our second focus concept— that must be made in order to completely describe a problem. They will be seen in more detail in section 3.

Domains and machines interact via shared phenomena. An event generated by a machine M that has an impact in a particular domain D, or a state of D that is visible to M, are examples of phenomena shared between M and D. Through this interaction, the machine can produce the desired effect in the application domain, and hence provide a solution to the problem. The requirement cannot 'share' phenomena, because, unlike machines and domains, it doesn't represent a part (existing or to-be) of the reality; it rather expresses the properties that are required in the application domain, in terms of phenomena of the domains. So two kinds of `DescriptionLinks` —our third focus concept— exist between `Descriptions`: the `Interfaces` of shared phenomena between domains and machines (or between domains) and the `RequirementReferences` between requirements and domains.

The scope [1, pp. 176–178] of a description (machine, domain or requirement) is the particular subset of phenomena the description is concerned with. The shared phenomena of an interface and the phenomena referenced in a requirement reference are other examples of phenomena sets. All those sets are represented by the `PhenomenaSet` class.

A `ProblemFrame` [1, pp. 159–162] is a kind of abstract or generic problem that captures the commonality of several problems. There is, of course, a link between the `Problem` and `ProblemFrame` concepts ; but saying, like in figure 1, that this link is similar to the one between `DescriptionLink` and `Interface`, for example, is not really correct. Actually, the problem / problem frame link in figure 1 is closer to 'inheritance by construction' than to true 'specialization' (as the other links are). The deeper semantic of it will become clearer with diagrams of sections 5 and 6.

[2]We'll see `Problem` decomposition into other `Problems` in section 7.

[3]The definition of 'description' given by Jackson [1, pp. 58–61], includes the `Phenomenon` class: indeed, a `Phenomenon` is actually a description of a true phenomenon that can be observed in the world. Moreover, this definition could also include the notion of `Problem`. We chose the term 'description' rather as a generalization of the notions of 'tangible description' and 'intangible description' given in Jackson [2, p. 209], that will be discussed in section 3.

2.2 Differences with Jackson

Jackson, in [2], defines three main notations : context diagrams [2, p. 20], problem diagrams [2, p. 48] and frame diagrams [2, p. 85]. In this model (at the semantic level, then), a unique set of concepts is defined. Why ?

- Firstly, we think that a problem diagram is just a context diagram that shows a bit more information —most of all, the requirement.
- Secondly, concepts at the problem and problem frame level are different only in the role they play but not in their nature: as Jackson says in [1, p. 86], when talking about fitting a particular problem to a problem frame, *it's exactly like matching the actual parameters to the formal parameters in a procedure call.*

3 Descriptions

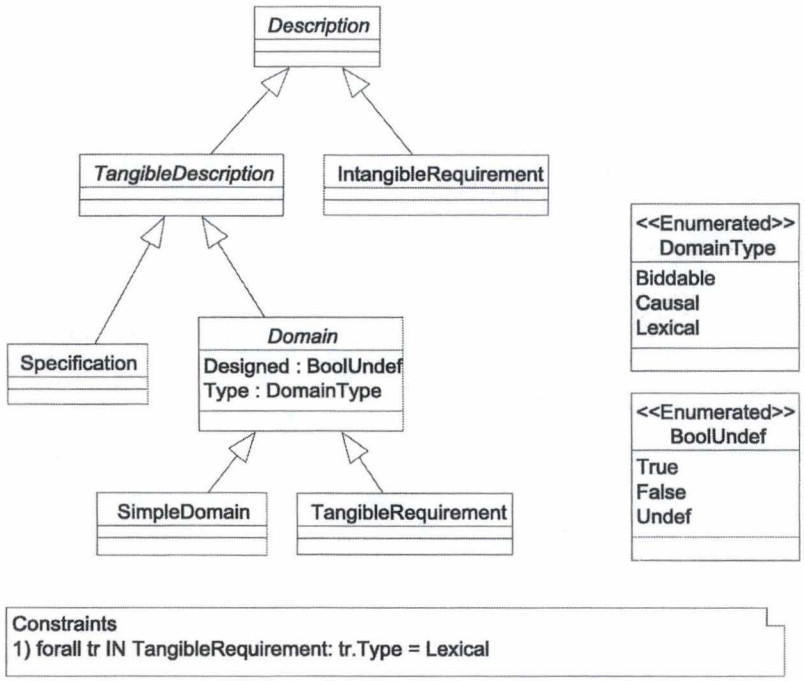


Figure 2: Kinds of descriptions

3.1 Comments

There are two kinds of descriptions : tangible and intangible ones. A tangible description is what Jackson calls a physical domain [2, p. 21]: it is a description

of a physical part of the customer's world, that already exists or will exist. So, as previously said in section 2.1, it embodies machines and domains.

- A specification can be seen as a description of a machine at its interface with the application domain : it can only be expressed in terms of phenomena shared between them [1, pp. 193–196]. The specification is the only description of the machine that we are interested in: looking at the machine's internals is a design issue. So from our point of view, the two terms are interchangeable and represented by the `Specification` class of figure 2.

- In his second book [2], Jackson refines the concept of domain and introduces the 'tangible requirement'. Normally, a requirement is intangible: it expresses wishes, not physical things. But in some cases, we may choose to write it down to a physical device (like a disk), in such a way that the machine can interpret it and conform to it through reading the device (it could be needed in cases the requirement is subjected to change very often). So when a `Domain` 'implements' a requirement in that way, it is a `TangibleRequirement`. When not, it is a `SimpleDomain`.

  A domain can be `Designed`: although making part of the application domain, we may need to structure and implement it in a particular way, such that the machine can be connected to it and interact with it. Typically, it will be the case for any information present in the application domain: the machine may need to access it or change it, but that could be impossible if we let this information in its current form —electronic, paper or even conceptual. In a problem frame, we don't care about the fact that domains are designed or not. So the possible values for the `Designed` attribute are `True`, `False` and `Undef`.

  Jackson, in [2, pp. 83–84], defines three types of domains.

  1. **Causal domains** *A causal domain is one whose properties include predictable causal relationships among its [...] phenomena.*
  2. **Biddable domains** *A biddable domain usually consists of people. The most important characteristic of a biddable domain is that it's physical but lacks positive predictable internal causality.*
  3. **Lexical domains** *A lexical domain is a physical representation of data [...].*

  So the `Type` of a domain can be `Causal`, `Biddable` or `Lexical`. As constraint 1 says on figure 2, tangible requirements are always lexical domains [2, p. 209].

An intangible description is anything but a tangible description. Requirements are the only intangible descriptions we have. As tangible requirements exist (see above), we will call those ones `IntangibleRequirements`.

3.2 Differences with Jackson

In [2, p. 209], Jackson gives an example of tangible requirement. It is represented by a solid oval, whereas an intangible one is represented by a dashed oval. Machines and normal domains are drawn as solid rectangles. So in our understanding, 'solid' means 'tangible' and 'dashed' means 'intangible'. But when Jackson says, in the same page, that *a tangible description is always a lexical domain*, he seems to talk about the intangible requirement only —because, of course, domains can be causal or biddable, too. So his 'tangible description' seems to correspond to our `TangibleRequirement` only, and not to our `TangibleDescription`.

3.3 Pending questions & perspectives

- More constraints could be expressed between the `Designed` and `Type` attributes of `Domain`. For example, are all designed domains also lexical? The definitions of 'designed' and 'lexical' domains in [2, pp. 21 & 84] don't help much: they are almost identical (both talk about *the physical representation of some information/data [...]*).

- Jackson says a lot more about domain properties and typologies (see [1, pp. 66–70, domain characteristics] and [2, pp. 143–174, frame flavours]). From [1], we understood the different domain characteristics as 'dimensions', each one being a continuum of values between two extremes. For convenience, within each dimension, a small number of key values are identified, named and described explicitly. Here are some examples of such dimensions:

    1. **Reactivity dimension.** A dimension giving behavioral information about the domain [1, pp. 66-70]. Key values:
        - **Static.** The domain is always in the same state (or there is no state): there is no notion of time.
        - **Inert.** The domain may evolve over time, but it does nothing from its own: things change only from external events.
        - **Reactive.** The domain can perform actions, but only in response to an external stimulus.
        - **Active.** The domain performs actions from its own, without needing external stimuli.
    2. **Tolerance dimension.** A dimension giving another kind of behavioral information about the domain [2, pp. 152–154]. Key values:
        - **Fragile.** The domain enters in an unknown state when an unexpected event occurs.
        - **Inhibiting.** The domain prevents unexpected events to occur.
        - **Robust.** The domains deals properly with unexpected events.

3. **Continuity dimension.** A dimension identifying the discrete or continuous nature of a domain [2, pp. 154–156]. Key values:
   - **Discrete.** The variables of the domain can take a limited number of different values.
   - **Continuous.** The variables in the domain can take an infinity of values.
4. **Structure dimension.** A dimension giving the structural characteristics of the domain ([1, p. 68] and [2, pp. 148–150]). Key values:
   - **Sequential.** The things of the domain are organized in a sequential way.
   - **Tree.** The things of the domain are organized into a tree.
   - **DAG.** The things of the domain are organized into a directed acyclic graph.
   - **Relational.** The things in the domain are organized in any graph.
5. **Formal dimension.** A dimension describing the formal aspect of the domain ([1, p. 68] and [2, pp. 162–168]). Key values:
   - **Informal.** The domain represents a part of the real world.
   - **Formal.** The domain represents a mathematical object.
6. **'Influenceable' dimension.** A dimension giving the capacity the customer or the software engineer has to influence the domain [1, p. 70]. Key values:
   - **Autonomous.** One has absolutely no influence on the domain. It is a law unto itself.
   - **Biddable.** One can bid the domain to behave in a given way, but there is no assurance that the domain will respect the instructions.
   - **Programmable.** One has complete control on the domain.

The three main categories of domains —causal, biddable and lexical— plus the 'Designed' characteristic, introduced in [2], don't match very well with the dimensions view. Does each category correspond to a particular point, or volume, in the space formed by the above dimensions ? It could be interesting to include the dimensions in the metamodel and maybe to define the main categories according to it. `Type` and `Designed` attributes are defined at the `Domain` level; dimensions could be defined at the `TangibleDescription` level, because we can see a machine/specification as a domain having special characteristics —something like [Reactive, ?, Discrete, ?, Formal, Programmable].

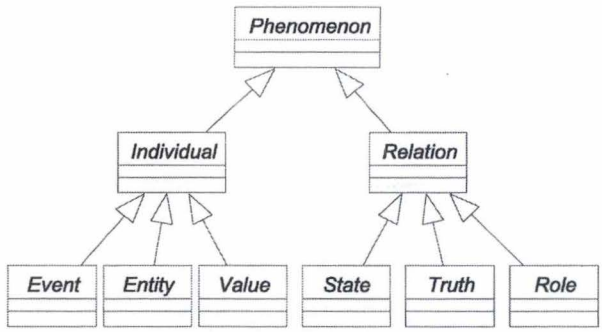


Figure 3: Kinds of phenomena

4 Phenomena

4.1 Comments

Jackson, in [2, pp. 78–83], defines the following phenomena taxonomy.

- **Individuals.** *An* `Individual` *is something that can be named and reliably distinguished from other individuals.*
  1. **Events.** *An* `Event` *is an individual happening, taking place at some particular point in time. Each event is* indivisible *and* instantaneous *[...].*
  2. **Entities.** *An* `Entity` *is an individual that persists over time and can change its properties and states from one point in time to another.*
  3. **Values.** *A* `Value` *is an intangible individual that exists outside time and space, and is not subject to change.*
- **Relations.** *A* `Relation` *is a set of associations among individuals.*
  1. **States.** *A* `State` *is a relation among individual entities and values; it can change over time.*
  2. **Truths.** *A* `Truth` *is a relation among individuals that cannot possibly change over time.*
  3. **Roles.** *A* `Role` *is a relation between an event and individuals that participate in it in a particular way.*

4.2 Differences with Jackson

Jackson [2, pp. 80–82] gives more detail about which individuals can participate in which relations. This is not yet incorporated in the metamodel.

4.3 Pending questions & perspectives

Phenomena can be thought of as a kind of assembly language for describing the world: you can express a wide range of facts about reality, but when one wants to describe a particular domain, specification or requirement, it is too general and low-level. In this case, a notation or language that is more adapted and specific to a particular description is needed. Examples of such specialized notations can be found in Jackson's books: state-machine diagrams, used for descriptions where the behavioral aspect is important (like in [2, p. 130]); tree diagrams (like in [2, p. 210]) or class diagrams (like in [2, p. 127]), used for expressing structural aspects of a description, etc.

Our idea is to allow the integration of such notations or languages in the metamodel through specialization of the (abstract) classes of figure 3 with concepts of those specific notations.

5 Problem (frame) content –1/2

Problems and problem frames are very close concepts: that's why this section and the next one present them (and their content) together. Of course, the differences between them will also arise. The link that can exist between problems will be introduced in section 7; the links between problems and problem frames are the subject of section 8.

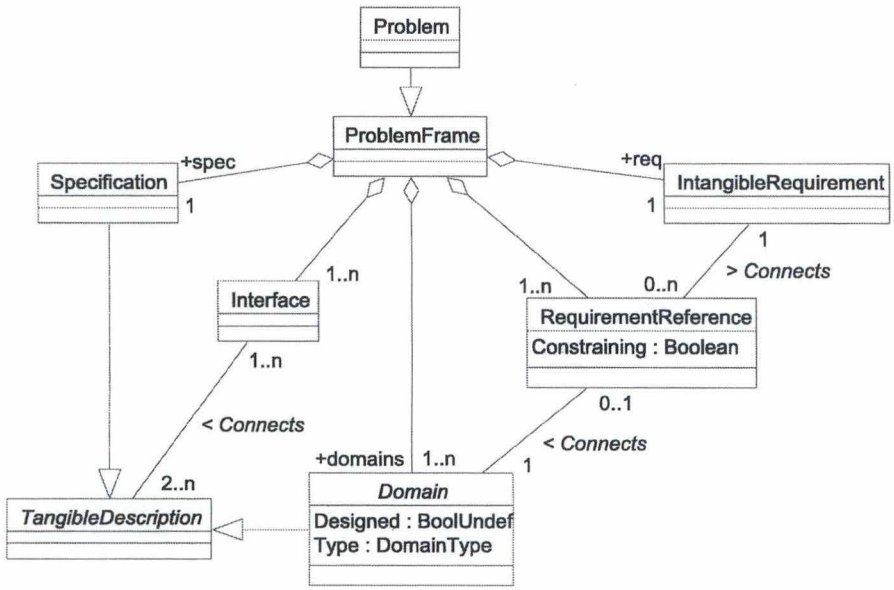


Figure 4: Content of a problem (frame) –1/2

5.1 Comments

A problem frame (or problem) contains exactly one specification and one intangible requirement and contains at least one domain. Tangible descriptions are connected with each other via interfaces; each requirement reference connects one domain to the intangible requirement.

A requirement reference can be `Constraining`: when it's the case, it means that the intangible requirement expresses properties that constraint the connected domain to behave in a specific way. When not, it means that the intangible requirement just refers to phenomena of the domain [2, p. 53].

```
Constraints
1) forall d In Description, forall dl In DescriptionLink : d Connects dl => exists 1! p IN
ProblemFrame: d IncludedIn p AND dl IncludedIn p
2) forall i IN Interface, forall p IN ProblemFrame: i IncludedIn p AND i Connects tds = {td IN
TangibleDescription} => NOT exists j IN Interface: j IncludedIn p AND j Connects s
3) forall rr IN RequirementReference, forall p IN ProblemFrame, forall d IN Domain, forall ir
IN IntangibleRequirement : rr IncludedIn p AND rr Connects d AND rr Connects ir => NOT
exists rr' IN RequirementReference: rr' IncludedIn p AND rr' Connects d AND rr' Connects ir
4) forall p IN Problem, forall d IN Domain : d.Designed IncludedIn { True, False }
5) forall p IN ProblemFrame, NOT IN Problem, forall d IN Domain : d.Designed = Undef
6) forall p IN ProblemFrame: exists rr IN RequirementReference: rr IncludedIn p AND
rr.Constraining = true
```

Figure 5: Constraints relating to figure 4

Constraint 1 in figure 5 simply forbids that things inside one problem frame be connected to things inside another one. Constraints 2 and 3 say that every description link —interface or requirement reference— is *complete*: once it connects a specific set of descriptions, no other description link can connect the same set. Constraints 4 and 5 express a small difference between problems and problem frames (already mentioned in section 3.1): in a problem, the fact that a domain is `Designed` or not must be specified, while in a problem frame the same fact is `Undefined`. Constraint 6 imposes every requirement to require something(!): in each problem (frame), at least one domain must be constrained.

5.2 Differences with Jackson

- 'Description references', that connect tangible requirements to other domains, are not yet taken into account in the metamodel.
- A problem must contain an intangible requirement, and there isn't a separate concept of 'Context': consequently, a context diagram displays a problem but hides its intangible requirement.

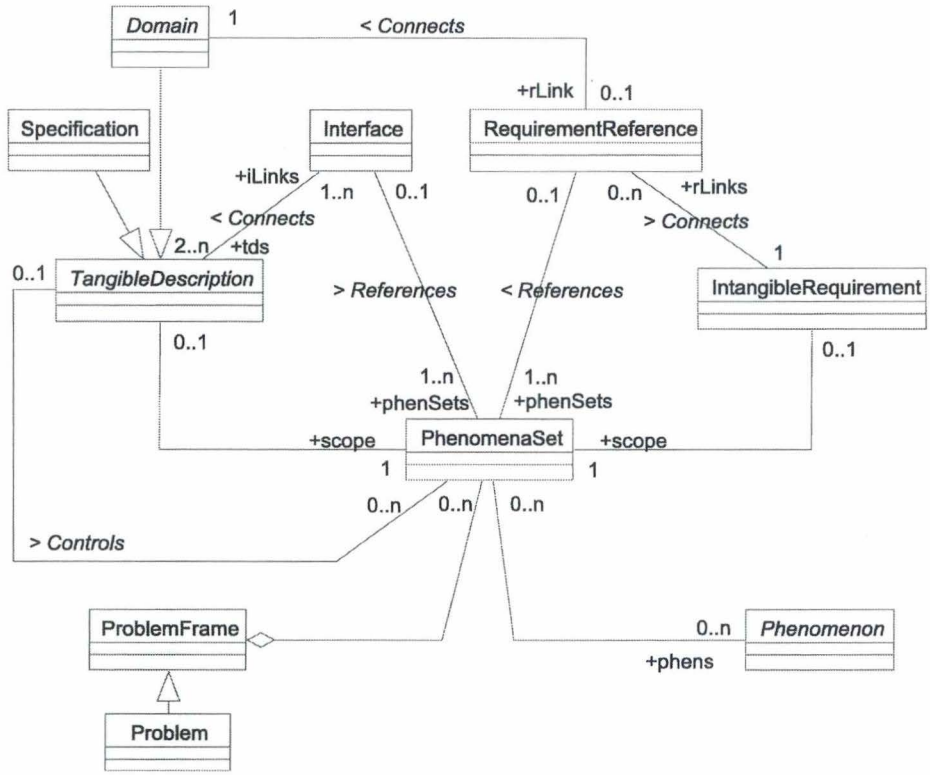


Figure 6: Content of a problem (frame) –2/2

6 Problem (frame) content –2/2

6.1 Comments

Figure 4 showed all PFL concepts included in a problem frame, excepted one: the `PhenomenaSet`, which is now visible on figure 6. On this last figure, we also see that a set regroups any number of phenomena, which, in turn, can be present in more than one set. Phenomena can even be present in sets that are included in different problems.

Figure 6 also shows all the associations existing between phenomena sets and the other PFL concepts included in a problem frame.

- As already mentioned in section 2.1, each description —tangible description or intangible requirement— has one (and only one) `scope`.
- In section 2.1, we also noticed that an interface `References` shared phenomena. But in figure 6, we see that from one to three sets can be referenced by one interface. In fact, we separate shared phenomena into sets in such a way that all phenomena of each set are *controlled* by the same

tangible description. Jackson introduces the notion of phenomena control in [2, p. 50]. Each phenomena in an interface is controlled by one domain: for example, a domain controls an event if it causes it to happen.

- So a tangible description `Controls` phenomena sets that are referenced by interfaces (or requirement references, see below).
- In 2.1, we also talked about the fact that a requirement reference references phenomena sets, too[4]. A requirement reference can `Reference` one *or more* sets, because we want to say explicitly when it references a set that corresponds to the union of several sets also referenced by interfaces.

The other associations shown in figure 6 are also present in figure 4. They are shown again in figure 6, with some role names, because we will mention it in the constraints linked to figure 6.

Constraints

Global constraints
1) forall dl IN DescritionLink, forall ps IN PhenomenaSet : dl References ps => exists 1!p IN ProblemFrame : dl IncludedIn p AND ps IncludedIn p
2) forall d IN Description, forall ps IN PhenomenaSet : d.scope = ps => exists 1!p IN ProblemFrame : d IncludedIn p AND ps IncludedIn p
3) forall td IN TangibleDescription, forall ps IN PhenomenaSet : td Controls ps => exists 1!p IN ProblemFrame : td IncludedIn p AND ps IncludedIn p
4) forall i IN Interface, forall ps IN PhenomenaSet : i References ps => exists td IN TangibleDescription : i Connects td AND td Controls ps
5) forall td IN TangibleDescription, forall ps IN PhenomenaSet : td Controls ps => exists i IN Interface : i Connects td AND i References ps
6) forall rr IN RequirementReference, forall i IN Interface, forall ps IN PhenomenaSet : rr References ps AND i References ps => exists td IN TangibleDescription : rr Connects td AND i Connects td

Constraints about PhenomenaSets
7) forall pf IN ProblemFrame, forall ps IN PhenomenaSet : pf NOT IN Problem AND ps IncludedIn pf => ps.phens = EMPTY
8) forall p IN Problem, forall ps IN PhenomenaSet : ps IncludedIn p => ps.phens NOT EMPTY
9) forall p IN Problem, forall ps1, ps2 IN PhenomenaSet : ps1 IncludedIn p AND ps2 IncludedIn p AND ps1 <> ps2 => ps1.phens <> ps2.phens
10) forall pf IN ProblemFrame, forall ps IN PhenomenaSet: ps IncludedIn pf =>
(exists dl IN DescriptionLink : dl References ps) OR
(exists d IN Description: d.scope = ps)

Figure 7: Constraints relating to figure 6 –1/2

6.1.1 Global constraints

Constraints 1 to 3 on figure 7 are similar to constraint 1 of figure 4: they just state that things inside one problem frame cannot be connected to things inside another one. Constraints 4 and 5 say the following:

[4]Interfaces and requirement references both 'reference' phenomena sets, but we will see that the two associations have quite different semantics.

- a phenomena set referenced by an interface must be controlled by a tangible description connected to this interface (constraint 4);
- by symmetry, a tangible description that controls a phenomena set must be connected to an interface that references this set (constraint 5);
- but an interface that connects a tangible description doesn't necessarily reference a phenomena set controlled by this description. In other words, in the context of an interface, one of the connected descriptions may control no phenomena at all.

Constraint 6 states that if an interface and a requirement reference both reference the same phenomena set, they must both be connected to a common tangible description[5].

6.1.2 Constraints on phenomena sets

According to constraint 7, problem frames that are NOT problems contain only empty phenomena sets. That's the most important semantic difference between problems and problem frames: frames 'abstract' completely the level of individual phenomena. Constraint 8 force problems to include only phenomena sets that are not empty. Constraint 9 says that all phenomena sets included in a problem are distinct sets of phenomena. It means that the concept of phenomena 'set' holds only in the context of a problem: two different phenomena sets included in two different problems can regroup exactly the same phenomena. Constraint 10 avoids the definition, in problem (frames), of 'isolated' phenomena sets (that are not linked to any description or description link).

6.1.3 Constraints on interfaces

All interfaces in a problem are disjoint [2, pp. 51–52]: shared phenomena appearing in one interface cannot appear in another one. Constraints 11 to 13 say it, with some refinements.

- Firstly, a phenomena set can be referenced by only one interface (that's constraint 11; notice that i and j can designate identical or different interfaces)[6].
- Secondly, two phenomena sets referenced by the same interface must not intersect (constraint 12). With global constraints, it implies that a shared phenomena can be controlled by only one tangible description.
- Finally, for any couple of interfaces included in a problem, the union of all sets referenced by one interface must not intersect with the union of

[5] In the context of a problem, this constraint is not necessary because of the presence of other constraints (see below) that concern phenomena sets. Constraint 6 is needed for problem frames that are not problems and that, as we will see, only contain empty phenomena sets.

[6] Again, this constraint is only needed for problem frames; for problems, constraints 12 and 13 are sufficient.

Constraints on interfaces
11) forall i IN Interface, forall ps IN PhenomenaSet : i References ps => NOT exists j IN Interface : j References ps
12) forall i IN Interface, forall ps, ps' IN PhenomenaSet : ps <> ps' AND i References ps AND i References ps' => ps.phens INTER ps'.phens = EMPTY
13) forall p IN Problem, forall i, j IN Interface : i, j IncludedIn p => i.phenSets[*].phens INTER j.phenSets[*].phens = EMPTY

Constraints on interfaces and scopes
14) forall p IN Problem, forall i IN Interface, IncludedIn p: i.phenSets[*].phens = (INTER(td.scope.phens), forall td IN i.tds) \ (UNION(INTER(td".scope.phens), forall td" IN ifc.tds), forall ifc: (ifc IN (INTER(td'.iLinks), forall td' IN i.tds) AND ifc.tds.Length > i.tds.Length))
15) forall sp IN Specification : sp.scope.phens = sp.iLinks[*].phenSets[*].phens
16) forall d IN Domain : d.scope.phens IncludedOrEqualTo d.iLinks[*].phenSets[*].phens UNION d.rLink.phenSets[*].phens
17) forall d1, d2 IN Domain, forall p IN Problem : d1, d2 IncludedIn p => (d1.scope.phens \ (d1.iLinks[*].phenSets[*].phens)) INTER (d2.scope.phens \ (d2.iLinks[*].phenSets[*].phens)) = EMPTY

Constraints on requirement references
18) forall rr IN RequirementReference, forall ps, ps' IN PhenomenaSet : ps <> ps' AND rr References ps AND rr References rr' => ps.phens INTER ps'.phens = EMPTY
19) forall rr IN RequirementReference, forall d IN Domain, forall ps IN PhenomenaSet, forall p IN Problem : rr IncludedIn p AND rr Connects d AND rr References ps => ps.phens <> AnyUnion(ps'), forall ps' IN PhenomenaSet : exists i IN Interface AND i IncludedIn p AND i References ps' AND i Connects d.

Constraints on requirement references and scopes
20) forall rr IN RequirementReference, forall d IN Domain : rr Connects d => rr.phenSets[*].phens IncludedIn d.scope.phens
21) forall ir IN IntangibleRequirement : ir.scope.phens = ir.rLinks[*].phenSets[*].phens

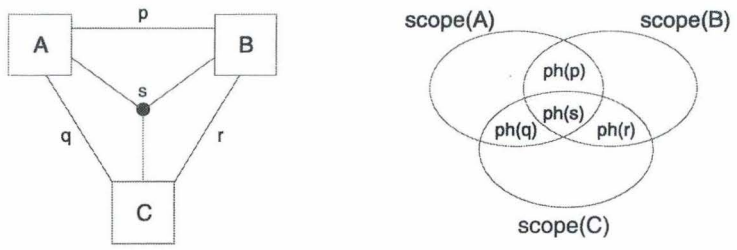
Figure 8: Constraints relating to figure 6 –2/2

all sets referenced by the other (constraint 13; notice that the star means 'all' and not 'any').

Figure 9 helps to understand these constraints and the following. Figure 9 (a) (taken from [2, p. 52]) shows three domains and all the possible interfaces between them, while figure 9 (b) shows the corresponding phenomena sets that come into play: the domain scopes and the shared phenomena referenced by the interfaces.

6.1.4 Constraints on interfaces and scopes

Constraints 14 say that an interface between domains represents the intersection of the domain's scopes. More precisely, if an interface i links n domains, the set of phenomena referenced by i is equal to the intersection of the n domains' scopes, minus the intersection(s) of the same domains with others, in the case they are linked by one or more other interfaces that links more than n domains. For example, on figure 9,



(a) Domains and interfaces

(b) Corresponding phenomena sets

Figure 9: Domains, interfaces and phenomena sets

$$ph(p) = scope(A) \bigcap scope(B) \setminus (scope(A) \bigcap scope(B) \bigcap scope(C)).$$

Constraints 15 to 17 talk about private phenomena (phenomena that are not referenced by any interface).

- A specification cannot have private phenomena: we cannot look at the machine's internals (constraint 15).
- A domain may of course have private phenomena. In general, they are referenced by the requirement reference connected to the domain, but it is not always the case: see [2, p. 145]. This is what constraint 16 explains.
- Constraint 17 expresses a quite obvious fact: in a problem, phenomena private to a domain cannot be private to another one.

6.1.5 Constraints on requirement references

A requirement reference can reference phenomena private to its connected domain, but also shared phenomena that this domain shares with some other tangible description(s). Consequently, requirement references can reference phenomena sets that are already referenced by interfaces. When that's the case, the metamodel must represent it explicitly. The following constraints and constraint 9 should be sufficient to enforce this.

- Constraint 18 is similar to constraint 12, but is applicable to requirement references instead of interfaces: two phenomena sets referenced by the same requirement reference must not intersect.
- Constraint 19 says that any phenomena set referenced by a requirement reference must not be equal to any union of two or more sets referenced by an interface. If we want to express something equivalent, the requirement reference must directly reference the sets referenced by the interface.

6.1.6 Constraints on requirement references and scopes

Constraint 20 says that the phenomena referenced by a requirement reference must be included in its connected domain's scope. Constraint 21 says that an intangible requirement cannot have 'private' phenomena —which has no sense.

6.2 Differences with Jackson

In frame diagrams, Jackson gives a type for phenomena sets that are referenced by interfaces or requirement references: causal, symbolic or events. This is not yet incorporated in the metamodel.

6.3 Pending questions & perspectives

Must we say that two requirement references must not intersect? It is not always the case : if domains D_1 and D_2 are linked by an interface i, requirement references R_1 and R_2 that are respectively linked to D_1 and D_2 can both reference phenomena of i.

7 Projections

In last sections, we have emphasized the content of one problem or problem frame. From now on, we will focus on the relations existing between problems and problem frames. We will begin with inter-problem relationships.

Realistic problems are rarely so simple that they can be represented, analyzed and fully understood via a single `Problem` of the metamodel. We thus need to decompose problems into subproblems [2, pp. 57–76]. In Jackson's approach, the way to decompose has two remarkable characteristics:

- Firstly, subproblems are *projections* of problems: they don't form partitions (see figure 10, taken from [2, p. 65]). Elements discussed in one subproblem can be found again in another subproblem. So in a problem, the focus is not on all aspects of a particular 'region' of the reality (that would be partitioning), but rather on one particular aspect that crosses several regions. Problem decomposition through projections is the topic of this section.

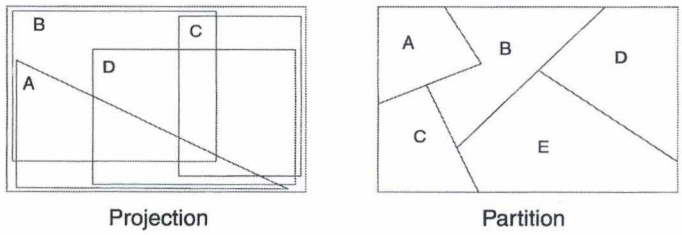


Figure 10: Projections and partitions

- Secondly, problem decomposition is driven by the concern for making the subproblems correspond to (or *fit*) well-known, well-understood and well-documented classes of problems: the problem frames. This will be the subject of next section.

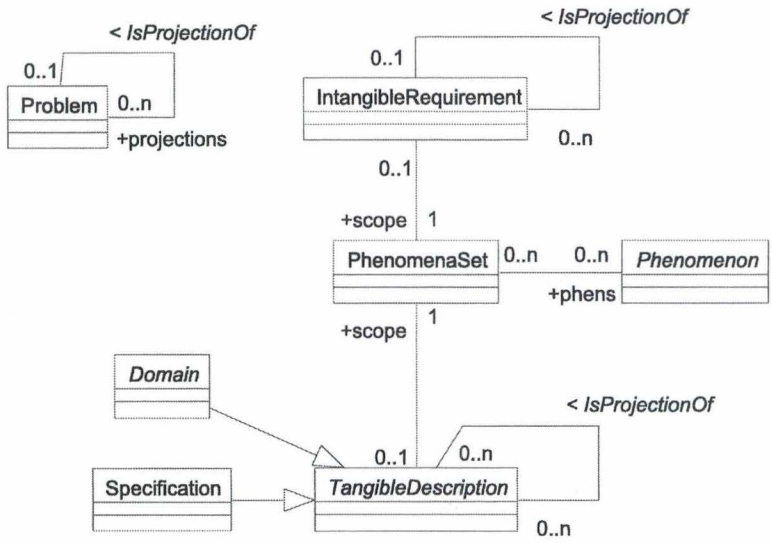


Figure 11: Projections

7.1 Comments

According to figure 11, a problem can be decomposed into any number of other problems; each subproblem being a `ProjectionOf` his parent problem. We will see when we will comment the constraints linked to figure 11, that projections between problems are defined in terms of projections between problem parts. At this 'parts' level, figure 11 shows that an intangible requirement can be the `ProjectionOf` another one; a tangible description can be the projection of many other tangible descriptions. Scopes and phenomena appear on figure 11 because projections of descriptions are defined in terms of it.

Figure 11 allows a domain to be the projection of a specification. Indeed, it is or prime importance to be able to consider, when analyzing a particular subproblem, that all the other subproblems are solved [2, p. 60], and then to be able to consider their subspecifications/submachines as making part of the application domain of the particular problem. We can model this by defining a domain of the particular problem as a projection of the machine/specification of the whole problem. This projection corresponds to subspecifications solved by other subproblems.

Figure 11 allows two domains included in a same problem (at level n) to be

projections of identical description(s) of level $n-1$, if, of course, the problem of level n respects all the constraints applicable to a problem.

```
Constraints

Global constraints
1) forall p, p' IN Problem: p IsProjectionOf p' => p <> p'

Constraints about projections between Problems and projections between Descriptions
2) forall p, p' IN Problem : p IsProjectionOf p' => forall d IN Description, IncludedIn p: d
IsProjectionOf ds AND forall d' IN ds: d' IncludedIn p'
3) forall ir, ir' IN IntangibleRequirement : ir IsProjectionOf ir' => exists p, p' IN Problem : p
IsProjectionOf p' AND ir IncludedIn p AND ir' IncludedIn p'
4) forall td, td' IN TangibleDescription : td IsProjectionOf td' => exists p, p' IN Problem : p
IsProjectionOf p' AND td IncludedIn p AND td' IncludedIn p'

Constraints about projections between TangibleDescriptions
5) forall sp IN Specification, forall td IN TangibleDescription : sp IsProjectionOf td =>
td.Class = Specification AND NOT exists td' IN TangibleDescription: sp IsProjectionOf
td'

Constraints about projections and scopes
6) forall ir1, ir2 IN IntangibleRequirement : ir2 IsProjectionOf ir1 => ir2.scope.phens
IncludedButNotEqualTo ir1.scope.phens
7) forall sp1, sp2 IN Specification : sp2 IsProjectionOf sp1 => sp2.scope.phens
IncludedButNotEqualTo sp1.scope.phens
8) forall d IN Domain : d IsProjectionOf tds = {td IN TangibleDescription} => (forall i IN
Integer, 0 <= i <= tds.Length-1 : d.scope.phens INTER tds[i].scope.phens <> EMPTY)
AND d.scope.phens IncludedOrEqualTo tds[*].scope.phens

9) forall p IN Problem: (UNION(sp.spec.scope.phens UNION sp.req.scope.phens UNION
sp.domains[*].scope.phens), forall sp IN p.projections) = p.spec.scope.phens UNION
p.req.scope.phens UNION p.domains[*].scope.phens
```

Figure 12: Constraints relating to figure 11

Constraint 1 on figure 12 says that a problem cannot be a projection of itself. This is also true for descriptions, but the following constraints imply it. Constraints 2 to 4 define problem projections from description projections.

- A problem is a projection of another one if all descriptions that are included in it are projections of descriptions included in the other problem (constraint 2).
- A description can be a projection of another one only if it is contained in a problem that is a projection of a problem that contains the other description (constraints 3 and 4).

Constraint 5 gives some precisions about the projection between domains: a `Specification` can be a projection of at most one description; this description must be another `Specification`.

Constraints 6 to 8 define projections between descriptions from scopes and phenomena.

- "An intangible requirement or a specification is a projection of another description" means that its `scope` must be included (but not equal to) the other description's scope (constraints 6 and 7).

- "A domain is a projection of one or more other description(s)" means that its `scope` must be included in the union of the other descriptions'`scopes`, and that the intersection of its `scope` with each other description's `scope` must not be empty.

At this point, a question may arise. A domain can have private phenomena, but a specification can't. Can the definition of a domain as a projection of a specification represent a way to look at the specification's private phenomena ? No, because the domain's scope must be included in the specification's scope.

Constraint 9 is about sub-problem completeness: every phenomena mentioned in a given problem must also be mentioned in at least one of its subproblems[7].

7.2 Differences with Jackson

- The 'IsProjectionOf' association between problems allows the definition of trees of decomposed problems at any depth. In Jackson's books ([1] and [2]), there is no example of more than two levels —a main problem decomposed into several subproblems. We don't see the utility of more levels, but we don't see reasons to forbid it.

- In the metamodel, it is not possible to define a description included in a subproblem at level n to be a projection of another description included in another subproblem of the same level. This can be surprising. Let's see it on an example. In [1, pp. 128–132], Jackson defines a main problem that consists in managing CASE objects in a company. The main problem is decomposed into three subproblems:

  1. Subproblem 1 focuses on the creation and manipulation of the CASE objects by the company's developers;
  2. Subproblem 2 concentrates on the need to deliver to managers information about the manipulation of CASE objects by developers;
  3. Subproblem 3 deals with the problem of prohibiting particular users to perform particular operations on the CASE objects.

  Let's call the machine of subproblem 1 the 'CASE editor'. The machine of subproblem 3 must limit access to operations that can be performed via the CASE editor: the latter is then a part of subproblem 3's application domain. In our metamodel, we cannot express any direct link between the CASE editor and a domain in subproblem 3. We just say that both the CASE Editor and the domain in subproblem 3 are projections of the main

[7] The `spec`, `req` and `domains` attributes of `Problems` are shown on figure 4.

problem's machine; nevertheless, we can indirectly find the link between them because we know the scope of each description.

7.3 Pending questions & perspectives

- A domain can be a projection of several tangible descriptions. Are there any constraints on those descriptions? For instance, can they have disjoint scopes? More generally, can a domain's scope contain groups of phenomena that have no relation with each other[8]?
- Suppose that a domain d is a projection of tangible descriptions td_1 and td_2 that are connected by interface i. The intersection of d's scope with td_1's scope equals i's shared phenomena. What does it mean to say (or not) that d is a projection of td_1? Formally speaking, It is sufficient to say that d is a projection of td_2.
- In last section, we introduced the concept of phenomena control. Is there any constraints between 'IsProjectionOf' associations linking descriptions and `Controls` associations involving the same descriptions ?
- The concept of projection is quite interesting but its definition in terms of scopes inclusions is maybe too general. We should explore ways to define this concept with more precision.

8 Fits

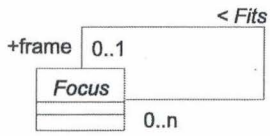
Problem frames are classes [2, pp. xii and 76] of problems, or kinds of patterns [2, p. 76] —they share the spirit of design patterns [2, p. xii]. We can also see a problem frame as a set of constraints that must be respected by all problems that fit it. In sections 5 and 6, we already defined problems and problem frames. This section will present the way to link them, which has lot in common with the matching of the actual parameters to the formal parameters in a procedure call.

8.1 Comments

How can we represent the fact that a problem can fit a particular problem frame? Figure 13 shows it: any focus concept can `Fit` another focus concept, which, in turn, can be the `frame` of several focus concepts. What does it mean exactly?

- At the problem / frame level, any problem can `Fit` a problem frame, that can be the `frame` of several problems (constraint 3).

[8] According to figure 4, every tangible description within a problem (frame) must be connected to another one. Maybe we should ensure that the same kind of property holds within a domain.



Constraints

Structural Constraints
1) forall f1, f2 IN Focus : f1 Fits f2 => f1 <> f2

Constraints restricting the subclasses of Focus used in the Fits association
2) forall f1, f2 IN Focus: f1 Fits f2 => NOT(f1.Class = f2.Class = ProblemFrame)
3) forall f1, f2 IN Focus: f1 Fits f2 AND f1.Class = Problem => f2.Class = ProblemFrame
4) forall f1, f2 IN Focus: f1 Fits f2 AND NOT f1.Class = Problem => f2.Class = f1.Class

Matching 1-1 between problem & problem frame elements
5) forall p IN Problem, forall pf IN ProblemFrame : p Fits pf <=> (forall p_elem IN Focus, IncludedIn p : exists pf_elem IN Focus : pf_elem IncludedIn pf AND p_elem Fits pf_elem) AND (forall pf_elem IN Focus, IncludedIn pf: exists p_elem IN Focus : p_elem IncludedIn p AND p_elem Fits pf_elem)

Figure 13: Fits

- At the problem / frame parts level, any focus concept included in a problem can fit another focus concept of the same type[9] included in a problem frame (constraints 4 and following).

Constraint 2 specifies that a problem frame cannot fit another problem frame, while constraint 1 says that a focus cannot fit itself.

Constraints 5 and 6 express that there must be a 1:1 correspondence between concepts at the problem and at the frame level. The next constraints, shown on figure 14, give more detail about this correspondence, concept by concept.

- **Requirement references.** A constraining reference fits another one only if they are both constraining or both not constraining (constraint 7); if the fitted requirement reference connects a given domain, the fitting requirement reference must connect a domain that fits it (constraint 8).
- **Interfaces.** An interface that fits another interface that references n phenomena sets must also reference n phenomena sets (constraint 9); if the fitted interface connects m tangible descriptions, the fitting interface must connect descriptions that fit (one by one) those m descriptions (constraint 10).
- **Tangible descriptions.** The fitted and the fitting tangible descriptions must be connected to the same number of interfaces (constraint 11).

[9] "Of the same type" means that they must be instances of the same class.

Fit between RequirementReferences
7) forall rr_p, rr_pf IN RequirementReference : rr_p Fits rr_pf => rr_p.Constraining = rr_pf.Constraining
8) forall rr_p, rr_pf IN RequirementReference, forall d_pf IN Domain : rr_p Fits rr_pf AND rr_pf Connects d_pf <=> exists d_p IN Domain : rr_p Connects d_p AND d_p Fits d_rp

Fit between Interfaces
9) forall i_p, i_pf IN Interface : i_p Fits i_pf => i_p.phenSets.Length = i_pf.phenSets.Length
10) forall i_p, i_pf IN Interface : i_p Fits i_pf AND i_pf Connects tds_pf = { td_pf IN TangibleDescription } <=> exists tds_p = { td_p IN TangibleDescription } : i_p Connects tds_p AND (forall td_pf IncludedIn tds_pf : exists td_p IncludedIn tds_p: td_p Fits td_pf) AND (forall td_p IncludedIn tds_p : exists td_pf IncludedIn tds_pf: td_p Fits td_pf)

Fit between TangibleDescriptions
11) forall td_p, td_pf IN TangibleDescription : td_p Fits td_pf => td_p.iLinks.Length = td_pf.iLinks.Length

Fit between Domains
12) forall d_p, d_pf IN Domain : d_p Fits d_pf => d_p.Type = d_pf.Type

Fit between IntangibleRequirements
13) forall ir_p, ir_pf IN IntangibleRequirement : ir_p Fits ir_pf => ir_p.rLinks.Length = ir_pf.rLinks.Length

Fit between PhenomenaSets
14) forall ps_p, ps_pf IN PhenomenaSet, forall i_pf IN Interface, forall rr_pf IN RequirementReference : ps_p Fits ps_pf AND i_pf References ps_pf AND rr_pf References ps_pf <=> exists i_p IN Interface, exists rr_p IN RequirementReference : i_p Fits i_pf AND rr_p Fits rr_pf AND i_p References ps_p AND rr_p References ps_p
15) forall ps_p, ps_pf IN PhenomenaSet, forall td_pf IN TangibleDescription : ps_p Fits ps_pf AND td_pf Controls ps_pf => exists td_p IN TangibleDescription : td_p Controls ps_p AND td_p Fits td_pf

Figure 14: Next constraints relating to figure 13

- **Domains.** The fitted and the fitting domains must be of same Type (constraint 12).
- **Intangible requirements.** The fitted and the fitting intangible requirements must be connected to the same number of requirement references (constraint 13).
- **Phenomena sets.** A phenomena set that fits another one that is referenced by both an interface and a requirement reference must also be referenced by an interface and a requirement reference. Moreover, the latter must fit the former (constraint 14). A phenomena set that fits another one that is controlled by a tangible description must be controlled by another tangible description that fits the first one (constraint 15).

8.2 Differences with Jackson

- Constraints of figure 13 imply what we would call *strong fitting*:

1. if, in the frame, an interface references sets s_1 and s_2, the fitted interface in the problem cannot reference only s_1 or s_2;
2. if, in the frame, an interface references only one set s_1, the fitted interface in the problem cannot reference s_1 plus other set(s);
3. if, in the frame, an interface references set s_1, and a requirement reference references s_1 too, the problem cannot define the fitted requirement reference to reference another set $s_2 \neq s_1$.
4. if, in the frame, an interface i references sets s_1 and s_2, and a requirement reference linked to a domain linked to i references set s_3, in the fitted problem, the fitted requirement reference cannot reference s_1 or s_2, or both.

The problem and the frame must match exactly, nothing more, nothing less. In [2], Jackson presents what we would call *weak fitting*: although he expresses the equivalent of point 3 above (*if two sets have the same name they are exactly the same phenomena [2, p. 88].*) and gives no counterexample of point 2, he shows an example that violates point 1[10] [2, p. 87] and refutes point 4 [2, p. 88]: *[...] two sets with different names might be the same phenomena in a particular problem.*

- As previously said, phenomena sets that are referenced by interfaces or requirement references are not yet given a type —causal, symbolic, events— in our metamodel. We think that when we want to check that a problem fits a given problem frame, we must notably deduce the `Type` of the problem's `Domains` and `PhenomenaSets`, and compare them with the types of the elements they fit in the frame. We believe that this deduction must be based on the kinds of phenomena (events, entities, states, roles, etc) that are present in the concerned phenomena sets (domains' scopes and sets referenced by interfaces and requirement references). The information necessary to perform this deduction is not yet included in the metamodel.
- Jackson envisages different kinds of links between problem frames: a frame can be a variant [2, p. 207] or a flavour [2, p. 142] of another one. These links are not yet included in the metamodel.

8.3 Pending questions & perspectives

- Strong fitting seems more precise and systematic to us, but isn't it too constraining ?
- Sometimes we may want(?) to say that in a problem frame, the set(s) referenced by a requirement reference that connects a given domain must not intersect any of the sets referenced by interfaces that are connected to the same domain. We have no means to say that, nor in the metamodel, nor with Jackson's notations.

[10] But he says on p. 89 about the same example, that it makes a huge difference if the fitted interface references or not a second set defined at the frame level.

- Aren't constraints 9, 11 and 13 superfluous?
- Must we enforce that only leaf `Problems` in projection trees can fit a problem frame ?
- A problem frame restricts, in the fitted problems, the number of descriptions, the way they are linked together, and the *type* of those descriptions and links, but it says nothing about the individual phenomena that are featured in the descriptions and links. Problem frames can be felt as too general —the boxes in the diagrams are empty! Patterns at the design level, on the contrary, already include some 'generic phenomena' —classes, methods, attributes, and collaborations between classes. In the piece of design that fits the pattern, you find equivalents of those generic phenomena, plus other more specific elements. In fact, this way to proceed is closer to the notion of *projection* than the notion of *fit*. So maybe we should try to explore the possibility to apply the notion of *projection* for both problem decomposition and problem frame fitting.

9 Name spaces

This section identifies the different name spaces that are present in the metamodel. Figure 15 shows all the relevant elements (that were all introduced in preceding sections). Notice that specifications, interfaces, domains, requirement references and intangible requirements *are* PFL concepts, and then have a `Name`. This is not shown in the diagram because this is not direct inheritance.

9.1 Comments

There is a global name space, where all the phenomena are defined. So each phenomena has a name that is unique (constraint 1 on figure 16). Each problem frame constitutes a name space: all focus concepts included in it have a name that is unique in it (constraint 2). A problem is also a name space: all focus concepts included in it or that are projections of it have a name that is unique in it (constraint 3).

Until now, we did'nt mention the existence of abbreviations: each tangible description has an abbreviation (an identifier made of two characters, constraint 5) that is used in problem or frame diagrams to indicate which tangible description controls a given set of phenomena [2, p. 50]. Constraint 4 says that abbreviations, too, are unique within a problem / frame name space.

9.2 Differences with Jackson

- In the metamodel, for homogeneity purposes, every concept has a name —the `PFLConcept` class has a `Name` attribute. In Jackson's diagrams (context, problem and frame diagrams), some of our names never appear. For

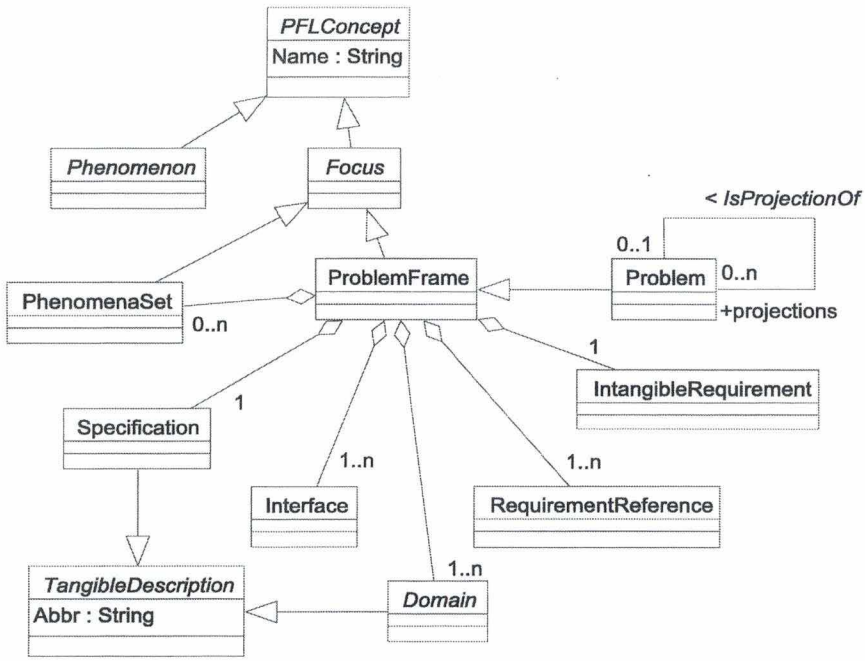


Figure 15: Concepts involved in name spaces

example, scopes have no name, phenomena sets are unnamed in problem diagrams and interfaces are unnamed in frame diagrams.

- The identifiers marking the lines representing interfaces in Jackson's problem diagrams correspond to the `Names` of `Interfaces` in this metamodel, but with a small divergence: according to section 6, it isn't possible to have two identifiers on one interface in a problem diagram. Jackson says in [2, p. 51] that it could be possible but didn't give an example of it.

- As already mentioned, names of phenomena sets do not appear in Jackson's problem diagrams; what appear is the names of interfaces and requirement references. In fact, it's a bit more subtle: if a requirement reference references exactly the phenomena set(s) referenced by an interface, the name appearing on the requirement reference will be the name of the interface; else, it will be the name of the requirement reference.

- Jackson gives a convention for naming the phenomena sets in problem frames, according to their type. This is not yet taken into account because phenomena sets types aren't.

9.3 Pending questions & perspectives

More name spaces should be created.

Constraints
1) forall ph1, ph2 IN Phenomenon : ph1.Name = ph2.Name <=> ph1 = ph2
2) forall pc1, pc2 IN ProblemFrameConcept, forall p IN ProblemFrame, NOT IN Problem: pc1 IncludedIn p AND pc2 IncludedIn p AND pc1.Name = pc2.Name <=> pc1 = pc2
3) forall pc1, pc2 IN ProblemFrameConcept, forall p IN Problem: (pc1 IncludedIn p OR pc1 IsProjectionOf p) AND (pc2 IncludedIn p OR pc2 IsProjectionOf p) AND pc1.Name = pc2.Name <=> pc1 = pc2

Constraints about abbreviations
4) for all td1, td2 IN TangibleDescription, forall p IN ProblemFrame : td1 IncludedIn p AND td2 IncludedIn p AND td1.Abbr = td2.Abbr <=> td1 = td2
5) forall td IN TangibleDescription : td.Abbr.Length = 2

Figure 16: Constraints relating to figure 15

- Names spaces for organizing and grouping problem frames should be defined.
- The global name space should be replaced by several 'project' name spaces: each one should contain a set of phenomena global to a project, plus a root problem.

10 Conclusion

We think that this metamodel covers an important part of the concepts found in Jackson's main notations (context, problem and frame diagrams). Two main aspects must still be investigated:

1. the relations between the different kinds of phenomena (that should enrich section 4);
2. the types of phenomena sets, and the information that tells which kinds of phenomena can be referenced in which kinds of phenomena sets and domains (according to their type). This is needed to complete the semantic of a fit between a frame and a problem (section 8).

A lot of the other ideas of Jackson around problem frames[11], that are expressed in a less precise way (compared to the ideas that are encapsulated in the three main notations) are not yet taken into account. The most important are the following.

1. The possibility to use *more specific and adapted notations* for the different descriptions found in a problem. Jackson says that no method is a panacea —a medicine that cures all diseases [1, p. 4]. A method is helpful for a particular class or problems —a problem frame. So each problem frame has (ideally) an associated method, that, among other things, proposes a

[11] In fact, the most part of it.

set of specific notations to use for the different descriptions that must be made.

2. We already introduced *frame flavours* [2, p. 143], or domain characteristics [1, pp. 66–70], that give more details about domain classifications.

3. A *frame concern* identifies, for a particular problem frame, *the descriptions you must make and how you must fit them together in a correctness argument [2, p. 103].* A frame concern is a kind of 'proof obligation': you must address it in order to 'prove' that you correctly analyzed a problem.

4. When analyzing a problem, in addition to the frame concern, other *particular concerns* [2, pp. 237–268] must often be addressed —like initialization, completeness or reliability.

5. The interactions among subproblems in a decomposed problem give rise to *composition concerns* [2, pp. 301–332]: are the subproblems consistent ? Do they interfere with each other ? etc.

6. A *frame variant* is an extension of a basic frame. *Typically —but not necessarily— a variant adds a domain to the problem context [2, p. 205].*

We think that, in a next step, we could incorporate ideas 1 and 2 in the metamodel, through:

1. the introduction of a mechanism to allow the integration of external notations or languages via the extension of the `Phenomena` class hierarchy —discussed in section 4;

2. the introduction of domain type 'dimensions' —discussed in section 3.

Finally, we should try to explore the possibilities to define with more precision the notion of projection, and the possibility to apply it consistently to both problem decomposition and problem frame fitting (see section 8).

References

[1] Michael Jackson. *Software Requirements and Specification: a lexicon of practice, principles and prejudices.* Addison-Wesley, 1995.

[2] Michael Jackson. *Problem Frames: analyzing and structuring software development problems.* Addison-Wesley, 2001.

Annexe B

Cas d'utilisation

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'Informatique
Année académique 2004 - 2005

Conception d'un outil d'aide à l'analyse par la méthode des Problem Frames de Jackson

Michel Gonze et Federico Martinez

Mémoire présenté en vue de l'obtention du grade de Licencié en Informatique.

Nous nous sommes basés sur la description des cas d'utilisation faite dans [Cockburn].

1.1 Acteurs

1.1.1 Acteur principal

1.1.1.1 Analyste

Analyste : dans le cadre d'un projet informatique, l'analyste souhaite spécifier les exigences d'un logiciel en utilisant l'approche des « Problem frames » de M. Jackson. Il veut disposer d'un outil lui permettant de présenter un dossier reprenant les exigences exprimées selon cette approche.

1.1.2 Acteur de second plan

1.1.2.1 Editeur Externe (notation)

Responsabilité : fournir un fichier dans une notation particulière, compatible avec les exigences du cahier des charges.

1.2 Cas d'utilisation

1.2.1 CAS D'UTILISATION – Spécifier les exigences d'un problème

Brève description
Ce cas d'utilisation décrit et indique les opérations permettant de gérer les problèmes c-à-d créer, modifier, supprimer ou enregistrer des problèmes.
Acteurs
Analyste
Déclencheurs
L'analyste choisit explicitement les opérations à l'aide de l'interface
Pré conditions
Le système est actif
Flot d'événements de base – Ouvrir, modifier, enregistrer un problème
1. L'analyste sélectionne un problème et sélectionne « Ouvrir un problème». 2. Le système affiche le problème à l'écran. Il affiche, dans une fenêtre, tous les éléments constitutifs sous forme d'arborescence. Il affiche également, dans une autre fenêtre, sous forme graphique correspondant à la notation de M. Jackson, le problème principal. 3. Le système offre la possibilité à l'analyste d'effectuer les actions suivantes sur le problème principal ou un sous-problème : a. Ajouter, supprimer ou modifier la spécification (cf. cas d'utilisation Gérer la spécification) b. Ajouter, supprimer ou modifier un domaine (cf. Cas d'utilisation : Gérer les domaines) c. Ajouter, supprimer ou modifier une interface (cf. Cas d'utilisation : Gérer les interfaces) d. Ajouter, supprimer ou modifier l'exigence (cf. Cas d'utilisation : Gérer l'exigence) e. Ajouter, supprimer ou modifier une RequirementReference (cf. Cas d'utilisation : Gérer les références à l'exigence) f. Créer un sous-problème, projection du problème courant (cf. Cas d'utilisation : Créer une projection) g. Créer un sous-problème à partir d'un ProblemFrame (cf. Cas d'utilisation : Créer un problème à partir d'un ProblemFrame) h. Valider un problème (cf. Cas d'utilisation : Valider un problème) i. Contrôler la correspondance entre un problème ou sous-problème et un ProblemFrame (cf. Cas d'utilisation : Faire correspondre un problème avec un ProblemFrame) L'analyste répète l'étape 3 autant de fois que nécessaire. 4. L'analyste demande l'enregistrement du problème. 5. Le système enregistre le problème dans l'espace de stockage. 6. Le cas d'utilisation se termine quand l'analyste demande de fermer le problème ou de quitter le système.
Flots alternatifs
(Etape 1) Créer un nouveau problème a. L'analyste sélectionne « Créer un nouveau problème ». Le système crée un

nouveau problème avec un nom par défaut.

b. Le flot se poursuit avec l'étape 2 du flot principal.

(Etape 1) Supprimer un problème

a. L'analyste sélectionne un problème dans l'explorateur puis sélectionne « Supprimer un problème ».

b. Le système affiche le problème (cf. flot de base étape 2) et demande à l'analyste de confirmer la suppression.

c. Le système supprime le problème du système de stockage.

(Etape 1) Créer un problème à partir d'un ProblemFrame

a. L'analyste sélectionne « Créer un problème à partir d'un ProblemFrame». (cf. cas d'utilisation Créer un problème à partir d'un ProblemFrame).

b. Le système affiche la liste des cadres de problème.

c. L'analyste sélectionne un ProblemFrame et sélectionne « Créer le problème ».

d. Le système demande le nouveau nom du problème et crée le problème principal en copiant le ProblemFrame. Le système copie les éléments et leurs caractéristiques. Le système fait correspondre le problème principal au ProblemFrame (« fitting »).

e. Le flot se poursuit avec l'étape 2 du flot principal.

(Etape 1) Copier un problème

a. L'analyste sélectionne un problème dans l'explorateur puis sélectionne « Copier un problème ».

b. Le système demande à l'analyste d'indiquer le nouveau nom du problème.

c. Le flot se poursuit avec l'étape 2 du flot principal.

(Etape 6) Le problème a été modifié mais n'a pas été enregistré

a. Le système propose à l'analyste de sauvegarder avant de quitter.

b.1 L'analyste demande de sauvegarder, le système sauvegarde et le cas d'utilisation se termine.

b.2 L'analyste choisit de quitter sans sauvegarder et le cas d'utilisation se termine.

Exigences particulières

(Etape 3 du flot de base)

a. L'analyste navigue à travers l'arborescence et sélectionne un élément dans l'arborescence ou sélectionne directement un élément dans la fenêtre graphique.

b. Le système affiche la fenêtre de propriété spécifique à l'élément sélectionné ainsi que la représentation graphique du problème (principal ou sous-problème) du niveau associé à l'élément. Les propriétés de cet élément peuvent être mises à jour.

(Etape 3 du flot de base)

Toute modification apportée à un problème fait passer son statut à « non valide ».

Post conditions

Le problème a été mis à jour

1.2.2 CAS D'UTILISATION – Valider un problème

Brève description
Ce cas d'utilisation permet de valider un problème et, le cas échéant, d'afficher ou d'imprimer les erreurs de validation.
Acteurs
Analyste
Déclencheurs
L'analyste choisit explicitement les opérations à l'aide de l'interface
Pré conditions
Le système est actif
Flot d'événements de base
1. L'analyste sélectionne un problème dans l'explorateur et sélectionne « Valider un problème ». 2. Le système vérifie si le problème respecte bien les contraintes exprimées dans le métamodèle. 3. Le système signifie à l'analyste que le problème est valide et fait passer le statut du problème à « valide ». 4. L'analyste prend connaissance du statut et le cas d'utilisation se termine.
Flots alternatifs
(Etape 2) Le statut du problème sélectionné est « valide » a. Le système signifie à l'analyste que le problème est valide et le cas d'utilisation se termine. **(Etape 3) Erreur(s) de validation** a. Le système affiche les erreurs de validation. Chaque erreur reprend la contrainte qui n'est pas respectée ainsi que le ou les élément(s) associé(s). Le système permet à l'utilisateur d'imprimer un rapport contenant les erreurs de validation. b. Le statut du problème reste à « non valide » et le cas d'utilisation se termine.
Exigences particulières
néant
Post conditions
Le statut du problème est « valide » ou l'analyste a pris connaissance des erreurs de validation.

1.2.3 CAS D'UTILISATION – Gérer la spécification

Brève description
Ce cas d'utilisation décrit les opérations permettant d'ajouter, supprimer ou modifier une spécification.
Acteurs
Analyste
Déclencheurs
L'analyste sélectionne les opérations dans le cas d'utilisation « Gérer des problèmes » ou « Gérer des cadres de problème »
Pré conditions
Un problème principal, un sous-problème ou un ProblemFrame a été sélectionné comme élément courant.
Flot d'événements de base – Créer la spécification
1. L'analyste sélectionne « Créer la spécification ». 2. Le système permet à l'analyste de spécifier les informations suivantes : nom, description, préfixe 3. L'analyste fournit l'information et demande de créer la spécification. 4. Le système crée la spécification et ajoute un nouvel élément « spécification » dans l'arborescence et la fenêtre graphique. La spécification est sélectionnée comme élément courant (le système rend donc disponible à l'édition les propriétés de la spécification). 5. Le système crée également le scope associé à la spécification. 6. Le système offre la possibilité de modifier le scope de la spécification (cf. Cas d'utilisation – Modifier le scope d'une description) 7. Le cas d'utilisation se termine quand l'analyste sélectionne un autre élément.
Flot d'événements de base – Modifier la spécification
1. L'analyste sélectionne la spécification. 2. Le système sélectionne la spécification comme élément courant et le cas d'utilisation se poursuit comme dans le « flot de base - Créer la spécification » à partir de l'étape 6.
Flot d'événements de base – Supprimer la spécification
1. L'analyste sélectionne la spécification et sélectionne « Supprimer la spécification ». 2. Le système demande confirmation de la suppression de la spécification et de tous les éléments associés à la spécification. 3. L'analyste confirme la suppression. 4. Le système supprime la spécification, le scope de la spécification, les interfaces attachées à la spécification ainsi que les ensembles de phénomènes référencés par ces interfaces (y compris les phénomènes inclus dans les ensembles de phénomènes contrôlés par la spécification).
Flots alternatifs – Créer la spécification
(Etape 2) L'élément sélectionné possède déjà une spécification. Le système signifie à l'analyste que le problème principal, le sous-problème ou le ProblemFrame sélectionné possède déjà une spécification et le cas d'utilisation se termine **(Etape 3) L'élément sélectionné est un sous-problème.** a. Le système permet à l'analyste de spécifier les informations suivantes : nom,

description et affiche la liste des domaines du problème de niveau supérieur dont le sous-problème est une projection. b. L'analyste fournit l'information, sélectionne, si nécessaire, le ou les domaines dont il estime que la spécification doit être la projection et demande de créer la spécification. c. Le système crée la spécification et établit une relation de projection avec la spécification de niveau supérieur mais, également, avec le ou les domaines sélectionnés à l'étape b ci-dessus. La spécification est sélectionnée comme élément courant. d. Le cas d'utilisation se poursuit comme dans le « flot de base - Créer la spécification » à partir de l'étape 5.
Flots alternatifs –Supprimer la spécification
(Etape 3) L'analyste ne confirme pas la suppression. Le cas d'utilisation se termine.
Exigences particulières
Néant
Post conditions
[en cas de succès] [*en cas de création ou de modification*] La spécification a été mise à jour [*en cas de suppression*] La spécification et toute référence à la spécification (scope, interface(s) et ensembles de phénomènes référencés (y compris les phénomènes inclus dans les ensembles de phénomènes contrôlés par la spécification) sont supprimées. **[en cas d'échec]** pas de modification à l'élément sélectionné

1.2.4 CAS D'UTILISATION – Gérer les domaines

Brève description
Ce cas d'utilisation décrit les opérations permettant d'ajouter, supprimer ou modifier un domaine.
Acteurs
Analyste
Déclencheurs
L'analyste sélectionne les opérations dans le cas d'utilisation « Gérer des problèmes » ou « Gérer des cadres de problème »
Pré conditions
Un problème principal, un sous-problème ou un ProblemFrame a été sélectionné comme élément courant.
Flot d'événements de base – Créer un domaine
1. L'analyste sélectionne « Créer un domaine ». 2. Le système permet à l'analyste de spécifier les informations suivantes : nom, description, préfixe, type, *designed*, *fragile*, *Structure* 3. L'analyste fournit l'information et demande de créer le domaine. 4. Le système crée le domaine et ajoute un nouvel élément « domaine » dans l'arborescence et la fenêtre graphique. Le domaine est sélectionné comme élément courant (le système rend donc disponible à l'édition les propriétés du domaine). 5. Le système crée également le scope associé au domaine. 6. Le système offre la possibilité de modifier le scope du domaine (cf. Cas d'utilisation – Modifier le scope d'une description) 7. Le cas d'utilisation se termine quand l'analyste sélectionne un autre élément.
Flot d'événements de base – Modifier un domaine
1. L'analyste sélectionne le domaine. 2. Le système sélectionne le domaine comme élément courant et le cas d'utilisation se poursuit comme dans le « flot de base - Créer un domaine » à partir de l'étape 6.
Flot d'événements de base – Supprimer un domaine
1. L'analyste sélectionne le domaine et sélectionne « Supprimer un domaine ». 2. Le système demande confirmation de la suppression du domaine et de tous les éléments associés à ce domaine. 3. L'analyste confirme la suppression. 4. Le système supprime le domaine, le scope du domaine (y compris les phénomènes contenus dans ce scope à l'exception des phénomènes inclus dans un des ensembles de phénomènes référencés par une des interfaces ci-après non contrôlés par le domaine), les interfaces et références à l'exigence attachées au domaine ainsi que les ensembles de phénomènes référencés par ces interfaces.
Flots alternatifs – Créer un domaine
(Etape 2) L'élément sélectionné est un sous-problème. a. Le système permet à l'analyste de spécifier les informations suivantes : nom, description, type, *designed*, *fragile*, *Structure* et affiche une liste reprenant la spécification et les domaines du problème de niveau supérieur dont le sous-problème est une projection.

b. L'analyste fournit l'information, sélectionne le ou les éléments de la liste dont il estime que le domaine doit être la projection et demande de créer le domaine. c. Le système crée le domaine et établit une relation de projection avec le(s) élément(s) de la liste sélectionné(s) à l'étape b ci-dessus. Le domaine est sélectionné comme élément courant. d. Le cas d'utilisation se poursuit comme dans le « flot de base - Créer un domaine » à partir de l'étape 5.
Flots alternatifs – Supprimer un domaine
(Etape 3) L'analyste ne confirme pas la suppression. Le cas d'utilisation se termine.
Exigences particulières
Néant
Post conditions
[en cas de succès] [*en cas de création ou de modification*] Le domaine a été mis à jour [*en cas de suppression*] Le domaine et toute référence au domaine (scope et phénomènes y inclus à l'exception des phénomènes inclus dans un des ensembles de phénomènes référencés par une des interfaces ci-après et non contrôlés par le domaine, interface(s) et ensembles de phénomènes référencés, références et ensembles de phénomènes référencés) sont supprimées. **[en cas d'échec]** pas de modification à l'élément sélectionné.

1.2.5 CAS D'UTILISATION – Gérer les interfaces

Brève description
Ce cas d'utilisation décrit les opérations permettant d'ajouter, supprimer ou modifier une interface.
Acteurs
Analyste
Déclencheurs
L'analyste sélectionne les opérations dans le cas d'utilisation « Gérer des problèmes » ou « Gérer des cadres de problème »
Pré conditions
Un problème principal, un sous-problème ou un ProblemFrame a été sélectionné comme élément courant.
Flot d'événements de base – Créer une interface
1. L'analyste sélectionne « Créer une interface ». 2. Le système offre un dialogue permettant à l'analyste de spécifier successivement les informations suivantes : a. caractéristiques de l'interface : nom, description, annotation b. descriptions tangibles connectées à l'interface : le système affiche une liste reprenant la spécification et les domaines de l'élément sélectionné (problème principal, sous-problème ou ProblemFrame). Au minimum 2 descriptions doivent être sélectionnées. c. Ensemble(s) de phénomènes référencé(s) par l'interface : pour chaque description tangible sélectionnée, l'analyste a la possibilité de définir un ensemble de phénomènes contrôlé par cette description ; il fournit ses nom et description et la liste des phénomènes sélectionnés dans le scope de la description. 3. L'analyste fournit l'information et demande de créer l'interface. 4. Le système crée l'interface et ajoute un nouvel élément « interface » dans l'arborescence et la fenêtre graphique (reliant les descriptions tangibles sélectionnées). L'interface est sélectionnée comme élément courant (le système rend donc disponible à l'édition les propriétés de l'interface). 5. Le système crée également le ou les ensemble(s) de phénomènes défini(s) par l'analyste et intègre, dans le scope des descriptions tangibles, les phénomènes contrôlés par les autres descriptions. 6. Le cas d'utilisation se termine quand l'analyste sélectionne un autre élément.
Flot d'événements de base – Modifier une interface
1. L'analyste sélectionne l'interface et sélectionne « Modifier ». 2. Le système sélectionne l'interface comme élément courant et propose le dialogue décrit au point 2 du flot de base – Créer une interface. 3. L'analyste modifie l'information proposée et demande de modifier l'interface. 4. Le système met à jour l'arborescence et la fenêtre graphique. 5. Le système effectue plus spécifiquement les opérations suivantes en fonction des choix de l'analyste : a. l'analyste a connecté une nouvelle description tangible et a défini un ensemble de phénomènes contrôlé par cette description : le système crée l'ensemble de phénomènes et intègre, dans le scope des autres descriptions tangibles connectées, les phénomènes inclus dans cet ensemble. b. l'analyste a supprimé la connexion avec une description tangible qui

contrôlait un ensemble de phénomènes référencé par cette interface : le système supprime l'ensemble de phénomènes et enlève les phénomènes inclus dans cet ensemble des scopes des autres descriptions connectées à l'interface. 6. Le cas d'utilisation se termine quand l'analyste sélectionne un autre élément.
Flot d'événements de base – Supprimer une interface
1. L'analyste sélectionne l'interface et sélectionne « Supprimer un interface ». 2. Le système demande confirmation de la suppression de l'interface et de tous les éléments associés à cette interface. 3. L'analyste confirme la suppression. 4. Le système supprime l'interface et les ensembles de phénomènes référencés par l'interface et le cas d'utilisation se termine.
Flots alternatifs – Créer une interface
(Etape 2c) L'élément sélectionné est un ProblemFrame. Le système offre un dialogue permettant à l'analyste de spécifier le(s) ensemble(s) de phénomènes référencé(s) par l'interface : pour chaque description tangible sélectionnée, l'analyste a la possibilité de définir un ensemble de phénomènes contrôlé par cette description ; il fournit ses nom et description. Dans le cas d'un ProblemFrame, les ensembles de phénomènes sont tous vides. Le cas d'utilisation se poursuit avec l'étape 3 du flot de base- Créer une interface. **(Etape 5) L'élément sélectionné est un ProblemFrame.** Le système crée également le ou les ensemble(s) de phénomènes défini(s) par l'analyste et le cas d'utilisation se poursuit avec l'étape 6 du flot de base- Créer une interface.
Flots alternatifs – Supprimer une interface
(Etape 3) L'analyste ne confirme pas la suppression. Le cas d'utilisation se termine.
Exigences particulières
Néant
Post conditions
[en cas de succès] [*en cas de création ou de modification*] L'interface a été mise à jour [*en cas de suppression*] L'interface, les ensembles de phénomènes référencés par l'interface sont supprimés. Les phénomènes (pas dans le cas d'un ProblemFrame) inclus dans ces ensembles sont enlevés des scopes des descriptions tangibles qui ne contrôlent pas les ensembles. **[en cas d'échec]** pas de modification à l'élément sélectionné.

1.2.6 CAS D'UTILISATION – Gérer les références à l'exigence

Brève description
Ce cas d'utilisation décrit les opérations permettant d'ajouter, supprimer ou modifier une RequirementReference.
Acteurs
Analyste
Déclencheurs
L'analyste sélectionne les opérations dans le cas d'utilisation « Gérer des problèmes » ou « Gérer des cadres de problème »
Pré conditions
Un problème principal, un sous-problème ou un ProblemFrame a été sélectionné comme élément courant. L'exigence est définie pour l'élément sélectionné.
Flot d'événements de base – Créer une référence
1. L'analyste sélectionne « Créer une référence ». 2. Le système permet à l'analyste de spécifier les informations suivantes : nom, description, annotation, *constraining* et affiche une liste des domaines non connectés à l'exigence. 3. L'analyste fournit l'information et sélectionne le domaine qui doit être connecté à l'exigence. 4. L'analyste définit les ensembles de phénomènes qui seront référencés par la référence de 2 manières possibles: 4.a. Il les sélectionne parmi les ensembles de phénomènes, référencés par les interfaces connectées au domaine, affichés par le système. 4.b. Il définit un nouvel ensemble de phénomènes (ou plusieurs); il fournit ses nom et description et la liste des phénomènes sélectionnés dans le scope du domaine. 5. L'analyste demande de créer la référence. 6. Le système crée la référence et ajoute un nouvel élément « référence » dans l'arborescence et la fenêtre graphique (reliant l'exigence et le domaine sélectionné). La référence est sélectionnée comme élément courant. 7. Le système crée également les nouveaux ensembles défini(s) par l'analyste. 8. Le cas d'utilisation se termine quand l'analyste sélectionne un autre élément.
Flot d'événements de base – Modifier une référence
1. L'analyste sélectionne la référence et sélectionne « Modifier ». 2. Le système sélectionne la référence comme élément courant (le système rend donc disponible à l'édition les propriétés du domaine). 3. Le système permet à l'analyste d'ajouter (cf. étape 4 du flot de base – Créer une référence) ou de supprimer un ensemble de phénomène de la liste. Notez qu'enlever de la liste un ensemble créé dans l'étape 4.b. du flot de base entraîne la suppression de cet ensemble. 4. Le cas d'utilisation se poursuit comme dans le flot de base – Créer une référence à partir de l'étape 7.
Flot d'événements de base – Supprimer une référence
1. L'analyste sélectionne le référence et sélectionne « Supprimer une référence ». 2. Le système demande confirmation de la suppression de la référence et de tous les éléments associés à cette référence. 3. L'analyste confirme la suppression.

4. Le système supprime la référence et les ensembles de phénomènes non référencés par les interfaces connectées au domaine et le cas d'utilisation se termine.
Flots alternatifs – Supprimer une référence
(Etape 3) L'analyste ne confirme pas la suppression. Le cas d'utilisation se termine.
Références particulières
Néant
Post conditions
[en cas de succès] [*en cas de création ou de modification*] La référence a été mise à jour [*en cas de suppression*] La référence, les ensembles de phénomènes référencés par la référence (et pas par les interfaces connectées au domaine) sont supprimés. **[en cas d'échec]** pas de modification à l'élément sélectionné.

1.2.7 CAS D'UTILISATION – Gérer l'exigence

Brève description
Ce cas d'utilisation décrit les opérations permettant d'ajouter, supprimer ou modifier l'exigence.
Acteurs
Analyste
Déclencheurs
L'analyste sélectionne les opérations dans le cas d'utilisation « Gérer des problèmes » ou « Gérer des cadres de problème »
Pré conditions
Un problème principal, un sous-problème ou un ProblemFrame a été sélectionné comme élément courant.
Flot d'événements de base – Créer l'exigence
1. L'analyste sélectionne « Créer l'exigence ». 2. Le système permet à l'analyste de spécifier les informations suivantes : nom, description 3. L'analyste fournit l'information et demande de créer l'exigence. 4. Le système crée l'exigence et le scope associée à l'exigence. Il ajoute un nouvel élément « exigence » dans l'explorateur et la fenêtre graphique. L'exigence est sélectionnée comme élément courant (le système rend donc disponible à l'édition les propriétés de l'exigence). 5. Le cas d'utilisation se termine quand l'analyste sélectionne un autre élément.
Flot d'événements de base – Modifier l'exigence
1. L'analyste sélectionne l'exigence. 2. Le système sélectionne l'exigence comme élément courant.
Flot d'événements de base – Supprimer l'exigence
1. L'analyste sélectionne l'exigence et sélectionne « Supprimer l'exigence ». 2. Le système demande confirmation de la suppression de l'exigence et de tous les éléments associés à l'exigence. 3. L'analyste confirme la suppression. 4. Le système supprime l'exigence, le scope de l'exigence, les références à l'exigence ainsi que les ensembles de phénomènes référencés uniquement par ces références.
Flots alternatifs – Créer l'exigence
(Etape 2) L'élément sélectionné possède déjà une exigence. Le système signifie à l'analyste que le problème principal, le sous-problème ou le ProblemFrame sélectionné possède déjà une exigence et le cas d'utilisation se termine. **(Etape 3) L'élément sélectionné est un sous-problème.** a. Le système permet à l'analyste de spécifier les informations suivantes : nom, description. b. L'analyste fournit l'information et demande de créer l'exigence. c. Le système crée l'exigence et le scope associé à l'exigence. Il établit une relation de projection avec l'exigence du problème de niveau supérieur. L'exigence est sélectionnée comme élément courant.
Flots alternatifs – Supprimer l'exigence
(Etape 3) L'analyste ne confirme pas la suppression.

Le cas d'utilisation se termine.
Exigences particulières
Néant
Post conditions
[en cas de succès] [*en cas de création ou de modification*] L'exigence a été mise à jour. [*en cas de suppression*] L'exigence et tous les éléments associés à l'exigence sont supprimés (scope de l'exigence, références à l'exigence, ensembles de phénomènes non référencés par une interface). **[en cas d'échec]** pas de modification à l'élément sélectionné

1.2.8 CAS D'UTILISATION – Modifier le scope d'une description

Brève description
Ce cas d'utilisation décrit les opérations permettant de modifier le scope d'une description. Il définit également les différences en fonction du contexte : problème principal ou sous-problème.
Acteurs
Analyste
Déclencheurs
L'analyste sélectionne l'opération dans les cas d'utilisation « Gérer la spécification » ou « Gérer des domaines » ou « Gérer l'exigence »
Pré conditions
Un domaine, une spécification ou une exigence a été sélectionné comme élément courant.
Flot d'événements de base
1. L'analyste sélectionne « Modifier le scope ». 2. Le système affiche la liste des phénomènes inclus dans le scope. Le système permet à l'analyste d'ajouter ou de supprimer un phénomène de le scope et de modifier un phénomène. Il permet également d'enrichir le scope de la description avec un objet externe (cf. Cas d'utilisation – Importer un objet externe). 3. L'analyste sélectionne « Ajouter un phénomène ». 4. Le système permet à l'analyste de spécifier les informations suivantes concernant le phénomène: nom, description, type de phénomène (choix entre Event, Entity,Value, State, CausalState, SymbolicState, Truth, Role). 5. L'analyste fournit l'information et choisit le type State, CausalState, SymbolicState, Truth ou Role : le système affiche une boîte dialogue pour exprimer le phénomène de type Relation (cf. Cas d'utilisation détail – Exprimer un phénomène de type Relation) 6. Le système crée le phénomène et l'ajoute à le scope de la description. 7. Les étapes 3 à 5 se répètent jusqu'à ce que l'analyste soit satisfait. 8. L'analyste signale au système qu'il a terminé les modifications et le cas d'utilisation se termine.
Flots alternatifs
(Etape 4) La modification se fait sur le scope d'une description d'un sous-problème a. Le système permet d'ajouter un phénomène de deux manières possibles : a.1 ajout d'un phénomène du niveau supérieur a.1.1. L'analyste sélectionne une description et le système affiche une liste reprenant les phénomènes inclus dans le scope de la description sélectionnée et non déjà inclus dans le scope à modifier. L'analyste sélectionne un phénomène. a.1.2. Le système ajoute le phénomène à le scope et le cas d'utilisation se poursuit comme dans le flot de base à partir de l'étape 6. a.2 création d'un nouveau phénomène a.2.1. Le système permet à l'analyste de créer un nouveau phénomène en spécifiant les informations suivantes : nom, description, type de phénomène (choix entre Event, Entity,Value, State,

CausalState, SymbolicState, Truth, Role) .

a.2.2. Le cas d'utilisation se poursuit comme dans le flot de base à partir de l'étape 5.

(Etape 3) L'analyste veut modifier un phénomène.

a. L'analyste sélectionne un phénomène dans la liste et sélectionne « modifier ».

b. Le système permet à l'analyste de modifier les informations suivantes concernant le phénomène: nom, description, type de phénomène (choix entre Event, Entity,Value, State, CausalState, SymbolicState, Truth, Role).

c. Le cas d'utilisation se poursuit comme dans le flot de base à partir de l'étape 5.

(Etape 3) L'analyste veut supprimer un phénomène.

a. L'analyste sélectionne un phénomène dans la liste et sélectionne « supprimer ».

b. Le système vérifie si le phénomène fait partie d'un autre ensemble de phénomène ou si le phénomène est référencé par un autre phénomène de type Relation.

c. Deux alternatives :

- le phénomène est utilisé par un autre élément : le système signale à l'analyste qu'il ne peut supprimer le phénomène et le cas d'utilisation se poursuit comme dans le flot de base à partir de l'étape 5.
- le phénomène n'est pas référencé par un autre élément : le système demande confirmation de la suppression.

d. L'analyste confirme la suppression.

e. Le système supprime le phénomène le cas d'utilisation se poursuit comme dans le flot de base à partir de l'étape 5.

(Etape 5) L'analyste choisit le type de phénomène : Event, Entity ou Value

Le cas d'utilisation se poursuit comme dans le flot de base à partir de l'étape 6.

Exigences particulières

néant

Post conditions

Le scope a été mise à jour.

1.2.9 CAS D'UTILISATION – Créer une projection

Brève description
Ce cas d'utilisation décrit et indique les opérations permettant de créer un sous-problème.
Acteurs
Analyste
Déclencheurs
L'analyste sélectionne l'opération dans les cas d'utilisation « Spécifier les exigences d'un problème ».
Pré conditions
Le système est actif
Flot d'événements de base – Ouvrir, modifier, enregistrer un problème
1. L'analyste sélectionne le problème principal ou un sous-problème et sélectionne « Créer une projection ». 2. Le système permet à l'analyste de spécifier les informations suivantes : nom et description. 3. L'analyste fournit l'information et demande de créer la projection. 4. Le système crée le sous-problème, ajoute un nouveau élément « sous-problème » à l'arborescence (niveau directement inférieur par rapport au problème principal ou au sous-problème à partir duquel s'est effectuée la projection) et établit la relation de projection avec le problème de niveau supérieur. 5. Le système sélectionne le nouveau sous-problème comme élément courant et le cas d'utilisation se termine.
Exigences particulières
La relation de projection entre un problème et le sous-problème impose toute une série de contraintes lors de la modification de ce sous-problème. Celles-ci sont détaillées dans les cas d'utilisation décrivant les modifications d'un problème.
Post conditions
Le sous-problème, projection d'un problème de niveau supérieur a été créé.

1.2.10 CAS D'UTILISATION – Créer un problème à partir d'un ProblemFrame

Brève description
Ce cas d'utilisation décrit les opérations permettant de créer un problème ou sous-problème à partir d'un ProblemFrame. Il décrit également les opérations et les contraintes de modification liées à de tels problèmes ou sous-problèmes.
Acteurs
Analyste
Déclencheurs
L'analyste sélectionne « Créer un problème à partir d'un ProblemFrame » ou « Créer un sous-problème à partir d'un ProblemFrame » dans le cas d'utilisation « Spécifier les exigences d'un problème ».
Pré conditions
Le système est actif.
Flot d'événements de base – Créer un sous-problème
1. Le système affiche la liste des cadres de problème. 2. L'analyste sélectionne un ProblemFrame dans la liste et demande de créer le problème. 3. Le système demande à l'analyste d'indiquer le nouveau nom du problème. 4. Le système crée un problème copie du ProblemFrame sélectionné et sélectionne le problème comme élément courant.. 5.
Flots alternatifs – Créer un sous-problème
(Etape 4) Un élément du sous-problème est créé avec un nom qui existe déjà. Le système demande à l'analyste d'indiquer le nouveau nom et le cas d'utilisation se poursuit comme dans le flot de base.
Exigences particulières
La relation de correspondance entre le problème et le sous-problème impose toute une série de contraintes lors de modifications ultérieures. Celles-ci sont détaillées dans les cas d'utilisation décrivant les modifications d'un problème.
Post conditions
Le problème principal ou sous-problème a été mis à jour et correspond (« fit ») au ProblemFrame.

1.2.11 CAS D'UTILISATION – Gérer les cadres de problème

Brève description
Ce cas d'utilisation décrit et indique les opérations permettant de gérer les cadres de problèmes càd créer, modifier, supprimer, enregistrer des cadres de problème.
Acteurs
Analyste
Déclencheurs
L'analyste choisit explicitement les opérations à l'aide de l'interface
Pré conditions
Le système est actif
Flot d'événements de base – Ouvrir, modifier, enregistrer un problème
1. L'analyste sélectionne « Cadres de problème ». 2. Le système affiche la liste des cadres de problème existant. 3. L'analyste sélectionne un ProblemFrame et sélectionne « Ouvrir ». 4. Le système affiche le cadre problème à l'écran. Il affiche, dans une fenêtre, tous les éléments constitutifs sous forme d'arborescence. Il affiche également, dans une autre fenêtre, sous forme graphique correspondant à la notation de M. Jackson, le problème principal. 5. Le système offre la possibilité à l'analyste d'effectuer les actions suivantes sur le ProblemFrame : a. Ajouter, supprimer ou modifier la spécification (cf. cas d'utilisation Gérer la spécification) b. Ajouter, supprimer ou modifier un domaine (cf. Cas d'utilisation : Gérer les domaines) c. Ajouter, supprimer ou modifier une interface (cf. Cas d'utilisation : Gérer les interfaces) d. Ajouter, supprimer ou modifier l'exigence (cf. Cas d'utilisation : Gérer l'exigence) e. Ajouter, supprimer ou modifier une RequirementReference (cf. Cas d'utilisation : Gérer les références à l'exigence) f. Valider un ProblemFrame (cf. Cas d'utilisation : Valider un problème) 6. L'analyste répète l'étape 5 jusqu'à ce qu'il soit satisfait. 7. L'analyste demande l'enregistrement du ProblemFrame. 8. Le système enregistre le ProblemFrame dans l'espace de stockage.
Flots alternatifs
(Etape 3) Créer un nouveau ProblemFrame a. L'analyste sélectionne « Créer un nouveau ProblemFrame ». Le système crée un nouveau ProblemFrame avec un nom par défaut. b. Le flot se poursuit avec l'étape 4 du flot principal. **(Etape 3) Supprimer un ProblemFrame** a. L'analyste sélectionne un ProblemFrame puis sélectionne « Supprimer un ProblemFrame ». b. Le système affiche le ProblemFrame (cf. flot de base étape 2) et demande à l'analyste de confirmer la suppression. c. Le système supprime le problème du système de stockage et réinitialise ces fenêtres. **(Etape 3) Copier un ProblemFrame** a. L'analyste sélectionne un ProblemFrame puis sélectionne « Copier un

ProblemFrame ». b. Le système demande à l'analyste d'indiquer le nouveau nom du ProblemFrame. c. Le flot se poursuit avec l'étape 4 du flot principal. **(Etape 6) Le ProblemFrame a été modifié mais n'a pas été enregistré** b. Le système propose à l'analyste de sauvegarder avant de quitter. b.1 L'analyste demande de sauvegarder, le système sauvegarde et le cas d'utilisation se termine. b.2 L'analyste choisit de quitter sans sauvegarder et le cas d'utilisation se termine.
Exigences particulières
(Etape 3 du flot de base) a. L'analyste navigue à travers l'arborescence et sélectionne un élément dans l'arborescence ou sélectionne directement un élément dans la fenêtre graphique. b. Le système affiche la fenêtre de propriété spécifique à l'élément sélectionné. Les propriétés de l'élément peuvent être mises à jour. (Etape 3 du flot de base) Toute modification apportée à un ProblemFrame fait passer son statut à « non validé ».
Post conditions
Le ProblemFrame a été mis à jour

1.2.12 CAS D'UTILISATION – Faire correspondre un problème à un ProblemFrame (« Fitting »)

Brève description
Ce cas d'utilisation permet de faire correspondre un problème à un ProblemFrame et, le cas échéant, d'afficher ou d'imprimer les erreurs de correspondance (fitting).
Acteurs
Analyste
Déclencheurs
L'analyste sélectionne l'opération dans les cas d'utilisation « Spécifier les exigences du problème »
Pré conditions
Un problème est ouvert..
Flot d'événements de base
1. L'analyste sélectionne le problème principal ou un sous-problème et sélectionne « Faire correspondre le problème». 2. Le système affiche la liste des cadres de problèmes. 3. L'analyste sélectionne un ProblemFrame et demande au système de vérifier la correspondance entre le problème et le ProblemFrame. 4. Le système vérifie si les règles de correspondance entre le problème et le ProblemFrame sont bien respectées. 5. Le système signifie à l'analyste que le problème correspond au ProblemFrame et établit la relation de correspondance au ProblemFrame . 6. L'analyste prend connaissance du statut de correspondance et le cas d'utilisation se termine.
Flots alternatifs
(Etape 4) Le statut du problème correspond déjà au ProblemFrame. a. Le système signifie à l'analyste que le problème correspond déjà au ProblemFrame sélectionné et le cas d'utilisation se termine. **(Etape 5) Erreur(s) de correspondance** a. Le système affiche les erreurs de correspondance. Chaque erreur reprend la règle de correspondance qui n'est pas respectée ainsi que le ou les élément(s) associé(s). Le système permet à l'utilisateur d'imprimer un rapport contenant ces erreurs. b. L'analyste prend connaissance des erreurs de correspondance.
Exigences particulières
Néant
Post conditions
[*en cas de succès*] La relation de correspondance entre le problème et le ProblemFrame est crée. [*en cas d'échec*] L'analyste a pris connaissance des erreurs de correspondance.

1.2.13 CAS D'UTILISATION Détail – Exprimer un phénomène de type Relation

Brève description
Ce cas d'utilisation décrit les opérations à effectuer pour définir un phénomène de type Relation (State, CausalState, SymbolicState, Truth, Role)
Acteurs
Analyste
Déclencheurs
L'analyste sélectionne l'opération dans le cas d'utilisation « Modifier le scope d'une description ».
Pré conditions
/
Flot d'événements de base
1. L'analyste sélectionne le type de phénomène CausalState. 2. Le système présente une boîte de dialogue et demande à l'utilisateur de lier deux phénomènes parmi les phénomènes existant dans le scope : un phénomène de type Value et un phénomène de type Entity. 3. L'analyste sélectionne les deux phénomènes.
Flots alternatifs
(Etape 1) 1. L'analyste sélectionne le type de phénomène SymbolicState. 2. Le système présente une boîte de dialogue et demande à l'utilisateur de lier deux phénomènes parmi les phénomènes existant dans le scope : un phénomène de type Value et un autre phénomène de type Value. 3. L'analyste sélectionne les deux phénomènes. **(Etape 1)** 1. L'analyste sélectionne le type de phénomène State. 2. Le système présente une boîte de dialogue et demande à l'utilisateur de lier deux phénomènes parmi les phénomènes existant dans le scope : un phénomène de type Value ou Entity et un autre phénomène de type Value. 3. L'analyste sélectionne les deux phénomènes.
Post conditions
Les deux phénomènes ont été sélectionnés.

Annexe C

Approche par les problem frames

Table des matières

Fonctionnalités

L'objectif de cette analyse (du problème) est de déterminer les fonctionnalités classiques de l'éditeur en utilisant l'approche des Problem Frames proposée par M. Jackson.

Context diagram

Le contexte général du problème peut être représenté par le « Context Diagram » suivant :

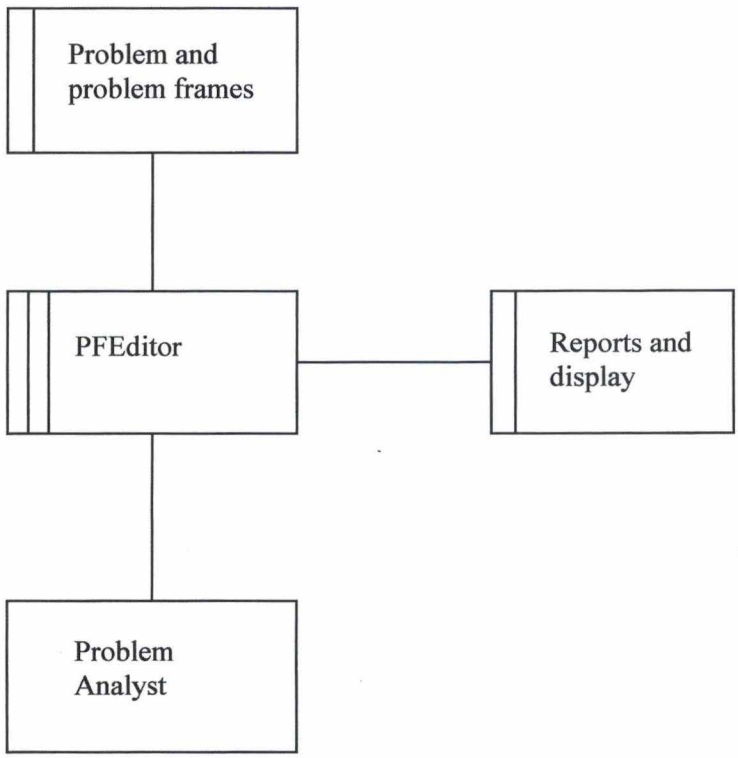


Figure 1 : Context diagram

Problem Analyst (PA): l'utilisateur.
Problem (PR) and problem frames (PF): la description du domaine correspond au métamodèle construit par G. Delannay complété par les éléments du chapitre 2.
Reports and display : ensemble des outputs de l'éditeur. « Display » est intégré au domaine car la représentation graphique des problèmes fait partie des exigences.

Découpe en sous-problèmes

Le problème peut être découpé en sous-problèmes principaux :

- Edition des PR/PF.
- Edition d'un PR projection d'un autre PR.
- Affichage des PR/PF.
- Validation des PR/PF.
- Correspondance (fitting) des PR par rapport au PF.
- Transcription et extraction des PR/PF.

Edition des PR et PF

L'analyste dispose d'une application qui manipule des objets PR et PF qui obéissent aux règles définies par le métamodèle. Les PR et PF en cours de construction ne doivent pas répondre à toutes les règles définies par le métamodèle. Par exemple, certaines cardinalités (1-...) associées aux figures 4 [MM, p9] et 6 [MM, p11] pourraient se transformer en cardinalités (0-...). Ces règles restent à définir. Exemple: un Domain peut ne pas être relié à un autre Domain via une interface; de même l' Interface ne relie qu'un Domain...

La vérification de la « syntaxe » du problème fait l'objet d'un autre sous-problème.

Le sous-problème est de la classe Manipulation d'objets (Simple Workpieces Frame) :

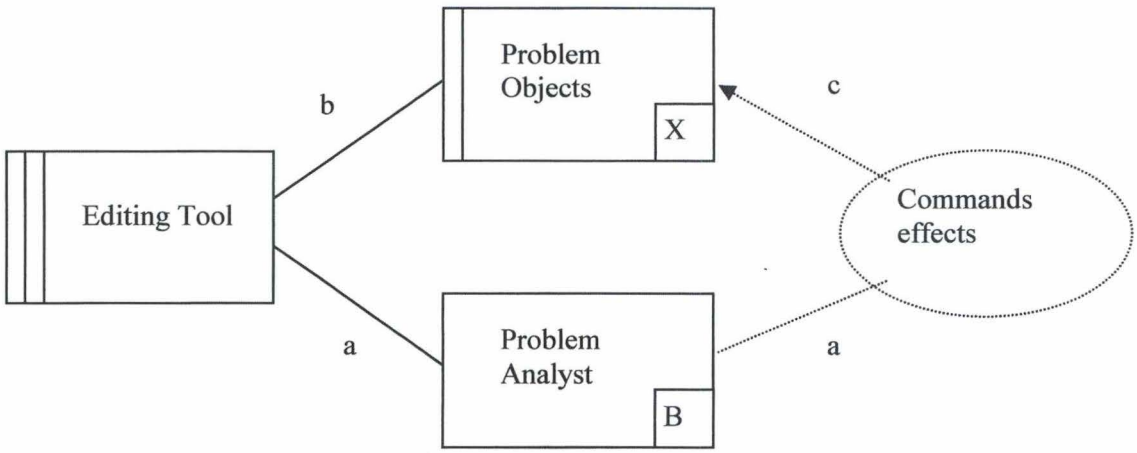


Figure 2 : Editing Tool subproblem

a : PA ! {Commands}
b : ET ! {Operations}
PO ! {Informations}
c : { PFLObjectsInformations Changes}

Commands (≈events)

- CreateProblem(*problemData*) : création d'un nouveau problème (ou Frame) dans le système.
- RemoveProblem(*problemId*) : suppression d'un problème du système.
- EditProblem(*problemId*) : modification d'un problème ; sous-commandes :
    - AddDomain(*domainData,problemId*) : ajouter un domaine au problème.
    - AddTangibleRequirement(*tangibleRequirementData,problemId*) : ajouter un « tangible requirement » au problème.
    - AddSpecification(*specificationData,problemId*) : ajouter la specification au problème.
    - AddIntangibleRequirement(*intangibleRequirementData,problemId*) : ajouter le requirement au problème.
    - EditDescription(*descriptionData*) : modification d'une description.
    - RemoveDescription(*descriptionId, problemId*) : enlever une description au problème.
    - AddInterface(*interfaceData,problemId*) : ajouter une interface au problème.
    - AddRequirementReference(*requirementReferenceData,problemId*) : ajouter un « requirement reference » au problème.
    - EditLink(*linkId*) : sous-commmandes :
        - LinkNewDescription(*descriptionId,linkId*) : relier une description.

        - RemoveDescriptionFromLink(*descriptionId,linkId*) : supprimer la relation entre la description et le lien
        - EditLinkProperties(*linkData*) : modifier les propriétés du lien.
    - RemoveLink(*linkId,problemId*): suppression d'un lien.
    - AddPhenomenaSetToInterface(*phenomenaSetData,interfaceId*) : ajout d'un nouveau set de phénomènes à une interface.
    - AddPhenomenaSetToRequirementReference(*phenomenaSetData,requirementReferenceId*) : ajout d'un nouveau set de phénomènes à un RequirementReference.
    - AddPhenomenonToPhenomenaSet (*phenomenonData,phenomenaSetId*) : ajout d'un phénomène au set.
    - RemovePhenomenonToPhenomenaSet (*phenomenonId,phenomenaSetId*) : enlève un phénomène au set.
    - ControlPhenomenaSet(*tangibleDescriptionId,phenomenaSetId*) : spécifie le domaine qui contrôle le set.
    - SpecifyScopeDescription(*descriptionId,phenomenaSetId*) : détermine le scope d'une description.
    - SpecifyInterfaceReference(*interfaceId,phenomenaSetId*) : spécifie qu'une interface référence un set de phénomènes.
    - SpecifyRequirementReferenceReference(*requirementReferenceId,phenomenaSetId*) : spécifie qu'un requirement référence un set de phénomènes.
    - BuildScopeDescription(*descriptionId)* : détermine le scope d'une description. Construction du scope à partir de tous les phenomenaSet reliés à une interface associées à la description.

Problem Objects

« Class diagrams » correspondant au figure 4 et 6 de [MM] + contraintes fig. 5,7 et 8 de [MM].

Operations (≈events)

- CreateProblemObject(*PR(PF)Data*) : création d'une nouvelle instance de Problem (ou ProblemFrame) et initialisation avec *PR(PF)Data* càd les données fournies par le PA.
- CreateInterfaceObject(*interfaceData*) : création d'une nouvelle instance de Interface et initialisation avec *interfaceData* càd les données fournies par le PA.
- CreateRequirementReferenceObject(*requirementReferenceData*) : création d'une nouvelle instance de RequirementReference et initialisation avec *requirementReferenceData* càd les données fournies par le PA.
- CreateSpecificationObject(*specificationData*) : création d'une nouvelle instance de Specification et initialisation avec *specificationData* càd les données fournies par le PA.
- CreateDomainObject(*domainData*) : création d'une nouvelle instance de Domain et initialisation avec *domainData* càd les données fournies par le PA.
- AddDescriptionToProblem(*descriptionId, problemId*) : la description devient partie composante du problème.
- RemoveDescriptionFromProblem(*descriptionId, problemId*) : la relation entre la description et le problème est supprimée.
- AddDescriptionLinkToProblem(*descriptionLinkId, problemId*) : le lien devient partie composante du problème.
- RemoveDescriptionLinkFromProblem(*descriptionLinkId, problemId*) : la relation entre le lien et le problème est supprimée.
- ConnectInterface(*interfaceId,tangibleDescriptionId*) : création d'une nouvelle association: un Interface (id = interfaceId) Connects un TangibleDescription (id = tangibleDescriptionId*)*.
- ConnectRequirementReference(*requirementReferenceId,intangibleRequirementId,domainId*) : création de 2 associations:

  - un RequirementReference (id = requirementReferenceId) Connects un IntangibleRequirement (id = intangibleRequirementId*)*.
  - un RequirementReference (id = requirementReferenceId) Connects un Domain (id = domainId*)*.
- DisconnectInterface(*interfaceId,tangibleDescriptionId*) : suppression de l'association: un Interface (id = interfaceId) Connects un TangibleDescription (id = tangibleDescriptionId*)*.
- DisconnectRequirementReference(*requirementReferenceId,intangibleRequirementId,domainId*) : suppression de 2 associations:
  - un RequirementReference (id = requirementReferenceId) Connects un IntangibleRequirement (id = intangibleRequirementId*)*.
  - un RequirementReference (id = requirementReferenceId) Connects un Domain (id = domainId*)*.
- CreatePhenomenonObject(*phenomenonData*) : création d'une nouvelle instance de Phenomenon et initialisation avec *phenomenonData* càd les données fournies par le PA.
- CreatePhenomenaSetObject(*phenomenaSetData*) : création d'une nouvelle instance de PhenomenaSet et initialisation avec *phenomenonData* càd les données fournies par le PA.
- AddPhenomenonToPhenomenaSet(*phenomenonId, phenomenaSetId*) : création d'une nouvelle association: un Phenomenon (id = phenomenonId) est relié à un PhenomenaSet (id = phenomenaSetId*)*.
- RemoveElementToPhenomenaSet(*phenomenonId, phenomenaSetId*) : suppression d'une association: un Phenomenon (id = phenomenonId) est relié à un PhenomenaSet (id = phenomenaSetId*)*.
- SetPhenomenaSetControl(*phenomenaSetId,tangibleDescriptionId) :* création d'une nouvelle association: un TangibleDescription (id = tangibleDescriptionId) Controls un PhenomenaSet (id = phenomenaSetId*)*.
- SetDescriptionScope(*phenomenaSetId,descriptionId) :* création d'une nouvelle association: un PhenomenaSet (id = phenomenaSetId*)* définit le scope d'un Description (id = descriptionId).
- SetInterfaceReference (*interfaceId, phenomenaSetId) :* création d'une nouvelle association: un Interface (id = interfaceId) References un PhenomenaSet (id = phenomenaSetId*)*
- SetRequirementReferenceReference (*requirementReferenceId, phenomenaSetId) :* création d'une nouvelle association: un RequirementReference (id = interfaceId) References un PhenomenaSet (id = phenomenaSetId*)*

Informations (≈states)
ProblemId(*problemData*) : l'id du Problem initialisé avec problemData.
DescriptionId(*descriptionData*) : l'id du Description initialisé avec descriptionData.
DescriptionLinkId(*descriptionLinkData*) : l'id du DescriptionLink initialisé avec descriptionLinkData.
ScopeId(*descriptionId*) : l'id du PhenomenaSet qui est le scope de Description identifié par *descriptionId*.

PhenomenaSetId(*phenomenaSetData*) : l'id du PhenomenaSet initialisé avec phenomenaSetData.
PhenomenonId(*phenomenonData*) : l'id du Phenomenon initialisé avec phenomenonData.
ListOfPhenomenaSet(*tangibleDescriptionId*) : liste d'id des PhenomenaSet que référencent les Interface ou les RequirementReference qui connectent tangibleDescriptionId.
ListOfPhenomenon(*phenomenaSetId*) : liste d'id des Phenomenon qui font partie d'un phenomenaSetId.
SpecificationId(problemId) : l'id de la Specification du problème identifié par problemId.
RequirementId(problemId) : l'id de l' IntangibleRequirement du problème identifié par problemId.

PFLObjectsInformations (≈states)
IsProblem(*p*) : état, p Problem (ou ProblemFrame).

IsDescription(d): état , d Description.
IsTangibleDescription(td): état , td TangibleDescription.
IsDomain(d): état , d Domain.
IsInterface(i) : état , i Interface.
IsIntangibleRequirement(R) : état , R IntangibleRequirement.
IsRequirementReference(r) : état , r RequirementReference.
IsElementOf(d,p) : état ; p Problem ; d Description ∨ DescriptionLink ∨ PhenomenaSet ; d est partie composante de p.
IsLink(l) : état ; l DescriptionLink .
IsPhenomenaSet(ps) : état ; ps PhenomenaSet .
IsPhenomenon(p) : état ; p Phenomenon .
Connects(l,d) : état ; l DescriptionLink ; d Description ; l Connects d.
Controls(td,ps) : état ; td TangibleDescription ; ps PhenomenaSet ; td Controls ps.
References(l,ps) : état ; l DescriptionLink ; ps PhenomenaSet ; l References ps.
Scope(ps,td) : état ; ps PhenomenaSet ; td TangibleDescription ; ps Scope td.

Commands effects

- CreateProblem(p) [¬IsProblem(p)] : IsProblem(p)
- AddDomain(d,p) [¬IsDomaine(d) ∧ IsProblem(p)] : IsDomain(d,p) ∧ IsElementOf(d,p)
- RemoveDescriptionOfProblem(d,p) [IsDescription(d) ∧ IsElementOf(d,p)] : ¬IsDescription(d,p)
- LinkNewDescription(d,l) [¬Connects(l,d)] : Connects(l,d)
- RemoveDescriptionFromLink(d,l) : [Connects(l,d)] : ¬Connects(l,d)
- RemoveLink(l) [IsLink(l)] : ¬IsLink(l)
- RemoveProblem(p) [IsProblem(p)] : ¬IsProblem(p)
- AddInterface(I,p) [¬IsInterface(i) ∧ IsProblem(p)] : IsInterface(i) ∧ IsElementOf(i,p)
- AddRequirementReference(r,p) [¬IsRequirementReference(r) ∧ IsProblem(p)] : IsRequirementReference (r) ∧ IsElementOf(r,p)
- AddTangibleDescriptionToInterface(td,i) [IsTangibleDescription(td) ∧ IsInterface(i) ∧ ¬Connects(i,t)] : Connects(i,td)
- AddDomainToRequirementReference(d,r) [IsDomain(d) ∧ IsRequirementReference(r) ∧ IsIntangibleRequirement(R) ∧ ¬Connects(r,d) ∧ ¬Connects(R,d)] : Connects(r,d) ∧ Connects(R,d)
- RemoveTangibleDescriptionToInterface(t,i) [Connects(i,t)] : ¬Connects(i,t).
- AddPhenomenaToPhenomenaSet(p,ps) [¬IsPhenomena(p) ∧ IsPhenomenaSet(ps)] : IsPhenomena(p) ∧ IsElement(p,ps)
- AddPhenomenaToScope(p,td) [¬IsPhenomena(p) ∧ IsTangibleDescription(td) ∧ Scope (td,ps)] : IsPhenomena(p) ∧ IsElement(p,ps)
- AddPhenomenaSetToInterface(ps,i) [¬IsPhenomenaSet(ps) ∧ IsInterface(i)] : IsPhenomenaSet(ps) ∧ Reference(i,ps)
- AddPhenomenaSetToRequirementReference(ps,rr) [¬IsPhenomenaSet(ps) ∧ IsRequirementReference (rr)] : IsPhenomenaSet(ps) ∧ Reference(rr,ps)
- RemovePhenomenaToPhenomenaSet(p,ps) [IsElement(p,ps)] : ¬IsElement(p,ps)
- ControlPhenomenaSet(td,ps) ∀j≠td, ∃i [IsTangibleDescription(td) ∧ IsTangibleDescription(j) ∧ Connects(i,td) ∧ Reference(i,ps) ∧ ¬Controls(j,ps)] : Controls(td,ps)
- ConstraintDomain(rr,d) ∀j≠td, ∃i [IsTangibleDescription(td) ∧ IsTangibleDescription(j) ∧ Connects(i,td) ∧ Reference(i,ps) ∧ ¬Controls(j,ps)] : Controls(td,ps)
- BuildScope (td) [¬Scope(td,ps) ∧ IsTangibleDescription(td) ∧ IsPhenomenaSet(ps)] : Scope(td,ps)

Comportement de Editing Tool

when CreateProblem(*PR(PF)Data*) :
generate event CreateProblemObject(*PR(PF)Data*)

when AddDomain(*domainData,problemId*) :
generate event CreateDomainObject(*domainData*)
generate event AddDescriptionToProblem(DomainId(*domainData*) ,*problemId*)

when RemoveDescription (*descriptionId,problemId*) :

generate event RemoveDescriptionFromProblem(*descriptionId, problemId*)
if Connects(l,d):
if IsInterface(l) generate event DisconnectInterface(*l,domainId*)
if Controls(d,p) generate event RemovePhenomenaSet(p)
il faut spécifier que des phénomènes sont effacés également
if IsRequirementReference(l) generate event DisconnectRequirementReference (*l,domainId*)

when AddInterface (*interfaceData,problemId*) :
generate event CreateInterfaceObject(*interfaceData*)
generate event AddDescriptionLinkToProblem(DescriptionLinkId(*interfaceData*) ,*problemId*)

when AddRequirementReference (*requirementReferenceData,problemId*) :
generate event CreateRequirementReferenceObject(*requirementReferenceData*)
generate event AddDescriptionLinkToProblem(DescriptionLinkId(*req.Ref.Data,problemId*)

when AddTangibleDescriptionToInterface (*tangibledescriptionId,interfaceId*) :
generate event ConnectInterface(*interfaceId ,tangibledescriptionId*)

when AddDomainToRequirementReference (*domainId, requirementReferenceId*) :
generate event ConnectRequirementReference(*requirementReferenceId,domainId*)

when RemoveTangibleDescriptionToInterface (*tangibledescriptionId,interfaceId*) :
generate event DisconnectInterface(*interfaceId ,tangibledescriptionId*)

when RemoveDomainToRequirementReference (*domainId, requirementReferenceId*) :
generate event DisconnectInterface(*requirementReferenceId,domainId*)

when RemoveDescriptionLink (*descriptionLinkId,problemId*):
generate event RemoveDescriptionLinkFromProblem(*descriptionLinkId, problemId*)
generate event DeletePFLObject(*descriptionLinkId*)

when RemoveProblem (*problemId*):
generate event DeletePFLObject(*problemId*)

when AddPhenomenonToPhenomenaSet (*phenomenonData,phenomenaSetId*) :
generate event CreatePhenomenonObject(*phenomenonData*)
generate event AddPhenomenonToPhenomenaSet(PhenomenonId*(phenomenonData),*
phenomenaSetId)

when AddPhenomenonToScope (*phenomenonData,tangibleDescriptionId*) :
generate event CreatePhenomenonObject(*phenomenonData*)

generate event AddPhenomenonToPhenomenaSet(PhenomenonId*(phenomenonData)*,
ScopeId*(tangibleDescriptionId)*)

when AddPhenomenaSetToInterface (*phenomenaSetData,interfaceId*) :
generate event CreatePhenomenaSetObject(*phenomenaSetData*)
generate event SetInterfaceReference(*interfaceId,* PhenomenaSetId*(phenomenaSetData)*)

when AddPhenomenaSetToRequirementReference (*phenomenaSetData,requirementReferenceId*) :
generate event CreatePhenomenaSetObject(*phenomenaSetData*)
generate event SetRequirementReference(*requirementReferenceId,*
PhenomenaSetId*(phenomenaSetData)*)

when RemovePhenomenonToPhenomenaSet (*phenomenonId,phenomenaSetId*) :
generate event RemoveElementToPhenomenaSet(*phenomenonId, phenomenaSetId*)
generate event DeletePFLObject(*phenomenonId*)

when ControlPhenomenaSet(*tangibleDescriptionId,phenomenaSetId*) :
generate event SetPhenomenaSetControl(*phenomenaSetId,tangibleDescriptionId)*

when BuildScopeDescription(*tangibleDescriptionId*) :
forall *phenomenaSetId* IN ListOfPhenomenaSet(*tangibleDescriptionId*)
forall *phenomenonId* IN ListOfPhenomenon(*phenomenaSetId*)
generate event AddPhenomenonToScope(*phenomenonatId,tangibleDescriptionId)*

Edition d'un PR projection d'un autre PR

L'analyste dispose d'une application qui manipule des objets PR (qui ne sont pas des PF). Le problème est similiare au sous-problème précédent (« Edition des PR et PF ») mais introduit la notion de projection. Chaque modification sur un problème se fait en tenant compte des contraintes supplémentaires imposées par les relations de projection. Nous décrivons uniquement les fonctionnalités spécifiques à la notion de projection.

Le sous-problème est de la classe Manipulation d'objets (Simple Workpieces Frame) :

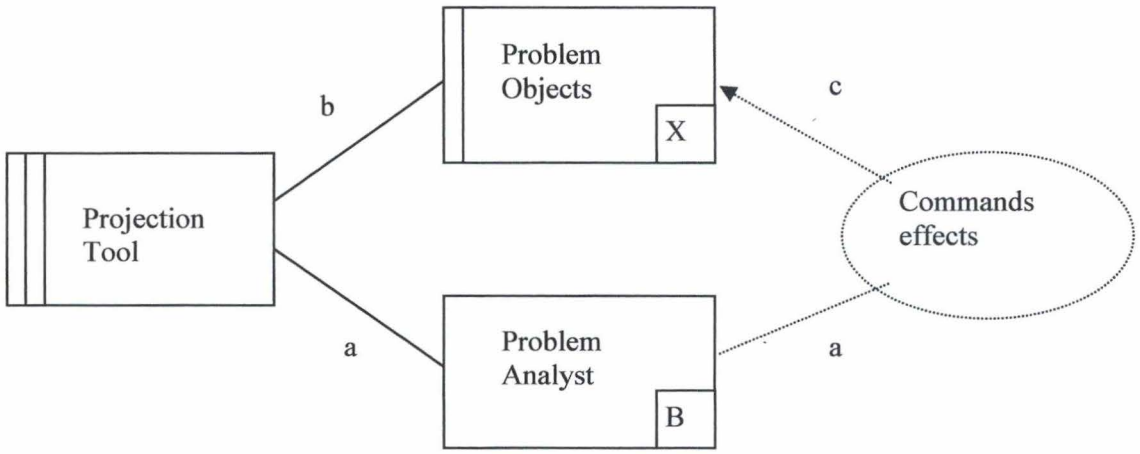


Figure 2 : Editing Tool subproblem

a : PA ! {Commands}
b : ET ! {Operations}
PO ! {Informations}
c : { PFLObjectsInformations Changes}

Commands (≈events)

- CreateProblemProjection(*subproblemData,projectedProblemData*) : création d'un nouveau problème projection d'un autre problème.
- DefineTangibleDescriptionScope(*tangibleDescriptionId, phenomenonData[]*) : définition du scope d'une tangibleDescription.
- CreateDomainProjection(*domainData, tangibleDescriptionId[]*) : création d'un nouveau domaine projection d'une ou plusieurs tangibleDescription.
- CreateSpecificationProjection(*subproblemDomainData, projectedTangibleDescriptionData[]*) : ajouter une spécification au sous-problème ; projection d'une ou plusieurs tangibleDescription.
- CreateIntangibleRequirementProjection(*intangibleRequirementData, projectedIntangibleRequirementId[]*) : création d'un nouveau intangibleRequirement projection d'une ou plusieurs intangibleRequirement.
- AddPhenomenonToPhenomenaSet (*phenomenonData,phenomenaSetId*) : ajout d'un phénomène au set.

Autres « Commands » permises et non étudiées au niveau de ce sous-probléme car identiques et déjà étudiées dans le sous-problème « Edition des PR etPF » :

- AddPhenomenaSetToInterface(*phenomenaSetData,interfaceId*) : ajout d'un nouveau set de phénomènes à une interface.

- AddPhenomenaSetToRequirementReference(*phenomenaSetData,requirementReferenceId*) : ajout d'un nouveau set de phénomènes à un RequirementReference.
- EditLink(*linkId*) : sous-commmandes :
    - LinkNewDescription(*descriptionId,linkId*) : relier une description.
    - RemoveDescriptionFromLink(*descriptionId,linkId*) : supprimer la description du lien
    - EditLinkProperties(*linkData*) : modifier les propriétés du lien.
- RemoveLink(*linkId,problemId*): suppression d'un lien.
- EditDescription(*descriptionData*) : modification d'une description.
- RemoveDescription(*descriptionId, problemId*) : enlever une description au problème.
- AddInterface(*interfaceData,problemId*) : ajouter une interface au problème.
- AddRequirementReference(*requirementReferenceData,problemId*) : ajouter un « requirement reference » au problème.
- RemovePhenomenonToPhenomenaSet (*phenomenonId,phenomenaSetId*) : enlève un phénomène au set.

ProblemObjects
Le Domaine correspond aux figures suivantes du métamodèle :
[MM, figure 4] – Problem (frame) content
[MM, figure 5] – Constraints relating to figure 4
[MM, figure 6] – Idem 4 with phenomenaSet
[MM, figure 7] – Constraints relating to figure 6 1/2
[MM, figure 8] – Constraints relating to figure 6 2/2
[MM, figure 11] – Projections
[MM, figure 12] – Constraints relating to figure 11

La relation Projection concerne uniquement les classes Problem , IntangibleRequirement et TangibleDescription. Une projection entre problèmes est toujours définie en terme de partie de problème (Requirement,Description). La projection entre Description est définie en terme de scope et de phenomena. Le concept de projection est défini en terme de scope « inclusion ».

Phenomena control :

Il semble logique d'inclure plusieurs contraintes concernant le contrôle :
- *(vers le haut)* si ,au niveau du sous-problème, un phenomenaSet est «contrôlé » par un td, il faut que chaque phénomène inclus dans ce phSet soit également inclus dans un phenomenaSet contôlé par une des tangibleDescription (au niveau du problème de niveau supérieur) dont td est une projetion.
- *(vers le bas)* et inversèment.

PFLObjectsInformations (≈states)
IsProjectionProblemOf(*p,g*) : état ; p et g : Problem; p est projection de g.
IsProjectionDomainOf(*d,td*) : état ; d Domain ; td TangibleDescription ; d IsProjectionOf td.
IsProjectionIntangibleRequirementOf(*pir,ir*) : état ; pir IntangibleRequirement ; ir IntangibleRequirement ; pir IsProjectionOf ir.
IsProjectionSpecificationOf(*ss,td*) : état ; ss Specification ; td TangibleDescription ; ss IsProjectionOf td.
IsInScope(ph,td) : état ; ph Phenomenon ; td TangibleDescription ; ∃! ps PhenomenaSet : ph phens ps ∧ ps scope ph

Commands effects

- CreateProblemProjection(sp,pp) [¬IsProblem(sp) ∧ IsProblem(pp)] : IsProblem(sp) ∧ IsProjectionProblemOf(sp,pp)
- CreateDomainProjection(spd, $td_{i:1..n}$) ∀i, ∃sp,pp: [¬IsDomain(spd) ∧ IsTangibleDescription(td_i) ∧ IsElementOf(td_i,pp)] : IsDomain(spd) ∧ IsElementOf(spd,sp) ∧ IsProjectionProblemOf(sp,pp) ∧ IsProjectionDomainOf(spd, td_i)
- CreateSpecificationProjection(ss, $td_{i:1..n}$) : ∀i, ∃sp,pp: [¬IsSpecification(ss) ∧ IsTangibleDescription(td_i) ∧ IsElementOf(td_i,pp)] : IsSpecification(ss) ∧ IsElementOf(ss,sp) ∧ IsProjectionProblemOf(sp,pp) ∧ IsProjectionSpecificationOf(ss, td_i)
- CreateIntangibleRequirementProjection(sir, $ir_{i:1..n}$) : ∀i, ∃sp,pp: [¬IsIntangibleRequirement(sir) ∧ IsIntangibleRequirement (td_i) ∧ IsElementOf(ir_i,pp)] ↦ IsIntangibleRequirement(sir) ∧ IsElementOf(sir,sp) ∧ IsProjectionProblemOf(sp,pp) ∧ IsProjectionIntangibleRequirementOf(sir, ir_i)
- DefineTangibleDescriptionScope(td, $ph_{i:1..n}$) : ∀i, ∃ptd: [¬IsPhenomenaSet(ps) ∧ (IsProjectionSpecificationOf(td,ptd) ∨ IsProjectionDomainOf(td,ptd)) ∧ IsInScope(ph_i,ptd)] : IsPhenomenaSet(ps) ∧ IsInScope(ph_i,ptd) ∧ Scope(td,ps)

Comportement de Projection Tool

when CreateProjection(*subProblemData, problemId*) :
generate event CreateProblemObject(*subProblemData*)
generate event LinkProblem(*ProblemId(subProblemData), problemId*)

when DefineTangibleDescriptionScope(*tangibleDescriptionId, phenomenonId []*) :
generate event CreatePhenomenaSetObject(*scopeData*)
forall *phenomenonId*
generate event
LinkPhenomenonToPhenomenaSet(*phenomenonId,ScopeId(tangibleDescriptionId*)

when CreateDomainProjection(*domainData, tangibleDescriptionId[]*)
generate event CreateDomainObject(*domainData*)
forall *tangibleDescriptionId*
generate event
CreateTangibleDescriptionProjectionLink(*DescriptionId(domainData),tangibleDescriptionId*)

when CreateSpecificationProjection(*specificationData, tangibleDescriptionId[]*)
generate event CreateSpecificationObject(*specificationData*)
forall *tangibleDescriptionId*
generate event
CreateTangibleDescriptionProjectionLink(*DescriptionId(specificationData),tangibleDescriptionId*)

when CreateIntangibleRequirementProjection(*intangibleRequirementData, tangibleDescriptionId[]*)
generate event CreateIntangibleRequirementObject(*intangibleRequirementData*)
forall *tangibleDescriptionId*
generate event
CreateIntangibleRequirementProjectionLink(*DescriptionId(intangibleRequirementData),tangibleDescriptionId*)

AddPhenomenonToPhenomenaSet (*phenomenonData,phenomenaSetId*) : ajout d'un phénomène

Operations (≈events)

- CreateProblemObject(*PR(PF)Data*) : création d'une nouvelle instance de Problem (ou ProblemFrame) et initialisation avec *PR(PF)Data* càd les données fournies par le PA.
- CreateInterfaceObject(*interfaceData*) : création d'une nouvelle instance de Interface et initialisation avec *interfaceData* càd les données fournies par le PA.
- CreateRequirementReferenceObject(*requirementReferenceData*) : création d'une nouvelle instance de RequirementReference et initialisation avec *requirementReferenceData* càd les données fournies par le PA.
- LinkPhenomenonToPhenomenaSet(*phenomenonId,phenomenaSetId*) : création d'une nouvelle association : un Phenomenon (id = phenomenonId) est relié à un PhenomenaSet (id = phenomenaSetId)
- CreateIntangibleRequirementProjectionLink(*intangibleRequirementId,projectedIntangibleRequirementId*) : création d'une nouvelle association : un IntangibleRequirement (id = intangibleRequirementId) IsProjectionOf un IntangibleRequirement (id = projectedIntangibleRequirementId)
- CreateTangibleDescriptionProjectionLink(*tangibleDescriptionId,projectedTangibleDescriptionId*) : création d'une nouvelle association : un TangibleDescription (id = tangibleDescriptionId) IsProjectionOf un TangibleDescription (id = projectedTangibleDescriptionId)
- LinkProblem(*subproblemId, problemId*) : création d'une nouvelle association : un Problem (id = subproblemId) IsProjectionOf un Problem (id = problemId)

- CreateTangibleDescriptionScope(*tangibleDescriptionId*) : reconstitution du scope de la tangibleDescription à partir des interfaces reliées à cette description (dans Edition des PR et PF)

Affichage des PF et PR

Les Problem Objects doivent être représentés selon des règles bien précises.

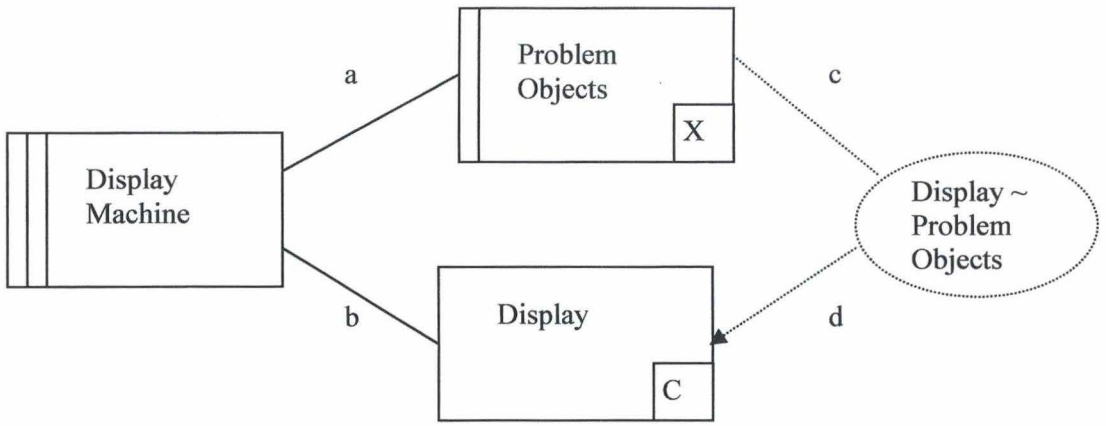


Figure 3 : Display Machine subproblem (with model)

a et c : PO ! {ProblemProperties}
b : DM ! {DisplayCommands}
d : DI ! {ObjectDisplayed}

ProblemProperties (≈states)
Le problème avec toutes ses propriétés (instances de classes et valeurs des attributs liés à ces instances pour l'instance Problem (ou ProblemFrame) considérée (hors projection)). Nous pensons que a et c sont les mêmes phénomènes c-à-d un objet Problem (ou ProblemFrame) dans un état déterminé auquel doit correspondre un affichage déterminé. C'est en même temps le « symbolic requirement phenomena » (c) et le « causal phenomena » (a) de PO.

DisplayCommands (≈events)
Ensemble de commandes d'affichage d'objets graphiques (exemple : rectangle, texte…)

ObjectsDisplayed (≈states)
Ensembles des objets graphiques devant être affichés

Display ~ Problem Objects
Le « requirement » doit exprimer la correspondance entre les propriétés du problem et les objets à afficher en respectant le formalisme de M. Jackson(voir [2, Appendix 1 :notations]).

Problem properties	ObjectsDisplayed
Domain	Box(name,type)
DesignedDomain	simplestripedBox(name,type)
Specification	doublestripedBox(name)
TangibleRequirement	Oval(name)
DesignedTangibleRequirement	simpleStripedOval(name)
IntangibleRequirement	dashedOval(name)
Interface	solidLine(annotation)
MultipleInterface	connectedHyperarc(annotation)
RequirementReference	dashedLine(annotation)
ConstrainedRequirementReference	dashedArrow(annotation)

PhenomenaSet Text(annotation,abbr,names[])

Règles d'affichage concernant les relations :
Interface Connects Tangible Description ConnectedObjects
RequirementReference Connects Domain ConnectedObjects
RequirementReference Connects IntangibleRequirement ConnectedObjects

$\forall$ x, y ObjectDisplayed, $x \neq y : x \cap y$ = connectedPoint(x,y).
rem : connectedPoint = $\varnothing$ si x n'est pas connecté à y.

Choix du problemFrame:
On peut faire correspondre ce sous-problème à un « information display frame » bien que le type du domaine ProblemObjects soit lexical. En effet, on peut considérer que ce domaine présente un aspect causal au niveau de ses opérations et de leurs effets. On pourrait également considérer ce sous-problème comme étant la deuxième partie d'un « decomposed information display frame » [2, p194] ; la modélisation étant déjà faite et le fossé entre a et c comblé.
Nous considèrons que PO est actif dans le sens où il génère *spontanément* des modifications aux objets Problem (même si ces modifications sont initiées par l'utilisateur ; ce que nous ignorons dans le cadre de ce sous-problème).

Validation des PR et PF

La validation du PR/PF est simplement la vérification de la « syntaxe » du PR/PF autrement dit le contrôle du respect de toutes les règles définies par le métamodèle (cardinalités, contraintes…).

Sous-problème de la classe Transformation :

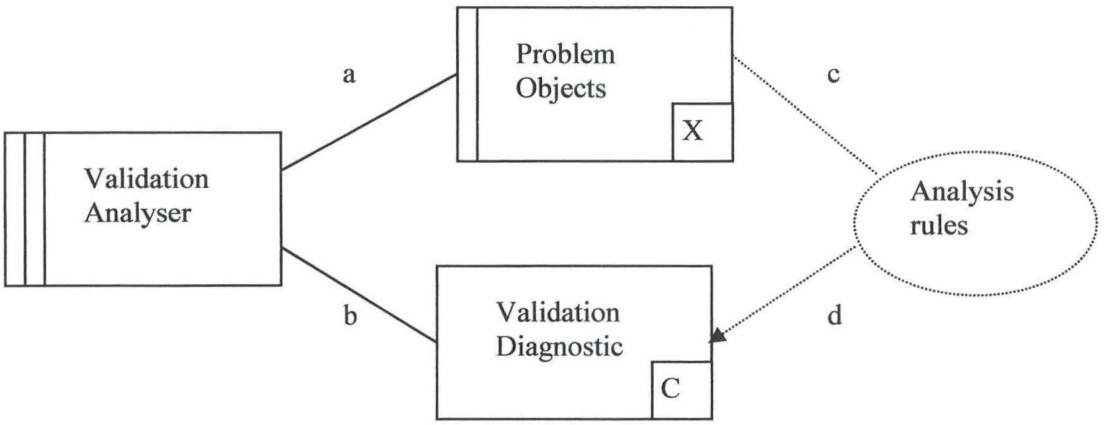


Figure 4 : Validation subproblem

a : PO ! {ProblemObject}
b : VA ! {Validation Report Line }
c : PO ! {ProblemProperties}
d : VD ! {Validation Line Data}

ProblemProperties (≈states)
Propriétés du problème (instances de classes et valeurs des attributs liés à ces instances pour l'instance Problem ou ProblemFrame considérée).
Ensemble de « states » qui expriment une règle à respecter
Exemple :
NumberOfSpecifications(p) : état ; p Problem ; # $\{s_0, \ldots, s_j, \ldots, s_i\}$: s_j Specification $\wedge$ IsElementOf(s_j,p)
NumberOfIntangibleRequirement(p) : état ; p Problem ; # $\{r_0, \ldots, r_j, \ldots, r_i\}$: r_j IntangibleRequirement $\wedge$ IsElementOf(r_j,p)
RespectConstraints(p) : état ; p Problem ;

ProblemObject (≈states)
l'objet Problem à analyser ainsi que tous les objets associés ; si ce problème est projection d'un autre problème => y compris le problème de niveau supérieur.

Problem(problemId) : l'objet Problem correspondant à problemId avec tous les objets « associés ».

Validation Line Data (≈states)
Ligne qui contient le nom de(s) entité(s), la propriété et/ou la contrainte non respecté(es) + un libellé.

Analysis rules
Ensemble des règles que doit respecter le Problem (ou le ProblemFrame):

• Considérations intraproblème	*Validation LineData*
- 1 et 1! Specification;	*Number of Specification ≠ 1*
- 1 et 1! IntangibleRequirement;	*Number of IntangibleRequirement ≠ 1*
- au moins 1 ExternalDomain;	*No External Domain*
- 1 Interface relie au moins 2 TangibleDescription;	

The Interface...must connect at least 2 TangibleDescription

- 1 RequirementReference relie 1 et 1! IntangibleRequirement avec 1 et 1! ExternalDomain; *The RequirementReference must connect one IntangibleReference and oneExternalDomain*
- contraintes 1 à 5 de [MM, figure 5]; *Constraint number ? not respected*
- au moins1 RequirementReference avec *Constraining* = True; *The IntangibleRequirement must be connected to at least one RequirementReference with Constraining = True*
- au moins1 RequirementReference avec *Constraining* = False; *The IntangibleRequirement must be connected to at least one RequirementReference with Constraining = False*
- contraintes de [MM, figure 7]; *Constraint number ? not respected*
- contraintes de [MM, figure 8]; *Constraint number ? not respected*
-

- Considérations interproblème (projections)
  - 1 Problem est projection de 0 ou 1 Problem; *The problem is projection of more than one problem*
  - 1 TangibleDescription est projection de 0 ou 1 TangibleDescription; *The tangibleDescription is projection of more than one tangibleDescription*
  - 1 IntangibleRequirement est projection de 0 ou 1 IntangibleRequirement ; *The intangibleRequirement is projection of more than one intangibleRequirement*
  - contraintes de [MM, figure 12]; *Constraint number not respected*

ProblemObjects

Le Domaine correspond aux figures suivantes du métamodèle :
[MM, figure 4] – Problem (frame) content
[MM, figure 5] – Constraints relating to figure 4
[MM, figure 6] – Idem 4 with phenomenaSet
[MM, figure 7] – Constraints relating to figure 6 1/2
[MM, figure 8] – Constraints relating to figure 6 2/2
[MM, figure 11] – Projections
[MM, figure 12] – Constraints relating to figure 11

Correspondance (Fitting) d'un problème

L'éditeur aide l'analyste à décomposer son problème : vérification de la correspondance (« fitting ») d'un sous-problème avec un Problem Frame.
La machine doit appliquer un PF au PT et donner son diagnostic FD:

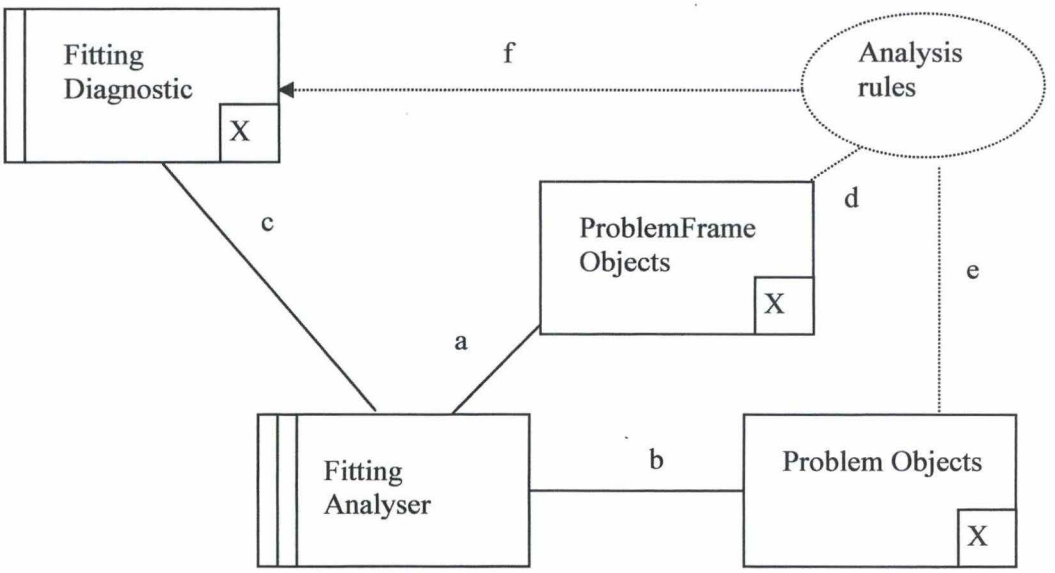


Figure 5 : Fitting subproblem

a : PF ! {ProblemFrameObject}
b : PO ! {ProblemObject}
c : FA ! {Fitting Report Line }
d : PO ! {ProblemFrameProperties}
e : PO ! {ProblemProperties}
f : VD ! {Fitting Line Data}

ProblemProperties (≈states)
Propriétés du problème (instances de classes et valeurs des attributs liés à ces instances pour l'instance Problem ou ProblemFrame considérée).
Ensemble de « states » qui expriment une règle à respecter
Exemple :
NumberOfSpecifications(p) : état ; p Problem ; # $\{s_0, \ldots, s_j, \ldots, s_i\}$: s_j Specification $\wedge$ IsElementOf(s_j,p)
NumberOfIntangibleRequirement(p) : état ; p Problem ; # $\{r_0, \ldots, r_j, \ldots, r_i\}$: r_j IntangibleRequirement $\wedge$ IsElementOf(r_j,p)
RespectConstraints(p) : état ; p Problem ;

ProblemObject (≈states)
l'objet Problem à analyser ainsi que tous les objets associés ; si ce problème est projection d'un autre problème => y compris le problème de niveau supérieur.

Problem(problemId) : l'objet Problem correspondant à problemId avec tous les objets « associés ».

Line Data (≈states)
Ligne qui contient le nom de(s) entité(s), la propriété et/ou la contrainte non respecté(es) + un libellé.

Analysis rules
Les règles à respecter dans le cadre d'un "Fitting" sont exprimées par les contraintes des figures 13 et 14 [MM, p 21-22].
rem: réflexion à faire au sujet des contraintes concernant les interfaces et les 'phensets, type de domaine...'

Problem Objects
Projection du domaine ProblemObjects (sous-problème « Edition des PR/PF »).
Reprend uniquement les Problem.

ProblemFrame Objects
Projection du domaine ProblemObjects (sous-problème « Edition des PR/PF »).
Reprend uniquement les ProblemFrame.

Analysis rules

Règles de validation
Générale : [MM, figure13]
Par concept : [MM, figure14]

Le concept de validation peut se voir comme une adéquation entre le ProblemFrame et le Problem.
Cette adéquation est mesurée également par les paramètres suplémentaires suivants :
Interface.type
Domain.type
RequirementReference.constraining

2 étapes :
- FittingAnalyser utilise l'objet PF pour en déduire des règles de fitting
- FittingAnalyser vérifie sur l'objet PR si les règles sont bien respectées.

Transcription d'un PR/PF
Transcription d'un objet Problem ou ProblemFrame sous forme d'un fichier (le format reste à déterminer).

Sous-problème de la classe Transformation:

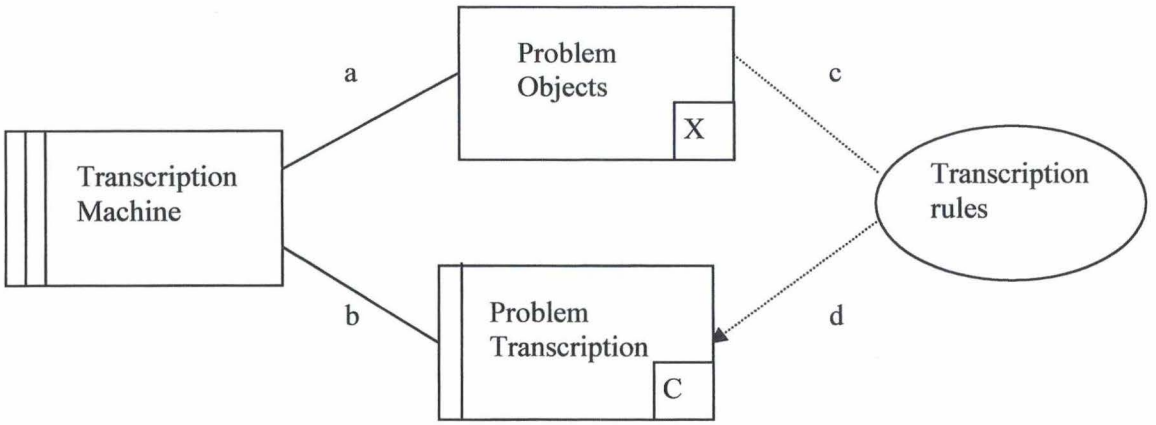


Figure 6 : Transcription subproblem

a et c : PO ! {ProblemObject}
b : TM ! {Transcription Line }
d : PT ! {Transcription Data}

ProblemObject (≈states)
l'objet problem à transcrire.

Transcription Line (≈events)
Line suivant le format de fichier utilisé.

Transcription Data (≈states)
Ensemble des données devant être transférées sur le fichier.

Type	Data
Problem	ProblemId, ProblemName
ProblemFrame	ProblemFrameId, ProblemFrameName
IsProjectionOf	ProblemId, ProblemId

D'une manière plus générique :

Class	ClassType, ClassId
Attribute	ClassId, AttributeName, AttributeValue
Relation	ClassId, ClassId

Extraction d'un PR/PF
Sous-problème corollaire au sous-problème précédent: extraction d'un objet Problem ou ProblemFrame à partir d'un fichier.