Un mécanisme de production de conseils exploitant les relations de composition et de précédence dans un arbre de tâches

François Pachet, Jean-Yves Djamen, Claude Frasson, Marc Kaltenbach

Abstract
ANG:advisor systems, intelligent tutorial systems, task graph, part-whole relation, temporal precedence relation, trace analysis, plan recognition.

Résumé
Cet article décrit un mécanisme permettant d'exploiter de manière systématique un arbre des tâches, pour la production de conseils pertinents donnés à des utilisateurs d'appareils dans un contexte de formation. Ce mécanisme exploite deux liens privilégiés entre tâches : le lien de composition, liant une tâche à ses sous-tâches, et le lien de précédence, représentant les contraintes de précédence temporelle entre sous-tâches. Nous illustrons notre mécanisme sur des exemples extraits d'un système d'apprentissage d'une pompe à infusion utilisée dans les unités de soins intensifs.

Un mécanisme de production de conseils exploitant les relations de composition et de précédence dans un arbre de tâches

François Pachet* – Jean-Yves Djamen – Claude Frasson** – Marc Kaltenbach*****

** Laforia-IBP - Université Pierre & Marie Curie*
Boîte 169, 4, place Jussieu, 75252 Paris Cedex 05

*** Université de Montréal - Département IRO, Laboratoire HERON*
C. P. 6128 Succ. Centre-Ville, Montréal, Québec, H3C 3J7, Canada

**** Bishop's University - Lennoxville*
Québec, J1M 1Z7, Canada

RÉSUMÉ. *Cet article décrit un mécanisme permettant d'exploiter de manière systématique un arbre des tâches, pour la production de conseils pertinents donnés à des utilisateurs d'appareils dans un contexte de formation. Ce mécanisme exploite deux liens privilégiés entre tâches : le lien de composition, liant une tâche à ses sous-tâches, et le lien de précédence, représentant les contraintes de précédence temporelle entre sous-tâches. Nous illustrons notre mécanisme sur des exemples extraits d'un système d'apprentissage d'une pompe à infusion utilisée dans les unités de soins intensifs.*

ABSTRACT. *This paper describes a mechanism that produces relevant advice by a systematic exploitation of a task tree. The mechanism exploits two relations : part-whole relation between tasks and sub-tasks, and temporal precedence relations between sibling tasks. We illustrate our mechanism with examples drawn from a tutorial system for an infusion pump used in intensive care units.*

MOTS-CLÉS : *systèmes conseillers, systèmes tutoriels intelligents, graphe des tâches, relation de composition, relation de précédence temporelle, diagnostic de trace, reconnaissance de plans d'actions.*
KEY WORDS : *advisor systems, intelligent tutorial systems, task graph, part-whole relation, temporal precedence relation, trace analysis, plan recognition.*

1. Introduction

Ce travail s'inscrit dans le cadre de la modélisation de l'usager dans un système tutoriel intelligent (STI), appelé SAFARI (Système d'aide à la formation par raisonnement interactif). Le problème que nous cherchons à résoudre est celui de la production de conseils "pertinents" [1] à un apprenant pendant qu'il résout un problème posé par le système. Nous nous attachons ici à étudier une forme particulière de conseils : ceux exploitant les relations de composition et de précédence entre tâches. D'autre part, nous nous intéressons uniquement à la génération automatique de conseils à partir d'analyse dynamique de traces utilisateur, excluant toute forme sophistiquée d'interaction entre le système de production de conseil et l'apprenant, comme dans le système WHY [Stevens 1977] par exemple.

La production automatique de conseils pertinents par un système informatique nécessite une reconnaissance, même sommaire, des intentions de l'utilisateur. Le problème de la reconnaissance des intentions [Cohen 1990] a fait l'objet de nombreux travaux dans la communauté d'intelligence artificielle, et se trouve à la croisée de diverses disciplines : systèmes tutoriels, langage naturel, compréhension d'histoires, modélisation de l'apprenant, systèmes multi-agents. En particulier, le problème de la reconnaissance de plans d'actions (*plan recognition*) suscite de nombreux efforts multidisciplinaires (voir par exemple [Allen 1990], ou le récent workshop IJCAI'95 sur *The Next Generation of Plan Recognition Systems : Challenges for and Insight from Related Areas of AI* à ce sujet).

S'il est aujourd'hui difficile de synthétiser tous les efforts de recherche faits dans ce domaine, on peut tout de même citer plusieurs approches remarquables du problème dans sa généralité, ayant chacune donné lieu à de nombreuses expérimentations. Les approches à base de grammaire et d'analyse syntaxique [Hoppe 1988 ; Sidner 1981] sont, historiquement, les premières à avoir été proposées (voir par exemple la comparaison intéressante de plusieurs techniques à base de grammaires par Desmarais, 1993). L'approche logique de [Kautz 1986], basée sur la circonscription, résout le problème dans sa généralité, mais nécessite une connaissance préalable de la librairie complète des plans d'actions possibles, et son algorithmique n'est pas complète [Kautz 1990]. D'autres approches, comme celle de Carberry (1990) ou celle de McKendree (1988), utilisent des modèles plus empiriques pour permettre la reconnaissance de plans incorporant une gestion de l'incertain.

De manière générale, les approches citées ci-dessus relèvent du principe de recouvrement ou *overlay* [Carr 1977]. Dans ce contexte, on considère le modèle de connaissances de l'apprenant comme un *sous-ensemble* de celui de l'expert. Dans l'*overlay* et dans un contexte de diagnostic interactif [Wenger 1987], les réponses ou les expérimentations de l'apprenant forment une *trace* plus ou moins sophistiquée, pouvant aller jusqu'à un véritable graphe de ses intentions [Djamen 1994]. Les réponses fournies par l'expert pour conduire les sessions tutorielles forment un graphe, appelé le plus souvent *graphe des tâches*, contenant toutes les solutions relatives au problème posé. Des variantes de l'approche *overlay* ont été proposées dans différents

[1] Nous reviendrons sur la définition de ce terme dans notre contexte.

travaux, pour représenter aussi les connaissances manquantes ou erronées (voir par exemple les bibliothèques d'erreurs [Brown 1980] ou les graphes génétiques [Goldstein 1982]). La trace est alors comparée au graphe des tâches, à l'aide d'un algorithme de filtrage plus ou moins complexe.

Les résultats que l'on peut attendre de cet algorithme de filtrage dépendent essentiellement de la représentation choisie pour le graphe des tâches. Dans Sherlock [Lesgold 1992] par exemple, le graphe des tâches (appelé *espace problème*) contient des nœuds qui sont des solutions partielles. Les arcs représentent différents passages d'une solution partielle à une autre, plus avancée. Plus précisément, l'espace problème est un treillis de sous-buts reflétant les multiples chemins vers la solution du problème. Ainsi, en partant du nœud racine, les liens désignent les tâches que l'usager doit effectuer, en mode "OU" implicite. Des représentations similaires ont été utilisées dans des systèmes tels que DARN [Mittal 1988] (utilisation de plans d'actions pour aider un technicien à réparer des machines) et ACE [Sleeman 1979] (compréhension de la démarche d'un apprenant en algèbre).

2. Contre les graphes de tâches

D'une manière générale, plusieurs problèmes se posent quand on choisit de représenter les graphes de tâches par des graphes quelconques. Voici à notre avis les plus importants d'entre eux.

Relation de composition entre tâches et niveaux d'abstraction

Très peu de systèmes prennent en compte les différents niveaux d'abstraction des tâches. Le niveau de connaissance prôné par Ernst et Newell [Ernst 1969] et l'abstraction décrite par Goldstein [Goldstein 1982] sont quelques exceptions. Cependant, dans la plupart des cas, les formalismes décrits permettent uniquement à l'auteur de décomposer son cours. Cette décomposition n'est pas exploitée pour fournir des conseils qui feraient référence aux bons niveaux d'abstraction des tâches, et non simplement aux actions atomiques.

Complexité calculatoire

Les algorithmes de *pattern-matching* sous-jacents aux approches par graphes (conceptuels ou réseaux sémantiques) ont une complexité élevée [Niem 1993 ; Sowa 1984]. Certains algorithmes de jointure ou de généralisation sont même parfois irréalisables en pratique.

Trace difficile à synthétiser

Avec le *pattern-matching* vient naturellement le problème de la synthèse de la trace, et de l'élimination des "bruits" (séquences d'actions non significatives). Ce problème est lui aussi algorithmiquement complexe, et entraîne le plus souvent la mise en œuvre de solutions ad hoc.

Relations de précédence non binaires

Les représentations à base de graphes ne permettent de représenter que des relations binaires entre concepts (tâches, actions ou autres). Or les relations binaires ne constituent qu'une petite partie des relations de précédence possibles et intéressantes en pratique. Il est par exemple difficile de représenter deux tâches quelconques (A et B) qui peuvent être effectuées dans n'importe quel ordre, ou bien les relations du type "2 tâches parmi (A, B, C)", dans n'importe quel ordre, etc. Ces relations entraînent un accroissement exponentiel du nombre de nœuds du graphe, et rendent les algorithmes difficile à maintenir et à modifier. Les liens génétiques de généralisation/spécialisation, de raffinement/simplification, d'analogie, de déviation/correction [Goldstein 1982], ne permettent pas non plus de représenter ce type de contraintes.

Résultats difficiles à exploiter

Le *pattern-matching* entre les graphes produit des résultats difficiles à exploiter, en particulier dans les cas d'échec. Ceci est essentiellement dû au fait que le résultat du filtrage est une liste de sous-graphes non nécessairement connexes (voir par exemple les algorithmes décrits dans [Ellis 1993]).

Conseils idiosyncratiques

Il est difficile d'incorporer dans les mécanismes de *pattern-matching* des conseils idiosyncratiques, spécifiques au domaine, et qui portent sur des informations non représentées dans le graphe des tâches. Par exemple, dans les problèmes décrits ci-dessous (pompe à infusion) on voudra incorporer des connaissances expertes portant sur le temps mis pour enchaîner des sous-tâches. Plus généralement, les algorithmes de *pattern-matching* sont des "boîtes noires" qu'il est difficile d'adapter.

Problème de l'identité des tâches

Les graphes peuvent être des structures idéales pour cacher certains problèmes ontologiques délicats. Typiquement la notion d'identité entre deux tâches "similaires" n'est pas claire. Le fait d'avoir un nœud ayant plusieurs prédécesseurs peut, en effet, représenter deux réalités différentes :

– Identité structurelle

La tâche doit être exécutée plusieurs fois, mais à chaque fois il s'agit de la "même" tâche. Par exemple dans un des problèmes que nous voulons résoudre (cf. figure 2), il faut à un moment donné commencer la tâche [Open the door], puis faire quelques actions, dont fermer la porte [close the pump door], puis re-ouvrir la porte [Open the door]. Dans ce cas, les deux tâches [Open the door] sont évidemment distinctes – il faut bien ouvrir la porte "deux fois" – bien que structurellement identiques, car définies par la même séquence d'actions.

– Identité physique

En revanche, une tâche peut apparaître plusieurs fois dans le graphe des tâches, avec comme sous-entendu (non représenté) qu'il s'agit bien de la même tâche, au

sens physique du terme (de la même "instance"). En particulier, si elle est effectuée une fois, elle est effectuée pour toutes ces occurrences !

Nous proposons pour ce problème une solution décrite en 4.1.3.

3. Notre approche

Nous sommes convaincus que le problème de la reconnaissance de plans d'actions d'un utilisateur est un problème insoluble dans sa généralité. Nous proposons ici un mécanisme général qui combine une analyse, rudimentaire, des séquences d'actions de l'utilisateur, avec un mécanisme de production de conseils pertinents et systématiques.

Notre approche, résolument empirique, s'inspire directement des travaux sur les *systèmes épiphytes* [Pachet 1994 ; Paquette 1995]. Dans les architectures épiphytes, on considère les modules conseillers comme des extensions non perturbatrices d'applications informatiques, capables de s'adapter à leur comportement. Le terme épiphyte, issu de la botanique, désigne en effet une classe de plantes qui vivent en se greffant sur des plantes "hôtes" sans leur porter préjudice (par opposition aux parasites et aux prédateurs). Le lierre et certaines orchidées sont des plantes épiphytes. Par analogie, les systèmes épiphytes sont des systèmes informatiques chargés d'observer et analyser l'activité d'un système informatique hôte. Leur structure est décrite par un arbre isomorphe à un treillis de tâches. La génération des analyses et conseils se produit alors par un mécanisme de remontée d'informations des feuilles vers la racine [Pachet 1994]. De manière analogue, les travaux de [Quast 1993] sont basés sur un mécanisme d'interprétation du graphe des tâches par remontée des feuilles vers la racine, à partir des actions de l'utilisateur.

Nous présentons ici une architecture permettant de concevoir et construire des systèmes conseillers capables de produire des conseils "pertinents" exploitant les relations de composition (liens entre tâches et sous-tâches) et de précédence temporelle (liens entre sous-tâches d'une même tâche) de cet arbre de tâches. L'architecture que nous présentons ici repose sur la double hypothèse suivante :

– L'espace problème peut être décrit par un *arbre* de tâches. Comme dans les approches classiques à base d'*overlay*, cet arbre décrit l'ensemble des solutions possibles typiques pour un problème donné. Cet arbre est en général produit par une analyse cognitive des tâches [Lesgold 1990]. Nous faisons de plus l'hypothèse que cet arbre des tâches est d'une complexité raisonnable. Il nous est fourni par les cogniticiens, validé et prêt à l'emploi.

– Nous posons comme principe que la production de conseils pertinents peut être systématisée, et représentée par un parcours adéquat de cet arbre, déclenché à chaque action de l'utilisateur. Bien sûr, la notion de pertinence n'a pas de sens en soi, mais seulement au sein d'un ensemble de tâches donné. Par pertinent, nous entendons donc ici "cohérent par rapport à un ensemble d'arbres de tâches".

De plus, cette architecture est conçue pour permettre la prise en compte de conseils idiosyncratiques, liés au domaine, et non directement représentés dans l'arbre des tâches par les deux relations de base. Cette architecture sert en outre de support à l'élaboration d'une base structurée d'informations contenant des informations sur les tâches effectuées, à faire, ou en cours, et ce, à tous les niveaux d'abstraction de l'arbre des tâches. Cette base d'informations peut à son tour alimenter un modèle de l'apprenant [VanLehn 1988]. Nous proposons aussi une typologie des conseils produits, permettant de les analyser *a posteriori*, de les classer ou de les trier, en fonction des utilisations du système. Enfin, notre modèle peut servir de support à la génération dynamique de textes tutoriels, adaptés à l'état courant de l'utilisateur.

Tous les exemples d'illustrations pris dans la suite de ce document portent sur la pompe à infusion Baxter [Baxter 1992], considérée comme domaine typique d'application du projet SAFARI. Cette pompe sert à administrer des perfusions aux personnes hospitalisées dans les unités de soin intensif, comme celui du Montreal General Hospital. Les tâches typiques à effectuer sur cet appareil sont les infusions primaires ou la détection de bulles d'air dans la pompe.

Le reste du document est structuré comme suit. Dans la section 4, nous présentons la structure de l'arbre des tâches, suivant les relations de composition et de précédence, en l'illustrant avec un arbre tiré de l'utilisation de la pompe Baxter [Sandrasegaran 1994]. Dans la section 5, nous donnons quelques exemples de conseils pertinents produits à partir de traces "illégales", ainsi que de conseils spécifiques issus de l'analyse cognitive des tâches. La section 6 décrit la notion d'état pour les tâches. Dans les sections 7 à 12, nous décrivons notre architecture, avec en particulier la gestion des états pour les tâches, et introduisons la notion de "classe de comportement" permettant d'exprimer des relations de précédence quelconques. Ces structures sont exploitées par un double mécanisme de parcours de l'arbre des tâches permettant de produire les conseils. Dans la section 13, nous proposons une liste d'applications possibles et discutons notre modèle.

4. Structure opérationnelle et arbre des tâches

La structure opérationnelle dans le projet SAFARI permet de représenter les différentes solutions possibles de l'auteur à un problème donné. Cette structure contient des informations diverses, comme les hypothèses formulables par l'usager, les buts à atteindre, et les observations qu'il est supposé faire pendant la résolution du problème posé ([Djamen 1995)]. Le système SAFARI propose plusieurs modes d'apprentissage à l'apprenant parmi : observation libre, démonstration, résolution de problèmes et cours complets. Le module conseiller que nous présentons ici intervient dans le cadre du mode "résolution de problèmes", pendant lequel un apprenant doit résoudre un problème donné sans intervention du système tutoriel. Le module

conseiller analyse les actions au fur et à mesure[2] et produit les conseils en exploitant une base d'arbres de tâches préétablie.

L'arbre des tâches sur lequel nous nous basons pour produire les conseils est un sous-ensemble de cette structure opérationnelle. Comme nous l'avons postulé, l'arbre des tâches, pour un problème donné, doit représenter l'ensemble complet des solutions possibles, c'est à dire des séquences d'actions "légales". Nous allons maintenant détailler la structure de cet arbre.

4.1. *Les tâches*

Nous proposons une représentation de l'arbre des tâches reposant sur deux relations privilégiées : la composition (lien entre tâches et sous-tâches) et la précédence (liens entre sous-tâches d'une même tâche).

4.1.1. *Composition*

Une tâche est une opération plus ou moins abstraite réalisée pour satisfaire un but. Par convention, les tâches sont représentées par des noms entre crochets [ ]. On distingue les tâches "abstraites", qui sont elles-mêmes décomposées en sous-tâches, des tâches "concrètes", primitives et donc indécomposables, que l'on appelle actions. Une action est représentée par une association outil/unité physique (cf.[Djamen 1993]). Par exemple, [Appuyer sur la touche Start], est une action représentée par l'association Main, "Touche-Start". En revanche, [Faire une infusion primaire] est une tâche (abstraite), qui se décompose en un ensemble partiellement ordonné de sous-tâches. Cette relation de composition induit naturellement une structure d'arbre, et est donc facilement représentable graphiquement. La figure 1 montre la décomposition de la tâche [Faire une infusion primaire], et la figure 2 la décomposition de la tâche permettant de résoudre le problème d'air dans la pompe Baxter.

4.1.2. *Précédence*

Les liens de précédence sont plus complexes car ils ne sont pas tous représentables par de simples relations binaires. Par exemple, la tâche [Faire une infusion primaire] doit s'effectuer en réalisant les trois sous-tâches [Appuyer sur la touche On-Off], [Entrer les taux], et [Appuyer sur Start] dans cet ordre strict et, implicitement, une seule fois ! En revanche, la tâche [Entrer les taux] doit s'effectuer en réalisant les deux sous-tâches [Entrer un volume] et [Entrer un débit], dans n'importe quel ordre. En réalité la contrainte est plus complexe : l'utilisateur peut revenir sur une des sous-tâches, et ainsi l'effectuer plusieurs fois. Nous prendrons cet aspect en compte ultérieurement. Il est clair que les relations de précédence peuvent devenir

---

[2] Les actions sont effectuées sur une simulation des appareils étudiés et non pas sur les appareils réels. Dans le projet SAFARI cette simulation est effectuée par le système VAPS (*Virtual Applications and Prototype Systems*) [VAPS 1993], intégré à l'environnement Smalltalk.

arbitrairement complexes : un des arbres que nous avons à traiter contient des relations du type "2 sur 3", dans n'importe quel ordre, etc.

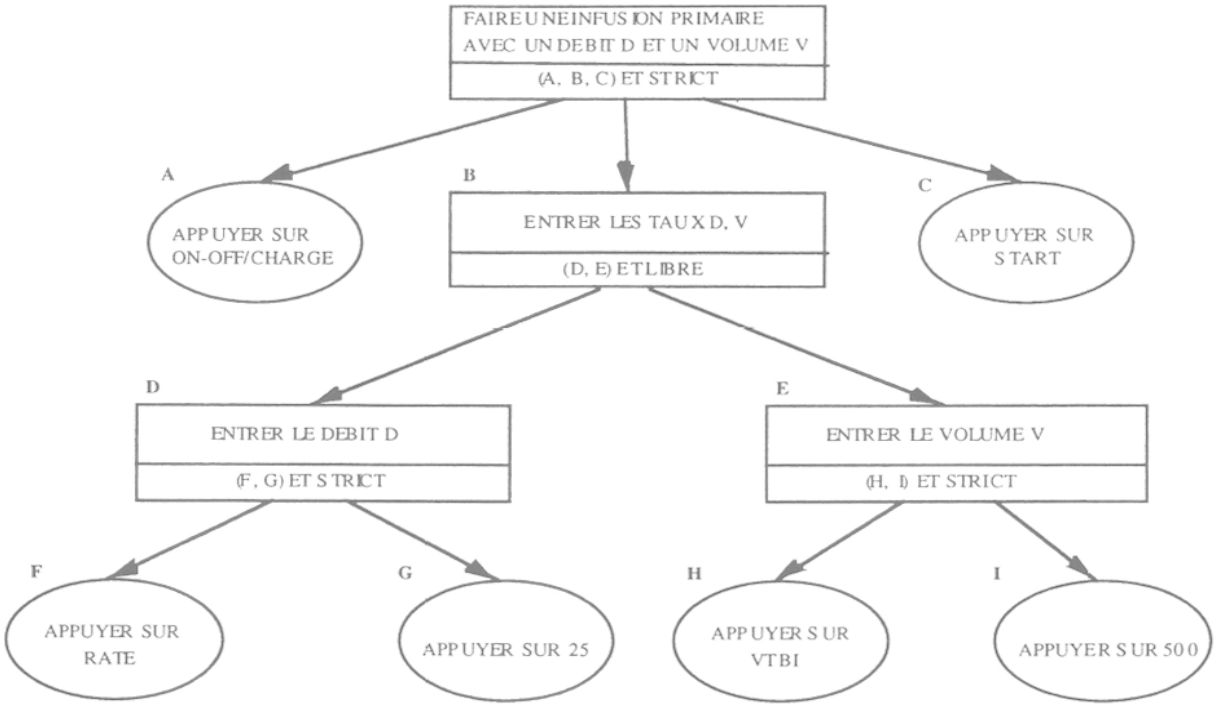

Figure 1. *Arbre des tâches pour une infusion primaire avec la pompe Baxter*

Au lieu d'essayer de trouver un formalisme pour représenter ces relations de manière graphique, nous introduisons la notion de *classe de comportement*, associée à chaque tâche. Une classe de comportement est une représentation abstraite de la relation de précédence existant entre les sous-tâches d'une tâche donnée. A chaque tâche de l'arbre des tâches est associée une classe de comportement et une seule. Nous reviendrons en détail sur la définition et la représentation de ces classes en section 7.

Figure 2. *Arbre des tâches pour le problème d'air. Pour des questions de visibilité, le dessin est un graphe, mais la structure sous-jacente est bien un arbre : les nœuds similaires sont dupliqués*

4.1.3. *Le problème de l'identité des tâches*

Comme solution au problème important de la représentation de l'identité des tâches (introduit en section 2), nous proposons de distinguer "classe" de tâches et "instances", instance ayant ici le sens "occurrence" dans le temps. Il est vital de pouvoir prendre en compte cette distinction, dans le cadre d'un système conseiller et de la construction d'un modèle de l'apprenant. Nous proposons ici de faire cette distinction en introduisant la notion de "ActionEvent", représentant une occurrence particulière d'une certaine "Action" (cf.section 10.1).

4.2. *Variables et problèmes*

Les tâches que nous représentons comportent des variables, instanciées au moment de la création des problèmes. Par exemple, nous avons représenté ici la tâche "Faire une infusion primaire (D, V)", où D et V sont des variables représentant res-

pectivement le débit d'infusion (en ml/sec), et le volume à infuser (en ml). Il est bien évident que les valeurs de ces variables sont déterminées le plus tard possible (en général à la génération du problème à résoudre), et que la structure de l'arbre est indépendante de ces valeurs. Il faut donc mettre en place un mécanisme permettant d'instancier ces valeurs pour un problème donné. Nous ne traiterons pas de ce problème ici, qui relève des travaux sur les curriculum. Nous reviendrons brièvement sur ce point avec le traitement particulier des tâches [Entrer une fraction (f)] en section 10.3.1.

5. Exemple de l'infusion primaire

L'observation du comportement de l'utilisateur a comme principal effet de produire une trace, qui n'est rien d'autre qu'une liste ordonnée d'actions. Tout le problème va consister à interpréter cette liste par rapport à notre arbre de tâches. Avant de présenter notre architecture, nous allons donner quelques exemples représentatifs de conseils pertinents liés à la comparaison entre une trace et l'arbre des tâches montré ci-dessus. Nous supposerons dans notre exemple que le problème à résoudre est [Faire une infusion primaire (25, 500)] (voir figure 1).

Il est bien entendu que le mécanisme que nous proposons n'a d'intérêt que pour les actions illégales de l'utilisateur. Lorsque la trace correspond en effet à un chemin possible dans l'arbre des tâches, aucun conseil n'est à donner a priori. Nous verrons par la suite comment insérer des conseils spécifiques, pouvant être produits y compris dans le cas d'actions légales. Comme cette insertion ne remet pas en cause notre mécanisme, nous ne nous intéressons ici qu'aux conseils liés aux actions illégales.

Noter que les deux traces légales sont, dans notre exemple, les suivantes (on utilisera des noms raccourcis quand il n'y a pas ambiguïté) :

- [On-Off], [Débit], [25], [Volume], [500], [Start]
- [On-Off], [Volume], [500], [Débit], [25], [Start]

On supposera que "500" et "25" sont des actions non décomposables : on ne se préoccupe pas du problème de l'interprétation des nombres à plusieurs chiffres ici (Cf. pour cela l'exemple complet incluant l'entrée des fractions en section 10.3.1).

5.1. *Exemples de traces illégales*

Nous donnons ici quelques exemples représentatifs de traces illégales, et les conseils pertinents correspondants.

- [On-Off], [Start]...

Dans ce cas, la réponse à donner est : "Vous devriez [Entrer les taux] avant de [Appuyer sur Start]". Cette réponse est au bon niveau d'abstraction (par rapport à l'arbre des tâches) : [Entrer les taux] n'est pas une action, mais bien une tâche abstraite.

- [On-Off], [Ouverture de la porte]...

Dans ce cas, la réponse à donner est : "Action inconnue dans le contexte [Faire une infusion primaire]. Vous devriez [Entrer les taux], après avoir appuyé sur On-Off". Cet exemple est à comparer au précédent, pour montrer que l'on peut exploiter l'information sur l'action erronée (ici inconnue, dans l'exemple précédent, connue, mais mal placée).

- [On-Off], [On-Off]...

Dans ce cas, la réponse à donner est : "Il faut appuyer un nombre impair de fois sur On-Off, car c'est un bouton de type Toggle". Ici, on voit qu'il faut donc être capable de mémoriser, dans chaque tâche, les actions déjà effectuées. De plus, on peut aller chercher de l'information dans la classe de l'objet.

- [On-Off], [500]...

Dans ce cas, la réponse à donner est : "Vous devriez [Appuyer sur VTBI] avant de rentrer le volume". Ici, le problème est d'arriver à reconnaître que 500 provient d'une "intention" de rentrer le volume, et donc de gérer correctement le "ou" de [Rentrer les taux].

- [500]...

Dans ce cas, la réponse à donner est : "Vous devriez [Appuyer sur On-Off] avant de faire quoi que ce soit".

•[On-Off], [Rate], [25], [Start].

Dans ce cas, la réponse à donner est : "Il faut terminer [Rentrer les taux], en effectuant [Rentrer un Volume]". Ici, on voit encore qu'il faut être capable de mémoriser, dans chaque tâche, les actions déjà effectuées.

• [On-Off], [Rate], [25], [Rate]...

Dans ce cas, la réponse à donner est : "Vous avez déjà fait [Rentrer un Débit]. Il faut maintenant [Rentrer un volume]". Même remarque que précédemment, mais à un niveau plus haut dans l'arbre.

• [On-Off], [100].

Dans ce cas, la réponse à donner est : "Vous devez [Rentrer un volume] OU [Rentrer un débit]. Pour ce faire, il vous faut [Appuyer sur Rate], soit [Appuyer sur Volume], avant de rentrer votre fraction". Ici le problème est plus complexe : il s'agit de "factoriser" les actions possibles, pour trouver un plus petit commun multiple.

• [On-Off], [VTBI], [500], [Rate], [25], [VTBI]...

Dans ce cas, il faut remarquer que la tâche [Entrer les Taux] a été correctement réalisée. Ainsi, l'action [VTBI] elle-même est déjà effectuée, la tâche [Entrer Volume] est déjà effectuée, et plus généralement la tâche [Entrer les taux] est déjà effectuée. On voit bien ici l'importance de la relation de composition, qui permet de produire des conseils à des niveaux d'abstraction différents !

5.2. *Détection de bulle d'air dans la pompe*

L'exemple de l'infusion primaire est simple, mais contient des informations assez riches pour que des conseils pertinents puissent être donnés. Pour montrer la validité de notre approche, nous donnons un autre exemple plus complexe illustrant des cas

qui n'apparaissent pas dans l'exemple de l'infusion primaire. Dans la figure 2, montrant la représentation graphique de cet exemple, trois problèmes sont susceptibles de se présenter pendant une infusion primaire. Le premier dénote l'absence du chargement de l'ensemble de la pompe (A). Le deuxième a trait au remplissage de la boîte contenant le liquide (B). Le troisième peut survenir du fait de la présence d'air dans le tube (C).

- remove I. V....

Il faut pouvoir détecter que [remove I. V.] a été effectué avec succès, mettant ainsi en cours les tâches [remove air], [remove air bubble], [look for air bubble], [find solution] et [air problem]. De plus, dans le cadre de [remove air], [remove I. V.] est prématuré ; dans le cadre de [air problem], [find solution] est prématurée. Les solutions possibles sont : "Vous devriez d'abord faire [remove pump set] (dans le cadre de [remove air])", "Vous devriez faire [press stop] avant de trouver la cause du problème".

- close set regulating...

Ici le problème est un peu plus compliqué. Il s'agit d'identifier la tâche qui doit être normalement en cours suite à l'activation de [close set regulating] (cf. A, B, C, D, E, I, W). La réponse à donner dépend de l'intention de l'apprenant. Nous reviendrons en détail sur ce point dans l'explication des classes d'actions.

6. La notion d'état pour les tâches

Comme nous l'avons déjà évoqué, nous avons besoin de gérer une notion d'état pour les tâches, afin de savoir à tout instant ce qui a été fait, ce qui est en cours, et ce qui n'a pas encore été commencé. Ces états nous permettent d'une part de calculer la liste des prochaines actions légales (cf. section précédente), et d'autre part de produire des conseils pertinents. En outre, la gestion des états que nous proposons ici permet de conserver l'historique des changements d'états, qui servira de base au modèle de l'apprenant. Nous proposons ici une représentation standard des états. On attribue aux tâches à effectuer un état pouvant prendre trois valeurs parmi : `termine`, `enCours`, ou `inactif`. Au début de la session, toutes les tâches et sous-tâches sont à l'état `inactif`.

La nouveauté de notre approche réside dans le mécanisme de changement d'état qui repose sur une exploitation systématique des relations de composition (de tâche à super-tâche) et de précédence. Au fur et à mesure que l'utilisateur agit, notre mécanisme va mettre à jour les états des tâches et super tâches concernées. Ces états vont permettre de produire les conseils pertinents que nous avons évoqués dans la section précédente.

6.1. *Représentation interne*

Nous représentons les états par de véritables objets (et non pas simplement des symboles), instances de la classe `Etat`. Cette classe définit simplement deux attributs :

- `type`

Représente l'état à proprement parler. C'est un symbole parmi `inactif`, `enCours` et `termine`

- `date`

La date à laquelle l'état est apparu (instance de la classe `Date`).

Par ailleurs, chaque tâche contient un attribut `etats` qui contient l'historique de tous ses états. Chaque changement d'état d'une tâche provoque l'empilement de l'ancienne valeur dans l'historique. Ainsi, deux méthodes principales (dans la classe `TacheAbstraite`) permettent de gérer les états des tâches :

- `etatCourant`

rend l'état courant de la tâche (le premier de la liste)

- `changeEtat(e)`

permet de changer l'état courant de la tâche, tout en mémorisant l'ancien.

7. Représentation des relations de précédence

7.1. *Des classes de comportement*

Nous introduisons maintenant la notion de classe de comportement, qui permet de représenter les relations de précédence entre sous-tâches d'une tâche donnée. Ces classes de comportement sont associées à chaque tâche de l'arbre des tâches. Elles permettent de spécifier une contrainte portant sur les sous-tâches directes de la tâche concernée.

Par exemple, dans le cas de l'infusion primaire, la tâche racine [Faire une infusion primaire], composée de trois sous-tâches ([Appuyer sur la touche On-Off], [Entrer les taux], et [Appuyer sur Start]) est associée à la classe de comportement appelée `Et-Strict`, pour signifier que les trois sous-tâches doivent être effectuées dans cet ordre strict. De même, nous associons la classe `Et-Strict` à la tâche [Faire une infusion primaire], ainsi qu'aux tâches [Entrer un volume] et [Entrer un débit]. En revanche, la classe `Et-Libre` est associée à la tâche [Entrer les taux], signifiant ainsi que les sous-tâches [Entrer Volume] et [Entrer Débit] peuvent être effectuées dans n'importe quel ordre. Nous garderons cet exemple volontairement simplifié ici, pour proposer différents scénarios de traces. Dans la représentation réelle d'une infusion primaire, l'usager a la possibilité d'entrer la durée de l'infusion en remplacement soit du débit, soit du volume (cf.la classe de comportement `2 tâches parmi 3`).

Enfin, dans l'exemple de détection de bulle d'air (Air Problem), la tâche [find solution] est associée à la classe de comportement `Ou-Exclusif`, signifiant

qu'une seule de ses trois sous-tâches doit être effectuée (et aussi qu'une seule des trois sous-tâches peut être en cours à la fois).

Notre postulat est que, pour une classe d'applications donnée, le nombre de classes de comportement "utiles" est limité. Dans le cas de l'apprentissage du fonctionnement de la pompe Baxter, 7 classes de comportement suffisent à décrire les scénarios les plus classiques. Le tableau 1 décrit ces classes de comportement. Noter que ces classes sont disponibles à partir de l'éditeur d'arbre des tâches (cf.section 10.3).

7.2. *Structure des classes de comportement*

Les classes de comportement vont avoir un rôle bien précis dans le système conseiller. Elles vont permettre d'une part de mettre à jour un état courant des actions et tâches effectuées par l'utilisateur. D'autre part, elles vont servir à produire des conseils pertinents, notamment lors de la violation des contraintes de précédence. Notre système est développé à l'aide d'un langage objet, Smalltalk-80. Dans ce contexte, chaque type de classe de comportement est représenté par une classe Smalltalk. Chaque tâche se verra associée une instance de la classe de comportement associée. De plus, chaque classe de comportement contient une variable d'instance `sousTachesEffectuees`, contenant la liste des sous-tâches (sous entendu de la tâche à laquelle elle est associée) déjà effectuées. Cette liste est mise à jour lorsque une sous-tâche est terminée (voir pour cela la notion de signal).

Nom de la classe	Exemple	Explication
Et-Libre	(A, B, C) [et libre]	A, B et C doivent être exécutées dans n'importe quel ordre
Et-Strict	(A, B, C) [et strict]	A, B et C doivent être exécutées dans l'ordre indiqué
Ou-Exclusif	(A, B) [ou exclusif]	soit A, soit B, mais pas les deux
Et-Optionnel-Strict	(A, B) [et optionnel strict]	A et B dans l'ordre indiqué, avec A optionnel
Et-Optionnel-Libre	(A, B) [et optionnel libre]	A et B dans n'importe quel ordre, avec A optionnel
N-Sur-M-Strict	(A, B...) [n/m strict]	n tâches parmi m dans l'ordre indiqué (n < m)
N-Sur-M-Libre	(A, B...) [n/m libre]	n tâches parmi m dans n'importe quel ordre (n < m)

Tableau 1. *La définition des classes de comportement standard*

7.3. *Calcul des prochaines actions et prochaines tâches*

Cette notion de classe de comportement nous permet déjà de calculer à tout instant la liste des prochaines tâches et actions "légales". C'est ce calcul qui permet de répondre à la question "Que puis-je faire ensuite ?". La réponse est une liste d'actions possibles. Si la liste est vide, c'est que le problème est terminé. Si elle est singleton, c'est qu'il n'y a pas de choix. Ce calcul se fait par un parcours en profondeur de l'arbre des tâches à partir de la racine ; il est représenté par la méthode `prochainesActionsPossibles` dans la classe `Tache`, définie comme suit :

(Fonction prochainesActionsPossibles dans la classe Tache)
prochainesActionsPossibles
SI étatCourant = #terminé
ALORS RIEN
SINON
"on délègue au comportement"
RETURN comportement . prochainesActionsPossibles

Comme on le voit, les tâches délèguent le calcul à leur comportement. Chaque classe de comportement est ainsi responsable de la définition de la méthode `prochainesActionsPossibles`. Voici par exemple ce que fait la classe `ComportementET` (les tâches doivent être toutes effectuées, dans n'importe quel ordre) :

(Fonction prochainesActionsPossibles dans la classe ComportementET)
prochainesActionsPossibles
SI une des sous-tâches (ST) est en cours
ALORS RETURN ST . prochainesActionsPossibles
SINON, RETURN la réunion (récursivement) de toutes les actions possibles pour les sous tâches non encore effectuées

Comme on le voit ici, cette méthode teste l'état des sous-tâches de la tâche, et s'appelle récursivement sur les sous-tâches en cours, ou bien, si il n'y en a pas, sur toutes les sous-tâches non effectuées (voir 7.4 pour la notion d'état). Cette méthode est redéfinie dans toutes les classes de comportement. Une méthode similaire permet de calculer la liste des prochaines tâches abstraites à effectuer (voir plus bas la méthode `prochainesTachesPossibles`).

7.4. *Gestion des changements d'état des tâches*

Comme nous l'avons dit, les classes de comportement ont aussi comme rôle de définir les règles de changement d'état. Ceci est réalisé par une méthode `estReussie`, définie dans chaque classe de comportement, et qui retourne vrai lorsque la

tâche est jugée terminée. Par exemple, voici ce que certaines classes définissent pour cette méthode :

(Fonction estReussie pour la classe ComportementETSuccesseur)
estReussie
RETURN vrai si la taille des sous tâches effectuées = taille des sous tâches

(Fonction estReussie pour la classe ComportementET)
estReussie
SI sousTachesEffectuees isEmpty ALORS RETURN false.
SINON RETURN(sousTachesEffectuees . last = tache . sousTaches. last)

(Fonction estReussie pour la classe ComportementOUExclusif)
estReussie
^sousTachesEffectuees . size = 1

7.5. *Définir une nouvelle classe de comportement*

Du point de vue de la représentation interne, les classes de comportement sont représentées par des classes Smalltalk, sous-classes de la classe abstraite `ComportementTache`. Pour introduire une nouvelle classe de comportement, il faut donc définir une sous-classe de `ComportementTache`, et y définir cinq méthodes obligatoires (voir la section 8.8 à ce sujet). Ce procédé manuel sera remplacé plus tard par un éditeur de classes de comportement générant le code correspondant automatiquement à partir de spécifications plus formelles.

8. L'algorithme général de parcours

Une fois les notions d'état et de classe de comportement introduites, nous allons maintenant décrire le mécanisme de parcours de l'arbre des tâches permettant de produire les conseils.

8.1. *Idée générale*

Notre processus d'analyse des actions de l'utilisateur repose sur un double mécanisme de parcours de l'arbre des tâches, exploitant essentiellement la relation de composition. Il s'agit donc en fait de parcours d'arbres (sauf pour ce qui concerne les actions ambiguës, cf. plus bas la notion de classe d'actions). Le principe de base consiste à analyser les actions une par une, par un parcours de l'arbre des tâches des feuilles vers la racine (bottom-up). Pour mettre en œuvre ce parcours, on introduit la notion de signal, représentant une information qu'une tâche transmet à sa super-tâche.

Plus précisément, le parcours de l'arbre des tâches est représenté par une suite de transmissions de signaux, du bas vers le haut. Conformément au principe de base de l'architecture épiphyte [Paquette 1995], chaque réception de signal par un nœud (tâche ou action) provoque systématiquement l'exécution d'une procédure (`accepteSignal`), qui elle-même se décompose en trois sous procédures, appelées dans cet ordre strict :

- Production de conseil au niveau du nœud traversé, en fonction du signal reçu,
- Changement éventuel de l'état de la tâche traversée en fonction du signal reçu,
- Contrôle de la remontée, par la production d'un signal vers la super-tâche en fonction du signal reçu.

8.2. *Boucle principale*

A chaque détection d'action de l'utilisateur, la procédure `accepteAction`, définie dans la classe `ArbreDesTaches`, est appelée. On distingue alors deux cas possibles, chacun se divisant en deux sous cas :

- action légale.

Si l'action détectée est légale (i.e. appartient à la liste des prochaines actions possibles, voir pour cela la section 5.2), alors elle est interprétée comme légale. Ceci relève en fait d'un postulat important : si une action peut être interprétée comme légale, on suppose que l'intention de l'utilisateur est la bonne !

Il y a alors deux sous-cas possibles, selon que l'action est ambiguë ou non. Une action est ambiguë s'il existe plusieurs actions possibles qui ont la même classe d'action (cf. section 10.1). Si l'action est non ambiguë, alors il y a transmission d'un signal de succès à cette action. Ce signal va lui-même provoquer une remontée d'autres signaux, récursivement, vers le haut de l'arbre.

Si l'action est ambiguë, le système pose alors explicitement la question à l'apprenant, pour savoir quelle est l'action de l'arbre des tâches qu'il convient d'exploiter[3]. La question est posée à l'aide d'un menu créé dynamiquement, qui propose toutes les possibilités. Une fois la réponse obtenue, le même signal que précédemment est construit et envoyé à l'action. La nature, le traitement, et l'effet du signal sont décrits dans la section suivante. Ils ont pour effet de mettre à jour les états des actions et tâches concernées, et de produire éventuellement des conseils.

- action illégale.

Une action est illégale si aucune action correspondante de l'arbre des tâches n'est possible au moment où elle est détectée. Dans ce cas, deux sous-cas se présentent, similaires aux sous-cas précédents : si l'action illégale est ambiguë (i.e. plusieurs actions de l'arbre des tâches partagent sa classe d'action), alors le système demande à l'utilisateur son "intention", (toujours à l'aide d'un menu dynamique) pour déterminer quelle action doit être exploitée. Si elle n'est pas ambiguë, un signal est construit

---

[3] Nous comptons coupler ce mécanisme avec la notation PIF_g [Djamen 1994] à l'aide de laquelle les intentions de l'apprenant sont connues d'avance. Cette extension nous permettrait de limiter le nombre de questions nécessaires à l'analyse.

(portant la mention `illégal`) et envoyé à l'action. De même, l'effet de ce signal est décrit dans la section suivante. Il aura pour effet de produire des conseils, mais sans mettre à jour les états des tâches.

8.3. *La notion de signal, transmission des signaux*

Le mécanisme de parcours de l'arbre des tâches repose sur la notion de signal, couplée à un parcours des feuilles (actions) vers la racine. Quand une action utilisateur (`ActionEvent`) est détectée, un signal lui est envoyé.

Chaque signal est composé des informations suivantes :

- l'émetteur du signal. La tâche ou l'action qui a fabriqué le signal,
- l'heure d'envoi,
- le type du signal. Ce type correspond en première approximation à l'état de la tâche qui envoie le signal (`enCours` ou `termine`),
- un Booléen `legal`, vrai si le signal provient d'une action légale, faux sinon.

8.4. *Traitement du signal*

Le traitement du signal par une tâche ou une action suit systématiquement le même schéma. La procédure principale est `accepteSignal(s)`, qui se décompose comme nous l'avons vu en trois sous-procédures :

- `produitConseilAPartirDe(s)`. Produit des conseils en fonction du signal reçu,
- `changeEtatAPartirDe(s)` . Change l'état de la tâche (en fait de son comportement) en fonction du signal reçu. Noter que ce changement d'état n'est effectué que si le signal est légal,
- `remonteSignalAPartirDe(s)`. Fabrique un nouveau signal pour la super-tâche, en fonction du signal reçu.

La méthode `accepteSignal(s)` est donc définie comme suit dans la classe `TacheAbstraite` :

```
(Méthode accepteSignal dans la classe TacheAbstraite)
accepteSignal (s)
  self . produitConseilAPartirDe (s)
  SI s . legal ALORS self . changeEtatAPartirDe(s).
  self . remonteSignalAPartirDe (s)
```

Chacune de ces procédures est à son tour définie comme suit, en déléguant le plus gros du travail aux comportements.

8.5. *Production des conseils*

Pour les actions, la production de conseil consiste simplement à dire que l'action a été effectuée avec succès :

```
(Classe Action)
produitConseilAPartirDe(s)
      self . ajouteConseil(#Succes).
```

Pour les tâches, la production des conseils est effectuée essentiellement par la classe de comportement :

```
(Classe TacheAbstraite)
produitConseilAPartirDe (s)
  comportement . produitConseilAPartirDe(s).
  self . produitConseilSpecifiqueAPartirDe(s)
```

A titre d'exemple, nous donnons ici la production de conseil pour la classe de comportement `Et-Strict`. Cette méthode prend en compte toutes les combinaisons possibles d'état des sous-tâches en fonction du type de signal reçu.

```
(Classe ComportementET)
produitConseilAPartirDe(signal)
 st := signal . emetteur.
 SI signal type = #noChange ALORS type := st etatCourant type
                                  SINON type := signal . type.
 SI (sousTachesEffectuees includes(st) ALORS
   SI type = #termine ALORS RETURN (self . ajouteConseil(#Deja,
      'tache ' , st.bracketName , ' deja effectuee') FINSI
    SI type = #enCours ALORS (self . ajouteConseil(#EnCoursDeja,
      'tache ' , st.bracketName , ' en cours mais deja effectuee') FINSI FINSI
"Puis on detecte si une autre sous-tache est deja terminee"
 already := premiere tache non terminee.
 SI already non nil ALORS RETURN self . ajouteConseil
(#EnCoursNonTermine,
      ('Une seule tâche à faire pour ',tache bracketName,' Or
',already.bracketName, ' deja effectuee')) FINSI.
"On detecte ensuite si une autre sous-tâche est deja en cours"
 already := premiere tache en cours.
 SI already non nil ALORS RETURN (self. ajouteConseil
(#EnCoursNonTermine,
      ('Une seule tâche en cours à la fois : ',already . bracketName,'
          deja en cours et non terminee'))).
```

8.6. *Changement d'état*

Pour une action, le changement d'état est systématique, en fonction du type du signal :

changeEtatFrom(s)
self.changeStateTo (s.type, s.heure)

Pour une tâche, le changement d'état consiste à demander au comportement de faire le travail.

changeEtatAPartirDe(s)
comportement changeEtatAPartirDe(s)
newState := comportement nouvelEtatAPartirDe(s)
SI newState <> #noChange ALORS (self . changeEtat(newState)) FINSI

Enfin, le changement d'état pour le comportement consiste toujours à mettre à jour la liste des sous-tâches effectuées :

(Classe ComportementTache)
changeEtatAPartirDe(s)
SI s . type = #termine ALORS
SI non(sousTachesEffectuees . includes : s) ALORS
(sousTachesEffectuees . add : (s.emetteur)) FINSI FINSI

8.7. *Remontée du signal*

Pour une action, la remontée du signal consiste à fabriquer un signal de type `succès` et à le transmettre à la super tâche. Pour une tâche, la remontée du signal consiste à fabriquer un signal dont le type est déterminé par le comportement (via la procédure `nouvelEtatAPartirDe`).

On peut voir sur la figure 3 la remontée d'un signal dans l'arbre des tâches de l'infusion primaire.

8.8. *Résumé des responsabilités des classes de comportement*

Comme nous l'avons vu, les classes de comportement portent une responsabilité essentielle dans ce mécanisme. Voici en résumé, la liste complète des méthodes définies dans les classes de comportement.

Toute nouvelle classe de comportement doit absolument redéfinir ces cinq méthodes (et ces cinq seulement[4]) :

- `prochainesTâchesPossibles`, calcule la liste des prochaines sous-tâches à effectuer ;
- `prochainesActionsPossibles`, calcule la liste des prochaines actions à effectuer ;
- `estReussi`, rend un Booléen, vrai si la tâche est terminée ;
- `produitTexte`, rend un texte décrivant comment effectuer la tâche ;
- `produitConseilAPartirDe`, fabrique un ou plusieurs conseils en fonction d'un signal reçu par une sous-tâche.

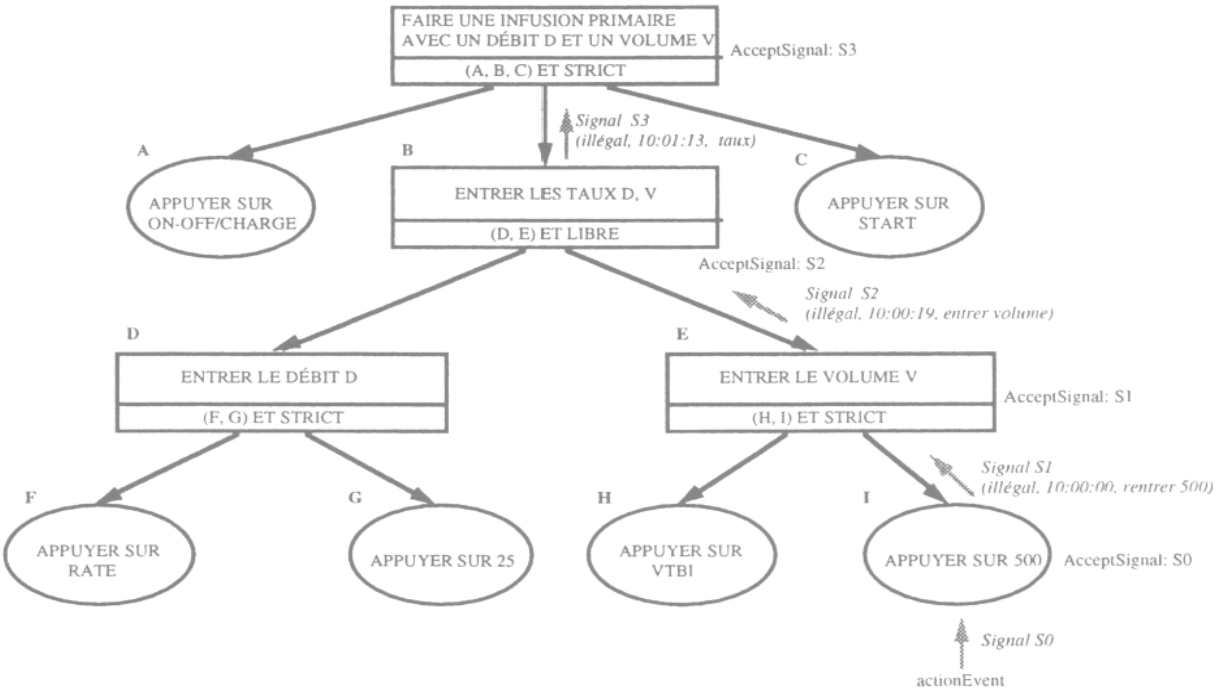

Figure 3. *Remontée du signal et production d'un conseil en fonction de l'état des tâches*

9. Typologie des conseils

9.1. *Définition des conseils*

Chaque conseil produit est représenté par un objet, instance de la classe `Conseil`. Cette classe définit les attributs suivants :

---

[4] Ces cinq méthodes représentent le caractère abstrait de la notion même de classe de comportement. En pratique, les langages de programmation par objets proposent des mécanismes adaptés pour les représenter (méthodes `subclassResponsibility` en Smalltalk ou `deferred` en Eiffel).

- `type`

Le type du conseil. Ce type est un symbole, permettant de classer les conseils selon leur nature. Parmi les types on trouve : `EnCoursDeja`, `Deja`, `Premature`, etc.

- `texte`

Le texte du conseil

- `tâche`

La tâche ou action qui a produit le conseil

- `heure`

Le moment ou le conseil a été produit

9.2. *Production de conseils*

La production des conseils passe systématiquement par un objet `GestionnaireDeConseils`, qui détient la liste des conseils produits pour chaque action utilisateur. Cette liste permet de sauvegarder l'historique des conseils (cf. figure 8), et permet la gestion ultérieure de l'affichage des conseils suivant diverses stratégies tutorielles.

10. Classes d'action et classes de tâches

10.1. *Classes d'actions*

Afin de préserver la structure d'arbre pour l'espace des solutions, nous introduisons la notion de "classe d'actions". Une classe d'actions représente une action, en dehors de tout contexte[5]. Une classe d'actions est définie par un couple (unité physique, état).

Il faut noter ici que l'on ne peut pas directement utiliser une classe d'actions dans l'arbre des tâches, pour éviter de "partager" des nœuds. En particulier, une classe d'actions ne contient pas d'état, ni de comportement. En revanche, toute action (dans l'arbre des tâches) connaît sa classe d'actions.

Pour résumer on distingue trois notions d'action :

- Action à proprement parler

représente une action dans l'arbre des tâches.

- Classe d'action

est représentée par un couple (unité physique, état), qui est transmis par le simulateur. C'est par cette classe que l'on pourra identifier les nœuds d'entrée de l'arbre des tâches, à chaque action de l'utilisateur. A chaque action de l'arbre des tâches est associée une classe d'actions.

---

[5] Dans le cas du projet SAFARI, c'est ce que renvoie le simulateur VAPS lorsqu'une action utilisateur est interceptée. Dans l'architecture épiphyte, toute action peut être interceptée grâce à un mécanisme d'espionnage [Pachet 1995].

- ActionEvent
représente une occurrence d'une action utilisateur dans le temps.

10.2. *Éditeur d'actions*

L'éditeur d'arbre des tâches permet de spécifier les liens entre tâches et sous-tâches. Pour chaque tâche ou action, on peut ensuite ouvrir un éditeur spécialisé permettant de définir certains paramètres. Pour les actions, on pourra ici spécifier la classe d'action correspondante, parcourir les classes existantes, ou en créer de nouvelles (cf.figure 4). Noter qu'une procédure permet de sauvegarder la liste des classes d'action à tout moment (voir le bouton "Save" dans l'éditeur d'actions).

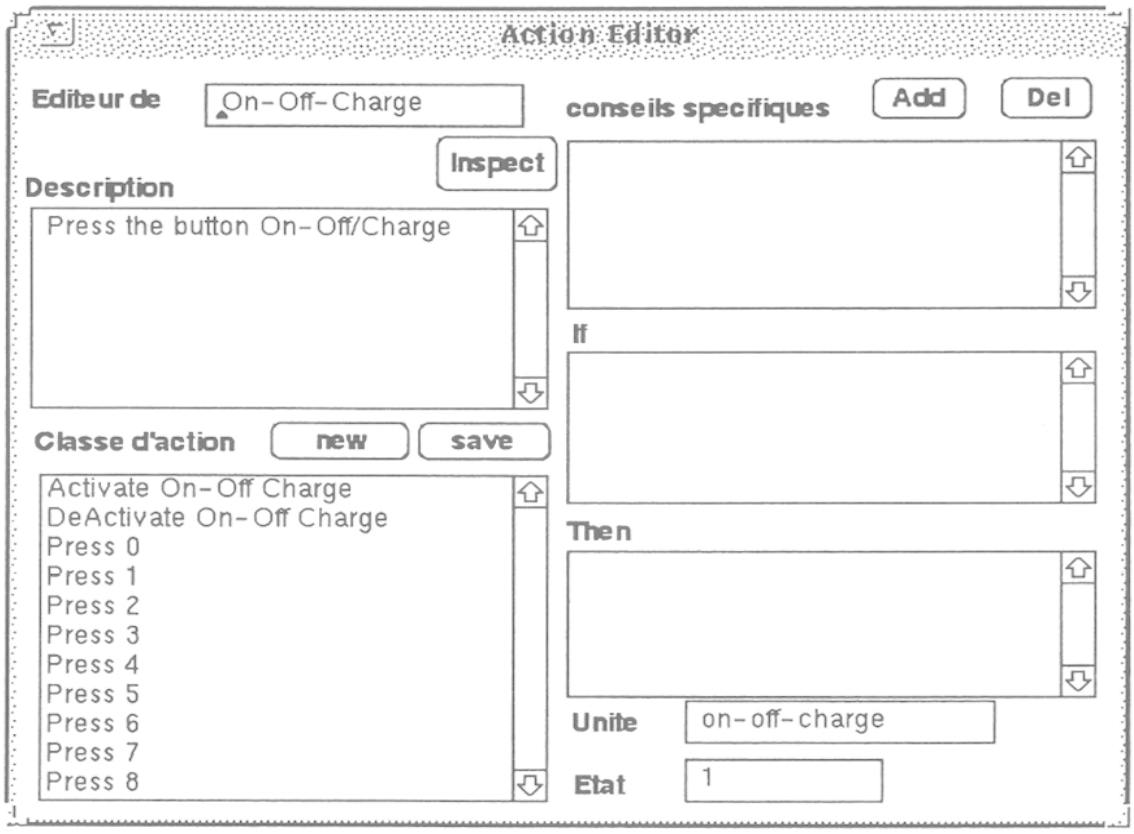

Figure 4. *Éditeur d'actions pour le bouton On-Off-Charge*

10.3. *Classes de tâches*

Il serait souhaitable d'introduire une distinction similaire pour les tâches (cf. éditeur, figure 5), entre "Tâche" et "Classe de tâche", qui nous permettrait de factoriser ce qui est commun. L'utilisateur pourrait alors définir des "classes de tâches" qu'il "instancierait" au besoin pour définir son arbre de tâches. Cependant le problème est plus compliqué pour les tâches, puisqu'il faut alors définir récursivement leur structure de manière abstraite. Nous proposons une solution intermédiaire ici,

qui consiste à fabriquer des classes de tâches particulières au coup par coup, à la main, pour les tâches les plus communes. Nous traitons ainsi le cas [Entrer une fraction (f)] comme un cas particulier, nécessitant une classe de tâche spécifique.

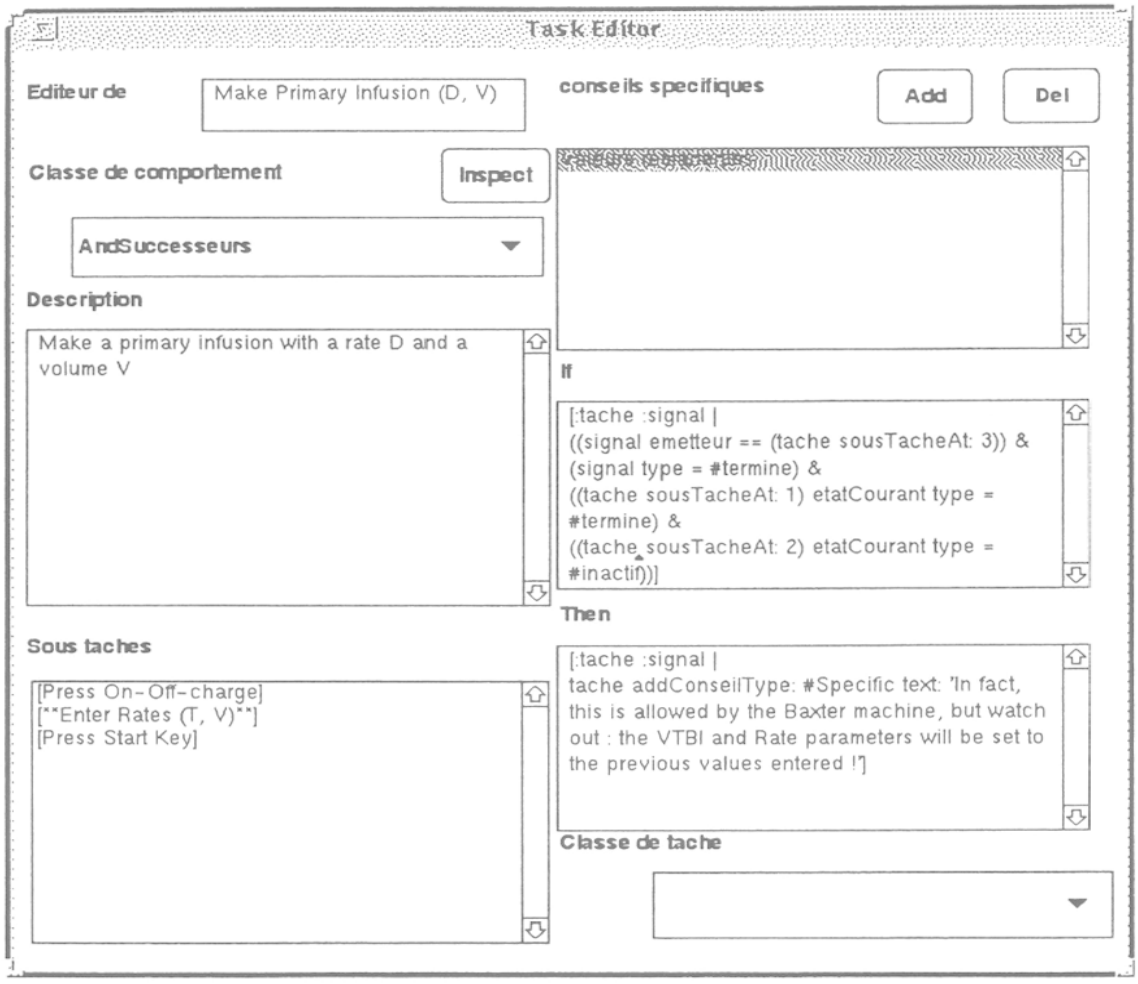

Figure 5. *Éditeur de tâche pour l'exemple de l'infusion primaire, avec conseil spécifique*

10.4. *Le cas de [Enter Fraction (f)]*

La tâche [Entrer une fraction] se retrouve souvent dans les arbres de tâches sur la pompe Baxter, comme c'est probablement le cas pour tous les arbres de tâches concernant l'utilisation d'appareils de ce type. On pourrait définir cette tâche en utilisant notre architecture telle quelle. Il nous faudrait pour cela définir une tâche [Entrer fraction] ayant 10 sous-tâches. Chaque sous-tâche serait l'action correspondant à la pression sur une des 9 touches numériques ([Appuyer sur 0], [Appuyer sur 1], etc.), plus une sous-tâche pour la touche "point" (décimal). De plus, il faudrait introduire

un comportement particulier pour cette tâche, qui consisterait à vérifier que la séquence d'actions permet effectivement de produire la fraction souhaitée.

Au lieu de procéder de cette manière — fastidieuse pour le concepteur — nous proposons une classe de tâche particulière `EnterFractionTask` qui représente cette tâche, une fois pour toute. Le concepteur peut alors "instancier" cette classe de tâche particulière, chaque fois qu'elle est nécessaire.

Ceci se fait en pratique en instanciant la classe `EnterFractionTask`, avec en paramètre la fraction correspondante. Dans l'éditeur graphique, une icône particulière lui est réservée. La figure 6 montre l'éditeur d'arbres de tâches comprenant, entre autres, une tâche [entrer fraction (f)].

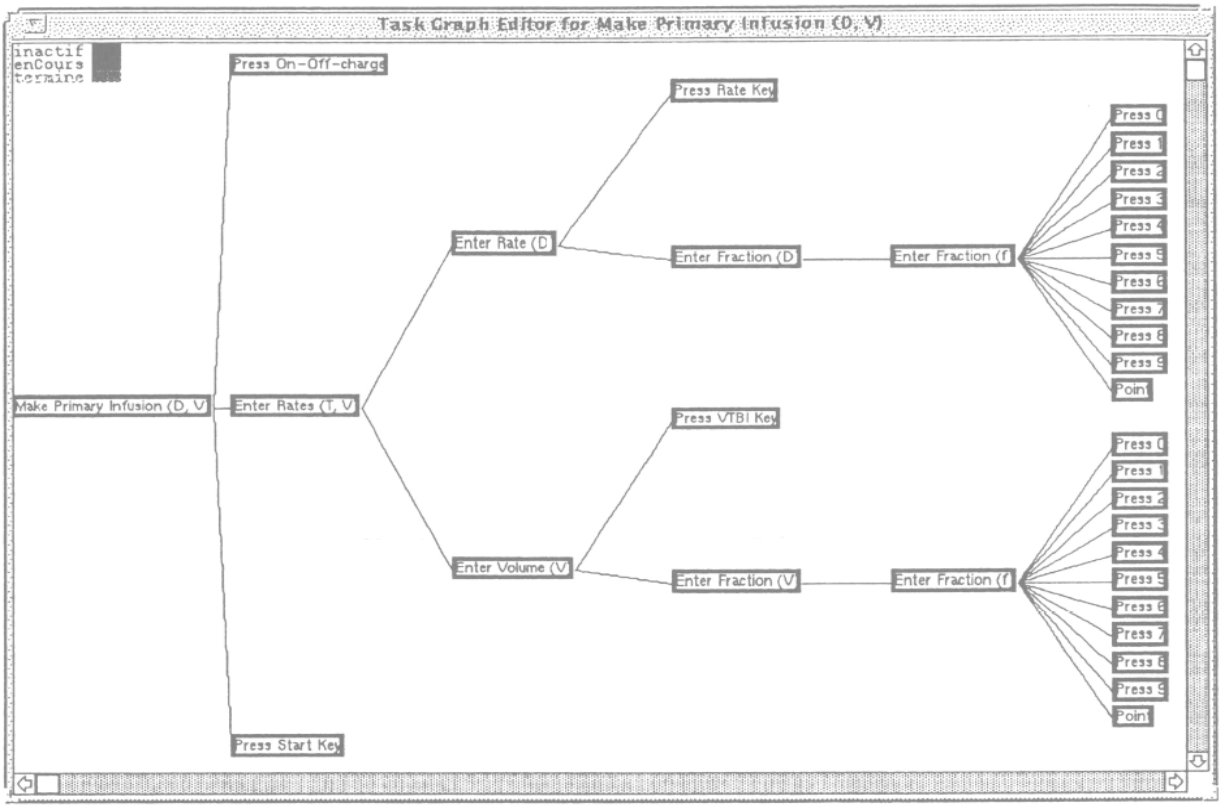

Figure 6. *Exemple complet de l'infusion primaire avec fraction*

11. Ajouter des connaissances dépendantes du domaine

11.1. *La notion de règle de conseil spécifique*

Le mécanisme que nous avons présenté ici permet de produire de manière systématique des conseils portant uniquement sur les relations de précédence, définies par les classes de comportement. Nous pouvons cependant greffer simplement sur ce mécanisme la production de conseils spécifiques quelconques. Pour ce faire, nous introduisons la notion de *règle de conseil spécifique*, qui peut être associée à chaque tâche (ou action), et se déclencher au moment voulu pour produire un conseil spéci-

fique. Une règle de conseil spécifique est définie par trois données (cf. éditeur figure 5) :

- une *condition de déclenchement*. Cette condition porte sur le signal reçu, et la tâche concernée. On accède alors aux états de la tâche et de ses sous-tâches, et aux informations portées par le signal. En pratique, elle est représentée par un bloc Smalltalk prenant deux arguments, liés au moment de l'appel, (cf. figure 5) : la tâche, et le signal,
- une *action* à effectuer. Généralement, cette action sera la production d'un conseil de type spécifique. Elle est aussi représentée par un bloc Smalltalk prenant deux arguments, liés au moment de l'appel : la tâche, et le signal,
- un *nom* (pour les besoins d'interface).

11.2. *Exemple : D'accord, Mais Attention !*

Voici un exemple typique de conseil spécifique : il s'agit, pour la tâche "Faire une infusion primaire" de produire un conseil lorsque l'utilisateur "oublie" de rentrer les taux. La condition du conseil est donc une triple condition :

- on reçoit un signal de la troisième sous-tâche (le bouton Start),
- la première sous-tâche est terminée (on a appuyé sur le bouton On-Off/charge avant),
- la deuxième sous-tâche ([Entrer les taux]) est à l'état inactif, i.e. n'a jamais été commencée.

Cette condition s'écrit donc simplement comme suit :

```
f (signal, tache)
  ((signal emetteur == (tache sousTacheAt : 3)) &
  (signal type = #termine) &
  ((tache sousTacheAt : 1) etatCourant type = #termine) &
  ((tache sousTacheAt : 2) etatCourant type = #inactif))
```

La partie action consiste à produire un conseil spécifique, dont le texte est par exemple :

'In fact, this is allowed by the Baxter machine, but watch out : the VTBI and Rate parameters will be set to the previous values entered !'

12. Interface

La figure 7 montre l'élément d'interface principal pour la production de conseils sur des simulations de traces.

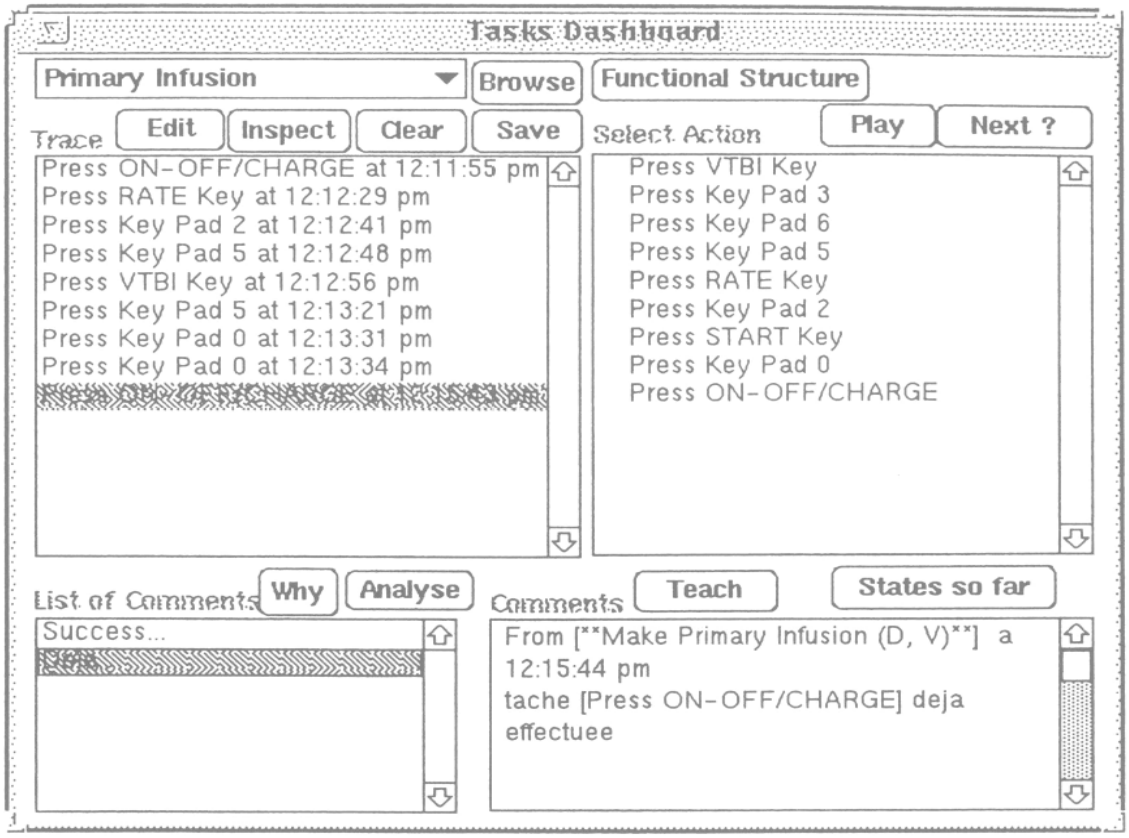

Figure 7. *Menu principal et exemple complet*

Cette figure comprend cinq parties. La première partie est un menu présentant la liste des problèmes traités (voir "Primary Infusion" au coin supérieur gauche). La deuxième partie, appelée *Select Action*, comprend la liste des actions possibles. Les boutons *Play* et *Next ?* permettent respectivement de jouer une action sélectionnée et d'afficher la liste des actions suivantes possibles. La troisième partie, *Trace*, est la trace effective. Quatre boutons y sont associés (*Edit*, *Inspect*, *Clear*, *Save*). Le bouton *Edit* permet d'afficher une fenêtre comprenant l'arbre des tâches (cf. figure 6). Les tâches (abstraites ou concrètes) contenues dans cet arbre ont chacune un état : `inactif`, `enCours` ou `termine`. *Inspect* permet d'inspecter l'arbre des tâches ; *Clear* sert à reinitialiser l'exécution de la tâche. *Save* permet de sauvegarder la trace d'exécution de la tâche lorsque celle-ci s'avère particulièrement intéressante.

La quatrième partie, *List of Advice*, permet d'afficher la liste des conseils relatifs à l'action sélectionnée dans le menu trace.

La cinquième partie, *Advice*, comprend le détail du conseil vis-à-vis du type de conseil dans *List of Advice*. Les boutons *Teach* et *States so far* permettent respectivement de produire un texte tutoriel explicatif en fonction des actions effectuées par l'usager et de fournir une fenêtre comprenant l'historique des états des tâches (figure 8).

Dans l'historique des tâches, *Tâches* et *Actions* énumèrent toutes les tâches de l'arbre des tâches ; *Historique des états* donne des informations sur les différents états par lesquels la tâche est passée pendant l'exécution de la trace. Tri permet d'af-

ficher les tâches selon certains critères : `toutes` pour tout état des tâches, `enCours` pour les tâches en cours d'exécution ; `termine` pour les tâches terminées et `inactif` pour les tâches non encore activées.

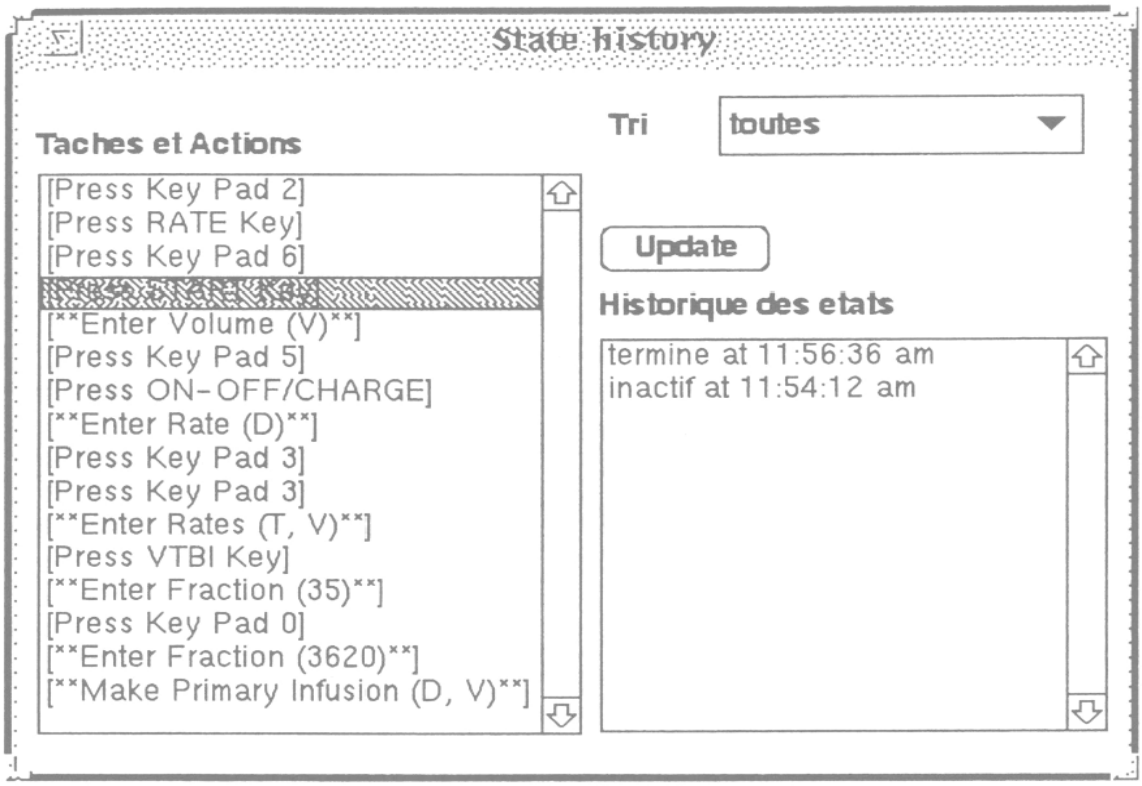

Figure 8. *Historique des états*

13. Bilan et perspectives pour les STI

13. 1. *Perspectives*

Notre approche peut être exploitée de diverses manières dans les STI.

Classification des types de conseils à donner à l'usager

Notre expérience avec les productions de conseils dans le projet SAFARI nous a persuadé qu'il était important de proposer des typologies des conseils produits. Ainsi, plutôt que de produire de simples textes, notre générateur de conseils produit des objets structurés, de classes distinctes en fonction de leur nature. Cette typologie permet de concevoir la superposition de modules d'analyse des conseils produits, selon diverses stratégies tutorielles. Ainsi, on peut choisir de ne présenter que certains types de conseils à l'apprenant en fonction de son niveau ou de stratégies tutorielles particulières, ou bien au contraire, renseigner le modèle de l'apprenant à partir d'analyses de régularités des conseils qui lui sont donnés.

Génération dynamique de documents

Un cours peut ainsi être donné en fonction du comportement de l'usager, pour donner par exemple des réponses contextuelles aux questions du type "comment faire... ". Cette génération se fait pour l'instant simplement et naïvement par un parcours en profondeur de l'arbre des tâches, exploitant les représentations d'état calculées par notre mécanisme. Ainsi, on peut produire des textes tutoriels insistant sur ce qui n'a pas encore été fait, ou mal fait, et occultant les tâches et actions n'ayant pas provoqué d'erreurs. Bien entendu, les structures calculées par le système permettent d'envisager la production de textes beaucoup plus élaborés.

Entrée du modèle de l'apprenant

Il est possible de communiquer au module traitant du modèle de l'apprenant des informations du type : telle tâche est passée à tel état à tel moment, etc. Ces informations peuvent ensuite être re-analysées pour alimenter différents modèles de l'apprenant.

Méthodologie pour une analyse cognitive des tâches directement exploitable par un STI

Une des applications immédiates de ce système est son utilisation par les concepteurs d'arbre des tâches eux-mêmes. En effet, la nature réactive du système, et sa relative légèreté algorithmique permettent de tester immédiatement des arbres de tâches sur des exemples de complexité progressive.

Références croisées pour les tâches (abstraites ou concrètes)

Un certain nombre d'outils d'environnement sont en cours de construction pour permettre aux concepteurs de maîtriser la complexité des arbres de taille substantielle, comme en particulier des outils de références croisées, permettant de visualiser les interdépendances entre tâches et actions.

Enrichissement de la structure de l'arbre de tâches

En particulier, nous étudions l'introduction d'autres types de relations et d'information comme celles d'"observations pertinentes", les conditions d'activation, et leur exploitation systématique dans le cadre de notre approche.

Système épiphyte

La connexion de notre architecture avec les techniques d'espionnage [Pachet 1995 ; Paquette 1995] est en cours, afin de permettre la fabrication de modules conseillers au-dessus d'applications objets déjà écrites, sans modifier leur code.

13.2. *Limitations du système*

Le mécanisme que nous proposons ici repose sur l'exploitation systématique d'arbres de tâches. La qualité de ces arbres et leur pertinence prennent donc une importance extrême, ce qui peut conduire à des productions de conseils inadaptés

lorsque les arbres de tâches sont mal conçus. En prenant un point de vue paradoxal mais réaliste, cette dépendance du module conseiller peut être tournée en avantage en considérant le module conseiller comme un moyen de validation de ces arbres de tâches.

Par ailleurs, le mécanisme proposé est particulièrement bien adapté aux problèmes bien formalisés, pour lesquels il existe des solutions standards connues et en nombre limité. Il semble que c'est le cas pour l'apprentissage de machines sophistiquées.

Une limitation importante du mécanisme est sa gestion des ambiguïtés. Dans l'état actuel de l'algorithmique, le système interprète les actions une par une, et n'est pas capable d'attendre la prochaine action pour interpréter les éventuelles ambiguïtés (il n'y a pas de "look-ahead"). Comme nous l'avons vu (section 8.2) le système demande explicitement à l'utilisateur de choisir une interprétation parmi celles possibles. Ceci peut devenir une limitation importante pour les graphes de tâches fortement ambigus (i.e. contenant beaucoup de répétitions de la même tâche).

14. Conclusion

Nous avons présenté un mécanisme permettant d'exploiter de manière systématique deux relations privilégiées dans un graphe de tâches, pour produire des conseils pertinents, demandant un minimum d'interaction avec l'apprenant.

Notre approche n'est pas sans rappeler certaines méthodes d'analyse syntaxique, appliquées à un langage particulier : celui des actions utilisateurs. La différence principale par rapport à ces méthodes est que l'analyse syntaxique cherche à interpréter des phrases syntaxiquement correctes, en rejetant, le plus souvent brutalement, celles qui ne le sont pas. Au contraire, nous nous intéressons ici essentiellement aux "phrases syntaxiquement incorrectes", i.e. les séquences d'actions illégales, pour essayer d'en déduire certaines intentions de l'apprenant en vue de mieux le conseiller. En ce sens, les mécanismes classiques d'analyse syntaxique ne peuvent pas être directement utilisés pour nos besoins.

Nous avons proposé ici une approche qui s'est avérée, une fois mise en pratique, très satisfaisante car offrant un bon compromis entre la complexité inhérente des mécanismes de reconnaissance de plan et les contraintes d'interactivité élevées propres au systèmes tutoriels. Notre système a été validé dans un premier temps comme module spécialisé, dans les phases préliminaires du projet SAFARI. Il est maintenant utilisé comme mécanisme de base pour les différentes modules d'analyses de tâches du système.

Remerciements

Cette recherche est financée par le ministère de l'industrie, du commerce, de la science et de la technologie du gouvernement du Québec, que nous remercions pour leur support. Nous remercions aussi vivement les relecteurs anonymes de cet article pour leurs critiques et conseils pertinents.

Bibliographie

[Allen 1990] J. Allen, J. Hendler, A. Tate, *Readings in Planning*, Morgan Kaufmann publishers, Inc., 1990.

[Baxter 1992] Baxter, Operator's Manual, Flo-Gard 6201, Volumetric Infusion Pump, Baxter Healthcare Corporation, 1992.

[Brown 1980] J.S. Brown, K. VanLehn, « Repair theory : a generative theory of bugs in procedural skills », *Cognitive Science*, 4, 1980, 379-426.

[Carberry 1990] S. Carberry, *Incorporating default inferences into plan recognition*, Eigth National Conference on Artificial Intelligence, 1990, 471-478.

[Carr 1977] B. Carr, I.P. Goldstein, *Overlays : a theory of modeling for computer-aided instruction*, Massachusetts Institute of Technology, 1977.

[Cohen 1990] P.R. Cohen, J. Morgan, M. Pollack, *Intentions in Communications*, MIT Press, 1990.

[Desmarais 1993] M.C. Desmarais, L. Giroux, S. Larochelle, *An advice-giving interface based on plan-recognition and user-knowledge assessment*, Int. Journal of Man-Machine Studies, 39, 1993, 901-924.

[Djamen 1995] J.-Y. Djamen, *Architecture de système tutoriel intelligent pour l'analyse du raisonnement de l'apprenant*, Ph. D., Université de Montréal, 1995.

[Djamen 1994] J.-Y. Djamen, C. Frasson, M. Kaltenbach, P. Obenson, A. Kabbaj, *PIF_g ou comment éditer les connaissances de l'apprenant dans un STI*, Deuxième conférence africaine sur la recherche en informatique, 1994.

[Djamen 1993] J.-Y. Djamen, M. Kaltenbach, C. Frasson, *An interactive planning and learning system*, AI-ED '93, 1993, 354-361.

[Ellis 1993] G. Ellis, R.1. Levinson, *Workshop on PEIRCE : a conceptual graph workbench*, accessible par ftp dans ftp.cs.adelaide.edu.au/pub/peirce, 1993.

[Ernst 1969] G.W. Ernst, A. Newell, *GPS : A case study in generality and problem solving*, Academic Press, 1969.

[Goldstein 1982] I.P. Goldstein, *The genetic graph : a representation for the evolution of procedural knowledge*, Intelligent Tutoring Systems, New York, Academic Press, 1982.

[Hoppe 1988] H.U. Hoppe, *Task-Oriented Parsing - a Diagnostic Method to be used by Adaptative Systems*, CHI '88, 1988, 241-247.

[Kautz 1990] H. Kautz, *A Circonscriptive Theory of Plan Recognition*, Intentions in Commnications, MIT Press, 1990.

[Kautz 1986] H.A. Kautz, J.F. Allen, *Generalized Plan Recognition*, AAAI '86, 1986, 32-37.

[Lesgold 1992] A. Lesgold, S.P. Lajoie, M. Bunzo, G. Eggan, *SHERLOCK : A Coached Practice Environment for an Electronic Troubleshooting Job*, Computer Assisted Instruction and Intelligent Tutoring Systems : Shared goals and complementary approches, Hillsdale, Lawrence Erlbaum Associates, 1992.

[Lesgold 1990] A. Lesgold, S.P. Lajoie, D. Logan, G. Eggan, *Applying cognitive task analysis and research methods to assessment*, Diagnostic monitoring of skills and knowledge acquisition, Hillsdale, Lawrence Erlbaum Associates, 1990.

[McKendree 1988] J. McKendree, J. Zaback, *Planning for Advising*, CHI '88, 1988, 179-183.

[Mittal 1988] S. Mittal, D. Bobrow, J. de Kleer, *DARN : Toward a Community Memory for Diagnosis and repair Tasks*, Expert Systems : The User Interface, Norwood, Ablex, 1988.

[Niem 1993] L. Niem, J. Fugere, P. Rondeau, R. Tremblay, *Defining the semantics of extended genetic graphs*, User Modeling and User-Adapted Interaction, 23, 1993,

[Pachet 1994] F. Pachet, S. Giroux, G. Paquette, *Pluggable Advisors as Epiphyte Systems*, Calisce '94, 1994, 167-174.

[Pachet 1995] F. Pachet, F. Wolinski, S. Giroux, *Spying as an object-oriented programming paradigm*, TOOLS' Europe 95, 1995, 109-118.

[Paquette 1995] G. Paquette, F. Pachet, S. Giroux, *EpiTalk, a generic tool for the development of advisor systems*, Journal of Artificial Intelligence in Education, à paraître, 1995,

[Quast 1993] K.-J. Quast, *Plan recognition for context-sensitive help*, International workshop on intelligent user interface, 1993, 86-96.

[Sandrasegaran 1994] N. Sandrasegaran, R. Bouchard, S. Lajoie, *Curriculum issues for the Baxter pump*, Universite de Montreal, projet SAFARI, 1994.

[Sidner 1981] C.L. Sidner, D. Israel, *Recognizing Intended Meaning and Speaker's Plans*, IJCAI, 1981,

[Sleeman 1979] D.H. Sleeman, R.J. Hendley, *ACE : a system which analyses complex explanations*, Int Jrnl Man Machine Studies, 11, 1979, 125-144.

[Sowa 1984] J. Sowa, *Conceptual Structures. Information processing in mind and machine*, Addison Wesley, 1984.

[Stevens 1977] A.L. Stevens, A. Collins, *The goal structure of a Socratic tutor*, National ACM Conference, 1977, 256-263.

[VanLehn 1988] K. VanLehn, *Student modeling*, Foundations of Intelligent Tutoring Systems, New Jersey, Lawrence Erlbaum Associates, Inc, 1988.

[Vaps 1993] Vaps, *Programmer's Guide*, Virtual Prototypes Inc., 1993.

[Wenger 1987] E. Wenger, *Artificial Intelligence and Tutoring Systems*, Morgan kaufmann publishers, Inc, 1987.

François Pachet *est ingénieur civil des Ponts et Chaussées, docteur en informatique de l'université Pierre et Marie Curie (Paris VI). Il est actuellement maître de conférences au laboratoire Laforia-IBP (équipe objets, modèles et connaissances, dirigée par Jean-François Perrot), de l'université Paris VI. Il s'intéresse à la représentation de connaissances et à l'intégration de mécanismes d'inférences dans les langages à objets.*

Jean-Yves Djamen *est ingénieur en informatique de l'Institut Africain d'Informatique (Libreville), et docteur en informatique de l'université de Montréal. Il s'intéresse à la conception et au développement de systèmes tutoriels intelligents et d'une manière générale à la modélisation des systèmes physiques pour des fins de formation. Sa thèse porte sur une architecture de systèmes tutoriels intelligents permettant d'analyser différentes formes de raisonnement d'un apprenant dans un contexte de formation. Cette thèse a contribué à la modélisation des connaissances dans le projet SAFARI. Il est actuellement chercheur dans le groupe HÉRON-MULTIMÉDIA.*

Claude Frasson *est professeur titulaire, directeur du GRITI et du laboratoire HERON, département d'informatique et de recherche opérationnelle, université de Montréal. Doctorat d'état en informatique de l'université de Nice. Après s'être spécialisé dans l'optimisation des performances des bases de données, Claude Frasson s'est orienté depuis huit ans dans le domaine des système tutoriels intelligents (STI). Il s'intéresse particulièrement à la modélisation des connaissances de l'étudiant, à l'architecture des STI et à l'analyse du raisonnement de l'apprenant. Cette dernière approche vise à déterminer les moyens formels qui permettent de suivre les actions de l'apprenant et d'interpréter ses différents niveaux de compréhension des tâches et d'acquisition de connaissances.*

Marc Kaltenbach *est professeur titulaire, Information and Decision Sciences, université Bishop's. Il est conjointement professeur associé au département d'informatique et de recherche opérationnelle de l'université de Montréal. Il a une maîtrise de l'université de Yale et un Ph.D. en génie électrique de l'université de Toronto. Après s'être intéressé pendant une dizaine d'années à des problèmes de modélisation et d'optimisation de systèmes dynamiques, depuis six ans il s'intéresse à la conception et au développement de systèmes informatiques de formation. Ses principaux axes de recherche dans ce domaine sont les interfaces personnes machines pour favoriser la transmission de concepts et d'informations complexes tels que trouvés dans des preuves en mathématiques, l'utilisation de composition iconiques pour représenter des cas (médicaux) et faciliter des processus de mémorisation et d'inférences.*