DEBOGUEURS SYMBOLYQUES POUR LANGAGES IMPERATIFS

Roche Michaël Pinelli Michaël

Juin 2004

Licence Informatique

Table des Matières

INTODUCTION

Le but de ce projet est de faire un tour d'horizon des débogueurs pour les langages impératifs existant. Les débogueurs sont des auxiliaires précieux pour éliminer les erreurs dans les programmes. Au premier abord, déboguer un programme consiste à l'arrêter sous certaines conditions pour examiner l'état de la pile d'appels et les valeurs stockées dans les variables. On arrête l'exécution en insérant des points d'arrêt (breakpoints). Ces points d'arrêt peuvent être inconditionnels (ils engendrent toujours un arrêt du programme lorsqu'ils sont rencontrés) ou conditionnels (ils arrêtent le programme seulement si une condition est vérifiée, condition spécifiée par l'utilisateur).
Un débogueur est dit complet s' il possède les commandes suivantes :

- Arrêt sur une ligne ou dans une fonction, et gestion des points d'arrêt
- Exécution pas à pas sans entrer dans le code des fonctions
- Exécution pas à pas en entrant dans le code des fonctions
- Continuer l'exécution jusqu'au prochain point d'arrêt
- Lancer une nouvelle exécution
- Afficher la valeur des variables
- Gérer l'affichage des différents fichiers sources

Ensuite, nous nous intéresserons plus particulièrement aux débogueurs qui permettent de visualiser la mémoire alors qu'un programme s'exécute. Cela permet de voir l'adresse des variables et leurs valeurs a tout moment dans l'exécution du programme. De plus, afin de permettre une meilleure visualisation des variables (plus intuitive), nous avons sélectionné essentiellement des débogueurs graphiques.

Dans un premier temps nous allons définir ce qu'est un débogueur, à quoi ça sert, puis nous ferons un petit tour d'horizon de quelques débogueurs pour les langages impératifs. Enfin Nous présenterons plus particulièrement 2 d'entre eux, GDB sous interface DDD et Deet.

I) Qu'est ce qu'un débogueur ?

Lorsque l'on crée un programme, il se peut qu'il compile mais ne s'exécute pas, ou alors, ne fait pas ce qu'on attends de lui. En effet, l'erreur du programme n'est pas forcément syntaxique ou lexicale mais sémantique, c'est surtout dans ce cas que nous avons besoin d' un débogueur afin de trouver plus facilement d'où vient le problème .
Si on a un programme sans erreur, compilé, que l'on peut exécuter, ceci ne veut pas forcément dire que le programme en question se comportera comme prévu; il se peut que l'exécution du programme renvoie des résultats aberrants, ou bien qu'elle ne renvoie pas de résultat du tout (c'est le cas par exemple si le programme comporte une boucle infinie). Ceci signifie que le programme est correct du point de vue de la « langue » , mais qu'il y a un problème au niveau du « sens » (son comportement) : c'est comme si on écrivait une phrase correcte du point de vue de l'orthographe et de la grammaire, mais qui n'exprime pas l'idée que l'on veut faire passer.

Un débogueur sert à inspecter l'exécution d'un programme qui ne se comporte pas comme prévu, afin d'isoler le ou les points où le programme ne fait pas ce que l'on désire. L'idée pour ce faire est de figer l'exécution d'un programme à des moments bien choisis, et d'inspecter l'état dans lequel il se trouve (i.e. quelle est la valeur des variables à ces instants).
La bonne utilisation d'un débogueur consiste à bien choisir ces points d'arrêts, en fonction de la structure du programme.

Un débogueur est un programme, généralement écrit en un langage de bas niveau (C ou assembleur) qui va permettre d'exécuter un programme tout en pouvant l'observer, et quelque fois le modifier. Aussi, si un programme crash, le débogueur montre l'endroit dans le code où le programme s'est arrêté . Cependant ils ne se limitent pas à cela, ils offres plusieurs fonctions permettant d'exécuter un programme pas à pas, ou d'arrêter l'exécution à un endroit précis du code . Cela permet de visualiser l'état des variables tout au long de l'exécution, l'état de la mémoire et quelque fois même certains débogueurs permettent de modifier les valeurs des variables.

Il existe deux sortes de débogueur, les débogueurs textuels et les débogueurs graphiques. Bien entendu les second sont plus simple d'utilisation, plus intuitifs et aussi plus performant car reposant le plus souvent sur des débogueurs textuels. Il y a bien une légère perte de rapidité mais ce n'est pas le point le plus important de ces programmes. Il y en a pour a peut prêt tous les langages (C, C++, java, fortran, perl, etc.) qui sont plus ou moins performant. D'une façon générale, les langages de programmation à niveau élevé, tels que Java, facilitent la correction, parce qu'ils ont des dispositifs tels que la manipulation d'exception qui facilitent le repérage des erreurs. Dans des langages de programmation de bas niveau tels que C ou assembleur, les bogues peuvent poser des problèmes silencieux tels que la corruption de mémoire, et il est souvent difficile de voir où le problème initial s'est produit; dans ces cas, les outils de correction sophistiqués peuvent être nécessaires.

Les débogueurs facilitent la création de programme car ils permettent un gain de temps précieux pour le programmeur. Faire la trace d'un programme entier, à la main, est souvent long et fastidieux. Les débogueurs rendent le procédé de trace beaucoup plus facile et plus rapide permettant d'arrêter le programme la où on le souhaite afin de faire afficher les valeurs et les adresses des variables voulues. Bref les débogueurs font tout se dont le programmeur a besoin. Par exemple grâce au breakpoint il n'est pas obligé de regarder tout le programme s'exécuter ligne par ligne, mais il peut l'arrêter exactement la où il le veut dans le code afin de trouver plus rapidement encore l'erreur.

Afin de montrer une utilisation simple et abstraite d'un débogueur, voici un programme sur lequel son utilisation permet de visualiser et de corriger les erreurs :

```
Début : Afficherligne(ligne)
        entier lgligne <-- longueur(ligne);
        si(vrai)
                tantque (ligne <--l   igne-1)
                        afficherCaractere(ligne[lgligne]);
                fintantque
```

```
        afficherCaractere(Saut de ligne)
        sinon afficher(ligne);
        finsi
fin
```

Ce programme prend une ligne de caractère en paramètre et affiche la ligne à l'envers. Lors de la compilation aucune erreur n'est constatée par le compilateur, or lors de l'exécution rien ne se passe comme prévu. De ce fait grâce à l'utilisation d'un débogueur pour ce langage abstrait, on ré exécute ce programme sous contrôle de cet utilitaire qui permet de vérifier le déroulement du programme et de consulter le contenu des variables, et cela nous permet de constater que c'est la valeur de lgligne qui doit décroître dans la boucle « tantque », pour faire afficher du dernier caractère au premier. En effet, l'usage de ce débogueur abstrait a permis de mettre l'erreur en évidence et a servis au programmeur de la changer afin de permettre le bon fonctionnement de son programme.

II) L'état de l'art

Dans cette partie nous allons faire un petit tour d'horizon des débogueurs. Nous n'avons pas pu décrire dans cette partie tous les débogueurs existants, ils sont trop nombreux. C'est pourquoi nous nous sommes limité aux débogueurs ayant le plus de documentation disponible (ce qui a exclus les débogueurs payant), et aussi bien sur les débogueurs pour langage impératif pour rester dans le sujet. Nous nous sommes efforcés de choisir des débogueurs permettant de visualiser l'état des variables en mémoire.

a) Débogueurs non-graphiques

1. DBX

DBX est un débogueur non graphique pour les langages C et fortran .

Fonctionnalités:

- Les breakpoints ne sont pas obligatoirement placés à une ligne spéciale. Ils peuvent aussi arrêter l'exécution lorsqu'on appelle une certaine procédure.
- Un historique des commandes déjà effectuées est disponible.
- Peut lister les appelles de procédure déjà effectués (afficher la pile des procédures).
- Permet aussi de garder un oeil sur une variable, c'est à dire regarder sa valeur tout au long de l'exécution du programme. Cependant cela ralenti quand même un peut l'exécution puisque la variable doit être réexaminé à chaque ligne du code source.

Dbx est compatible avec Emacs et peut être invoqué dans celui-ci. Cela permet d'afficher le code source lors du débogage et permet une plus grande clarté du débogage.

Néanmoins malgré la facilité d'utilisation de ce débogueur il ne permet pas de voir réellement l'état de la variable en mémoire. Mais nous l'avons énoncé ici car c'est un débogueur textuel pratique qui fait le minimum de ce que l'ont peut attendre d'un tel programme.

2) COCA

COCA est un débogueur non graphique automatiser pour le langage C, ou le mécanisme des points d'arrêt est basé sur des événements relatif à des constructions du langage.
Ces événements ont un sémantique alors que les lignes de codes de la plupart des débogueurs n'en ont pas. Le langage d'interrogation de trace est prolog, augmenté par une poignée de primitive, Coca ne nécessite aucun stockage, l'analyse est faite a la volée. Coca est donc plus puissant que les débogueurs dont les points d'arrêt sont des lignes de code source et plus efficace que les débogueurs relationnels.
Fonctionnalité:

- Coca propose des breakpoints qui sont basés sur l'événement lié à la construction du langage,
- il propose un mécanisme de trace ou le contrôle des flots et de données sont recherché pendant l'exécution de la trace,
- Un autre avantage est la flexibilité, car il est très facile de déclarer un autre type d'événement juste pour introduire une nouvelle relation,

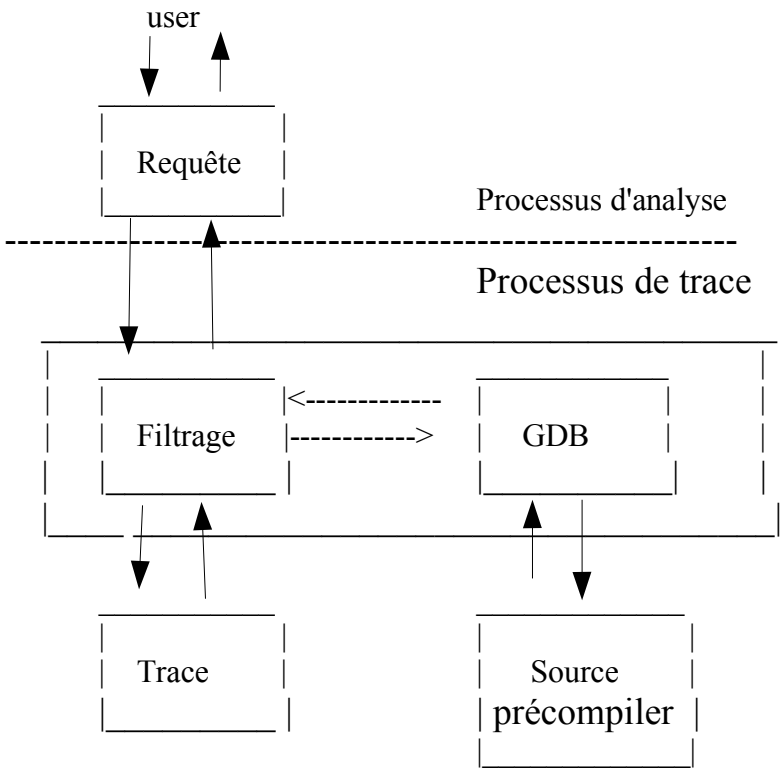
La trace est constitué d'événement qui correspond à la construction de C, par exemple:

Syntax	**Event name ::=Event component**
For(EXPR1;EXPR2;EXPR3)	FOR_EV ::= (enter, for) INIT_EV1 TEST_EV2 {BLOCK_EV INCR_EV3 TEST_EV2}* (exit, for)

Les variables sont définies dans une structure qui est elle même définie de la manière suivante:

- name: description de la variable
- type: type de variable du langage C
- val: la valeur de la variable
- addr: l'adresse de la variable en mémoire
- size: la taille de l'adresse en mémoire
- linedecl: la ligne où la variable est déclarée
- filedecl: le fichier où la variable est déclarée

Voici une figure représentant l'architecture du système de Coca:



Le débogueur est constitué de trois modules distincts:
- l'extraction avec la recherche des traces
- analyse qui montre le mécanisme de la trace et le niveau de l'exécution
- visualisation

En fait, Coca est une analyse de trace pour les programmes C. Il utilise le contrôle des flots et des données en même temps, donc est plus efficace que les débogueurs rationnels car il n'a pas besoin d'information pour être charger à l'exécution. De plus, les performances de Coca sont raisonnable car:
- les breakpoints sont utilisés pour restreindre la recherche des événements
- seulement les informations nécessaires sont recherchées
- la communication entre processus est réduite par une procédure filtrante

3) LADEBUG

Ladebug est un débogueur pour les programmes écrit en C, C++ ou Fortran (77 et 90). Il permet le choix d'utilisation graphique ou non. Cependant il ne fonctionne que sur les systèmes :

- Compaq Tru64™ UNIX systems
- Linux pour Compaq Alpha™ systems

Ladebug fonctionne aussi sur les programmes COBOL et Ada mais de façon limité. C'est un débogueur de bas niveau très complet.

- il gère tout a fait les threads permettant de visualiser leur état, leur sous-état, leur priorité. Il permet aussi de voir quel signal les a interrompus.
- Il permet de visualiser la pile des appels de procédure avec l'adresse mémoire de chacune d'elle. On peut monter et descendre dans la pile pour examiner le code source exécuté pour chaque appel.
- Il est aussi possible de créer des alias pour les commandes complexe que l'on veut faire exécuter par le débogueur, ainsi que des commandes composées ou conditionnelles.
- Il peut lister tous les breakpoints d'un programme, les désactiver et les supprimer.
- Il permet l'affichage des variables et de leur valeur bien sur, mais aussi de leur état en mémoire, ou bien leur type . L'affichage des registres est aussi possible. On peut aussi grâce à Ladebug lire une case mémoire arbitraire afin de voir son état.

Bref Ladebug est un débogueur très complet, permettant réellement de voir la mémoire. De plus il permet de faire tourner le programme dans tous les sens, en remontant dans les appels de procédure. Cependant il ne marche que sous certains systèmes bien spécifiques et est assez difficile d'utilisation pour un utilisateur non expérimenté. Malgré tout cela, il reste un très bon débogueur qui mérite qu'on s'y intéresse lorsqu'on veut pouvoir voir comment marche le programme au niveau même de la mémoire.

Voilà un petit exemple de ce que peut faire Ladebug :

```
(ladebug) list 59: 2
        59 const unsigned int biggestCount = 10;
        60 static Moon *biggestMoons[biggestCount];

(ladebug) print biggestMoons

        [0] = 0x1400043c0,[1] = 0x140004720,[2] = 0x140004420,[3] = 0x140004300,[4] =
        0x140004120,[5] = 0x140004360,[6] = 0x140004ae0,[7] = 0x1400049c0,[8] =
0x1400046c0,[9] = 0x140004a20

(ladebug) print biggestMoons[3]

        0x14000430

(ladebug) print *biggestMoons[3]
        class Moon {
          _radius = 1815;
          _name = 0x1200020b0="Io";       // class Planet::HeavenlyBody
          _innerNeighbor = 0x0;           // class Planet::HeavenlyBody
          _outerNeighbor = 0x140004360;   // class Planet::HeavenlyBody
          _firstSatellite = 0x0;          // class Planet::HeavenlyBody
          _lastSatellite = 0x0;           // class Planet::HeavenlyBody
          _primary = 0x1400042a0;         // class Planet::Orbit
          _distance = 422;                // class Planet::Orbit
          _name = 0x140005a80="Jupiter 1";              // class Planet::Orbit
        }
```

Ici nous affichons un tableau ainsi que la valeur individuelle d'un tableau.

4) JDB

Débogueur java non graphique. Il est écrit en java puisqu'il s'appuie sur Java Platform Debugger Architecture (définit plus bas).
fonctionnalités :
Il a toutes les fonctionnalités standard d'un débogueur et fournit en plus :
- l'affichage des valeurs des objets . Pour celui-ci une courte description est affichée. En utilisant la commande dump de jdb on affiche la valeur courante de chaque champs de l'objet.
- la faculté de lister les threads et d'afficher leur nom et leur statut.
- l'affichage de la pile des fonctions .

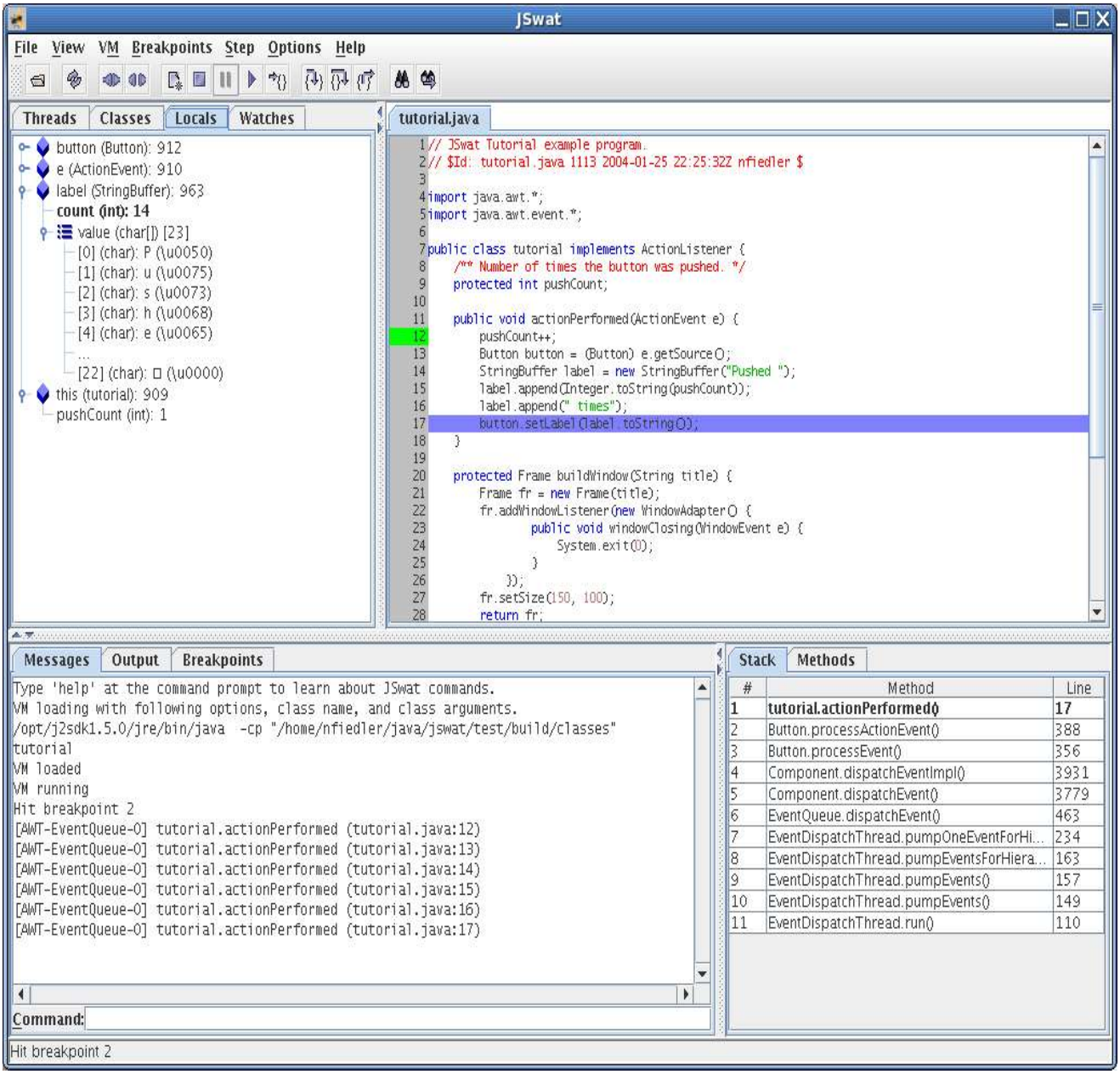
Bien qu'il ne soit pas graphique il peut le devenir en le mettant sous-débogueur de DDD. Cependant JDB n'est pas très utilisé, en parti à cause de son mode seulement textuel (ce qui n'est pas très convivial), mais aussi car il n'est pas très poussé ne fournit que la base de ce qu'un débogueur peut faire. C'est plutôt une démonstration faite par Sun de ce que l'ont peut faire avec la Java Platform Debugger Architecture .

b) Débogueurs graphiques

1. JSWAT

Débogueur java graphique. Il s'appuie lui aussi sur Java Platform Debugger Architecture.
Fonctionnalités:
- Il permet les breakpoints (comme tout débogueur) mais ceux-ci peuvent être conditionnels. Ce qui veut dire que le programme s'arrêtera sous certaine condition (par exemple lorsqu'une variable est supérieur à 10)
- Il affiche aussi les threads.
- Il permet de visualiser la pile ainsi que les classes chargées.
- Met le code source en couleur pour une meilleur visibilité.
- Permet un mode non graphique. Ce mode a 2 avantages. Il est beaucoup plus rapide (une interface graphique est toujours lourde à gérer) et ne requiert pas les classes AWT et JFC de java qui sont seulement utiles au graphisme. Le mode textuel est très ressemblant à un environnement Unix ou Emacs (avec historique des commandes par exemple), ce qui ne déstabilisera pas un habitué de ces environnements.
  Cependant dans ce cas le code source visible est très limité, et les message de JSWAT et le « debugging » sortent tous les deux sur la console. De plus les alertes de JSWAT peuvent séparer l'entrée du curseur et l'entrée du prompt. Bref ce mode nuit à la clarté du débogage.

En définitive JSWAT marche de la même manière que JDE mais avec des fonctionnalités supplémentaire et surtout un mode graphique qui nous le font préférer à ce dernier.



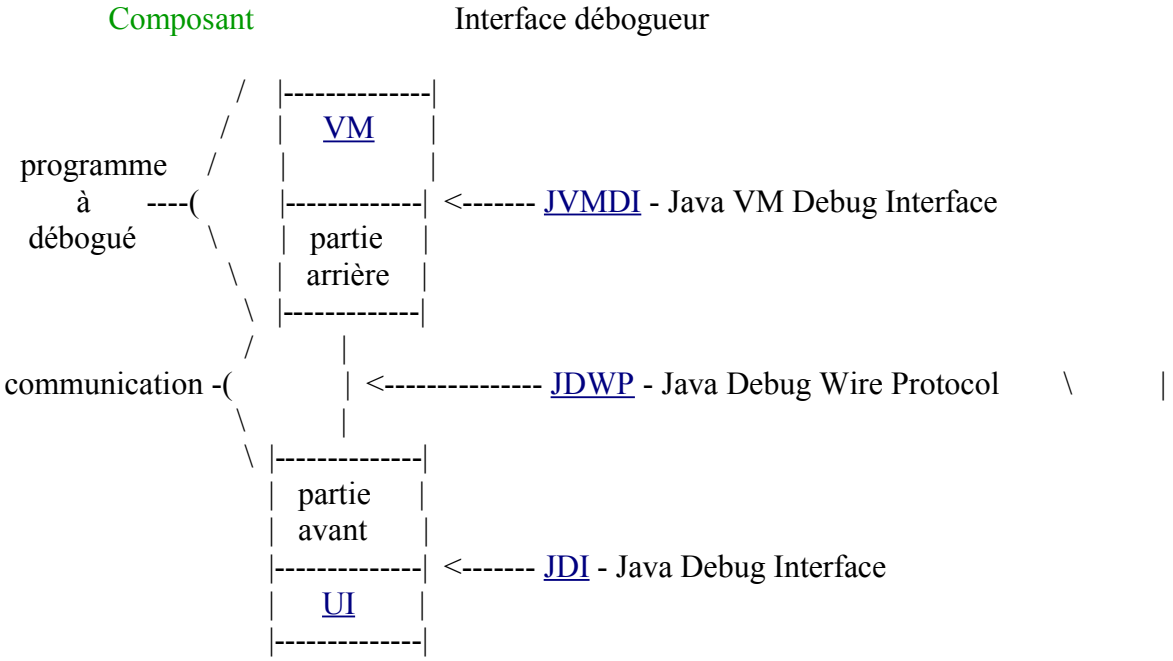
Java Platform Debugger Architecture (JPDA): Cela consiste en 3 interfaces java pour l'utilisation des débogueurs dans un environnement de développement pour les ordinateurs de bureau. Les trois interfaces sont :

- Java Virtual Machine Debugger Interface : définit les services que la machine virtuelle doit fournir pour le débogage
  Cela inclut les requêtes pour les informations (ex: la pile), les actions (ex: mettre un breakpoint), les évènements (ex: lorsqu'on rencontre un breakpoint).

- Java Debug Wire Protocol : définit le format des informations et des requêtes transférées entre le programme à déboguer et le débogueur . Elle implémente la « Java Debug Interface ».
  C'est en fait un protocole qui permet au programme à déboguer et au débogueur de marcher sous 2 machines virtuelles ou plate-formes différentes. Cela permet aussi à la partie avant du débogueur d'être écrite dans un autre langage que java

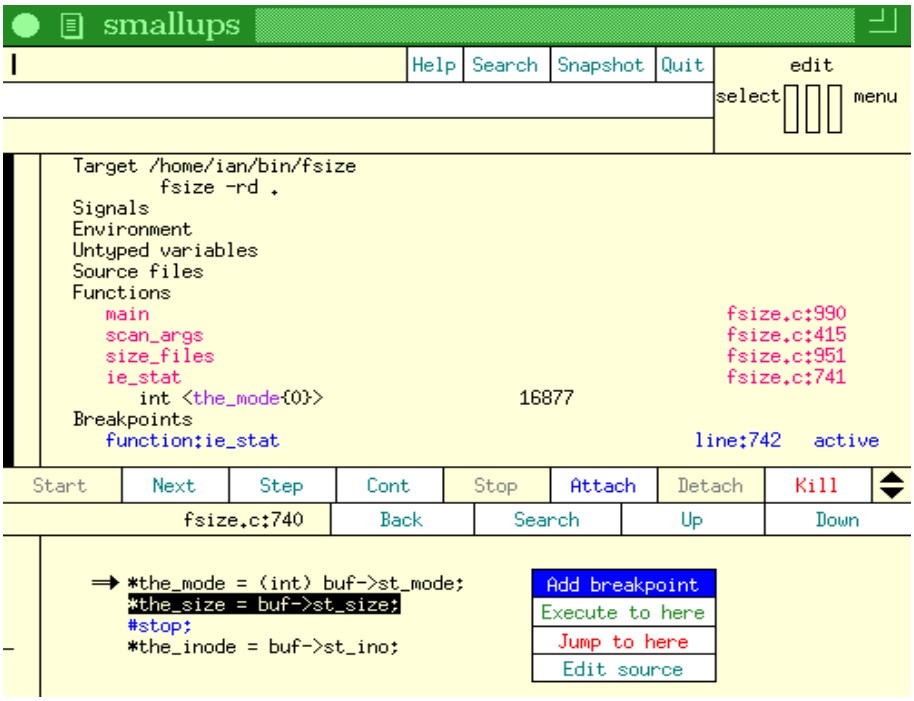
- Java Debug Interface : définit les informations et les requêtes au niveau du code utilisateur.
  Cette interface facilite grandement l'intégration des capacités de débogage dans les environnements de développement. Elle est recommandée pour tous les développements de débogueur en java.

The Java™ Platform Debugger Architecture est structurée ainsi :



2) UPS

UPS a été écrit par Mark Russell du service d'informatique à l'université de Kent à Cantorbéry, et était à l'origine une partie de la suite d'outils de logiciel de Kent.
Ups est un débugueur graphique pour les langages C, C++ et Fortran qui fonctionne sous X11 ou SunView.
La fenêtre principale se présente ainsi:



Elle est constituée de deux régions principales:

- dans la première, l'état actuel des données du programme cible (appelé zone de visualisation). Son utilisation principale doit montrer l'état du programme lorsqu'il s'est arrêté
- dans la seconde, la source qui est exécutée (appelé région source). La région source est employée pour montrer la ligne du code source qui est sur le point d'être exécutée.

Un des avantages de Ups est l'affichage permanent des variables: quand on ajoute une variable à l'affichage, elle reste là tant qu'on avance dans le code. La trace de la pile courante est toujours visible. Quand la cible s'est arrêtée, l'état de la cible est affiché sous forme de trace de la pile dans la zone de visualisation. Ceci se compose d'une ligne pour chaque fonction active donnant le nom de la fonction, du numéro de ligne du code source qui était exécuté, et du nom du fichier source contenant la fonction.
De plus, UPS inclut un interprète de C qui permet d'ajouter des fragments de codes en les éditant dans la fenêtre du code source (le fichier source lui-même n'est pas modifié). Ceci permet d'ajouter des appels de « printf » de correction sans recompiler le programme cible. Le débogueur UPS permet en plus d'observer le changement de valeur des variables pendant l'exécution de la cible. Le programmeur a la possibilité de choisir les types de variable qu'il souhaite voir s'afficher, c'est à dire soit les variables locales (en ayant le choix d'en sélectionner une ou plusieurs), soit les variables globales. Il est également possible d'examiner la valeur des macros si cette information a été incluse par le compilateur. Ceci peut être fait en cliquant sur le nom d'une macro dans la région de source. Une ligne est ajoutée à la région d'affichage.
UPS est très utile pour modifier le code source sana avoir à recompiler, cela peut être utile pour tester tous les cas d'erreurs d'un programme plus rapidemment.

3) TURBO DEBUGGER

Le Turbo Debugger permet d'analyser pas à pas le programme avec la possibilité d'observer les registres, la décharge de mémoire, les différentes variables et le code pendant l'exécution de la trace du code.
Ce débogueur donne la possibilité de regarder les registres pendant l'exécution du programme, des segments de code, le langage machine, le langage assemblé et enfin le code objet dans la mémoire. Il permet de plus de montrer les valeurs de variables, c'est à dire que ces valeurs sont montrées sans interruption pendant pendant que le programme fonctionne, et vous pouvez voir facilement comment elles changent.
Ce débogueur permet d'exécuter un programme en entier en ne tenant pas compte des valeurs intermédiaires afin de voir s'il y a un erreur et dans ce cas, où elle se trouve. Puis on peut exécuté chaque instruction du programme séparément. Et à chaque fois lorsque l'on tape sur la touche F7, la prochaine instruction sera exécutée. On peut s'arrêter entre les instructions afin d'examiner les registres, les données, la décharge, etc...
De plus, Turbo Debugger permet de placer un point d'arrêt dans le programme. Lors de l'exécution du programme celui s'arrête au breakpoint, ce qui est très utile si le programme est long ou si l'on veut étudier juste une partie du programme.

Turbo Debugger est très agréable pour pouvoir exécuté chaque instruction du programme séparément afin de procéder plus méthodiquement au debogage.

4) GDB avec DDD

Gdb est un débogueur écrit en C pour les programmes écrits en C, C++, Objective-C, Fortran, java, assembleur et Modula-2. Il est gratuit, sous licence GNU et est le sous-débogueur utilisé par défaut par DDD.
Nous verrons ses capacités sous interface graphique DDD dans une troisième partie.

5) DEET

DEET est un débogueur puissant, graphique pour C ANSI et java. Il diffère des autres débogueurs car il est extensible, ne dépend pas de la machine utilisée, programmable et petit. Il est implémenté en tksh.
Nous allons étudier ce débogueur plus précisément dans une troisième partie.

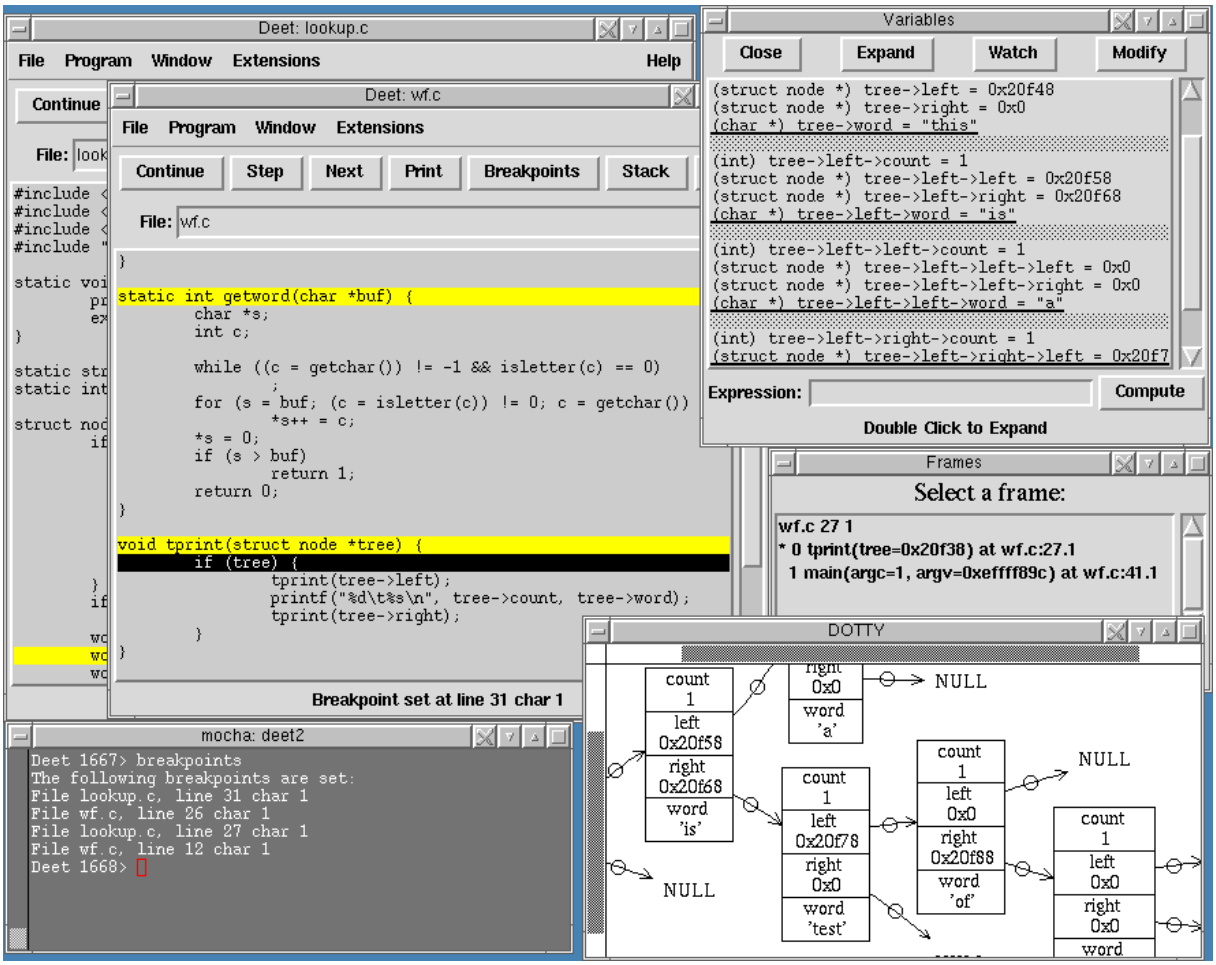
III)Approfondissement de l'utilisation de 2 debogueurs

a) DEET

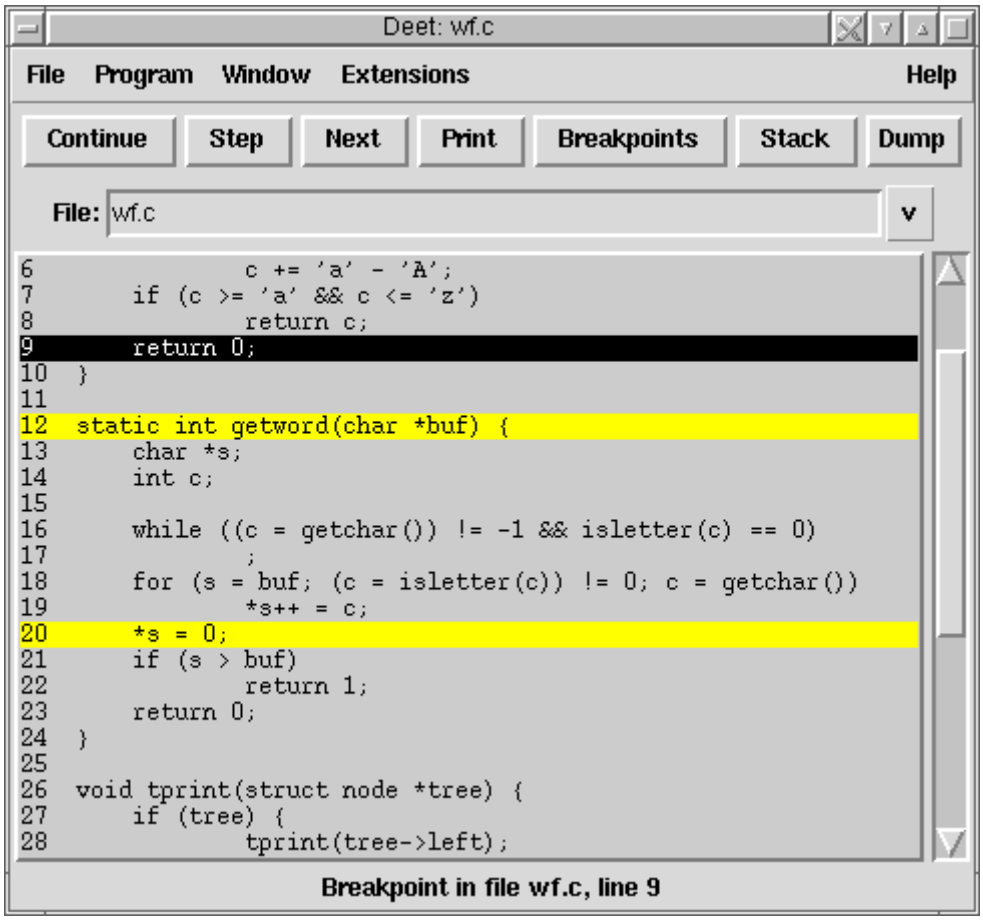
Deet est débogueur simple mais puissant pour les langages C ANSI et java. Il diffère des autres débogueurs car il est gratuit, extensible, programmable, distribuer, graphique et il peut être utiliser sur différentes machines. Il est implémenté en tksh, qui est une extension du shell « Korn » fournissant les éléments graphique de Tcl/Tk. Il permet l'utilisation des fichiers sources, des breakpoints, montre les variables pendant l'exécution, et examine les données de structures. Toute la fonctionnalité de ce débogueur est mise en application avec un petit ensemble de commandes, appelé « nub ».
Beaucoup de débogueurs sont longs et compliqués, deet a seulement 1500 lignes de shell avec quelques lignes pour le code spécifique des machines. Il est plus facile de le comprendre, de le modifier et de de l'étendre.

Les débogueurs traditionnels de UNIX sont des outils indispensables pour traiter les erreurs des programmes. En dépit de leur importance, il existe des insuffisances qui limite leur rentabilité. Par exemple, les débogueurs ont souvent des interfaces utilisateur textuelles qui sont au mieux caché. Tandis que la plupart des débogueurs fonctionnent seulement sur un logiciel d'exploitation et une architecture, les débogueurs d'Unix doivent traiter le problème de portabilité. Les débogueurs sont notamment des programmes dépendants de machine; ils dépendent de l'architecture de cible, du logiciel d'exploitation, du compilateur, et de l'éditeur de liens. Ainsi, la mise en communication d'un débogueur d'une variante d'Unix à l'autre peut exiger une quantité substantielle d'effort. Par exemple, environ un tiers du code source de Gdb est dépendant d'une machine**.**
De plus, la plupart des débogueurs ont des programmes longs et complexes; par exemple Gdb contient 150 000 ligne de C. Donc, il est habituellement difficile d'étendre un débogueur car il peut être difficile de comprendre et de modifier leur réalisation.
Deet (« desktop error elimination tool) fournit des interfaces textuelles et graphiques pour une utilisation facile. Les utilisateurs peuvent effectuer la plupart des actions de correction en se dirigeant et en cliquant avec la souris. Les structures de données peuvent être montrées graphiquement.
Presque toute l'exécution de Deet n'est pas liée à un type de machine particulier.

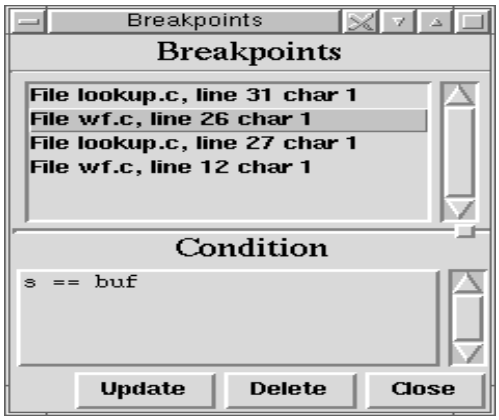
La figure 1 montre l'écran d'une session de correction de Deet



Voici un exemple d'utilisation du débogueur DEET:
Pour lancer Deet faire « DEBUGGER=deet a.out ».
Dans notre exemple on lance un fichier wf.c dont voici l'interface graphique avec Deet:



Pour placer un breakpoint, il faut simplement cliquer sur la ligne choisie, de plus Deet permet une représentation de tout les breakpoints comme dans la figure ci-dessous, en montrant des informations relatives à ce ce dernier; tel que l'endroit où il apparaît et sa condition de coupure. Ces conditions sont des expressions Deet qui sont évaluées à chaque fois qu'un breakpoint est atteint. Si la condition est vrai, la cible s'arrête. Quand la cible s'arrête à un breakpoint, la fenêtre courante du code source montre le nom du fichier, la ligne du breakpoint et la vidéo inverse marque la ligne où se trouve ce breakpoint.



Le débogueur Deet se divise en deux parties: une partie agit avec le programmeur et l'autre partie agit avec le programme source. L'interface utilisateur est écrite en Tksh une nouvelle version du shell de Korn. Le programme cible est commandé par un « nub » qui fournit des primitives de correction. Ainsi le programmeur peut modifier les deux parties de Deet en écrivant du code en Tksh.

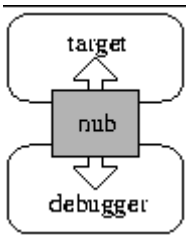


Illustration 1architecture de cdb

Deet est basé sur le cdb. Le cdb est un débogueur qui ne dépend d'aucune machine et qui permet d'éliminer les dépendances de machines en ajoutant des informations dans le programme cible lors de la compilation.

Il y a quatre composantes de cdb:
- l'interface de « nub », qui se tient entre la cible et le débogueur,
- l'exécution de « nub », qui comprend les fonctions de l'interface de « nub », un emballage autour de l'éditeur de liens pour charger le « nub » et la table des symboles non liée à un type de machine particulier,
- un format de table des symboles indépendantes d'une machine qui est émis par le compilateur et incorporé dans la cible,
- une base de débogueur textuel qui est utilisée pour fournir la fonctionnalité minimale du « nub ». Ce débogueur est prévu pour être remplacé par des débogueurs plus sophistiqué, comme Deet.

Employer Deet n'implique directement des changements à aucun de ces composants, mais pour mettre en application Deet, il a fallu modifier le « nub » et le format de la table des symboles au delà de leurs conceptions originales.
Le tksh est utilisé comme langage de correction pour Deet en raison de ces avantages comme langage de commandes de gestion interactif. De ce fait, Deet utilise les équipements interactifs de tksh, comme l'édition de la ligne de commande et la gestion des JOBS.

D'autre part, Deet inclus des fonctions de « nub » et des fonctions de la table des symboles qui sont utilisée avec le Tcl ou le Tksh. Ces commandes de tksh diffèrent des routines de C de deux manières: Elles sont à un niveau plus élevé, parce qu'elles manoeuvrent des symboles, des types, des valeurs, et elles acceptent et renvoient des « String », de sorte qu'elles puissent être employées par TCL ou tksh. La liste complète, dans le schéma ci-dessous, permet de connaître les fonctions de Deet permettant notamment de voir l'adresse, la valeur d'une variable, l'ajout et la suppression des breakpoints, etc ... .

deet_open	initialize the target
deet_breakpoint { -set | -delete | -list } *file line character*	set, remove, and list breakpoints
deet_frame [*n*]	get/set current frame
deet_getval *type address*	read a value of *type* from *address*
deet_putval *type address value*	write the *value* of *type* to *address*
deet_continue	resume execution
deet_sym { -all | -files | -locals | -params | -name *name* }	finds the symbol-table entries
deet_type *type*	get *symbol*'s type information

Les différentes modification de Deet :

- Deet lui-même est écrit en TCL et en tksh, en utilisant les fonctions de Deet (décrites dans le tableau ci-dessus). Les utilisateurs peuvent modifier Deet en écrivant des commandes en TCL et en tksh : par exemple, utiliser des breakpoints conditionnels. Deet peut également être étendu par des programmes externes. Des extension simples peuvent être écrites directement dans le tksh. Par exemple, le script suivant montre tous les éléments nuls dans une liste, dont le nom est fourni comme argument.

```
function nullElements {
    typeset arr=$1
    integer i s=$(arraySize $arr)
    for (( i=0 ; i < s ; i++ )); do
        if [[ $(var "$arr[$i]") == 0x0 ]]
        then
            print "Element $arr[$i] null"
        fi
    done
}
```

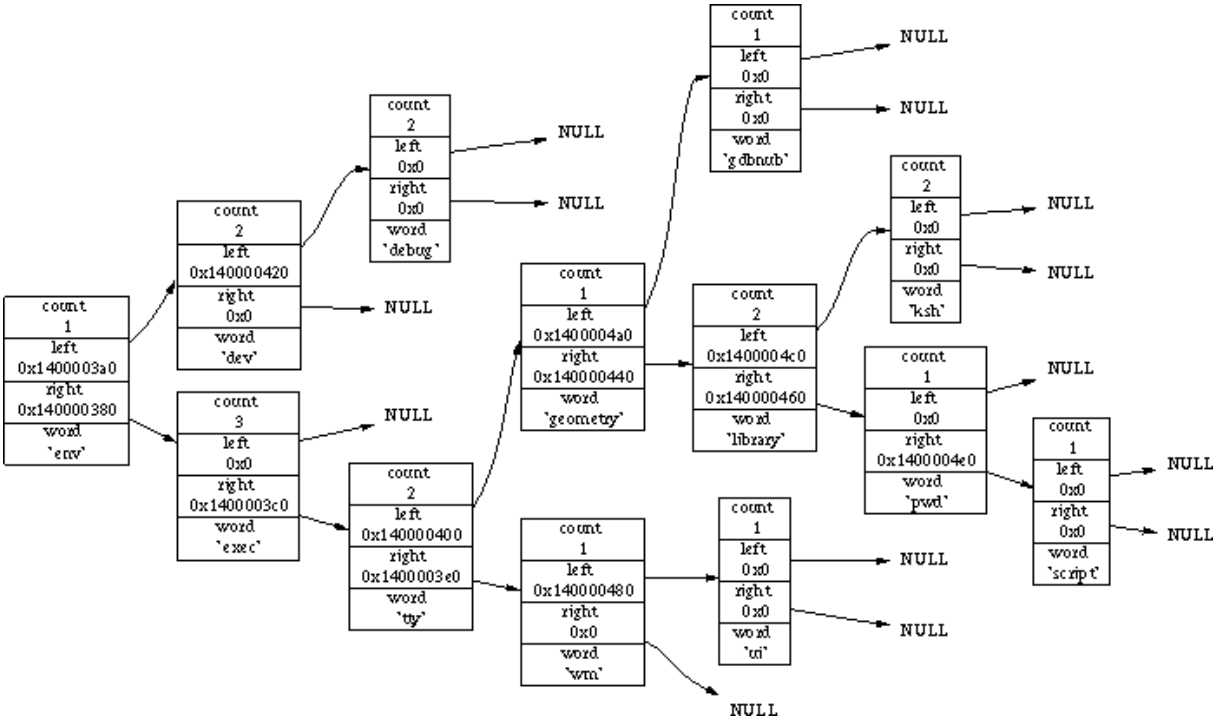
nullElement utilise deux fonctions externe du tksh: arraySize, qui renvoie le nombre d'élément dans une liste, et var, qui renvoie la valeur d'une variable

- Les fonctions définies pour l'utilisateur peuvent également modifier l'interface de deet. Par exemple, si nous vérifions à plusieurs reprises les éléments nuls dans une hashtable, nous pouvons construire un bouton pour faire le travail dont voici le script:

```
    toplevel .null
pack $(button .null.b \
    -text "Print Null Elements" \
    -command "nullElements hashtable")
```

Et ce code donne le bouton suivant:

Les possibilités de deet sont facilement étendues en écrivant des scripts en TCL et en tksh qui emploient les commandes intégrées du débogueur. L'avantage est qu'il peut utiliser n'importe quel outil externe. Par exemple, il est relativement facile d'étendre deet pour montrer les structures de données directement dans un graphique. Ce dispositif est semblable à celui fourni dans Data Display Debugger (DDD), mais l'exécution est beaucoup plus simple, parce que Deet utilise l'outil « dotty » pour tracer les graphiques.
Le schéma ci-dessous montre un exemple de « dotty ». Le script de tksh qui appelle « dotty » contient seulement 60 lignes de code, et il manipule n'importe quelle structure de données.

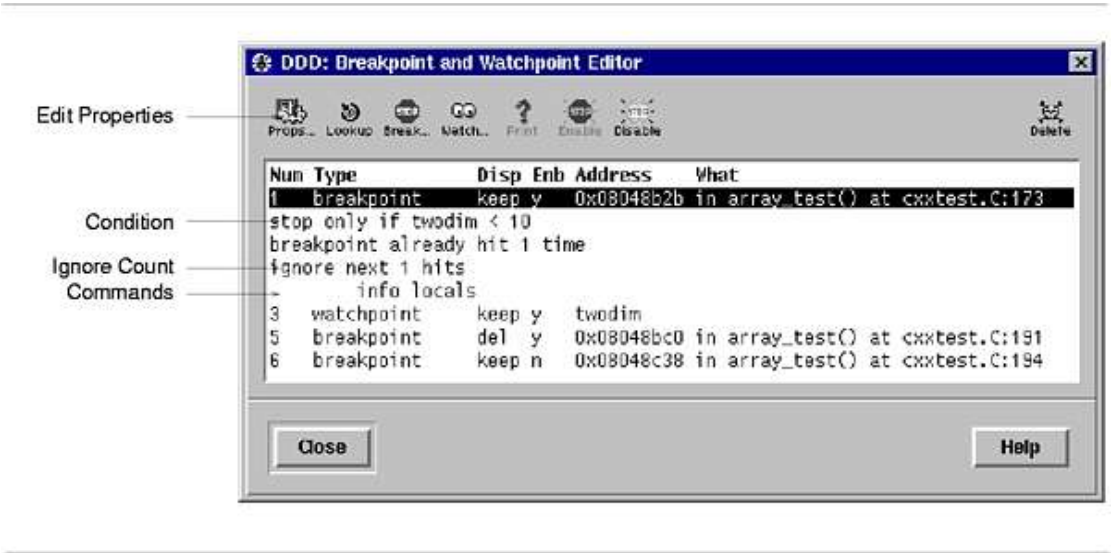


Finalement, Deet marche sur n'importe quel machine sur lequel tksh fonctionne, permet le placement des breakpoints, la visualisation des variables et surtout par rapport à d'autre débogueurs il est extensible.

b) GDB (avec DDD)

Gdb est un débogueur puissant et très complet. Il peut déboguer beaucoup de langages différents comme nous l'avons dit plus haut (C, C++, Objective-C, java, assembleur, fortran, perl et Modula-2). Il est implémenté en C.

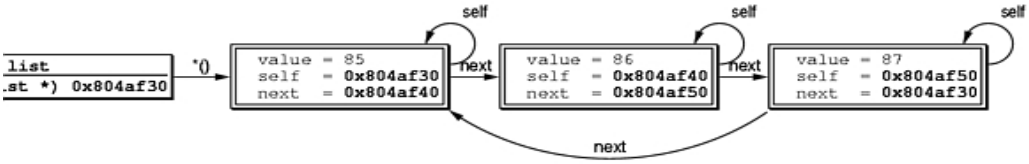
Il permet bien sûr toutes les fonctionnalités de base d'un débogueur avec plusieurs autres supplémentaires. Par exemple, les breakpoints placés peuvent êtres conditionnels, c'est à dire que l'exécution s'arrête lorsqu'une variable a atteint une certaine valeur par exemple, temporaire si l'ont veut que l'arrêt ne ce fasse qu'une fois à cet endroit ou bien encore on peut placer un breakpoint qui sera actif au bout d'un certain nombre de fois. Enfin lorsqu'un breakpoint est rencontré on peut définir dans DDD une commande à effectuer automatiquement. Un autre style de breakpoint peut être utilisé , cela s'appelle un watchpoint. Un watchpoint est associé à une variable, il stoppera l'exécution dès que la valeur de celle-ci changera. Bien sur vu le nombre de breakpoints spéciaux pouvant être mis, on peut lister tous les breakpoints afin de voir leurs propriétés.



The Breakpoint Editor

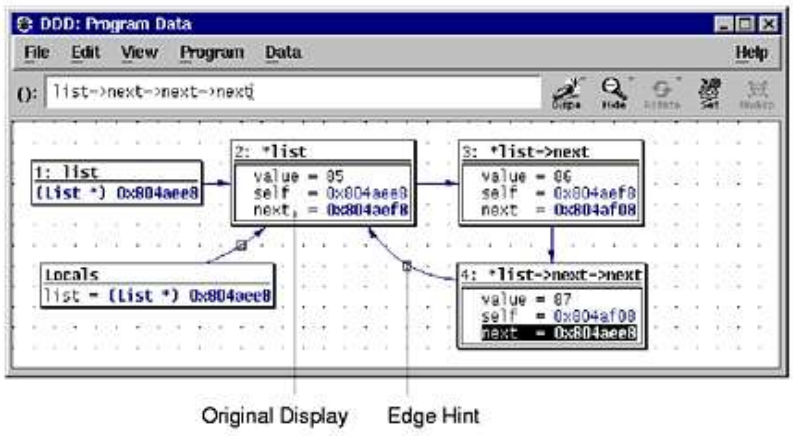
Nous nous apercevons, avec l'image ci-dessus, que nous pouvons voir l'adresse mémoire des breakpoints. Ce qui nous amènes à un des but de ce TE, c'est à dire la visualisation des variables en mémoire. Bien sûr GDB permet d'afficher une variable avec son adresse en mémoire. Tout type de variable est affichable, que ce soit une structure, des pointeurs ou des tableaux. Pour un tableau il faut préciser la longueur de ce dernier grâce au caractère @ pour qu'il soit affiché en entier. Chaque variable peut être visualisée dans une fenêtre « display » de DDD, ce qui permet de voir les changements de valeur de celles-ci tout au long de l'exécution. Chaque valeur inscrite grâce à la commande « print » est sauvegardée dans « l'historique des valeurs » de GDB. Cela permet d'y référer d'une façon simple dans d'autres expressions, de la manière suivante : $*num*. où num est donné lorsqu'on « print » la variable. Enfin les valeurs des variables peuvent être changées au milieu de l'exécution, cela permet de voir comment le programme réagirait dans certaines conditions.

Lorsqu'on veut afficher des structures qui pointent sur d'autres structures, l'affichage peut s'effectuer sous forme de liste ou sous forme d'arbre afin de rendre cette visualisation plus claire.



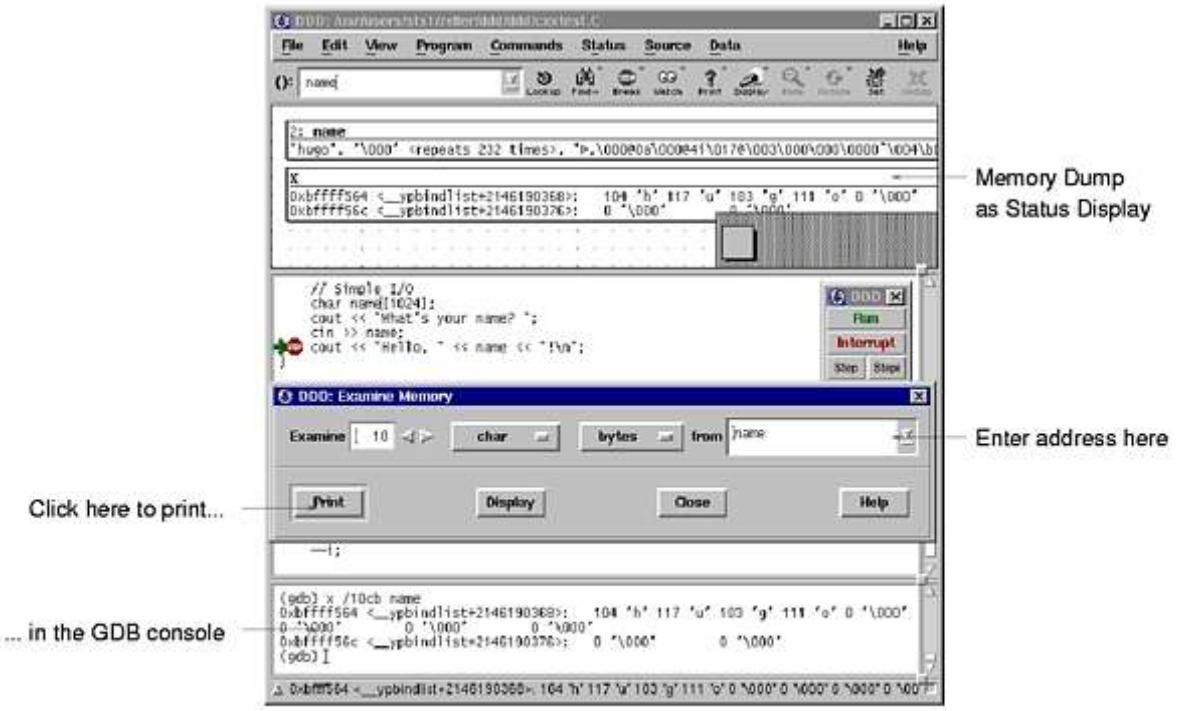
Cette affichage peut ensuite être imprimer pour illustrer un programme par exemple.

Par défaut DDD ne reconnaît pas 2 objets ayant la même adresse mémoire comme un seul objet . Cependant il existe le mode « Alias Detection » qui permet de fusionner 2 objets ayant la même adresse afin d'avoir une meilleur visualisation d'ensemble



Examining Shared Data Structures

GDB permet aussi de visualiser une partie de la mémoire sous plusieurs formats indépendamment des types de données du programme. Pour visualiser la mémoire il suffit de préciser l'adresse de départ, le nombre d'unité mémoire à afficher et le format sous lequel afficher. L'adresse examinée par défaut est juste après la dernière adresse visualisée. L'adresse à afficher n'est pas forcement l'adresse mémoire d'une variable définie dans le programme, cela peut être n'importe quelle adresse mémoire dans laquelle Gdb à un droit de lecture.

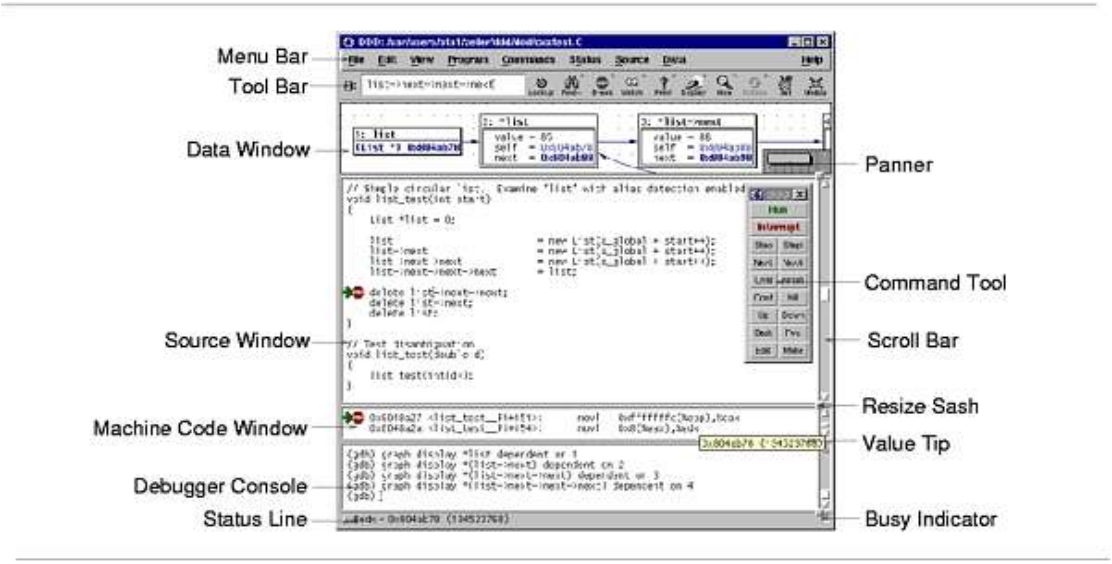


Examining Memory

Il est aussi possible d'examiner l'exécution du programme au niveau du code machine. Cela ouvre une nouvelle fenêtre affichant les instructions assembleurs du programme. Cette fenêtre fonctionne à peu prêt de la même façon que la fenêtre contenant le code source. On peut en effet y placer, enlever des breakpoints. Ce code machine peut être afficher spécifiquement pour une certaine fonctions ou une adresse spécifique. Cela permet de ne pas se perdre dans les instructions assembleur lorsqu'on s'intéresse à un endroit particulier du programme. Les registres sont aussi visibles. Il faut quand même faire attention lorsqu'on exécute le programme instruction machine par instruction machine, des variables peuvent apparaître avec de mauvaises valeurs jusqu'à ce que la pile soit complètement construite.

DDD au dessus de GDB permet une utilisation plus intuitive et facile de GDB, tout en gardant toutes les fonctionnalités de ce dernier. De plus DDD est personnalisable. On peut définir de nouveaux boutons. Cela peut être utile car GDB permet de définir d'autre commandes, qui sont en fait des combinaisons de commande de GDB, ou bien, on peut aussi avoir besoin de définir un bouton qui effectue une action que l'on utilise souvent pour déboguer le programme. DDD permet aussi de visualiser totalement le code source en même temps qu'on le débogue et aussi de faire les actions les plus courante (comme placer un breakpoint) d'un seul clic. C'est pour cela que DDD est si utile. Il garde toutes les fonctionnalité de GDB tous en simplifiant son utilisation.

Voici un affichage de DDD dans son ensemble :



The DDD Layout using Stacked Windows

Étant gratuit sous licence GNU une très grande documentation est fournis avec GDB et DDD, ce qui n'est pas le cas avec tous les débogueurs. De plus il est totalement extensible, et tout le monde peut travailler dessus pour le rendre meilleur, ajouter d'autres langages, le rendre plus performant, ajouter de nouvelles fonctionnalités. C'est pourquoi il est si complet. De part ce fait des règles ont été instaurées pour programmer et étendre GDB. La mise en page, surtout, est réglementée. Il faut bien prendre cela en compte pour que son travail dessus soit reconnu. Bien sûr, toutes les sources sont disponibles et une documentation interne explique la plupart des macros utilisées et certaines fonctions, afin de rendre l'amélioration de ce programme accessible à tout le monde.

CONCLUSION

En définitive nous avons vu qu'il existe un très grand nombre de débogueurs pour tous les langages. Ils sont plus ou moins performants et portables. Nous avons fait un tour d'horizon des débogueurs en sélectionnant ceux qui nous ont parus les plus efficaces, pertinent et les mieux documentés.
Les débogueurs sont pratiquement indispensables pour les programmeurs car ils permettent un gain de temps considérable dans la recherche des erreurs d'un programme. En effet, grâce à l'utilisation d'interface graphique, les débogueurs sont plus facile à utiliser. On peut par exemple placer un breakpoint en un clic de souris ou avoir accès à la pile des appels, la valeur des variables dans une fenêtre à part, qui change dynamiquement tout au long de l'exécution. L'affichage des pointeurs donne une meilleure vue d'ensemble de l'état des variables dans le programme.
De plus, ils peuvent aider à une meilleure compréhension du fonctionnement interne du programme grâce à la visualisation de la mémoire. Certains débogueurs donnent la possibilité de pouvoir interactivement modifier le programme sans avoir besoin de le recompiler, de visualiser les instructions assembleurs, les registres.
Enfin, principalement pour les deux derniers débogueurs traités (DEET et GDB), ils sont totalement extensibles et de ce fait leurs fonctionnalités peuvent vraiment évoluer. Cela permet à un programmeur de personnaliser son débogueur afin de le rendre plus performant et pratique pour son utilisation personnelle.
Finalement, l'apport personnel de chaque utilisateur, surtout pour les logiciels libres style GNU, permettra peut-être dans le futur de « grossir » leurs possibilités et de donner tous les outils dont le programmeur aura besoin à la conception d'un programme.

Pour nous le meilleur débogueur serait un programme qui aurait toutes les fonctionnalités suivantes :

- Il devra être graphique, car cela facilite grandement son utilisation, principalement pour les programmeurs débutants qui n'en utilise pas souvent car le mode textuel n'incite pas, au premier abord, à les utiliser.
- Il devra être protable afin de pouvoir l'utiliser sur différentes machines et différents systèmes d'exploitations. Cela pour ne pas à avoir à changer de débogueur, ce qui est désagréable lorsqu'on est habituer à en utiliser un en particulier.
- De même il devra fonctionner pour un maximun de langages pour les raisons évoquées ci-dessus.
- Un bon affichage des variables est important. En effet, la possibilité de voir l'évolution des valeurs des variables, leur adresse mémoire et un bon affichage de tous les types de structures à tout moment de l'exécution, facilite grandement le débogage.
- Une documentation complète, bien réalisée et avec des exemples est le meilleur moyens pour bien diffuser son débogueur. En effet, un débogueur très peu documenté n'est pas très attreillant pour l'utilisateur, qui devrait passer un long moment de test pour commencer à savoir bien l'utiliser.
- Dernier point, le fait qu'un débogueur soit extensible est un atout non négligeable pour l'utilisateur qui pourra le personnaliser et même peut être l'améliorer pour des versions futures.

Voilà ce que devrait être un bon débogueur.

Parmi ceux qui ont attiré notre attention, gdb avec interface DDD est le meilleur débogueur. Il répond à toutes les attentes énumérées ci-dessus avec une petit bémol qui est la taille de son code source (environs 150000 lignes). A cause de cela il est assez difficile à étendre mais ce n'est pas impossible pour des programmeurs expérimenté, car tout est fait pour que cela puisse être fait.

BIBLIOGRAPHIE:

. sites officiels de GNU pour les déboguers GDB et l'interface DDD :
http://www.gnu.org/software/gdb/gdb.html
http://www.gnu.org/software/ddd

. Documentation sur DBX
http://www.physics.utah.edu/~p573/hamlet/lessons/dbx/dbx/dbx.html

. COCA
COCA: A Debugger for C Based on File Grained Control Flow and Data Events
Rapport de recherche n3489 de septembre 1998 de Mireille Ducassé INRIA Rennes
http://www.irisa.fr/lande/ducasse

. site HP invent pour LADEBUG debugger :
http://h21007.www2.hp.com/dspp/tech/tech_TechSoftwareDetailPage_IDX/1,1703,5218,00.html

.site Sun pour la documentation de JDB:
http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/jdb.html

.Site officiel de JSWAT:
http://www.bluemarsh.com/java/jswat

. documentation de UPS:
http://www.concerto.demon.co.uk/UPS

. documentation de TURBO DEBUGGER:
http://www.sci.brooklyn.cuny.edu/~jones/CIS4.1/turbodebug.html

. site du débogueur DEET:
http://www.cs.princeton.edu/~jlk/deet

Autres Bibliographie:

1

K. Arnold and J. Gosling. The Java Programming Language. Addison-Wesley, Reading, MA, 1996.

2

M. Bolsky and D. Korn. The New KornShell Command and Programming Language. Prentice Hall, Upper Saddle River, NJ, second edition, 1995.

3

C. W. Fraser and D. R. Hanson. A Retargetable C Compiler: Design and Implementation. Addison-Wesley, Menlo Park, CA, 1995.

4

M. Golan and D. R. Hanson. DUEL--a very high-level debugging language. In Proceedings of the Winter USENIX Technical Conference, pages 107-117, San Diego, CA, Jan. 1993.

5

D. R. Hanson. Variable associations in SNOBOL4. Software--Practice and Experience, 6(2):245-254, Apr. 1976.

6

D. R. Hanson and M. Raghavachari. A machine-independent debugger. Software--Practice and Experience, 26(11):1277-1299, Nov. 1996.

7

D. G. Korn. ksh: An extensible high level language. In Proceedings of the Very High Level Languages Symposium (VHLL), pages 129-146, Santa Fe, NM, October 1994.

8

J. L. Korn. Tksh: A Tcl library for KornShell. In Proceedings of the USENIX Tcl/Tk Workshop, pages 149-159, Monterey, CA, July 1996.

9

E. Koutsofios and S. C. North. Applications of graph visualization. In Proceedings of Graphics Interface 1994 Conference, pages 235-245, Banff, Canada, May 1994.

10

P. Maybee. NeD: The network extensible debugger. In Proceedings of the Winter USENIX Technical Conference, pages 145-153, San Antonio, TX, July 1992.

11

Microsoft Corp., Redmond, WA. Microsoft Visual C++, Reference Volume II, 1993.

12

J. K. Ousterhout. Tcl and the Tk Toolkit. Addison-Wesley, Reading, MA, 1994.

13

28

N. Ramsey and D. R. Hanson. A retargetable debugger. Proceedings of the SIGPLAN'92 Conference on Programming Language Design and Implementation, SIGPLAN Notices, 27(7):22-31, July 1992.

14

R. M. Stallman and R. H. Pesch. Using GDB: A guide to the GNU source-level debugger, GDB version 4.0. Technical report, Free Software Foundation, Cambridge, MA, July 1991.

15

P. Winterbottom. Acid: A debugger built from a language. In Proceedings of the Winter USENIX Technical Conference, pages 211-222, San Francisco, CA, Jan. 1994.

16

A. Zeller and D. Lütkehaus. DDD -- a free graphical front-end for UNIX debuggers. SIGPLAN Notices, 31(1):22-27, January 1996.

17

P. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. In Workshop on Parallel and Distributed Debugging, pages 1122, Sigplan Notices, Vol 24, Number 1, January 1989.

18

J. Bovey, M. Russel, and O. Folkestadt. Direct manipulation tools for Unix workstations. In Proceedings of the EUUG Autumn'88, pages 311319, October 1988.

19

D. Cohen, M. S. Feather, K. Narayanaswamy, and S. Fickas. Automatic monitoring of software requirements. In Proceedings of the 19th International Conference on Software Engineering, pages 602603. IEEE Press, 1997.

20

M. Consens, M. Hasan, and A. Mendelzon. Visualizing and querying distributed event traces with Hy+. In W. Litwin and T. Risch, editors, Applications of Databases, First International Conference, pages 123141. Springer, Lecture Notes in Computer Science, Vol. 819, 1994.

21

Ducassé. Abstract views of Prolog executions in Opium. In V. Saraswat and K. Ueda, editors, Proceedings of the International Logic Programming Symposium, pages 1832, 1991. MIT Press. ILPS'91.

22

M. Ducassé. Opium: An extendable trace analyser for Prolog. The Journal of Logicprogramming, 1999. RR n3489 20 M. Ducassé

23

29

Ducassé and J. Noyé. Logic programming environments: Dynamic program analysis and debugging. The Journal of Logic Programming, 19/20:351384, May/July 1994.

24

M. Golan and D. Hanson. DUEL- A very high-level debugging language. In Proceedings of the Winter USENIX Technical Conference, 1993.