février 1998 – Journées Francophones des Langages Applicatifs – JFLA98

Conception par objets d'un système pour combiner raisonnement formel et satisfaction de contraintes

Anne Liret, Pierre Roy, François Pachet

(1) LIP6, Boîte 169, Université Pierre et Marie Curie, 4 place Jussieu,
75252 PARIS cedex 05, France.
e-mail: {liret, roy, pachet}@poleia.lip6.fr

Résumé

La programmation par satisfaction de contraintes (CSP) est un outil puissant qui permet de résoudre de nombreux problèmes combinatoires. Cependant, il existe des situations dans lesquelles un raisonnement formel sur les contraintes trouve le résultat plus vite et plus facilement que les techniques classiques de CSP. Dans cette optique, nous reconstruisons le système Alice, qui proposait de combiner les techniques de CSP avec l'introduction de contraintes redondantes. Dans le système AliceTalks, nous proposons d'utiliser un mécanisme général de calcul symbolique, la réécriture, comme support au raisonnement formel sur les contraintes. De plus, afin d'identifier clairement les situations "favorables" au raisonnement formel, nous avons choisi d'utiliser la programmation par objets. Le système AliceTalks contient donc son propre système de réécriture de contraintes, à base d'objets. Cet article décrit le module de raisonnement formel d'AliceTalks et montre en quoi la programmation par objets a permis d'élaborer un environnement modulaire et extensible pour l'expérimentation de nouvelles stratégies de résolution.

1. Introduction

La programmation par satisfaction de contraintes (CSP) est un outil puissant qui permet d'exprimer et de résoudre de nombreux problèmes combinatoires (problèmes d'emploi du temps, d'ordonnancement, de planification). Un problème de satisfaction de contraintes est défini par un ensemble de variables, chacune associée à un domaine de valeurs, et un ensemble de contraintes. Une solution est une bijection qui associe à l'ensemble des variables, un n-uplet de valeurs, tel que toutes les contraintes soient respectées.

Depuis quelques années, beaucoup de travaux ont porté sur l'intégration des techniques de CSP dans les langages à objets. Un de leurs objectifs était d'exploiter les facilités de la programmation par objets, pour concevoir des systèmes ouverts et intégrables. Ces derniers fonctionnent comme des extensions du langage hôte. Il existe maintenant plusieurs systèmes efficaces de satisfaction de contraintes à base d'objets. Le système LAURE [3], généralisé dans CLAIRE [4], est un langage de programmation par objets pour la construction de système de CSP. CLAIRE étend le langage de base, avec des structures comme les démons et les règles. D'autres systèmes, comme IlogSolver [16] et BackTalk [17], proposent une librairie de classes et d'algorithmes réutilisables pour la satisfaction de contraintes.

Les systèmes de satisfaction de contraintes mettent en oeuvre deux mécanismes : une procédure d'énumération de type *Génération-Test* qui associe une valeur à une variable non encore instanciée, et un ensemble de techniques qui réduisent l'arbre de recherche des solutions, avant chaque nouvelle instanciation. Les techniques classiques de CSP, comme l'arc-cohérence [15, 2], ont pour but de propager les instanciations dans l'ensemble des contraintes du problème. Plus généralement, les techniques utilisées pour améliorer l'énumération, déduisent de nouvelles contraintes qui transforment le problème en un problème équivalent, plus facile à résoudre [8]. En particulier, l'arc-cohérence consiste en une réduction de domaine, ou filtrage, qui élimine les valeurs susceptibles de provoquer une incohérence. Un autre type d'inférence, le raisonnement formel, initialement proposé dans le système Alice [11], raffine l'ensemble des contraintes, en ajoutant des contraintes redondantes plus simples. Cette technique, bien que séduisante, est intrinsèquement lente et n'a jamais été, à notre connaissance, étudiée de manière systématique. De ce fait, dans la plupart des systèmes actuels, seul le filtrage de contraintes est utilisé. Cependant il existe des problèmes dans lesquelles les techniques classiques de CSP sont rédhibitoires. Nous nous sommes donc intéressés aux situations dans lesquelles un raisonnement formel trouve un résultat plus vite et plus facilement qu'une énumération classique. Par exemple, nous citons les situations suivantes :

- Soit l'ensemble d'équations suivant : $\{(1)\ X+Y = 10\ ;\ (2)\ X-Y = 0\}$. En substituant X par Y dans (1) on déduit immédiatement que $X = Y = 5$, sans faire référence au domaine de X ou de Y. De même, Si l'on considère l'ensemble de contraintes $\{(1)\ X = Y\ ;\ (2)\ Z = Y\ ;\ (3)\ X \neq Z\}$. Par un raisonnement formel, on peut facilement prouver que cet ensemble est incohérent car (1) et (2) impliquent la contrainte $[X = Z]$, qui contredit (3) de manière évidente. Enfin, soit l'équation suivante dans N : $4m + 3n^2 = 34$; en raisonnant sur la parité de 34, on déduit que n^2 et $4m + 3n^2$

doivent être multiples de 4, ce qui n'est pas le cas de 34. On trouve donc que l'équation n'a pas de solution. Dans ces trois situations, une stratégie classique d'arc-cohérence ne trouverait ce résultat qu'avec une exploration combinatoire des domaines de valeurs des variables.

- Certains problèmes réels, comme les problèmes d'emploi du temps ou de prêt bancaire, se formulent à l'aide de contraintes complexes non linéaires, pour lesquelles il n'existe pas d'algorithmes de filtrage efficaces. Souvent, un raisonnement formel sur ces contraintes peut permettre de déduire symboliquement une expression analytique de la solution.

Ce type de situation peut apparaître au cours de la résolution d'un problème combinatoire, alors qu'une partie des variables sont instanciées. Il est donc intéressant de pouvoir combiner les deux stratégies, filtrage et raisonnement formel au cours d'une résolution. Cette idée fut initialement exploitée dans le système Alice et validée sur de nombreux exemples [12]. Mais, paradoxalement, cette expérience n'a pas été poursuivie. Plus récemment, de nombreux travaux se sont intéressés à la combinaison de méthodes pour résoudre des problèmes de contraintes [10, 18, 19, 20]. Il en ressort que le problème de base reste la complexité des méthodes de simplifications formelles et la définition d'une "bonne" stratégie. Nous reconstruisons donc Alice, à la lumière des récents travaux sur les CSP, en utilisant la programmation par objets. Le système obtenu, AliceTalks, est à la fois modulaire et extensible. Cet article décrit le système AliceTalks et détaille particulièrement le module de raisonnement symbolique sur l'ensemble des expressions arithmétiques et logiques reconnues par le système. Avec AliceTalks, nous proposons ainsi un environnement adapté à l'étude de stratégies et de déductions formelles.

2. Le système Alice

Une description complète d'Alice se trouve dans le chapitre 8 de [13]. De plus, J. Pitrat a développé une version déclarative d'Alice, dans son langage de règles, Maciste (Cf. chapitre 12 de [14]), apportant ainsi des éclaircissements sur le fonctionnement interne du module de filtrage. Néanmoins, il n'a jamais été mené d'étude systématique du raisonnement formel présent dans Alice.

2.1. Description

Alice signifie "A Language for Intelligent Combinatorial Exploration". Alice est donc, à la fois un langage de formulation de problème combinatoires numériques et un système de résolution. Le langage, basé sur la théorie des ensembles, permet de poser un problème à l'aide d'expressions mathématiques et d'un opérateur fonctionnel général. Le système est conçu en cinq modules : un analyseur syntaxique, un graphe, un ensemble de contraintes, un module de choix des heuristiques et un algorithme de contrôle de la résolution. Chaque module est responsable d'une tâche particulière : l'analyseur syntaxique est utilisé pour générer une représentation interne d'un CSP, à partir de sa formulation textuelle. Le graphe représente le domaine des variables et exécute le filtrage des contraintes. L'ensemble des contraintes maintient en permanence une liste des contraintes originales, augmentée des contraintes normalisées qui ont été ajoutées lors du raisonnement formel. Enfin le module des heuristiques détermine les choix de résolution, en fonction de l'état du graphe et de l'ensemble des contraintes.

L'algorithme de résolution est basé sur une procédure générale de *retour-arrière avec propagation* (forward-checking), comme décrit dans [7]. A chaque étape, le problème est simplifié selon deux stratégies : 1) la propagation de contrainte classique réduit les domaines dans le graphe ; et 2) le raisonnement formel génère des contraintes plus simples dans l'ensemble des contraintes. La réduction de domaine d'Alice est comparable aux techniques de filtrage telles que décrites dans [2]. Mais le raisonnement formel est unique. Nous le décrivons dans la section suivante.

2.2. Raisonnement formel en Alice

Le raisonnement formel en Alice apparaît sous deux formes : la normalisation de contraintes et la combinaison de contraintes. La normalisation d'une contrainte consiste à la réécrire sous une unique forme (sa forme normale). Cette étape garantie une représentation uniforme des contraintes ; elle est donc indispensable pour la manipulation symbolique de contraintes. Au cours de cette étape, le système trouve les instanciations et incohérences évidentes. Par exemple, la forme normale de $[X + 2.Y = X]$ est $[0 = Y]$; et celle de $[X = X + 1]$ est simplement $[false]$.

La déduction de contraintes consiste à créer de nouvelles contraintes plus simples. Cet ajout est particulièrement intéressant lorsqu'il permet de détecter l'absence de solution comme dans le problème $[4m + 3n^2 = 34]$, ou bien de déduire la valeur d'une variable, comme dans le problème

send + *more* = *money* (Cf. section 3.4). Dans un autre domaine, l'introduction manuelle d'une contrainte redondante permet de trouver très rapidement une borne optimale pour la solution de problème d'emploi du temps, comme illustré dans [5]. Mais la déduction de contraintes est un processus intrinsèquement lent et coûteux [18]. De plus la confluence des règles de déduction reste en général difficile à assurer [21]. C'est pourquoi il ne peut être appliqué systématiquement. Le problème est donc de savoir quand et comment déduire des contraintes. Cette question est difficile. Même si le système Alice donnait certains éléments de réponse, ceux-ci étaient cachés. C'est en partie la raison pour laquelle le système était si difficile à comprendre.

3. AliceTalks : un Alice à base d'objets

AliceTalks est né de la volonté d'avoir un système se comportant comme Alice, qui soit ouvert et adaptable. Nous avons choisi *Smalltalk* comme langage d'implémentation pour la grand quantité et la qualité des composants disponibles. Cette section reporte les principales caractéristiques de conception et d'utilisation du système, et montre, en section 3.3, en quoi la programmation par objets a particulièrement facilité l'intégration d'un module de manipulation symbolique et numérique sur l'ensemble des expressions arithmétiques et logiques.

3.1. Description

La conception d'AliceTalks correspond à la décomposition du système de J.L. Laurière en modules. AliceTalks se compose de cinq modules distincts.

- L'analyseur syntaxique améliore le langage d'Alice de deux manières : la syntaxe est plus naturelle et plus flexible, car l'analyseur est généré grâce à l'architecture d'applications *Parser-Generator*.
- Le graphe, contrairement à celui d'Alice, est totalement indépendant des autres modules.
- L'ensemble des contraintes étend les mécanismes de raisonnement formel d'Alice. Il contient une hiérarchie de contraintes et d'expressions avec des extensions pour leur manipulation symboliques et numériques. Cette hiérarchie est détaillée dans la section 3.3.
- Le module de trace, décrit en section 3.4, permet de comprendre la résolution.
- L'explorateur d'heuristiques (par analogie avec les flâneurs de Smalltalk), permet d'expérimenter des stratégies de résolution.

Chaque heuristique est calculée par une méthode, qui est exécutée en un point précis du programme (étiquette). L'ensemble des étiquettes est prédéfini dans le système. Pour chaque étiquette, on change l'heuristique utilisée en associant à l'étiquette une nouvelle méthode qui calcule l'heuristique.

3.2. Raisonnement formel en AliceTalks

Comme en Alice, le raisonnement formel en AliceTalks met en oeuvre deux processus que nous avons étendus : la normalisation et la déduction de contraintes. La normalisation des contraintes est maintenant conçue comme une application de la réécriture [9, 6]. La théorie de la réécriture, qui prend ses origines dans la théorie équationnelle, a rapidement trouvé une application directe dans les systèmes de calcul formel, comme Macsyma et Maple. Elle a été plus récemment appliquée à la conception de système de prototypage [10] et d'outil d'aide à la preuve de spécification de programmes, basé sur la théorie des types [5]. La réécriture consiste à remplacer des sous-termes d'une expression donnée, par des termes égaux. L'objectif d'un système de réécriture est de produire la forme normale d'un terme si elle existe. Cette existence est garantie lorsque la base de règle est convergente, i.e. toutes les séquences de réécriture sont finies (propriété de terminaison) et se termine avec une expression irréductible unique (propriété de confluence).

Dans le système actuel, la base de règle de réécriture assure la normalisation des termes de l'algèbre des expressions arithmétiques et logiques, augmentée des termes fonctionnels. Par exemple, l'égalité $[z + y.x + (x.y + t - z + x) = 2.f(x) + (x.2.y) + (-2.f(x + 0))]$ est réécrite en $[0 = t + x]$. La confluence de la base de règle d'AliceTalks a été vérifiée, à l'aide du programme de complétion disponible dans le système de prototypage sous contraintes, ELAN [10].

Les règles de déduction de contraintes peuvent aussi être conçues, d'un point de vue théorique, comme des règles de réécriture particulières. Le système de réécriture s'applique cette fois sur un ensemble de contraintes (au lieu d'une seule) et consiste à remplacer l'ensemble de contraintes par un ensemble contenant à la fois les contraintes initiales et un ensemble de contraintes redondantes. Dans AliceTalks, ces règles sont conçues comme des règles de production s'appliquant sur l'ensemble des contraintes. Par exemple, considérons les deux contraintes $[N = 1 + E]$ et $[10 + E = R + R1 + N]$ qui apparaissent au cours de la résolution du problème de crypto-arithmétique $send + more = money$. Le domaine de R1 est {0,1}, celui de N, R et E est {1... 8}. Une déduction intéressante consiste à ajouter ces deux contraintes, de manière à créer la contrainte $[9 = R + R1]$. Cette contrainte est évidemment redondante avec les

contraintes initiales. Néanmoins, un filtrage de cette contrainte permet de déduire deux instanciations: $R = 8$ et $R1 = 1$. En l'absence de raisonnement formel, cette situation n'est exploitable qu'avec une énumération globale des domaines. Cet exemple est détaillé en section 3.4.

La section suivante détaille la typologie des expressions supportant le raisonnement formel.

3.3. Réécriture en AliceTalks

Cette section décrit le système de réécriture qu'AliceTalks utilise pour la normalisation. Le système est entièrement intégré à la représentation par objets des termes de l'algèbre d'AliceTalks. Les règles ne sont pas réifiées. Cette conception a l'avantage d'éviter la création de nouveaux objets qui seraient uniquement dédiés à la réécriture. Il est important de le noter car ce mécanisme est à la base de toute modification ou création de contraintes. En particulier on veut pouvoir rajouter des règles de réécriture sans craindre la saturation du système. De plus, l'implantation de ce module dans un langage à objets permet d'étendre le système de réécriture à d'autres types d'expressions plus généraux, comme les contraintes de parité ou les sommes n-aires indexées sur un ensemble.

```
AExpression ('weight' 'unknownsNumber')
    AEConstant ('value')
    AEVariableAbstract ('degree' 'value' 'name' 'graphVariable')
          AEVariable ()
          AEVariableBoolean ()
          AFunctionalExpression ('functionId' 'operande'
'isManipulable' 'graph')
    ALogicExpression ()
          ABinLogicExpression ('left' 'right')
                AEAnd ()
                AEImply ()
          AUnLogicExpression ('operande')
                AENot ()
          ARelationExpression ('left' 'right')
                AEEquality ()
                AEInferiorOrEqual ()
    ArithmeticExpression ()
          ABinArExpression ('left' 'right')
                AEAdd ()
                AEDiv ()
                AEMul ()
                AESub ()
          AUnArExpression ('operande')
                AEOpposite ()
    (...)
```

FIG. 1 – *Extrait de la hiérarchie des classes d'expressions arithmétiques et logiques en AliceTalks.*

3.3.1. La hiérarchie d'expressions arithmétiques et logiques

Dans AliceTalks, les contraintes mathématiques sont exprimées par une formule logique infixée et représentées par une expression logique objet. D'un point de vue théorique, l'algèbre des expressions se compose de trois types de base (Constante, Variable, Expression fonctionnelle), et de l'ensemble des opérateurs classiques. L'algèbre définit aussi l'ensemble des comportements de ses opérateurs à l'aide d'une base d'équations. D'un point de vue pratique, les équations sont orientées selon un ordre strict sur les termes et le comportement des opérateurs est implanté par un système de réécriture. Notre conception suit exactement cette idée. La Figure 1 montre une partie de la hiérarchie de classes d'expressions.

On note que les termes fonctionnels sont considérés comme des variables. Leur domaine de valeur est stocké dans le graphe. La variable d'instance *'graphVariable'* est un pointeur vers la partie du graphe qui correspond au domaine. Ce lien permet d'implanter un mécanisme de démons entre le graphe et les expressions traitées par le raisonnement formel : quand celui-ci détecte une réduction de domaine au cours d'une propagation, il mémorise la variable concernée. A la fin du processus, il la signale à toutes les variables mémorisées. Les expressions contenant les variables sont alors normalisées si nécessaire.

3.3.2. L'ordre sur les expressions

L'ordre défini sur cette algèbre, est basé sur un ordre de précédence fixe. La précédence est modélisée par une collection ordonnée des classes d'expressions, triées par ordre croissant de priorité. Il y a donc deux listes de précédence, *ArithmeticPrecedence* et *LogicPrecedence* définies ainsi :

```
(ArithmeticPrecedence inspect)--->#(AEConstant, AEVariable,
          AFunctionalExpression, AEAdd, AEMul, AEOpposite)
(LogicPrecedence inspect)--->#(AEConstantBoolean,
      AEVariableBoolean, AFunctionalExpression, AEEquality,
AEDifference, AEInferiorOrEqual, AEOr, AEXor, AEAnd, AENot)
```

Deux expressions comparables sont ordonnées suivant l'ordre de précédence défini pour leur type (arithmétique ou logique). Si ce dernier ne suffit pas à déterminer un ordre strict, les deux expressions sont ordonnées selon un ordre de chemin lexicographique (lpo signifie lexicographic-path-ordering) (Cf.[9] pour une définition complète de cet ordre). L'ordre de base est décrit sur la Figure 2.

Cet ordre est nécessaire pour garantir l'unicité du résultat de la normalisation. Nous nous concentrons dans la suite sur la conception du système de réécriture qui utilise cet ordre.

```
basicGreaterThan: anExp
 | prec c |
 prec := self precedence.
 ((prec includes: self class)
       and: [prec includes: anExp class])
       ifTrue:
             [c := (prec indexOf: self class)
                              - (prec indexOf: anExp class).
             c = 0 ifFalse: [^c > 0]].
 ^self lpoGreaterThan: anExp
```

FIG. 2 – *Méthode de comparaison de deux expressions.*

3.3.3. Le système de réécriture des expressions

Un des objectifs initiaux d'AliceTalks était de réutiliser les frameworks existants. Le système de réécriture a donc été dans une première version implanté à l'aide du logiciel *MeiProlog* [1], une version de C-Prolog en Smalltalk. *MeiProlog* avait l'avantage d'offrir une syntaxe agréable pour la définition et le déclenchement de la base de règle. Le système traduisait ces descriptions, ainsi que l'expression à réécrire, en une représentation interne plus adaptée. Cette nécessité de traduire systématiquement les expressions ralentissait le système. C'est pourquoi dans une deuxième version, nous avons implanté notre propre système de réécriture.

A la base, une règle est l'association d'un test et d'une action à exécuter si le test est vrai. Dans les systèmes classiques, le terme est filtré par les variables muettes de la règle, créant une fonction de substitution de ces variables. Le terme réécrit est le résultat de l'application de cette substitution dans le membre droit. Dans une optique de programmation par objets, nous avons choisi une approche différente. Toute expression est modélisée comme un objet susceptible de se réécrire en sa forme normale. En d'autres termes, la réécriture d'une expression devient un simple envoi de message à l'objet représentant l'expression. Par exemple, l'envoi du message *normalize* à l'expression $[0 = 20 + 9.X]$ retourne l'expression $[0 = (20/9) + X]$. Nous avons implanté une base de règles comme un ensemble de méthodes de la classe de l'expression à réécrire. Chaque règle est implantée par une méthode de test (suffixée par *Test*) et une méthode d'action (suffixée par *Action*). Par exemple, la Figure 3 illustre une des règles de la base implantée dans la classe des égalités (AEEquality).

Si le message *equal3Test* est envoyé à une égalité **e**, la méthode teste si **e** est une expression de la forme $'0 = a + b.X'$, a et b étant des constantes, X représentant un terme quelconque. Si le résultat de l'envoi de message *(e equal3Test)* est vrai alors, le message *equal3Action* est envoyé à **e**. Le résultat de l'application de la règle sur **e**, est l'expression

```
equal3Test
    "(0= a + b*x) = (0 = a/b + x)"

    ^left isConstant: 0 and: (right class == AEAdd and: [right
left isConstant and: [right right isMonome or: [right right
class == AEMul and: [right right left isConstant]]]]]

equal3Action
    "(a + b*x=0) = (a/b + x= 0)"

   ^(AEConstant value: 0)
          @= ((AEConstant value: right left value / right right
coefficient value) + right right variable)
```

FIG. 3 – *La méthode de test et la méthode d'action de la règle de réécriture '[0 = a + b.X] [0 = (a/b) + X]'.*

retournée par la méthode *equal3Action*. Celle-ci est par construction, de la forme $'0 = (a/b) + X'$. Notons que le message *coefficient* (respectivement *variable*) renvoie la partie constante (respectivement variable) d'un terme. Cette règle est très utile car elle peut provoquer une instanciation de variable. Par exemple, dans le problème linéaire {[X+Y=10], [X-Y=0]}, la substitution de X par Y génère la contrainte $[Y + Y = 10]$. La normalisation la réécrit en $[0 = -10 + 2.Y]$, puis en $[0 = -5 + Y]$. Cette conception a l'avantage de réduire le coût de recherche des règles candidates aux seules règles concernées par l'expression. De plus,contrairement aux méthodes classiques, elle évite le calcul des substitutions. Cette remarque est importante, car le processus de normalisation est un des mécanismes d'AliceTalks les plus utilisés.

3.4. Comprendre la résolution

Un des objectifs d'AliceTalks est de permettre à un utilisateur de comprendre la résolution du problème. Pour cela, une interface de traçage du système (le *Tracer*) informe non seulement sur les étapes suivies (différents envois de message) mais aussi sur l'état des différents modules du système. De plus chaque trace est typée en fonction du module qui l'affiche (AG pour Alice Graph, AC pour Alice Constraint,...) et de l'étiquette identifiant le contenu de la trace. Cette typologie peut être utilisée pour filtrer les informations intéressantes. La figure 4 montre un extrait de la résolution du problème $send + more = money$, avec des heuristiques d'instanciation classiques (variable ayant le plus petit domaine, valeur minimale du domaine). La stratégie d'AliceTalks est simple : lorsque le filtrage de la contrainte la plus informante ne permet plus de réduire les domaines, AliceTalks essaie d'appliquer une des méthodes de raisonnement formel, associée à cette contrainte.

Dans cette partie de la résolution, le système agit de manière non triviale, combinant une forme de raisonnement formel (la substitution)

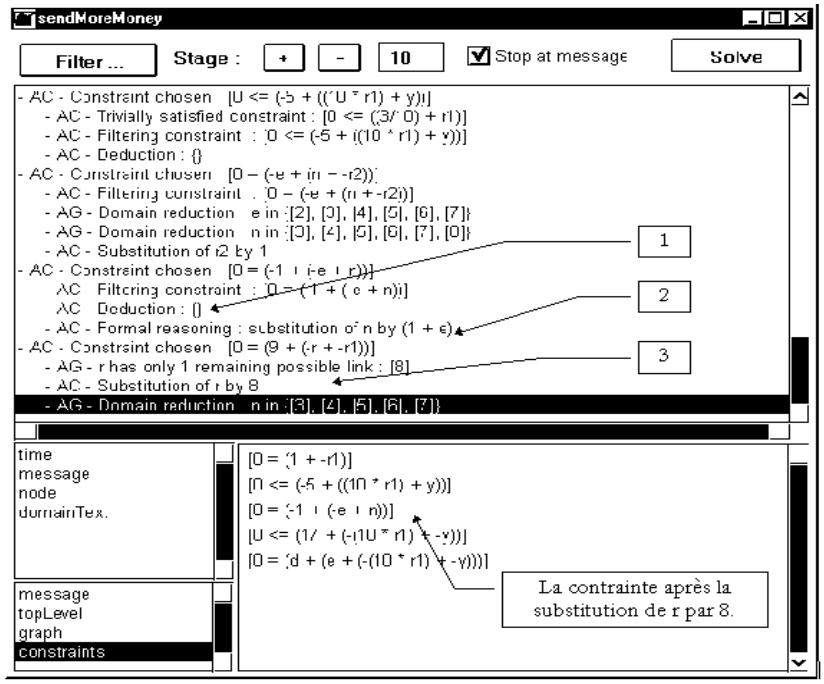

FIG. 4 – *Extrait de la trace de résolution du problème send + more = money.*

avec le filtrage de contraintes. A ce moment de la résolution, la variable s est déjà instanciée par 9 et la retenue $r2$ par 1. Le *Tracer* nous a permis d'identifier trois étapes importantes dans le raisonnement :

1. AliceTalks ne peut plus déduire de réduction de domaine par filtrage.
2. AliceTalks choisit une manipulation symbolique de la contrainte $[0 = (-1 + (-e + n))]$, qui consiste à substituer la variable n par l'expression $(1 + e)$ dans toutes les contraintes. La contrainte $[0 = 10 + e - r - r1 - n]$ est alors réécrite en $[0 = 9 - r - r1]$, par le système de réécriture décrit en section 3.3.
3. La contrainte $[0 = 9 - r - r1]$ est choisie. Elle est filtrée par le graphe qui en déduit $r = 8$ et $r1 = 1$.

Aucun choix d'instanciation ou retour-arrière n'est nécessaire pour trouver l'unique solution du problème.

4. Conclusion

La satisfaction de contraintes attire par ses nombreux algorithmes efficaces. Cependant nous avons identifié plusieurs situations dans lesquelles les techniques classiques de CSP se réduisent à une énumération systématique, tandis qu'un raisonnement formel sur les contraintes trouve la solution de manière plus directe. Il est donc intéressant de pouvoir combiner les deux stratégies, filtrage et raisonnement formel au cours d'une résolution. En reconstruisant le système Alice, dans un langage à base d'objets, nous avons pu mettre en évidence deux supports au raisonnement formel : les règles de production et les règles de réécriture. Toutes les contraintes sont traitées dans leur forme normale et les règles de production génèrent de nouvelles contraintes redondantes. Par ailleurs, la programmation par objets apporte un avantage certain : les modules sont clairement définis et indépendants. En particulier, notre système, AliceTalks, comprend un module de manipulation symbolique et numérique des expressions arithmétiques et logiques. Les systèmes de réécriture sont directement associés aux opérateurs d'expressions, ce qui accélère le processus de normalisation.

AliceTalks constitue ainsi un environnement de CSP compréhensible et adapté à l'expérimentation de nouvelles stratégies. Nous l'utilisons dans le but de spécifier les situations "favorables au raisonnement formel". Cependant un problème de performance se pose : l'application de règles reste un processus exponentiellement couteux, qu'il est nécessaire de contrôler. Dans cette optique, nos travaux futurs portent sur la comparaison de stratégies. Pour cela, nous voulons évaluer, pour une stratégie donnée, le potentiel de raisonnement formel intéressant, au cours de la résolution. La confrontation de ces résultats devrait permetttre d'avoir une meilleure connaissance de la stratégie à appliquer, en fonction de l'état de résolution du problème.

Références

[1] A. Aoki. Object-Oriented Analysis and Design Techniques. Software Research Center. Tokyo, Japan, 1994.

[2] C. Bessière and J.Ch. Régin. Arc-consistency for General Constraint Networks : Preliminary Results. *Proceedings of IJCAI'97, Nagoya, Japan*, vol. 1, pp. 398-404, août 1997.

[3] Y. Caseau. Constraint Satisfaction with an Object-Oriented Knowledge Representation Language. *Proceedings of JAIPS'93*,

1993.

[4] Y. Caseau and F. Laburthe. Improved CLP Scheduling with Task Intervals. *Proceedings of the 11th International Conference on Logic Programming*, MIT Press, 1994.

[5] C. Cornes, J. Courant, J-C. Filliatre, G. Huet, P. Manoury, C. Munoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi and B. Werner. The Coq Proof Assistant Reference Manual, version 5.10. INRIA, report 0177, 1995.

[6] N. Dershowitz and J.P.Jouannaud. Rewrite Systems. Handbook of Theoretical Computer Science "Formal Models and Semantics", vol. B, Elsevier, pp. 243-309, 1990.

[7] R. Haralick and G. Elliot. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, vol. 14, pp. 263-313, 1980.

[8] Ph. Jegou. Contribution a l'etude des problemes de satisfaction de contraintes: algorithmes de propagation et de resolution. Propagation de contraintes dans les reseaux dynamiques. Université de Montpellier 2, Montpellier, 1991.

[9] J.P. Jouannaud and P. Lescanne. La Réécriture. Techniques et Sciences informatiques, vol. 5(6), pp. 433-452, 1986.

[10] C. Kirchner, H. Kirchner and M.Vittek. Implementing Computational Systems with Constraints. *Proceedings of the first Workshop on Principles and Practice of Constraint Programming, Providence, USA*, Aris Kanellakis, Jean-Louis Lassez and Vijay Saraswat, vol. 1, pp. 166-175, 1993.

[11] J.L. Laurière. Un langage et un programme pour énoncer et résoudre des problèmes combinatoires. Ph. D. Thesis, University Pierre et Marie Curie, Paris, 1976.

[12] J.L. Laurière. A Language and a Program for Stating and Solving Combinatorial Problems Artificial Intelligence. *Artificial Intelligence*, vol. 10, pp. 29-127, 1978.

[13] J.L. Laurière. Intelligence artificielle, résolution de problèmes par l'homme et la machine. Eyrolles, Paris, 1986.

[14] J. Pitrat. Penser autrement l'informatique. Hermès, Paris, 1993.

[15] P. Prosser. Domain filtering can degrade intelligent backjumping. *Proceeding of the 13th. Joint conference on Artificial Intelligence*, Chambery, pp. 262-267, 28 août-3 septembre 1993.

[16] J.F. Puget. Programmation Par Contraintes Orientée Objet. *12th International Conference on Artificial Intelligence, Expert Systems and Natural Language*, Avignon, France, pp. 129-138, 1992.

[17] R. Roy and F. Pachet. Conception de problèmes par objets et contraintes. *Journées Francophones des Langages Applicatifs-JFLA'97*. INRIA, Collection didactique, Dolomieu, France, pp. 169-187, janvier 1997.

[18] H. Beringer and B. De Backer. Combinatorial problem solving in constraint logic programming with cooperative solvers. Logic Programming: formal methods and practical applications, ed. C.Beierle and L.Plumer, Elsevier, 1995.

[19] H. Hong. Confluency of cooperative constraint solvers. Technical report, Research Institute for Symbolic Computation, Johannes Kepler University. Linz, Autriche, 1994.

[20] E. Monfroy. Collaboration de solveurs pour la programmation logique à contraintes. Thèse de Doctorat. Université Henri Poincaré. Groupe Protheo, Nancy 1, novembre 1996.

[21] S. Abdennadher. Operational Semantics and Confluence of Constraint Propagation rules. *proceedings of CP'97*. LNCS, Springer-Verlag, vol. 1330, Linz, Autriche, pp. 252-266, 29 octobre-1 novembre 1997.