HAL
archives-ouvertes.fr

# Conception de traces, et applications, vers une méta-théorie des traces (Towards a Meta-Thoery of traces)

Pierre Deransart

▶ **To cite this version:**

Pierre Deransart. Conception de traces, et applications, vers une méta-théorie des traces (Towards a Meta-Thoery of traces). 2009. inria-00370211

**HAL Id: inria-00370211**
**https://hal.inria.fr/inria-00370211**

Preprint submitted on 23 Mar 2009

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Conception de traces, et applications (vers une méta-théorie des traces)

‘‘document de travail’’

23 mars 2009

Pierre Deransart

Inria Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France
Pierre.Deransart@inria.fr

**Résumé** Ce document est une synthèse en évolution constante de quelques travaux et réflexions conduits après 2004 (la fin du projet OADymPPaC) et concernant une méthodologie de construction de traces.

Par “trace” nous entendons toute suite discrète non bornée d'événements constitués chacun d'un ensemble d'attributs et susceptibles de rendre compte en partie du comportement d'un système ou phénomène observé.

Ce que nous voulons étudier ici est le processus par lequel une trace est finalement conçue. Le développement de traces pour un processus ou une famille de processus repose sur un cycle d'observations et d'analyses qui conduit progressivement à identifier les concepts éventuellement mesurables et utiles à une analyse. Cette analyse permettra éventuellement elle-même de trouver de nouveaux concepts utiles à la compréhension du phénomène observé, qui, à leur tour, pourront amener à introduire de nouveaux concepts mesurables et de nouveaux attributs dans la trace. L'identification des concepts est liée à une activité de modélisation et d'abstraction, et leur transcription dans une trace relève de la recherche de leur représentation sous une forme aisément transmissible, mais aussi utilisable pour les analyses.

Ce que nous attendons d'une telle étude, outre bien sûr une meilleure compréhension de la manière dont sont construites les traces, c'est une méthode rigoureuse de développement de traces susceptible de s'appuyer sur des outils. Un résultat intéressant, par exemple, peut être un langage de prototypage de trace indépendant des processus observés.

Au stade où en est cette étude l'auteur n'a pu conduire que des expérimentations dans le domaine des solveurs de contraintes. L'observation par des traces du comportement parfois inattendu d'un solveur, où chaque contrainte peut être vue comme un agent agissant sur un domaine, a mis en évidence l'intérêt potentiel d'une approche méthodologique. On a pu montrer en particulier comment il était possible d'améliorer la réalisation de traceurs grâce à cette approche tant au plan de la conception que de l'implantation.

De manière générale on retrouve le concept de trace dans de nombreux champs d'études allant du génie logiciel aux traces mémorielles dans le système nerveux, en passant par l'apprentissage en environnement médié.

L'un des objectif de cette étude est de favoriser, grâce à une démarche systématisée, une possible cross-fertilisation entre des domaines aussi variés que le génie logiciel, l'analyse de données, l'éducation, la médecine ou la philosophie.

## Préambule

Cette étude vise à développer le cadre méthodologique concernant les traces génériques tel qu'il a été développé empiriquement dans le projet OADymPPaC [29] de manière à en apprécier toute la portée et permettre d'autres développements. On peut en effet déjà attendre d'un approfondissement d'une approche systématique des traceurs et leurs traces, une meilleure compréhension a posteriori des travaux menés dans cadre du projet OADymPPaC, en particulier ceux qui ont été publiés dans [59] en 2003 et [60] en 2004. On sera alors mieux à même d'apprécier la force de la notion de trace générique, idée centrale dans le projet OADymPPaC. Mais on peut en attendre bien plus.

En effet ce projet s'est concentré sur la construction rapide et immédiate (le projet durait 3 ans) d'une trace dite "générique" destinée à permettre un développement rapide d'analyseurs sophistiqués de divers solveurs de contraintes, tout en réalisant sa mise au point progressive. De ce fait les conséquences de cette approche "générique" n'ont pu être étudiées. En particulier la nature même d'une trace "générique", ses liens avec la sémantique de chaque solveur, l'utilisation pratique d'une sous-trace, sa non contiguité possible (sauf dans quelques cas précis d'interruption et de reprise), la présence ou non dans la trace effective des concepts que l'on veut analyser, la construction d'une trace générique à partir de composants eux-mêmes munis d'un traceur, n'ont pu être proprement appréciés et étudiés. C'est cela que nous étudions. De plus, au stade du projet OADymPPaC, il n'était pas possible d'établir des liens très précis avec d'autres domaines, pratiques ou théoriques (autres que le débogage dynamique direct) où les traces sont étudiées en tant qu'objets à manipuler ou à analyser (fouille de données, flux de données, systèmes événements/ actions, langages de trace, ...).

Voir également le colophon, après la bibliographie.

# Table des matières

Conception de traces, et applications (vers une méta-théorie des traces)
‘‘document de travail’’ 23 mars 2009 . . . . . 1
*Pierre Deransart*
1 Avant propos . . . . . 6
2 Introduction . . . . . 7
3 Méthodologie de développement de trace . . . . . 7
3.1 Traces virtuelle et effective . . . . . 8
3.2 Trace intégrale . . . . . 9
3.3 Evolution de la trace intégrale . . . . . 11
3.4 Méthodologie . . . . . 12
4 Traces intégrales contigues . . . . . 12
4.1 Trace virtuelle intégrale contigue . . . . . 12
4.2 Trace effective intégrale contigue . . . . . 13
4.3 Signature . . . . . 14
5 Sémantiques des traces . . . . . 14
5.1 Sémantique interprétative (SI) d’une trace effective . . . . . 14
5.2 Sémantique observationnelle (SO) . . . . . 16
5.3 Représentations des SI et SO . . . . . 19
6 Fondements de la Sémantique Observationnelle . . . . . 22
6.1 SO comme une sémantique des traces partielles . . . . . 23
6.2 Interprétation abstraite . . . . . 25
6.3 Sémantique de programmes logiques et interprétation abstraite . . 27
6.4 Propriétés des signatures . . . . . 30
7 Fidélité . . . . . 33
7.1 Fidélité d’une SO . . . . . 33
7.2 Méthodes de vérification de fidélité . . . . . 34
8 Transformations de traces . . . . . 37
8.1 Enrichissement . . . . . 38
8.2 Sélection . . . . . 38
8.3 Fusion . . . . . 39
8.4 Abstraction . . . . . 39
8.5 Généricité . . . . . 40
9 Enrichissement . . . . . 40
10 Sélection . . . . . 41
11 Fusion . . . . . 41
12 Abstraction . . . . . 42
13 Généricité . . . . . 42
14 Domaines fondateurs et applicatifs . . . . . 42
14.1 Construction de traceurs pour l’analyse dynamique de programmes . . . . . 43
14.2 Modélisation et Abstraction . . . . . 46

14.3 Fouille de donnés, analyse et interrogation de flots de données ... 54
14.4 Modèles événements/actions ... 57
14.5 Fusion de données et analyse du comportement humain ... 59
14.6 Modélisation conceptuelle et WEB sémantique ... 65
14.7 Auxiliaire de mémoire ... 68
14.8 Epistémologie ... 72
14.9 Conclusions sur les domaines fondateurs et applications ... 73
15 Conclusion (à venir) ... 76
16 Remerciements ... 76
Références ... 78
Annexe : Exemples ... 85
17 Exemple : la fonction de Fibonacci ... 85
18 Exemple : modèle des boîtes pour Prolog ... 86
18.1 Sémantique observationnelle ... 86
18.2 Sémantique interprétative ... 94
19 Thielscher's Office Delivery Robot Example ... 97
19.1 The robots world ... 97
19.2 OS and IS Semantics ... 98
19.3 OS in Prolog style ... 101
19.4 OS Properties ... 105
19.5 IS in Prolog style ... 108
19.6 OS in Flux style ... 109

## 1 Avant propos

D'une manière générale, on veut s'intéresser à l'observation de processus dynamiques à partir des traces qu'ils laissent ou qu'on leur fait produire.

On s'intéresse plus particulièrement ici à donner des sémantiques aux traceurs, c'est à dire à des générateurs de traces, et aux traces qu'ils produisent. On cherche des sémantiques aussi indépendantes que possible de celles des processus tracés ou de la manière dont leurs traceurs les produisent.

Il faut également bien distinguer ce qui relève de ce que nous appelons ici "trace" et ce qui relève d'outils d'analyse de processus ("monitoring", présentation particulière de la trace ou "jolies" impressions, visualisation, analyse de performance, débogage, ...) qui tous d'une manière ou d'une autre, directement ou indirectement, de l'intérieur ou indépendemment, en mode synchrone ou asynchrone, utilisent des traces à des fins d'analyse diverses mais ne les produisent pas. Cette étude ne concerne pas la nature ni la forme de ces processus observateurs. Ceux-ci sont toujours par définition externes au processus observé et utilisent leur trace comme une donnée externe.

On peut toujours considérer qu'entre un observateur (ou processus observateur) et un phénomène (ou processus) observé il y a un objet que nous appellerons *trace effective*. La trace effective est l'empreinte reconnaissable laissée par un processus, ou plusieurs processus pris ensembles, et donc "lisible" par d'autres processus, en temps réel ou a posteriori. Le phénomène observé sera considéré ici comme un seul processus fermé (ceci concernant toutes les donnés et fonctions qu'il manipule) dont on ne connaît que la trace effective.

On introduira également la notion de *trace virtuelle*. La trace virtuelle est une suite d'événements représentant l'évolution d'un état qui contient tout ce que l'on peut ou veut connaître de ce ou ces processus, c'est à dire toutes ses "observables". Cette évolution peut être éventuellement formalisée par un modèle de transition d'états, c'est à dire par un domaine d'états et une fonction de transition formalisant le passage d'un état à un autre. Cette sémantique, munie de l'ensemble des opérations d'extraction de la trace effective, sera appelée *sémantique observationnelle* (SO) car elle représente ce que l'on est susceptible de connaître ou de décrire du processus, vu de l'"extérieur". Toutefois la trace virtuelle elle-même, prise isolément, ou, plus précisemment l'ensemble des traces qu'un processus est susceptible d'émettre, peut être considérée également comme une sémantique.

Finalement, cette approche s'appuie sur la notion de *trace (effective ou virtuelle) intégrale* en ce sens que le totalité de la connaissance (à un moment donné) concernant le processus observé s'y trouve explicitement ou implicitement contenue. Ceci constitue une base qui permet étudier la construction de traceurs et de traces modulaires à partir des seules notions d'enrichissement de trace, de sous-trace et de trace générique. On donne alors quelques pistes pour étudier la construction des traces à l'aide de manipulations telles que l'enrichissement d'une trace par raffinement et sélection d'une sous-trace par oubli sélectif. On s'intéresse également à la généricié d'une trace, c'est à dire la possiblilité

de donner une sémantique à un traceur, valable pour une famille de processus observés.

## 2 Introduction

Cette étude porte sur la méthodologie de développement de traces. Par trace nous entendons toute forme d'observation discrète de phénomènes, toute collection d'informations discrétisées qui peut être utilisée à des fins d'analyse sans référence directe au système qui les a produites. Les traces sont des objets produits par des phénomènes ou systèmes qui peuvent être pris comme une entité unique autonome, laquelle ne peut être perçue et appréhendée que par ses traces. Les traces sont les seuls objets à partir desquels une analyse du phénomène observé est possible. L'analyse dont il est question ici sous-entend que l'observation du phénomène est séparée et indépendante du phénomène lui-même. Par contre traiter de la question du développement d'une trace signifie que la trace n'est pas nécessairement indépendante de l'analyse que l'on veut pouvoir mener. Parler de développement de trace peut signifier que l'on puisse intervenir sur le phénomène pour lui faire produire une trace qui contienne les informations nécessaires à l'analyse que l'on souhaite mener.

La construction de trace peut se décomposer en opérations élémentaires dont l'ensemble constitue ce que nous appelons une "algèbre de traces" dont nous étudierons quelques propriétés.

Cett étude débute par les notions de traces intégrales virtuelle et effective contigues (section 4) et leur sémantique (section 5), et les rapports qu'elles entretiennent (section 7).

Il faut noter que la plupart des notions introduites ici peuvent évoluer en fonction du développement même de cette étude et des résultats aquis dans de nouveaux chapitres.

## 3 Méthodologie de développement de trace

Le développement de traces pour un processus ou une famille de processus repose sur un cycle d'observations et d'analyses qui conduit progressivement à identifier les concepts mesurables et utiles à l'analyse L'analyse permettra éventuellement de trouver de nouveaux concepts utiles à la compréhension de l'objet d'observation, qui, à leur tour, pourront amener à introduire de nouvelles mesures et de nouveaux concepts mesurables. L'identification des concepts est liée à une activité de modélisation et d'abstraction et leur transcription dans une trace relève de la recherche de leur représentation sous une forme aisément transmissible, mais aussi utilisable pour les analyses.

La méthode considérée se décompose en trois démarches imbriquées.

La première démarche peut être vue comme l'élaboration d'une "trace première", elle-même définie comme la décomposition, et représentation sous forme d'une

suite d'événements, de l'évolution des concepts représentant l'état du système. Ceci conduit à définir deux formes de traces. L'une, dite virtuelle, décrit l'évolution du système observé et l'autre, dite effective, en est une représentation. Il est essentiel que la représentation, la trace effective, soit aussi fidèle que possible au modèle original, la trace virtuelle. La raison pour laquelle la suite des états possibles du système observé est appelé ici trace virtuelle vient de la possibilité d'utiliser indifféremment l'une ou l'autre puisqu'elles sont toujours équivalentes.

La deuxième démarche consiste à définir un certain niveau d'observation. Dans le processus de recherche d'une trace utile, on peut être amené à découvrir de plus en plus de caractéristiques utiles, de nouveaux détails comme de nouveaux concepts, qui peuvent amener à mettre de plus en plus de choses dans la trace. Ce processus, qui ne peut être limité a priori, peut conduire à faire croître la trace considérablement. La trace obtenue ainsi sera appelée "trace intégrale", c'est à dire une trace primitive qui contient tous les niveaux de détails possibles et qui également engendre des cascades d'événements très "rapprochés", c'est à dire pour lesquels la différence entre deux états successifs est minimale. La trace intégrale est la trace la plus précise propre à un système observé. Il s'en suit que toute observation particulière n'utilisera en pratique qu'une petite partie de cette trace, correspondant à un niveau de trace utile pour l'observation. Comme le niveau d'observation auquel correspond la trace intégrale ne peut être défini dans l'absolu (puisqu'il dépend des besoins d'analyses - lesquels ne peuvent tous être définis a priori), un niveau d'observation peut se définir comme un niveau d'abstraction. La trace intégrale correspond alors au niveau le plus précis dont les autres niveaux sont des abstractions. Elle peut être vue aussi - mais a posteriori - comme une trace première à partir de laquelle différents niveaux d'observation peuvent être définis.

Alors que les deux premières démarches aboutissent à des traces et niveaux arbitrairement figés, la troisième démarche correspond aux modifications qui peuvent être apportées aux traces dans un processus de construction dynamique. Cette démarche est décrite par les transformations ou les combinaisons que l'on peut réaliser sur des traces. Par exemple la définition de la trace commune à plusieurs phénomènes observés et déjà pourvus de leurs propres traces ou l'observation de systèmes complexes qui amènent à définir plusieurs traces intégrales conrrespondant à différents points de vue, lesquelles sont finalement "fusionnées", puis abstraites à nouveau.

### 3.1 Traces virtuelle et effective

Il est commun de comprendre qu'un processus équipé de moyens d'observation discrète que nous appellerons "traceur" produise une "trace". Il s'agit ici de ce que nous appelons la trace effective, celle qui a un sens physique. Il est moins évident de lui associer une autre trace que nous qualifierons de "virtuelle" car elle n'a pas de consistence physique mais a un caractère conceptuel. La trace virtuelle peut être vue de deux manières : comme une suite d'états abstraits du processus, ou comme une interprétation de la suite des événements de trace effective. C'est cette vision de la suite des états observables d'un processus

comme une forme de trace en correspondance forte avec la trace effective qui lui donne également un statut de trace. C'est ces rapports entre ces formes de traces que nous voulons formaliser ici. Si maintenant on part de la trace effective pour essayer de comprendre le processus qui lui a donné naissance, on voit qu'il s'agit alors de retrouver - on dira "reconstruire" - la séquence d'états correspondant à la trace virtuelle ; la trace effective n'est qu'une représentation de la trace virtuelle.

Le point de vue commun consiste à considérer que l'observation d'un phénomène complexe, fut-il ou non totalement décrit, "formalisé" ou "mathématisé", relève d'une "simple" abstraction. Toutes traces constituée - au moins pour celles pouvant se traduire par une forme d'échantillonnage ou de discrétisation - résulterait donc d'un processus d'abstraction dont le résultat serait la trace dite ici "effective". De ce point de vue, la trace effective est une abstraction du processus observé. La question qui se pose alors est celle - assez classique- de la relation entre le processus observé et sa trace, ainsi que les propriétés que l'on peut en inférer.

Dans notre approche nous considérons qu'il faut distinguer deux étapes dans la démarche d'observation : la recherche d'une forme d'observation, celle que nous appelons "trace virtuelle", et sa formalisation pratique, sa concrétisation, ou sa représentation, sous forme de nouveaux objets. La trace concrète n'est pas vue ici simplement comme une abstraction, mais comme une abstraction suivie d'un codage.

Le fait d'avoir une vue conservative entre les traces virtuelle et effective se justifie directement par l'introduction de la sémantique interprétative, comme sémantique de la trace, c'est à dire la possibilité d'identifier - on dira reconstruire - la trace virtuelle à partir de la trace effective. Certes si l'on ne connait pas a priori les objets dont la trace effective est supposée tracer l'histoire (c'est à dire l'état virtuel traduit dans la trace), on entrera dans un processus de découverte ou d'invention. Il s'agit alors de rechercher un modèle susceptible de produire une telle trace, lequel sera susceptible de la reproduire "intégralement". Selon notre terminologie on recherchera un modèle "fidèle".

Ce que nous voulons préserver dans cette approche, c'est la symétrie des démarches qui consistent d'un côté à extraire une trace d'un processus observé et de l'autre à inférer un processus à partir de traces d'observation. Ces démarches sont en fait complémentaires et souvent liées. Il se peut en effet que lors du développement d'un traceur pour un processus connu et bien contrôlé, l'observation des traces produites suggère l'introduction de nouveaux paramètres dans sa trace, impliquant de la sorte une modification de la sémantique observationnelle.

### 3.2 Trace intégrale

Comme on l'a vu,on peut considérer, à un instant donné, qu'il existe une trace virtuelle "ultime", celle qui idéalement rend compte de tous les détails

d'UNE réalité supposée, observée. Ce point de vue nous amène précisemment à introduire la notion de trace effective comme distincte d'une abstraction directe de la trace virtuelle ultime. En effet les différents niveaux d'abstractions possibles correspondent alors à différentes traces virtuelles possibles. Celles-ci peuvent être vues comme autant de points de vue distincts, ou différents degrés de raffinement d'observations. Dans cette approche l'usage du terme de "trace virtuelle" se justifie car la trace virtuelle "intégrale" correspond en fait à un niveau d'observation relatif et a priori arbitraire ; cela veut dire aussi que ce niveau peut évoluer dans le temps (par exemple au fur et à mesure que les outils d'observation s'améliorent ou que les besoins d'observation se multiplient) et ceci indépendamment de l'usage d'une trace effective qui n'en est qu'une représentation particulière.

La figure 1 illustre l'aspect relatif des traces intégrales. D'un côté la SO de la trace virtuelle intégrale peut être plus ou moins abstraite (OS' sur la figure est plus abstraite que OS), et d'un autre côté une sous trace actuelle de la trace actuelle intégrale peut être vue comme une abstraction de celle-ci et munie d'une sémantique observationnelle (OS' sur la figure) qui est exactement une abstraction de la SO de la trace virtuelle intégrale de référence.

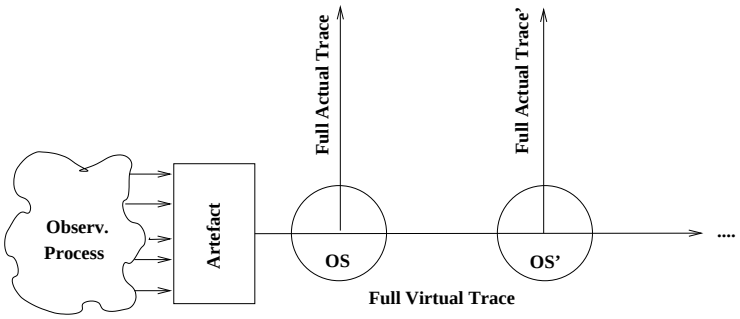


**Fig. 1.** Illustration of the OS abstraction levels

De ce point de vue, l'axe horizontal sur la figure (trace virtuelle intégrale) peut être interprété comme un axe d'abstraction, c'est à dire d'oubli des détails.

Ce qui est important de remarquer ici est le fait que cette trace effective correspond aussi à une trace virtuelle mais moins précise. L'idée majeure ici est que les deux traces (virtuelle et effective) gardent toujours un rapport de "fidélité" - concept que nous développerons ultérieurement- c'est à dire qu'il n'y a pas de perte d'information entre la trace virtuelle et la trace effective, laquelle en étant un codage "fidèle". Certes il est toujours possible de voir une trace effective moins précise comme une abstraction d'une trace virtuelle plus précise (il est toujours possible de composer abstraction et extraction, vue aussi comme une abstraction), mais il est intéressant de distinguer clairement ce qui relève

de l'abstraction pure de ce qui correspond à des besoins de représentation ou codage. Là aussi on pourrait voir la combinaison de ces deux étapes comme la composition de deux abstractions, mais nous préférons limiter la démarche d'abstraction à la première étape, liant et limitant en cela la démarche d'abstraction à la recherche de modèle d'observation.

Le modèle formel correspondant au niveau d'observation le plus raffiné - disons du moment - est décrit par ce que nous appelons une sémantique observationnelle, laquelle rend compte de suites détats virtuels possibles. Une suite chronologique d'états est nommée ici trace virtuelle. L'usage du mot trace pour justifier un modèle détat vient du fait qu'il ne s'agit pas ici de décrire un artéfact susceptible de produire effectivement des suites d'états cohérentes possibles (cohérentes signifie ici que l'on ne produirait que les seules suites observées), mais seulement de rendre compte de suites d'états observables selon un degré de précision arbitraire, mais suffisant pour pouvoir être qualifié de modèle de traceur.

### 3.3 Evolution de la trace intégrale

A un instant donné de l'élaboration d'une trace, on peut considérer qu'il y a une seule trace intégrale. Mais cette trace évolue en fonction des besoins d'analyses ou des modifications du champ d'observation. Celui-ci peut en effet intégrer de nouveaux aspects ou domaines, prendre en compte de nouveaux paramètres. La nouvelle trace est obtenu par une série d'opérations qui combinent agrégation, synthèse et oubli. La figure 2 illustre ce processus.

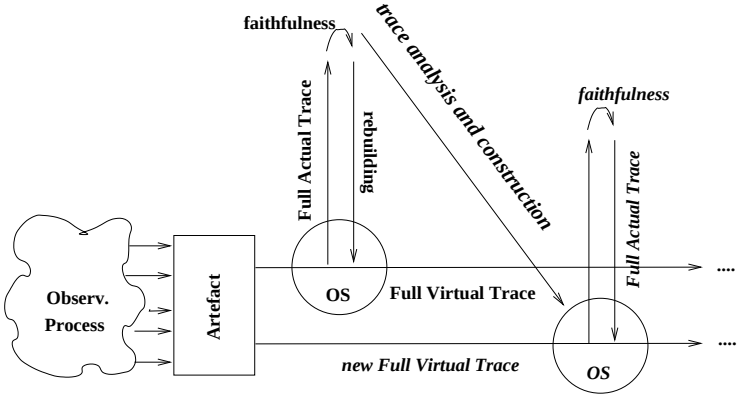


**Fig. 2.** Evolution des traces intégrales

Sur cette figure l'obtention d'une nouvelle trace intégrale est symbolisée par la flèche "trace analysis and construction". Chaque trace intégrale correspond à

une étape de construction et à chaque étape correspond un couple trace virtuelle et sa représentation, la trace effective fidèle.

### 3.4 Méthodologie

L'approche proposée ici consiste à considérer qu'à une étape de développement existe une trace intégrale dont certaines parties, définies comme sous-trace (ou trace partielle), vont être utilisées pour des analyses.

Des développements ultérieurs, obtenus par des opérations élémentaires sur et avec d'autres traces pourront conduire à proposer une nouvelle trace intégrale.

L'intérêt de la notion de trace intégrale est de permettre de voir, à un moment donné de l'élaboration d'une trace, toute autre trace comme une sous-trace, puisque par définition la trace intégrale correspond à la description la plus rafinée connue à ce moment là.

## 4 Traces intégrales contigues

On introduit ici les deux traces susceptibles d'être associées, à un moment donné de l'élaboration d'une trace, à un unique processus muni d'un traceur.

### 4.1 Trace virtuelle intégrale contigue

Une trace virtuelle intégrale est définie sur un domaine d'état. Soit $\mathcal{P}$ un ensemble fini de (noms de) paramètres $p_i$ définis sur des domaines $\mathcal{P}_i$. Les $\mathcal{P}_i$ sont des domaine d'objets quelconques. Ceux-ci peuvent également avoir des relations (fonctionnelles ou autres) entre eux et ils peuvent être de taille infinie.

Un domaine d'états $\mathcal{S}$ est défini sur le produit cartésien des domaines de paramètres : $S \subseteq \mathcal{P}_1 \times ... \times \mathcal{P}_n$.

**Definition 1 (Trace virtuelle intégrale contigue).**
*Une trace virtuelle intégrale contigue est une suite d'événements de trace qui sont de la forme* $e_t : (t, r_t, s_t),\ t \geq 1$ *où :*

- $t$ *: est le* **chrono***, temps propre à la trace. C'est un entier incrémenté d'une unité à chaque événement. Pour désigner une valeur du chrono particulière, on parlera de* moment *de la trace.*
- $r_t$ *: un identificateur d'***action** *caractérisant le type des actions réalisées pour effectuer la transition de l'état* $s_{t-1}$ *à* $s_t$*.*
- $s_t$ *est un élement du domaine d'état.* $s_t = p_{1,t}, ..., p_{n,t}$ *est l'état courant atteint au moment* $t$*, et les* $p_{i,t}$ *sont les valeurs des* **paramètres** $p_i$ *au moment* $t$*.* $s_t$ *est* l'état virtuel intégral courant.

Une trace virtuelle finie de $t$ ($t > 0$) événements sera dénotée $T_t^v =< s_0, \overline{e_t} >$, où $s_0$ est l'état virtuel intégral initial et $\overline{e_t}$ représente la suite $e_1 \dots e_i \dots e_t$.

L'ensemble de toutes les traces virtuelles possibles (paires constituées d'un état virtuel initial et d'une séquence d'événements de trace virtuelle) sera dénoté

$\mathcal{T}^v$ ; L'ensemble de toutes les traces virtuelles possibles de taille finie $t$ sera dénoté $\mathcal{T}_t^v$.

La trace virtuelle intégrale est *contigue* dans la mesure où tous les moments de l'intervalle $[1..t]$ sont présents dans tous les événements d'une trace $T_t^v =< s_0, \overline{e_t} >$.

### 4.2 Trace effective intégrale contigue

La trace virtuelle intégrale représente ce que l'on souhaite ou ce qu'il est possible d'observer d'un processus donné. Elle décrit par étapes successives l'évolution de ce processus sous la forme de l'évolution d'un état qui contient les observables. Comme l'état courant (virtuel) du processus peut être intégralement représenté dans cette trace, on ne peut espérer ni la produire ni la communiquer efficacement. En pratique on effectuera une sorte de "compression" de l'information véhiculée par les états virtuels et leur évolution, communiquée ou communicable aux processus observateurs, et on s'assurera que ces processus puisse la "décompresser". Cette information effectivement communiquée est la *trace effective*.

Une trace effective intégrale est définie sur un domaine d'états effectifs. Soit $\mathcal{A}$ un ensemble fini de (noms d') attributs $a_i$ définis sur des domaines d'attributs $\mathcal{A}_i$. Les attributs peuvent avoir des relations entre eux (ils ne sont pas forcément indépendants) et ils peuvent être de taille infinie.

Un domaine d'états effectifs $\mathcal{A}$ est défini sur le produit cartésien des domaines d'attributs : $\mathcal{A} \subseteq \mathcal{A}_1 \times ... \times \mathcal{A}_n$.

**Definition 2 (Trace effective intégrale contigue).**

*Une trace effective intégrale contigue est une suite d'événements de trace de la forme*
$w_t : (t, a_t),\ t \geq 1$ *où*
$t$ *est le chrono et* $a_t \in \mathcal{A}$ *dénote une suite finie de valeurs d'attributs.* $a_t$ *est l'état effectif intégral courant. Le nombre d'attributs des événements de trace a une borne* $n$*. Chaque état* $a_t$ *contient au plus* $n$ *attributs dont le nombre dépend exclusivement du type d'action l'ayant produit.*

Une trace effective finie de $t$ ($t > 0$) événements est dénotée $T_t^w =< s_0, \overline{w_t} >$, où $s_0$ est l'état virtuel initial commun aux deux traces et $\overline{w_t}$ représente la suite $w_1, \ldots w_i, \ldots, w_t$.

L'ensemble de toutes les traces effectives possibles (paires constituées d'un état initial et d'une séquence d'événements de trace effective) sera dénoté $\mathcal{T}^w$ ; l'ensemble de toutes les traces effectives possibles de taille finie $t$ sera dénoté $\mathcal{T}_t^w$.

La trace virtuelle intégrale est un cas particulier de trace effective où les attributs sont le type d'événement et les paramètres, i.e. $\forall t, a_t = (r_t, s_t)$.

### 4.3 Signature

On appellera *signature* d'une trace (virtuelle ou effective), la suite des types d'action occurrant dans la suite des événements de la trace virtuelle. L'ensemble des signatures finies d'un traceur forme un langage dont le vocabulaire est $R_0$.

*Example 1.* Dans l'exemple de la section 17, il n'y a qu'un type d'action mg, le langage de la signature est le langage régulier $\mathsf{mg}^*$.

Dans l'exemple de la section 18, le vocabulaire de la signature est
{Init, Leafreached, Lfrcd&godown, Treesuccess, Treesuc&goright, Tree failed, Backtrack, Bkt&gd, Final},
et le langage des signatures terminant par Final est algébrique.
○

Dans la suite on notera parfois les traces par leur signature en omettant le chrono et l'état virtuel.

## 5 Sémantiques des traces

On établit ici les sémantiques associées aux traces effective et virtuelle, ainsi que leurs relations.

### 5.1 Sémantique interprétative (SI) d'une trace effective

Toute trace effective a une interprétation qui lui donne un sens en associant à chacun de ses événements un événement de la trace virtuelle.

La sémantique interprétative utilise une *fonction d'interprétation* $I$, également dite *fonction de reconstruction*. L'interprétation associe à une trace effective intégrale un type d'action susceptible d'avoir engendré cet état et un état virtuel intégral.

**Definition 3 (Sémantique interprétative d'une trace effective intégrale contigue).**
*SI* : $< A, R_O, S, I >$ *où*

- $A$ *ensemble d'états effectifs,*
- $R_O$ *est un ensemble fini de type d'événements de trace (types d'actions provoquant un changement d'état),*
- $S$ *ensemble d'états virtuels,*
- $I$ *fonction d'interprétation (ou de reconstruction) des états effectifs*

  $$I : \mathcal{T}^w \rightarrow \mathcal{T}^v$$

  *qui, à chaque trace effective finie* $\mathcal{T}_t^w$ *associe la trace virtuelle correspondante* $\mathcal{T}_t^v$.

$I$ est un *schéma d'interprétation*, ou *schéma de reconstruction.*

La sémantique interprétative constitue un premier niveau de compréhension de la trace effective qui consiste à traduire la trace, extraite du processus observé,

en une autre trace, dite "trace virtuelle". C'est elle qui donne le sens recherché aux valeurs brutes observées. Par exemple, un éclaireur avisé pourra reconnaître dans des marques régulières sur le sol une succession d'empreintes de pas laissées par un animal ou un humain, et saura les distinguer, en les interprétant avec d'autres indicateurs (mouvement, forme des pieds, alternance droits/gauches, vitesse de progression ...). Les informations recueillies seront dites "virtuelles" car non présentes dans les "mesures", mais déduites de celle-ci, à partir de l'observation "brute"[1].

La fonction de reconstruction existe toujours en ce sens que la trace effective peut être identique à la trace virtuelle (fonction identité). En pratique ce sera le cas pour certains paramètres qui se retrouveront tel quels comme attributs et dont la signification pourra être connue qu'à travers la sémantique observationnelle.

Si ce n'est pas le cas, chaque paramètre qui n'est pas égal à un attribut doit pouvoir être défini, et c'est une première condition, à partir d'un ou plusieurs attributs présents dans la trace effective.

Une seconde condition est liée à la "constructibilité" des paramètres, reconstruits en n'utilisant que la trace connue au moment $t$ et non la trace à venir. Considérons par exemple un paramètre de la trace virtuelle dénotant un ensemble de taille $n$, seulement représenté dans la trace effective par un attribut qui dénote un élément ajouté ou retranché de l'ensemble. Si cet ensemble apparaît à un moment $t > 0$ dans un état virtuel (non initial), mais n'est jamais donné dans la trace effective on ne pourra connaître l'ensemble complet d'origine que si à un moment donné il devient vide. Au mieux, et s'il n'y a que des retraits, il faudra $n$ événements de trace effective pour reconstruire la valeur du paramètre dans le trace virtuelle. Il n'est dont pas possible au moment $i < n$ de construire le paramètre courant "ensemble" de la trace virtuelle sans connaître les $n - i$ événements suivants de la trace effective. En général, et en l'absence d'information sur la taille de l'ensemble (qui peut être non borné ou de taille infinie), cette reconstruction n'est pas possible.

On dira qu'un paramètre, ou un type d'action, est $k$-constructible si sa reconstruction est possible à tout moment avec au plus $k$ événements de trace effective au delà du moment courant. Si tous les paramètres, y compris les actions, sont $k$-constructibles ($k$ est alors le maximum des $k$ pour tous les paramètres reconstruits), on dira que la SI est $k$-constructible, et pour obtenir une trace virtuelle de taille $t$, il faut alors une trace effective de taille au plus $t + k$.

On étudiera les conditions nécessaires pour qu'une SI soit 0-constructible. Par exemple, un paramètre correspondant à un objet tracé introduit dés l'état initial, et décrit dans la trace effective de telle manière que son évolution soit complètement décrite par ses attributs, est 0-constructible. Ainsi, si le paramètre

[1] Le calcul des observables peut être arbitrairement complexe. Selon sa complexité, l'observable figurera dans la trace ou ne pourra être connue que par une analyse ultérieure plus poussé. La différence entre les observables figurant dans la trace et celles compréhensibles après une phase d'analyse spécifique est arbitraire et relève de considératrions pratiques.

dénotant un ensemble évolue à partir d'une valeur initiale figurant dans l'état initial de la trace virtuelle, alors il est possible de connaître son état à chaque moment à partir de la seule suite des ajouts/retraits d'éléments figurant comme attribut de la trace effective.

Afin de pouvoir en donner une représentation finie de la fonction de reconstruction $I$, on utilisera une fonction locale $I_l : A^+ \texttt{ x } S \rightarrow R \texttt{ x } S$. On note $\overline{a_k}$ un éléments de $A^*$ ($k \geq 0$, séquence d'événements de trace effective), et $a_{t,k}$ la séquence $\overline{a_{k-1}}$ débutant au moment $t$, soit $a_t ... a_{t+k}$. $a_{t,0} = a_t$. La fonction de reconstruction locale $I_l$ est définie :

$\forall t > 0, I_l(s_{t-1}, a_{t,k}) = (r_t, s_t)$

L'idée est que la reconstruction consiste à retrouver la trace virtuelle complète, à partir d'un état virtuel initial donné et des règles appliquées récursivement à un état virtuel reconstruit courant. $k$ est supposé fixé et est en général limité aux valeurs 0 ou 1. En effet, en toute généralité la reconstruction d'un état virtuel peut nécessiter la suite non bornée de tous les événements de trace effective. Cependant il est en théorie toujours possible d'introduire des attributs suffisamment complexes qui évitent ainsi de se référer à des attributs passés. L'utilisation de l'état reconstruit antérieur permet donc en fait de n'utiliser, en général, qu'un événement de trace effective.

La fonction de reconstruction $I : \mathcal{T}^w \rightarrow \mathcal{T}^v$ peut alors être définie à partir de $I_l$ de la manière suivante. On note $w^a_{t,k}$ la séquence des états effectifs contenue dans la séquence des événements effectifs $w_{t,k}$, et respectivement $I^r_l$ et $I^s_l$ les premiers et second éléments du codomaine de $I_l$.

$\forall t > 0, I(< s_0, \overline{w_{t+k}} >) =< s_0, \overline{e_t} >$, avec
$\forall i, 0 < i \leq t$, $e_i = (i, I^r_l(s_{i-1}, w^a_{i,k}), I^s_l(s_{i-1}, w^a_{i,k}))$

**Remarque [Schéma de reconstruction]**
La SI décrit une manière de construire une trace virtuelle à partir d'une trace effective, c'est à dire d'un flot d'observations élémentaires. A ce titre elle réalise une fonction de production de trace en "imaginant" en quelque sortes (ici, par un calcul) des actions et des objets dont la trace virtuelle obtenue modèle l'évolution. A ce titre la SI joue le rôle de modèle ou d'hypothèse d'interpretation de traces.

### 5.2 Sémantique observationnelle (SO)

La Sémantique observationnelle comporte deux parties : une fonction de transition d'états et une fonction d'extraction de trace.

La première partie constitue un modèle abstrait de l'enchainement des événements de trace virtuelle. Celui-ci est définie comme un système de transition étiqueté (Labelled Transition System ou LTS). C'est une sémantique de la trace virtuelle en ce sens que, étant donnée une trace virtuelle intégrale $T^v_t =< s_0, \overline{e_t} >$, elle "explique" la succession des événements $e_t$ par une fonction de transition appliquée récursivement à partir d'un état virtuel initial.

Il est important d'observer qu'une telle fonction de transition n'existe pas toujours, en particulier que l'évolution de certains paramètres de l'état virtuel ne peut s'expliquer simplement à l'aide d'une fonction de transition (ce serait le cas en général d'un attribut "temps CPU"). Dans le cas où elle existe, c'est à dire que l'on peut décrire l'évolution de la plupart des paramètres, la SO constitue alors une forme de sémantique du traceur.

La seconde partie, la fonction d'extraction, produit ce que l'on est susceptible de "voir" effectivement du processus observé. Cette fonction a comme arguments l'état courant et le type d'action, et produit les attributs de la trace effective.

**Definition 4 (Sémantique observationnelle).**
*Une sémantique observationnelle est définie par le n-uplet* $< S, I_f, R_O, A, E, T, S_0 >$ *où*

- $S$ *est un* domaine d'états virtuels*, où chaque état est décrit par un ensemble de paramètres,*
- $I_f$ *est un* domaine d'influences*, ensemble de* facteurs d'influence*. Un facteur d'influence est un paramètre dont les valeurs appartiennent à un domaine d'influence mais dont l'évolution n'est pas décrite dans la SO. Toute relation faisant intervenir des paramètres et au moins un facteur d'influence est une* relation d'influence*.*
- $R_O$ *est un ensemble fini de type d'action, ensemble d'identificateurs utilisés comme étiquettes pour les transitions.*
- $A$ *est un* domaine d'état effectifs*, où chaque état est décrit par un ensemble d'attributs,*
- $E_l$ *fonction d'extraction locale de l'état effectif a lors d'une transition r à partir de l'état s,* $E : R \texttt{ x } S \rightarrow A$ *qui vérifie par définition,* $E_l(r, s) = a$ *(*$a \in A$*, ensemble d'états effectifs). Plus précisemment, l'ensemble des attributs de l'évévement t de la trace effective est dérivé de l'état courant au moment* $t-1$ *de la trace virtuelle et de la transition étiqueté par le type d'action* $r_t$*, soit*

$$E_l(r_t, s_{t-1}) = a_t$$

- $T$ *fonction de transition d'état* $T : R \texttt{ x } S' \texttt{ x } I_f \rightarrow S'$*, tel que* $S' \subseteq S$*, l'inclusion portant sur une sous ensemble des paramètres.*
- $S_0 \subseteq S$*, ensemble des états initiaux.*

*Une SO est* complète *pour une trace virtuelle, si le domaine d'état utilisé dans la fonction de transition est égal au domaine d'état de la trace virtuelle. Elle est* partielle *si elle concerne un sous-ensemble stricte des paramètres du domaine d'état de la trace virtuelle concernée.*

A la fonction d'extraction locale $E_l$, il peut être associé une fonction d'extraction $E : \mathcal{T}^v \rightarrow \mathcal{T}^w$ qui a toute trace virtuelle $T_t^v =< s_0, \overline{e_t} >$ associe l'unique la trace effective $T_t^w =< s_0, \overline{w_t} >$, construite avec $E$ et définie par :

$E(< s_0, \overline{e_t} >) =< s_0, \overline{w_t} >$, où
$\forall i, i \leq t \ , \ e_i = (i, r_i, s_i) \ \Rightarrow \ w_i = (i, E_l(r_i, s_i))$

La définition de la SO peut se décomposer en deux parties : une partie ne mentionnant pas l'extraction qui décrit en quelque sorte la "machine abstraite" produisant ou "justifiant" la trace virtuelle, que l'on appellera *schéma de traceur*.

Un schéma de traceur est donc caractérisé par $< S, I_f, R_O, T, S_0 >$ où
- $S$ *domaine d'états virtuels*,
- $I_f$ *domaine d'influences*,
- $R_O$ *ensemble fini de types d'action*,
- $T$ *fonction de transition d'état* $T : R \texttt{ x } S \texttt{ x } I_f \rightarrow S$,
- $S_0 \subseteq S$, *ensemble des états initiaux*.

Celui-ci correspond à un "labelled transition system" (LTS), dont les transitions sont étiquetées par les actions $R_O$. Il spécifie un ensemble de traces virtuelles. La séquence des actions correspondant à une trace virtuelle, nommée ici "signature" (voir section 4.3) est parfois juste appelé *trace* [48,37].

Un LTS est dit *déterministe* si la fonction de transition $T$ est déterministe. Il est dit *complet* si tous les paramètres d'un nouvel état sont calculés à partir de l'état courant et de l'action qui a produit la transition.

La deuxième partie correspond à la description de la fonction d'extraction pour chaque type d'action et constitue le *schéma de trace*.

Un schéma de trace est donc caractérisé par $< S, R_O, A, E >$ où
- $S$ *domaine d'états virtuels*,
- $R_O$ *ensemble fini des type d'action*,
- $A$ *domaine d'états effectifs*,
- $E$ *fonction d'extraction de l'état effectif*.

Noter que la fonction d'extraction n'utilise pas de condition d'influence.

**Remarque 1** **[Fonctionnalité de $T$ et non déterminisme de la SO]** Dans la SO, la fonction de transition $T$ utilise un domaine d'influence qui ne se retrouve pas dans le codomaine. La conséquence pratique est que $T$ peut ne pas être une fonction, et que pour un même état de départ, l'état d'arrivée peut dépendre de l'influence. La transition, vue du seul point de vue de l'état observable, est alors non déterministe. Comme nous n'imposons pas a priori que la SO corresponde à un automate déterministe, l'introduction de facteurs d'influence ne fait qu'introduire un élément de non déterminisme supplémentaire.

**Remarque 2** **[Schéma de traceur]**

La distinction, au sein de la SO, de deux schémas descriptifs est essentielle. On peut voir en effet chaque schéma indépendamment de l'autre, mais en même temps ils sont liés. Ils sont en effet liés parce que seules les informations extraites ou objets et quantités observables dans la trace effective ont une réalité physique et que leur existence dépend tant de l'existence d'un traceur que de sa capacité à les extraire. Ces deux fonctions, production des objets observables (trace virtuelle) et communication (trace effective), sont a priori celles d'un même module, le traceur. Pour autant, si on admet l'existence de cet objet primal qu'est la trace virtuelle, les deux fonctions peuvent être vues indépendemment. La trace virtuelle est alors l'objet intermédiaire antre deux sous-modules du traceur, l'un

toujours appelé traceur et l'autre appelé "pilote" [62], et les fonctions de transition et d'extraction peuvent faire l'objet de descriptions séparées.

Chaque composant du traceur (traceur et pilote) peut être plus ou moins connu. De ce point de vue la fonction d'extraction existe toujours car elle peut se réduire à l'identité (le type d'action étant codé comme un attribut), et la fonction de transition peut être ignorée.

On verra aussi que le module traceur peut être spécifié indépendemment de tout processeur.

Langage de signature d'une SO dans laquelle aucun schéma de traceur n'est fourni peut se définir a priori comme le monoide libre construit avec $R_O$.

### 5.3 Représentations des SI et SO

On s'intéresse ici à la manière de représenter ces sémantiques de manière finie. On en ignorera cependant, pour l'instant, les aspects langage. Non que ces aspects soient sans importance (la forme de la trace et les formes d'interrogation ou de filtrage ont un rôle important à jouer pour les applications potentielles), mais ils peuvent être examinés dans un second temps.

#### Représentation de la SI : schéma de reconstruction

La SI est donnée par $< A, R_O, S, I >$ (voir plus haut) où la fonction de reconstruction $I : \mathcal{T}^w \rightarrow \mathcal{T}^v$ est donnée par une fonction de reconstruction locale $I_l(s, a_{k+1}) = (r_l, s')$.

Un schéma de reconstruction (fonction $I_l$) décrit de quelle manière, à partir d'un état virtuel reconstruit et de quelques événements de trace effective (au plus $k+1$ événements contigus), un nouvel état virtuel peut être reconstruit.

- **AType** : (Type d'action) un identificateur $r_l \in R_O$ (le type d'action reconnu correspondant à la transition $(s, s')$ ci-dessus et identifiée par la condition ci-dessous)
- **Utrace** : { (Used Trace) les attributs utiles des événements de trace effective utilisés concernant au plus $k+1$ événements }
- **OVState** : { (Original Virtual State) les éléments utiles de l'état virtuel-courant reconstruit $s$ }
- **AICond** : { (Action Identification Condition) la condition d'identification du type d'action de la transition effectuée }
- **RVState** : { (Reconstructed Virtual State) les calculs des éléments du nouvel état virtuel reconstruit $s'$}

*Example 2.* On donne ici la règle Bkt&gd de l'exemple de la section 18 et la règle du schéma de reconstruction correspondante.

**AType** :Bkt&gd
**Utrace** : { $< r \;\; l \;\;$ **Redo** $\;\; p \;\; ch >$ ; $< r' \;\; p'$ **Call** $\;\; lc >$ }
**OVState** : { $v \leftarrow nu^{-1}(r)$}

**AICond** : { } (la condition implicite ici est $port' =$ **Call**, une condition équivalente serait $r' > r$)
**RVState** : { $u' \leftarrow ncpd_1(v)$, $T' \leftarrow T - \{y|y > v\} \cup \{u'\}$,
$nu' \leftarrow fupdt(nu, \{y|y > v \in T\}, \{(u', r')\})$,
$pd' \leftarrow fupdt(pd, \{y|y > v \in T\}, \{u', p'\})$,
$cl' \leftarrow fupdt(cl, \{y|y \geq v \in T\}, \{(v, mrkcc(cl(u), ch)), (u', lc)\})$,
$fst' \leftarrow fupdt(fst, \{y|y > v \in T\}, \{(u', true)\})$
$n' \leftarrow r'$, $ct \Rightarrow (ct' \leftarrow false)$, $flr' \leftarrow false$ }

Explications : (voir l'exemple de la section 18) l'état virtuel reconstitué est l'état virtuel courant qui comporte 9 paramètres (la SO est complète), l'état initial étant connu. Le paramètre essentiel est l'arbre de preuve étiqueté courant. Sa valeur initiale est constituée d'un nœudracine dénoté $u = \epsilon$. Sans donner ici la signification des paramètres, on rappelle ici leurs noms : $S : \{T, u, n, nu, pd, cl, fst, ct, flr\}$.

- **AType** : Correspond au cas : "retour arrière" et développement d'une branche descendante.
- **Etrace** : La trace est 1-constructible. On utilise donc deux événements de trace successifs (états $a, a'$, mais seuls trois atttributs de $a'$ sont utilisés.
- **OVState** : On note ici un calcul intermédiaire concernant l'état virtuel courant reconstruit.
- **AICond** : Le type d'action correspondant aux conditions ci-dessous est reconnu. Le LTS étant déterministe, et compte-tenu des ports présents dans la trace (ici **Redo**), il faut choisir entre deux types d'action seulement : augmentation du nombre de nœuds(Bkt&gd) ou stabilité (Backtrack). Noter que la prise en compte du port **Call** dans le deuxième événement est redondante et pourrait aussi servir de critère de choix, car dans les cas d'une action de type Backtrack, les types d'action des événements de trace suivants ne peuvent être **Call**.
- **RVState** : Le nouvel état est reconstruit.

∘

### Représentation de la SO : schémas de traceur et de trace

La SO est décrite par un ensemble fini de "règles", une par type d'événement de $R_O$, et comportant 5 éléments descriptifs.

Un état virtuel $s$ (resp. une influence $i_f$), est défini par un ensemble de variables ou *paramètres* (resp. *facteurs d'influence*), et un état effectif $a$ est défini par un ensemble de variables ou *attributs*.

Chaque règle de la SO se présente donc de la manière suivante :

- **AType** : un identificateur $r \in R_O$
- **ACond** : { calculs auxiliaires à partir de l'état virtuel courant et conditions d'exécution de l'action, portant sur l'état virtuel courant : une formule logique du premier ordre utilisant des prédicats sur les paramètres }

- **ECond** : { conditions externes : conditions minimale d'exécution de l'action faisant intervenir au moins un facteur d'influence et ne pouvant pas de ce fait figurer dans ACond}
- **VSEffect** : { l'effet de l'action $r$ sur l'état courant $s$, résultant en un nouvel état $s'$, et calculs auxiliaires concernant les attributs de la trace effiective extraite }
- **Etrace** : { les attributs de la trace produite par l'action $r : a$, état effectif extrait. Au cun paramètre d'influence ne dot être utilisé dans cette partie}

Une remarque s'impose concernant ce que nous appelons ici "influence". L'influence est une condition d'exécution d'une action, mais elle porte sur un ensemble de paramètres dont certains ne font partie d'aucun état de $S$. Leur description ne fait pas partie de la SO. Ils entretiennent un lien avec le processus observé, mais ce lien n'est pas utile pour décrire la sémantique du traceur. Du point de vue de la SO, lorsque cette transition a lieu, on doit alors supposer que la condition d'influence est vérifiée.

**Schéma de traceur et schéma de trace** En pratique on pourra présenter le schéma de traceur et le schéma de trace séparément. Le schéma de traceur correspond simplement à tous les items de la présentation à l'exception du dernier (**Etrace**).

Un shéma de trace ne décrit que l'extraction. Il comportera donc une règle pour chaque action de $R_O$ etn'aura que 3 items.

- **AType** : un identificateur $r_l \in R_O$
- **VSEffect** : {calculs à partir de l'état virtuel courant et description de l'effet de l'action $r$ sur l'état courant $s$, résultant en un nouvel état $s'$ }
- **Etrace** : { les attributs de la trace produite par l'action $r : a$, état effectif extrait}

*Example 3.* On donne ici une règle de l'exemple de la section 18 et la version restreinte de la règle du schéma de trace correspondante (les explications se trouvent dans la section correspondante). Noter que l'arbre $T$ est toujours un argument implicite des fonctions utilisées.

**AType** :Bkt&gd
**ACond** : $\{v \leftarrow gcp(u), \quad w \leftarrow ncpd_1(v),$
$(\neg fst(u) \wedge hcp(u) \wedge \neg ft(gcp(u)) \wedge (flr \ \vee \ ct))\}$
**ECond** : $\{(scs(v, _) \wedge dcl(ncpd_2(w), c))\}$
**VSEffect** : $\{T' \leftarrow T - \{y|y > v\} \cup \{w\}, \quad u' \leftarrow w, \quad n' \leftarrow n + 1,$
$nu' \leftarrow fupdt(nu, \{y|y > v \in T\}, \{(w, n')\}), \quad flr' \leftarrow false,$
$pd' \leftarrow fupdt(pd, \{y|y > v \in T\}, \{w, ncpd_2(w)\}),$
$cl' \leftarrow fupdt(cl, \{y|y \geq v \in T\}, \{(v, mrkcc(cl(u))), (w, c)\}),$
$fst' \leftarrow fupdt(fst, \{y|y > v \in T\}, \{(w, true)\}), \quad ct \Rightarrow (ct' \leftarrow false)\}$
**Etrace** : $\{nu(v), lp(v), Redo, pd(v), ch(v)\}$

Explications :

- **AType** : (Backtrack and go down) nom d'action signifiant "retour arrière" et développement d'une branche descendante à partir du nœud courant de l'arbre de recherche.
- **ACond** : Calculs intermédaires sur l'état courant. Cette action correspond à une reprise à partir d'un nœudde choix $v$ existant dans le sous-arbre de racine $u$ (condition $hcp(u)$ (has_a_choice-point)) et à partir duquel un nouveau sous-arbre de preuve peut être développé avec une clause non éduite à un fait (condition $\neg ft(gcp(u)$, $ft$ pour "fact").
- **ECond** : La conditions externes portent sur le succès de l'unification (opération non décrite dans la SO) au nouveau nœudcourant $v$ et les clauses initiales de la nouvelle boîte$w$.
- **VSEffect** : Calculs intermédaires sur l'état courant concernant les attributs et construction du nouvel état $s'$.
- **Etrace** : Extraction de l'événement de trace effective ; l'attribut $cc(v)$ correspond à la clause choisie qui n'est donc pas un fait. Les clauses associées au nouveau nœud$w$, $cl(w)$, ne seront connue que dans l'événement de trace suivant (trace 1-reconstructible).

La règle correspondante du schéma de traceur est : **AType** :Bkt&gd

**ACond** : $\{v \leftarrow gcp(u),\quad w \leftarrow ncpd_1(v),$
$(\neg fst(u) \wedge hcp(u) \wedge \neg ft(gcp(u)) \wedge (flr \ \vee \ ct))\}$
**ECond** : $\{(scs(v, _) \wedge dcl(ncpd_2(w), c))\}$
**VSEffect** : $\{T' \leftarrow T - \{y|y > v\} \cup \{w\},\quad u' \leftarrow w,\quad n' \leftarrow n + 1,$
$nu' \leftarrow fupdt(nu, \{y|y > v \in T\}, \{(w, n')\}),\quad flr' \leftarrow false,$
$pd' \leftarrow fupdt(pd, \{y|y > v \in T\}, \{w, ncpd_2(w)\}),$
$cl' \leftarrow fupdt(cl, \{y|y \geq v \in T\}, \{(v, mrkcc(cl(u))), (w, c)\}),$
$fst' \leftarrow fupdt(fst, \{y|y > v \in T\}, \{(w, true)\}),\quad ct \Rightarrow (ct' \leftarrow false)\}$

La règle correspondante du schéma de trace est :

**AType** : Bkt&gd
**VSEffect** : $\{v \leftarrow gcp(u)\}$ (on décrit ici les calculs utiles, mais on ne donne plus le nouvel état ; l'arbre de preuve $T$ courant de racine $u$ et ses labels ($\{nu, pd, cl, fst\}$) sont implicites)
**Etrace** : $\{nu(v), lp(v), redo, pd(v), cc(v)\}$

∘

## 6 Fondements de la Sémantique Observationnelle

On présente ici quelques fondements théoriques possibles de la sémantique observationnelle (SO) utilisable pour spécifier des traces virtuelles spécifiques ou génériques. Celle-ci est définie comme un système de transitions étiqueté (LTS) auquel correspond de manière équivalente un ensemble de traces virtuelles. On propose ici une représentation utilisant la programmation en logique, mais ce n'est qu'une possiblilité. Finalement on montre comment certaines propriétés

des signatures, susceptibles d'aider à la réalisation de telles sémantiques, peuvent être établies en utilisant des techniques d'interprétation abstraite.

### 6.1 SO comme une sémantique des traces partielles

Dans [64,24] les auteurs définissent ce qui peut être vu comme une sémantique opérationnelle basée sur des traces pour un programme. Dans cette approche, un état est un couple formé d'un "environnement" où sont stockées les valeurs des variables et une "instruction" du programme, et les types d'actions correspondent donc aux instructions du programme qui engendrent des changement d'états. La fonction de transition définit tous les états successeurs possibles d'un état donné. Ceci correspond à ce qui est appelé "small step operational semantics". Cette sémantique opérationnelle définit un ensemble de toutes les traces possibles pour un programme donné. Une trace est alors constituée d'une suite de couples (environnement, instruction) et correspond dans notre terminologie à une trace virtuelle (paramètres, type d'action) possible du programme. Cette sémantique peut être vue comme une SO où la fonction de transition est décrite par le programme lui-même, et la fonction d'extraction désigne l'instruction utilisée lors de la transition. Une telle sémantique correspond à une SO complète. Des formalisations analogues de la sémantique de programmes ont été largement étudiées [54,1,71].

De manière générale, on peut appliquer cette approche pour donner une définition plus formelle à la sémantique observationnelle.

Soit une SO restreinte au 4-uplet $< S, R, T, S_0 >$ (partie schéma de traceur de la SO) :

- $S$ : *domaine des états virtuels*, éventuellement étendu avec le domaine d'influ-ence, $s \in S$.
- $R$ : *ensemble fini de type d'actions* $r \in R$.
- $T$ *fonction de transition d'états* $T : R \times S \rightarrow S$, non déterministe.
- $S_0 \subseteq S$, *ensemble des états initiaux.*

On dénote $[\![SO]\!]$ l'ensemble de toutes les traces virtuelles susceptibles d'être produites par la fonction de transition $T$.

Celle-ci s'étend à un ensemble de traces par un *opérateur de transition immédiate* $\mathcal{T}_{SO}$ appliquable à un ensemble de préfixes de traces finies $d$ (on suppose que tous les préfixes d'une trace sont présents dans $d$ et on ommet par simplicité les types d'actions) :

$$\mathcal{T}_{SO}(d) = \{tss' | ts \in d \wedge ss' \in T\}$$

Si la SO (i.e. la fonction de transition) est formalisée par un programme $P$, un opérateur sémantique, dénoté $\mathcal{T}_P$, peut lui être associé. Cet opérateur agit sur un domaine sémantique qui est en général un treilli complet, ou, plus généralement un ordre partiel complet.

La signification du programme, dénotée $[\![P]\!]$, est définie comme étant le plus petit point fixe de l'opératuer $\mathcal{T}_P$, i.e., $[\![P]\!]$ =lfp($\mathcal{T}_P$). Il peut être défini opérationnellement par $[\![P]\!] = \bigcup_{i=0}^{\infty} \mathcal{T}_P^i(\emptyset)$.

L'ensemble des préfixes finis des traces spécifiées par la SO peut alors être défini comme le plus petit point fixe d'un opérateur $\mathcal{T}_{SO}$ : $[\![SO]\!] = \text{lfp}(\mathcal{T}_{SO})$.

Si $\mathcal{T}_P$ est continu[2], le plus petit point fixe est la limite de l'application itéré au plus $\omega$ fois de l'opérateur$\mathcal{T}_P$ à partir de l'élément inférieur du treilli.

Dans une approche duale, on peut aussi considérer comme premier, l'ensemble de toutes les traces possibles qui constitue alors la seule sémantique concrète du processus observé. Si donc on "ignore" le processus qui a produit cet ensemble de traces, ou si on cherche à le modéliser par une trace générique, on peut considérer l'ensemble des traces partielles, c'est à dire de l'ensemble de tous les préfixes des traces finies ou infinies dont on dispose.

L'ensemble des traces peut alors être vu comme l'unique point fixe d'un opérateur agissant sur un sous-ensemble complet de préfixes de traces.

Soit $D$ un ensemble de traces virtuelles partielles et $\mathcal{T}_D$ l'opérateur ainsi défini ($d$ est un sous-ensemble de traces partielles de $D$ incluant tous leurs préfixes finis) :

$\mathcal{T}_D(d) = \{ts | t \in d \wedge ts \in D\}$

$D$ est de manière trivialle le seul point fixe de $\mathcal{T}_D$ (tous les points fixes contiennent $D$ et $\mathcal{T}_D$ ne contient que des traces de $D$).

Au même ensemble de traces $D$, il peut être associée une fonction de transition d'état (non déterministe $T_D$) :

$T_D(s) = \{s' | \ \exists ts \in D \text{ such that } \ tss' \in D\}$.

Noter qu'une telle fonction de transition d'état est telle que, quelle que soit la manière d'atteindre un état $s$, tous ses successeurs sont des continuations possibles. Il s'en suit que l'opérateur défini à partir de la fonction de transition $T_D$, obtenue comme l'union de toutes les transitions effectuées dans toutes les traces de $D$ :

$\mathcal{T}'_D(d) = \{tss' | ts \in d \wedge ss' \in T_D\}$

produit un sur-ensemble, soit $\mathcal{T}_D(d) \subseteq \mathcal{T}'_D(d)$. On verra que les points fixes de $\mathcal{T}'_D$ sont en général des sur-approximations de $D$ (en ce sens qu'ils contiennent $D$.

Il est donc possible d'obtenir un opérateur $\mathcal{T}'_D$ en inférant une fonction de transition $T_D$ à partir des traces de $D$.

Il est important de noter qu'une telle fonction $T_D$ (comme les opérateurs correspondants $\mathcal{T}_D$ ou $\mathcal{T}'_D$) existe toujours, mais il est cependant possible qu'elle ne puisse pas avoir une représentation finie en logique du premier ordre.

Cette sémantique, définie comme l'ensemble (ou un sur-ensemble) de tous les préfixes finis de toutes les traces, est appelé "sémantique des traces partielles". Dans le cas de programmes équipés de traceurs générant des traces virtuelles intégrales qui incluent les événements de la sémantique "small step", on voit que celle-ci est un raffinement de la sémantique transitionnelle telle définie dans [24].

[2] Soit X($\sqsubseteq, \bot, \top, \sqcup, \sqcap$) un treilli complet, la fonction $f : X \to X$ est continue si, pour tout sous-ensemble $Y$ de $X$, on a $\sqcup f(Y) = f(\sqcup Y)$.

### 6.2 Interprétation abstraite

La modélisation ou l'observation de processus à partir de leurs traces posent plusieurs types de problèmes, en particulier au stade de la conception : celui du niveau d'analyse auquel on veut ou on peut se situer et celui de la réalisation du modèle (cf. figure 1).

Nous nous intéressons ici à la réalisation du modèle. Cette section est une introduction générale à l'interprétation abstraite, limitée à son utilisation pour la recherche de propriétés utiles lors de sa conception. We borrow from [42,14,26,44] this presentation.

One of the most successful techniques for approximating the actual semantics of a program[3] is *abstract interpretation* [25]. In this technique a program is interpreted over a non-standard domain called *abstract* domain and the semantics w.r.t. this abstract domain, i.e., the *abstract semantics* of the program is computed (or approximated) by replacing the operators in the program by their abstract counterparts. An abstract semantic object is a finite representation of a, possibly infinite, set of actual semantic objects in the concrete domain ($\mathcal{C}$). The set of all possible abstract semantic values represents an *abstract domain* ($\mathcal{A}$) which is usually a complete lattice or cpo which is ascending chain finite.

In the algebraic setting of abstract interpretation, a domain is a lattice $L(\sqsubseteq, \bot, \top, \sqcup, \sqcap)$ defined by a partial order $(L, \sqsubseteq)$, where $\bot$ and $\top$, elements of $L$ and $\sqcup$, $\sqcap$, binary operators on $L$, respectively denote the least element, the greatest element, the least upper bound and the greatest lower bound. Intuitively, the partial ordering represents the information loss : the lesser the more informative, the greater the bigger loss of information.

As it is often the case in program analysis, the concrete domain and the abstract domains considered for analyzing observational semantics or virtual traces, will be power-sets, i.e. set lattices $\mathcal{P}(\mathcal{S})(\subseteq, \emptyset, \mathcal{S}, \cup, \cap)$ ordered by inclusion, with the empty set as $\bot$ element, and the base set $\mathcal{S}$ as $\top$ element.

An *abstraction* is formalized by a Galois connection between a concrete domain $\mathcal{C}$ and an abstract domain $\mathcal{A}$, as follows [25] :

**Definition 5.** *A Galois connection* $\mathcal{C} \underset{\gamma}{\overset{\alpha}{\longleftrightarrow}} \mathcal{A}$ *between two lattices* $(\mathcal{C}, \sqsubseteq_C)$ *and* $(\mathcal{A}, \sqsubseteq_\mathcal{A})$ *is defined by an abstraction function* $\alpha : \mathcal{C} \to \mathcal{A}$*, and a concretization function* $\gamma : \mathcal{A} \to \mathcal{C}$*, that are monotonic :*

1. $\forall\ c, c' \in \mathcal{C} : c \sqsubseteq_\mathcal{C} c' \ \Rightarrow\ \alpha(c) \sqsubseteq_\mathcal{A} \alpha(c')$
2. $\forall\ a, a' \in \mathcal{A} : a \sqsubseteq_\mathcal{A} a' \ \Rightarrow\ \gamma(a) \sqsubseteq_\mathcal{C} \gamma(a')$

   *and are adjoint :*
3. $\forall c \in \mathcal{C}, \forall a \in \mathcal{A} : c \sqsubseteq_\mathcal{C} \gamma(a) \Leftrightarrow \alpha(c) \sqsubseteq_\mathcal{A} a.$

For any Galois connection, we have the following properties :

[3] The "actual semantics" is the formal description of the meaning of the program.

4. $\gamma \circ \alpha$ is extensive (i.e. $c \sqsubseteq_{\mathcal{C}} \gamma \circ \alpha(c)$) and represents the information lost by the abstraction
5. $\alpha \circ \gamma$ is contracting (i.e. $\alpha \circ \gamma(a) \sqsubseteq_{\mathcal{A}} a$)
6. $\gamma \circ \alpha$ is the identity *iff* $\gamma$ is onto (surjective) *iff* $\alpha$ is one-to-one (bijective)
7. $\alpha$ preserves $\sqcup$, and $\gamma$ preserves $\sqcap$
8. $\gamma(a) = \max\, \alpha^{-1}(\downarrow a) = \sqcup \alpha^{-1}(\downarrow a)$
9. $\alpha(c) = \min\, \gamma^{-1}(\uparrow c) = \sqcap \gamma^{-1}(\uparrow c)$
10. the composition of two Galois connections is a Galois connection.

where $\downarrow a = \{b \mid b \sqsubseteq a\}$ and $\uparrow a = \{b \mid a \sqsubseteq b\}$.

If $\alpha \circ \gamma(a) = a$ the Galois connection is said to be a *Galois insertion*. If $\gamma \circ \alpha(c) = c$ the abstraction $\alpha$ loses no information, and $\mathcal{C}$ and $\mathcal{A}$ are isomorphic from the information standpoint (although $\alpha$ may be not onto and $\gamma$ not one-to-one).

It is equivalent in the definition of Galois connections to replace the condition of adjointness (3) by conditions (4) and (5), or by condition (8) which also entails the monotonicity of $\gamma$.

If $\sqsupseteq$ is used in (4) and (5) (hence in (3)) instead of $\sqsubseteq$, we obtain a dual construction, termed a *reversed Galois insertion*.

Furthermore we may use the fact that in powerset domains, the pointwise extension of any function from the base set of the concrete domain to the abstract domain forms a Galois connection :

**Theorem 1.** [42] *Let $\mathcal{C}$ and $\mathcal{A}$ be two sets, and $\alpha : \mathcal{P}(\mathcal{C}) \longrightarrow \mathcal{P}(\mathcal{A})$ be a function such that $\alpha(c) = \bigcup_{e \in c} \alpha(\{e\})$. Then the functions $\alpha$ and $\gamma(a) = \cup \alpha^{-1}(\downarrow a)$ form a Galois connection $\mathcal{P}(\mathcal{C}) \underset{\gamma}{\overset{\alpha}{\longleftrightarrow}} \mathcal{P}(\mathcal{A})$ between $(\mathcal{P}(\mathcal{C}), \subseteq)$ and $(\mathcal{P}(\mathcal{A}), \subseteq)$.*

We will assume in some cases the following properties for $\alpha$ and $\gamma$ :

11. $f(x) \sqcap f(y) = \emptyset \Rightarrow\ x \sqcap y = \emptyset$ and $f(x \cap y) = f(x) \cap f(y)$
    which is equivalent in a Galois connection and powerset domains to
12. $f(x \cap y) = \emptyset \Rightarrow\ x \cap y = \emptyset$

The abstract domain $\mathcal{A}$ is usually constructed with the objective of computing approximations of the semantics of a given program. Thus, all operations in the abstract domain also have to abstract their concrete counterparts. In particular, if the semantic operator $\mathcal{T}_P$ can be decomposed in lower level operations, and their abstract counterparts are locally correct w.r.t. them, then an abstract semantic operator $\mathcal{T}_P^{\alpha}$ can be defined which is correct w.r.t. $\mathcal{T}_P$. This is formalised by the following theorem.

**Theorem 2.** [26,44] *Fixpoint abstract approximation*[4]

[4] Version avec hypothèses restrictives dans lesquelles les ordres des CPOs (pour hypothèse de continuité) sont respectivement les mêmes que ceux des treillis.

*Etant donnés une connection de Galois* $\mathcal{C} \overset{\alpha}{\underset{\gamma}{\longleftrightarrow}} \mathcal{A}$ *entre deux treillis* $(\mathcal{C}, \sqsubseteq_{\mathcal{C}})$ *and* $(\mathcal{A}, \sqsubseteq_{\mathcal{A}})$*, et deux opérateurs* $\mathcal{T}_{\mathcal{C}}$ *et* $\mathcal{T}_{\mathcal{A}}$ *opérant respectivement sur les domaines concret et abstrait. Si les propriétés suivantes sont satisfaites :*

13. $\mathcal{C}$ *et* $\mathcal{A}$ *sont des CPOs (ordres partiels complets),*
14. $\mathcal{T}_{\mathcal{C}}$ *est monotone et (sup-)continu, et* $\mathcal{T}_{\mathcal{A}}$ *(sup-)continu (définition en note de section 6.1),*
15. $\alpha(\bot_{\mathcal{C}}) \sqsubseteq_{\mathcal{A}} \bot_{\mathcal{A}}$ *et* $\alpha \circ \mathcal{T}_{\mathcal{C}} \circ \gamma \sqsubseteq \mathcal{T}_{\mathcal{A}}$[5]*,*

*alors* $\alpha(lfp\mathcal{T}_{\mathcal{C}}) \sqsubseteq lfp\mathcal{T}_{\mathcal{A}}$ *et, de manière équivalente,* $lfp\mathcal{T}_{\mathcal{C}} \sqsubseteq \gamma(lfp\mathcal{T}_{\mathcal{A}})$*.*

From the point of view of observational semantics with a concrete operator denoted $\mathcal{T}^{\mathcal{C}}_{OS}$ and an abstract operator denoted $\mathcal{T}^{\mathcal{A}}_{OS}$ , $\gamma(\mathcal{T}^{\mathcal{A}}_{OS}(\alpha(x))$ is an approximation of $\mathcal{T}^{\mathcal{C}}_{OS}(x)$ in $\mathcal{C}$, and consequently, $\gamma(lfp(\mathcal{T}^{\mathcal{A}}_{OS}))$ is an approximation of $[\![OS]\!]$. We denote $lfp(\mathcal{T}^{\mathcal{A}}_{OS})$ as $[\![OS]\!]_{\mathcal{A}}$. The following relations hold :

16. $\forall x \in D : T^{\mathcal{C}}_{OS}(x) \sqsubseteq \gamma(T^{\mathcal{A}}_{OS}(\alpha(x))$,
17. $[\![OS]\!] \sqsubseteq \gamma([\![OS]\!]_{\mathcal{A}})$ equivalently $\alpha([\![OS]\!]) \subseteq [\![OS]\!]_{\mathcal{A}}$.

An abstract operator $\mathcal{T}^{\mathcal{A}}_{OS}$ is said to be *precise*, if instead it satisfies that

18. $[\![OS]\!] = \gamma([\![OS]\!]_{\mathcal{A}})$ equivalently $\alpha([\![OS]\!]) = [\![OS]\!]_{\mathcal{A}}$

Note that the construction presented allows obtaining over-approximations of $[\![OS]\!]$.

In the case of a reversed Galois insertion, the dual relations of (16) and (17) also hold in this case.

In practice, the abstract domains should be sufficiently simple to allow effective computation of semantic approximations.

### 6.3 Sémantique de programmes logiques et interprétation abstraite

In this approach we restrict ourselves to the important class of semantics of logic programs referred to as *fixpoint semantics*. We borrow from [14] this presentation.

An example of a set-based, fixpoint semantics for (constraint) logic programs is the traditional least model semantics [52]. The semantic objects in this case are so called $D$-atoms. A $D$-atom is an expression $p(d_1, \ldots, d_n)$ where $p$ is an $n$-ary predicate symbol, $d_1, \ldots, d_n \in D$ and $D$ is the domain of values.

For example, in classical logic programming $D$ is the Herbrand universe[6] ; for CLP(R) $D$ is the set of real numbers and of terms (for example lists) containing real numbers[7].

[5] Notation abrégée de : $\forall a, a \in \mathcal{A}, \alpha \circ \mathcal{T}_{\mathcal{C}} \circ \gamma(a) \sqsubseteq \mathcal{T}_{\mathcal{A}}(a)$.

[6] The actual semantics of a logic program can be represented by infinite sets of terms [36] which is defined on a lattice $\mathcal{P}(TERM)(\subseteq, \emptyset, TERM, \cup, \cap)$, where $TERM$ is the set of all ground predications (the Herbrand universe).

[7] Usually it is assumed that $D$ is given together with a fixed interpretation of the symbols that can occur in constraints. For instance for CLP(R), + is interpreted as addition and > as the "greater than" relation on reals.

The semantic operator for program $P$ is $T_P$ (the immediate consequence operator[8]) and $[\![P]\!] = \mathrm{lfp}(T_P) = \bigcup_{i=0}^{\infty} T_P^i(\emptyset)$. An important property is that $[\![P]\!]$ is the least $D$-model of the program. Any ground instance[9] of a computed answer (for an atomic query) is a member of $[\![P]\!]$.

*Example 4.* For example, given the following CLP program, over the domain of integers :

$sorted(X) \leftarrow X = [\,].$
$sorted(X) \leftarrow X = [Y].$
$sorted(X) \leftarrow X = [H1|T1], T1 = [H2|T2], H1 > H2, sorted(T1).$

we have that $[\![P]\!] = \{\, sorted([\,])\,\} \cup \{\, sorted([X]) \mid X \in D\,\} \cup \{\, sorted([X_1, \ldots, X_n]) \mid n \geq 2,\ X_1 > \cdots > X_n\,\}$. So for instance $[\![P]\!]$ contains $sorted([7])$, $sorted([a])$, $sorted([[\,]])$, $sorted([2,1,0])$ and does not contain $sorted([0,2])$, $sorted([2,1,a])$.

∘

Abstract interpretation is used to define automated proofs of some properties of the program by computing over approximations of its semantics. For example, Herbrand interpretations of some alphabet may be mapped into an abstract domain where each element represents a typing of predicates in some type system. For a given program $P$ the abstract operator $T_P^{\mathcal{A}}$ would allow then to compute a typing of the predicates in the least Herbrand model of $P$.

In this framework we essentially use the properties of the immediate consequence operator and its abstract counterpart :

19. $\forall x \in D : T_P(x) \subseteq \gamma(T_P^{\mathcal{A}}(\alpha(x))$
20. $[\![P]\!] \subseteq \gamma([\![P]\!]_{\mathcal{A}})$ equivalently $\alpha([\![P]\!]) \subseteq [\![P]\!]_{\mathcal{A}}$

*Example 5.* A simple example of abstract interpretation in logic programming can be constructed as follows. The concrete semantics (least Herbrand model) of a program $P$ is $[\![P]\!] = lfp(T_P)$. So the concrete domain is $D = \wp(B_P)$ (where $B_P$ is the Herbrand base of the program).

We consider over-approximating the set of "succeeding predicates", i.e those whose predicate indicators[10] of the predicates used in $[\![P]\!]$. A possible abstraction is as follows. The abstract domain is $D_{\mathcal{A}} = \wp(B_P^{\mathcal{A}})$, where $B_P^{\mathcal{A}}$ is the set of predicate indicators of $P$. Let $pred(A)$ denote the predicate symbol for an atom $A$. We define the abstraction function :

$$\alpha : \mathcal{C} \rightarrow \mathcal{A} \text{ such that } \alpha(I) = \{pred(A) \mid A \in I\}.$$

[8] $T_P(D) = \{d | d : -B \in Inst(P) \wedge B \subseteq D\}$, where Inst(P) is the set of all ground instances of the clauses of the logic program $P$.

[9] In CLP, by a ground instance of a constrained atom $A \leftarrow c$ we mean any $D$-atom $A\theta$ such that $c\theta$ is true ; here $A$ is an atom, $c$ a constraint and $\theta$ is a valuation assigning elements of $D$ to variables.

[10] According to the ISO-Prolog standard, a predicate indicator has the form (predicate name / arity).

The concretization function is defined as :

$$\gamma : \mathcal{A} \to \mathcal{C} \text{ such that } \gamma(I_{\mathcal{A}}) = \{A \in B_P \mid pred(A) \in I_{\mathcal{A}}\}.$$

For example,

$$\alpha(\{\texttt{p(a,b),p(c,d),q(a),r(a)}\}) = \{\texttt{p/2,q/1,r/1}\}$$
$$\gamma(\{\texttt{p/2,q/1}\}) = \{\texttt{p(a,a), p(a,b), p(a,c),..., q(a), q(b),...}\}.$$

Note that $\mathcal{C} \underset{\gamma}{\overset{\alpha}{\longleftrightarrow}} \mathcal{A}$ is a Galois insertion. The abstract semantic operator $T_P^{\mathcal{A}} : \mathcal{A} \to \mathcal{A}$ is defined as :

$$T_P^{\mathcal{A}}(I_{\mathcal{A}}) = \{pred(A) \mid \exists\, (A \leftarrow B_1, \ldots, B_n) \in P\ \forall\, i \in [1, n] : pred(B_i) \in I_{\mathcal{A}}\}.$$

Since $\mathcal{A}$ is finite and $T_P^{\mathcal{A}}$ is monotonic, the analysis (applying $T_P^{\mathcal{A}}$ repeatedly until fixpoint, starting from $\emptyset$) will terminate in a finite number of steps $n$ and $[\![P]\!]_{\mathcal{A}} = T_P^{\mathcal{A}} \uparrow n$ approximates $[\![P]\!]$. For example, for the following program $P$,

```
p(X,Y) :- q(X), r(Y).
t(X) :- l(X).
m(X) :- s(X).
q(a). q(b).
r(a). r(c). r(X).
```

we have $B_P^{\mathcal{A}} = \{p/2, q/1, r/1, s/1, t/1, l/1, m/1\}$, and :

$T_P^{\mathcal{A}}(\emptyset) = \{\texttt{q/1, r/1}\}$ $\quad$ $T_P^{\mathcal{A}}(\{\texttt{q/1, r/1}\}) = \{\texttt{q/1, r/1, p/2}\}$
$T_P^{\mathcal{A}}(\{\texttt{q/1, r/1, p/2}\}) = \{\texttt{q/1, r/1, p/2}\}$

So $T_P^{\mathcal{A}} \uparrow 2 = T_P^{\mathcal{A}} \uparrow 3 = \{q/1, r/1, p/2\} = [\![P]\!]_{\mathcal{A}}$ ∘

La sémantique “small steps”, ou transitionnelle, des traces virtuelles peut être décrite par un programme logique (non nécessairement fini) de la manière suivante. On utilise la syntaxe de Prolog standard, le chrono est omis et les traces sont représentées dans l'ordre de chrono décroissant[11].

```
%tr(s): s is a finite trace in SO

tr([(_,S)]) :- S belongs_to S_0.   %As many facts as states in S_0
tr([(R',S'), (R,S) | T_v]) :- te(S, R', S'), tr([(R,S)| T_v]).

%te(s, r, s'): s' belongs to T(r,s)

te(S, R, S') :- ...
                %As many clauses as there are action types r in R_0
                % and states in T(r,s)
```

[11] Il suffit donc d'inverser le trace obtenue pour obtenir la trace virtuelle $T_t^v =< s_0, \overline{e_t} >$ conforme à la définition donnée en section 4.

This program verifies

$$[\![P]\!] = \{\texttt{tr(n,s}_\texttt{n}\texttt{)} \text{ — } \texttt{n} \in N \text{ and } s_\texttt{n} \text{ is a trace of size } n \text{ issued from } s_0\},$$

and its immediate consequence operator allows to compute, by increasing sizes of the traces the least fixpoint which is $D$ (section 6.1, the set of all finite prefixes of traces specified by the OS.

Comme indiqué à la section 6.1, rien ne garantit cependant, dans ce formalisme, que les transitions (prédicat `te/3`) puissent être décrites avec un nombre fini de faits ou clauses

On pourrait également définir une représentation de la SO en programmation logique et avec des traces infinies et une sémantique utilisant le plus grand point fixe (voir par exemple [26,80,45] et [2,74,75,51] pour une approche générale).

### 6.4 Propriétés des signatures

L'élaboration d'une SO, c'est à dire la construction d'un modèle de traceur générique, peut être facilitée par l'observation de propriétés du modèle. Nous nous intéressons ici, à titre d'illustration de cette démarche, à une propriété simple qui consiste à réduire la sémantique de la SO, c'est à dire l'ensemble des traces virtuelles qu'elle spécifie, à un graphe d'états fini, aussi réduit que possible. Ce graphe est tel que chaque nœud correspond à un ensemble détats de la SO, ses arcs sont étiquetés par un type d'action de la SO, et l'absence d'arc entre deux nœuds implique qu'il n'y a aucune transition entre aucun des états correspondants à ces deux nœuds dans la SO. L'intérêt d'une telle représentation est multiple : il permet d'avoir une vue globale immédiate de chemins possibles entre classes détats, et il fournit des informations sur l'absence de transition entre ces classes, informations qui, soit peuvent être utilisées dans des preuves de correction ou de fidélité, soit peuvent révéler des incomplétudes dans la SO.

Ainsi dans l'exemple des robots (section 19), le fait que l'on puisse décrire l'évolution de la scène par un automate d'état fini avec deux ou six états (cf. section 19.4), tels que chaque transition puisse être étiquetée par un ou plusieurs types d'actions, donne une vision globale immédiate de la scène, qui aide à comprendre le (bon) fonctionnement du modèle.

Un autre exemple serait le cas d'un programme avec une sémantique opérationnelle "small step" telle que mentionnée au début de cette section et dont les types d'actions des événements de trace virtuelle correspondent à la dernière instruction exécutée et les paramètres correspondent aux variables. Une représentation abstraite de ses exécutions, dans laquelle on s'intéresse par exemple à une seule de ses variables qui prend ses valeurs sur une ensemble fini d'intervalles, peut être donné par un graphe dont les nœuds correspondent aux intervalles et dont les arcs sont étiquetés par l'instruction ayant conduit à l'obtention d'une valeur dans l'intervalle d'arrivée. Cela peut contribuer à la localisation rapide des causes d'obtention d'une valeur erronée (par détection de l'absence d'une transition attendue).

De manière générale, on s'intéresse ici aux propriétés qui consistent à réduire les états potentiels à un nombre fini réduit d'états abstraits par une partition des états virtuels, et à étiqueter les transitions entre les etats abstraits avec les types d'actions. Ceci constitue donc une propriété de l'ensemble des signatures des traces décrites par le LTS, c'est à dire une propriété du langage des signature des traces considérées (sur-approximation du langage des signatures).

Si l'on confond l'ensemble des traces $D$ avec l'ensemble de leurs signatures, la propriété recherchée est $D \subseteq L(G)$ où $G$ est le graphe abstrait correspondant à $D$ et $L(G)$ le langage régulier spécifié par l'automate fini défini par $G$ et étendu avec tous les préfixes finis des mots du langage.

Pour une telle analyse on utilise un domaine abstrait constitué de graphes étiquetés et dont l'ensemble des sous-graphes forme un treilli complet.Un graphe étiqueté est un triplet $< V, L, \Gamma >$ où $V$ est un ensemble de nœuds, $L$ un ensemble d'étiquettes, $\Gamma$ un ensemble d'arcs étiquetés définis comme une relation sur le produit cartésien des nœuds et des étiquettes, soit $\Gamma \subseteq V \mathrm{x} V \mathrm{x} L$.

Deux graphes sont disjoints si leurs ensembles de nœuds sont disjoints. Le graphe $G_1 :< V_1, L_1, \Gamma_1 >$ est inclus dans le graphe $G_2 :< V_2, L_2, \Gamma_2 >$ si $V_1$ et $L_1$ sont respectivement sous-ensembles de $V_2$ et $L_2$, et $\Gamma_1$ la restriction de $\Gamma_2$ à $V_1$ et $L_1$. Le graphe $G_1$ et dit sous-graphe de $G_2$. L'union de deux graphes (non nécessairement disjoints) est le graphe résultant de l'union de chaque composant. L'intersection de deux graphes $G_1$ et $G_2$ est le plus grand sous-graphe commun à $G_1$ et $G_2$. Dans la suite, nous considérerons l'ensemble des étiquettes non vide, fini et invariant. L'ensemble des sous-graphes d'un graphe donné forme un treilli complet dont le minimum est le graphe vide (tous les ensembles, sauf les étiquettes sont vides) et le maximum est le graphe donné ; on le notera $\mathcal{G} : \mathcal{P}(G)(\subseteq, \emptyset_g, G, \cup, \cap)$.

On définit maintenant une connexion de Galois (en fait une insertion) $\mathcal{P}(D) \underset{\gamma}{\overset{\alpha}{\longleftrightarrow}} \mathcal{P}(G)$
entre un domaine concret de traces $\mathcal{D}$ et un domaine abstrait de graphes $\mathcal{G}$.

Le domaine concret est le treilli construit avec l'ensemble des traces spécifiées par une SO, donc l'ensemble des traces produites par une SO ($D = [\![SO]\!]$), qui, dans une vision duale, peut être donné a priori. Par hypothèse il contient tous les préfixes finis et toutes les traces sont issues d'un état initial.

L'ensemble des sous-graphes du graphe abstrait construit à partir de l'ensemble de traces $D$ constitue un domaine abstrait $\mathcal{A}$.

Les fonctions d'abstraction et de concrétisation sont donc définies relativement à un ensemble de traces donné $D$. Dans tous les cas on considérera qu'il contient tous les préfixes finis de chaque trace.

On notera d'abord qu'à tout ensemble (singleton, fini ou infini) de traces virtuelles (finies ou infinies) $D$ on peut associer un graphe (fini ou infini)
$< S, R, \Gamma >$, où $S$ est l'ensemble des états virtuels apparaissant dans les traces, R l'ensemble (supposé fini ici) des types d'événements et $\Gamma = \{(s, s', r) | \exists t, e, e', t',\ tee't' \in$

$D \wedge e = (r, s) \wedge e' = (r', s')\}$. Le graphe obtenu constitue en général une abstraction. Dans ce cas, à chaque état correspond un nœud du graphe.

Plus généralement on va considérer que l'ensemble des états virtuels (c'est à dire ceux rencontrés dans les traces) peuvent être partitionnés de manière à ce que l'on puisse associer à chaque partie, de manière bi-univoque, un nœud du graphe. Noter qu'une telle partition est toujours possible car elle peut se réduire à l'ensemble des états $S$ caractérisé par la formule toujours vraie *true*.

Une telle partition sera spécifiée par un ensemble de formules logiques $\Phi$ tel que tout état virtuel satisfait au moins une et une seule formule de $\Phi$. On introduit alors un ensemble de nœuds abstraits en bijection avec $\Phi$. On dira qu'un nœud abstrait $v$ est une abstraction d'un état virtuel concret $s$, et on notera $v = \alpha(s)$, ssi $s$ vérifie la propriété de $\Phi$ correspondant au nœud $v$.

A une trace virtuelle $t$ construite sur un ensemble $S$ d'états virtuels et $R$ de types d'actions, on peut alors associer un graphe $g :< V, R, \Gamma >$ ainsi défini.

- $V$ est le sous ensemble des nœuds abstraits tels qu'un état $s$ au moins rencontré dans la trace $t$ vérifie la formule correspondante dans $\Phi$,
- à toute transition $(r, s, s')$ dans $t$, correspond une transition $(\alpha(s), \alpha(s'), r)$ dans $\Gamma$.

On définit ainsi une fonction d'abstraction $\alpha$ pour un ensemble de traces :

Pour une trace $t$ : $\alpha(t)$ est le graphe tel que défini ci-dessus
et pour un ensemble de trace $d$ : $\alpha(d) = \bigcup\{\alpha(t) | t \in d\}$

La fonction de concrétisation fait correspondre à un graphe $a$ de $\mathcal{A}$ l'ensemble de tracse ainsi défini :

$\gamma(a) = \{t | t \in D \wedge \alpha(t) \subseteq a\}$

$\mathcal{P}(D) \underset{\gamma}{\overset{\alpha}{\longleftrightarrow}} \mathcal{P}(G)$ est une insertion de Galois. En effet les fonctions $\alpha$ et $\gamma$ sont

monotones : $\forall\ d, d' \in D : d \subseteq d' \ \Rightarrow\ \alpha(d) \subseteq_{\mathcal{G}} \alpha(d')$ et
$\forall\ a, a' \in \mathcal{A} : a \subseteq_{\mathcal{G}} a' \ \Rightarrow\ \gamma(a) \subseteq \gamma(a')$,
extensives (4) $d \subseteq \gamma \circ \alpha(d)$,
contractantes (5) $\alpha \circ \gamma(a) \subseteq_{\mathcal{G}} a$ avec $\alpha \circ \gamma(a) = a$.

Il est donc possible de calculer des graphes "couvrants", c'est à dire des graphes $G$ qui satisfont $D \subseteq L(G)$. En effet, par construction on a bien $D \subseteq L(\alpha(D))$.

Si l'on dispose d'un opérateur abstrait $T_{\mathcal{A}}$, qui vérifie les conditions du théorème 2, il est alors possible de calculer un graphe G comme point fixe de cet opérateur $G = lfpT_{\mathcal{A}}$, lequel sera obtenu après un nombre fini d'itérations, et vérifiera $D \subseteq \gamma(G) \subseteq L(G)$.

En son absence, on devra se contenter d'observer des approximations croissantes : $\alpha(\bigcup_{i=0}^{n} T_D^i(\emptyset))$.

Nous n'avons pas décrit ici de méthode automatique ou automatisable pour construire un opérateur abstrait. C'est un travail en cours. On retiendra cependant que cette construction est nécessaire pour prouver la propriété recherchée.

Cette démarche s'apparente donc à une démarche de preuve. La méthode proposée sera en particulier toujours applicable si l'ensemble de traces étudié est fini et ne comporte que des traces finies.

Dans le cas où l'on ne dispose que d'approximations croissantes (l'ensemble des traces ou des états est infini), la démarche s'apparente à du test. L'apparition d'anomalie dans le graphe abstrait n'est qu'un symptôme d'incorrection, mais non une certitude. De telles observations peuvent néanmoins être extrêmement utiles en cours de mise au point du modèle.

Enfin on s'est limité ici à ne considérer que des graphes abstraits avec un petit nombre de nœuds, permettant un examen "à vue" d'anomalies de signatures ou de certaines propriétés. La méthode s'applique bien sûr à d'autres propriétés comme des propriétés d'atteignabilité ou de vivacité d'états par exemple. Par ailleurs la méthode présentée, basée sur l'interprétation abstraite, fait partie d'une panoplie de méthodes développées dans le cadre du "model checking" qui montre qu'il est possible de traiter ainsi des systèmes de traces avec un nombre d'états, et mêmes de nœuds abstraits, extrêmement grand [21,19].

## 7 Fidélité

L'idée de fidélité correspond au fait qu'il doit être possible de reconstituer la partie de la trace virtuelle intégrale qui a donné naissance à une trace effective partielle. Cela suppose en particulier qu'à partir d'une trace effective intégrale on puisse reconstituer la trace virtuelle intégrale qui l'a produite. D'où l'idée d'existence d'une SI telle que les shémas d'extraction et de reconstruction commutent. On ne considère ici que des traces intégrales contigues.

### 7.1 Fidélité d'une SO

On distinguera deux notions de fidélité selon que le shéma de traceur est connu avec une SO complète ou que la SO est incomplète. Dans le premier cas la fonction de transition est connue et la fidélié consiste à s'assurer que la trace extraite reproduit bien la même suite détats que celle produite par le modèle du traceur. Dans le second cas la fidélité ne peut porter que sur la partie des paramètres et attributs dont l'évolution est décrite par une fonction de transition. Dans le pire cas (il n'y a pas de schéma de traceur), on ne peut parler de fidélité qu'avec la connaissance de la seule sémantique dont on dispose, à savoir un ensemble de traces effectives. Il s'agit alors de s'assurer que les traces virtuelles correspondantes peuvent bien être reconstruites à partir des traces effectives qu'elles sont supposées avoir produites ; seules les fonctions d'extraction et de reconstruction interviennent alors, mais on ne pourra établir qu'une sorte de cohérence entre ces fonctions qui peut-être vue comme une forme de correction partielle de la trace effective.

Dans les deux cas on parlera *fidélité de la SO*, mais dans le premier cas, qui fait intervenir les deux schémas de la SO et le schéma de reconstruction, on aura une *fidélité forte* et dans le second cas, où seuls les schémas d'extraction et de

reconstruction interviennent, mais sur un ensemble connu de traces effectives finies, on n'aura qu'une *fidélité faible*[12].

**Definition 6 (Fidélité faible).**
*Etant donnés une SO* $< S, I_f, R_O, A, E, T, S_0 >$ *et un ensemble de traces effectives* $\mathcal{T}^w$*, la SO est* faiblement fidèle*, ou simplement* fidèle *pour* $\mathcal{T}^w$*, s'il existe une SI* $< A, R_O, S, I >$ *k-constructible telle que toute trace effective de* $\mathcal{T}^w$ *vérifie*

$I(T^w_{t,k}) = T^v_t$ *et* $E(T^v_t) = T^w_t$.

**Definition 7 (Fidélité forte).**
*Une SO complète* $< S, I_f, R_O, A, E, T, S_0 >$ *est* fortement fidèle*, ou simplement* fidèle *si elle est fidèle pour toutes les traces effectives produite par son schéma de traceur.*

*Plus précisemment, une SO complète est* fidèle*, s'il existe une SI* $< A, R_O, S, I >$ *k-constructible telle que toute trace virtuelle* $T^v_t$ *produite et toute trace effective extraite telles que* $E(T^v_t) = T^w_t$*, on a*

$I(T^w_{t,k}) = T^v_t$.

*Pour une SI donnée, on dira alors que la SO est fidèle pour la SI donnée, ou que la SI* est fidèle à *la SO.*

Si l'on fait abstraction du $k$ de la reconstructibilité, la notion de fidélité exprime que le produit de $I$ et $E$ commute est égal à la fonction identité, soit $I \; o \; E = E \; o \; I = id$[13].

Il résulte de la fidélité d'une trace, qu'à partir de toute trace effective intégrale contigue, il est possible de retrouver la trace virtuelle intégrale à partir de laquelle elle peut être produite. Il est donc possible de l'interpréter intégralement.

La fidélité faible correspond au même schéma de commutativité, mais appliqué au seul schéma de trace et à une famille de traces données.

### 7.2 Méthodes de vérification de fidélité

Une preuve de fidélité peut s'effectuer à partir de SO et d'une SI donnée à l'aide de la fonction de transition de la SO et des fonctions locales d'extraction et de reconstruction. On considère ici seulement les cas 0- et 1-reconstrtuctible.

[12] Si on a une trace effective finie, on vérifie qu'on peut en inférer une trace virtuelle qui peut la produire. Un parallèle est fait ici avec la correction de programme où l'on distingue la terminaison et la correction partielle qui n'a d'utilité en fait que si le programme termine, i.e. si on a une trace finie d'exécution.

[13] Ceci doit être pris plus comme une image que comme une propriété car les deux foncteurs n'opèrent pas sur les mêmes objets. Une formulation plus exacte serait : $E(T^v) = T^w$ et $I(T^w) = T^v$, donc $I(E(T^v)) = T^v$, et $E(I(T^w)) = T^w$. De plus, la fidélité ne peut s'exprimer ainsi que dans le cas 0-constructible ou pour des traces infinies où l'on dispose toujours de suffisamment d'événements de trace effective pour réaliser la reconstruction (voir cependant la remarque après la proposition 3).

**Proposition 1 (Preuve de fidélité forte (cas 0-constructible)).**

*Une SO complète* $< S, I_f, R_O, A, E, T, S_0 >$ *est fidèle s'il existe une SI :* $< A, R_O, S, I >$ 0*-constructible, telle que pour chaque type d'action* $r \in R_O$*, les propriétés suivantes sont satisfaites :*

$\forall\ s, s' \in S, \forall\ a \in A$*, tels que* $T(r, s) = s'$ *et* $E_l(r, s) = a$*, on a*
$\exists r_1 \in R_O$ *tel que* $I_l(s, a) = (r_1, s')$ *et* $r_1 = r$*.*

La figure 3 illustre la preuve de fidélité par commutativité des actions d'extraction et reconstruction.

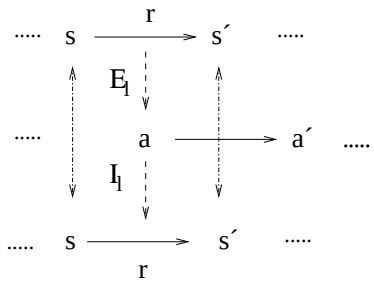


**Fig. 3.** Illustration de la fidélité forte (cas 0-constructible)

**Proposition 2 (Preuve de fidélité forte (cas 1-constructible)).**

*Une SO complète* $< S, I_f, R_O, A, E, T, S_0 >$ *est fidèle s'il existe une SI :* $< A, R_O, S, I >$ 1*-constructible, telle que pour chaque type d'action* $r \in R_O$*, les propriétés suivantes sont satisfaites :*

$\forall\ s, s' \in S, \forall\ a, a' \in A, \forall\ r', \in R_O$ *tels que* $T(r, s) = s'$*,* $E_l(r, s) = a$*, et* $E_l(r', s') = a'$ *on a*
$\exists r_1 \in R_O$ *tel que* $I_l(s, aa') = (r_1, s')$ *et* $r_1 = r$*.*

La figure 4 illustre la preuve de fidélité forte dans le cas 1-constructible.

Dans [35], on donne une preuve de fidélité pour une trace effective non intégrale 1-constructible. Ici on donnera méthode et exemple pour une trace intégrale, propriété que l'on utilisera ensuite pour montrer la fidélité de traces partielles (même si une méthode directe, comme dans [35], restera toujours possible).

Dans le cas de fidélité faible, on ne peut à proprement parler de preuve, vu que la notion ne s'applique qu'à un ensemble donné de traces effectives finies. Il s'agit donc d'une sorte de vérification de cohérence d'une famille de traces et de la SI.

**Proposition 3 (Vérification de fidélité faible).**

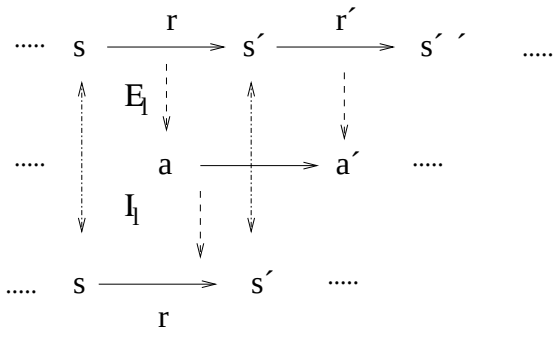


**Fig. 4.** Illustration de la fidélité forte (cas 1-constructible)

*Étant donnés un schéma de trace* $SO < S, R_O, A, E >$ *et un ensemble de traces effectives* $\mathcal{T}^w$*,* $SO$ *est* fidèle *pour* $\mathcal{T}^w$ *s'il existe* $SI : < A, R_O, S, I >$ $k$*-constructible, tel que pour chaque* $T_t^w \in \mathcal{T}^w$

$\forall\ i, 0 < i \leq t - k,\ \forall\ a_{i,k} \in T_t^w$ *on a :*
$I_l(s_{i-1}, a_{i,k}) = (r_i, s_i)$ *et*
$E_l(r, s_{i-1}) = a_i$.

La figure 5 illustre la preuve de fidélité faible.

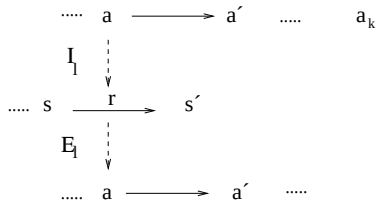


**Fig. 5.** Illustration de la fidélité faible

Le concept de fidélité est essentiel. C'est une sorte d'invariant que l'on doit retrouver dans toutes les manipulations de traces.

**Remarque**. Il est intéressant d'analyser le principe de la vérification de fidélité faible afin de mieux comprendre ce qu'il apporte. On ne dispose pas de schéma de traceur, et l'essentiel de la sémantique se retrouve dans le schéma de reconstruction. La vérification consiste à analyser une trace effective donnée à partir de l'état virtuel initial $s_0$ donné avec la trace. La fonction de reconstruction peut alors reconstruire l'état $s_i, 0 < i \leq t - k$ ($i = 1$ à la première étape) et identifier l'action $r_i$ utilisée. On peut alors vérifier que le premier événement

de la trace effective utilisé, $a_i$ (premier élément de $a_{i,k}$), est bien extrait de l'état courant précédent $s_{i-1}$ par l'action $r_i$. Comme ceci sera vérifié pour les $k$ événements suivants, la portion de trace utilisée $a_{i,k}$ est bien extraite de la trace virtuelle. Ce raisonnement ne vaut plus pour les $k$ derniers événements de la trace $T_t^w$, ce qui suppose que le facteur $k$ de constructabilité décroisse au fur et à mesure que l'on se rapproche de la fin de la trace, l'état final de la trace étant 0-constructible.

On observe que ce type de vérification peut être interprété de deux manières : une certaine assurance de la cohésion des fonctions d'extraction et de reconstruction, ou une certaine corretion de la trace fournie. Ceci peut être intéressant pour des analyses post mortem de traces effectives dont on ne connaît pas l'origine.

*Property 1.* Si une SO complète est fortement fidèle, alors la vérification qu'une trace effective donnée est fidèle peut se faire de deux manières équivalentes :

- montrer que chaque évément de trace est le résultat d'une transition de la SO effectuée à partir de l'état précédent,
- vérifier étape par étape que les événements de trace vérifient les conditions de la proposition 3.

## 8 Transformations de traces

Une idée majeure liée à la notion d'intégralité de la trace est que la fabrication d'un traceur est toujours incrémentale et qu'il n'est pas nécessaire de modifier ce qui a déjà été fait car, en général, il suffit d'ajouter de nouveaux paramètres à ceux déjà existants. Certes le volume de la trace va augmenter (nombre d'événements et taille des états), mais il est alors possible de ne considérer qu'une partie de la trace, celle dont on a besoin au moment d'effectuer un travail d'analyse précis. Une telle idée, simple en apparence, nécessite de préciser le genre d'opération que l'on peut faire sur les traces et comment leurs sémantiques évoluent.

La nature même de la trace intégrale, sa capacité à contenir toutes les observables, repose sur deux opérations en apparence contradictoires : l'enrichissement par ajout de paramètres et types d'actions possibles, ce qui lui permet de croître, et la restriction à un sous-ensemble de paramètres de la trace virtuelle, qui constitue une sélection, et permet de n'en voir ou n'utiliser qu'une partie, c'est à dire de n'en considérer qu'une sous-trace.

Dans les deux transformations mentionnées ci-dessus, une question importante est de comprendre comment les sémantiques (SO et SI) évoluent.

En génie logiciel, mais dans d'autres domaines aussi comme les systèmes de capteurs, on est amené à combiner plusieurs traces pour n'en constituer qu'une seule. Une telle opération sera nommée fusion de traces.

Une autre transformation consiste, après avoir enrichi une trace virtuelle avec de nouveaux paramètres construits à partir de paramètres existants, à ne retenir que ces nouveaux paramètres. On obtient ainsi une nouvelle trace virtuelle

à partir de laquelle les paramètres d'origine ne peuvent plus être reconstruits. On peut considérer que la nouvelle trace est une abstraction de la première. Cette abstraction peut être vue comme une combinaison d'enrichissements particuliers et de sélections. Les particularités de l'enrichissement (détaillées plus bas) nous amèneront à considérer cette combinaison comme une transformation remarquable que nous appelons abstraction.

On s'intéressera enfin à des traces générales, c'est à dire susceptibles de refléter le comportement, non plus d'un seul processus, mais d'une famille de processus. Chaque processus est indépendant et observé séparément, mais on a pu identifier des observables communes dont l'évolution peut se décrire par une trace dite "générique". La généricité répond au besoin d'avoir une trace intégrale unique pour plusieurs processus supposés faire des choses suffisamment similaires pour que leurs observables (les objets et actions) soient les mêmes. Il y a à la fois un aspect enrichissement, car tout ce passe comme si on fusionnait différents processus en un seul, mais aussi un aspect d'oubli, car on ne retient alors d'"intéressant" que des observables communes à tous les processus de la famille.

### 8.1 Enrichissement

L' enrichissement est l'opération qui consiste à ajouter des paramètres et actions à la trace virtuelle et donc des attributs à la trace effective.

La trace virtuelle décrit l'évolution d'une collection d'entités diverses (objets, relations, événements, ...). Elle constitue une "histoire" qui peut résulter de l'interaction d'un grand nombre de processus, et qui, bien souvent, est la seule sémantique dont on dispose. Il se peut en effet qu'on ne dipose d'aucun modèle du processus ni de son traceur. Une chose cependant est sûre, plus l'ensemble des mécanismes sous-jacents à décrire est complexe, plus la trace va être riche tant en nombre d'événements qu'en nombre de paramètres.

Les sources d'enrichissement proviennent essentiellement du raffinement d'un composant (augmentation des capteurs d'événements au sein d'un processus) : le nombre de paramètres (donc le nombre d'attributs) va augmenter, mais aussi le nombre d'événements de trace puisque des changements d'état se produiront plus fréquemment. La raison principale pour augmenter le nombre des "capteurs" est la variété des utilisations que l'on peut vouloir faire de la trace ou le besoin d'une plus grande précision. Dans tous les cas le nombre d'observables et d'états croît.

Un raffinement peut également provenir de l'ajout de paramètres (resp. attributs) redondants, soit qu'ils soient calculables à partir d'autres paramètres (resp. attributs), soit qu'ils soient calculables à partir de l'évolution d'autres paramètres (resp. attributs).

### 8.2 Sélection

La sélection ou "oubli sélectif" ou "filtrage" consiste à considérer une sous-trace définie à partir de critères de sélection. La sélection s'opère en pratique sur

la trace effective. Deux types de sous-traces doivent être considérés selon que le langage de signature est modifié ou non.

Dans le premier cas une sous-trace effective consiste à sélectionner seulement un sous-ensembles d'attributs, mais les chronos originaux sont gardés et la trace garde la même signature. Ni la SO ni la SI ne sont modifiées ; il faut seulement veiller à ce que la sous-trace conserve sa propriété de fidélité. Cette approche est celle utilisée dans [61].

Dans le second cas on peut en plus ne sélectionner qu'un sous-ensembles d'événements. La sous-trace devient une trace segmentée. On peut la considérer comme une trace contigue, mais cela ne peut se faire qu'en modifiant la SO. Pour conserver la propriété de fidélité, il faut alors, en plus de veiller à ne pas "oublier" des attributs intervenant dans la reconstruction des paramètres de la sous-trace, veiller à ce que les actions des événements oubliés n'influent pas sur les paramêtres d'intérêt (conservation de la fidélité).

### 8.3 Fusion

Une raison importante d'enrichissement vient de la combinaison de plusieurs composants logiciels dont chacun est déjà muni de son traceur et donc de sa propre trace virtuelle intégrale (intégration de plusieurs processus en un seul).

Il s'agit du regroupement de composants (groupement d'un ensemble de processus interagissants et ayant chacun leurs traces). La trace effective de l'ensemble peut comporter tous les attributs des traces effectives de chaque processus augmentés d'attributs portant sur les interactions entre processus. Comme, en principe, de l'extérieur on doit pouvoir encore analyser n'importe lequel des processus interne au groupe, chaque trace virtuelle (resp. effective) peut se retrouver dans la trace virtuelle (resp. effective) du groupe.

Ici on étudie comment les sémantiques de la trace du système constitué de l'ensemble de ses composants peuvent ête construites à partir de l'ensemble des sémantiques des traces de chaque composant. On étudie également comment les traceurs et leurs pilotes sont construits dans ce type d'architecture.

*Example 6.* Cette section sera illustrée par un exemple assez riche et original consistant à combiner les SO de Prolog telle que décrite ici et la SO du solveur GNU-Prolog telle que décrite dans [59].

○

### 8.4 Abstraction

On appelle *abstraction* l'opération qui consiste à enrichir une trace avec des méta-paramètres puis sélectionner la sous-traces fidéle qui ne contient que les méta-paramètres. Une trace peut être le résultat d'une suite d'abstractions constituant ainsi une série de couches de traces. Avant de passer à la couche supérieure, une trace peut avoir été elle même enrichie.

### 8.5 Généricité

La généricité se caractérise par le fait que l'on peut se donner une SO indépendemment de tout processus. La SO est alors une sémantique formelle de traceur (sans pilote) qui peut être qualifié alors de traceur générique. C'est l'approche qui a été utilisée par exemple dans le projet OADymPPaC [29] ; la SO était donnée comme une sémantique de traceur susceptible de s'appliquer à toute une famille de processus ayant les mêmes observables. La question pratique qui se pose dans cette approche est se savoir si le traceur peut être implanté dans chaque processus de la famille que l'on veut pouvoir observer. En d'autres termes, quelle relation il y-a-t-il entre les sémantiques SO et SI du traceur générique et celles de chaque processus.

Une approche possible consiste à considérer que le schéma de traceur générique est une abstraction d'une sémantique du processus qui ne retient que certains paramètres de l'état du processus, ainsi que certaines étapes de son évolution. Par ailleurs, chaque processus de la famille observée a ses propres SO et SI. Il s'agit alors de mettre en correspondance les sémantiques de chaque processus et celles de la trace générique. Il doit y avoir une correspondance biunivoque entre une partie significative de la trace virtuelle de chaque processus de la famille et une partie de la trace générique virtuelle.

## 9 Enrichissement

On appelle *enrichissement* l'opération qui consiste à ajouter des paramètres au domaine détat virtuel, sans modification du processus supposé unique qui est à l'origine de la trace observée.

On distingue deux types d'enrichissement : la redondance et le raffinement.

Dans tous les cas la trace virtuelle ou effective intégrale comportera plus de détails sous forme de paramètres ou d'attributs nouveaux.

Une trace virtuelle (resp. effective) redondante contient plus de paramètres (resp. attributs) que nécessaires. Les nouveaux paramètres ou attributs n'apportent pas d'information supplémentaire. Ils n'ont pour but que de coder une information avant son extraction ou sa diffusion afin d'éviter d'avoir à la reconstruire ultérieurement. Il peut être en effet intéressant d'introduire des paramètres redondants si le coût de construction interne est inférieur au coût d'extraction. Il peut être intéressant d'introduire des attributs redondants si le coût de reconstruction des observables correspondantes à partir des attributs initiaux est supérieur au coût d'extraction plus celui de la transmission des attributs redondants. Les paramètres ou attributs redondants constituent un enrichissement qui n'a pas d'impacte sur le nombre d'événements de la trace virtuelle. La redondance des attributs n'affecte que la trace effective, la trace virtuelle restant inchangée. De manière générale, un attribut redondant est un attribut qui peut être supprimé sans nuire à la fidèlité de la SI pour une SO donnée.

Le raffinement consiste à introduire de nouveaux paramètres non redondants, des facteurs d'influence et de nouveaux types d'actions. L'état virtuel courant

est modifié ainsi que la sémantique observationnelle. Le nombre d'événements de trace peut augmenter. Ce type d'enrichissement est caractéristique de l'idée de trace intégrale : on peut ajouter autant d'informations et de détails que l'on souhaite dans la trace qui croît en fonction des ajouts successifs suivant les besoins d'observation ou de communication. Il induit la méthodologie sous-jacente de développement des traceurs qui consiste à ne pas refaire ce qui a déjà été fait, mais à l'enrichir.

## 10 Sélection

On appelle *sélection* l'opération qui consiste à sélectionner une sous-traces (oubli sélectif) tout en préservant la fidélité.

*Example 7.* Cette section sera illustrée par l'exemple tel que décrit dans [34,33], mais avec une trace effective intégrale (et non celle de Byrd), en prenant différents sous-ensembles $Q$ (voir poster [32]).

○

## 11 Fusion

On appelle *fusion* l'opération qui consiste à regrouper plusieurs traces de manière à obtenir une nouvelle trace unique fidèle.

Elle correspond au fait de réunir plusieurs processus en un seul, chaque processus étant déjà muni de son traceur. Les traces (virtuelle et effective) résultantes comportent plus d'événements et de paramètres ou attributs que la simple "somme" (qui résulterait d'une combinaison des événements de chacune des traces), ainsi que de la simple "composition" des états. Il faut imaginer qu'il y a alors un "super" traceur qui intègre en quelques sortes les traces produites par les traceurs de chaque processus. Une sous-trace intéressante peut-être sa restriction à un seul des processus observés. Une bonne propriété de la sous-trace virtuelle dans ce cas est qu'elle corresponde également à une sous-trace du processus observé. On verra alors que la seule condition est que, par exemple pour la trace virtuelle, la sous-trace correspondant à la restriction aux paramètres propres à un des processus avant regroupement produit la trace originale de ce processus pris isolément. Du point de vue méthodologique, le nouveau traceur intègre les traceurs de chaque processus en y ajoutant un module avec des action spécifiques (ordonnancement des événements, paramètres et attributs particuliers ...).

Raffinement et composition sont en pratique de même nature. Un raffinement peut consister à mettre en évidence différents aspects d'un même processus ou résulter de l'ajout de composants au système de processus que l'on veut observer.

*Example 8 (***Illustration sur les signatures).**

Ensemble de deux composants logiciel avec chacun leur trace dont les signatures appartiennent aux langages $L_1 = a^*$ et $L_2 = b^*$ définis sur les vocabulaires

respectivement $R_1 = \{a\}$ et $R_2 = \{b\}$ . La signature de la trace de l'ensemble est définie sur le langage $L_{12} = (a|b)^*$ de vocabulaire $V_{12} = \{a, b\}$

$s_1$ :*aaaaaaaa*....

$s_2$ :*bbbbbbbb*....

Trace possible pour le processus $s_{12}$ : *abbbaaabbbabababaabba*...

Il est clair que la SO des traces fusionnées ici est non déterministe. ○

## 12 Abstraction

On appelle *abstraction* l'opération qui consiste à enrichir une trace avec des méta-paramètres puis sélectionner la sous-traces fidéle qui ne contient que les méta-paramètres. Une trace peut être le résultat d'une suite d'abstractions constituant ainsi une série de couches de traces. Avant de passer à la couche supérieure, une trace peut avoir été elle même enrichie.

## 13 Généricité

La généricité n'est pas une opération mais une proprété d'une trace qui est définie par une SO sans référence à un processus particulier. La question qui se pose alors concerne les relations possibles entre cette trace et la famille de processus qu'elle est sensée refléter. Cette problématique est en quelques sortes orthogonale aux opérations sur les traces considérés précédemment.

*Example 9.* Cette section sera illustrée par l'exemple de l'instanciation de la trace générique GenTra4CP dans GNU-Prolog [59]. ○

## 14 Domaines fondateurs et applicatifs

Enrichissement, sélection, fusion et abstraction sont les quatre transformations fondamentales que l'on puisse opérer sur et avec des traces (virtuelles et effectives). Elles constituent une esquisse d'algèbre de traces.

Dans cette section on cherche à étudier les liens potentiels qu'il peut y avoir entre l'approche des traces développée ici et différents champs de recherche. Ceux-ci constituent des champs d'application pour la théorie développée ici. Ces domaines touchent au génie logiciel, le génie des connaissances, les sciences cognitives ou la philisophie, qui tous ont en commun, parfois, l'utilisation du paradigme de trace.

Le fait même de mettre en relation ce domaine et différents domaines de recherche offre également un point de vue sur les relations croisées que peuvent entretenir ces différents domaines entre eux.

Neuf domaines plus ou moins indépendants sont examinés, en tous cas le point de vue sous lequel ils sont examinés sont différents. Pour chacun, on résume en fin de section l'apport potentiel espéré que notre approche des traces peut avoir sur le domaine ainsi que le retour d'expérience que l'on peut envisager du domaine ou d'une utilisation formelle de traces dans celui-ci.

## 14.1 Construction de traceurs pour l'analyse dynamique de programmes

Le premier domaine d'application examiné ici est la construction de traceurs pour l'analyse dynamique de programmes. Ce que l'on recherche ici est de pouvoir observer le comportement d'un processus unique, lui-même éventuellement composé d'un assemblage de plusieurs processus.

Historiquement, la notion de trace est liée en génie logiciel à la trace de programme. Classiquement le débogage de programmes repose sur deux piliers : l'analyse statique et l'analyse dynamique. La première permet de détecter certaines bogues avant toute exécution mais ne suffit pas en général ; la seconde repose essentiellement sur l'analyse de traces d'exécutions. Elle inclut les tests d'exécution. Traditionnellement, l'analyse dynamique de comportement se fait par une modification du compilateur qui permet, sur option, de faire fonctionner le programme dans un mode "debogage", ou par instrumentation directe du code source qui calcule et publie des informations particulières à la demande. Ces deux approches consistent à réaliser un traceur qui a également fonction d'analyseur. Ainsi, lorsque l'on veut effectuer une évaluation de performance, par exemple réaliser des statistiques sur des points de passage ou sur les temps d'exécution de certaines portions de programme (faire du "monitoring"), on introduit dans le code du programme des capteurs d'informations d'une part et des calculs d'informations d'autre part. De cette manière les fonctions de trace et d'analyse sont entièrement contenue dans l'instrumentation du programme (grâce à des ajouts de code spécifique au code original du programme).

L'idée de séparer la partie "capteurs" de la partie "analyse" s'est imposée progressivement au fur et à mesure que d'une part les langages devenaient plus complexes et que, surtout, les besoins d'analyse devenaient plus élaborés. En 1993, M. Ducassé et J. Noyer observent que, dans le cas de Prolog, un langage de règles logiques dont le traceur, comme les besoins d'analyse de performance, sont assez complexes, il y avait un grand intérêt à rendre indépendantes les tâches de production de trace et d'analyse [39]. Si donc ces tâches sont indépendantes, elles peuvent être accomplies par des personnes différentes, avec des métiers différents. La seule contrainte est alors de se mettre d'accord sur l'information qui doit être transmise par le traceur et garantir qu'elle ait la bonne propriété de pouvoir récupérer toute l'information initialement récoltée. Cette approche a été à la base du projet OADymPPaC [29] et a abouti à la notion de trace générique, une trace valable pour un domaine de calcul particulier et qui contient toutes les informations utiles pour une famille d'analyses potentielles, ainsi qu'à poser le problème d'avoir une sémantique propre à la trace, la sémantique observationnelle. La SO peut être vue comme une sémantique de traceur telle que toutes les instances implantées dans chaque processus d'une famille produisent la même trace, ou au moins une partie de celle-ci.

Cette approche soulève toute une série de questions que nous examinons maintenant. Pour certaines d'entre elles, et pour les solveurs de contraintes, le projet OADymPPaC a proposé des réponses.

**Trace intégrale et pilote efficace** La trace utile à toute une famille d'analyses potentielles peut devenir gigantesque car son niveau de raffinement peut croître en fonction des besoins d'analyse. La réponse proposée correspond d'une part à la notion de trace effective *intégrale*, et d'autre part à l'ajout au traceur d'un pilote chargé d'opérer la sélection d'une sous-trace contenant les seuls éléments utiles à une analyse particulière. La première question porte sur l'efficacité pratique d'une telle approche.

Dans [61] M. Ducassé et L. Langevine montrent qu'un traceur standard (pour le langage logique avec contraintes GNU-Prolog) susceptible de produire une très grande trace potentielle, mais muni d'un pilote, peut produire une trace efficace et utile pour de nombreux outils d'analyse dynamique qui utilisent la trace soit au vol soit a posteriori. La notion de trace intégrale permet de prendre en compte dans les traces de très nombreux aspects permettant la réalisation d'analyses diverses. Néanmoins de telles traces peuvent être extrêmement grandes, de l'ordre de plusieurs gigabytes pour quelques secondes d'exécution seulement. Il ne peut donc être question de les générer intégralement, car en fait, pour un type d'analyse donnée ou pour un type d'analyseur, seule une partie des informations fournies dans la trace est utile. C'est la fonction du pilote de filtrer la trace utile en fonction des besoins. Ces questions sont également discutées dans [30] concernant l'extraction d'une trace intégrale et l'analyse des problèmes d'efficacité de cette approche.

**Langage d'interrogation de trace** Cette action de sélection impose de disposer d'un langage d'interrogation de trace. Dans [38] les mêmes auteurs proposent un ensemble de primitives d'interrogation de trace qui permet de spécifier une sous-trace. Ces auteurs se limitent cependant aux cas où la signature est préservé et où le filtrage ne s'opère que sur un événement de trace à la fois. Des expérimentations ont été conduites dans [62] montrant que en restreignant l'interrogation à un sous-ensemble d'attributs, on gardait une bonne efficacité même dans le cas où plusieurs analyseurs travaillent simultanément. Le fait de considérer ici des sous-traces ne préservant pas la signature est une généralisation.

**Interactions avec le traceur** Cette approche impose une architecture particulière entre les analyseurs et le traceur. Les analyseurs deviennent des processus clients du traceur (inclus dans le processus observé) qui lui-même devient serveur ; ceux-ci doivent pouvoir adresser des ordres au traceur (en fait par l'intermédiaire du pilote), en particulier permettre un arrêt puis reprise de celui-ci. Ceci impose de pouvoir communiquer un état (éventuellement) effectif intégral, sinon à tout moment, au moins à un nombre suffisant de points de reprise. Le fait de garder la possibilité d'accès presque à tout moment à un état intégral, peut ralentir considérablement un programme instrumenté pour le produire. Cela reste inhérent au fait qu'aucune limite n'est imposée à la taille d'une trace intégrale et reste pour le moment une limite potentielle à l'utilisation de cette approche en génie logiciel.

Par ailleurs, la définition de bonnes primitives pour communiquer avec le traceur selon différents modes (synchrones ou asynchrones) reste, dans ce domaine d'application, une question ouverte.

**Optimisation de la trace** La fidélité garantit également une forme de conservation de l'information observable, mais il en résulte un accroissement de la taille de la trace effective intégrale. Or ce qui coûte le plus cher est la communication entre le programme observé et un analyseur. En effet celle-ci passe par une écriture dans un buffer intermédiaire. Il peut donc être primordial de réduire la taille de la trace effective. Ceci revient à se poser la question de la répartition de la charge des calculs liés aux observables : en partie lors de l'extraction à la source (au niveau du serveur) ou en partie lors de la reconstruction (au niveau du client). En effet le fait de ne sélectionner qu'une sous-trace ne permet pas toujours de réduire suffisamment la taille de la trace effective. C'est le cas notamment lorsque plusieurs analyses sont conduites simultanément et nécessitent d'extraire une trace finalement très volumineuse pour satisfaire l'ensemble des besoins. Il peut alors être utile de connaître l'usage qui va être fait des observables communiquées dans la trace (après reconstruction) et les nouveaux objets (éventuellement plus abstraits) qui vont être construits à partir de la trace virtuelle reconstruite. Ces objets peuvent être considérés aussi comme des paramètres redondants (puisque construits à partir de paramètres ou attributs existants et, éventuellement en prenant en compte plusieurs événements de trace) et sont de ce fait appelés *méta-paramètres*. Il peut alors être plus intéressant de construire à la source des méta-paramètres si on peut en extraire des attributs moins volumineux et de se fait parvenir à réduire la taille de la trace effective en communiquant directement les informations qui permettront de transmettre de manière plus efficace l'objet plus abstrait recherché. Les calculs tant à la source (lors de l'extraction des attributs) qu'à l'arrivée (lors de la recosntruction des observables) peuvent être plus coûteux, mais l'économie de temps réalisée lors de la transmission peut largement compenser ces pertes. Dans ce cas, la solution peut donc consister à enrichir la trace effective avec des méta-paramètres utiles aux analyses pressenties, les calculer à la source (ce qui revient à transférer une partie de la charge des analyseurs vers le traceur[14]), et à ne communiquer que ceux-ci dans le trace effective.

La recherche d'un bon compromis entre les attibuts qui doivent être calculés à la source et les observables reconstruites à l'arrivée est une question d'ordre pratique. La notion d'abstraction, vue comme une composition d'enrichissement avec des méta-paramètres, suivi de sélection d'une sous-trace a pour but de donner une approche rigoureuse à ce problème.

**Traceur et composants logiciels** Une partie des travaux développés ici concerne la mise en commun de plusieurs composants logiciels, chacun étant déjà muni d'un traceur avec son propre pilote, opérant ainsi une "fusion" de traces. L'opération de fusion, effectuable sous certaines conditions concernant

[14] Ceci revient en fait à enrichir la trace virtuelle intégrale avec de nouveaux paramètres, mais de telle manière que la sous-trace effective le plus souvent utilisée soit réduite

les traces propres à chaque processus et leurs signatures, offre une approche formelle éclairante. Dans ce cadre, on peut tenter d'élaborer des réponses aux questions suivantes.

- (filtrage réparti) Comment une sous-trace de la trace intégrale de l'ensemble se décompose-t-elle en sous-traces des traces intégrales des composants ?
- (interactions réparties) Comment les interactions entre les analyseurs et les pilotes de chaque composant se répartissent-elles entre les composants ?

L'approche des traces proposée ici conduit dans le domaine des traceurs de programmes à une certaine méthodologie de construction et manutention des traceurs. Les questions que l'on doit se poser alors concernent l'ensemble des paramètres et attributs à mettre dans une trace intégrale et les formes de dialogue possibles entre le traceur et les analyseurs.

**Des traces vers la construction de traceurs** La notion de trace effective correspond aux traces usuellement produites par les traceurs de programme. L'approche présentée ici vise à répondre en premier lieu aux questions soulevées par la réalisation de traceurs en vue de permettre l'élaboration d'analyseurs d'exécutions de programmes. L'algèbre de traces, les sémantiques proposés et les propriétés propres aux traces donnent une cadre théorique à la réalisation de traceurs et aux systèmes susceptibles d'aider à les manipuler et à les construire. C'est évidemment un domaine privilégié d'application de l'approche théorique d'une algèbre des traces.

**De la construction de traceurs vers les traces** L'expérience acquise dans la réalisation de traceurs et d'environnements de programmation utilisant comme entrée des traces produites indépendemment, a été la source de cette théorisation. On peut s'attendre à des évolutions de celle-ci en fonction des problèmes rencontrés dans le cadre de travaux plus avancés concernant le débogage de programmes et les environnements de programmation.

### 14.2 Modélisation et Abstraction

On veut s'intéresser ici aux relations qu'il peut y avoir entre la méthode d'élaboration d'une trace avec ses sémantiques, et les théories de l'abstraction, et plus particulièrement l'interprétation abstraite. Il ne s'agit pas ici à proprement parler d'un champ d'application de l'algèbre des traces (ce serait plutôt l'inverse), mais d'éxaminer les relations que cette approche des traces entretient avec un champ de théorisation possible.

La notion de fidélité montre que, dans la mesure où cette propriété est respectée, il n'y a pas d'abstraction entre la trace virtuelle et la trace effective ; c'est à dire que la trace effective n'est pas une abstraction de la trace virtuelle, pas plus que la fonction d'extraction[15] n'est une fonction d'abstraction. On peut donc traiter indifféremment de la trace virtuelle ou de la trace effective, leurs

[15] Dans la mesure où cette fonction est définie à partir de la trace virtuelle.

sémantiques étant en quelque sortes interchangeables. Chacune peut comprendre des méta-paramètres (ou méta-attributs) qui peuvent être vus comme des abstractions réalisées à partir d'autres paramètres (ou attributs), mais il ne s'agit que de redondance. On peut donc se limiter à considérer les trace virtuelles munies d'une sémantique observationnelle. Dans ce contexte, les questions qui se posent concernent les rapports de cette SO avec le processus dont elle rend compte, c'est à dire les différents niveaux d'observation du ou des processus produisant ses traces.

En supposant qu'il existe une SO complète pour les traces produites par le processus observé, les différentes questions se réduisent à quatre questions concernant différentes étapes dans l'élaboration de traces : la construction de modèles (SO) susceptible de rendre compte de traces, les rapports entre une SO et la sémantique du processus observé, en particulier quand celle-ci est formalisée, les différents niveaux d'observations (traces plus ou moins abstraites), c'est à dire les rapports entre des SO plus ou moins abstraites, et enfin les propriétés des sémantiques.

Ceci est illustré sur la figure 6.

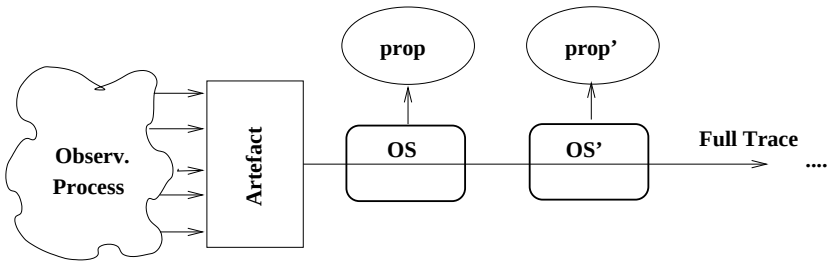


**Fig. 6.** Observation-abstraction d'un processus

Cette figure montre qu'il y a en amont du ou des processus observés un artéfact qui produit une trace intégrale. Ce qui est illustré ici c'est à la fois la manière d'aborder l'observation d'un phénomène et le concept de trace intégrale. Le processus (naturel ou artificiel) observé ne peut l'être que par l'intermédaire, soit d'appareils de mesure, s'il s'agit d'un processus naturel, soit d'une description formelle, donc dans un langage formel en quelques sortes mathématisé, par exemple s'il s'agit d'un programme exprimé dans un langage de programmation. Dans ce dernier cas, l'artéfact (qui se confond alors avec le processus) correspond à un programme muni de son traceur ou à une modélisation formelle du ou des processus observés. Par ailleurs, alors que l'artéfact peut être muni de nombreux capteurs, il ne produit en sortie qu'une seule trace, la trace intégrale qui contient potentiellement tout ce que l'on est susceptible de reconnaître et de discrétiser sous forme de trace.

La figure 6 illustre également les différentes SO qui peuvent être produites ainsi que leurs propriétés. La première peut être vue comme une formalisation minimale de l'artéfact, c'est à  dire une une sémantique qui rend compte de la production de la trace intégrale. Les SO suivantes correspondent alors à des abstractions possibles de la SO originale. Pour ces autres SO, il ne peut s'agir cependant que de sous-traces qui correspondent à des changements de niveau de description, et donc de sémantiques plus ou moins précises.

L'étape correspondant à ce que nous avons appelé "construction de modèle" correspond à la fois à la réalisation de l'artéfact (appareil de mesures ou collecteur d'information) et à sa description formelle éventuelle. Dans la recherche de modèle, il faut considérer deux activités complémentaires : la découverte et/ou la définition des observables, ce que nous appelons ici "paramètres", et la construction formelle d'une description de leur evolution. Une approche formelle de la modélisation consiste donc à exhiber un LTS correspondant au schéma de traceur de la SO (ensemble d'états et fonction de transition)[16]. La recherche des observables relève d'une activité d'analyse dans laquelle des techniques automatiques de type fouille de données peuvent avoir leur place. Ainsi dans le domaine de l'analyse de dispositifs d'apprentissage [18], le premier travail consiste à rechercher des observables pertinentes. Ces paramètres, la construction de modèle de leur évolution, ou même la possibilité d'en construire un, vont dépendre du champ d'étude concerné (ici l'apprentissage d'un domaine de connaissances). Chaque activité complémentaire (découverte des observables et d'un modèle d'évolution), dans la mesure où le point de départ repose sur des traces empiriques, relève de la recherche des sémantiques respectives SI et SO fidèles pour de telles traces.

L'interprétation abstraite est une théorie de l'abstraction qui a été introduite en génie logiciel pour l'analyse de programmes [25]. Elle permet d'organiser de multiples sémantiques d'un langage de programmation donné en une hiérarchie de sémantiques correspondant à des niveaux de détails différents, ainsi que pour définir des systèmes de types pour les langages de programmation. En particulier les relations entre des sémantiques de niveaux d'abstraction imbriqués peuvent être décrites formellement par des connections de Galois.

Dans [22] les auteurs s'intéressent à différentes observables de programmes logiques, en particulier celles concernant l'arbre de resolution (SLD-tree), et ce à différents niveaux d'abstration. Ils montrent comment différentes sémantiques de plus en plus abstraites (c'est à dire qui "oublient" de plus en plus de "détails" opérationnels) peuvent être reliées par des connections de Galois. Par exemple la description la plus raffinée rend compte de l'évolution de l'arbre SLD alors que les approximations successives des solutions exactes (prédications correspondant aux nœuds succès) peuvent être décrites simplement dans le cadre de la s-sémantique, approximation des solutions. Dans le domaine des langages de programmation, le processus observé (un programme en cours d'exécution) dispose

[16] Cette approche basée sur des automates est assez générale, mais d'autres formes de formalisation sont évidemment possibles comme on le verra plus bas.

a priori d'une description formelle complète[17]. Les SO correspondant à des niveaux d'observation différents, donc à des traces plus ou moins rafinées, sont des abstractions de la sémantique formelle du processus observé. Elles reproduisent un échantillonnage du comportement du processus observé.

Dans [53], différents modèles de traces Prolog semblables à ceux présentés en exemple dans cette étude, sont définis à partir d'une sémantique dénotationnelle de Prolog instrumentée avec production d'événements de trace (la fonction d'extraction est codée dans la sémantique). On peut voir cette sémantique formelle comme un schéma de traceur, on peut aussi se demander si un modèle plus abstrait ne serait pas suffisant. Si la sémantique dénotationnelle donne une schéma explicatif complet (c'est un interprète complet qui est en fait décrit ici), elle contient de nombreux détails inutiles si l'on veut se contenter d'"expliquer" l'enchaînement des événements de trace et leur extraction[18]. Cet article donne un bon exemple de ce que peut être un modèle de l'artéfact.

Une démarche semblable est utilisée dans [55] pour spécifier un traceur CHR(FD). Des éléments de codage formel nécessaires à l'extraction d'une trace sont introduits dans une sémantique formelle de CHR(FD). Dans ce travail, la méthode consiste à raffiner une sémantique opérationnelle de CHR(FD) de telle manière que la sémantique déclarative soit préservée, et jusqu'à ce que tous les éléments recherchés (toutes les observables) puissent y être exprimés. Ceux-ci sont spécifiés par le domaine des états virtuels de la trace que l'on veut produire. On construit de cette manière une abstraction de la sémantique opérationnelle[19], et donc un modèle de l'artéfact.

Dans [42], les auteurs utilisent les connexions de Galois pour relier différentes sémantiques et décrire des propriétés abstraites de systèmes biologiques. Dans cette approche la SO est décrite par une "machine abstraite biochimique" formalisée par un système hybride décrit par des règles agissant sur des paramètres biologiques, physiques et chimiques. Dans cette approche les mécanismes de la cellule vivante que l'on souhaite modéliser (i.e. le processus observé) peuvent être décrits de manière pseudo-formelle par des diagrammes de Kohn. Le point de départ est un modèle formel d'une certaine réalité, et sa validation s'obtient en vérifiant l'adéquation de son comportement avec des données expérimentales en comparant des traces obtenues avec le système formel avec des traces expérimentales,

[17] Ceci peut toujours être contesté puisque le programme est exécuté sur une machine physique pour laquelle il a été compilé, et que pour que son comportement corresponde à ce qui est décrit dans sa définition formelle il faut que le compilateur ait été certifié "correct" (i.e. qu'il préserve la sémantique originale) et que la machine se comporte également comme prévu, i.e. qu'elle préserve la sémantique de son propre code. Ceci suppose qu'elle respecte absolument les lois physiques auxquelles elle se réfère.

[18] Dans le cas du papier cité, la sémantique dénotationnelle est réduite au minimum nécessaire pour rendre compte de la trace émise. On n'est donc pas très éloigné en fait d'une SO complète et déterministe décrite par une sémantique avec continuations.

[19] Ce travail n'établit par cependant la preuve formelle que l'abstraction est une connexion de Galois.

ou certaines de leurs propriétés communes. Cette approche s'apparente aux méthodes utilisées en vérification de modèles ("model checking" [20]) où l'on cherche, en particulier, à vérifier des propriétés de sûreté sur des automates d'état (LTS dont les états peuvent être en nombre infini).

Dans [6], les auteurs s'intéressent aux systèmes hybrides utilisés dans de très nombreux domaines d'applications comme le contrôle automatique, les processus chimiques ou la robotique. Les systèmes hybrides sont constitués de systèmes dynamiques continus et discrets. Cet article établit que, sous certaines conditions concernant le système étudié, il est possible d'obtenir une abstraction discrète préservant les propriétés du système hybride. Le système discret obtenu permet alors de prouver automatiquement des propriété dynamiques du système original. La recherche d'un tel système discret n'est pas sans rapport avec ce qui est recherché dans une SO, à la différence importante près que cette dernière n'est pas destinée a priori à servir à analyser les propriétés du système observé. L'intérêt pour nous de cette approche réside essentiellement dans le lien qu'elle établit entre la définition d'une trace comme résultant de la discrétisation et mémorisation du comportement d'un processus observé, et les opérations d'abstraction que cette opération peut mettre en jeu.

Si l'on considère que le domaine sémantique d'un processus observé est l'ensemble de ses traces partielles finies, un modèle de sémantique possible est défini comme le plus petit point fixe d'une fonction de transition d'états (voir par exemple dans [24] "partial trace semantics"). Une telle fonction existe toujours, mais il n'y a pas de garantie de pouvoir lui donner une représentation finie. Cette SO, en quelques sortes initiale et que nous qualifierons de sémantique formelle (SF), peut être considérée comme le modèle original. Celui-ci est donné soit comme modélisant l'artéfact, soit comme un a priori théorique. Les SO successives peuvent alors être considérées comme des abstractions de leurs prédécesseurs.

L'interprétation abstraite peut donc intervenir à des stades bien spécifiques de la construction de trace qui sont illustrés sur la figure 7.

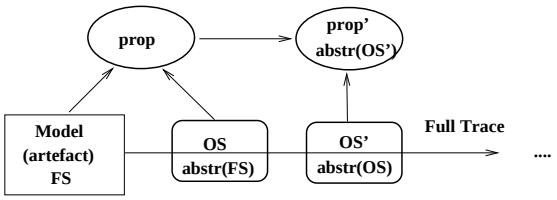


**Fig. 7.** Etapes d'abstractions possibles avec un processus

– **abstr(FS) = OS** : *Du processus observé vers son modèle formel* (deux premières boîtes à gauche). La sémantique formelle (FS) correspond à la sémantique "initiale" du processus observé et la "première" sémantique

observationnelle (OS) sur le schéma en est une abstraction. Cette dernière constitue la SO de la trace, et c'est une abstraction du modèle "initial". Ainsi les traces virtuelle ou effective peuvent être vues comme une abstraction du processus observé. Si le processus observé ne dispose pas d'une description formelle (processus naturel comme dans les exemples ci-dessus issus de l'observation de phénomènes biologiques), sa description formelle est alors une abstraction de la sémantique "initiale" des traces, et joue le rôle de modèle. Noter que cette approche qui revient à se donner un niveau élémentaire de description sémantique est comparable à l'approche sémantique "small step" pour les langages de programmation [24,76].

- **abstr(OS) = OS'** : *Changement de niveau de trace vu comme processus d'abstraction.* Par des opérations d'enrichissement, avec ajout de méta paramètres et sélection, la sous-trace obtenue constitue une abstraction de la trace originale. Celle-ci dispose en théorie d'une sémantique observationnelle SO' qui peut éventuellement être formalisée comme une interprétation abstraite de SO, la sémantique observationnelle de la trace originelle.
- **abstr(OS) = prop** : *Propriété du processus observé comme propriété des traces.* L'interprétation abstraite comme le "model checking" permettent d'établir des propriétés de la trace (celles-ci sont notées dans des élipses sur la figure). Par transitivité, ici par composition d'abstractions, ces propriétés observées sont également des propriétés du processus observé, ou en tous cas, de sa représentation formelle ; cependant les proprétés de SO plus abstraites et qui s'appliquent donc à des ensembles plus larges de traces partielles sont toujours vérifiées par les SO moins abstraites (la flèche qui les relie sur la figure correspond à l'idée d'affaiblissement des propriétés caractéristiques). La connaissance de propriétés d'une SO est utile pour effectuer des preuves de fidélité (voir annexes).

La figure 8 reprend les figures précédentes dans le cas de la trace générique.

On ne peut assimiler la famille des processus "couverts" par la trace générique à un seul processus comme dans la figure 6. Il s'agit bien d'une famille de processus distincts, mais qui sont cependant indistinguables du point de vue de l'observation. Chaque processus de la famille a donc une trace intégrale propre, chacune étant représentée sur la figure par une flèche sortante, comme sur les figures précédentes. Chaque processus de la famille est également relié à la trace générique par une double flèche en pointillés représentant le fait qu'il y a isomorphisme entre une partie de la trace intégrale générique et une partie de la trace intégrale spécifique. Une autre manière de voir les chose est que pour chaque trace spécifique il correspond une sous-trace générique qui en est une abstraction. Il est alors important d'observer que les propriétés établies avec la sous-trace générique correspondant à un processus sont bien aussi des propriétés (éventuellement plus faible) de ce processus. Mais il est possible également qu'une propriété de la SOG s'applique à tous les processus de la famille.

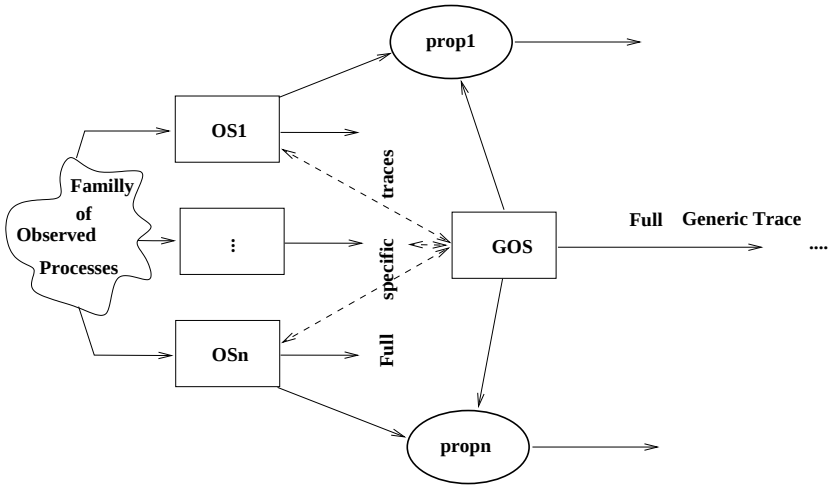


**Fig. 8.** Etapes d'abstractions possibles avec une SO générique

Une abstraction particulière des traces telles que présentés ici consiste à ne considérer que les sous-traces dont le seul paramètre est le type d'action. Une sémantique des traces partielles correspond alors au langage défini par l'ensemble des mots construits sur l'alphabet (supposé fini) des types d'actions associés à chaque trace, c'est à dire l'ensemble des signatures partielles. Le filtrage à la source s'apparente alors à la recherche de sous-séquences dans les signatures.

Dans les travaux de [61] on s'intéresse à identifier les événements de trace non seulement pas leur type d'action, mais également par certains attributs. Ceci est obtenu en se limitant à des langages réguliers de signatures, de telle manière que la reconnaissance d'événements puisse être faite par des automates d'états finis. Cette limitation a pour but de garder une grande efficacité à l'extraction de sous-traces car l'objectif est de pouvoir extraire plusieurs sous-traces d'une trace intégrale simultanément, ce qui s'obtient avec une union de langages réguliers qui est conservative. Il est ainsi montré expérimentalement que l'extraction de plusieurs sous-traces simultanément ne nuit pas à la performance. Cette approche permet de sélectionner des événements de trace particuliers en tenant compte de la succession des événements comme d'une partie de leur contenu, tout en préservant une certaine efficacité.

L'intérêt d'une approche restreinte au langage des signatures est de pouvoir regarder certaines opérations sur l'algèbre des traces comme des opérations sur des langages de signatures. En particulier certaines abstraction et certains raffinements. Une sous-trace (ou abstraction) peut alors être vue comme résultant d'un filtrage opéré sur les mots d'un langage, lequel peut se réduire à associer à des séquences particuliès un type d'action spécifique qui les caractérise alors (à chaque mot fini du langage original correspond un mot plus "synthétique").

Au contraire un raffinement peut être vu comme associant à certaines actions un ensemble de séquences d'actions codées dans un vocabulaire de plus "bas" niveau [12]. Enfin on pourra regarder la signature d'un ensemble de composants logiciels qui tous sont déjà munis de traceurs comme une "union ad hoc" de traces issues d'un système parallèle et distribué.

Une approche générale consisterait donc à étudier les rapports que la méthode de construction de traces peut entretenir avec la théorie des traces telle que originellement introduite par Mazurkiewicz) [5,83,37] et ses développements.

La sémantique observationnelle (partie schéma de traceur) peut alors être vue comme une forme de spécification d'automates de traces. Inversement l'analyse de traces effectives peut être comme une tentative de lui donner une sémantique sous-forme d'automate. Ceci pour une partie de la modélisation des traces.

Par ailleurs, pour certaines preuves de fidélité forte on aura besoin de propriétés dont certaines sont des propriétés du langage de trace qui peuvent être automatisées avec cette approche théorique. C'est par exemple le dans [35], où certaines des propriétés utilisées pourraient être prouvées automatiquement à partir d'une spécification du langage de trace (dans ce cas la propriété peut être caractérisée par une sous-trace correspondant à un langage régulier).

**Des traces vers modélisation et abstraction** Le rapprochement qui est fait ici entre la démarche de recherche d'une SO (et SI fidèle correspondante) et les besoins d'abstraction revient à élargir les champs d'applications, déjà très larges, des approches de la modélisation et des théories de l'abstraction. La recherche d'une méthodologie de construction de traces est à relier à la recherche et l'élaboration d'abstractions pertinentes.

**De la modélisation et l'abstraction vers les traces** De cette brève analyse il ressort que les travaux sur les théories de l'abstraction sont d'une grande utilité pour caractériser formellement certaines opérations de l'algèbre de traces, en particulier les combinaisons d'abstraction et de sélection. De plus, la possibilité d'utiliser cette approche (utilisation de domaines abstraits, techniques de "model checking" appliquées aux LTS) pour prouver des propriétés utiles à la réalisation de preuves de fidélité est une application importante, rendue nécessaire par la taille que peut atteindre le description d'une SO, quelque soit le langage adopté pour la décrire.

Un aspect important, non abordé ici encore, est le langage dans lequel peut être exprimée la SO de la trace intégrale. Différentes approches peuvent être considérées incluant des langages de règles mais dépendant en général du domaine de la famille de processus considérés. Comme le domaine des traces peut être étendu aux traces infinies, les représentations de la sémantique observationnelle peuvent se fonder sur des approche inductives comme des approches co-inductives [2,75].

### 14.3 Fouille de donnés, analyse et interrogation de flots de données

Les traces une fois produites, le but est de les analyser. Le lien entre analyse de trace et fouille de données se fait de plus en plus étroit au fur et à mesure que les tailles des traces augmentent.

La fouille de données est un domaine particulièrement actif compte tenu de la croissance vertigineuse de masses de données dans tous les domaines, en partie produites de manière purement automatique, et dont seul le recours à des moyens automatiques permettra de suivre l'évolution, d'en assurer la cohérence, d'en connaître le sens ou tout simplement de les gérer et de les utiliser. C'est devenu aussi un domaine de recherche en lui même, les chercheurs concernés étant en quelque sorte des "mineurs"[20]. Les données se présentent en général sous forme d'ensemble de traces ou de flots ininterrompu de données, mais la fouille proprement dite peut opérer une succession d'abstractions sur les traces, obtenant ainsi une trace ad-hoc sur laquelle diverses analyses pourront être effectués afin d'en extraire l'infomation recherchée. Le point important ici est que les différents niveaux de trace constituent chacun une trace effective possible, mais aussi que la trace résultant de la fusion des différents niveaux peut être vu comme un enrichissement de la trace initiale.

On mentionne ici quelques systèmes existants qui utilisent différents moyens pour analyser des traces, de manière à en extraire des connaissances particulières. Ce qui suit correspond aussi à différents domaines d'applications où les techniques de fouille de donnés sont utilisées d'abord sur des traces dont on connaît l'origine, puis sur des traces dont l'origine est inconnue. Dans ce dernier cas on ne parle plus de trace, mais de flot de données. Qu'il s'agisse de trace ou de flot de données, les mêmes méthodes d'analyses peuvent s'appliquer, le point commun étant le gigantisme des traces examinées.

L'analyse dynamique de programme appartient également au domaine de la fouille de données sur des traces. C'est par exemple explicitement le cas dans les travaux de Zaidman & al. [85] qui appliquent des techniques de fouilles de données, initialement destinées aux moteurs de recherche sur le Web, pour détecter des classes clefs utiles à la compréhension de programmes. De tels travaux mettent en évidence l'augmentation continue des tailles des traces issues de programmes de plus en plus complexes sur lesquelles on souhaite porter l'analyse. Dans ces travaux on ne se préoccupe pas de retrouver un modèle de comportement du processus observé pour le comparer à une spécification, mais on suppose que l'on va trouver dans la trace suffisemment d'information pour pouvoir en déduire quelque chose de significatif pour orienter la recherche des bogues dans le programme.

Le système SEQ.OPEN [48] est destiné à manipuler et analyser au vol ou à postériori des ensembles de traces produites par un ou plusieurs programmes dans le but, en particulier de faire de la détection d'intrusion. Les traces manipulées sont des traces effectives qui sont en fait des signatures produites par des chemins d'exécution d'un LTS sous-jacent dont les états sont des états actuels.

[20] Les gueules noires modernes sont devenues des "gueules blanches".

L'objectif ici est d'analyser des propriétés de ces traces dans un cadre de logique temporelle, c'est à dire en fait des propriétés du modèle sémantique sous-jacent. Dans ces travaux on utilise une notion de signature éventuellement plus riche, plus proche de la trace effective. Les états du LTS sous-jacent correspondent aux états virtuels.

Une autre approche de l'analyse de trace consiste à utiliser des outils généraux de visualisation pour apprécier l'évolution de certaines observables. La visualisation peut, dans certains cas, faciliter l'observation de caractéristiques remarquables du processus que l'on veut étudier. C'est le cas par exemple de symétries que l'œil humain est particulièrement apte à détecter. Un exemple est donné avec "ILOG Visual CP" [10] ou Infovis [43]. Ces outils utilisent des données prétraitées qui doivent se présenter sous forme de tables homogènes (tableaux à double entrés, dites aussi "table models") sinon complètes et dans lesquels chaque colonne correspond à un paramètre. Un outil comme Infovis produit alors des arbres ou des graphes dérivés de ces tables avec de nombreuses formes de repésentations possibles. La table utilisée est construite par une reconstruction des observables à partir des attributs de la trace effective. Dans le cas où la SO du processus observé est connue, la fidélité de celle-ci et donc la correction de la reconstruction des observables (la SI) sont essentielles.

De manière générale, on peut construire des tables contenant des méta-paramètres. Dans ce cas, chaque entrée de la table peut être également vue comme un événement d'une trace plus abstraite. Une question importante qui se pose alors est l'efficacité de la reconstruction des observables et la complexité du calcul de méta-paramètres.

Cette question est particulièrement cruciale lorsque les observables ne sont pas connues, en particulier parce que l'origine du flot de données est inconnu. Le Web est un exemple de production de flots de données gigantesques et ininterrompu. Cela donne lieu au développement d'une algorithmique des flux massifs (à titre d'illustration, voir [46]), contraints par le besoin de bonne performance. Pour cela, par exemple, on ne cherchera pas à identifier un ensemble particulier, mais seulement des caractéristiques comme sa cardinalité. On pourra alors détecter l'existence d'"éléphants" (recherche d'intrusion sur un réseau) ou au contraire des "souris".

Ce qui est intéressant ici est le lien qu'il peut y avoir entre les algorithmes d'analyse de flot de données et ceux d'extraction d'attributs, de reconstruction d'objets virtuels (les observables ou paramètres), et de construction de méta-paramètres.

Notre approche se situe plutôt en amont. Elle concerne la définition et la (re)construction d'observables élémentaires, celles à partir desquelles on recherchera d'autres observables à l'aide d'outils de fouille de données, de visualisation ou d'autres. Ce qui est en fait recherché dans notre approche, ce sont les éléments minimaux (observables élémentaires) à partir desquels tous les autres pourraient être construits. Cela suppose évidemment que l'on ait une idée de ce que l'on veut construire (les objets que l'on veut analyser) et que l'on ait les moyens d'introduire dans la trace ces éléments minimaux. Du point de vue de l'effica-

cité, cela veut dire que l'on recherche un équilibre entre les temps d'extraction, d'encodage et de transmission des événements de trace et ceux de sa réception, décodage et reconstruction. Mais de toutes manières le temps total de toutes ces étapes est limité, et la plus grande efficacité doit être recherchée, afin de ne pas ralentir excessivement le processus observé. Ceci bien sûr ne vaut que dans le cas où l'origine de flot de données est connue et dans une certaine mesure contrôlable.

Ce qui nous rapproche aussi des travaux sur les flots de données, c'est d'une part le besoin de donner un sens aux flots, c'est à dire de leur donner une sémantique, et d'autre part de définir des langages d'interrogation.

L'aspect langage d'interrogation de traces effectives, dans le but d'en extraire une sous-trace, est abordé dans [61], mais de manière limitée par la nécessité de ne pas ralentir le processus observé. On peut observer ici que les travaux portant sur l'interrogation continue de flots de données [8] peuvent s'appliquer, avec cependant une contrainte spécifique. En effet, le but de l'interrogation ici est non seulement de spécifier une sous-trace virtuelle, mais également d'en déduire la sous-trace effective, juste suffisante, à extraire.

Les outils sémantiques proposés ici peuvent aider à rapprocher les problèmes et à proposer des solutions générales. La sémantique interprétative décrite ici est de même nature que la sémantique des flots de données telles que celles proposées dans [65] et les systèmes de gestion de flots de données (DSMS) utilisent des modèles généraux très proches de le notion de trace effective présenté ici comme dans [16]. Certes ces travaux insistent plus sur les opérateurs sémantiques et algèbres permettant des interrogations aussi sophistiquées que possible, mais tous supposent en général ("Sémantique Web" oblige) que les sources sont incontrolables. Ce n'est pas toujours le cas ; et dans ce cas, il est intéressant de pouvoir séparer, dans l'interrogation, ce qui relève des besoins de l'analyse et ce qui relève de ceux de l'extraction.

**Des traces vers la fouille de données** Il est clair que la fouille de données ou l'analyse de flot de données se situent en aval de notre approche, c'est à dire une fois que la trace à pu être reconstruite comme donnée entrante pour une analyse. Comme on l'a vu le résultat d'une fouille peut se traduire par la production d'une trace plus abstraite. Dans ce cas la formalisation des traces peut aider à mieux structurer les liens entre différentes analyses, en particulier lorsque celles-ci concernent des ensembles de traces issues de différents modules dont les traces ont étgé fusionnées.

Finalement, comme indiqué ci-dessus, l'approche formelle de la production de trace permet de distinguer clairement ce qui relève de l'extraction/reconstruction de trace et ce qui relève de l'analyse. Ceci doit pouvoir clarifier certains aspects de l'analyse de flot de données où l'on pourrait gagner à distinguer reconstruction et analyse avec des sémantiques différentes.

**De la fouille de données vers les traces** : la fouille de traces. D'une relation plus formelle entre ces domaines, on peut espérer des retombées, en utilisant des algorithmes développés dans le champs de l'analyse de flots de données,

sur les manières de construire des traceurs (extraction, reconstruction, abstraction). Deux aspects peuvent être particulièrement concernés. D'une part les algorithmes d'extraction et de reconstruction des observables de la trace virtuelle, comme des attributs de la trace effective. D'autre part les algorithmes d'analyse de propriétés de la SO, quand celle-ci est connue.

Par ailleurs les travaux sur les langages d'interrogation de flots de données peuvent avoir des applications directes sur les langages de sélection utilisés dans les interactions entre analyseurs et traceurs, mais aussi dans leur implantation dans les pilotes.

### 14.4 Modèles événements/actions

Une question qui se pose dans la spécification de modèles "Événement/Condition/Action" (ECA : "on observed Event if Cond do Action") est de déterminer qu'est ce qui est un événement et qu'est ce qui est une action. Le type d'action codée dans la trace virtuelle correspond bien à une action qui a été produite par un processus connu ou inconnu et qui elle-même a produit l'événement de trace effective. L'action en question peut être elle-même un événement de trace. L'action qui a produit la trace peut être très brève alors que sa trace est rémanente (cas de la sensation d'un long éblouissement laissée par un flash lumineux), ou au contraire l'action peut perdurer alors que sa trace est extrêmement brève (cas du flash lumineux, trace d'une explosion nucléaire observée de très loin). On voit ainsi que le même phénomène (le flash lumineux) peut être à la fois action et événement ; c'est une question de point de vue. On peut donc considérer que tout événement de trace est une action potentielle et que toute action peut produire des événements de trace. Le processus observé produit des événements de trace éventuellement sous l'action d'événements produits par d'autres processus, mais dans tous les cas un événement de trace est la conséquence d'une action.

Ce qui est déterminant ici c'est que le processus observé sert de point de référence. Il est le siège d'actions et produit une trace qui est une succession d'événements. La symétrie du modèle peut se voir de la manière suivante :.

Pour la SI :
- **Utrace** : Événement : le ou les événements (de trace) pris en compte
- **AICond** et **OVState** : Condition (identification d'une action pour un état donné)
- **AType** et **RVState** : Action produite (obtention d'un nouvel état)

Pour la SO :
- **AType** : Événement (interne au processus observé)
- **ACond** et **ECond** : Condition (mais une condition non déterminante sur l'état courant)
- **Etrace** et **VSEffect** : Action produite sur un état courant (obtention d'un nouvel état)

Dans le premier cas (SI) un événement de trace est vu comme un événement, et dans l'autre (SO), comme une action. Il ne s'agit pas de contradiction, mais

simplement le fait qu'un événement de trace joue selon le point d'observation le rôle de l'un ou de l'autre.

Ces observations ont pour seul objectif de faire le lien avec les systèmes réactifs et les algèbres d'événements. Cela doit permettre de situer les représentations des sémantiques utilisées ici par rapport aux travaux sur ces systèmes et algèbres, en particulier l'usage fait des LTS.

La revue de Alferes et May [4] montre qu'il y a beaucoup à espérer de tels rapprochements. On peut noter en particulier que dans un cadre ECA, la condition (cas de la SI) correspond à une question (query), qui est précisemment une interrogation de la trace effective. Par ailleurs les LTS sont activement utilisés dans ce domaine et les résultats acquis peuvent grandement faciliter la réalisation de prototypes et vérificateurs de SO par exemples.

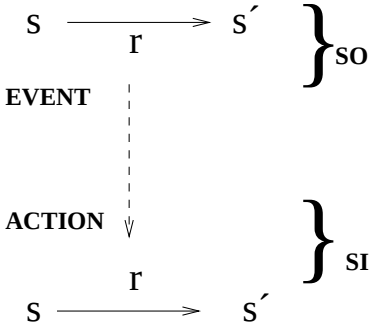


**Fig. 9.** Dualité evénement action : question de point de vue ou de causalité temporelle

Revenant à la question posée au début de cette section, la dualité mise en évidence par les sémantiques SI et SO, peut également s'ennoncer : un événement est une action réalisée dans le passé et une action un événement pour le futur. Ceci est illustré par la figure 9 où la dualité événement /action est illustrée par une dualité des sémantique, au moins dans le cas de fidélité. Un événement de trace décrit également l'action qui le produit. Il est en quelque sorte à la fois événement et action. Mais on peut aussi voir l'événement de trace comme la cause qui va provoquer de nouvelles actions : ce qui n'est au départ qu'un effet se transforme en l'action qui va générer un nouvel événement etc.

**Des traces vers les modèles événements/actions** Les sémantiques observationnelle et interprétative ont ceci de particulier qu'elles ne cherchent pas néces-sairement à décrire complètement le phénomène observé, et elles peuvent se limiter aux fonctions d'extraction et de reconstruction. Lorsque l'on s'intéresse à l'étude de combinaisons de traces laissées par plusieurs processus, il peut en résulter des modèles simplifés d'événements /actions.

**Des modèles événements/actions vers les traces** Les modèles développés dans les cadres ECA peuvent donner des voies pour la réalisation de prototypes et vérificateurs de SO. Ce peut être le cas de modèles simplifiés du "situation calculus" [68,63], ou du "fluent calculus" [82] ou "event calculus" [79] entre autres. Ces formalismes peuvent aider à conduire mieux formaliser les opérations sur les traces.

## 14.5 Fusion de données et analyse du comportement humain

Nous rapprochons ici deux domaines qui sont la fusion de données, essentiellement issues de réseaux de capteurs, et l'observation du comportement humain dans des situations médiées. Chacun de ces deux domaines d'applications constitue un domaine de recherche en lui-même. Il y a cependant des liens forts du fait que, du côté des réseaux de capteurs, on est intéressé à utiliser des techniques de fouille de données pour approcher une representation globale d'un contexte (par exemple, en domotique, l'état de l'habitat et des personnes dans l'habitat) et l'observation du comportement humain dans des situations en interaction avec ce contexte. Nous considérons ici trois applications particulières qui sont la conduite automobile, le débogage de programmes et l'organisation d'un atelier de déstockage et empaquetage.

La "fusion de données" se donne comme problème d'intégrer les informations venant de différentes sources de différentes formes. Cela peut aller de simples réseaux de capteurs à des traitements simultanés d'images, de son et de mouvements. Le terme de fusion de données est abondamment utilisé dans de nombreux domaines applicatifs allant du multimédia à la sécurité et à la défense en passant par la robotique et la médecine. C'est l'exploitation d'informations de sources multiples à propos d'un même environnement pour en tirer plus d'informations que n'en apporte chaque source individuellement.

Présenté ainsi la fusion de données a une traduction directe dans l'algèbre de traces : la fusion/abstraction de traces intégrales en est une théorisation. Les données fusionnées peuvent se présenter sous la forme d'une vaste trace intégrale, résultant d'une fusion/abstraction de traces primitives, et dont il s'agit d'extraire une trace d'un niveau d'abstraction supérieur.

Dans [78] un système à base de traces pour l'apprentissage humain est présenté. Cette approche utilise des traces qui a priori ont un niveau d'abstraction supérieur aux traces effectives intégrales définies ici, mais il ne s'agit en fait que d'une différence de niveau d'abstraction. Ce qui est remarquable c'est la similarité des approches. Ces travaux sur l'apprentissage se situent dans le cadre d'environnements informatiques pour l'apprentissage humain (EIAH). Ce cadre est très général puisque tout système de simulation peut être considéré comme un EIAH, c'est à dire un système dans lequel un opérateur humain peut être observé par un réseau de capteurs (qui peut être éventuellement réduit à la trace de ce qu'il frappe sur un clavier) ainsi que le contexte dans lequel il agit.

Le travail de recherche va consister à définir préalablement de bonnes observables et à obtenir une trace d'un niveau adéquate susceptible de rendre compte

de l'évolution de l'opérateur humain dans son contexte, lui-même en évolution. Ceci fournit les éléments indispensables à la découverte des protocoles d'action utilisés par l'opérateur dans des contextes particuliers. Chaque protocole constitue ce que nous appelons ici un scénario.

**Conduite automobile**

Le projet "Abstract"[21] [49], se donne comme objectif d'étudier les moyens de mieux comprendre une activité humaine à partir d'enregistrements informatisés de cette activité. Une de ses applications phares est l'analyse du comportement du conducteur automobile.

La méthode employée débute par le recueil d'une "trace d'activité" constituée d'événements élémentaires collectés lors d'expérimentations de conduite avec un véhicule instrumenté.

Les données sont obtenues par des caméras, des capteurs (vitesse, angle du volant, enfoncement des pédales, télémètre, position GPS, oculomètre - appareil enregistrant les mouvements des yeux-), et par interview du conducteur. Cette étape de pré-processing inclut des traitements de bas niveau tels que la calibration des capteurs, le filtrage du "bruit", l'élimination des variables non intéressantes. La trace collectée se présente comme une suite d'événements estampillés avec un chrono et est le résultat d'une "discrétisation" des données. Par exemple on extrait les points remarquables des courbes analogiques (Seuils, Mins et Max locaux, Points d'inflexion) ; ou des événements de déclenchement de zones d'intérêt de l'oculomètre. Dans tous les cas l'observation doit produire un ensemble de données datées, c'est-à-dire que chaque élément d'information est rattaché à un instant de l'activité repéré par un "Time-Code".

A la suite de ce premier travail, une trace "première"[22] est donc obtenue qui contient à la fois des observables concernant le comportement du conducteur (gestuelle, témoignage) et contenant des éléments de contexte (position du véhicule, orientation sur la route, position des autres véhicules sur la route, etc...). L'obtention de cette trace première résulte d'une fusion au sein d'un réseau de traces, chaque capteur en particulier produisant sa propre sous-trace première de plus faible niveau d'abstraction.

La trace première est ensuite modélisée dans un système d'ingéniérie des connaissances [58] qui utilise en particulier des techniques de fouille de données. Cet outil logiciel permet de mener un processus progressif d'abstraction des données collectées afin de les rattacher à des concepts relevant de l'étude menée. Ceci peut aboutir à un relevé exhaustif ou à la mise au point de protocoles remarquables, ou scénarii, susceptibles d'utilisation dans différents champs : compréhension du comportement du conducteur (applications dans le champ cognitif) et études sur la prévention d'accidents (champ sécurité), ou amélioration

[21] Description extraite de `http ://liris.cnrs.fr/abstract/abstract.html`.

[22] Dans [27], la première trace construite est dite trace "première, alors que la trace plus abstraite obtenue par construction de nouvelles observables pertinentes pour l'étude que l'on veut mener est dite trace "métier". Dans notre approche la trace métier est donc simplement de trace obtenue à partir de la première par abstraction.

de l'apprentissage (applications dans le champ de l'ergonomie ou de l'enseignement).

**Débogage de programmes**

La mise au point de programmes fait partie des mystères de la programmation, et il n'est pas question ici de se livrer à son étude exhaustive. On se limitera à considérer brièvement cette question sous l'angle de l'analyse du comportement d'un programmeur situé dans le contexte "automatisé", c'est à dire et pour fixer les idées, utilisant un langage de programmation dont la sémantique est bien formalisée, un environnement de programmation pour ce langage tout aussi bien formalisé (avec des possibilités d'édition et d'analyses d'exécutions) et attaché à coder un problème dont l'algorithme est déjà connu[23].

Le débogage de programme est un élément de la mise au point de programmes, laquelle peut être vue comme une spirale où les phases de production de code, essai et débogage s'enchaînent jusqu'à aboutir à un certain niveau de qualité. La phase de débogage vise en général à corriger des défauts (ou "erreurs") suite à une phase de tests non satisfaisants mettant en évidence des symptômes d'erreur. L'observation du programmeur porte sur son comportement lors de la localisation d'une erreur et les modifications conséquentes apportées alors au programme. Elles peuvent également porter sur la manipulation des outils mis à sa disposition (en particulier ceux utilisant des traces d'exécutions).

Déjà en 1988, M. Ducassé note dans [40] la variété des paramètres qui doivent être pris en compte comme la représentation mentale du programme escompté, la compréhension du programme réellement écrit, la maîtrise du langage de programmation, l'expertise générale en programmation, la connaissance du domaine d'application, la connaissance pragmatique des erreurs possibles, et bien sûr la connaissance de l'environnement de déboggage. Elle souligne également l'importance de l'interaction entre le programmeur et l'outil.

Malgré tous les progrès réalisés dans ce domaine jusqu'à aujourd'hui, P. Greussay questionné en 2005 par P. Berger [50] sur l'acte de création en mathématiques (s'agisssant ici essentiellement de la création dans un cadre formel et la programmation) cite ce dialogue typique entre un enseignant et son étudiant.

> *La programmation nous donne un exemple bien connu : la situation où votre programme ne marche pas bien, ou pas du tout, et après tous vos efforts, après des heures de vérification, où vous avez mis tout votre savoir, votre flair, vos outils... vous ne voyez pas, vous ne voyez pas...*
>
> *Alors, vous allez demander de l'aide à quelqu'un, à moi, par exemple. Je viens, je me mets devant le terminal et le dialogue s'engage :*
>
> *- Montre moi le source. - Tu ne veux pas que je t'explique ? - Non, parce que, même si l'explication est intéressante, tu ne m'as pas fait venir pour comprendre la chose, mais pour que je trouve ton erreur.*

[23] Ce point peut être plus délicat à bien formaliser. Une manière de contourner cette difficulté est de considérer que l'algorithme existe dans un autre langage bien formalisé.

> *Alors le source commence à défiler, vous expliquez ce qui se passe, comme d'habitude, avec toutes sortes de gestes, d'emphases, des diminutions du timbre de la voix, des regrets, des excuses quelquefois devant du code que vous avez rédigé de façon un peu hâtive et, à un moment donné, vous dites :*
>
> *- Là, tu vois, lorsque la table prend l'indice n+1 ... - Quoi ? - Tu vois bien, le tableau qui accède à l'élément d'indice n+1. - ... Ah, mais tu as mis n-1 ! - Aaaah !*
>
> *Ainsi, je n'ai rien compris, rien voulu comprendre. Tout ce que j'ai fait, c'est de comparer ce que vous me disiez, ce que vous me disiez de voir, avec ce que je voyais moi-même.*

Les sytèmes d'aide à la mise au point de programmes tendent à proposer des stratégies de recherche d'erreur à partir de symptômes. Ils vont proposer des outils pour identifier des symptômes (analyses statique et dynamique), réaliser des découpages de programmes en portions réduites suspectes (slicing), proposer des stratégies de localisation d'erreur (comme dans le débogage déclaratif où le système se charge de poser des questions pertinentes selon plusieurs stratégies possibles) et finalement faciliter les modifications de programme (éditeurs).

Le dialogue qui précède et la méthode qu'il utilise ne peuvent être envisagé sur des programmes comportant des milliers de modules et des millions de lignes de code. Il montre cependant que, aussi sophistiqués soient-ils, les outils automatiques peuvent ne pas être suffisants. Il illustre simplement une stratégie proprement humaine de localisation d'erreur. Il suggère que toute stratégie automatisée, aussi perfectionnée soit-elle a des limites. Il montre aussi qu'il peut être intéressant de prendre en compte le dialogue éventuel entre programmeurs, c'est à dire une dimension collective et humaine de la construction d'un programme pour l'inclure dans les environnements de mise au point.

Rarement en effet l'analyse directe du comportement du programmeur s'intègre dynamiquement dans un environnement de programmation. Celui-ci contient de manière figée l'expérience personnelle du concepteur et/ou les élément d'enquêtes de besoins auprès de programmeurs (voir [3]), qui guident les concepteurs d'environnement. Parfois ces environnements intègrent des avancées techniques originales comme la création automatique de documentation ("literate programming" [57] et ses successeurs) ou l'édition collaborative. Mais peu d'environnements encore, permettent de prendre en compte les aptitudes des programmeurs, leurs actions et leurs interactions. ***voir Scapin, Bisseret

Le projet OADymPPaC [29] a tenté en 2003 ce type d'approche dans un contexte particulier : le projet ayant développé une trace (format générique pour la résolution de contraintes appelé GenTra4CP) susceptible de permettre l'observation de solveurs de contraintes[24] ; il s'agissait alors d'identifier de bonnes

[24] Dans les solveurs de contraintes (il s'agit ici de réduction de domaines finis) les stratégies de réduction de domaines sont souvent obscures ou imprévues, d'où l'intérêt de pouvoir les observer pour tenter de comprendre certains comportements de solveurs.

observables du comportement d'un programme, c'est à dire celles que l'on pourrait montrer au programmeur pour le mettre sur la piste de corrections à faire ou d'améliorations à apporter. Il semblait alors intéressant de partir de l'observation du comportement des programmeurs. Une expérience a été montée dans ce but. De petits groupes ont été formés, chacun traitant un problème caractéristiques de diverses classes de problèmes en PAC. Chaque groupe devait décrire, sous la forme d'un scénario, les étapes allant de la programmation à la mise au point, en passant par les essais, le monitoring, éventuellement une analyse de trace. Toutes les étapes souhaitées devaient être proposées, le plus librement possible, sur papier.

L'expérience en fait n'a pas produit de résultat exploitable car pour mener à bien ce type d'étude, le projet ne disposait pas d'outils permettant d'observer de manière suffisamment précise le comportement des groupes de programmeurs qui élaboraient les programmes, comme un environnement de travail multimédié[25].

Dans ce type d'approche l'utilisation de système de gestion de traces pour suivre des co-acteurs comme dans [69] (deux sujets tentent ici d'élaborer ensemble la description du mode opératoire d'un origami montré en video) peut être un élément fondamental du dispositif de recherche.

En fait ce que l'on recherche d'essentiel dans ce type d'approche c'est la possibilité de décrire le comportement du ou des programmeurs aussi précisemment que possible, y compris avec tous leurs éléments subjectifs, c'est à dire à la fois le comportement des acteurs et leurs réflexions.

Ceci devrait aboutir à la réalisation de scénarii de mise au point. Ils peuvent être utilisés pour renforcer une démarche d'apprentissage. C'est le cas dans [40] où des scenarii très courts caractérisent des modes de débogage de programmes ([38] décrit une autre langage pour le même objectif). Ils peuvent être utilisés dans un mode réflexif pour un renforcement autonome. Un système de type e-Lycée [27] permet à l'utilisateur de travailler dans un mode adaptatif en mémorisant ses essais et tirant des enseignements de l'analyse de leur trace.

**Observation d'un atelier de (dé)stockage et empaquetage**

Le projet européen NetWMS débuté en 2007 [41], a pour objectif d'intégrer des techniques de réalité virtuelle et d'optimisation dans le cadre de la gestion globale (et en réseau) d'ateliers de stockage et d'expédition. Le noyau du projet repose sur un modèle de remplissage de modules contenants à différents niveaux de granularité (boîtes, palettes, conteneurs, lignes de production, train ou camion,...) et la gestion de leurs déplacements, sa programmation par des techniques d'optimisation reposant sur la programmation par contraintes, et sa simulation par des techniques de réalité virtuelle. Ce noyau doit contribuer à la mise au point d'un produit de niveau industriel.

[25] Le projet OADymPPaC, n'étant pas pluridisciplinaire, n'avait pas prévu que la recherche de bonnes observables pouvait passer par une intervention d'approches cognitives. Les budget étant strictement fléché à l'avance ne permettait pas de telles inflexions.

Le nombre de paramètres intervenant dans ce contexte (de l'atelier de stockage au transport par train d'éléments parfois très complexes -pièces d'un moteur par exemple) peut être extrêmement élevé. Outre les paramètres relativement standards réglant le remplissage de l'espace en fontion de la forme des objets, de leur poids, des moments de chargement/déchargement, de leur fragilité, il y a des paramètres dont la prise en compte est moins facilement formalisables. Les objets sont-ils déformables, pliables, glissant les uns sur les autres ? peut-on les empiler en laissant des espaces, craignent-ils les vibrations, que se passe-il si l'un d'eux se casse, comment peut-on les saisir pour les placer avec un appareil de levage ou un opérateur mais sans contorsion dangereuse, etc...

Deux systèmes ont été développés : un langage de contraintes permettant de produire automatiquement des paquetages dans différents contextes [66,67] et un environnement de simulation basé sur la réalité virtuelle permettant soit une visualisation d'une programmation, soit la recherche manuelle de soultion dans des cas difficiles.

Le langage de contraintes permet de définir des contraintes très élaborées spécifiant des règles métier. Pour autant ce type de modélisation n'aboutit pas toujours à une solution, ou aboutit à une solution insatisfaisante. La simulation permet de faire rechercher manuellement alors, une solution soit en levant des contraintes, soit en les modifiant. Il y a en effet un élément particulier dans ce projet : on peut envisager de tracer tant la partie automatisée (recherche des meilleures solutions avec les paramètres pris en compte) que ce qui reste de non automatique, car, à moins de standardiser complètement une ligne de déstockage/chargement (formes et propriétés prédéfinie des objets et des boîtes, garantissant une solution dans tous les cas), il y aura toujours un frontière à partir de laquelle une intervention humaine sera requise.

La définition d'une trace assez complète avec des observables appropriées permet alors d'envisager un perfectionnement semi-automatique des contraintes utilisées en permettant la découverte et la formulation de nouvelles règles. Trois étapes peuvent aboutir à ce résultat :

- Identification par simulation de scénarii (contexte et procédure utilisée) où l'approche automatisée est en échec.
- Recherche d'une solution par manipulation humaine avec reconnaissance de règles caractérisant cette solution.
- Enrichissement du langage de contrainte avec ces nouvelles règles.

**Des traces vers l'analyse du comportement humain** Le recueil de traces est un outil de base pour l'analyse du comportement humain en contexte. Notre approche met en évidence plusieurs formes de sémantique liées aux traces : compréhension (SI), modèle de production sous-jacent (SO) et abstraction (passage à un autre niveau sémantique). Une telle systématisation peut aider à préciser les niveaux d'analyse et d'observation et à utiliser les bonnes techniques d'analyse au bon niveau. L'algèbre de traces peut aider à améliorer l'utilisation de traces pour l'analyse du comportement humain comme de systèmes automatiques.

**De l'analyse du comportement humain vers les traces** L'approche fusion de données et la nécessité de concevoir et tracer des situations de contexte extrêmement complexes (il s'agit en fait d'un "jeu" à trois entre humains et artéfacts dans un contexte médiké) pose le problème de gestion d'un grand nombre de traces, pouvant amener de ce fait à faire évoluer l'algèbre des traces (donc la méthodologie de développement de traces), probablement par un raffinement de celle-ci et une meilleure connaissance des sémantiques.

## 14.6 Modélisation conceptuelle et WEB sémantique

Lorsque l'on cherche à spécifier un système d'information, la modélisation conceptuelle ("conceptual modeling") est l'activité de spécification et de mise en œuvre des connaissances générale dont le système a besoin pour accomplir certains services.

Cette activité est particulièrement stimulée par le "WEB sémantique", le "WEB 2.0", siège d'accumulation de connaissances, d'interactions de toutes natures, de réseaux sociaux, de collèges visibles et invisibles, de calculs à grande échelle, de bases de données sur tous les domaines numérisables, de services commerciaux ou sans but lucratif, qui se développent, s'enchevêtrent et interagissent de manière de toujours plus variée et accélérée. Selon [28], les technologies du Web sémantique rendront possible des représentations structurelles et sémantiques de documents offrants ainsi des possibilités totalement nouvelles : recherche intelligente au lieu de sélection par mots clés, réponses interprétant les questions au lieu de simples recherche d'information, échange de documents entre les départements d'une entreprise par mise en correspondance élaborées d'ontologies, définition de vues personnalisées de documents. Il est maintenant reconnu que les domaines centraux du WEB sémantique concerne l'automtisation de la structuration des connaissances par des ontologies et le développement de capacités de raisonnement associées[26].

Le réseau européen Rewerse[27] a mis en avant le fait que les langages permettant d'incorporer des raisonnements sont essentiels pour le développement des systèmes WEB. La plupart des applications les plus avancées relevant du WEB semantique font appel à du raisonnement (caractérisé comme capacité logique, déductive ou calculs basés sur des règles).

Une caractéristique du WEB sémantique est que l'acquisition des connaissances nécessaires pour un service particulier est, pour l'essentiel, totalement automatisée (par exemple "logs" des accès à un site, relevé des habitudes de consommation, surveillance des communications téléphonique, collecte d'images d'un télescope, observation de l'activité des utilisateurs de Windows, ..., etc.), puis les connaissances acquises sont réutilisées également de manière totalement

[26] "WEB sémantique" est ici opposé aux aspects techniques et théoriques propres à l'infrastructure WEB, comme les moyens de communication et réseaux, les performances, la fiabilité et la sécurité.

[27] "Reasoning on the Web with Rules and Semantics", `http ://rewerse.net/publications.html`

automatisée. Cette automatisation globale rend particulièrement nécessaire la possibilité de suivre le cheminement des calculs et raisonnements réalisés pour un service particulier. Cette nécessité correspond à des besoins variés allant de la mise au point d'applications, à l'utilisation d'explications afin d'améliorer une application, par exemple, par une méthode auto-adaptative, ou simplement de pouvoir comprendre et justifier le comportement a posteriori du système automatisé.

De nouveaux produits et systèmes exigent la participation de professionnels couvrant de multiples disciplines et de divers autres intervenants, notamment les clients, qui ont besoin de communiquer et de partager une compréhension commune de toutes les caractéristiques du système, ses subtilités, et leurs implications, en partageant un espace commun de concepts. Un élément essentiel dans la solution de ce problème réside dans un accord entre ces spécialistes, et l'adoption d'un langage générique de modélisation conceptuelle, d'un cadre, d'une ontologie et de la méthodologie susceptibles de fournir une base commune pour toutes les parties concernées.

Il y a une convergence croissante en terme de méthodes, langages et outils de développement, pour répondre aux besoins des concepteurs de logiciels qui imposent de combiner des technologies allant des méthodes plus traditionnelles de développement de logiciels comme celles basées sur l'approche MDA ("Model Driven Architecture") ou MDE ("Model Driven Engineering), aux langages de modélisation tels que UML (Unified Modelling Language), RDF ("Resource Description Framework") ou OWL ("Web Ontology Language") par exemple. Les entrepôts de données "Data Warehouse (DW)" et les technologies OLAP ("Online Analytical Processing") sont au cœur des systèmes d'aide à la décision ou des services multimédia.

On voit ainsi qu'au delà de questions du choix d'un langage susceptibles de répondre aux besoins, se posent des problèmes de généricité, c'est à dire la capacité de décrire et de réaliser la modélisation conceptuelle du système projeté, indépendamment de l'implantation dans un langage particulier.

La mise au point et l'analyse du comportement de systèmes d'information réunissant plusieurs domaines différents et plusieurs formes de support de l'information nécessitent de pouvoir observer le déroulement des diverses opérations et calculs dont ils sont le siège. Modélisation conceptuelle et WEB sémantique soulèvent le problème de pouvoir disposer de traces génériques pour analyser l'ensemble d'un système avec des observables susceptibles de couvrir plusieurs aspects.

Sans chercher à apporter de réponse globale à ce type de préoccupation, nous illustrons cette recherche d'observables dans le contexte de la modélisation conceptuelle avec deux approches : conception collaborative de modèles UML, définition de traces pour des extensions du solveur de contraintes CHR.

**Conception collaborative de modèles UML**

Une application remarquable du point de vue de l'utilisation de traces dans le cadre de la modélisation conceptuelle a été réalisée par Xavier Blanc et al.

[13]. Il s'agit d'un moteur d'inférence Prolog capable de détecter les violations de contraintes structurelles et méthodologiques dans le cadre de modèles de pré-spécifications développés en UML2.1, lesquels sont intégrés dans deux environnements de modélisation Eclipse EMF et "Rational Software Architect" (RSA). Le système global est développé par parties de manière distribuée. Chaque développement d'une partie est représenté par la trace de ses modifications. Ce n'est que lorsque l'on cherche à assembler des parties que les traces sont utilisées pour vérifier la cohérence de l'ensemble. Les règles de consistence structurelles et méthodologiques peuvent alors être exprimées uniformément par des contraintes logiques portant sur les séquences de traces propres à chaque sous-modèle utilisé.

Dans ce type d'approche chaque trace joue un rôle explicatif en décrivant précisément les modifications apportés localement à une partie du modèle global. Cette approche relève de l'approche plus générale de la traçabilité dans la conception de modèles [7]. On est alors confronté à la nécessité de traiter ensemble plusieurs niveaux de traces : les traces éventuelles de chaque composants, les traces des programmes explictifs et les traces des modèles à un niveau en quelques sortes supérieur. Il s'agit alors de gérer des traces à différents niveaux d'implantation et de conceptualisation. Cette situation impose des méthodes avec une vision suffisamment générale des objets manipulés que sont ces traces.

**Traces dans CHR$^\vee$ et applications**

Dans [72], J. Robin et J. Vitorino présentent un projet de système, appelé ORCAS, dont le but est de constituer un cadre méthodologique aisément extensible pour des composants logiciels de raisonnement réutilisables. Son originalité tient essentiellement dans la juxtaposition au sein d'un même modèle de génie logiciel de techniques de développement de logiciel basées à la fois sur les approches MDA, composants logiciels, orientation aspects et méthodes formelles. Cette approche repose sur l'utilisation à tous les niveaux d'un moteur d'inférence basé sur CHR[28], qui sert à la fois à encapsuler les données (ou connaissances) et leurs traitements au sein de chaque composant. Cette approche est mise en application dans [73] où il est proposé de reformuler la plupart des tâches impliquant du raisonnement automatique sous forme système de résolution de contraintes. Parmi les conditions essentielles au développement de ce type de plateforme, les auteurs soulignent la nécessité de disposer d'interfaces accueillantes susceptible de fournir des "explications" du comportement du solveur.

Le besoin de pouvoir traiter toutes sortes de raisonnement, y compris des révisions de connaissances ou raisonnement non monotonne, on amené les auteurs à introduire une extension de CHR, CHR$^\vee$ [84][29] muni, entre autres particularités, de la possibilité de comportement adaptatif. La mise en œuvre de ce type de comportement implique de pouvoir produire un trace de l'activité du solveur, l'analyser et modifier les heuristiques de résolution en fonction de l'évolution de certains paramètres. Cela permet gálement de mêler plusieurs

[28] Contraints Handling Rules, `http ://www.informatik.uni-ulm.de/t3-pm/fileadmin /pm/home/fruehwirth/chr-team/publications_index.htm`

[29] Plus précisemment, il est proposé d'utiliser une combinaison de CHR muni de capacités d'auto-adaptation et de CHR$^\vee$, une extension de CHR avec disjonction.

formes de recherche de solutions (exploration dirigée de l'espace de recherche, recherche locale ou autre méthode).

ORCAS, implanté avec les extensions CHR, contient des besoins de trace à la fois au niveau le plus profond (CHR), au niveau méthodologique (basé UML) et au niveau de l'application (raisonnement accessible aux utilisateurs). Ces aspects concernent précisemment la définition de traces intégrales avec enrichissement, abstraction, sélection et fusion ; trace d'autant plus vaste que les outils potentiels d'observation sont plus nombreux. Une question cruciale concerne le contenu de la trace d'un système de raisonnement : fonctionnement détaillé du solveur, simple utilisation des règles, conditions de réveil ou évolution du domaine de connaissances. Dans ces différents cas, le niveau des paramètres n'est pas le même et c'est toute l'algèbre des traces qui doit être mise en œuvre dans un cadre méthodologique méticuleux.

**Des traces vers la modélisation conceptuelle** La notion de trace générique concerne directement la modélisation conceptuelle puisqu'il s'agit de parvenir à définir les observables communes à des domaines hétérogènes et correspondant à différents niveaux d'observation. L'approche algèbre de trace ne fournit pas de méthode directe pour l'obtention de ces observables, puisque celles-ci dépendent du champ d'application. Par contre elle offre un cadre méthodologique pour les rechercher. A ce titre, elle constitue un des éléments à prendre en compte dans l'élaboration du modèle.

**De la la modélisation conceptuelle vers les traces** La modélisation conceptuelle constitue un champ d'application privilégié pour l'algèbre des traces, susceptible de fournir des exemples élaborés de traces génériques. A ce titre elle peut contribuer à l'approfondissement de l'approche. L'approche par la modélisation conceptuelle et la variété des applications multidomaines munies d'ontologies ad-hoc constituent un potentiel d'expériences visant à produire des observables propres à un champ d'applications. La définition de traces pour des moteurs d'inférence susceptibles de modéliser des systèmes informatiques constitue un champ d'expérience privilégié.

## 14.7 Auxiliaire de mémoire

On s'intéresse finalement aux rapports entre l'approche des traces telle que proposée ici et certaines formes de mémoire humaine (au pluriel car il n'y a pas une seule mémoire). Il ne s'agit pas de faire une analyse exhaustive des phénomènes cognitifs liés à ce que l'on appelle couramment la mémoire, mais de voir dans quelle mesure notre approche des traces pourrait jouer un rôle dans l'élaboration d'artéfacts susceptibles d'aider les humains à renforcer leur mémoire, que nous appellerons ici *auxiliaires de mémoire*.

Il faut tout d'abord remarquer que le fait d'externaliser sa propre mémoire dans le but plus ou moins explicite de ne pas la perdre ou simplement de l'entretenir, occupe une partie importante de l'activité humaine. Prendre des notes, les classer, utiliser un agenda, constituer une bibliothèque, organiser son envi-

ronnement participent à un tel but. Les personnes âgées vivent dans un environnement rempli d'objets souvenirs qui contribuent à leur qualité de vie, c'est à dire qu'ils les aident en partie à conserver une partie de leur mémoire. Cette attitude est en particlier trè joliement décrite dans le roman de Delphine Coulin intitulé "Les Traces" [23] où les parsonnes àgées, au delà des traces matérielles de leur environnement qu'elles conservent et qui éventuellement disparaissent à leur mort, peuvent être elles-mêmes vues comme des mémoires vivantes, traces de vie qui progressivement s'estompent. On voit ainsi que la simple automatisation d'une bibliothèque relève de l'utilisation d'un artéfact numérique à des fins d'amélioration d'une mémoire individuelle ou collective. Si l'idée d'auxiliaire de mémoire peut englober en fait de vastes systèmes sociaux, nous nous limitons ici à ce qui est usuellement appellé "mémoire personnelle" plus proche d'un agenda perfectionné susceptible d'accompagner la personne à chaque moment de sa vie, en excluant ici la réalisation d'un cadre global qui serait dans le champ de la domotiques.

En 1945 Vannevar Bush a écrit un article intitulé "As We May Think" dans lequel il jetait les bases de Memex [15] : "un appareil dans lequel un individu peut stocker tous ses livres, musiques et autres éléments de communication, et mécanisé de telle sorte qu'il peut être consulté très rapidement et de manière très flexible". Memex devait avoir une mémoire virtuellement illimitée. Il aurait des annotations, ce que l'on pourrait appeller aujourd'hui des hyperliens. Il imaginait déjà que l'ordinateur pourraît devenir si puissant et traiter des données si variées, qu'il pourraît être utilisé comme extension, sinon parfaite, au moins exhaustive de la mémoire humaine.

On a bien compris aujourd'hui que, malgré toute l'augmentation de la puissance de calcul et de mémoire de machines de plus en plus miniaturisées, la mémoire humaine ne peut être vue simplement comme des mécanisme d'accès à des informations accumulées sans arrêt ni limite. Malgré toute sa richesse combinatoire, aucun cerveau humain n'aurait une capacité de stockage suffisante. La mémoire relève de capacités particulières de stockage temporel, d'abstraction, d'oubli et de rappel sélectif. Ennoncé ainsi, on observera combien ces fonctionnalités de la mémoire sont analogues à l'algèbre des traces étudiée ici, et dont les opérations pourraient s'appliquer comme l'enrichissement, la fusion, l'abstraction et la sélection d'événements de diverses portions de traces produites par nos différents sens.

D'une manière tout à fait classique, comme le résume G.Chapouthier [17] p.64 et suivantes, on peut caractériser la mémoire humaine selon trois axes : sensoriel, temporel et abstrait. Sur l'*axe sensoriel* les sensations tactiles, auditives, visuelles, olphactives, etc. L'*axe temporel* se réfère à la rémanence du souvenir : brève est la mémoire de travail (au plus quelques minutes), dite aussi épisodique ou mémoire transitoire ; plus durable (de quelques heures à plusieurs années) la mémoire dite de référence. Celle-ci correspond aux acquis stables. Enfin sur l'*axe abstrait* s'opposent la mémoire procédurale ou implicite (les habitudes acquises) et la mémoire déclarative ou explicite (les significations). Dans la première le

rappel est spontanné et immédiat (mouvement travaillé du geste sportif par exemple mais dont la réalisation correcte se fait selon des circuits neuronnaux rapides), alors que dans la seconde un appel à la réflexion est nécessaire. C'est celle qui nécessite en particulier le recours à des formes de raisonnement.

Ainsi sommairement classées, chaque mémoire a son mode d'utilisation ou de *rappel*. Par exemple pour la mémoire implicite, le rappel est inconscient et "automatique" -c'est un "savoir comment"-, alors que pour la mémoire explicite, le rappel est conscient et relève de l'application de règles (lesquelles auraient une fonction plus sémantique) -c'est un "savoir que"-. Ces mécanismes de rappel, plus ou moins rapides sont associés à un phénomène d'oubli qui opère un tri dans les deux sens sur ce qui doit être "enregistré" ou non, et ce qui doit être récupéré ou pris en compte.

Notre but ici n'est pas de chercher à réaliser un modèle de la mémoire vivante évoquée ci-dessus, mais essentiellement de rechercher des moyens de renforcement externe, des formes d'extention "mécaniques" non vivantes, qui permettraient de la renforcer.

Disposer d'une mémoire auxiliaire externe aurait pour résultat une amélioration de la mémoire vivante qu'elle est destinée à renforcer. Un tel dispositif pourrait également être utilisé pour améliorer la vie de personnes atteintes de certains troubles mnésiques. Une manière d'aborder cette question est de ne considérer de la mémoire vivante que ce qui peut être externalisé.

On va donc s'intéresser à une forme de mémoire correspondant à des informations digitalisées (axe sensoriel), durables (axe temporel), et essentiellement conscientes (axe abstrait). Toute sensation qui peut être numérisée peut en effet être codée dans un paramètre d'une trace virtuelle (image, son, texte, ...) ; il n'y a donc pas lieu d'être plus spécifique. Nous ne nous préocupons pas ici de la question de l'acquisition ni du codage particulier de ces informations. Ces questions font l'objet de nombreuses recherches (voir en particulier la série de colloques [11]) mais ne nous concernent pas directement ici.

La mémoire de travail n'a pas à être distinguée ici de la mémoire à long terme. Tout événement de trace à une marque de temps (le chrono). La question de savoir s'il s'agit d'un événement propre à la mémoire de travail ou de référence est a priori une simple question d'ancienneté de l'événement[30]. Une analyse plus fine relève du fonctionnement proprement humain de la mémoire, mais surtout, et comme il ne s'agit pas ici d'imiter la mémoire humaine, la présence d'un chrono permet d'envisager toutes sortes de degrés de temporalité.

Enfin sur l'axe abstrait, il n'y a pas non plus à distinguer les traitements rapides inconscients des traitements conscients plus lents ; cette distinction est plus physiologique que calculatoire. En effet la distinction entre une mémoire

[30] Une propriété importante de la mémoire de référence est que le réveil d'un souvenir ancien le fragilise à nouveau de telle sorte qu'il devient labile à son tour, c'est à dire qu'il devra suivre à nouveau un processus de stockage à long terme avec un risque d'oubli. Dans le cadre d'un artéfact tous les événements de trace restent stockés de toutes manières, ce qui constitue d'ailleurs son principal intérêt.

dont le rappel est automatique et donc reposerait éventuellement sur un algorithme, et une mémoire dont le rappel est explicite et donc reposerait sur des règles et leur utilisation dans un "raisonnement", correspond seulement à une représentation différente des calculs à effectuer. La distinction ici est juste une question de codage. Ceci ne veut pas dire que certains traitement opérés sur des traces, et selon les procédés de calcul utilisés, ne puissent être plus ou moins efficaces.

De nombreux travaux concernent plus ou moins directement l'idée de réalisation de mémoires externalisées. Nous en considérons deux. L'intérêt de ces travaux ici est qu'ils illustrent deux aspects complémentaires de l'organisation possible d'une mémoire artificielle : le mémoire vue comme un fichier unique d'événement de traces, ou la mémoire vue comme une collection de fichiers (correspondant chacun à un domaine particulier de stockage d'informations) qu'il faut pouvoir gérer comme un système fichiers.

Dans [56] A.Kiss et J.Quinqueton presentent un système où les événements de trace ont la forme de "stances" situées dans un temps et un espace. Ce sont des fragments significatifs d'histoire personnelle, des fragments contextuels ou des résumés de tranches de vie, qui contiennent des aspects événementiels codables dans un langage et avec des données digitalisées. Le traitement envisagé consiste à stocker séparément les données digitalisées (séquences de bits à partir desquelles on peut reconstituer image, son, vidéo ou tout autre sorte élément d'information reconstituable) et le réseau de "stances". Le rappel des "souvenirs" se fait au final à partir du réseau des stances stockées, indexées et liées par des relations de sens. Ce réseau sert de méthodes d'accès aux ressources. Une implantation a été réalisée sous forme d'une base généalogique de souvenirs personnels, gérée comme une application WEB.

Une autre forme d'approche, complémentaire, est suggérée par un système comme LISFS [70]. C'est une approche logique des systèmes d'informations où chaque composant élémentaire d'information est automatiquement classé (approche ontologique et abstraite basée sur l'analyse de concept) et rappelé au besoin selon un raisonnement plus ou moins efficace dépendant de la logique utilisée. Diverses applications de cette approche ont été réalisées, en particulier une implantation originale et non hiérarchique d'un système de gestion de fichiers. Du point de vue de la gestion de traces mémorielles, cette approche serait applicable en considérant des groupes d'événements de trace comme des fichiers, un raisonnement logique plus ou moins élaboré fournissant une méthode de rappel.

Ce n'est pas l'objet ici de discuter tous les aspects liés à la réalisation de tels artéfacts, ni d'évoquer toutes les voies possibles ni tous les problèmes à résoudre pour y arriver, ni les limites pratiques qu'il faut surmonter pour qu'une mémoire vivante puisse piloter avec profit une mémoire auxiliaire mécanique[31]. Il est clair

[31] Une machine de Turing qui réalise des opérations arithmétique peut être qualifiée de "machine" ou "mécanisme" au même titre que la calculatrice de Pascal constituée de roues dentées et qui réalise les mêmes opérations. Par extension tout algorithme, toute fonction calculable peuvent être qualifiés de machine.

cependant que dans de tels travaux la notion de trace (ici trace mémorielle) est centrale.

**Des traces vers les mémoires** Les mécanismes de rappels dans la mémoire vivante sont certainement plus nombreux et complexes. Ce qui apparait cependant ici est que l'algèbre de traces (accumulation, fusion, sélection et abstraction d'événements de traces) peut aider à décrire certains aspects de la mémoire et ses fonctionnalités de base et de ce fait aider à concevoir des auxiliaires de mémoire.

**Des mémoires vers les traces** On ne peut formuler à ce stade de l'analyse du domaine des retombées directes. Il est certain cependant, comme on le voit ci-dessus, qu'une fois des liens formels établis entre les recherches cognitives sur la mémoire et une approche formelle des traces, des enrichissements propres de l'algèbre des traces ou des adaptations puissent en résulter.

## 14.8 Epistémologie

L'histoire (moderne ou ancienne) peut être vue comme une recherche de traces. Il en est de même dans la géologie, la physique ou l'astronomie comme dans de nombreuses sciences. L'existence de ces familles de traces et les réflexions sur les manières de les construire et de les manipuler est au cœur de l'épistémologie vue comme théorie de la connaissance (au sens large) ou philosophie de la science.

La particularité des traces à laquelle nous nous attachons est le fait qu'une trace est toujours le résultat de la discrétisation d'un éventuel continuum d'origine, dont elle transforme la "ligne de temps" en un relevé discrétisé (chaque événement étant estampillé par le chrono) et spatialisé (déposé sur un support dont une dimension au moins correspond à la ligne de temps et permettant diverses formes de manipulation). Par exemple un livre autobiographique peut être vu comme la trace objectivée d'une histoire subjective, discrétisé par les numéros de page et spatialisée sous forme d'un support papier dont la forme et la consistance permettent certaines manipulations du récit (retour arrière par exemple). Le même livre, aujourd'hui numérisé, permettra bien d'autres manipulations puisque l'on pourra alors par toutes sortes de calculs en extraire toute sortes de nouvelles traces. Une démonstration peut être vue comme le cheminement qui permet le passage d'une connaissance à une autre. Une fois connue, une démonstration peut être analysée, découpée, éventuellement simplifée ; elle peut devenir une source d'inspiration pour d'autres démonstrations.

L'idée de discrétisation de la connaissance a été théorisé par Sylvain Auroux sous forme de la théorie, inspirée de l'évolution des langues et du langage, de la *grammatisation* [9]. Dans sa conférence [81], Bernard Stiegler en rappelle quelques étapes historiques. Deux millénaires après le début du Néolithique, se développent les premiers systèmes de numération permettant de discrétiser le mouvement des étoiles, les crues des fleuves, les réserves de grains, ..., etc. A partir du douxième siècle avant JC environ la société grecque se constitue sur la discrétisation de la continuité de la parole par un système alphabétique. L'imprimerie amène la troisième étape de la grammatisation, permettant désormais

une reproductibilité hors norme de l'écrit qui envahit alors la société. Une quatrième étape est franchie vers la fin du dixseptième siècle avec le développement de la machine outil. Ce type de machine n'est, au final qu'un artéfact capable de reproduire un geste après l'avoir discrétisé. A partir de 1834, des technologies analogiques de discrétisation apparaissent, reproduisant images et sons, permettant de développement d'industries culturelles. Cette évolution conduit par ailleurs à tenter de faire adopter au consommateur des modèles comportamentaux conformes aux investissements industriels. Enfin, selon B.Stiegler, nous en sommes actuellement à une sixième étape où la discrétisation/spatialisation envahit tous les secteurs de l'activité humaine, en liaison avec une dissémination accélérée et sans précédent des moyens de traitement de l'information par la dissémination des moyens de calcul. Cette situation permet le développement d'une véritable et nouvelle économie de la connaissance[32], ou d'une économie dite de l'immatériel[33].

Le point fondamental ici est que toute connaissance objectivée est la trace de quelque chose qui constitue un facteur d'influence ou d'action sur d'autres connaissances, c'est à dire d'autres traces. Ces traces s'enrichissent ensembles et influencent d'autres traces qui agissent sur d'autres niveaux de connaissances.

**Des traces vers l'épistémologie** Voir la démarche scientifique comme création et manipulation de traces résultant d'un (lent) processus de grammatisation par l'homme de la nature, peut aider à identifier certaines démarches d'analyses effectuées dans le champ du domaine de concerné. Une bonne connaissance de ce qui relève de la manipulation de traces peut aider à mieux reconnaître la nature et les significations potentielles des étapes de progression dans le champ de connaissances concerné, ainsi qu'à mieux comprendre la manière dont différents domaines peuvent s'influencer.

**De l'épistémologie vers les traces** Beaucoup de domaines d'étude ont comme fondement la réalisation de traces. Ceci est valable dans les domaines culturel, économique, sociologique, cognitif, nature, physique ou mathématique. Ceci aide à voir les traces comme des connaissances, et les manières selon lesquelles chaque champ de connaissance effectue ces manipulations (abstraction, mise en relation, articulation entre niveaux, explications) sont une source d'enrichissement pour une possible algèbre des traces.

### 14.9 Conclusions sur les domaines fondateurs et applications

Dans cette section nous avons mis en perspectives huit domaines où la notion de trace nous paraissait jouer un rôle significatif : l'ingéniérie de logiciel (sec-

[32] Elle porte aussi en germe ce que certains auteurs considèrent comme une révolution industrielle.

[33] Il peut être utile d'observer que le côté "immatériel" de la connaissance ne tient pas, contrairement aux apparences au coût de plus en plus réduit de ses supports, mais plutôt au fait qu'elle est acquise par l'intermédiaire d'artéfacts, c'est à dire par des procédés d'investigation faisant essentiellement appel à des moyens (calculs, machines, ...) automatisés.

tion 14.1), les théories liées à l'observation et la modélisation (section 14.2), l'analyse des flots de données (section 14.3), les modèles multi agents (section 14.4), l'analyse d'environnements complexes et hétérogènes avec des réseaux de capteurs et celle du comportement humain dans des environnements multimédiés (section 14.5), l'ingéniérie des connaissances à grande échelle comme dans le WEB sémantique (section 14.6), la gestion de données personnelles (section 14.7), et enfin l'epistémologie dans un sens où la connaissance actuelle puise ses sources à la fois dans l'analyse du passé et l'observation du présent (section 14.8). Ces domaines ne sont bien entendu pas indépendants. L'ingéniérie de logiciel concerne tant la mise au point des programmes que des données à analyser, et les approches de modélisation concernent tous les domaines. La gestion de connaissances se retrouve dans les modèles multi agents, les environnements complexes, la gestion de données collectives, individuelles ou même historiques.

Cette étude a pour but de faire ressortir ce qu'ils ont en commun : un besoin de construction de traces avec un niveau de théorisation plus ou moins élaboré selon le caractère plus ou moins empirique ou plus ou moins formel du domaine concerné, au sens où il s'agit de l'observation de phénomènes plus ou moins artificiels.

Dans l'exploration de ces différents domaines on retrouve cinq aspects principaux de la méthodologie de construction de traces :

- **Modélisation du phénomène observé** ou, tout au moins, de ce que l'on veut ou peut en observer. Cet aspect est lié à la définition de traces virtuelles et de sémantiques observationnelles associées.
- **Choix de la représentation des événements de trace et leur interprétation**, c'est à dire définition de traces effectives et la recherche de sémantiques interprétatives associées.
- **Interactions Observant/Observé dans un contexte plus ou moins connu**, c'est à dire définir les rapports que les analyseurs et les phénomènes observés entretiennent, leurs liens de dépendance et les informations qu'ils doivent échanger, tout en tenant compte d'un contexte éventuel.
- **Exploitation de la trace** (analyse) qui amène à reconsidérer certains paramètres ou à en inventer d'autres, ce qui amène à faire évoluer la trace elle-même.
- **Généricité**. Les traces conçues pour un domaine particulier se veulent aussi générales que possible, et dans chaque domaine se pose également la question d'obtenir ainsi des traces aussi réutilsables que possible.

Chaque aspect correspond à une série de problèmes que l'on doit résoudre pour obtenir au fil des expériences une trace utile et satisfaisante. C'est la combinaison de ces problèmes qui permet d'obtenir systématiquement de telles traces et de développer les analyses recherchées afin de mieux comprendre les phénomènes observés.

Bien sûr chaque aspect est plus ou moins présent dans un domaine particulier, de plus certains aspects comportent quelque défis auxquels une approche rigoureuse de la construction de trace peut apporter une contribution.

1. Dans *l'ingéniérie de logiciel* où l'on agit dans un environnement complètement formalisé, il est envisageable de décrire le processus de recherche et construction de traces comme une "algèbre de traces". Cette algèbre comporte essentiellement deux opérations fondamentales l'enrichissement et l'oubli, c'est à dire l'ajout de paramètres d'observation ou au contraire leur élimination. Dans un tel domaine et pour une application particulière (par exemple construction d'un traceur pour un langage formel donné) il pourra être possible de caractériser formellement ces opérations. De plus, on pourra lui appliquer des méthodes développées en génie logiciel pour introduire une approche de modélisation conceptuelle dans le développement des traces et la réalisation de traceurs. Le défi ici correspond à la possibilité d'une approche systématique et très rigoureuse du développement de traceurs.
2. Les théories liées à *l'observation et la modélisation* fournissent des éléments théoriques nécessaires à la compréhension et la formalisation des différents aspects mentionnés, et plus particulièrement les sémantiques et l'étude de propriétés associées. Le défi principal lié à ce domaine est la possibilité de donner un cadre théorique et assez général à cette algèbre de traces. Certes les théories générales de l'abstraction de "disent" pas grand chose, mais elles peuvent fournir un cadre théorique utile à la méthodologie de construction de traces.
3. L'*analyse des flots de données* fournit de nombreux algorithmes pour aider à la recherche de modèles permettant d'approcher les sémantiques. Le défi principal lié à ce domaine semble bien être la définition des langages pour l'interrogation de trace et la sélection de sous-traces compatibles avec les langages de description des sémantiques. L'un des problèmes clés est l'efficacité de la sélection. Dans le domaine du génie logiciel, ceci correspond par exemple à la réalisation de pilotes de traceur efficaces.
4. Les *modèles multi agents* de type ECA ont pour objectif en particulier de décrire les interactions entre processus. Comme on l'a vu ils peuvent permettre également de décrire des interactions entre traces (dans la mesure où celles-ci sont destinées à rendre compte à la fois des processus et d'une certaine communication entre ceux-ci). La SO est une sémantique de l'action et la SI une sémantique de l'événement. Une question intéressante concerne ce qui peut émerger des interactions entre traces, ce qui est lié au fait de tracer un système de traces. Le défi ici est de définir ce que serait une telle trace. Il s'agit également de trouver une approche assez générale pour décrire les sémantique de traces et les tester. Des approches comme celles basées sur le "fluent calculus" [82] pourraient fournir quelques pistes.
5. L'*analyse du comportement humain* dans des d'environnements complexes et multimédiés repose sur l'accumulation de traces d'observations tant de l'opérateur humain que des artéfacts utilisés. Les environnements considérés portent souvent sur un partage de compétences entre des machines et des humains. Le défi ici serait de permettre l'observation dans un tel système de la frontière des capacités, là où les parties coopèrent ou là où elles s'affrontent, c'est à dire d'une certaine manière les limites de l'automatisaqtion.

6. L'*ingéniérie des connaissances* repose sur la construction de vastes ensembles de règles, leur manipulation et leur utilisation à travers des algorithmes de résolution. Le comportement des systèmes qui en résultent peut être particulièrement difficile à saisir et l'utilisation de traces génériques peut aider à l'appréhender. Le défi dans ce cas concerne la possibilité de définir des traceurs pour des sytèmes de règles produisant des traces aussi génériques que possibles afin de faciliter le développement d'outils d'analyse et de contrôle.
7. La *gestion de données personnelles* est un problème particulièrement complexe et la production d'artéfacts, nécessitant le moins d'apprentissage possible et simple d'accès est un véritable chalenge. Nous nous sommes focalisés sur les aides à la mémoire, sous forme de mémoires exosomatiques, mais au delà il peut s'agir de toutes sortes d'outils d'aide à la réalisation de tâches nécessitant un recours constant à des connaissances préalablement stockées. Le défi ici est évident dans la mesure où très peu d'outils encore répondent aux critères d'efficacité pratique et de simplicité.
8. A l'autre extrêmité du "tout formalisé" correspondant au premier domaine considéré, le domaine qualifié d'"*epistémologie*" ne relève guère de formalismes mathématiques. Malgré cela on se rend bien compte de l'influence grandissante de la numérisation dans tous les domaines d'activités humaines, et par ce biais, des mathématiques -tout au moins celles utilisées dans cette numérisation-, ce que Michel Serres appelle "une révolution culturelle et cognitive" [77]. Dans ce contexte, le défi proposé ici consiste à étudier ce que l'algèbre des trace peut apporter dans quelques domaines en proposant une méthode d'analyse de phénomènes dont l'analyse ne relève pas de méthodes purement formelles.

L'étude de quelques domaines fondateurs et applicatifs a permis de mettre en évidence les points cruciaux sur lesquels une étude sur la méthodologie de développement des traces peut être expérimetée. Nous avons essayé également d'identifier sous forme de "défis" les apports principaux qu'une telle étude pouvait apporter à quelques domaines d'application et les apports qu'elle pouvait en tirer pour ses propres développements. Cela constitue en quelques sortes un plan de travail.

## 15 Conclusion (à venir)

Synthèse concernant le "design" des traces comme une forme d'optimisation globale du processus de production, transmition et analyse de traces.

Approche centrée sur l'observation ("externalisation") dans laquelle la généricité est une notion centrale.

## 16 Remerciements

Beaucoup d'idées présentées ici ont été élaborées avec Mireille Ducassé, Gérard Ferrand et Ludovic langevine, originellement dans le cadre du projet OADymP-

PaC [29]. Chacun a contribué à identifier des champs d'application ou d'enrichissement particulier : M. Ducassé et L. Langevine pour le débogage et l'analyse dynamique de programme, et G. Ferrand pour les liens entre la SO et certaines approches algébriques.

## Références


1. L. Aceto, W.J. Fokkink, and C. Verhoef. Structural Operational Semantics. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, pages 197–292. Elsevier Science Publishers, North-Holland, 2001.
2. P. Aczel. Lecture on semantics : The initial algebra and final coalgebra perspectives. In H. Schwichtenberg, editor, *Logic of Computation*. Springer Verlag, 1997.
3. Massimo Fabris & al. Debugging Systems for Constraint Programming (DiSCiPl) - D.WP1.1.M1.1-1 - CP Debugging Needs. Technical report, Inria Rocquencourt, University of Linkoeping, University of Madrid, University of Orléans, Cosytec, ICON, OM Partners, PrologIA, april 1997. Esprit Projet. `http ://web.archive.org/web/20060720225337/discipl.inria.fr/deliverables1.html`.
4. José-Júlio Alferes and Wolfgang May. Reasoning on the Web with Rules and Semantics. TR, Rewerse Project, August 2004.
5. Rajeev Alur and D.L. Dill. Automata for modelling real time systems. In *Proceedings of ICALP'90, LNCS 443*, pages 322–335. Springer-Verlag, 1990.
6. Rajeev Alur, Tom Henzinger, Gerardo Lafferriere, George, and J. Pappas. Discrete abstractions of hybrid systems. In *Proceedings of the IEEE*, pages 971–984, 2000.
7. Bastien Amar, Hervé Leblanc, and Bernard Coulette. A Traceability Engine Dedicated to Model Transformation for Software Engineering. In Jon Oldevik, Goran K. Olsen, Tor Neple, and Richard Paige, editors, *ECMDA Traceability Workshop 2008, Berlin, 12/06/08-12/06/08*, pages 7–16, http ://www.springerlink.com, juin 2008. Springer.
8. Arvind Arasu, Shivnath Babu, and Jennifer Widom. An abstract semantics and concrete language for continuous queries over streams and relations. TR, Stanford University, November 2002.
9. Sylvain Auroux. *La révolution technologique de la grammatisation. Introduction à l'histoire des sciences du langage*. Editions Pierre Mardaga, Liège, 1994.
10. Thomas Baudel. Ilog visual cp. Technical report, ILOG, Gentilly, 2004. voir aussi Réalisation OADymPPaC D4.3.1 et prototype sur `http ://www2.ilog.com/preview/Discovery/gentra4cp/`.
11. Gordon Bell, editor. *The 3rd ACM Workshop on Capture, Archival and Retrieval of Personal Experiences, Santa Barbara, California, USA*. ACM, October 2006. First workshop `http ://research.microsoft.com/CARPE2004/schedule.htm`.
12. Béatrice Bérard, Antoine Petit, and Paul Gastin. Timed Automata with non observable actions : expressive power end refinement. Technical Report LIAFA 97/23, LIAFA, septembre 1997.
13. Xavier Blanc, Isabelle Mounier, Alix Mougenot, and Tom Mens. Detecting model inconsistency through operation-based model construction. In *ICSE '08 : Proceedings of the 30th international conference on Software engineering*, pages 511–520, New York, NY, USA, 2008. ACM.
14. Francisco Bueno, Pierre Deransart, Wlodzimierz Drabent, Gérard Ferrand, Manuel V. Hermenegildo, Jan Maluszynski, and Germán Puebla. On the role of semantic approximations on validation and diagnosis of contraint logic programs. In *AADEBUG*, pages 155–169, 1997.

15. Vannevar Bush. As we May Think. *The Atlantic Monthly*, July 1945. The electronic version was prepared by Denys Duchier, April 1994, `http ://ccat.sas.upenn.edu/~jod/texts/vannevar.bush.html`.
16. Michael Cammert, Jurgen Kramer, Bernhard Seeger, and Sonny Vaupel. A Cost-Based Approach to Adaptive Resource Management in Data Stream Systems. *IEEE Transactions on Knowledge and Data Engineering archive*, 20(2) :230–245, February 2008.
17. Georges Chapouthier. *Biologie de la mémoire*. Odile Jacob, February 2006.
18. Christophe Choquet and Sébastien Iksal. Modèlisation et construction de traces d'utilisation d'une activité d'apprentissage : une approche langage pour la réingénierie d'un EIAH. *Revue STICEF*, 14, 2007.
19. Edmund M. Clark Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.
20. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 2000. See also `http ://fr.wikipedia.org/wiki/Model_checking`.
21. E.M. Clarke, O. Grumberg, and D.E. Long. Model Checking and Abstraction. *CM Transactions on Programming Languages and Systems*, 16(5) :1512–1542, 1992.
22. M. Comini, G. Levi, and M. C. Meo. A Theory of Observables for Logic Programs. *Information and Computation*, 169 :23–80, 2001.
23. Delphine Coulin. *Les traces*. Editions Bernard Grasset, Paris, 2004.
24. P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Proc. of POPL 2002*, pages 178–190, 2002.
25. Patrick Cousot and Radhia Cousot. Abstract Interpretation : a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages, POPL 1977, Los Angeles*, pages 238–252. ACM Press, New York, 1977.
26. Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2/3) :103–179, 1992.
27. Damien Cram. Visualisation de traces : Application aux traces réflexives d'elycée. Rapport de stage, Master recherche Informatique, LIRIS, Université de Lyon 1, Lyon, France, June 2007.
28. Fensel D., Hendler J., Lieberman H., and Wahlster W. Introduction to the semantic web. In Fensel D., Hendler J., Lieberman H., and Wahlster W., editors, *Spinning the Semantic Web. Bringing the World Wide Web to Its Full Potential*, pages 1–25. MIT Press, Cambridge, 2003.
29. P. Deransart & al. Outils d'Analyse Dynamique Pour la Programmation Par Contraintes (OADymPPaC). Technical report, Inria Rocquencourt and École des Mines de Nantes and INSA de Rennes and Université d'Orléans and Cosytec and ILOG, May 2004. Projet RNTL. `http ://contraintes.inria.fr/OADymPPaC`.
30. P. Deransart. On using Tracer Driver for External Dynamic Process Observation. In Alexander Serebrenik and S. Muñoz-Hernandez, editors, *Proceedings of the 16th Workshop on Logic-based Methods in Programming Environments (WLPE'06), a pre-conference workshop of ICLP'06*, Seattle, USA, August 2006.
31. P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog, The Standard ; Reference Manual*. Springer Verlag, April 1996.

32. Pierre Deransart. Comprendre les traces (une approche sémantique des traces). In Nicolas Gregori, editor, *Actes du colloque de l'association pour la recherche cognitive, ARCo 07, Poster*, Nancy, France, November 2007.

33. Pierre Deransart, Mireille Ducassé, and Gérard Ferrand. Observational semantics of the resolution box model. In Wim Vanhoof and Patricia Hill, editors, *Proceedings of the 17th Workshop on Logic-based Methods in Programming Environments (WLPE'07), a post-conference workshop of ICLP'07*, Porto, Portugal, September 2007. `à paraître dans le Computing Research Repository (CoRR)`.

34. Pierre Deransart, Mireille Ducassé, and Gérard Ferrand. Une sémantique observationnelle du modèle des boîtes pour la résolution de programmes logiques. In François Fages, editor, *Actes des troisièmes Journées Francophones de Programmation par Contraintes, JFPC 07*, Rocquencourt, France, June 2007.

35. Pierre Deransart, Mireille Ducassé, and Gérard Ferrand. Une sémantique observationnelle du modèle des boîtes pour la résolution de programmes logiques (version étendue). RR-6229, INRIA, May 2007. `http ://hal.inria.fr/inria-00151285`.

36. Pierre Deransart and Jan Maluszynski. *A Grammatical View of Logic Programming*. The MIT Press, 1993.

37. V. Diekert and G. Rozenberg. *The Book of Traces*. World Scientific Publishing, Singapoore, 1995.

38. M. Ducassé and L. Langevine. Automated analysis of CLP(FD) program execution traces. In P. Stuckey, editor, *Proceedings of the International Conference on Logic Programming*. Lecture Notes in Computer Science, Springer-Verlag, July 2002. Poster. Extended version available at `http ://www.irisa.fr/lande/ducasse/`.

39. M. Ducassé and J. Noyé. Logic Programming Environments : Dynamic Program Analysis and Debugging. *The Journal of Logic Programming*, 19/20 :351–384, May/July 1994.

40. Mireille Ducassé. Scénarios : un paradigme permettant la mise en œuvre de stratégies de localisation d'erreurs de programmation. In *Première ERGO-IA,Biarritz, France*, 1988.

41. F. Fages, A. aggoun, N. Beldiceanu, M. Carlsson, F. Carvalho, P. Gravez, A. Kovcs, and J. Martin. State-of-the-art of enabling technologies for packing and planning in future wms. Technical report, Inria Rocquencourt and KLS Optim and SICS and EMN, 2007. deliverable, no D3.1, European Project, STREP Net-WMS Constraint Optimization in Wharehouse Management Systems, `http ://net-wms.ercim.org`.

42. F. Fages and S. Soliman. Abstract Interpretation and Types for Systems Biology. *TCS*, 14, 2007.

43. Jean-Daniel Fekete. The InfoVis Toolkit. Technical report, INRIA, Centre de Recherche de Saclay, Orsay, 2005. `http ://ivtk.sourceforge.net/`.

44. Gérard Ferrand. Notes sur l'interprétation abstraite, April 1998. Notes établies selon Cousot-Cousot, JLP 1992 corrigé.

45. Gérard Ferrand. Induction et coinduction, 2007. Livre rouge.

46. Philippe Flageolet. Algorithmes probabilistes sur de grandes masses de données. In Gérard Berry, editor, *Chaire d'Innovation technologique - Liliane Bettencourt*. Collège de France, 2007. `http ://www.college-de-france.fr/default/EN/all/inn_tec/`.

47. Thom Fruehwirth. Constraint handling rules : the story so far. In *Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 13–14. ACM, 2006.

48. H. Garavel and R. Mateescu. SEQ.OPEN : A Tool for Efficient Trace-Based Verification. In *Proceedings of the 11th International SPIN Workshop on Model Checking of Software, SPIN'2004 (Barcelona, Spain)*, number 2989 in LNCS, pages 150–155. Springer Verlag, April 2004.

49. Olivier Georgeon, Matthias J. Henning, Thierry Bellet, and Alain Mille. Creating Cognitive Models from Activity Analysis : A Knowledge Engineering Approach to Car Driver Modeling. In *International Conference on Cognitive Modeling*, pages 43–48. Taylor & Francis, July 2007.

50. Patrick Greussay. Stic hebdo, asti, February 2005. `http ://www.ibisc.univ-evry.fr/~asti/Archives/Hebdo/sh44/sh44.htm`.

51. Bart Jacobs. Introduction to coalgebra..., 2005. Draft on internet.

52. J. Jaffar and M.J. Maher. Constraint Logic Programming. *Journal of Logic Programming*, 19/20 :503–581, 1994.

53. E. Jahier, M. Ducassé, and O. Ridoux. Specifying Prolog trace models with a continuation semantics. In K.-K. Lau, editor, *Proc. of LOgic-based Program Synthesis and TRansformation*, London, July 2000. Technical Report Report Series, Department of Computer Science, University of Manchester, ISSN 1361-6161. Report number UMCS-00-6-1.

54. Gilles kahn. Natural semantics. In *Proceedings of STACS'87*, number 247 in LNCS, pages 22–39. Springer Verlag, 1987.

55. Salim Kalla. Trace générique pour CHR sur les domaines finis. Technical report, INRIA, septembre 2007. Rapport de DEA.

56. Adorjan Kiss and Joel Quinqueton. UNISCRIPT : a Model for Persistent and Incremental Knowledge Storage. In *The First ACM Workshop on Continuous Archival and Retrieval of Personal Experiences, CARPE 2004, Columbia University New York*. Microsoft, October 2004.

57. Donald E. Knuth. Literate programming. In *CSLI*, number 27 in Lecture Notes. Center for the Study of Language and Information, Stanford, California, 1992.

58. Julien Laflaquière, Lotfi Sofiane Settouti, Yannick Prié, and Alain Mille. A trace-based System Framework for Experience Management and Engineering. In *Second International Workshop on Experience Management and Engineering (EME 2006) in conjunction with KES2006*, October 2006.

59. L. Langevine, P. Deransart, and M. Ducassé. A propagation tracer for GNU-Prolog : from formal definition to efficient implementation. In C. Palamidessi, editor, *Proc. of 19th International Conference on Logic Programming (ICLP 2003)*, volume 2916, pages 269–283. Springer Verlag, septembre 2003.

60. Ludovic Langevine, Pierre Deransart, and Mireille Ducassé. A generic trace schema for the portability of cp(fd) debugging tools. In K.R. Apt, F. Fages, F. Rossi, P. Szeredi, and Jozsef Vancza, editors, *Recent Advances in Constraints, 2003*, number 3010 in LNAI. Springer Verlag, May 2004.

61. Ludovic Langevine and Mireille Ducassé. A Tracer Driver for Hybrid Execution Analyses. In ACM Press, editor, *Proceedings of the 6th Automated Debugging Symposium*, September 2005.

62. Ludovic Langevine and Mireille Ducassé. A Tracer Driver for Versatile Dynamic Analyses of Constraint Logic Programs. In Alexander Serebrenik and S. Muoz-Hernandez, editors, *Proceedings of the 15th Workshop on Logic-based Methods in Programming Environments (WLPE'05), a pre-conference workshop of ICLP'05*, Sitges, Spain, October 2005. Computer Research Repository cs.SE/0508105.

63. Hector Levesque, Fiora Pirri, and Ray Reiter. Foundations for the Situation Calculus. *Electronic Transactions on Artificial Intelligence*, 2 :159–178, 1998.

64. Salvador Lucas. Observable Semantics and Dynamic Analysis of Computational Processes. Technical Report LIX/RR/00/02, Laboratoire d'Informatique LIX, 2000.

65. David Maier, Jin Li, Peter Tucker, Kristin Tufte, and Vassilis Papadimos. Semantics of data streams and operators. In T. Eiter and L. Libkin, editors, *ICDT 2005*, number 3363 in LNCS, pages 37–52. Springer Verlag, 2005.

66. J. Martin and F. Fages. From business rules to constraint programs in wharehouse management systems. In *Doctoral programme of the 13th Conference on Principles and Practice of Constraint Programming, CP'07*, septembre 2007. `http ://voter.engr.uconn.edu/~ldm/CP2007/DP-proc2007.pdf`.

67. J. Martin and F. Fages. From rules to constraint programs with the rules2cp modelling language. RR-6495, Inria Rocquencourt, April 2007.

68. J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence, 4*, pages 463–502. Edinburgh University Press, 1969.

69. Magali Ollagnier-Beldame. Suivre à la trace l'activité de deux co-acteurs : Le cas d'une rédaction conjointe médiée par un artefact numérique. Technical Report RR-LIRIS-2007-024, LIRIS UMR 5205 CNRS/INSA de Lyon/Université Claude Bernard Lyon 1/Université Lumière Lyon 2/Ecole Centrale de Lyon, September 2007.

70. Yoann Padioleau, Benjamin Sigonneau, and Olivier Ridoux. Lisfs : a logical information system as a file system. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *ICSE*, pages 803–806. ACM, 2006.

71. G.D. Plotkin. A Structural Approach to Operational Semantics. *Journal of Logic and Algebraic Programming*, 60/61 :17–140, 2004. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Denmark, September 1981.

72. Jacques Robin and Jairson Vitorino. ORCAS : Towards a CHR-Based Model-Driven Framework of Reusable Reasoning Components. In *Proceedings of the 20th Workshop on (Constraint) Logic Programming (WLP'06)*, pages 192–199, 2006. `http ://www.kr.tuwien.ac.at/events/wlp06/proceedings.html`.

73. Jacques Robin, Jairson Vitorino, and Armin Wolf. Constraint Programming Architectures : Review and a New Proposal. *Universal Computer Science (J.UCS)*, 13(6) :701–720, 2007.

74. J.J.M.M. Rutten. Universal Coalgebra : a Theory of Systems. *TCS*, 249(1) :3–80, 2000.

75. J.J.M.M. Rutten. On streams and Coinduction. *AMS*, 23 :3–80, 2004.

76. David A. Schmidt. Abstract interpretation of small-step semantics. In *Proceedings of the 5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages. LNCS 1192*, pages 76–99. Springer-Verlag, 1996.

77. Michel Serres. Les nouvelles technologies : révolution culturelle et cognitive. In *L'INRIA a quarante ans*. INRIA, December 2008. http ://www.inria.fr/40ans/forum/pdf/conf-stiegler.pdf.

78. Lotfi Sofiane Settouti, Yannick Prié, Alain Mille, and Jean-Charles Marty. Système à base de trace pour l'apprentissage humain. In *colloque international TICE 2006 Technologies de l'Information et de la Communication dans l'Enseignement Supérieur et l'Entreprise*, October 2006.

79. Murray Shanahan. The event calculus explained. In *Artificial Intelligence Today : Recent Trends and Developments*, number 1600 in LNCS. Springer Berlin / Heidelberg, 1999.

80. Luke Simon, Ajay Mallya, Ajay Bansal, and Gopal Gupta. Coinductive logic programming. In Sandro Etalle and Miroslaw Truszczynski, editors, *ICLP*, volume 4079 of *Lecture Notes in Computer Science*, pages 330–345. Springer, 2006.

81. Bernard Stiegler. Le réseau numérique à l'origine d'un nouveau modèle industriel. In *L'INRIA a quarante ans*. INRIA, December 2007. http ://www.inria.fr/40ans/forum/pdf/conf-stiegler.pdf.

82. M. Thielscher. Introduction to the fluent calculus. *Electronic Transactions on Artificial Intelligence*, 2 :179–192, 1998.

83. Thomas W. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 133–191. Elsevier Science Publishers, North-Holland, 1990.

84. Armin Wolf, Jacques Robin, and Jairson Vitorino. Adaptive CHR meets $\text{CHR}^\vee$ : An Extended Refined Operational Semantics for $\text{CHR}^\vee$ based on Justifications. In *Proceedings of the Fourth Workshop on Constraint Handling Rules (CHR 2007)*, 2007. `http ://chr2007.workshops.free.fr/`.

85. Andy Zaidman, Toon Calders, Serge Demeyer, and Jan Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering (CSMR'05), Manchester, UK*, pages 134–142. IEEE Computer Society, March 2005.

## Colophon

Éléments historiques et explicatifs

Ce document est une ébauche, distribué aux collègues intéressés à fin de collaboration, mais rien n'est encore garanti et les concepts et définitions, encore imparfaits peuvent encore évoluer. Mai 2007 : première ébauche de définition de SO avec LTS.
Août 2007 : première ébauche de définition de SI et lien avec la notion d'adéquation devenue "fidèlité"
Novembre 2007 : reprise de la SO à Recife avec Robin pour tenir compte des aspects "composants logiciels". La notion de "fidèlité" commence à se formaliser proprement (commutativité de $E'$ et $I'$)
Décembre 2007 : Première rédaction faisant apparaître les composants avec trois paradigmes nouveaux : sous-trace, trace enrichie et trace générique.

Février 2008 : la collaboration avec J. Robin doit permettre d'étudier la construction de trace générique à partir de composants munis de leur propre trace générique, et d'obtenir d'autres exemples, en particulier dans le domaine de l'ordonnancement (et GUI Gant Chart) et de jeux multi agents. Le projet Net-Wms peut être un champ d'application en recherchant un enrichissement de gentra4cp dans un cadre de contraintes globales.
Mai 2008 : première version complète de l'approche trace avec définition complète des sémantiques et de leur représentation. Reformulation des notions de fidélité et de l'exemples GNU-Prolog avec trace complète.
Juin, Juillet 2008 : ajout applications et traduction d'une partie en anglais.
Septembre, Octobre 2008 : fin des domaines applicatifs, travail avec J. Robin et Raphael.
Novembre, 2008 : section 3 sur la méthodologie et embrion d'introduction.
Décembre, 2008 : section 14.9, conclusions sur les domaines fondateurs et applications.
Janv à mars, 2008 : section 6 sur les fondements et quelques propriétés, implantation de l'exemple des robots en Prolog et "fluent calculus".

## Annexe : Exemples

On donne ici les exemples de description de traceurs et de traces.

## 17 Exemple : la fonction de Fibonacci

Idealized (biologically unrealistic) rabbit population. Les lapins mettent un mois avant de tous se reproduire et ils ne meurent jamais. La population à l'instant $t$ est donc celle à $t-1$ augmentée des rejetons de la population à $t-2$.

Sémantique Observationnelle $< S, I_f, R_O, A, E, T, S_0 >$, $I_f$ est vide, $T$ est donné par la fonction de transition locale $T_l$.

$S : \mathcal{N}_+^*$ (listes de nombres entiers positifs), $s_t$ is the complete evolution of the population from moment 0 until moment $t+1$ : $s_t = [popu_0, \ldots, popu_t, popu_{t+1}]$

$R_O : \{mg\}$ (monthly growing)

$A : \mathcal{N}_+$ nombres entiers positifs, $a_t$ est la population à l'instant $t+1$ ($popu(t+1)$).

$E : E(mg, s) = plast(s) + last(s)$. There is one rule only to describe $E$.

$T_l : T(mg, s) = s\ o\ [plast(s) + last(s)]$ (respectivement avant dernier et dernier élément de la liste $s$, $o$ denote la concaténation de listes). Le nouvel état au moment $t$ est l'état précédent auquel on ajoute la somme des deux derniers éléments.

$S_0 : s_0 = [1, 1]$.

Une seule règle suffit à décrire la fonction de transition locale $T_l$ et le schéma de trace $E$. Il n'y a pas de facteurs externes et la SO est complète.

**AType** : $mg$
**ACond** : $\{\ true\}$
**ECond** : $\{\ true\}$
**VSEffect** : $\{v \leftarrow plast(s) + last(s), \quad s' \leftarrow s\ o\ [v]\ \}$
**Etrace** : $\{v\}$

*Example 10.*
$T_5^v =< [1,1], [(1, mg, [1,1,2]), (2, mg, [1,1,2,3]), \ldots,$
$(4, mg, [1,1,2,3,5,8]), (5, mg, [1,1,2,3,5,8,13])] >$
$T_5^w =< [1,1], [(1,2), (2,3), (3,5), (4,8), (5,13)] >$
∘

Le schéma de trace est le suivant :

**AType** : $mg$
**VSEffect** : $\{v \leftarrow plast(s) + last(s), \quad s' \leftarrow s\ o\ [v]\ \}$
**Etrace** : $\{v\}$

La sémantique interprétative $< A, R_O, S, I >$ est donnée de la manière suivante :

$A : \mathcal{N}_+$

$R_O : \{mg\}$

$S : \mathcal{N}_+^*$ (liste de nombres entiers positifs), $s_i$ is the complete evolution of the population from moment 0 until moment $i$+1 : $s_i = [popu(0), \ldots, popu(i), popu(i+1)]$

$I : I(< [1,1], \overline{w_t} >) = (i, mg, [1, 1, 2, \ldots, att(w_i), \ldots, att(w_t)])$ $(att(w_i) = popu(i)$ est l'unique attribut de l'événement $w_i)$.

Le schéma de reconstruction est donné par la fonction locale $I_l$

**AType** : $mg$

**Utrace** : $\{v\}$ (attribut unique de l'événementde la trace qui est 0-constructible)

**OVState** : $\{ s \}$ (variable dénotant l'état courant)

**AICond** : $\{ v = plast(s) + last(s)\}$ (condition d'identification de la règle qui est appliquée)

**RVState** : $\{s' \leftarrow s\ o\ [v]\ \}$

Noter que dans la condition d'identification de l'action (**AICond**), la condition peut être vue comme une qualification de l'action "montly growing" (croissance mensuelle), qui assure que l'on reste dans une trace de la "loi de Fibonnacci". Mais elle peut également être vue comme comme une propriété que les traces effectives doivent satisfaire pour être reconstructibles. Ainsi seules sont "reconnues" (car reconstructibles) les traces qui sont des séquences de Fibonnacci.

La SI étant 0-constructible et la SO ne comportant qu'une règle, il est aisé de vérifier que la SO est (fortement) fidèle.

## 18 Exemple : modèle des boîtes pour Prolog

Cet exemple est repris de [34]. Pour une présentation détaillée des motivations historiques de cet exemple et la notion de "port", voir [35].

### 18.1 Sémantique observationnelle

$S : \{T, u, n, nu, pd, cl, fst, ct, flr\}$ (9 paramètres décrits ci-dessous)

$R$ : {Init, Leafreached, Lfrcd&godown, Treesuccess, Treesuc&goright, Tree failed, Backtrack, Bkt&gd, Final}

$A : \{r, l, port, p, c, cl\}$ (6 attributs précisés ci-dessous).

$T$ : voir description ci-dessous

$E$ : voir description ci-dessous

$S_0 : \{s_0\}$ :

$\{\{\epsilon\}, \epsilon, 1, \{(\epsilon, 1)\}, \{(\epsilon, goal)\}, \{(\epsilon, list_of_goal_claus)\}, \{(\epsilon, true)\}, false, false\}$

Notes :

Le programme $P$, qui reste invariant dans ce modèle, est une variable externe à l'état virtuel ; c'est une variable d'influence. C'est aussi une variable implicite de la fonction qui sélectionne les clauses définissant un prédicat.

L'état virtuel contient des éléments d'arbre de preuve partiel courant $T, pd$[34]. Celui-ci est argument de tous les paramètres. Pour alléger la présentation, il sera omis quand cela ne nuit pas à la compréhension du paramètre. $T$ sera représenté par un ensemble de nœuds munis d'un ordre total lexicographique ; $T \in \mathcal{P}(Nodes)$ où $Nodes$ est l'ensemble (non fini) des nœuds ordonnés et $Trees = \mathcal{P}(Nodes)$ l'ensemble de ses parties.

$nu, pd, cl$ et $fst$ sont des fonctions. Toutes sont représentées par un ensemble de paires sur $Dom$ x $CoDom$ ("domain" et "range").

Notations : $T/v$, $v \in T$, le sous arbre de $T$ de racine $v$. $>>$ est l'ordre lexicographique strict sur les nœuds de $T$, $>>=$ est l'ordre incluant l'égalité. Une clause a la forme $p_0 : -p_1, ..., p_j, ..., p_n$ avec $n > 0$ ; $p_0$ est la tête (head) et $p_1, ..., p_j, ..., p_n$ est le corps (body). Si $n = 0$, la clause est un fait. La mise à jour d'une fonction, représentée par un ensemble de couples, nécessite deux opérations : un retrait de couples et un ajout d'autres couples. Une telle mise à jour sera spécifiée par une fonction générale *updt* avec trois arguments : la fonction à mettre à jour, les retraits (ensemble d'éléments premiers dans les paires à retirer) et l'ensemble des nouvelles paires.

**Paramètres :**

1. ***T*** est un ensemble de nœudsreprésentant un arbre (squelette d'arbre de preuve partiel).
2. ***u*** est un nœud de $T$ (le nœud courant) : $u \in Nodes$.
3. ***n*** est un entier positif ($u \in \mathcal{N}^+$), numéro du dernier nœud créé dans $T$. Il n'est utilisé que dans la trace.
4. ***nu*** (number) est une fonction de domaine $Nodes$ et codomaine $\mathcal{N}^+$ (entiers positifs). $nu(v)$ est le numéro de création associé au nœud $v$ dans $T$. On utilisera la fonction inverse $nu_{-1}(n)$ qui à un entier donné $n$ fait correspondre le nœudde l'arbre $T$ dont $n$ est le numéro de création. La fonction inverse n'est définie que pour les numéros de création présents dans les labels de l'arbre $T$.
5. ***pd*** (predication) est une fonction de domaine $Nodes$ et codomaine $H$ ($H$ ensemble de termes). $pd(v)$ est la prédication associée au nœud $v$ dans $T$.
6. ***cl*** (candidate clauses list) est une fonction de domaine $Nodes$ et codomaine "liste de clauses" (en fait liste de clauses marquées). $cl(v)$ est la liste de clauses choisies parmi celles qui définissent le prédicat de la prédication $pd(v)$ associée au nœud $v$ dans $T$ ; l'une d'entre elles est éventuellement "marquée" comme choisie pour développer un nœud). Cette liste est initialisée par une

[34] L'arbre décrit n'est pas exactement un arbre de preuve partiel, car les effets des unifications ne sont propagés qu'aux nœudsvisités. Il en résulte que lors d'un "Exit" l'attribut prédication contient bien une instance résultat, mais les prédications attachées à des nœudsantérieurs peuvent correspondre à des états antérieurs d'unification et ne pas avoir été mis à jour. Ceci provient de ce que l'unification est un facteur d'influence qui n'intervient pas directement sur l'ensemble des labels de l'arbre décrit.

condition externe et mise à jour chaque fois que le nœud est visité. Tant que cette liste est non vide, le nœud $v$ est un point de choix (choice point).

7. ***fst*** (first) est une fonction boolénne de domaine $Nodes$. $fst(v)$ est vrai ssi $v$ est un nœud de $T$ qui n'a pas encore été visité (le nœud est nécessairement une feuille et la valeur vraie dans l'état initial).
8. ***ct*** (complete tree) est un booléen, vrai ($true$) ssi $T$ est un squelette complètement visité (le nœud courant correspond alors à la racine).
9. ***flr*** (failure) est un booléen, indicateur d'état d'échec du sous-arbre $T/u$ de racine $u$ ($true$ si en échec, $false$ sinon, ce qui n'est pas synonyme de succès).

**Fonctions auxiliaires** opérant sur les paramètres et utilisées pour calculer les conditions, les effets (fonction de transition) et l'extraction de la trace. Toutes les fonctions ont $T$ en argument et celui-ci est omis.

- ***pt*** (parent). $pt(v)$ est l'ancêtre direct de $v$ dans $T$. C'est une fonction de domaine et codomaine $Nodes$. Elle est définie par $pt(ui) = u$ et $pt(\epsilon) = \epsilon$.
- ***lf*** (leaf). $lf(v)$ est vraie ssi $v$ est une feuille dans $T$. C'est une fonction booléenne de domaine $Nodes$. Elle est définie par $\nexists w$ such that $vw \in T$.
- ***mhnb*** (may have a new brother). $mhnb(v)$ est vrai ssi $pd(v)$ n'est pas la dernière prédication dans le corps de la clause choisie au nœud $pt(v)$ pour développer ce nœud. C'est une fonction booléenne de domaine $Nodes$. Si la clause choisie a la forme
  $p_0 : -p_1, ..., p_j, ..., p_n$ et $pd(v) = p_j$, alors $mhnb(v)$ est vrai ssi $1 < j < n$. La racine ($\epsilon$) n'a pas de frère ($mhnb(\epsilon) = false$). Noter que, si la définition est suffisante sur le plan conceptuel, son implantation effective nécessite d'introduire dans l'état virtuel la clause choisie ou de ne la supprimer de la liste de clauses qu'à la dernière visite.
- ***nbpd*** (next brother and predication). $nbpd(v)$ est le couple (frère puisné de $v$ dans $T$ et prédication associé). C'est une fonction de domaine $Nodes$ et codomaine $Nodes$ `x` $H$. Si la clause choisie au nœud $pt(v)$ a la forme
  $p_0 : -p_1, ..., p_j, ..., p_n$ et $pd(v) = p_j$ et $v = uj$, alors $nbpd(v) = (u(j + 1), p_{j+1})$ (défini si $mhnb = true$).
- ***ncpd*** (new child and predication). $ncpd(v)$ est un couple (nouvel enfant $w$ de $v$ dans $T$ et prédication associée). C'est une fonction de domaine $Nodes$ et codomaine $Nodes$ `x` $H$. Si la clause choisie au nœud $v$ a la forme
  $p_0 : -p_1, ..., p_j, ..., p_n, n \geq 1$ alors $w = v1$ et $= ncpd(v) = (w, p_1)$.
- ***hcp*** (has a choice point). $hcp(v)$ est vrai ssi il existe un point de choix $w$ dans le sous-arbre de racine $v$ dans $T$ (i.e. $\exists w \in T/v$ such that $cl(w) \neq []$). C'est une fonction booléenne de domaine $Trees$ `x` $Nodes$.
- ***gcp*** (greatest choice point). $w = gcp(v)$ est le plus grand point de choix dans le sous-arbre de racine $v$ selon l'ordre lexicographique des nœuds dans $T$. C'est une fonction de domaine $Nodes$ et codomaine $Nodes$. $gcp(v) = w$ tel que $\forall y \in T/v$ et $cl(y) \neq []$, $w >>= y$.

- ***ft*** (is a fact). C'est une fonction boolénne de domaine $Nodes$. $ft(v)$ est vrai ssi la clause choisie dans $cl(v)$ est un fait (elle a la forme $p_0$.).
- ***fupdt*** (function update). $f' = fupdt(f, A, B)$ où $f'$ est la fonction $f$ dont on a retiré les couples définis par l'ensemble $A$ (ensemble d'èléments modifiés du sous-domaine) et ajouté ceux de l'ensemble $B$ (ensemble de couples).
- ***mrkcc*** (mark chosen clause). C'est une fonction à un ou deux arguments. Avec un argument, la clause choisie n'est pas précisée et les domaine et codomaine sont $List$ (en fait liste de clauses marquées, mais il n'est pas nécessaire à ce niveau de donner plus de détails[35]). $l' = mrkcc(l)$ où $l'$ est la liste $l$ dans laquelle on a marqué un élément. On ne précise pas ici lequel, c'est celui correspondant à la clause qui a été utilisée lors de l'unification traitée de manière "externe". Dans le cas de deux arguments, le deuxième correspond à la clause choisie, marquée dans la liste résultat. $l' = mrkcc(l, c)$ ($c$ est une clause de la liste $l$) où $l'$ est la liste $l$ dans laquelle on a marqué l'élément $c$.
- ***cpudt*** (choice point update). C'est une fonction de domaine et codomaine $List$. $l' = cpudt(l)$ où $l'$ est la liste $l$ dont on a retiré la clause marquée et utilisée (retrait effectué aux événements de ports **Fail** ou **Exit**).
- ***lp*** (length). C'est une fonction de domaine $Nodes$ et codomaine $N$ (entiers naturels). $lp(v)$ est la longueur du chemin de la racine au nœud $v$ dans l'arbre $T$. $lp(\epsilon) = 0$.
- ***ch*** (chosen clause) $ch(v)$ est la clause de $cl(v)$ qui a été choisie pour développer le nœudcourant. La fonction de choix n'est pas précisée ici.

**Fonctions et prédicats auxiliaires** utilisant des facteurs d'influence dans les "conditions externes".

- ***scs*** (success). C'est un prédicat sur de domaine $Trees$ x $Nodes$ x $H$. Deux cas.
  1- le nœud $u$ est une feuille de $T$, alors $scs(T, u, p)$ est vrai ssi la prédication $pd(u)$ est unifiable avec la tête de la clause choisie dans $cl(u)$ dont la forme est $p_0$: $-p_1, ..., p_j, ..., p_n$ avec $n \geq 0$ et $p$ est $p_0$ modifié par l'unification si $n = 0$, ou $p_1$ modifié par l'unification si $n > 0$.
  2-le nœud n'est pas une feuille et dans ce cas il a déjà été unifié, et le sous arbre est un sous-arbre de preuve complet (sous-arbre en succès). Cependant la prédication associée a pu être modifiée par d'autres unifications dans le sous-arbre. $scs(T, u, p)$ est vrai ssi $u$ est racine d'un sous-arbre de preuve complet dans $T$, et $p$ est la prédication éventuellement modifiée associée à $u$.
- ***dcl*** (defining clauses). C'est un prédicat sur de domaine $H$ x $List$ (il s'agit d'une liste de clauses). $dcl(p, c)$ est vrai ssi $c$ est la liste de clauses définissant le prédicat de la prédication $p$.
- ***flr/2*** (external failure). C'est un prédicat sur de domaine $Trees$ x $Nodes$. Deux cas.

[35] Il serait nécessaire d'être plus précis pour une implantation.

1- $T/u$ est une feuille ($= \{u\}$) alors $flr(T,u)$ est vrai ssi la prédication $pd(u)$ n'est unifiable avec aucune des têtes des clauses de $cl(u)$.
2- $T/u$ n'est pas une feuille ($T/u \neq \{u\}$) alors $flr(T,u)$ est vrai ssi le sous-arbre de racine $u$ dans $T$ est en échec (une des feuilles de $T/u$ satisafit la première condition).

**Trace effective**

La trace effective a au plus 6 attributs et chaque événementa la forme

```
t    r    l    Init
t    r    l    Call    p    c   lc
t    r    l    Exit    p
t    r    l    Fail
t    r    l    Redo    p    c
t    r    l    End
```

où

- `t` est le chrono.
- `r` entier, est le numéro de création du nœud$u$ concerné par l'événement de trace, soit $nu(u)$.
- `l` entier, est la profondeur dans l'arbre $T$ du nœudconcerné, soit $lp(u)$.
- `port` est un type général d'événement de trace, elément de { **Init, Call, Exit, Fail, Redo, End** }.
- `p` terme, est la prédication associée au nœudconcerné, soit $pd(u)$.
- `c` clause, est la clause choisie pour développer le nœud$u$ concerné (n'existe que pour les événementsde type **Call** ou **Redo**), soit $ch(u)$.
- `lc` liste de clauses, est la liste des clauses utilisables à un nœud pour développer le nœud $u$ concerné (n'existe que pour les événementsde type **Call** ou **Init**), soit `lc`$= cl(u)$.

L'exemple 1 ci-dessous présente un programme et la trace extraite correspondant au but `:-goal.` ($u$ nœudcourant)

```
c1: goal(X):-p(X),eq(X,b).
c2: p(a).
c3: p(b).
c4: eq(X,X).

:- goal(X).

chrono nu(u) lp(u)  port    pd(u)     ch(u)   cl(u)

  1     1     0     Init
  2     1     0     Call    goal(X)    c1     {c1}
  3     2     1     Call    p(X)       c2    {c2,c3}
  4     2     1     Exit    p(a)
```

```
 5     3     1     Call    eq(a,b)    c4     {c4}
 6     3     1     Fail
 7     2     1     Redo    p(a)       c3
 8     2     1     Exit    p(b)
 9     4     1     Call    eq(b,b)    c4     {c4}
10     4     1     Exit    eq(b,b)
11     1     1     Exit    goal(b)
12     1     0     End
```

Noter que cette trace diffère de la trace usuelle de Prolog du fait des attributs supplémentaires pour les ports **Call** et **Redo**, mais aussi de l'absence de la prdication pour le port **Fail**. Mis à part l'attribut `lp(u)`, qui est une fonction de l'arbre $T$ et du nœudcourant $u$, la trace obtenue est sans redondance.

**AType** : Init
**ACond** : $\{fst(u) \wedge u = \epsilon \wedge T = \{\epsilon\}\}$
**ECond** : $\{\}$
**VSEffect** : $\{ct' \leftarrow false\}$
**Etrace** : $\{nu(u), lp(u), \mathbf{Init}\}$

**AType** : Leafreached
**ACond** : $\{fst(u) \wedge lf(u) \wedge \neg ct \wedge ft(u)\}$
**ECond** : $\{\}$
**VSEffect** : $\{cl' \leftarrow fupdt(cl, \{u\}, (u, mrkcc(cl(u)))),$
$fst' \leftarrow fupdt(fst, \{u\}, \{(u, false)\}), \quad flr' \leftarrow false\}$
**Etrace** : $\{nu(u), lp(u), \mathbf{Call}, pd(u), ch(u), cl(u)\}$

**AType** : Lfrcd&godown
**ACond** : $\{fst(u) \wedge lf(u) \wedge \neg ct \wedge \neg ft(u)\}$
**ECond** : $\{scs(T, u, _) \wedge \ dcl(ncpd_2(u), c')\}$
**VSEffect** : $\{u' \leftarrow ncpd_1(u), \quad T' \leftarrow T \cup \{u'\}, \quad n' \leftarrow n + 1,$
$nu' \leftarrow fupdt(nu, \emptyset, \{(u', n')\}),$
$pd' \leftarrow fupdt(pd, \emptyset, \{(u', ncpd_2(u))\}),$
$cl' \leftarrow fupdt(cl, \{u\}, \{(u, mrkcc(cl(u))), (u', c')\}),$
$fst' \leftarrow fupdt(fst, \{u\}, \{(u, false), (u', true)\}), \quad flr' \leftarrow false\}$
**Etrace** : $\{nu(u), lp(u), \mathbf{Call}, pd(u), ch(u), cl(u)\}$

**AType** : Treesuccess
**ACond** : $\{\neg fst(u) \wedge \neg mhnb(u) \wedge \neg ct \wedge \neg flr\}$
**ECond** : $\{scs(T, u, p)\}$

**VSEffect** : $\{u' \leftarrow pt(u),\quad pd' \leftarrow updt(pd,\{u\},\{(u,p)\}),$
$cl' \leftarrow fupdt(cl,\{u\},\{(u,cpudt(cl(u)))\}),\quad (u=\epsilon) \Rightarrow (ct' \leftarrow false)\}$
**Etrace** : $\{nu(u), lp(u), \mathbf{Exit}, p\}$

**AType** : Treesuc&goright
**ACond** : $\{\neg fst(u) \wedge mhnb(u) \wedge \neg ct \wedge \neg flr\}$
**ECond** : $\{scs(T,u,p) \wedge dcl(nbpd_2(u),c)\}$
**VSEffect** : $\{u' \leftarrow nbpd_1(u),\quad T' \leftarrow T \cup \{u'\},\quad n' \leftarrow n+1,$
$nu' \leftarrow nu \cup \{(u',n')\},$
$pd' \leftarrow fupdt(pd,\{u\},\{(u,p),(u',nbpd_2(u))\}),$
$cl' \leftarrow fupdt(cl,\{u\},\{(u,cpudt(cl(u))),(u',c)\}),$
$fst' \leftarrow fupdt(fst,\emptyset,\{(u',true)\})\}$
**Etrace** : $\{nu(u), lp(u), \mathbf{Exit}, p\}$

**AType** : Tree failed
**ACond** : $\{\neg fst(u) \wedge \neg ct \wedge \neg hcp(u)\}$
**ECond** : $\{flr(T,u) \ \vee \ flr\}$
**VSEffect** : $\{u' = pt(u),\quad (u=\epsilon) \Rightarrow (ct' = true),\quad flr' = true\},$
$cl' \leftarrow fupdt(cl,\{u\},\{(u,cpudt(cl(u)))\})\}$
**Etrace** : $\{nu(u), lp(u), \mathbf{Fail}\}$

**AType** : Backtrack
**ACond** : $\{\neg fst(u) \wedge hcp(u) \wedge ft(gcp(u)) \wedge (flr \ \vee \ ct)\}$
**ECond** : $\{\}$
**VSEffect** : $\{u' \leftarrow gcp(u),\quad T' \leftarrow T - \{y | y > u'\},$
$cl' \leftarrow fupdt(cl,\{y | y \geq u' \in T\},\{(u',mrkcc(cl(u)))\}),$
$ct \Rightarrow (ct' = false),\quad flr' \leftarrow false\}$
**Etrace** : $\{nu(u'), lp(u'), \mathbf{Redo}, pd(u'), ch(u')\}$

**AType** : Bkt&gd
**ACond** : $\{\neg fst(u) \wedge hcp(u) \wedge \neg ft(gcp(u)) \wedge (flr \ \vee \ ct)\}$
**ECond** : $\{v \leftarrow gcp(u),\quad w \leftarrow ncpd_1(v),\quad (scs(T,v,_) \wedge dcl(ncpd_2(w),c))\}$
**VSEffect** : $\{T' \leftarrow T - \{y | y > v\} \cup \{w\},\quad u' \leftarrow w,\quad n' \leftarrow n+1,$
$nu' \leftarrow fupdt(nu,\{y | y > v \in T\},\{(w,n')\}),\quad flr' \leftarrow false,$
$pd' \leftarrow fupdt(pd,\{y | y > v \in T\},\{w, ncpd_2(w)\}),$
$cl' \leftarrow fupdt(cl,\{y | y \geq v \in T\},\{(v,mrkcc(cl(u))),(w,c)\}),$
$fst' \leftarrow fupdt(fst,\{y | y > v \in T\},\{(w,true)\}),\quad ct \Rightarrow (ct' \leftarrow false)\}$
**Etrace** : $\{nu(v), lp(v), \mathbf{Redo}, pd(v), ch(v)\}$

**AType** : Final
**ACond** : $\{u = \epsilon \wedge ct \wedge \neg hcp(u) \wedge \neg fst(u)\}$
**ECond** : $\{\}$
**VSEffect** : $\{\}$
**Etrace** : $\{nu(u), lp(u), \mathbf{End}\}$

**Schéma de trace associé** Le schéma de traceur s'obtient directement à partir de la SO en "oubliant" dans la description précédcente la partie **Etrace**. Pour le schéma de trace, on garde **AType**, **Etrace** et **VSEffect** dans lequel on ne met que les éléments de calcul nécessaires aux calculs des attributs.

Rappel :
$s_0 = \{\{\epsilon\}, \epsilon, 1, \{(\epsilon, 1)\}, \{(\epsilon, goal)\}, \{(\epsilon, list_of_goal_claus)\}, \{(\epsilon, true)\}, false, false\}$

**AType** : Init
**VSEffect** : $\{ct' \leftarrow false\}$
**Etrace** : $\{nu(u), lp(u), \mathbf{Init}\}$

**AType** : Leafreached
**VSEffect** : $\{\}$
**Etrace** : $\{nu(u), lp(u), \mathbf{Call}, pd(u), ch(u), cl(u)\}$

**AType** : Lfrcd&godown
**VSEffect** : $\{\}$
**Etrace** : $\{nu(u), lp(u), \mathbf{Call}, pd(u), ch(u), cl(u)\}$

**AType** : Treesuccess
**VSEffect** : $\{\}$
**Etrace** : $\{nu(u), lp(u), \mathbf{Exit}, pd'(u)\}$

**AType** : Treesuc&goright
**VSEffect** : $\{\}$
**Etrace** : $\{nu(u), lp(u), \mathbf{Exit}, pd'(u)\}$

**AType** : Tree failed
**VSEffect** : $\{\}$
**Etrace** : $\{nu(u), lp(u), \mathbf{Fail}\}$

**AType** : Backtrack

**VSEffect** : {}
**Etrace** : $\{nu(u'), lp(u'), \mathbf{Redo}, pd(u'), ch(u')\}$

**AType** : Bkt&gd
**VSEffect** : $\{v \leftarrow gcp(u)\}$
**Etrace** : $\{nu(v), lp(v), \mathbf{Redo}, pd(v), ch(v)\}$

**AType** : Final
**VSEffect** : {}
**Etrace** : $\{nu(u), lp(u), \mathbf{End}\}$

### 18.2 Sémantique interprétative

La SI est donnée par $< A, R_O, S, I >$ où la fonction de reconstruction $I : \mathcal{T}^w \rightarrow \mathcal{T}^v$ est donnée par une fonction de reconstruction locale $I_l(s, a_{k+1}) = (r_l, s')$, dont la description est un schéma de reconstruction. Celui-ci décrit de quelle manière, à partir d'un état virtuel courant reconstruit $s_0 = \{\{\epsilon\}, \epsilon, 1, \{(\epsilon, 1)\}, \{(\epsilon, goal)\}, \{(\epsilon, list_of_goal_claus)\}, \{(\epsilon, true)\}, false, false\}$ et de, au plus, 2 événements de trace effective (la SI est 1-constructible), un nouvel état virtuel peut être reconstruit.

Rappel :
- **AType** : (Type d'action) un identificateur $r_l \in R_O$ (le type d'action reconnu correspondant à la transition $(s, s')$ ci-dessus et identifiée par la condition ci-dessous)
- **Utrace** : { (Used Trace) les attributs utiles des événementsde trace effective utilisés concernant au plus $k + 1$ événements}
- **OVState** : { (Original Virtual State) les éléments utiles de l'état virtuel-courant reconstruit $s$ }
- **AICond** : { (Action Identification Condition) la condition d'identification du type d'action de la transition effectuée }
- **RVState** : { (Reconstructed Virtual State) les calculs des éléments du nouvel état virtuel reconstruit $s'$}

**AType** : Init
**Utrace** : $\{< r \;\; l \;\; \mathbf{Init} >\}$
**OVState** : {}
**AICond** : $\{fst(u) \wedge u = \epsilon \wedge T = \{\epsilon\}\}$ (vérification facultative que l'on est bien dans un état initial)
**RVState** : $\{ct' \leftarrow false\}$

**AType** : Leafreached

**Utrace** : $\{< r \;\; l \;\; \textbf{Call} \;\; p \;\; ch \;\; lc > ; \; < r' >\}$
**OVState** : $\{\}$
**AICond** : $\{r' = r\}$
**RVState** : $\{cl' \leftarrow fupdt(cl, \{u\}, (u, mrkcc(lc, ch))),$
$fst' \leftarrow fupdt(fst, \{u\}, \{(u, false)\}), \;\; flr' \leftarrow false\}$

**AType** : Lfrcd&godown
**Utrace** : $\{< r \;\; l \;\; \textbf{Call} \;\; p \;\; ch \;\; lc > ; \; < r'p' >\}$
**OVState** : $\{\}$
**AICond** : $\{r' = r + 1\}$
**RVState** : $\{u' \leftarrow ncpd_1(u), \;\; T' \leftarrow T \cup \{u'\}, \;\; n' \leftarrow r',$
$nu' \leftarrow fupdt(nu, \emptyset, \{(u', r')\}),$
$pd' \leftarrow fupdt(pd, \emptyset, \{(u', p')\}),$
$cl' \leftarrow fupdt(cl, \{u\}, \{(u, mrkcc(cl(u), ch)), (u', lc)\}),$
$fst' \leftarrow fupdt(fst, \{u\}, \{(u, false), (u', true)\}), \;\; flr' \leftarrow false\}$

**AType** : Treesuccess
**Utrace** : $\{< r \;\; l \;\; \textbf{Exit} \;\; p > ; \; < r' >\}$
**OVState** : $\{\}$
**AICond** : $\{r' < r \vee u = \epsilon\}$
**RVState** : $\{u' \leftarrow pt(u),$
$pd' \leftarrow updt(pd, \{u\}, \{(u, p)\}),$
$cl' \leftarrow fupdt(cl, \{u\}, \{(u, cpudt(cl(u)))\}),$
$(u = \epsilon) \Rightarrow (ct' \leftarrow false)\}$

**AType** : Treesuc&goright
**Utrace** : $\{< r \;\; l \;\; \textbf{Exit} \;\; p > ; \; < r' \;\; \textbf{Call} \;\; p' \;\; lc' >\}$
**OVState** : $\{\}$
**AICond** : $\{r' > r \wedge u \neq \epsilon\}$ (signifie que $r'$ est le numéro de création du frère $u'$ (nouveau nœud) du nœudcourant $u$.
**RVState** : $\{u' \leftarrow nbpd_1(u), \;\; T' \leftarrow T \cup \{u'\}, \;\; n' \leftarrow r',$
$nu' \leftarrow nu \cup \{(u', r')\},$
$pd' \leftarrow fupdt(pd, \{u\}, \{(u, p), (u', p')\}),$
$cl' \leftarrow fupdt(cl, \{u\}, \{(u, cpudt(cl(u))), (u', lc')\}),$
$fst' \leftarrow fupdt(fst, \emptyset, \{(u', true)\})\}$

**AType** : Tree failed
**Utrace** : $\{< r \;\; l \;\; \textbf{Fail} >\}$
**OVState** : $\{\}$
**AICond** : $\{u = nu^{-1}(r)\}$ (condition de “vérification” facultative)

**RVState** : $\{u' = pt(u), \quad (u = \epsilon) \Rightarrow (ct' = true), \quad flr' = true\}$,
$cl' \leftarrow fupdt(cl, \{u\}, \{(u, cpudt(cl(u)))\})\}$

**AType** : Backtrack
**Utrace** : $\{ < r \;\; l \;\; \textbf{Redo} \;\; p \;\; ch > ; < r' >\}$
**OVState** : { }
**AICond** : $\{ r' = r \}$ (pas de création de descendant, on pourrait aussi tester que $port'$ est égal à **Exit** ou **Fail**)
**RVState** : $\{ u' \leftarrow nu^{-1}(r), \quad T' \leftarrow T - \{y | y > u' \}$ ,
$cl' \leftarrow fupdt(cl, \{y | y \geq u' \in T\}, \{(u', mrkcc(cl(u), ch))\})$,
$ct \Rightarrow (ct' = false), \quad flr' \leftarrow false\}$

**AType** :Bkt&gd
**Utrace** : $\{ < r \;\; l \;\; \textbf{Redo} \;\; p \;\; ch > ; < r' \;\; p' \; \textbf{Call} \;\; lc > \}$
**OVState** : $\{ v \leftarrow nu^{-1}(r)\}$ (calcul d'une information sur l'état courant)
**AICond** : { } (la condition implicite ici est $port' =$ **Call**, une condition équivalente serait $r' > r$)
**RVState** : $\{ u' \leftarrow ncpd_1(v), \quad T' \leftarrow T - \{y | y > v\} \cup \{u'\}$,
$nu' \leftarrow fupdt(nu, \{y | y > v \in T\}, \{(u', r')\})$,
$pd' \leftarrow fupdt(pd, \{y | y > v \in T\}, \{u', p'\})$,
$cl' \leftarrow fupdt(cl, \{y | y \geq v \in T\}, \{(v, mrkcc(cl(u), ch)), (u', lc)\})$,
$fst' \leftarrow fupdt(fst, \{y | y > v \in T\}, \{(u', true)\})$
$n' \leftarrow r', \quad ct \Rightarrow (ct' \leftarrow false), \quad flr' \leftarrow false \}$

**AType** :Final
**Utrace** : $\{ < r \;\; l \;\; \textbf{End} >\} \}$ (identification d'un état final dans la trace, un deuxième événementde trace n'est pas requis)
**OVState** :
**AICond** : $\{ u = \epsilon \wedge ct \wedge \neg hcp(u) \wedge \neg fst(u) \}$ (vérification que l'on est bien dans un état final)
**RVState** : (état non modifié)

On observe que la SI semble mimer la SO donnée précédemment. La SI a pour objectif en effet de décrire l'évolution d'un état virtuel à partir des seules informations recueillies dans la trace effective. On observe bien cependant que des hypothèses sont faites sur ce que la trace est sensée décrire. Les successions d'événements ne sont pas arbitraires. Le reconstruction fait donc des hypothèses sur l'origine et la signification de la trace. Elle tient compte en quelques sortes d'un modèle supposé de production de la trace interprétée.

Une vérification de fidélité forte de la SO, réalisée avec cette SI, permettra de s'assurer que la SO est en fait un bon modèle. La vision duale est que la SI proposée est une bonne fonction de reconstruction qui permet de retrouver l'intégralité des informations observables du phénomène observé à partir de la trace effective transmise.

En tout état de cause on se rend compte ici à quel point la vérification de la fidélité est une garantie d'utilité de la trace et que c'est une propriété essentielle.

## 19 Thielscher's Office Delivery Robot Example

This example is inspired from the office delivery robot's world example of Thielscher [82].

### 19.1 The robots world

This world consists of agents (the robots) moving in a space structured by rooms connected by doors, able to carry objects they find in the rooms. A requisition is an order to seek for objects and carry them from some place to an other one. The scene description at some moment is the current state and consists of a set of facts. A "situation" corresponds to a succession of trace events. The current state corresponding to a given situation is obtained here by the reconstruction function (interpretation semantics).

In fluent calculus, facts are named "fluents" and requisitions (or requests) are similar to Prolog goals. The way the requisitions are computed is not described by the observational semantics. The requests are thus treated as influence factors.

**PARAMETERS**

**House** :

The house has 4 rooms (also named *offices*) $R$ and 7 doors $D$. Each room is the office of some person $P$.

$officeof : P \rightarrow R$
$closed : D \rightarrow Bool$

**Objects** $O$
$atdoor : A \cup O \rightarrow D \cup \perp$
$inroom : A \cup O \rightarrow R$
$carried : O \rightarrow A$ ($carried(o) = a : o$ is carried by $a$).

**Agents** $A$
$atdoor : A \cup O \rightarrow D \cup \perp$
$inroom : A \cup O \rightarrow R$
$carry : A \rightarrow O^*$
$has_code_key : A \rightarrow D^*$

### 19.2 OS and IS Semantics

**INFLUENCE FACTOR**

$request \ \rightarrow \ Q$

**STATES**

Static Part :
*house* (door labelled graph, persons) : $R, D, P$
$officeof : P \rightarrow D$
$connect : R \ \mathtt{x} \ D \rightarrow R$

Dynamic Part :
$inroom : A \cup O \rightarrow R$
$atdoor : A \cup O \rightarrow D \cup \perp$
$closed : D \rightarrow Bool$
$carry : A \rightarrow O^*$
$has_code_key : A \rightarrow D^*$

**INITIAL STATE**

A state has 6 parameters describing the scene.

1 Static parameter *house* :
$R = \{r401, r402, ...\}$
$D = (d12, d23, ..., da1, da2, ...)$
$P = \{alice, bob\}$
$officeof = \{(alice, r402), (bob, r404)\}$
*connect* (see programs below).

5 Dynamic parameters
$inroom(a007) = r401$
$atdoor(a007) = d12$
$closed(d12) = true, \ closed(da1) = false$
$carry(a007) = \emptyset$
$has_code_key(a007) = \{d12, da4\}$

**INFLUENCE STATE**

$q = \exists \ o, request(officeof(alice), o, officeof(bob))$
$request : q \in Q, Q \subseteq R \ \mathtt{x} \ O \ \mathtt{x} \ R \rightarrow Bool$

**ACTIONS TYPES**

**pickup** pick an object (if any)
**drop** drop the carried object (if any)
**gotodoor** go to the quoted door (if any)
**enteroom** enter the quoted room (if the door is open)
**open** open the door (if it is closed)

**TRACE EVENTS** [**attributes**]

Attribute `a` stands for "agent"
Attribute `o` stands for "object"
Attribute `r` stands for "room"
Attribute `d` stands for "door"

```
pickup a o r
drop   a o r
walk   a d
walk   a r
open   a d
```

**AUXILIARY FUNCTIONS and PREDICATES**

$remove : O^*$ `x` $O \rightarrow O^*$
$insert : O^*$ `x` $O \rightarrow O^*$
$carried : O \rightarrow A \cup \perp$.
$is_at_any_door : A \rightarrow Bool$.

**AXIOMS**

Sets like ($O^*$) are represented by lists (lists of objects); the operator `"."` denotes "cons" operator.

$o \notin l \ \Rightarrow \ remove(o.l, o) = l$
$o \in l \ \Rightarrow \ remove(o.l, o) = remove(l, o)$
$o \neq x \ \Rightarrow \ remove(x.l, o) = x.remove(l, o)$
$insert(l, o) = o.l$
$carried(o) = \perp \Leftrightarrow \forall a, o \notin carry(a)$
$carried(o) = a \Leftrightarrow o \in carry(a)$
$carried(o) = a \Rightarrow inroom(o) = inroom(a)$
$inroom(a) = r \wedge carried(o) = a \ \Rightarrow \ inroom(o) = r$
$is_at_any_door(a) \Leftrightarrow \exists \ d, atdoor(a) = d$
$is_open(d) \Leftrightarrow \neg closed(d)$

**Observational Semantics (OS)**

- **AType** : pickup
- **ACond** : $\{inroom(a) = inroom(o) = r \wedge o \notin carry(a) \wedge \neg is_at_any_door(a)\}$
- **ECond** : $\{\exists \ r \ request(r, o, r)\}$
- **VSEffect** : $\{carry(a) \leftarrow insert(carry(a), o)\}$
- **Etrace** : {`pickup a o r`}

- **AType** : drop
- **ACond** : $\{o \in carry(a) \wedge inroom(a) = r \wedge \neg is_at_any_door(a)\}$

- **ECond** : $\{\exists\ r'\ request(r', o, r)\}$
- **VSEffect** :$\{carry(a) \leftarrow remove(carry(a), o)\}$
- **Etrace** : {`drop a o r`}

- **AType** : gotodoor
- **ACond** : $\{inroom(a) = r \wedge \exists\ r'\ connect(d, r, r') \wedge \neg is_at_any_door(a)\}$
- **ECond** : {}
- **VSEffect** :$\{atdoor(a) \leftarrow d\}$
- **Etrace** :{`walk a d`}

- **AType** : enteroom
- **ACond** : $\{atdoor(a) = d \wedge inroom(a) = r \wedge is_open(d) \wedge connect(d, r, r')\}$
- **ECond** :{}
- **VSEffect** :$\{inroom(a) \leftarrow r',\ \ atdoor(a) \leftarrow \bot\}$
- **Etrace** :{`walk a r'`}

- **AType** : open
- **ACond** : $\{atdoor(a) = d \wedge d \subseteq has_code_key(a)\}$
- **ECond** :{}
- **VSEffect** :$\{closed(d) \leftarrow false\}$
- **Etrace** :{`open a d`}

In the original example it is : **ACond** : $\{atdoor(a) = d \wedge (\neg closed(d) \wedge connect(d, r, r'))\}$

**Interpretation Semantics (IS)**

**INITIAL STATE** (repeated)

$house$ :
$R = \{r401, r402, ...\}$
$D = (d12, d23, ..., da1, da2, ...)$
$P = \{alice, bob\}$
$officeof = \{(alice, r402), (bob, r404)\}$
($connect$ is ommitted).
$inroom(a007) = r401$
$atdoor(a007) = d12$
$closed(d12) = true,\ closed(da1) = false$
$carry(a007) = \emptyset$
$has_code_key(a007) = \{d12, da4\}$

- **AType** : pickup
- **Utrace** : {`pickup a o`}
- **OVState** : {}
- **AICond** : {}
- **RVState** : $\{carry(a) \leftarrow insert(carry(a), o)\}$

- **AType** : drop
- **Utrace** : {`drop a o`}

- **OVState** :{}
- **AICond** :{}
- **RVState** :{$carry(a) \leftarrow remove(carry(a), o)$}

- **AType** : gotodoor
- **Utrace** : {`walk a d`}
- **OVState** :{}
- **AICond** : {$door(d)$}
- **RVState** :{$atdoor(a) \leftarrow d$}

- **AType** : enteroom
- **Utrace** :{`walk a r`}
- **OVState** :{}
- **AICond** :{$room(r)$}
- **RVState** :{$inroom(a) \leftarrow r, \quad atdoor(a) \leftarrow \perp$}

- **AType** : open
- **Utrace** : {`open a d`}
- **OVState** :{}
- **AICond** :{}
- **RVState** :{$closed(d) \leftarrow false$}

Notice that only two events need a condition to be expressed `gotodoor` and `enteroom`, in order to identify the corresponding virtual trace event. Such conditions would not be necessary if the type of the third attribute is coded in the actual trace event. Conditions may also be completed by conditions occurring in the OS.

### 19.3 OS in Prolog style

We describe a simplified version of the example of [82] with 3 rooms instead of 4.

The simplified robot's world is depicted on Figure 10.

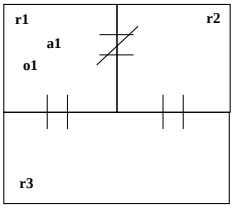


**Fig. 10.** A simple robot world

Initially there is one object and one robot both located in the same room, and the door `d12` is locked. The robot has its key.

We present an implementation of the OS in ISO-Prolog.

This presentation assumes a pure logical Prolog data base update view. Explicite predicate manipulations are omitted here[36]. Furthermore the CWA approach is used (closed world assumption)[37].

A current state is described by a set of atoms.

A successful call of `try(+n, ?at, ?vt)`, where `n` is a given positive integer and `at` a variable, instanciates `at` by an actual trace of length `n` (a list of trace events, each event beeing a list with the chrono and 2 or 3 attributes).

If `at` is a given trace of length `n`, it is a verification that the given trace complies the OS.

As `try/3` is resatisfiable, all possible traces of given length `n` compatible with transition function are given.

In all cases `vt` is the virtual trace whose extracted actual trace is `at`.

```
Context and Types:

room(r1),room(r2),room(r3),
door(d12),door(d13), door(d23),
object(o1),
agent(a1),
connect(r1,d12,r2), connect(r2,d12,r1), connect(r1,d13,r3),
connect(r3,d13,r1), connect(r2,d23,r3), connect(r1,d23,r2).
```

Remark : the robot cannot walk back through a door, since connect is not reflexive (It needs to be in state "atdoor" to go through).

```
Possible initial state (evolving fluents)

inroom(a1,r1),
inroom(o1,r1),
closed(d12),
haskey(a1,[d12]).

%try(+n,?at,?vt) if n is a positive integer and at a list of length n,
%then at is a trace sample of length n (all samples which are consistent
%with the OS are produced),
%and vt is the virtual trace whose extracted trace is at.

try(N, ATr, [S0|VTr]) :- length(ATr, N), s0, state(S0), so(1, ATr, VTr).

%so(+n,+at,?vt): if n is a positive integer and at a list of length m
%(m >= 0), then at is a trace sample of length m (all samples which are
%consistent with the OS are produced).
```

[36] Indeed in ISO-Prolog a different data base update policy is used [31], therefore specific database update must be included (see full version of the programs).

[37] The CWA asssumes that a non true fact is false.

```
%Remark: n+m is constant.
%Remark: a current state is supposed given. Therefore this predicates
%does not work alone.

so(_, [], []) :- !.
so(Ch, [[Ch|AtE]|ATr], [[Ch|VS]|VTr]):- tr_extract(Ch,AtE), state(VS),
                                        Chp is Ch+1, so(Chp, ATr, VTr).

%OS transition et extraction
%tr_extract(+ch, ?et): if ch is a positive integer (the chrono of a trace
%event), then et is the trace event corresponding to a transition of
%chrono ch.

tr_extract(Ch, [pickup,A,O,R]) :- trans(Ch, pickup(A,O,R)).
tr_extract(Ch, [drop,A,O,R])   :- trans(Ch, drop(A,O,R)).
tr_extract(Ch, [walk,A,D])   :-  trans(Ch, gotodoor(A,D,_)).
tr_extract(Ch, [walk,A,R])   :- trans(Ch, enteroom(A,_,R)).
tr_extract(Ch, [open,A,D])   :- trans(Ch, open(A,D)).

%trans(n,tr): if n is a positive integer (the chrono of a trace event),
%then (on success) a transition caracterized by the term tr (wich contains
%all information to generate the corresponding trace event) is realized.
%Remark: only the "gotodoor" case is nondeterministic (as many cases as
%there are doors in a room).
%Remark: state database updates are omitted here.

trans(Ch, pickup(A,O,R)) :-
 agent(A), object(O),
 inroom(A,R), inroom(O,R), \+is_at_any_door(A), \+carry(A,O),
 asserta(carry(A,O)).

trans(Ch, drop(A,O,R)) :-
 agent(A), object(O),
 inroom(A,R), inroom(O,R), \+is_at_any_door(A), carry(A,O),
 retract(carry(A,O)).

trans(Ch, gotodoor(A,D,R)) :-
 agent(A),
 inroom(A,R), \+is_at_any_door(A), chose_door(Ch, R, A, D).

trans(Ch, enteroom(A,D,Rp)) :- restore_allS(Ch),
 agent(A),
 inroom(A,R), atdoor(A,D), connect(R,D,Rp), is_open(D),
 retract(inroom(A,R)), retract(atdoor(A,D)), asserta(inroom(A,Rp)),
 (carry(A,O) -> (retract(inroom(O,R)), asserta(inroom(O,Rp))) ; true).
```

```
trans(Ch, open(A,D)) :-
 agent(A),
 atdoor(A,D), closed(D), can_open(A,D),
 retract(closed(D)).

%chose_door(c, r, a, d): if c is the chrono, r a room and a an agent,
%then d is a possible door where to go.
%Remark: the predicate is resatisfiable as many times there are doors
%in a room; the fluents are updated accordingly to the choosen door.

chose_door(Ch, R, A, D) :- connect(R,D,_), asserta(atdoor(A,D)).

%----------------------------------------AXIOMS

is_open(D) :- \+closed(D).

is_at_any_door(A) :- agent(A), door(D), atdoor(A,D).

can_open(A,D) :- haskey(A,L), member(D,L), !.
```

Here is a sample of traces of size 3, obtained with goal `try(3,T)`.

```
[1,pickup,a1,o1,r1] , [2,drop,a1,o1,r1] , [3,pickup,a1,o1,r1]

[1,pickup,a1,o1,r1] , [2,drop,a1,o1,r1] , [3,walk,a1,d12]

[1,pickup,a1,o1,r1] , [2,drop,a1,o1,r1] , [3,walk,a1,d13]

[1,pickup,a1,o1,r1] , [2,walk,a1,d12]   , [3,open,a1,d12]

[1,pickup,a1,o1,r1] , [2,walk,a1,d13]   , [3,walk,a1,r3]

[1,walk,a1,d12]     , [2,open,a1,d12]   , [3,walk,a1,r2]

[1,walk,a1,d13]     , [2,walk,a1,r3]    , [3,walk,a1,d13]

[1,walk,a1,d13]     , [2,walk,a1,r3]    , [3,walk,a1,d23]
```

Or some long observation : a kind of loop (a possible trace 10 événements)

```
[1,pickup,a1,o1,r1] , [2,drop,a1,o1,r1] , [3,pickup,a1,o1,r1] ,
[4,drop,a1,o1,r1] , [5,pickup,a1,o1,r1] , [6,drop,a1,o1,r1] ,
[7,pickup,a1,o1,r1] , [8,drop,a1,o1,r1] , [9,pickup,a1,o1,r1] ,
[10,drop,a1,o1,r1]
```

or a sort of hesitation (go to room `r2` then come back to drop the object in the original room).

```
[1,pickup,a1,o1,r1] , [2,walk,a1,d12] , [3,open,a1,d12] ,
[4,walk,a1,r2] , [5,walk, a1, d12] , [6,walk,a1,r1] ,
[7,drop,a1,o1,r1] , [8,pickup,a1,o1,r1] , [9,drop,a1,o1,r1] ,
[10,walk,a1,d13]
```

**Remarque**

On obtient également une trace de taille donnée comme définie à la section 6.3 avec le programme suivant.

```
%os2(n, tr): if n is a positive integer, tr is a finite actual trace in S0 of size n.

os2(N,T) :- tr(N,U), reverse(U,T).

tr(N, [(_,S)]) :- N =< 0, s0, state(S), !.
tr(N, [E2,E1 | T_v]) :-  N2 is N-1, tr(N2, [E1| T_v]), tr_extract(N, E2).
```

### 19.4 OS Properties

Dans cet exemple, on se donne un état initial et les traces issue de cet état sont potentiellement infinies.

Une premiè méthode consiste à produire des ensembles de préfixes de traces de plus en plus grands et à construire le graphe correspondant dans le domaine abstrait. Ceci peut aider à trouver un graphe abstrait intéressant.

Une autre méthode consiste à raisonner directement sur le domaine abstrait.

Le programme ci-dessous produit le graphe abstrait correspondant à une ensemble donné de traces.

```
%fg(n,g): if n is a positive integer, then g is the set of labeled arcs
%         of the graph abstraction of the a set of traces of length n
fg(N, Gamma) :- findall(Sarcs, tr(N, Sarcs), B), make_arcs(B, Gamma).

%make_arcs(b,g): if b is a list of bags, then g is the set containing
%                all elements in the bags.
```

**Domaine abstrait à deux nœuds**

On considère le domaine abstrait avec deux nœuds abstraits caractérisés par la propriété :

```
%Abstract graph states definition
gs(1) :- carry(_,_), !. %agent carry an object
gs(2).                  %no object carried
```

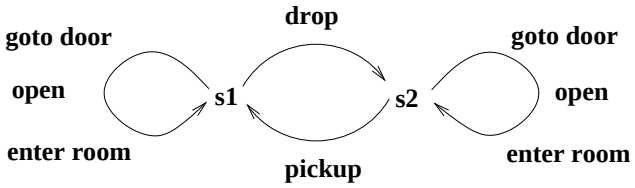


**Fig. 11.** A first graph with two states (some object carried or not)

Tous les états satisfont l'une ou l'autre de ces assertions. Le point fixe est obtenu dés avec des traces de taille 3.

Le graphe peut être construit sur la base d'un raisonnement direct sur l'existence ou non d'arc étiqueté entre deux états abstraits selon le type d'opération . Ici il est évident de remarquer que seules l'action sur un objet modifie l'tat abstrait, donc seules les actions *pickup* et **drop** sont en cause. Par contre toutes les autres actions ne modifient pas l'état.

Un autre exemple à deux états est le suivant.

```
%Abstract graph states definition
gs(1) :- inroom(_,r1), !.   %some agent in room r1
gs(2) :- inroom(_,r2), !.   %some agent in room r2
gs(u).                      %other state
```

Dans cette abstraction on ne s'intéresse qu'aux déplacements entre deux pièces particulières (celles dont la porte communicante est fermée). Le graphe obtenu avec 3 nœuds (figure 12) montre des états 1 et 2 (correspondant aux pièces r1 et r2) qui jouent des rles symétriques : on pourrait les considérer ensembles, et ne comparer ces deux pièces ensembles par rapport aux autres (en fait r3 qui est le seule pièce sans porte communiquante fermée).

**Domaine abstrait à trois nœuds**

Les trois propriétés suivantes caractérisent trois nœuds abstraits et une et une seule d'entre elles est satisfaites à un état quelconque. Le premier état abstrait correspond à avoir un agent au milieu d'une pièce, le second à à avoir un agent proche d'une porte close d'où il peut soit la franchir, soit revenir sur ses pas -noter que dans cet exemple il a la clef de la seule porte fermée, donc cette possibilité n'apparaî pas) et le troisième à franchir la porte (déjà ouverte).

```
%Abstract graph states definition
gs(1) :- agent(A), inroom(A,_), \+is_at_any_door(A), !.
                              %agent in the center of a room
gs(2) :- atdoor(_,D), closed(D), !.
                              %agent at some door in a room
gs(3) :- atdoor(_,D), is_open(D), ! %agent has to open a door
```

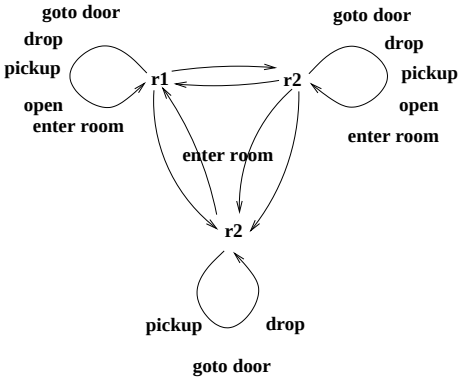


**Fig. 12.** A graph with three states to distinguish rooms

Le graphe abstrait obtenu (figure 13) montre par exemple clairement que la prise d'objet n'a lieu qu'au milieu d'un pièce.

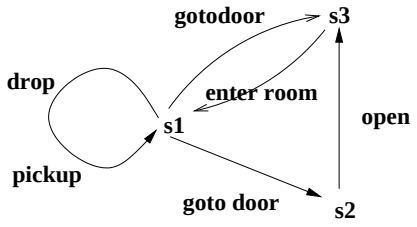


**Fig. 13.** A graph with three states (actions around doors)

**Domaine abstrait à six nœuds**

Enfin le dernier graphe abstrait revient à dupliquer le graphe précédent en distiguant les parcours de l'agent portant ou non un objet.

```
%Abstract graph states definition
gs(1) :- carry(_,_), agent(A), inroom(A,_), \+is_at_any_door(A), !.
gs(2) :- carry(_,_), atdoor(_,D), closed(D), !.
gs(3) :- carry(_,_), atdoor(_,D), is_open(D), !.
gs(4) :- agent(A), inroom(A,_), \+is_at_any_door(A), !.
gs(5) :- atdoor(_,D), closed(D), !.
gs(6) :- atdoor(_,D), is_open(D), !.
```

Cet exemple illustré par la figure 14 montre que le choix des états abstraits ne doit pas se faire sur la base de critères dichotomiques, ce qui entraîne in-

manquablement une explosion combinatoire des nœuds qui peut ne pas apporter d'information supplèmentaire. Le graphe précédent est plus synthétique et riche d'enseignements. Il montre en particulier que les actions de déplacement ne dépendent pas du fait de porter ou non un objet.

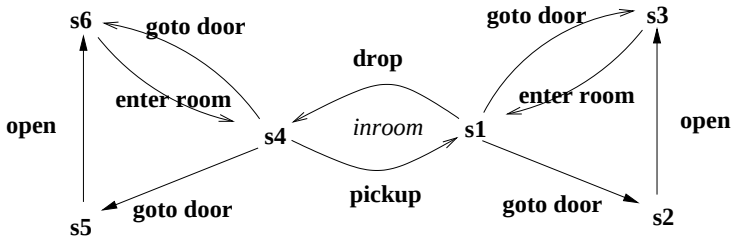


**Fig. 14.** A graph with six states (actions around doors and carried oject)

Ce graphe est également obtenu avec des traces de taille 3.

### 19.5 IS in Prolog style

The predicate find/2 simulates virtual trace generation from an actual trace. The above predicates presentation is simplified.

```
%find(at,vt): if s0 is the initial state and at a given finite
%portion of actual trace
%(trace events only), then vt is the corresponding sequence of
%virtual trace event.
%Uses the initialisation predicate: s0 which cleans the Prolog database
%and initialises it with the evolving fluents.

find(Tr,Tv) :- s0, interpr(Tr,Tv).

%interpr(+at,?vt): if at is a trace sample of length n, then vt is the
%corresponding reconstructed virtual trace.
%Remark: a current (initial) state is supposed given.

interpr([], []) :- !.
interpr([[Ch|AtE]|ATr], [[Ch,A,VtE]|VTr]) :-
                              rebuild(AtE, A), state(VtE), interpr(ATr,VTr).

rebuild([pickup,A,O,_], pickup) :- asserta(carry(A,O)).
rebuild([drop,A,O,_], drop)     :- retract(carry(A,O)).
rebuild([walk,A,D], gotodoor)   :- door(D), asserta(atdoor(A,D)).
rebuild([walk,A,R], enteroom)   :- room(R), inroom(A,RO), atdoor(A,D),
```

```
                                connect(R0,D,R),
                            retract(inroom(A,R0)), retract(atdoor(A,D)),
                            asserta(inroom(A,R)),
                            (carry(A,O) -> (retract(inroom(O,R0)),
                                            asserta(inroom(O,R))) ; true).
rebuild([open,_,D], open)       :- retract(closed(D)).

%Example

| ?- find([[1,pickup,a1,o1,r1],[2,walk,a1,d12],[3,open,a1,d12],
[4,walk,a1,r2],[5,drop,a1,o1,r2],[6,pickup,a1,o1,r2],[7,drop,a1,o1,r2],
[8,walk,a1,d12],[9,walk,a1,r1],[10,walk,a1,d13]],Vt).

Vt = [
[1,pickup,[inroom(a1,r1),carry(a1,o1),closed(d12),inroom(o1,r1)]],
[2,gotodoor,[inroom(a1,r1),carry(a1,o1),atdoor(a1,d12),closed(d12),inroom(o1,r1)]],
[3,open,[inroom(a1,r1),carry(a1,o1),atdoor(a1,d12),inroom(o1,r1)]],
[4,enteroom,[inroom(a1,r2),carry(a1,o1),inroom(o1,r2)]],
[5,drop,[inroom(a1,r2),carry(a1,o1),inroom(o1,r2)]],
[6,pickup,[inroom(a1,r2),carry(a1,o1),inroom(o1,r2)]],
[7,drop,[inroom(a1,r2),carry(a1,o1),inroom(o1,r2)]],
[8,gotodoor,[inroom(a1,r2),carry(a1,o1),atdoor(a1,d12),inroom(o1,r2)]],
[9,enteroom,[inroom(a1,r1),carry(a1,o1),inroom(o1,r1)]],
[10,gotodoor,[inroom(a1,r1),carry(a1,o1),atdoor(a1,d13),inroom(o1,r1)]]] ? ;
(10 ms) no
```

### 19.6 OS in Flux style

The OS in prolog style suffers of the following drawbacks : necessity to handle explicitely the state updates and uneficiency.

The fluent calculus is an alternative.

An implementation in Flux based on CHR [47] in SICSTUS has been realized by Rafael Ferreira Oliveira. Here is a comparative exceptr (only a part of the code is shown in Prolog followed by flux representation).

States are similarely described.

Main loop :

```
so(Ch, [E|T]):-
        tr_extract(Ch, E),
        Ch1 is Ch+1, so(Ch1, T).

main_loop(Z, Ch, [E|T]) :-
poss(A,Z), state_update(Z,A,Z1), extraction(Z1,A,Ch,ET),
Ch1 is Ch+1, main_loop(Z1, Ch1, T)).
```

The state is implicitely handled in Prolog. There is a separation in Flux between the condition of action (expressed by the predicate `poss` and the transition itself.

Extraction function for the **open** event.

```
tr_extract(Ch, [Ch,open,A,D]) :- trans(Ch, open(A,D)).

extraction(Z, open(A,D), Ch, [Ch,open,A,D]).
```

Transition step for "open" event.

```
trans(Ch, open(A,D)) :-
 agent(A),
 at_door(A,D), closed(D), has_key_code(A,D),
 retract(closed(D)).


poss(open(A, D),Z) :-
 holds(at_door(A,D),Z), holds(closed(D),Z), holds(has_key_code(A,D),Z).

state_update(Z1, open(A, D), Z2) :-
 holds(closed(D),Z1), update(Z1,[],[closed(D)],Z2).
```

In Prolog the state database is updated explicitely, so the way to open a door is just to retract the fact it is closed. Also the typing here is explicit ("agent(A)"). In Flux, the condition of action (**ACond**) is given in the declarations `poss`. Both codes are similar. `update` is an operation of Flux.

Differences : use explicit of requests, expressiveness : holds, not_holds, not_holds_all. (to be developed)