Un pilote de traceur pour la PLC

Déboguer, auditer et visualiser une exécution avec un même traceur

Ludovic Langevine* — **Mireille Ducassé****

* *INRIA Rocquencourt – BP 105 – F-78 153 Le Chesnay Cedex*

** *IRISA/INSA de Rennes – Campus de Beaulieu – F-35042 Rennes Cedex*

Ludovic.Langevine@inria.fr, Mireille.Ducasse@insa-rennes.fr

RÉSUMÉ. *Un traceur fournit des informations utiles sur les exécutions. Les contextes d'utilisation d'un traceur sont variés : embarqué dans une console interactive, source d'information d'un outil de visualisation, ou bien « espion » d'une surveillance automatisée du comportement de l'exécution (*monitoring*). Les traces nécessaires peuvent considérablement différer. Pouvoir interagir avec le traceur pour paramétrer les informations tracées est donc primordial. Dans cet article, nous proposons un mécanisme appelé* pilote de traceur. *Il transforme un même traceur en un frontal puissant pour des outils d'analyse variés, tout en contrôlant le coût de la génération de trace. Des motifs d'événements spécifient les événements pertinents et une large variété d'informations peut être tracée. Le pilote permet une utilisation aussi bien synchrone qu'asynchrone du traceur. Ce mécanisme a été conçu et expérimenté dans le cadre de la programmation logique par contraintes, avec Codeine, un traceur pour GNU-Prolog.*

ABSTRACT. *Tracers provide useful information about program executions. A tracer can be used for many purposes. Embedded in an interactive console, it enables users to investigate program executions. It can also be the front-end of a visualization tool, or it can be used by an automatic monitoring process. For each purpose, the requested data can be different. It is therefore important to be able to interact with the tracer in order to tune the sent data. In this paper we propose a so-called "tracer driver". From a single tracer, it provides a powerful front-end for multiple dynamic analysis tools while limiting the overhead of the trace generation. The tracer driver can be used both synchronously and asynchronously. The relevant execution events are specified by flexible event patterns and a large variety of trace data can be given either systematically or "on demand". The proposed tracer driver has been designed and experimented in the context of constraint logic programming, within GNU-Prolog.*

MOTS-CLÉS : *Trace d'exécution, traceur, débogage, mise au point, surveillance, programmation par contraintes, domaines finis.*

KEYWORDS: *Execution trace, tracer, debugging, tuning, monitoring, constraint programming, finite domains.*

Article extrait des actes des Journées Francophones de Programmation Logique et de programmation par Contraintes (JFPLC 2004, Angers, France), édités par Fred Mesnard et publiés par Hermès Science.

1. Introduction

Les traceurs fournissent des informations utiles sur les exécutions des programmes. Une trace d'exécution est une séquence d'événements d'exécution. Traditionnellement, les traceurs décident statiquement des événements observés. Les compromis sur la quantité d'information à tracer sont faits une fois pour toutes ou, au mieux, avant chaque exécution. Cependant, tant que l'objectif de la session de trace reste inconnu, il est difficile de décider précisément des informations à tracer. Lors du débogage, par exemple, l'utilisateur ne peut pas dire a priori quels détails seront nécessaires pour localiser son erreur. Si le traceur décide arbitrairement de ne pas tracer le détail de la propagation de contraintes sous prétexte que ces détails volumineux sont trop coûteux, de précieux indices pourront manquer à l'utilisateur à certains moments. De même, si un outil de visualisation de l'arbre de recherche n'a généralement pas besoin de connaître l'état des domaines à chaque nœud, cette information peut parfois être utile à l'utilisateur, par exemple sur les racines des sous-arbres d'échec, afin d'évaluer quel algorithme de filtrage aurait pu élaguer la branche. Il est donc important de pouvoir tracer à différents niveaux de détails, mais aussi de pouvoir modifier dynamiquement la granularité de la trace, ce qui nécessite d'interagir avec le traceur.

Dans la suite, nous appelons « analyseur » un module qui est connecté au traceur. Dans sa forme la plus simple, l'analyseur est une simple sortie de la trace dans un fichier (on parle de *traceur primitif*). Dans *les traceurs standards*, l'analyseur gère l'interaction avec l'utilisateur à chaque arrêt sur un événement d'intérêt (appelé point d'arrêt ou *breakpoint*). Il affiche quelques informations de trace et réagit aux commandes de l'utilisateur. Des outils plus sophistiqués nécessitent des analyseurs plus complexes. Un mécanisme de requête de trace semblable à l'interrogation d'une base de données peut ainsi être connecté au traceur et permettre à l'utilisateur d'examiner l'exécution d'une manière plus approfondi. C'est le cas de l'outil Opium pour Prolog [DUC 99]. Si l'efficacité n'est pas un soucis majeur, une véritable base de données peut même être utilisée, comme cela a été fait pour le système distribué Hy^+ [CON 94]. Des outils de surveillance peuvent aussi être connectés au traceur afin de superviser l'exécution et de récupérer des informations. Par exemple, Morphine est capable d'évaluer la qualité d'un jeu de test pour un programme Mercury [JAH 02]. L'outil EMMI, pour Icon, est capable de détecter certaines erreurs de programmation [JEF 94]. De nombreux outils de visualisation utilisent des traces pour produire des vues graphiques. Dans la communauté PLC, nous pouvons citer les outils Oz Explorer [SCH 97], Oz Constraint Investigator [MÜL 00], les outils issus du projet DiSCiPl [DER 00], le visualiseur « Christmas Tree » d'Ilog [BRA 01] ou encore CLPGUI [FAG 02]. D'autres exemples existent en dehors de la communauté PLC : DAQV propose des clients graphiques affichant différentes vues de programmes Fortran Haute Performance [HAC 98] et CCI offre une bibliothèque graphique pour visualiser les exécutions de programmes C [JEF 98].

Dans les exemples précédents, différents modes d'interaction entre traceur et analyseur apparaissent. Pour les traceurs primitifs et la visualisation simple, le traceur sort juste les informations de trace dans un canal de communication. Il n'y a pas de synchronisation entre traceur et analyseur. Les traceurs standards et les systèmes de requêtes, quant à eux, sortent la trace sur un canal (tel une console ou une *socket*) et gèrent les requêtes de l'utilisateur de manière totalement synchrone : l'exécution est bloquée dans l'attente d'une action de l'utilisateur. Jusqu'à présent, tous ces outils sont disjoints et leurs implantations actuelles difficiles à fusionner.

Mixer les différents modes d'interaction entre traceur et analyseur dans un seul et même outil a plusieurs avantages. Premièrement, l'utilisateur n'a pas besoin de passer d'un outil à l'autre pour réaliser différentes tâches de mise-au-point. Il utilise un même outil pour tracer, déboguer, auditer et visualiser l'exécution. Deuxièmement, l'intégration de tous ces usages dans un outil plus puissant offre plus que la simple juxtaposition des différents outils. Par exemple, on peut vouloir, en parallèle, collectionner des données à visualiser et rechercher des erreurs classiques de programmation. Si une erreur est trouvée, l'outil peut lancer une session de débogage synchronisée avec l'exécution. L'outil de visualisation peut également changer la granularité de la trace produite en fonction du contexte. Par exemple, il serait possible d'obtenir plus d'informations sur certains nœuds de l'arbre de recherche.

Dans cet article, nous proposons un « pilote de traceur ». À partir d'un seul traceur, il offre une interface puissante pour de multiples outils d'analyse dynamique, tout en limitant le coût de l'implantation de la génération de trace. Ses trois principales caractéristiques sont : 1^{O} les motifs permettant de spécifier la trace à générer ; 2^{O} le langage d'interaction entre le pilote et l'analyseur ; 3^{O} les mécanismes de filtrage à la volée de la trace. Les événements de traces ont des attributs sur lesquels les motifs posent des conditions, utilisant différents opérateurs logiques. Le langage d'interaction permet à la fois la communication synchrone et asynchrone entre le pilote et l'analyseur. À notre connaissance, la description du protocole d'un traceur offrant les deux types d'interactions est originale. Le filtrage des événements de trace en fonction des motifs est effectué à la volée. Une exécution génère communément plusieurs millions d'événements à la seconde, rendant le mécanisme de filtrage critique. Nous compilons un automate similaire à ceux évaluant les expressions booléennes dans les langages impératifs. Ces automates minimisent le nombre de conditions à vérifier [WIL 95].

Le pilote de traceur décrit dans cet article a été implanté dans Codeine, un traceur pour GNU-Prolog [LAN 03] tenant compte de la propagation de contraintes. Ce pilote a été expérimenté avec différents outils de débogage, audit et visualisation.

Cet article présente trois contributions. Premièrement, il identifie la nécessité d'un pilote de traceur afin d'intégrer des analyses dynamiques différentes au sein d'un même outil. La nécessité des communications synchrones et asynchrones y est plus particulièrement démontrée. Deuxièmement, il décrit en détail les mécanismes nécessaires à la mise-en-œuvre d'un tel pilote. Enfin, il montre que traceur primitif, traceur standard, requête de trace, audit et visualisation peuvent tous être implantés dans un tel cadre.

Dans la suite, la section 2 donne un aperçu du pilote de traceur et, en particulier, des interactions qu'il permet entre traceur et analyseur. La section 3 présente les mo-

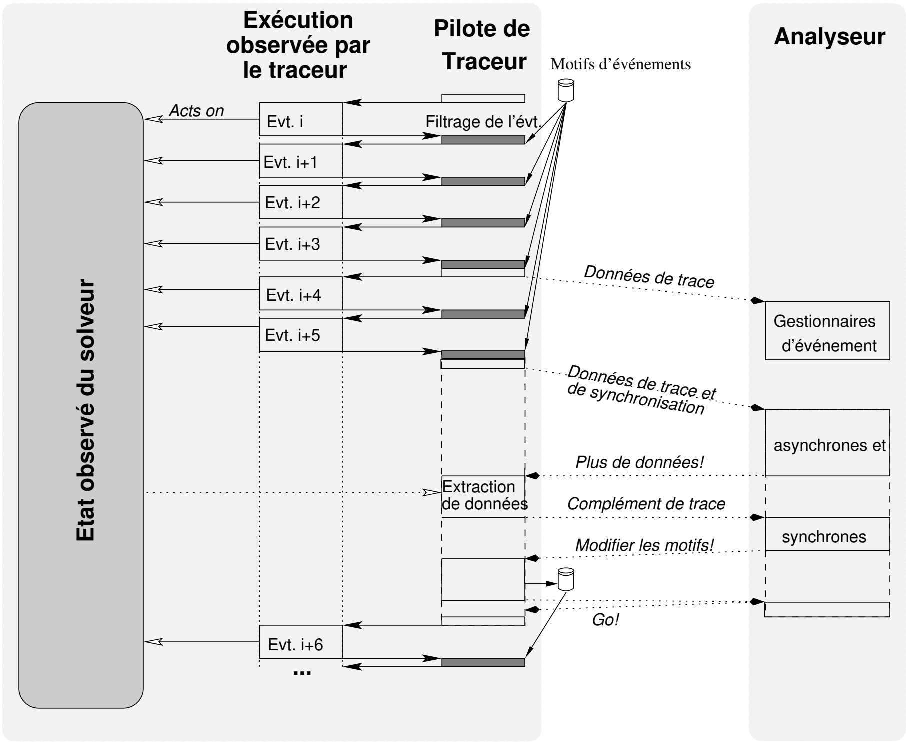

Figure 1. *Interactions synchrones et asynchrones entre traceur et analyseur*

tifs d'événements. La section 4 liste les requêtes qu'un analyseur peut envoyer à notre pilote, et comment elles sont prises en compte. La section 5 détaille le mécanisme de filtrage et son implémentation. La section 6 montre comment des mécanismes spécifiques d'analyse peuvent être implémentés dans notre outil. Enfin, la section 7 discute des travaux connexes.

2. Aperçu du pilote de traceur

Cette section décrit l'architecture du pilote de traceur et, en particulier, les interactions qu'il permet entre un traceur et un analyseur.

Comme mentionné dans l'introduction, les modes d'interactions synchrones et asynchrones sont tout deux nécessaires. Si l'analyseur a besoin d'un complément d'information sur un événement particulier, il faut que l'exécution soit bloquée tant que l'analyseur le souhaite. Au contraire, si l'analyseur désire seulement collecter des informations simples sur certains événements, il n'est pas utile de synchroniser l'exécution à chaque événement.

Une trace d'exécution est une séquence d'événements observés. Chaque événement a des *attributs*. L'analyseur spécifie quels événements il souhaite observer au moyen de *motifs d'événements*. Un motif d'événements est une condition sur les attributs d'un événement (cf. section 3). Chaque événement d'exécution est confronté à l'ensemble des motifs. Un événement est concordant pour un motif si et seulement si la condition du motif est vérifiée par les attributs de l'événement.

Un *motif asynchrone* spécifie que, à chaque événement concordant, certaines informations de trace doivent être envoyées à l'analyseur sans bloquer l'exécution. Un *motif synchrone* spécifie que, à chaque événement concordant, certaines informations de trace doivent être envoyées à l'analyseur, et que l'exécution reste bloquée, attendant les ordres de l'analyseur. Un *gestionnaire d'événements* est une fonction définie par l'analyseur et appelée à chaque événement concordant.

La fig. 1 illustre le traitement des deux types de motifs. L'exécution est présentée comme une séquence de blocs élémentaires (les événements d'exécution) agissant sur les données du solveur. À chaque événement, le pilote est appelé pour filtrer l'événement. Si l'événement courant ne satisfait aucun des motifs spécifiés, l'exécution se poursuit (évt. $i, i+1, i+2, i+4$ et $i+6$). Si l'événement courant satisfait un motif, des informations de trace sont envoyées à l'analyseur (évt. $i+3$ et $i+5$). Si le motif est asynchrone, l'analyseur traite la trace reçue de manière asynchrone (évt. $i+3$). Si le motif est synchrone, l'exécution est bloquée, attendant une requête de l'analyseur (évt. $i+5$). L'analyseur traite la trace reçue et peut demander des informations supplémentaires sur l'état de l'exécution. Le pilote peut récupérer des données utiles dans l'état du solveur à la demande. L'analyseur peut également ordonner l'ajout ou la suppression de motifs. Finalement, l'analyseur demande au pilote la reprise de l'exécution jusqu'au prochain événement pertinent (commande *go*).

L'architecture du pilote permet la gestion de plusieurs motifs, tous identifiés par un label. Un événement d'exécution peut satisfaire plusieurs motifs : la liste des labels des motifs satisfaits est envoyée avec les informations de trace. Ainsi, l'analyseur peut appeler le gestionnaire associé à chaque motif et distribuer les informations de trace à chacun. Si au moins un des motifs activés est synchrone, l'analyseur attend l'acquittement de chaque gestionnaire synchrone avant d'envoyer la commande *go*. Du point de vue d'un gestionnaire d'événements, l'activation simultanée d'autres gestionnaires pour un même événement est complètement transparente.

3. Motifs d'événements

Un motif d'événement est une condition portant sur les attributs des événements. Il se compose principalement d'une formule logique du premier ordre combinant des conditions élémentaires sur les attributs. Dans la suite, la section 3.1 résume le format des événements de trace et la section 3.2 spécifie le format des motifs d'événements. La section 3.3 donne quelques exemples de motifs.

3.1. *Événements de trace*

Le format concret de la trace n'a pas d'influence sur le principe du pilote de traceur. Seul compte le fait que les événements sont de différents types, ont des attributs et que certains attributs sont spécifiques au type d'événement. Toutefois, afin de faciliter la compréhension de la suite de l'article, cette section donne un aperçu des différents événements de trace utilisés dans notre prototype. Ce modèle de trace est défini formellement dans [LAN 03].

En programmation logique, le type des événements est appelé *port*. Il y a 14 ports différents dans notre traceur. Les événements de « contrôle » relatent la gestion du store et de la recherche. new variable introduit une nouvelle variable ; new constraint déclare une nouvelle contrainte ; post signale qu'une contrainte déclarée est introduite dans le store en tant que contrainte active ; new child déclare l'état courant du solveur comme un nouveau nœud de l'arbre de recherche ; quitter l'état courant du solveur pour revenir à un ancien point-de-choix est signalé par jump to ; solution déclare l'état courant du solveur comme solution (au sens de succès *toplevel*, en Prolog) ; failure déclare l'état courant comme inconsistant. Six ports décrivent les réductions de domaine et la propagation. Chaque réduction (reduce) est tracée et génère des événements-solveurs à propager. Quand la contrainte active ne peut plus réduire de domaine, elle est suspendue (suspend). Une contrainte active qui est détectée comme vraie est dite *impliquée par le store* (entail). Au contraire, une contrainte qui est détectée comme insatisfiable est dite rejetée (reject). Les événements-solveur sont traités à tour de rôle (schedule) : toutes les contraintes qui en dépendent sont réveillées (awake). Finalement, end of trace notifie la fin de la trace.

Des attributs communs et spécifiques sont associés à chaque événement. Les attributs communs sont : le port, le numéro d'événement (*chrono*), la profondeur du nœud courant dans l'arbre de recherche, l'état du solveur (domaines, store de contraintes et file de propagation), ainsi que le temps utilisateur écoulé depuis le début de l'exécution. Les attributs spécifiques de new variable sont l'identificateur de la variable et son domaine initial. Pour les ports décrivant l'arbre de recherche, le seul attribut spécifique est le label du nœud. Pour new constraint, post, suspend, entail et reject, le seul attribut spécifique est la contrainte courante. Pour le port reduce, les attributs spécifiques sont la contrainte courante, la variable réduite, les valeurs retirées et le type de réduction. Pour schedule, seul est ajouté l'événement-solveur à propager. Pour awake, sont ajoutés la contrainte réveillée et l'événement-solveur à propager.

La figure 2 présente le début de la trace d'un programme jouet, afin d'illustrer les différents types d'événements. Ce programme, `fd_element(I, [2,5,7],A), (A#=I ; A#=2)`, déclare `A` comme variable de domaine fini de valeur comprise dans $\{2, 5, 7\}$, `I` étant l'indice de la valeur de `A` dans la liste. De plus, `A` est égale soit à `I`, soit à 2. Seule la seconde option est possible. La trace se lit ainsi : les deux premiers événements déclarent les deux variables `v1` et `v2`, (respectivement `I` et `A`). En Gnu-Prolog, les variables sont créées avec un domaine maximum (de 0 à 268.435.455). La première contrainte est ensuite créée : `fd_element` (évt. 3). Cette contrainte fait

```
1 newVariable   v1 =[0-268435455]
2 newVariable   v2 =[0-268435455]
3 newConstraint c1  fd_element([v1,[2,5,7],v2])
4 reduce   c1   v1 =[1,2,3]   W=[0,4-268435455]
5 reduce   c1   v2 =[2,5,7]   W=[0-1,3-4,6,8-268435455]
6 suspend  c1
```

Figure 2. *Trace partielle de* `fd_element(I,[2,5,7],A), (A#=I; A#=2)`. *« a-b » signifie* de *a* à *b et « a,b » signifie a* et *b. Extrait de [LAN 03]*

motif	: :=	***label* : when** *formule op_synchro liste_actions*	
formule	: :=	*formule* **or** *formule*	**(1)**
		| *formule* **and** *formule*	**(2)**
		| **not** *formule*	**(3)**
		| (*formule*)	**(4)**
		| *cond_élémentaire*	**(5)**
op_synchro	: :=	**do** | **do_synchro**	
liste_actions	: :=	*action* **,** *liste_actions* | *action*	
action	: :=	**current**(*liste_attributs*) | **call**(*procédure*)	
cond_élémentaire	: :=	***attribut*** *op2* ***valeur*** | *op1*(***attribut***) | **true**	
op2	: :=	**< | > | = | \\= | >= | =< | in | notin**	
		| contains | notcontains	
op1	: :=	**isNamed**	

Figure 3. *Grammaire des motifs. En gras, les terminaux et en italique, les non-terminaux*

deux réductions de domaine (évt. 4 et 5) : le domaine de `I` devient $\{1, 2, 3\}$ et celui de `A`, $\{2, 5, 7\}$. Après ces réductions, la contrainte est suspendue (évt. 6). 32 autres événements (non reproduits ici) sont nécessaires pour trouver la solution `A=2` et `I=1`.

3.2. *Motifs*

La figure 3 présente la grammaire des motifs d'événements. Un motif contient quatre parties : un label, la formule du motif, un opérateur de synchronisation et une liste d'actions. La formule du motif est une composition de conditions élémentaires utilisant les conjonction, disjonction et négation logiques. Un opérateur de synchronisation spécifie si le motif est synchrone (**do_synchro**) ou bien asynchrone (**do**). Une action est soit le traçage de valeurs d'attributs (**current(***liste_d_attributs***)**), soit l'appel d'une procédure de l'analyseur **call(***procédure***)**. Notez que la procédure est écrite dans un langage que l'analyseur peut exécuter. Ce langage est indépendant du pilote de traceur. Une condition élémentaire porte sur un attribut d'événement. Il y a différents types d'attributs. Chaque type a un ensemble d'opérateurs associés permettant

```
visu_tree: when port in [choicePoint,backTo,solution,failure]
        do current(port=P and node=N and depth=D and utime=T),
            call search_tree(P,N,D,T)
visu_cstr: when port = post
        do current(cstr=C and cstrRep=Rep and varC(cstr)=VarC),
           call new_cstr(C, Rep, VarC)
visu_prop: when port = reduce and isNamed(var)
     and(not cstrType='assign') and delta notcontains([maxInt])
        do current(cstr=C and var=V),
           call spy_propag(C,V)
leaf:      when port in [solution, failure]
        do_synchro current(port=P and node=N and depth=D),
                   call new_leaf(P,N,D)
symbolic:  when port in [reduce,suspend]
     and(cstrType='fd_element_var' or cstrType='fd_exactly')
        do_synchro call symbolic_monitor
```

Figure 4. *Exemples de motifs pour la visualisation et le* monitoring

de construire les conditions élémentaires. Par exemple, la plupart des attributs sont entiers (chrono, profondeur, label de nœud). Les opérateurs classiques de comparaison peuvent être utilisés avec ces attributs ($=$, $\neq$, $<$, $\leq$, $>$ et $\geq$). L'attribut *port* est de type énuméré. Les opérateurs $=$ et $\neq$ peuvent être utilisés pour le port, ainsi que les opérateurs ensemblistes **in** et **notin**. Les solveurs de contraintes manipulent de nombreuses contraintes et variables. Souvent, une trace ne s'intéresse qu'à un petit sous-ensemble d'entre elles. Les opérateurs **in** appliqués aux identificateurs des entités ou aux noms des variables et contraintes peuvent spécifier de tels sous-ensembles. Les opérateurs **contains** et **notcontains** sont utilisés pour exprimer des conditions sur les domaines.

3.3. *Exemples de motifs*

Cinq motifs concurrents sont présentés en figure 4. Les trois premiers sont orientés visualisation : le premier permet la construction de l'arbre de recherche (point-de-choix, backtrack, échec et solution) avec quelques détails sur chaque nœud (label, profondeur et temps utilisateur). Cela permet une visualisation des sous-arbres pondérée par le temps passé dans leur parcours. Le second motif demande la trace de chaque ajout de contrainte avec l'identificateur de la contrainte (un entier unique), sa représentation (nom de la contrainte et paramètres), ainsi que la liste des variables impliquées (*varC*). Le troisième motif demande la trace de toutes les réductions de domaine faites par les contraintes qui ne viennent pas de la procédure de labeling et qui ne supprime pas la valeur maxInt. Il enregistre la contrainte active et la variable réduite. Ces données peuvent être utilisées pour calculer certaines statistiques et visualiser l'impact de chaque contrainte sur ces variables. Ces trois motifs sont asynchrones : les données demandées suffisent à la visualisation et les motifs n'ont pas à être modifiés. Le quatrième motif synchronise l'exécution à chaque feuille de l'arbre de recherche (solution ou échec) et appelle la fonction `new_leaf` pour interagir avec le traceur et examiner l'état de l'exécution. Le dernier motif est dédié à la surveillance de l'exécution. Il gèle l'exécution à chaque réduction de domaine due à une contrainte symbolique telle que « *element* » sur une liste de variables ou « *exactly* ». Ce motif permet la vérification des algorithmes de filtrage utilisés pour ces deux types de contrainte.

4. Aperçu de l'analyseur

L'analyseur traite la trace : il spécifie au pilote de traceur quels événements sont requis et peut exécuter des actions spécifiques à chaque type d'événement. L'analyseur peut mener plusieurs analyses en même temps. Chaque analyse a son but propre et utilise des éléments spécifiques de la trace. L'indépendance des analyses simultanées est assurée par un *médiateur* qui centralise la communication avec le pilote du traceur et distribue les informations de trace à toutes les analyses en cours, en fonction de leurs besoins.

4.1. *Primitives de requêtes*

Nous décrivons ici les requêtes que l'analyseur peut envoyer au pilote du traceur. Elles sont de trois types. Premièrement, l'analyseur peut demander des informations sur l'événement courant. Deuxièmement, l'analyseur peut modifier les motifs d'événements que le pilote prend en compte pour filtrer la trace. Nous parlerons de motifs *actifs* dans la suite. Enfin, l'analyseur peut notifier la fin d'une session de communication synchrone.

- `current` *`attributs`* : spécifie une liste d'attributs d'événement à extraire de l'état courant de l'exécution. Le traceur récupère les données demandées et les envoie sous forme d'une liste de paires $(attribut, valeur)$;
- `reset` : supprime tous les motifs actifs ;
- `remove` *`labels`* : supprime les motifs actifs dont les labels sont passés en paramètres ;
- `add` *`motifs`* insère dans les motifs actifs le motif d'événements spécifié en paramètre, conformément à la grammaire de la figure 3.
- `go` notifie au pilote du traceur la fin d'une session synchrone : l'exécution tracée reprend.

4.2. *Médiation d'analyses concurrentes*

L'analyseur est un processus ou module séparé du traceur. Il implante des gestionnaires d'événements pour les différents motifs. L'analyseur peut gérer plusieurs analyses simultanément. Les actions de ces analyses concurrentes ne doivent pas interférer. Afin d'assurer la cohérence des communications entre les différentes analyses et le pilote, nous avons adopté une architecture centralisée. Chaque analyse est un module séparé (aussi séparé que le langage d'implantation le permet) et le médiateur est le seul interlocuteur du pilote.

Le médiateur met en œuvre la communication avec le pilote du traceur. Il propose aux modules d'analyse une interface avec les primitives `add`, `remove` et `reset` présentées ci-dessus. Il décide de quand envoyer la requête `go`. Pour éviter la corruption du système de motifs d'événements, un nom de module est ajouté à chaque label de motif. Un module d'analyse donné ne peut supprimer ou réinitialiser que ses propres motifs. Le médiateur implante aussi le système de récupération de données, sous forme d'une bibliothèque de fonctions.

Un module d'analyse peut être un simple ensemble de fonctions à appeler en tant que gestionnaires d'événements, ou bien un processus léger (*thread*) dédié, suspendu dans l'attente d'un événement spécifique. La communication synchrone et asynchrone est envisageable dans les deux cas, la seule différence est le jeu de primitives pouvant être utilisées par le gestionnaire : un gestionnaire asynchrone ne peut pas utiliser le mécanisme de récupération de données, ni modifier ses motifs.

Quand le pilote du traceur envoie les données d'un nouvel événement, le médiateur reçoit la liste des motifs intéressés par l'événement et distribue ainsi les données de trace aux différents gestionnaires. Les gestionnaires d'événements synchrones sont appelés en premier tour à tour. Une fois les gestionnaires exécutés, le médiateur envoie la requête `go` pour que l'exécution reprenne. Un gestionnaire peut être une simple fonction dont le médiateur attend le retour, ou bien un *thread* dédié. Dans ce dernier cas, le *thread* est bloqué sur une file de messages où sont placés les événements à traiter. Le médiateur débloque le *thread* en envoyant dans cette file un message dans cette file avec toutes les données de trace nécessaires. Dans le cas d'un gestionnaire synchrone, le médiateur est bloqué à son tour, attendant l'acquittement (ou l'arrêt) du *thread*. Notez qu'une approche « multi-threadée » a un coût négligeable si les données de trace sont partagées et non pas copiées entre les *threads*.

5. Mécanisme de filtrage

Cette section décrit en détail le point critique du filtrage. À chaque événement d'exécution, ce mécanisme est appelé pour tester la pertinence de l'événement vis-à-vis des motifs actifs. Sachant que l'exécution d'un programme avec contraintes peut conduire à plusieurs millions d'événements par seconde, l'efficacité du filtrage est critique. La suite expose l'algorithme global du pilote de traceur, puis décrit les automates qui conduisent le filtrage des événements. Quelques optimisations sont présentées en troisième partie, avant de donner quelques exemples d'automates. La gestion incrémentale des patterns est également exposée. Enfin, le sixième paragraphe de cette section explique le calcul paresseux des attributs.

```
procedure tracerDriver(P : ensemble des motifs actifs)
  marqués ← ∅
  pour chaque p ∈ P faire
    si concorde(p) alors marqués ← marqués ∪ {p}
  fin pour
  T ← {donnes_requises(p) | p ∈ marqués}
  envoie_trace(T, label(marqués))
  si synchrones(marqués) ≠ ∅ alors
    notifie(synchrones(marqués))
    repète
      requête ← reçoit_de_analyseur()
      execute(requête)
    jusqu'à requête = go
  fin si
fin proc
```

Figure 5. *Algorithme du pilote de traceur*

Algorithme du pilote de traceur. À chaque événement d'exécution, le traceur est appelé, collecte certaines informations pour maintenir à jour ses structures de données, puis appelle son pilote dont l'algorithme est donné en figure 5. Le mécanisme de filtrage peut gérer de multiples motifs simultanément. Chaque motif concordant avec l'événement courant est marqué comme activé. Ensuite, le pilote calcule l'union des données requises par tous les motifs marqués. Cet ensemble de données est tracé. Si au moins un motif marqué est synchrone, un signal de synchronisation est envoyé à l'analyseur et le pilote écoute et exécute les requête de l'analyseur jusqu'à réception de la primitive **go**. Une fois la procédure du pilote exécutée, l'exécution reprend son cours jusqu'au prochain événement.

Motifs et automates. Le test de la concordance entre un événement et un motif est guidé par un automate où chaque état est décoré par une condition élémentaire avec deux transitions possibles (*vrai* ou *faux*). L'automate a deux états finaux, $\mathbf{true}$ et $\mathbf{false}$. Si l'état $\mathbf{true}$ est atteint, l'événement correspond au motif.

Chaque automate résulte de la compilation d'un motif d'événements. Cette compilation est spécifiée par la grammaire attribuée de la figure 6, similaire à la grammaire d'évaluation des expressions booléennes dans les langages impératifs (voir par exemple [WIL 95]). Cette grammaire minimise le nombre de conditions à vérifier. L'automate final de l'expression est la composition de sous-automates correspondent aux sous-expressions. Les attributs de la grammaire sont $entry$ (l'état initial de l'automate), $ifTrue$ (transition si vrai) et $ifFalse$ (transition si faux). $entry$ est synthétisé, $ifTrue$ et $ifFalse$ sont hérités. $\mathbf{true}$ et $\mathbf{false}$ sont les deux états finaux. Les états intermédiaires sont créés au niveau des conditions élémentaires par la fonction $\mathrm{new\_node}(C, E.IfTrue, E.IfFalse)$. La continuation $ifTrue$ (resp. $ifFalse$) de la condition est connectée à la continuation $ifTrue$ (resp. $ifFalse$) du nœud précédent. Des exemples d'automates sont donnés dans la suite.

(0) $E \to E'$
$E.entry \leftarrow E'.entry$
$E'.ifTrue \leftarrow \textbf{true}$
$E'.ifFalse \leftarrow \textbf{false}$

(1) $E \to E_1 \textbf{ or } E_2$
$E.entry \leftarrow E_1.entry$
$E_1.ifTrue \leftarrow E.ifTrue$
$E_1.ifFalse \leftarrow E_2.entry$
$E_2.ifTrue \leftarrow E.ifTrue$
$E_2.ifFalse \leftarrow E.ifFalse$

(2) $E \to E_1 \textbf{ and } E_2$
$E.entry \leftarrow E_1.entry$
$E_1.ifTrue \leftarrow E_2.entry$
$E_1.ifFalse \leftarrow E.ifFalse$
$E_2.ifTrue \leftarrow E.ifTrue$
$E_2.ifFalse \leftarrow E.ifFalse$

(3) $E \to \textbf{not } E_1$
$E.entry \leftarrow E_1.entry$
$E_1.ifTrue \leftarrow E.ifFalse$
$E_1.ifFalse \leftarrow E.ifTrue$

(4) $E \to (\ E_1\ )$
$E.entry \leftarrow E_1.entry$
$E_1.ifTrue \leftarrow E.ifTrue$
$E_1.ifFalse \leftarrow E.ifFalse$

(5) $E \to C$
$E.entry \leftarrow$
$\text{new\_node}(C, E.IfTrue,$
$E.IfFalse)$

Figure 6. *Grammaire attribuée pour générer les automates des motifs*

Optimiser le filtrage grâce au port. Comme mentionné en section 3, le port est un attribut spécial puisqu'il donne le *type* de l'action tracée. Cela a trois conséquences. Premièrement, un port correspond à des partie spécifiques du code du solveur où un appel à une fonction (une *prise*) du traceur a été placé. Par exemple, dans Codeine, il y a 5 prises pour le port reduce, placées dans 5 fonctions correspondant aux 5 façons possible de réduire un domaine. Deuxièmement, les attributs spécifiques dépendent du port. Troisièmement, une même utilisation de la trace n'exploitera généralement qu'un petit ensemble de ports, ou bien aura un traitement différencié des ports.

Le port est donc central dans la définition des motifs. Pour la plupart des motifs, une condition porte explicitement sur le port. Lorsqu'un événement arrive, il est inutile d'appeler le pilote du traceur si aucun motif n'est interessé par le port de cet événement. Ainsi, pour chaque port, un drapeau est fixé au niveau de la prise pour vérifier si le port correspondant nécessite l'appel au pilote, voire au traceur. Les drapeaux sont mis-à-jour à chaque modification des motifs d'événements. Ce mécanisme simple évite des appels inutilement coûteux au pilote.

Lors de l'ajout d'un motif, l'ensemble des ports pertinents est calculé. Différentes parties du motif peuvent concerner différents sous-ensembles de ports. Dans ce cas, le motif est coupé en sous-motifs où l'ensemble des ports pertinents est constant. Chaque sous-motif est compilé en un automate. Les automates sont ensuite liés à leurs ports respectifs. Cette compilation des dépendances permet d'éviter les conditions sur le port lors de l'évaluation des filtres.

Exemples d'automates. La figure 7 montre la représentation interne des motifs présentés en section 3.3. Les 14 ports sont représentés à gauche. Les ports non pertinents

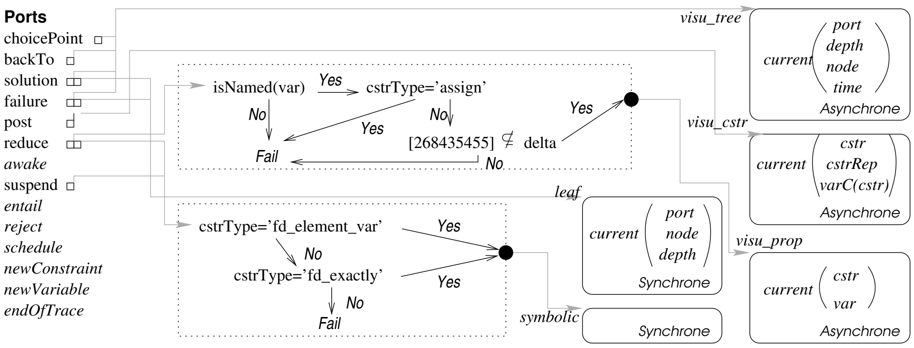

Figure 7. *Représentation interne de 5 motifs concurrents*

(aucun motif attaché) sont en italique. Les ports pertinents sont liés aux motifs correspondants. Seulement deux automates sont nécessaires puisque les conditions de trois des cinq motifs ne portaient que sur le port. Un ensemble d'actions est affecté à chaque motif. Cet ensemble est accompagné des mentions “synchrone” ou “asynchrone”, ainsi que du label du motif.

Gestion incrémentale des motifs. Puisque chaque motif actif est un automate spécifique (ou une liste d'automates s'il a été coupé), la primitive `add` a juste à compiler les n nouveaux motifs en m automates ($m \geq n$), les ajouter dans les listes chaînées des ports concernés et associer les labels aux automates. Au contraire, la primitive `remove` doit supprimer les automates associés aux labels spécifiés et mettre à jour les listes. Après chaque opération sur les motifs, les drapeaux associés aux ports sont mis-à-jour.

Calcul paresseux des attributs d'un événement. Pour vérifier une condition élémentaire, l'attribut impliqué doit être disponible. Certains attributs sont toujours calculés par le traceur : par exemple, le chrono et la profondeur sont deux variables entières du traceur qu'il suffit de lire. D'autres attributs doivent être récupérés dans l'état du solveur (par exemple, la contrainte active lors d'une réduction) voire recalculés à partir d'indices enregistrés par le traceur (par exemple, les valeurs retirées lors d'une réduction de domaine ne sont jamais calculées explicitement par le solveur mais peuvent être retrouvées). Pour ces attributs, le traceur est paresseux : sur un événement donné, il ne les calcule qui si le pilote les demande. Une fois calculés, il peuvent être réutilisés sans surcoût.

6. Mise-en-œuvre de mécanismes spécifiques d'analyse

Cette section montre comment des mécanismes spécifiques d'analyse de trace peuvent être mis en œuvre dans notre outil.

```
step :-
  reset,
  add([step:when true dosynchro call(tracer_toplevel)]),
  go.

skip_reductions :-
  current(cstr = CId and port = P),
  reset,
  (  P == awake
  -> add([sr:when cstr = CId and port in [suspend,reject,entail]
                 dosynchro call(tracer_toplevel)]),
  ;  add([step:when true dosynchro call(tracer_toplevel)])),
  go.
```

Figure 8. *Implantation de deux commandes d'un traceur standard*

Nous traitons en premier du traçage primitif et de la visualisation simple. Puis nous donnons les exemples d'implantation d'un traceur standard et de requêtes de trace. Enfin, nous donnons l'implantation d'un logimètre (*monitor*) utilisant le pilote de traceur.

Traçage primitif et visualisation simple. Les traceurs primitifs génèrent une trace par défaut dans un canal donné, afin d'alimenter un outil d'analyse. Ils sont purement asynchrones : une fois la trace par défaut spécifiée, l'exécution se déroule jusqu'à sa fin tout en traçant tous les événements d'intérêt avec un sous-ensemble de leurs attributs. Par exemple, Codeine a été adaptée pour générer des fichiers XML de trace d'exécution, chacun de taille conséquente (plusieurs centaines de méga-octets). Au début de l'exécution, une requête fixe quelques motifs asynchrones (un motif par port d'intérêt). La même version a été utilisée pour alimenter Pavot[1], un ensemble d'outils de visualisation pour la programmation par contraintes développés à l'Inria Rocquencourt.

Traceurs standard. Les traceurs standard exécutent un programme pas à pas. À certains pas, des informations sur l'événement courant sont affichées dans la console et l'utilisateur peut interagir avec le traceur grâce à des commandes simples. Dans un tel traceur, l'analyseur est une sous-routine du traceur. L'analyse est synchronisée avec l'exécution. Le traceur présente une *trace par défaut* qui peut être exprimée par des motifs d'événements synchrones au début de la session de trace. Chaque événement pertinent est affiché avec un sous-ensemble d'attributs. Une commande égale à la primitive `current` permet de récupérer des informations. Les commandes du traceur autorisent la modification des motifs actifs.

L'implantation de deux commandes dans notre approche est illustrée par la figure 8. La commande qui permet d'aller directement au prochain événement, `step`, réinitialise juste les motifs et en ajoute un qui appellera de manière synchrone la console du traceur à chaque événement d'exécution. Une autre commande possible,

1. http://contraintes.inria.fr/~arnaud/pavot/

```
fget(Pattern):-
       reset,
       add([opium:when Pattern dosynchro resume(fget_ack)]),
       fget_do(Event).
fget_do :-
       wait(fget_ack).
fget_do :-
       fget_do.
```

Figure 9. *Implantation d'une primitive de requête de trace resatisfiable*

```
void record_leafs()
{
  reset_solution_table ();
  reset_failure_table ();
  Add("collect:when port in [solution,failure] do
       current(chrono,port),call(rec_leaf)");
}

void rec_leaf(EventAttr *event)
{
  if (event->port == Solution)
    add_solution(event->chrono);
  else add_failure(event->chrono);
}
```

Figure 10. *Implantation d'un moniteur enregistrant tous les échecs et solutions*

`skip_reductions`, permet de passer sous silence les détails des réductions lorsqu'une contrainte est réveillée. Elle vérifie tout d'abord le port : s'il s'agit de awake elle demande d'aller à la prochaine suspension de la contrainte. Si elle est appelée sur un événement d'un autre type, elle agit comme `step`.

Primitives de requête de trace. La principale spécificité d'un système de requête de trace comme Opium [DUC 99] est qu'il s'exécute en coroutine avec le traceur, tandis qu'un traceur standard appelle l'analyseur (sa console) comme une simple sous-routine. Ainsi, l'exécution d'une requête de trace peut se poursuivre sur plusieurs événements d'exécution, tandis que la portée d'un traceur standard est d'un seul événement : l'événement courant. La primitive centrale d'Opium est `fget`, qui va au prochain événement satisfaisant un motif donné. `fget` est resatisfiable comme illustré en figure 9. Pour intégrer Opium dans un cadre plus général, nous voulons autoriser d'autres modules d'analyses concurrents, avec leurs motifs respectifs : la console Opium est un *thread* séparé, bloqué dans l'attente d'un événement d'exécution pertinent (grâce à l'opération *wait* sur un sémaphore). L'événement attendu doit satisfaire le motif « resume » qui demande au médiateur de débloquer le *thread*. Le médiateur attend à son tour le blocage ou la fin du *thread* Opium. Comme le motif est destiné à être spécifié à la volée, le modèle original d'Opium autorise un seul motif à la fois. Avec notre approche, plusieurs motifs pourraient être pris en compte sans surcoût d'implantation.

Moniteurs. Morphine [JAH 02] permet la surveillance de programmes écrits en Mercury [SOM 96]. Il offre une primitive de haut niveau basée sur un traceur. La primitive de surveillance, appelée `foldt`, est un « `fold` » opérant sur une liste d'événements et gérant un accumulateur. Un moniteur `foldt` a trois parties : l'initialisation de l'accumulateur, le « collect » (mise-à-jour de l'accumulateur à chaque événement pertinent) et le post-traitement (finalisation de l'accumulateur). Il faut remarquer que l'accumulateur est en fait n'importe quel objet Mercury, l'opération `collect` pouvant utiliser la pleine puissance de ce langage. Des accumulateurs complexes peuvent ainsi être développés. La procédure `collect` peut récupérer n'importe quelle donnée de l'état observé. Notre approche permet de réaliser de tels moniteurs. La figure 10 donne l'exemple d'un moniteur comptant le nombre de *backtracks* d'une exécution. Dans cet exemple, il n'est pas besoin de synchronisation. L'efficacité de la surveillance dépend du coût d'un appel à "collect". L'appel d'une procédure locale au pilote du traceur serait ici intéressant afin d'éviter les deux changements de contexte entre processus tracé et processus de surveillance. La synchronisation inter-processus est alors remplacée par un simple appel de fonction au sein d'un même processus.

7. Travaux connexes

Kraut [BRU 83] implante une machine à états finis pour détecter des séquences d'événements d'exécution correspondant à certains motifs, appelés *règle de chemin* (*path rules*). Plusieurs motifs sont autorisés and ils peuvent être activés ou désactivés durant l'exécution, grace à un système de labels. Lorsqu'une règle est satisfaite, les actions correspondantes sont déclenchées. Cependant, ces actions sont limitées à quelques primitives de débogage, telles que l'affichage d'un message ou l'incrémentation d'un compteur. Le principal intérêt de cet outil est l'abstraction de la trace (une séquence d'événements génère un événement de plus haut niveau). L'analyse de la trace est nécessairement synchrone et ne bénéficie pas de la pleine puissance d'un langage de programmation.

Dalek [OLS 90] est une puissante extension de *gdb*. Il permet à l'utilisateur d'associer à des séquences d'événements d'exécution des gestionnaires synchrones écrits dans un langage impératif dédiés. Ce langage inclut des primitives de récupération de données additionnelles et de synchronisation de l'exécution. La gestion des gestionnaires n'est pas incrémentale. Une fonctionnalité clé de Dalek, est la "file d'événements" qui contient les événements d'exécution passés. L'utilisateur peut explicitement retirer des événements de cette file et ajouter des événements de plus haut niveau. Cette approche requiert un stockage coûteux d'une partie de la trace mais permet à la fois la surveillance, le débogage et le profilage d'exécutions.

Dans EBBA [BAT 95], des relations entre événements d'exécution modélisent les comportements attendus du programme. Ces relations sont ensuite comparés avec le comportement réel de l'exécution. EBBA tente de reconnaître des séquences pertinentes d'événements et de vérifier certaines contraintes sur ces séquences. Un automate est construit afin de trouver les instances des modèles. Les événements sont d'abord tracés avant d'être filtrés par l'automate. Notre approche permet de filtrer les événements directement dans le traceur, mais la différence majeure avec EBBA est que leur automate est utilisé pour reconnaître des séquences d'événements, et non des événements seuls. En ce sens, notre approche pourrait être utilisée en amont de la reconnaissance de séquences. La gestion incrémentale des motifs peut être utilisée pour adapter les événements pertinents aux différents états de l'automate.

Ces outils reconnaissent des séquences d'événements et non des événements isolés, ce qui peut suffire au débogage d'un langage non déclaratif (recherche d'une séquence d'événements symptôme d'une erreur). Mais les séquences d'événements ne sont pas les seules abstractions possibles d'une trace, et notre outil propose plutôt de filtrer une trace volumineuse en amont d'un outil d'analyse. Toutefois, la reconnaissance efficace des séquences est prévue pour une extension future du pilote.

CLPGUI [FAG 02] et Oz Explorer [SCH 97] permettent d'interagir avec le solveur (ajout de contrainte, recherche interactive ou recalcul). Ces fonctionnalités supposent que l'outil soit synchronisé avec le solveur. Cette synchronisation peut être effectuée par le pilote de traceur mais l'interaction et le traçage sont en fait deux questions disjointes : un traceur ne doit pas altérer le déroulement de l'exécution. Le pilote de traceur aurait donc à rediriger les demandes d'interactions vers un « pilote de solveur ». L'implantation du solveur peut limiter les événements où ces interactions sont possibles. Un travail est en cours pour exécuter CLPGUI avec Codeine : Codeine pilote la trace fournie aux visualiseurs de CLPGUI, et CLPGUI récupèrent les interactions à certains ports.

8. Conclusion

Dans cet article, nous avons présenté un pilote de traceur qui permet des analyses de trace synchrones et asynchrones, indifféremment dans la même exécution. Un même outil comble ainsi tous les besoins des utilisations classiques d'un traceur. Nous avons défini un langage de motifs d'événements expressif qui décrit les événements pertinents par des formules logique du premier ordre accédant à la plupart des données auxquelles le traceur a accès. Des primitives spécifiques permettent de récupérer à la demande de nombreuses informations, mais aussi d'adapter les motifs d'événements aux évolutions des besoins de l'analyseur.

Ce travail a été en partie financé par le projet RNTL OADymPPaC[2] et par la région Île-de-France. Les auteurs souhaitent remercier les partenaires du projet pour les discussions fructueuses qu'ils ont eues, ainsi que Guillaume Arnaud et Pierre Deransart pour leur utilisation assidue de Codeine et leurs nombreuses suggestions.

2. http://contraintes.inria.fr/OADymPPaC/

9. Bibliographie

[BAT 95] BATES P. C., « Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior », *ACM Trans. on Computer Systems*, vol. 13, n° 1, 1995, p. 1–31.

[BRA 01] BRACCHI C., GEFFLOT C., PAULIN F., « Combining Propagation Information and Search-Tree Visualization using OPL Studio », KUSALIK A., Ed., *Proc. of WLPE'01*, Cyprus, Décembre 2001, Computer Research Repository cs.PL/0111040, p. 27–39.

[BRU 83] BRUEGGE B., HIBBARD P., « Generalized Path Expressions : A High-Level Debugging Mechanism », *The Journal of Systems and Software*, vol. 3, 1983, p. 265-276.

[CON 94] CONSENS M., HASAN M., MENDELZON A., « Visualizing and Querying Distributed Event Traces with Hy+ », LITWIN W., RISCH T., Eds., *Applications of Databases, First International Conference*, Springer, LNCS, vol. 819, 1994, p. 123-141.

[DER 00] DERANSART P., HERMENEGILDO M., MALUSZYNSKI J., Eds., *Analysis and visualization tools for constraint programming*, vol. 1870 de *LNCS*, Springer-Verlag, 2000.

[DUC 99] DUCASSÉ M., « Opium : An extendable trace analyser for Prolog », *The Journal of Logic programming*, vol. 39, 1999, p. 177-223.

[FAG 02] FAGES F., « CLPGUI : a generic graphical user interface for constraint logic programming over finite domains », TESSIER A., Ed., *Proc. of WLPE'02*, Copenhagen, Aug 2002, Computer Research Repository cs.SE/0207048.

[HAC 98] HACKSTADT S., MALONY A., « DAQV : Distributed Array Query and Visualization Framework », *Theoretical Computer Science*, vol. 196, n° 1–2, 1998, p. 289–317.

[JAH 02] JAHIER E., DUCASSÉ M., « Generic Program Monitoring by Trace Analysis », *Theory and Practice of Logic Programming*, vol. 2, n° 4-5, 2002, p. 611-643.

[JEF 94] JEFFERY C., GRISWOLD R., « A framework for execution monitoring in Icon », *Software-Practice and Experience*, vol. 24, n° 11, 1994, p. 1025-1049.

[JEF 98] JEFFERY C., ZOU W., TEMPLER K., BRAZELL M., « A lightweight architecture for program execution monitoring », *Workshop on Program Analysis for Software Tools and Engineering*, ACM SIGPLAN, June 1998.

[LAN 03] LANGEVINE L., DUCASSÉ M., DERANSART P., « A Propagation Tracer for Gnu-Prolog : from Formal Definition to Efficient Implementation », PALAMIDESSI C., Ed., *Proc. of the 19th Int. Conf. in Logic Programming*, Springer-Verlag, LNCS 2916, December 2003.

[MÜL 00] MÜLLER T., « Practical Investigation of Constraints with Graph Views », DECHTER R., Ed., *Proc. of the 6th Int. Conf. on Principles and Practice of Constraint Programming*, vol. 1894 de *LNCS*, Springer-Verlag, 2000, p. 320–336.

[OLS 90] OLSSON R., CRAWFORD R., HO W., « Dalek : A GNU, Improved Programmable Debugger », *Proc. of the Summer 1990 USENIX Conference : June 11–15, 1990*, 1990, p. 221–232.

[SCH 97] SCHULTE C., « Oz explorer : a visual constraint programming tool », NAISH L., Ed., *Proc. of the 14th Int. Conf. on Logic Programming*, MIT Press, 1997, p. 286-300.

[SOM 96] SOMOGYI Z., HENDERSON F., CONWAY T., « The execution algorithm of Mercury, an efficient purely declarative logic programming language », *Journal of logic programming*, vol. 29, 1996, p. 17-64.

[WIL 95] WILHELM R., MAURER D., *Compiler design*, Addison-Wesley, 1995.