Université de Montréal

Technique de visualisation pour l'identification de l'usage excessif d'objets temporaires dans les traces d'exécution

par
Fleur Duseau

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des arts et des sciences
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

Décembre, 2011

© Fleur Duseau, 2011.

Université de Montréal
Faculté des arts et des sciences

Ce mémoire intitulé:

Technique de visualisation pour l'identification de l'usage excessif d'objets temporaires dans les traces d'exécution

présenté par:

Fleur Duseau

a été évalué par un jury composé des personnes suivantes:

Pierre Poulin,	président-rapporteur
Bruno Dufour,	directeur de recherche
Houari Sahraoui,	codirecteur
Stephan Monnier,	membre du jury

Mémoire accepté le: . . . . . .

RÉSUMÉ

De nos jours, les applications de grande taille sont développées à l'aide de nombreux cadres d'applications (*frameworks*) et intergiciels (*middleware*). L'utilisation excessive d'objets temporaires est un problème de performance commun à ces applications. Ce problème est appelé *"object churn"*. Identifier et comprendre des sources d'*"object churn"* est une tâche difficile et laborieuse, en dépit des récentes avancées dans les techniques d'analyse automatiques.

Nous présentons une approche visuelle interactive conçue pour aider les développeurs à explorer rapidement et intuitivement le comportement de leurs applications afin de trouver les sources d'*"object churn"*. Nous avons implémenté cette technique dans Vasco, une nouvelle plate-forme flexible. Vasco se concentre sur trois principaux axes de conception. Premièrement, les données à visualiser sont récupérées dans les traces d'exécution et analysées afin de calculer et de garder seulement celles nécessaires à la recherche des sources d'*"object churn"*. Ainsi, des programmes de grande taille peuvent être visualisés tout en gardant une représentation claire et compréhensible. Deuxièmement, l'utilisation d'une représentation intuitive permet de minimiser l'effort cognitif requis par la tâche de visualisation. Finalement, la fluidité des transitions et interactions permet aux utilisateurs de garder des informations sur les actions accomplies. Nous démontrons l'efficacité de l'approche par l'identification de sources d'*"object churn"* dans trois applications utilisant intensivement des cadres d'applications *framework-intensive*, incluant un système commercial.

Mots-clefs : visualisation, applications *framework-intensive*, *object churn*, analyse dynamique, traces d'exécution

ABSTRACT

Nowadays, large framework-intensive programs are developed using many layers of frameworks and middleware. Bloat, and particularly *object churn*, is a common performance problem in framework-intensive applications. Object churn consists of an excessive use of temporary objects. Identifying and understanding sources of churn is a difficult and labor-intensive task, despite recent advances in automated analysis techniques.

We present an interactive visualization approach designed to help developers quickly and intuitively explore the behavior of their application with respect to object churn. We have implemented this technique in Vasco, a new flexible and scalable visualization platform. Vasco follows three main design goals. Firstly, data is collected from execution traces. It is analyzed in order to calculate and keep only the data that is necessary to locate sources of object churn. Therefore, large programs can be visualized while keeping a clear and understandable view. Secondly, the use of an intuitive view allows minimizing the cognitive effort required for the visualization task. Finally, the fluidity of transitions and interactions allows users to mentally preserve the context throughout their interactions. We demonstrate the effectiveness of the approach by identifying churn in three framework-intensive applications, including a commercial system.

Keywords : visualization, framework-intensive applications, object churn, dynamic analysis, execution traces

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

REMERCIEMENTS

A l'écriture de ce mémoire, je pense à remercier en particulier : Bruno Dufour, mon directeur, pour avoir cru en moi. Merci pour ton soutien depuis le début, tes avis et tes encouragements. Houari Sahraoui, mon co-directeur, pour m'avoir fait entrer dans l'équipe du laboratoire GEODES. Merci pour ton soutien. Mes parents, pour leur éternel amour. Merci pour votre soutien moral et financier. Mes proches et amis pour leur soutien psychologique sans lequel je n'aurais pu avancer.

Je pense également à Jean-Lou Desbarbieux, professeur à l'université Pierre et Marie Curie, de m'avoir donné la curiosité de continuer dans le domaine de la programmation grâce à sa passion et sa gentillesse. Merci aussi de m'avoir soutenue pour l'entrée à l'université de Montréal.

CHAPITRE 1

INTRODUCTION

1.1 Contexte

De nos jours, les sources des programmes sont de plus en plus accessibles, ce qui augmente la réutilisation de code et la création de beaucoup de cadres d'applications (*frameworks*) et d'intergiciels (*middleware*) de grande taille. Les applications qui utilisent intensivement des cadres d'applications et intergiciels sont appelées applications *framework-intensive*. Par exemple, les applications web sont généralement constituées d'une grande quantité de cadres d'applications et intergiciels complexes.

Dans des travaux précédents, Dufour et al. [10] ont étudié l'utilisation excessive d'objets temporaires, appelée *object churn*, un problème de performance commun aux applications *framework-intensive*. Ce problème peut provoquer une augmentation du temps d'exécution, d'une part à cause du ramasse-miettes (*garbage collector*), mais surtout en raison du coût de l'initialisation de ces objets. Il n'est pas rare pour ces applications de prendre un temps considérable pour initialiser ces objets temporaires qui sont éliminés juste après. Par exemple, dans l'application Trade, la méthode `DateSerializer.getValu` recrée à chaque appel des objets calendrier pour chaque valeur devant être désérialisée. De plus, le constructeur `Calendar` crée 99 objets sur neuf invocations. Or, ces objets pourraient être facilement réutilisés pour éviter le coût associé à les recréer.

1.2 Problématique

Les problèmes de performance dans ces applications *framework-intensive* sont généralement très difficiles à identifier. Tout d'abord, les problèmes ne se trouvent que très rarement dans une seule méthode ou classe, mais se répartissent sur une région de l'exécution. Les techniques de profilage existantes donnent des informations relatives à une méthode, ce qui ne permet pas de trouver les régions qui sont responsables des sources d'*"object churn"*. De plus, les développeurs sont rarement familiers avec les cadres et

les bibliothèques utilisés par ces applications. Ils ne peuvent donc pas prévoir l'impact qu'auront leurs décisions sur la performance de l'application.

Récemment, Shankar et al. [27] ont proposé Jolt, une optimisation de compilateur à la volée (*Just-in-time* ou *JIT compiler*) spécialement conçue pour corriger les sources d'*"object churn"*. Bien que cette technique soit plus efficace que les précédentes, elle manque encore de nombreuses possibilités d'optimisation. De plus, elle élimine le coût des allocations et désallocations des objets temporaires, mais non le coût de l'initialisation.

Comme les techniques d'optimisation conventionnelles sont incapables d'éliminer le problème, il est important d'aider les développeurs à localiser, comprendre et éliminer les sources d'*"object churn"* dans leurs applications. La plupart des outils d'analyse existants, basés sur des techniques statiques ou dynamiques, sont souvent trop limités par la taille des fichiers en entrée pour gérer les applications framework-intensive, ou sont limités dans les informations qu'ils fournissent.

Dans leurs travaux, Dufour et al. [10] ont développé un outil d'analyse, Elude, qui peut identifier des objets potentiellement temporaires dans une exécution de programme. Elude utilise une analyse statique d'échappement sur une région d'un programme déterminée par une analyse dynamique (appelée analyse d'échappement *combinée*). L'analyse d'échappement permet d'identifier les objets qui ne sont accessibles que localement et donc "capturés" dans une région d'un programme. Elude peut être utilisé pour identifier les endroits qui sont potentiellement des sources d'*"object churn"*. Ce processus reste la plupart du temps manuel pour deux raisons principales. Premièrement, l'outil peut générer des faux positifs car les notions d'*"object churn"* et même d'objets temporaires sont toujours informellement définies. Deuxièmement, même dans des cas où les objets temporaires identifiés sont réels, déterminer la cause principale de ces objets temporaires est une tâche difficile qui consiste à suivre la circulation des objets temporaires à travers plusieurs méthodes d'une même région, et requiert une bonne compréhension du comportement du programme.

Plus récemment, des techniques automatiques ont été proposées pour identifier les copies excessives entre objets [34], les conteneurs utilisés inefficacement [37], les struc-

tures de données à faible utilité [36] ou identifier les objets temporaires créés à l'intérieur de boucles [5]. Ces analyses visent certaines formes spécifiques d'*"object churn"*, et ne sont donc pas en mesure d'identifier toutes les sources d'*"object churn"*. De plus, il est difficile pour un développeur de comprendre les problèmes identifiés par ces outils et de décider de la marche à suivre pour les corriger. Par exemple, il peut être difficile ou même impossible de corriger certains problèmes identifiés parce que le code problématique se trouve à l'extérieur de l'application. Les techniques automatiques actuelles identifient souvent les *symptômes* d'un problème et non sa cause. Déterminer la cause d'*"object churn"* nécessite de suivre le flot des objets à travers plusieurs niveaux de méthodes, une tâche difficile à effectuer sans un outil approprié.

Donc, il n'existe pas d'outils entièrement automatiques qui peuvent identifier les sources d'*"object churn"*, et permettre aux développeurs de comprendre comment leur programme peut être modifié pour éliminer ce problème. Par conséquent, les utilisateurs doivent inspecter manuellement une très grande quantité de données d'analyse, un processus long et fastidieux. Il est donc nécessaire de développer de nouveaux outils et techniques pour faciliter l'identification et l'élimination d'*"object churn"* dans les applications.

1.3 Contributions

L'objectif de ce mémoire est de présenter nos trois principales contributions qui suivent. Tout d'abord, nous avons développé une technique de visualisation spécifiquement conçue pour explorer et identifier les sources d'*"object churn"* dans les applications *framework-intensive*. Cette technique repose sur la métaphore du Sunburst [29] pour visualiser les résultats de l'analyse d'échappement combinée faite sur les données de la trace d'exécution tout en réalisant un compromis entre l'extensibilité (*scalability*) et la précision.

Ensuite, nous avons implémenté un outil utilisant cette technique, appelé Vasco, qui supporte le processus d'exploration interactive à travers des mécanismes soigneusement conçus sur des principes de la perception préattentive [14] afin de minimiser l'effort

cognitif nécessaire à explorer le comportement complexe des applications *framework-intensive* du monde réel. Le problème d'*"object churn"* traversant différentes méthodes et même les limites du *framework*, notre approche permet de suivre les objets temporaires de leur création à leur utilisation. De plus, l'*"object churn"* est souvent le résultat d'un ensemble de méthodes dans une région donnée de l'exécution, notre approche permet à un utilisateur de rapidement et facilement identifier les régions présentant un comportement suspect. Les régions du programme sont représentées explicitement et intuitivement. Finalement, l'outil fournit des vues différentes des données, à des niveaux de granularité différents, afin de soutenir le processus d'exploration, sans surcharger l'utilisateur avec trop d'informations. De plus, les transitions entre les différentes vues sont les plus transparentes et les plus naturelles possible pour réduire l'effort cognitif requis par l'utilisateur.

Notre approche visuelle affiche deux types d'information simultanément : les séquences d'appels dans une exécution et les données de l'analyse d'échappement combinée. Les séquences d'appels représentent une abstraction du flot de contrôle dynamique dans l'application, et sont extraites à partir d'une trace d'exécution. La métaphore permet d'afficher explicitement les invocations de méthodes ainsi que les régions du programme. Les données de l'analyse d'échappement sont ensuite ajoutées à la métaphore visuelle. Nous utilisons des abstractions naturelles qui sont facilement perceptibles (la taille et la couleur des entités visuelles) pour que l'attention de l'utilisateur se porte sur les parties importantes de la visualisation. Notre prototype supporte aussi un nombre de caractéristiques qui sont nécessaires au processus d'exploration typique suivi par les utilisateurs.

Pour finir, nous avons fait des études empiriques sur trois applications réelles de type *framework-intensive* (dont une industrielle). Ces études montrent l'utilité de l'approche par l'identification de divers cas de sources d'*"object churn"*.

1.4 Structure du mémoire

Ce mémoire est structuré comme suit. Dans le chapitre 2, nous discutons des travaux connexes existants. Dans le chapitre 3, nous expliquons la structure montrant les ap-

pels entre les méthodes que nous utilisons dans notre approche, ainsi que les différentes analyses appliquées sur ces structures. Dans le chapitre 4, nous détaillons notre outil de visualisation, Vasco, et ses fonctionnalités qui permettent de faciliter la recherche des sources d'*"object churn"* dans les structures d'appels. Puis, dans le chapitre 5, nous présentons trois études de cas qui démontrent l'efficacité de Vasco. Nous concluons en rappelant les points importants de ce mémoire, ainsi qu'en proposant plusieurs travaux futurs possibles.

CHAPITRE 2

ÉTAT DE L'ART

Dans ce chapitre, nous présentons des travaux existants dans le domaine de la visualisation visant à étudier le comportement des programmes. Ces travaux comprennent des travaux qui utilisent des traces d'exécution comme séquences d'événements, ainsi que des approches utilisant la structure du programme. Nous décrivons aussi les approches se basant sur une structure d'appels. Puis, nous présentons les travaux qui ont étudié les comportements des applications de grande taille.

2.1 Approches et outils utilisant la visualisation pour comprendre le comportement des programmes

Les approches les plus simples affichent toutes les informations d'une exécution sans analyser ni sélectionner les données. Par exemple, la majorité des profileurs existants présentent l'information recueillie sous forme de tableau. Sur chaque ligne figure de l'information telle que le nom de la méthode appelée, le nombre d'invocations, le temps d'exécution, etc. Par exemple, l'outil **TPTP** [31] (anciennement Hyades) utilise cette représentation (figure 2.1).

Highest 10 base time

Method	Class	Package	<Base Time ...	Average Base Time...	Cumulative Time...	Calls
instance(int) weka.core.Instance	Instances	weka.core	14.732695	0.000127	23.893199	115868
setInstances(weka.core.Instances) void	PreprocessPanel	weka.gui.explorer	13.037805	13.037805	13.037824	1
find(java.lang.Class, java.lang.String) java.util.Vector	ClassDiscovery	weka.core	10.677044	0.197723	13.580628	54
showDialog(java.awt.Component, java.lang.String) int	ConverterFileChooser	weka.gui	10.066024	5.033012	10.066812	2
elementAt(int) java.lang.Object	FastVector	weka.core	9.721956	0.000073	9.721956	132964
Main()	Main	weka.gui	8.579389	8.579389	8.579389	1
classIndex() int	Instance	weka.core	6.773018	0.000156	7.821481	43406
partition(int, int, int) int	Instances	weka.core	6.463654	0.001197	20.036775	5400
value(int) double	Instance	weka.core	5.715326	0.000059	5.715326	96806
LogWindow()	LogWindow	weka.gui	5.499440	5.499440	5.734171	1

Figure 2.1 – Exemple d'une visualisation par tableau : TPTP (Hyades)

2.1.1 Approches basées sur le temps

Une trace d'exécution est une séquence d'événements survenus à l'exécution (par exemple l'entrée ou la sortie d'une méthode). Les événements dans la trace sont ordonnés en fonction du moment auquel ils sont survenus durant l'exécution. Plusieurs approches permettent de visualiser les traces d'exécution en représentant le temps explicitement.

Par exemple, certaines approches se basent sur les graphiques à barres en utilisant un de leurs axes comme ligne de temps. Chaque appel à une méthode correspond donc à une barre dans le graphe. Les méthodes apparaissent dans leur ordre d'arrivée dans l'exécution du programme. L'autre axe est utilisé pour représenter une information relative à chaque méthode appelée. Une deuxième information sur la méthode peut être représentée par la couleur de la barre. Cette représentation des informations permet une comparaison facile et rapide entre deux méthodes. On voit par exemple, rapidement quelle méthode est représentée avec la plus grande taille.

EVolve [33] est un outil de visualisation des données d'exécution et du comportement d'un programme. Il a été développé avec comme objectif de pouvoir visualiser de nombreux types de données (flexible) et de pouvoir facilement ajouter de nouveaux types de visualisation (extensible). EVolve permet de visualiser une même trace d'exécution de différentes façons. La figure 2.2 illustre un exemple de graphique sur lequel sont représentées les invocations de méthodes (en classement lexical) sur l'axe des ordonnées et le temps en termes du nombre d'invocations sur l'axe des abscisses. Comme la représentation est trop compacte pour afficher l'identité des méthodes, le nom des entités apparaît au mouvement de la souris sur celles-ci.

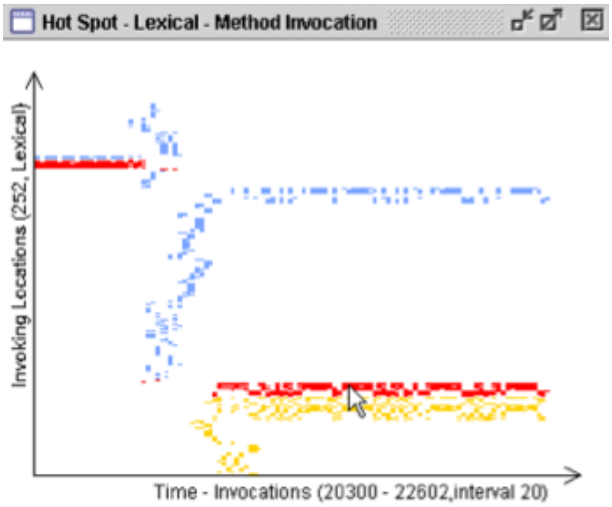


Figure 2.2 – Exemple d'une visualisation par graphique : EVolve

VisTrace [25] est un outil qui combine plusieurs vues dans le but de donner aux développeurs une interface riche qui permet une analyse du comportement du programme. VisTrace est un cadre générique pour la visualisation de traces des différents états (valeurs associées aux variables), plutôt que des traces contenant des invocations. L'outil permet un grand degré de flexibilité dans son exploration du contenu des traces. Une des vues supportées par VisTrace est une vue temporelle. Les données sont représentées sous forme de graphique à barres où l'axe des abscisses est une ligne temporelle et l'autre représente de l'information sur les différents états du système sélectionnée par un utilisateur. La figure 2.3 montre un exemple de cette vue. Chaque barre représente une entité (méthode ou classe), deux métriques sont associées à la hauteur des barres et à la couleur de celles-ci (à l'aide d'un dégradé de orange). Afin de permettre la visualisation d'une grande quantité de données, l'outil permet de visualiser la trace entière ou une plage sélectionnée grâce à une barre de contexte, située au-dessus de la visualisation principale.

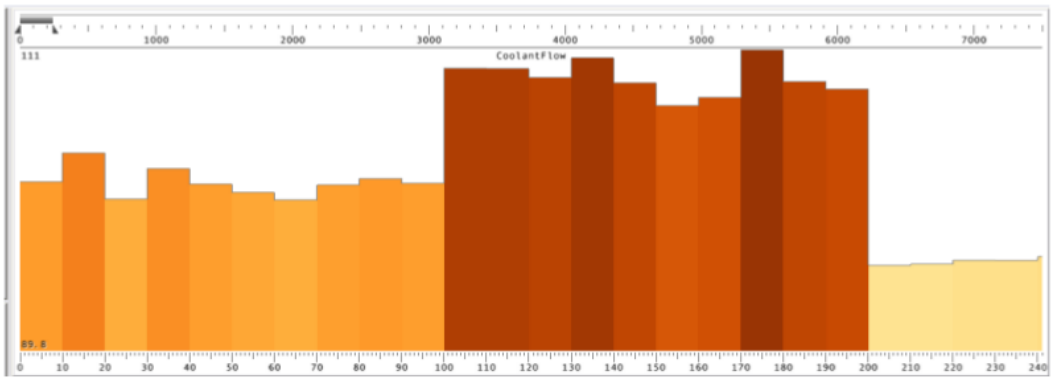


Figure 2.3 – Exemple d'une visualisation par graphique : VisTrace

Les traces d'exécution contiennent généralement une énorme quantité d'information. Les approches précédentes sont donc limitées dans leur représentation de l'information. Représenter toute l'information de façon linéaire rend donc la visualisation illisible. Afin de permettre la visualisation efficace de traces de grande taille, Jerding et al. [16] ont introduit la métaphore du mur d'information ("Information mural") qui permet d'utiliser deux représentations simultanément à différents niveaux de granularité. Jerding et al. définissent un mur d'information comme une méthode de représentation qui affiche un espace d'information dans une vue. Lorsque l'information est une trace d'exécution, la méthode de représentation s'appelle mur d'exécution ("Execution mural"). Cette métaphore consiste en une représentation miniature de la trace et donne un aperçu rapide des différentes phases de l'exécution d'un programme. Cette méthode a été utilisée avec succès pour trouver des séquences répétitives.

Deux outils, **Extravis** et **IsVis**, utilisent cette métaphore. Dans les deux cas, la structure du système est représentée ainsi que les liens d'appels permettant de comprendre globalement le comportement du programme. Le mur d'information permet de donner une vue globale de la trace dynamique ; une deuxième vue permet de la détailler.

Extravis [8] est un outil qui utilise deux visualisations, la vue de séquences massives (*"massive sequence view"*) et la vue de liasses circulaires (*"circular bundle view"*). La première donne un aperçu rapide de la trace d'exécution complète. Sur le haut se trouvent les différentes classes du programme, et en dessous les liens d'appels entre ces classes. Les liens sont représentés par des lignes qui fonctionnent grâce à un code de couleur,

du vert au rouge, qui signifie que la classe sous laquelle commence la ligne en vert a une méthode qui appelle une autre qui se trouve dans la classe où se termine la ligne en rouge. Les liens sont représentés selon leur ordre d'exécution. La deuxième vue montre en détail un passage sélectionné de la trace d'exécution. La figure 2.4 montre un exemple de visualisation à l'aide d'Extravis.

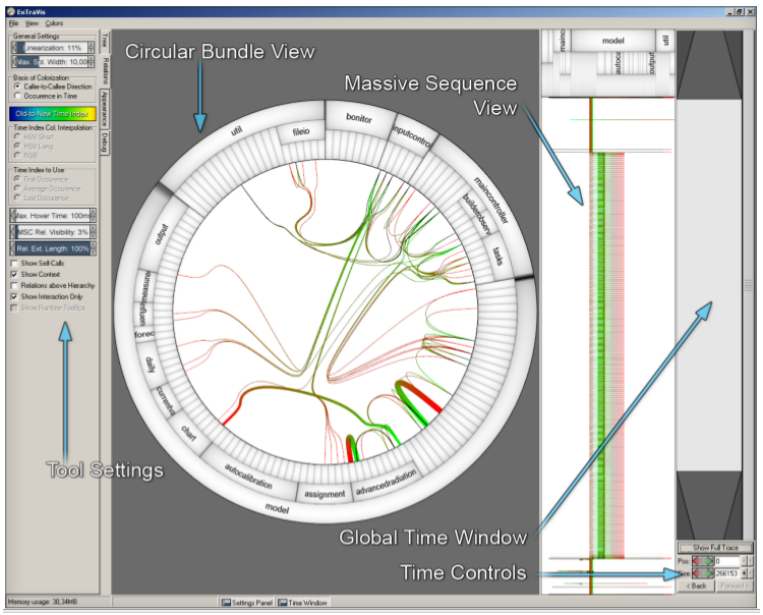


Figure 2.4 – Extravis

IsVis [15] est un outil qui a été développé dans le but de supporter la navigation et l'analyse des traces d'événements provenant d'exécution de programmes. Il est utile lors de tâches de génie logiciel nécessitant une compréhension du logiciel. Il combine deux vues utilisant deux types de diagrammes : le mur d'exécution et le diagramme de messages temporels (figure 2.5). Le mur d'exécution permet la navigation dans la trace alors que le diagramme de messages temporels fournit plusieurs fonctionnalités pour aider un analyste à construire un modèle abstrait du système ainsi que de comprendre le comportement du programme.

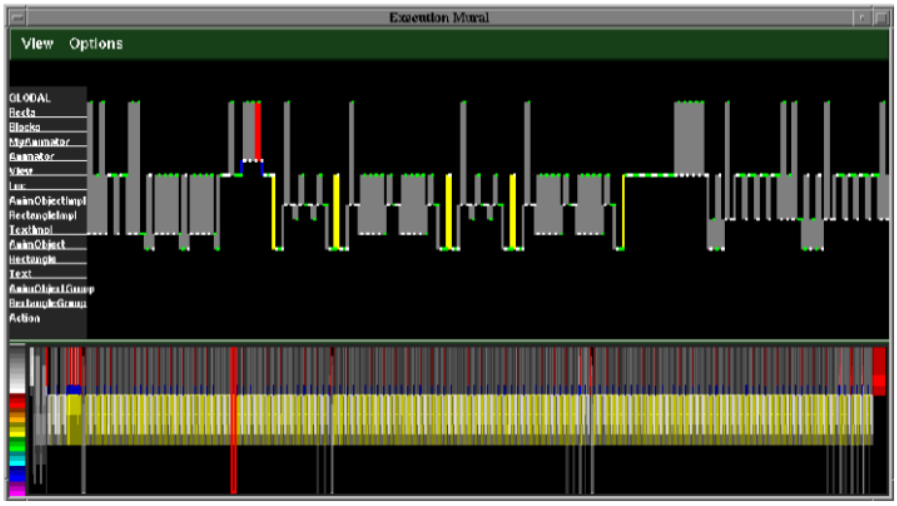


Figure 2.5 – IsVis

Les approches basées sur le mur d'information représentent la trace de façon linéaire, et sont donc limitées au nombre de lignes disponibles soit horizontalement ou verticalement pour la représentation miniature de la trace. Renieris et Reiss [26] ont proposé **Almost**, un outil qui représente les données de traces temporelles sous forme d'une spirale pour maximiser l'utilisation de l'espace. L'outil combine deux vues, l'une horizontale et l'autre en spirale. Ces deux représentations de données de la trace sont simultanées et présentent différents niveaux de granularité. La vue en spirale représente la trace complète de l'exécution (figure 2.6(b)). Cette vue permet de facilement reconnaître les différentes phases d'une exécution. La vue horizontale (figure 2.6(a)) permet de détailler une partie de la spirale. L'axe horizontal représente le temps (de gauche à droite) et l'axe vertical représente le niveau d'appel de la méthode. Les méthodes sont représentées par des barres dont la taille est définie par leur temps d'exécution.

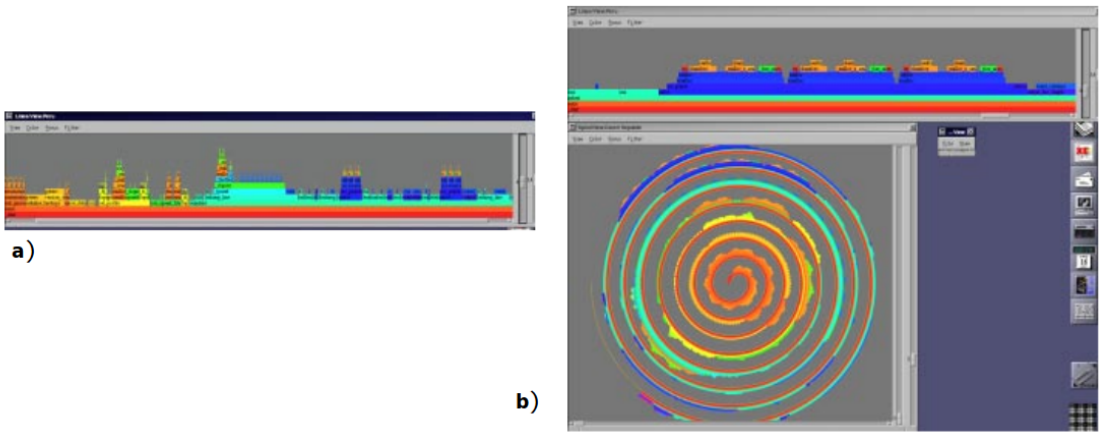


Figure 2.6 – Almost

Les approches reposant sur des visualisations de traces détaillées représentent l'information à un niveau de granularité trop fin pour être applicables à l'identification des objets temporaires pour les applications de grande taille. Ces techniques passent difficilement à l'échelle lorsque la taille de la trace augmente, et ne permettent pas de visualiser l'information avec suffisamment d'abstraction.

2.1.2 Approches basées sur la structure

Certaines approches favorisent un autre type de fichier d'entrée au lieu des événements d'une trace d'exécution. Elles se concentrent sur la structure du programme.

La métaphore du TreeMap [17] a été introduite au début des années 1990 par Shneiderman dans le but d'organiser de manière visuelle le contenu des disques durs d'un ordinateur, représenté par des arbres. Grâce à cette technique, l'utilisateur peut voir plus facilement l'organisation de ses fichiers et dossiers. La métaphore du TreeMap utilise un algorithme basé sur une technique de division de l'espace (figure 2.7). Un rectangle représente la racine de l'arbre puis à chaque niveau, les enfants divisent le rectangle en sous rectangles. La division se fait alternativement horizontalement et verticalement.

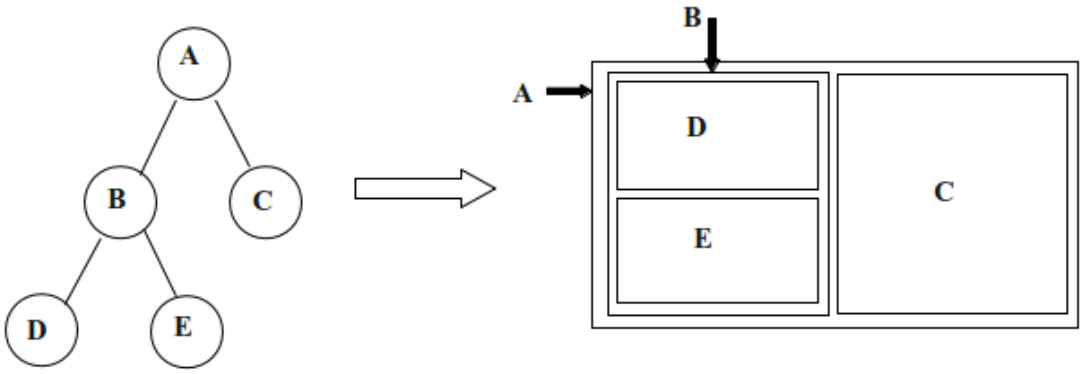


Figure 2.7 – Algorithme du TreeMap

Cette métaphore utilise de façon optimisée l'espace, il est donc possible de visualiser des programmes de grande taille. De plus, la taille et la couleur d'un rectangle peuvent être utilisées pour permettre la visualisation d'information additionnelle à propos des éléments représentés. Cette métaphore permet une compréhension rapide, et est encore utilisée dans plusieurs outils permettant de visualiser une hiérarchie de fichiers (par exemple dans le but d'identifier et supprimer les fichiers ou dossiers utilisant beaucoup d'espace).

Lorsqu'appliquée à la visualisation du comportement d'un programme, cette métaphore présente certaines limitations. Plus la taille du programme est grande, plus il est difficile de comprendre les liens hiérarchiques entre les différentes entités. Par exemple, la figure 2.8 montre une représentation utilisant le TreeMap pour un million d'items.

Figure 2.8 – TreeMap de grande taille

Effectivement, cette métaphore est plus utile pour les cas d'inclusion (contenu/contenant) telle que la hiérarchie des dossiers et fichiers. La visualisation des traces d'exécution requiert une métaphore utilisant une représentation plus claire des liens.

Langelier et al. [20] ont proposé VERSO, un outil basé sur un algorithme de TreeMap modifié. Ils utilisent une métaphore 2.5D qui représente la structure du programme à différents niveaux de granularité (figure 2.9). Chaque classe est représentée par un pavé et chaque interface par un cylindre. Ces dernières se trouvent dans des packages qui sont divisés à l'aide de l'algorithme du TreeMap. Différentes informations peuvent être associées aux attributs visuels des entités : la couleur, la taille et l'orientation des pavés. Bouvier et al. ont ajouté la visualisation des appels à VERSO en utilisant la technique des grappes de liens (*edge bundles*). La figure 2.10 montre un exemple de visualisation des liens dans VERSO.

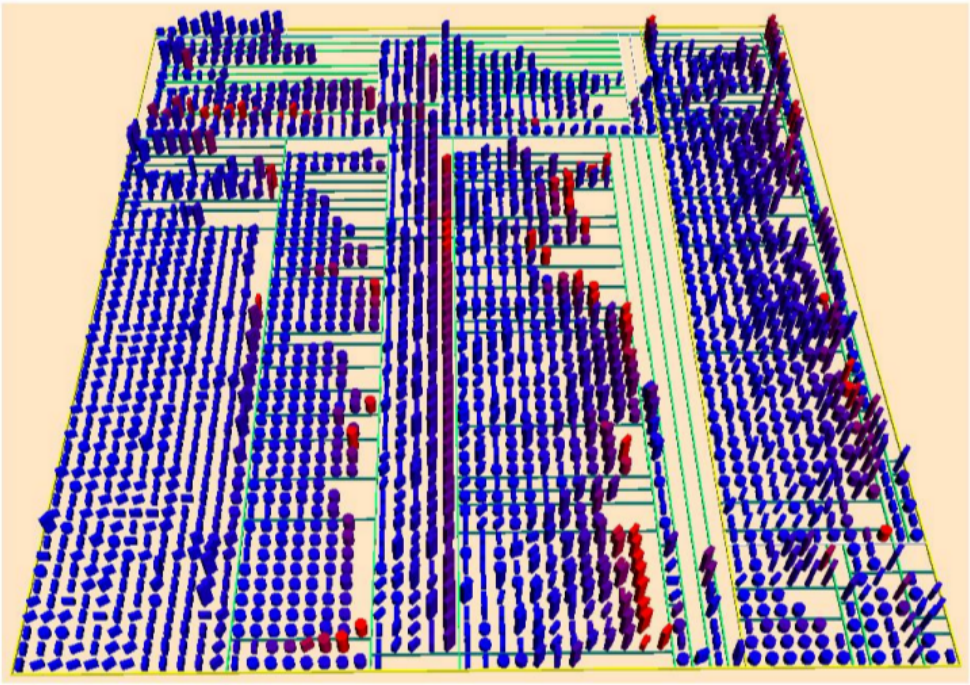


Figure 2.9 – Exemple : représentation de Azureus en utilisant VERSO, tiré de [19]

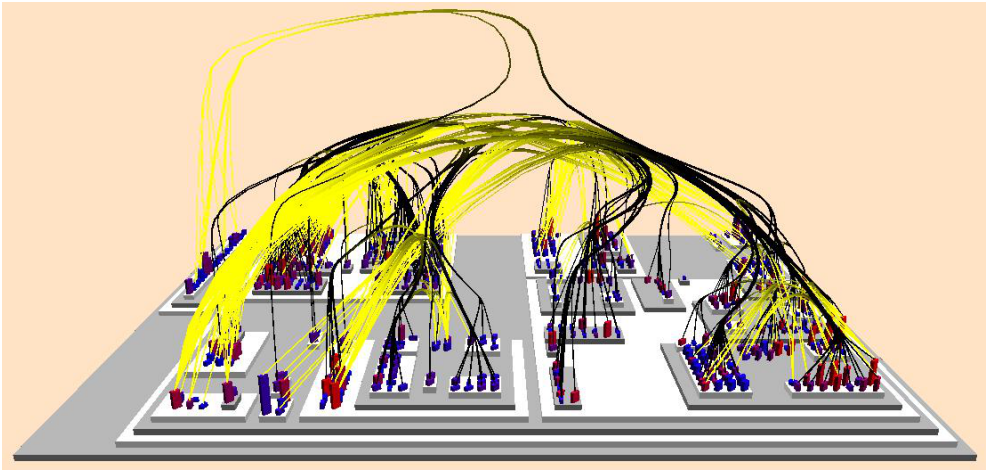


Figure 2.10 – Exemple : représentation des liens d'invocation par VERSO.

D'autres outils, tels que **Evospace** [11], utilisent une métaphore 2.5D similaire représentant une ville, où les classes sont visuellement représentées comme des bâtiments dans la ville et disposés sur un plan, et les packages sont les quartiers. Cette métaphore se base sur l'algorithme du TreeMap. Les appels entre les classes sont représentés avec des liens

dans l'espace 3D entre les bâtiments. L'affichage de la trace au complet est impossible à cause du grand nombre de données. Les auteurs ont donc développé une technique de segmentation de la trace et de filtrage de l'information qui est appliquée afin de réduire de manière significative les données à afficher. La figure 2.11 montre un exemple d'utilisation de cet outil. L'environnement familier utilisé (bâtiments, quartiers...) peut aider à comprendre plus rapidement la visualisation.

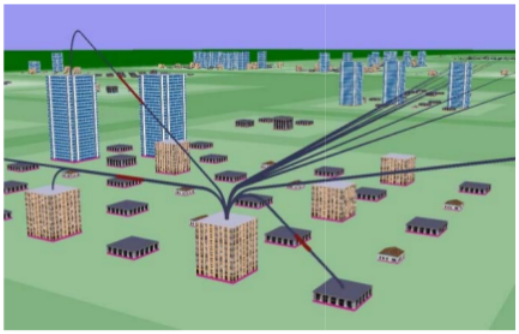


Figure 2.11 – Evospace

Les approches de visualisation basées sur la structure du logiciel plutôt que son comportement passent plus facilement à l'échelle, mais ne permettent pas de visualiser des régions de l'exécution de façon explicite et intuitive.

2.1.3 Approches basées sur une représentation d'appels

D'autres approches se concentrent sur la représentation explicite des appels. Une approche simple consiste à représenter les appels à l'aide d'un diagramme de séquence UML. Un diagramme de séquence montre les différentes interactions entre les objets, c'est-à-dire les appels de méthodes, lors d'une exécution. **OSE** [4] est un exemple d'outil permettant de visualiser de tels diagrammes à différents niveaux de détails. La figure 2.12 montre les différentes vues utilisées dans l'outil OSE. Les différents appels de méthodes apparaissent dans leur ordre d'exécution du haut vers le bas de la visualisation. La partie B contient la trace entière et la partie A montre une partie de cette trace en détail.

En raison de sa représentation détaillée du comportement, cette approche n'est généralement utilisée que pour représenter un seul cas d'utilisation ne contenant que quelques

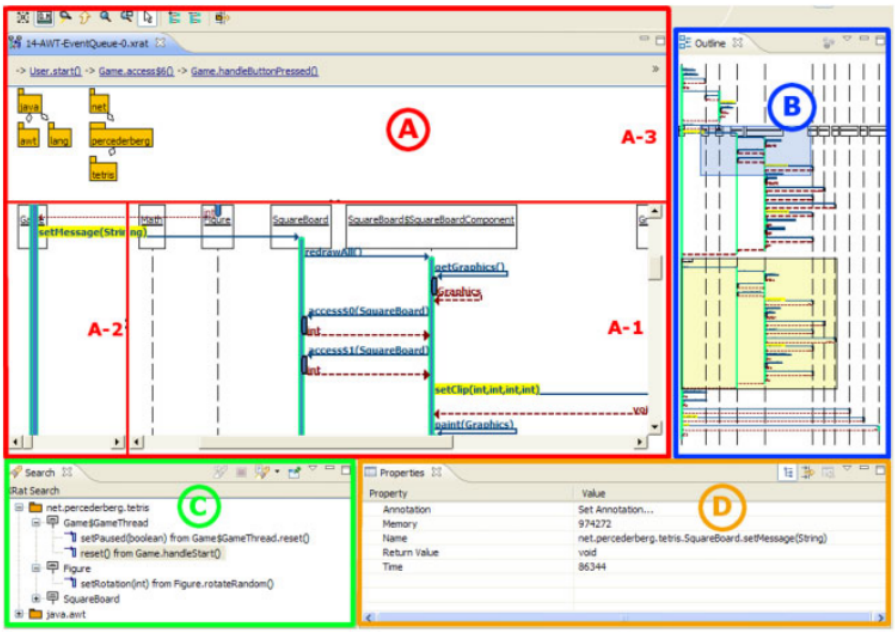


Figure 2.12 – Exemple d'une visualisation par diagramme UML : OSE

appels et ne pourrait pas s'étendre à des programmes de grande taille.

D'autres approches représentent les arbres d'appels sous forme visuelle. Par exemple, **Program Explorer** [18] supporte deux vues basées sur des arbres : une des objets et l'autre des appels. Sur la première, chaque nœud de l'arbre représente un objet et chaque arc représente une invocation des méthodes. Sur la deuxième, chaque nœud de l'arbre représente une invocation et chaque arc représente les relations d'appels entre les méthodes. À la base, cet outil était utilisé pour visualiser du code C++, mais il peut être étendu à d'autres langages orientés objet. La figure 2.13 montre la représentation d'un arbre d'objets par Program Explorer. Bien que cette visualisation est intéressante par sa simplicité de compréhension et sa capacité à représenter facilement des informations, elle est limitée à de petits programmes à cause d'une représentation du comportement détaillée.

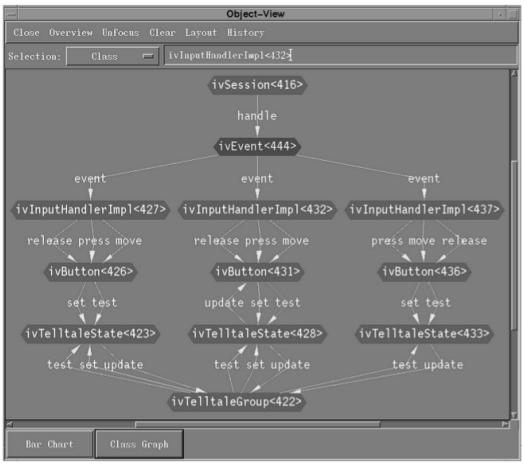


Figure 2.13 – Exemple d'une visualisation par graphe : Program Explorer

Pour permettre l'extensibilité, d'autres approches se basent plutôt sur la métaphore du Sunburst introduite par Stasko et al. [29]. Cette métaphore représente un arbre sous forme de cercles concentriques. La racine de l'arbre est placée au centre de la visualisation et les enfants sont placés autour de la racine au même niveau que celui qu'ils ont dans l'arbre. L'espace de chaque niveau est partagé entre chaque enfant en fonction de son importance pour le parent. La figure 2.14 montre un exemple de Sunburst.

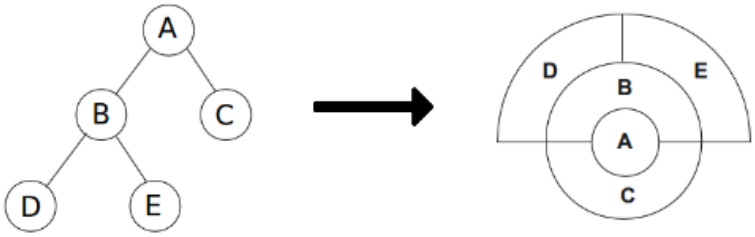


Figure 2.14 – Visualisation d'un arbre d'appels en Sunburst

Le Sunburst a une représentation explicite de la structure de l'arbre, ce qui permet une compréhension rapide de la hiérarchie. De plus, il permet une bonne utilisation de l'espace, grâce à sa représentation implicite des liens hiérarchiques (il n'y a donc pas de sacrifice d'espace pour ceux-ci).

Moret et al. [24] ont développé un outil explorant et visualisant des arbres d'appels grâce au Sunburst : **Calling Context Ring Charts (CCRCs)**. Ils utilisent des arbres de contexte d'appels (*"Calling Context Tree"* ou CCT) en entrée de leur outil. Un contexte d'appels (*"Calling Context"* ou CT) correspond à une séquence de méthodes représentée sur la pile des appels à un moment durant l'exécution du programme. Deux méthodes qui sont invoquées dans des contextes d'appels différents sont représentées par des nœuds distincts dans le CCT. Au contraire, deux méthodes qui sont invoquées dans le même contexte d'appels sont représentées par un nœud identique.

Dans cet outil, l'utilisateur a un choix de trois visualisations. La première montre seulement les contextes d'appels sans associer aucune information aux attributs visuels. La deuxième représente les contextes d'appels et une information est associée à l'angle des arcs, qui est proportionnel à la contribution du contexte d'appels correspondant à une mesure dynamique choisie, relative à la contribution du parent respectif. Avec une telle visualisation, l'aire des arcs peut être trompeuse, un arc proche de la racine peut avoir une plus petite aire que celle qu'il aurait s'il se trouvait plus loin de la racine (figure 2.15). Pour minimiser ce problème, la troisième visualisation représente les contextes d'appels et une information associée à l'angle et l'épaisseur des arcs qui diminue en fonction de l'éloignement de l'arc de la racine (figure 2.16).

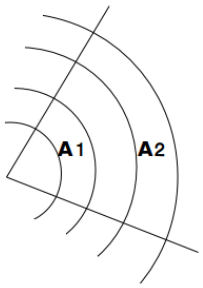


Figure 2.15 – L'aire A1 est plus petite que l'aire A2 alors que les deux arcs ont le même angle

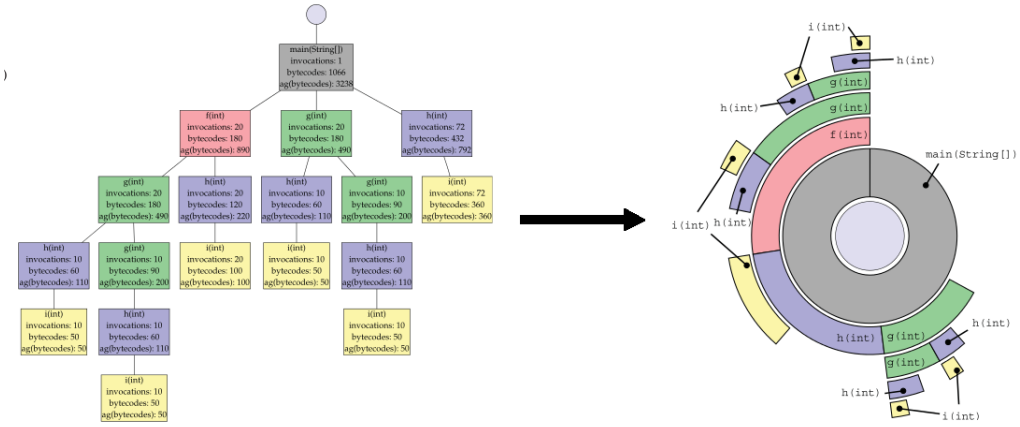


Figure 2.16 – Exemple d'un CCT représenté par l'outil CCRCs avec la troisième visualisation

Pour pouvoir visualiser de grands CCT, CCRCs permet la sélection de sous arbres et la limitation de la profondeur. De plus, l'outil utilise des CCT transformés en éliminant la récursion des sous-arbres. Puisqu'il peut être difficile de connaître la contribution d'une méthode si elle est utilisée dans plusieurs contextes d'appels différents, Moret et al. proposent une transformation qui ignore toutes les informations du contexte d'appels. Chaque noeud du CCT est représenté par un seul noeud qui stocke la somme des informations pour cette méthode. L'arbre qui en résulte a donc une profondeur de 1 et peut être représenté comme un camembert. L'outil peut appliquer cette transformation à des sous-arbres du CCT (figure 2.17).

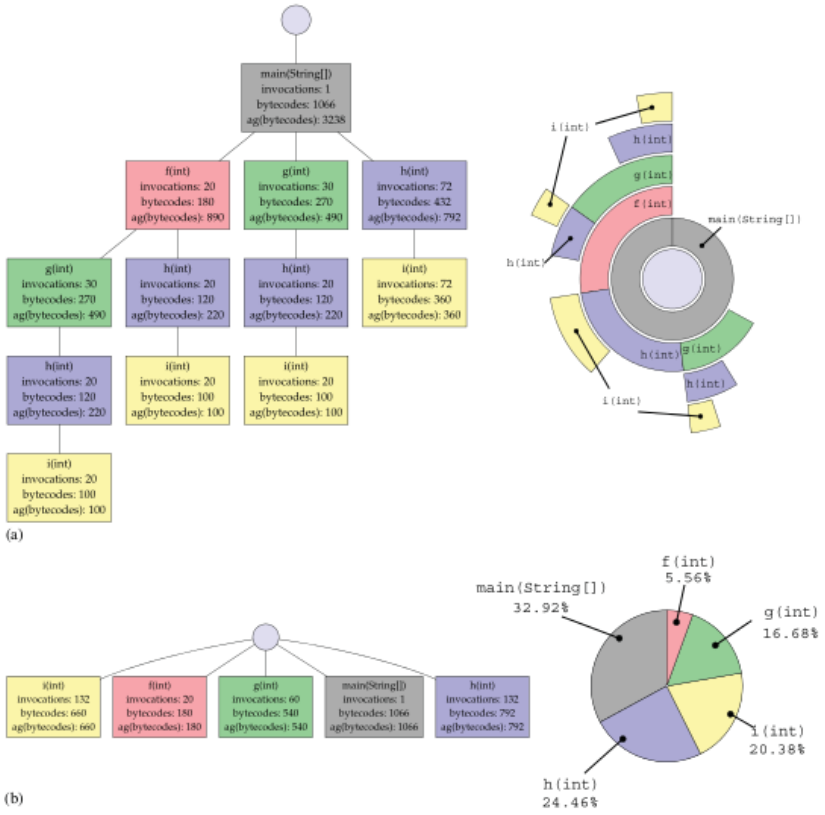


Figure 2.17 – a) Élimination des récursions (par rapport à la figure 2.16), b) camembert qui regroupe toutes les invocations d'une méthode en un nœud

Moret et al. ont réalisé deux études de cas sur JLex (analyseur lexical) et Shamir's Secret Sharing (cryptographie). Les deux études ont montré que l'outil aide à localiser rapidement les endroits à problèmes dans des profils de contexte d'appels. Ils ont pu effectuer une optimisation des applications qui a entraîné une réduction significative du temps d'exécution.

Adamoli et Hauswirth [1] ont développé Trevis, un outil de visualisation générique et extensible basé sur la représentation en Sunburst d'un CCT. Trevis supporte plusieurs opérations visant à étudier et comparer plusieurs CCTs. La figure 2.18 montre un exemple de CCT tel que visualisé par Trevis. L'efficacité de Trevis est évaluée dans le contexte d'un autre outil de profilage, FlyBy, qui vise à identifier la cause de problèmes de performance. FlyBy permet de recueillir de l'information de profilage lorsque des problèmes

de performance sont observés durant l'exécution d'un programme. Cette information est alors utilisée pour identifier les régions du programme responsables du ralentissement.

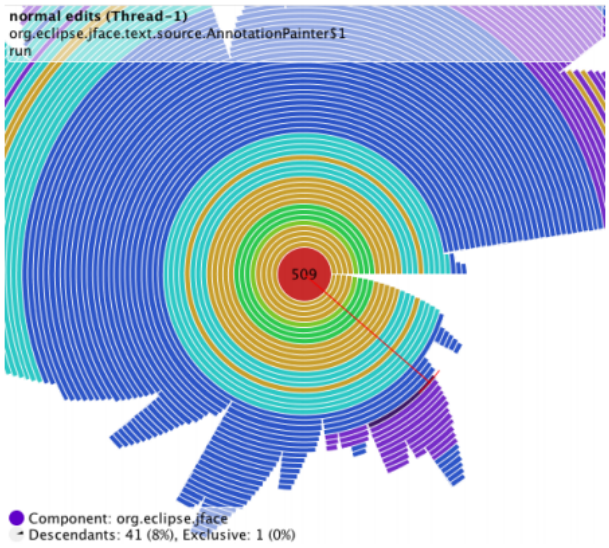


Figure 2.18 – Exemple de visualisation d'un CCT par Trevis

Fisher et al. [12] ont développé un plugin Eclipse, HI-C, qui permet d'explorer et de comprendre les résultats des analyses sur les exécutions. Ce plugin fournit des visualisations dynamiques et interactives. Il présente deux vues, dans la première, un graphe d'appels ou un CCT représente les liens d'appels entre les méthodes. Dans la deuxième vue, un graphe de connexions représente les relations de références entre les objets dans une méthode donnée. La figure 2.19 montre un exemple d'utilisation du plugin. Les rectangles arrondis représentent un contexte d'appels (CCT) ou une méthode (graphe d'appels). Les chiffres affichés dans le CCT sur les rectangles correspondent aux nombres d'instances capturées, la couleur permet de différencier plus rapidement les CT. Le rouge est choisi quand il y a beaucoup d'instances capturées, l'orange ou le jaune quand il y en a moins, et le vert quand il y en a peu ou pas du tout.

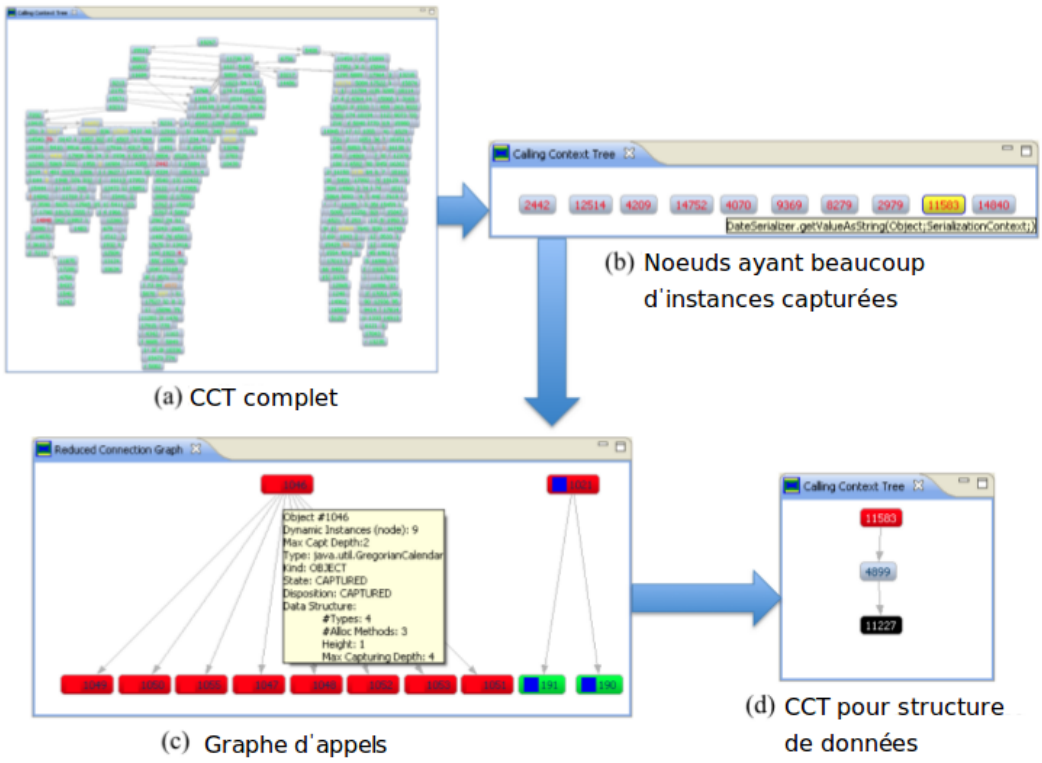


Figure 2.19 – Exemple d'utilisation du plugin HI-C, tiré de [12]

On remarque sur la figure 2.19(a) qu'il y a beaucoup de nœuds et qu'il est donc difficile de naviguer et d'identifier les CT qui nous intéressent. Les utilisateurs peuvent alors choisir de n'afficher que les CT qui ont un fort taux d'instances capturées (figure 2.19(b)). Ils peuvent aussi sélectionner un nœud dans le CCT pour afficher le graphe d'appels correspondant (figure 2.19(c)). Chaque noeud dans le graphe d'appels correspond à un objet utilisé dans l'analyse d'échappement et chaque arc indique un lien de référence entre ces objets. A l'intérieur du graphe d'appels, les noeuds sont colorés grâce à un code couleur qui indique le statut d'échappement correspondant à l'objet. Le rouge indique que l'objet est capturé, bleu que l'objet est échappé par argument et le vert que l'objet est globalement échappé. Lorsque l'utilisateur sélectionne un noeud du graphe d'appels, le panneau du CCT change pour montrer les contextes où les objets correspondants sont visibles (figure 2.19(d)).

Bien que les auteurs aient démontré que Hi-C permet de détecter certaines sources

d'objets temporaires, l'outil présente des limitations importantes en pratique. Premièrement, Hi-C utilise une représentation en arbre pour les appels qui rend difficile l'identification des régions de l'exécution. Deuxièment, afin de réduire la quantité de nœuds affichés, Hi-C ne montre que certains contextes jugés *intéressants*, c'est-à-dire ceux qui allouent ou capturent des objets. Il est donc difficile pour un utilisateur de suivre le flot des objets à travers les appels puisque la chaîne d'appels n'est pas représentée explicitement par la métaphore. De plus, la nature conservatrice de l'analyse implique que la véritable cause du problème est souvent absente de la visualisation. Finalement, la métaphore utilisée par Hi-C reflète la structure des données brutes générées par une analyse automatisée, et n'est donc pas facile à inspecter par des utilisateurs.

2.2 Autres études du comportement des applications de grande taille

De nombreuses études se sont aussi concentrées sur le caractère des comportements des applications utilisant des *frameworks* intensivement, par exemple, pour identifier des problèmes de performance. Ammons [3] propose une approche pour identifier les chemins vitaux de l'exécution dans des systèmes de grande taille en utilisant une analyse dynamique. Il part du principe que trouver ces chemins vitaux permettrait d'améliorer considérablement le temps d'exécution. Srinivas [28] propose une technique pour identifier les méthodes responsables d'une partie importante du temps d'exécution dans un système utilisant intensivement des *frameworks*. Ces applications sont souvent assemblées à partir de plusieurs couches de composants qui ont été développés par des équipes différentes de développeurs. Elles ont donc des chemins d'appels profonds et de nombreuses invocations pouvant causer des problèmes de performance. Cette technique a été testée sur sept applications réelles et a permis l'identification d'un petit ensemble d'invocations (10 à 93) coûteux en temps d'exécution qui représentait 82 à 99% des coûts de la performance globale de l'application. Tripp et al. [32] ont développé une analyse de teinte (*taint analysis*) visant spécialement les applications web.

Beaucoup de travaux ont été faits sur l'utilisation excessive des ressources. Mitchell et al. ont étudié beaucoup d'aspects de l'utilisation excessive des ressources. Les fuites

de mémoire peuvent causer des défaillances (*crashs*) de serveurs, alors que les serveurs sont de plus en plus utilisés dans les applications Java. Ils sont souvent de grande taille et ont un usage intensif de *frameworks*. Il est donc difficile de retrouver les régions responsables de la fuite de mémoire. Mitchell et al. [23] ont proposé un nouvel outil, LeakBot, qui permet de diagnostiquer les régions responsables. Quelques années plus tard, Mitchell [22] et son équipe ont analysé beaucoup d'applications qui font un usage intensif de *frameworks*. Ils ont conclu que même pour des fonctionnalités simples, les applications effectuent typiquement une quantité de travail étonnamment élevée. Une grande partie de cette activité implique la transformation de l'information en raison du couplage avec le *framework*. Ils proposent une nouvelle approche de modélisation et de quantification du comportement pour savoir quelles transformations accomplir. En 2009, Mitchell et al. [21] ont développé un nouvel outil Yeti qui résume l'utilisation de la mémoire afin de prévoir les coûts de conception. Yeti est utilisé par les équipes de développement et de service au sein d'IBM et a été efficace dans la résolution de nombreux problèmes.

Chis et al. [6] ont proposé une caractérisation des sources d'inefficacité dans les applications industrielles de grande taille. Les applications Java de grande taille peuvent facilement consommer de manière excessive la mémoire. Les auteurs ont travaillé durant deux ans avec des équipes de tests d'applications Java de grande taille. Ces équipes mesurent pendant leurs tests la mémoire maximale utilisée par les programmes, mais ils ne comprennent pas d'où provient cette consommation excessive de la mémoire. Les auteurs ont développé une approche qui est basée sur des modèles de mémoire. Ils ont étudié des centaines d'applications pour rassembler des modèles faits à partir d'erreurs de conception trouvées dans ces applications. Par exemple, il est commun que les développeurs utilisent les constructeurs par défaut des collections : new ArrayList(). Si le code ne stocke que quelques objets dans ces listes, seulement quelques-uns des pointeurs alloués sont utilisés. Cette approche, si elle est basée sur un bonne gamme de modèles, permet d'expliquer la plupart des cas de mémoire utilisée excessivement. Les auteurs ont classé onze modèles de mémoire les plus couramment utilisés, comme les collections vides, ou les collections avec des tailles fixes. De plus, ils ont développé le modèle

ContainerOrContained, une abstraction qui peut être utilisée pour la détection des occurrences des différents modèles de mémoire. Pour finir, les auteurs ont développé une analyse qui détecte les occurrences des modèles et un outil qui implémente cette analyse. Cet outil est utilisé par les équipes de tests au sein d'IBM

Zhao et al. [38] ont étudié l'extensibilité des applications Java sur des plates-formes multicoeurs. Ils ont réalisé deux études de cas afin de comprendre si les problèmes de performance peuvent être résolus au niveau de la micro-architecture (mémoire, processeur, coeurs) ou au niveau du code source. L'étude porte sur deux benchmarks populaires SPECjbb2005 et SPECjvm2008 sur une plate-forme multicoeurs incluant Intel Clovertown et AMD Phenom. Ils ont analysé le comportement de ces programmes de tests et ont identifié une corrélation entre la dégradation de l'extensibilité et le taux d'allocations d'objets. Ce taux d'allocations d'objets est limité par la mémoire en écriture disponible. Ils ont donc mesuré pour ces programmes à fort taux d'allocations, la bande passante d'écriture en mémoire et celle disponible par le matériel. Ils ont pu en conclure que la bande passante d'écriture en mémoire est le facteur limitant des applications Java qui allouent intensivement sur les nouvelles plates-formes multicoeurs. Ils ont appelé ce phénomène "mur d'allocation". Après une étude sur la manière dont Java gère la mémoire, ils ont remarqué que les problèmes d'efficacité sont liés au mur d'allocation. Les auteurs ont finalement analysé les conséquences de l'ajout de coeurs au processeur sur la performance des programmes. Malheureusement, le problème ne peut pas être résolu au niveau du matériel car ajouter des coeurs au processeur n'a pas amélioré la performance. Donc, la seule solution est de résoudre ce problème de performance au niveau code source. Actuellement, la seule solution efficace est de réduire le taux d'objets alloués. L'analyse d'échappement peut réduire ce taux d'objets alloués, mais seulement 12% des objets alloués sont éliminés. Ce n'est pas suffisant pour améliorer la performance des programmes. La réutilisation d'objets est efficace mais elle est faite manuellement. De plus, le profilage et la modification du code pour la réutilisation des objets sont difficiles pour les applications de grande taille et complexes.

Shankar et al. [27] ont développé une technique afin d'éliminer automatiquement certaines sources du *churn* durant la compilation *just-in-time* (JIT). Ces sources sont

souvent présentes dans les applications ayant une utilisation intensive des *frameworks*. Shankar et al. ont développé deux mesures, capture et contrôle, afin de trouver les sources du *churn* dans le graphe d'appels. Ils ont ensuite, développé des analyses dynamiques pour mesurer ces deux métriques. Pour finir, ils ont développé un algorithme qui permet l'optimisation des exécutions en enlevant le *churn* grâce à une analyse d'échappement. JOLT est l'optimisateur utilisant cet algorithme. Les auteurs ont intégré JOLT dans le compilateur JIT de la JVM commerciale d'IBM J9 et ont évalués JOLT sur une application de grande taille utilisant de nombreux *frameworks*, incluant Eclipse et JBoss. L'approche que nous nous proposons de développer doit être complémentaire à la leur, dans le fait que cela permet aux développeurs d'identifier et de corriger les sources du *churn* qui ne peuvent pas être automatiquement optimisées.

Plusieurs approches se sont penchées sur l'identification automatique de certains problèmes de performance liés au type d'applications visées par ce travail. Xu et al. ont proposé des approches automatiques permettant d'identifier les opérations excessives de copie entre des objets [34], les conteneurs utilisés inefficacement [37], les structures de données à faible utilité [36] et les fuites de mémoire [35]. Bhattacharya et al. [5] ont proposé une approche permettant d'identifier certains objets temporaires créés dans des boucles. Malgré l'avancement dans ce domaine, il reste nécessaire de développer des outils permettant aux développeurs d'identifier les objets temporaires pour lesquels aucune technique n'est encore disponible, et pour faciliter la compréhension des résultats générés par les techniques automatiques.

CHAPITRE 3

COLLECTE DES DONNÉES À VISUALISER

Les traces d'exécution contiennent des informations sur l'exécution d'un programme. Elles peuvent aider à la compréhension du comportement d'un programme. Pour cela, il faut les analyser afin de garder ou de calculer les informations nécessaires à l'accomplissement de notre tâche, qui est la recherche des sources *d'object churn*. Nous réutilisons une analyse d'échappement combinée existante dans le but de trouver les informations sur les allocations et captures des méthodes. Dans ce chapitre, nous décrivons les traces d'exécution dont nous nous servons, puis nous expliquons l'analyse d'échappement combinée qui sert à obtenir une structure d'appels agrégée qui permet la visualisation avec un compromis entre l'extensibilité et la précision des données.

3.1 Traces d'exécution et structure d'appels

Pour identifier une région avec une utilisation excessive d'objets temporaires, nous avons besoin d'informations sur le flot de contrôle durant une exécution du programme qui consiste à connaître le déplacement du contrôle d'une méthode à une autre. Une région est un sous-arbre de l'arbre d'appels. Une trace d'exécution est typiquement constituée d'une série d'événements contenant en particulier les entrées et sorties des méthodes. De nombreuses techniques et outils permettent de récupérer les traces d'exécution, par exemple l'outil Jinsight [9]. Jinsight permet aussi d'enregistrer d'autres événements de l'exécution, par exemple le type des allocations et le nombre d'allocations de chaque type. Nous utilisons Jinsight pour produire des traces d'exécution contenant une séquence d'invocations de méthodes ainsi que les informations sur les objets créés par chacune des invocations.

Une trace d'exécution peut aussi être vue comme un arbre où chaque nœud représente une invocation de méthode et les arcs représentent les relations appelante/appelée entre les invocations (figure 3.1(a)). Cette structure est appelée arbre d'appel . Les arbres d'ap-

pels sont très précis, car ils contiennent toutes les informations, mais ils sont trop grands pour être visualisés. On peut obtenir des arbres plus concis en agrégeant les nœuds dans l'arbre d'appels. À l'extrême, tous les nœuds qui représentent des invocations d'une même méthode seraient fusionnés. On obtiendrait alors un graphe d'appels qui est un graphe dirigé où les nœuds représentent les méthodes du programme et les arcs représentent les appels entre les méthodes (figure 3.1(b)). Bien que cette stratégie d'agrégation réduit considérablement la taille de la structure, il y a une perte importante d'informations précieuses sur le contexte. Par exemple, dans la figure 3.1(b) le chemin M-D-A-C est possible alors qu'il est inexistant dans l'arbre d'appels dynamique.

Pour contrer ce problème, nous utilisons des arbres de contexte d'appels (*"Calling Context Tree"* ou CCT). Un contexte d'appels correspond aux méthodes représentées sur la pile des appels à un moment donné durant l'exécution du programme. Les arbres de contexte d'appels ont été introduits par Ammons et al. [2]. Chaque nœud d'un CCT correspond à un contexte d'appels et enregistre les résultats des mesures dynamiques. Si la même méthode est appelée dans des contextes d'appels distincts, les différentes invocations sont représentées par des nœuds distincts dans le CCT. Par contre, si elle est appelée plusieurs fois dans le même contexte d'appels, alors les différentes invocations ne sont représentées qu'une seule fois et les mesures dynamiques sur les objets créés sont enregistrées dans le même nœud du CCT. Les arbres de contexte d'appels (figure 3.1(c)) permettent une agrégation en perdant moins de précision que le graphe d'appels. Par exemple, dans la figure 3.1(c), le chemin M-D-A-C qui était possible dans le graphe d'appels ne l'est pas dans le CCT.

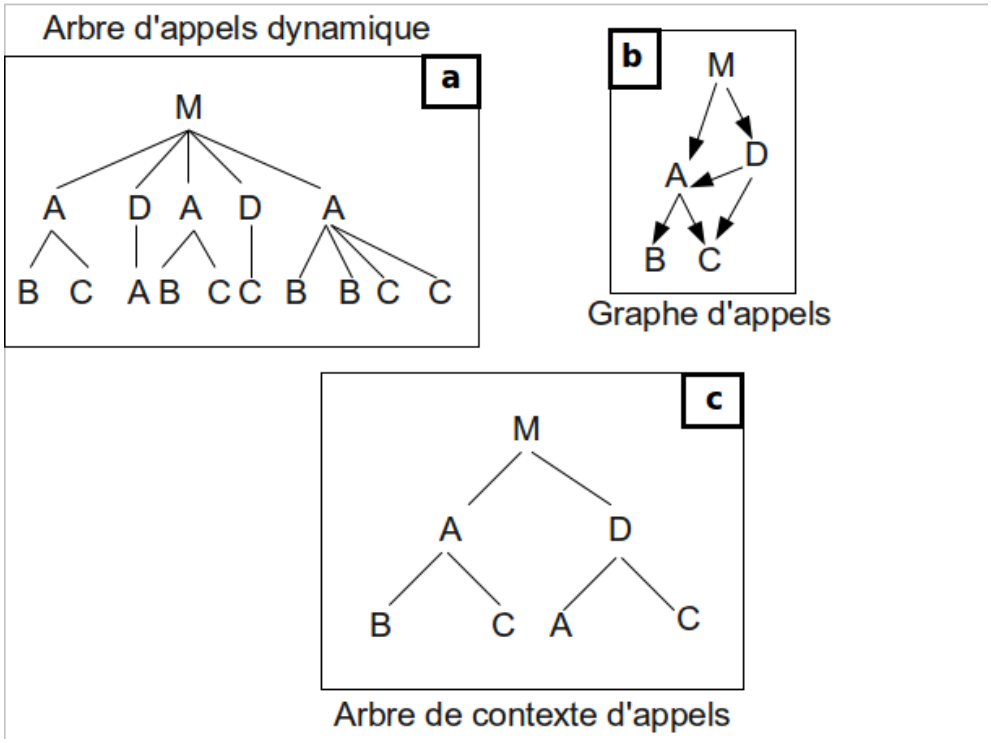


Figure 3.1 – Exemple d'un CCT comparé avec un graphe d'appels et un arbre d'appels dynamique

La définition originale du CCT permet les cycles en présence d'appels récursifs. S'il est nécessaire d'avoir une structure d'arbre, il est possible d'interdire les cycles et de représenter tous les appels explicitement dans le CCT.

3.2 Analyse combinée

Lorsqu'elle est appliquée à des programmes de grande taille, l'analyse statique est une analyse inutilement coûteuse, car elle analyse tout le code source alors qu'une grande partie de celui-ci ne sera jamais exécutée. Au contraire, l'analyse dynamique est beaucoup plus précise, elle ne s'applique que sur les éléments concernés par une exécution donnée. Mais cette analyse prend beaucoup de temps à s'exécuter. Pour contourner ce problème, nous utilisons une analyse combinée (*"blended analysis"*). Cette analyse effectue une analyse statique sur une structure d'appels obtenue à partir d'une analyse dynamique. Ainsi, cela assure que seulement les méthodes exécutées seront analysées.

Le but de cette analyse est d'atteindre un niveau de précision comparable à celui d'une analyse dynamique complète, mais à un coût plus accessible.

3.3 Analyse d'échappement

Les objets temporaires coûtent cher, d'une part à cause de l'allocation mémoire et du ramasse-miettes (ou *"garbage collector"*), mais aussi pour la quantité de travail réalisée pour les initialiser. Par conséquent, nous voulons connaître les régions qui sont responsables d'une utilisation excessive d'objets temporaires. L'analyse d'échappement (*"escape analysis"*) est une analyse statique qui permet de calculer des limites sur la durée de vie des objets au cours de l'exécution. Nous réutilisons une implémentation d'une analyse d'échappement existante basée sur l'algorithme de Choi et al. [7]. Cette analyse englobe l'analyse de références qui détermine, pour toute référence dans le programme, l'ensemble des objets auxquels elle peut pointer. Elle associe aussi à chaque objet un état d'échappement qui permet de déterminer la région du programme dans laquelle cet objet est confiné au cours de l'exécution. Comme la majorité des analyses statiques, l'analyse d'échappement requiert une structure d'appels qui permet de définir la région du programme à explorer. De tels graphes sont normalement générés par une analyse statique, mais peuvent être remplacés par une structure dynamique telle que le CCT ou le graphe d'appels dynamique pour obtenir une analyse combinée. À la fin de l'analyse, un graphe de connexion est associé à chaque nœud du CCT ou du graphe d'appels, qui montre les connexions possibles entre les différents objets du programme. De plus dans le graphe chaque objet donne son état d'échappement.

Un objet statique est une abstraction représentant un certain nombre d'objets créés durant l'exécution, contrairement à un objet dynamique qui est un objet réellement créé durant une exécution. L'analyse d'échappement utilise les sites de création comme représentation des objets statiques, un choix très commun pour les analyses statiques semblables. Chaque objet statique reçoit un état d'échappement distinct calculé par l'analyse. L'état d'échappement d'un objet statique indique si l'objet est seulement accessible durant l'exécution de m (il est donc considéré comme capturé), s'il échappe à

m à travers une référence globale (globalement échappé) ou à travers des paramètres ou valeurs de retour (échappé par argument). Durant une analyse, un objet statique peut avoir différents états d'échappement, on se réfère donc à l'état final. Les objets capturés, à l'état final, sont particulièrement intéressants, car ils peuvent être considérés comme temporaires.

La figure 3.2 montre un exemple d'objets capturés. Les classes *Y* et *Z* étendent la classe *X*. La méthode *f* créée soit un objet de type *Y* soit un objet de type *Z* (en fonction de la condition *cond*). L'objet de type *Y* est créé lors de l'appel à la méthode *identity*. Il va donc s'échapper à travers l'argument, puis dans la méthode *identity* il va de nouveau s'échapper par la valeur de retour. Pour finir, la variable *inst* le récupère puis il n'est plus accessible à la fin de l'exécution de la méthode *f*, donc son état final est capturé. L'objet de type *Z* est créé lors de l'appel à la méthode *escape*. Il va donc s'échapper à travers l'argument. Puis, dans la méthode *escape* il s'échappe globalement grâce à la variable *G*, donc son état final est globalement échappé.

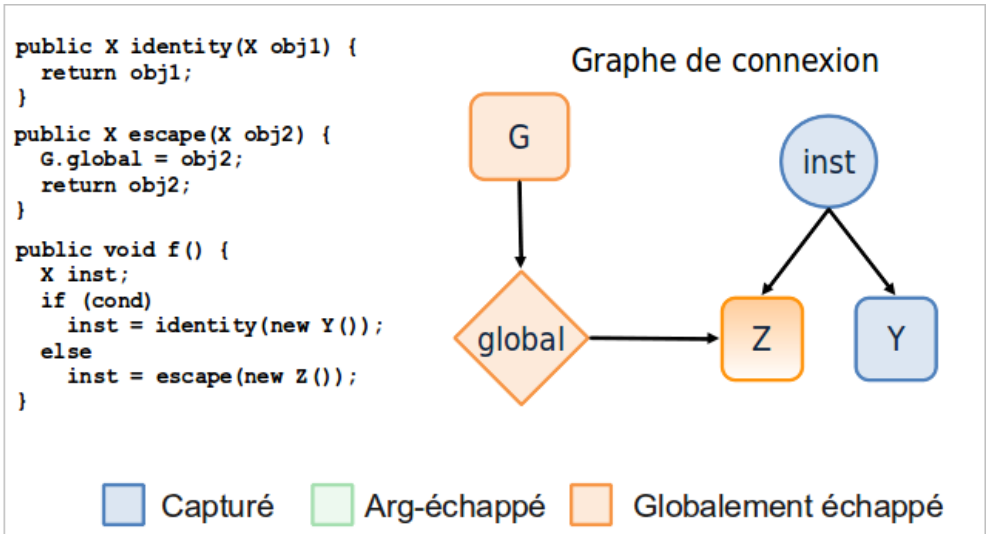


Figure 3.2 – Exemple d'objets capturés

Un objet créé ne peut être capturé que dans une région, c'est à dire il ne pourra être capturé que par une méthode au dessus de lui dans le sous-arbre des appels le contenant. Nous appelons ceci la capture par région. Notons que l'analyse est conservatrice. Des

objets peuvent être marqués comme échappés alors qu'ils sont capturés (faux négatifs), mais les objets ne peuvent pas être marqués comme capturés s'ils sont en réalité échappés (pas de faux positifs).

3.4 Analyse d'échappement combinée

L'analyse d'échappement combinée consiste en trois phases. Dans la première, la trace d'exécution est collectée à l'aide d'un outil de profilage existant. Le profileur collecte l'arbre d'appels et les informations sur les allocations des objets pour une exécution particulière du programme. Par la suite, les informations dans l'arbre d'appels sont agrégées pour construire un CCT avec en plus les informations sur les instances dynamiques allouées à l'intérieur de chaque contexte. Le choix du CCT est motivé par le fait que des travaux précédents par Fisher II et al. [13] ont montré que la précision additionnelle des CCTs améliore de façon significative le temps d'exécution de l'analyse, et permet même parfois d'obtenir une meilleure précision en comparaison avec des graphes d'appels dynamiques (sans contexte). La seconde phase consiste en l'analyse statique d'échappement où le graphe de connexion est récupéré pour chaque noeud du CCT. De plus, le statut d'échappement est inclus pour chaque objet du graphe de connexion. La troisième phase consiste à visualiser les informations de l'instance dynamique du CCT sur les objets statiques utilisés dans l'analyse d'échappement. Pour chaque contexte d'appels du CCT, le profileur identifie le nombre des instances pour chaque type d'allocations.

La figure 3.3 est un schéma récapitulatif du chemin permettant d'avoir des CCT acycliques réduits. Pour commencer, la trace dynamique de l'exécution est collectée par l'outil Jinsight [9], puis ce même outil construit, à partir de la trace d'exécution, un CCT. Elude [10] effectue l'analyse d'échappement sur le CCT pour associer un graphe de connexion à chaque nœud du CCT. Enfin, l'outil Churni [13] fusionne le fichier en sortie d'Elude avec les informations d'allocations récupérées par Jinsight et produit un CCT avec des graphes de connexions réduits.

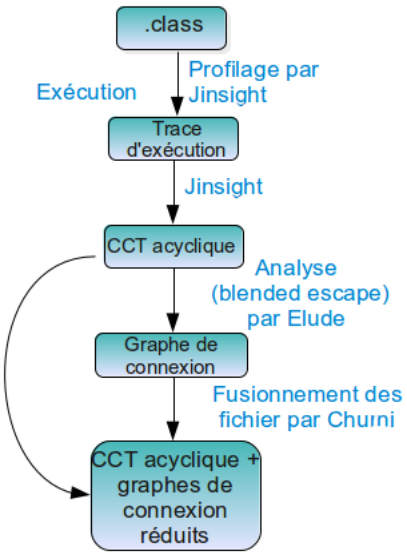


Figure 3.3 – Schéma récapitulatif du chemin permettant d'avoir des CCT acycliques réduits

CHAPITRE 4

VASCO

Les outils de visualisation existants ne permettent pas de représenter l'information nécessaire à l'identification des objets temporaires, ou ne sont pas en mesure de représenter la quantité d'information nécessaire à l'analyse des programmes de grande taille. Nous avons donc développé un outil de visualisation spécifiquement pour cette tâche. Cet outil permet de visualiser l'information en provenance de traces d'exécution de façon intuitive et efficace. Dans ce chapitre, nous discutons premièrement des critères qui ont guidé nos choix de conception de l'outil. Nous présentons la métaphore visuelle utilisée, le mappage des données d'exécution à cette métaphore, ainsi que les détails du calcul des données visualisées. Finalement, nous décrivons comment notre outil supporte le processus typique suivi par un utilisateur pour accomplir sa tâche.

4.1 Objectifs

La conception d'un outil de visualisation interactive doit être guidée par certains critères afin de maximiser l'efficacité du processus de visualisation. Nous avons identifié trois critères importants pour notre application : la possibilité de représenter une grande quantité de données, la réduction de l'effort cognitif requis par un utilisateur, ainsi que la fluidité des transitions et des interactions.

Quantité de données représentées. La complexité des applications qui ont typiquement une utilisation excessive d'objets temporaires requiert une attention particulière au niveau de la visualisation. Par exemple, il est très fréquent d'observer des centaines de milliers d'appels pour n'effectuer qu'une seule opération à l'intérieur d'une exécution. Il est donc nécessaire de choisir une métaphore qui utilise efficacement l'espace disponible, mais surtout qui permet de présenter l'information d'une façon claire et compréhensible par l'utilisateur. Il est également souvent souhaitable de réduire la quantité de données à présenter de façon à privilégier la tâche à accomplir, tout en tentant de minimiser l'im-

pact sur la précision.

Effort cognitif. Il est important de s'assurer qu'un utilisateur n'ait pas un grand effort cognitif à fournir pour comprendre la visualisation. Pour ce faire, il doit pouvoir comprendre rapidement la métaphore utilisée. Par exemple, il est judicieux d'utiliser des formes familières dans la métaphore utilisée, ainsi que des attributs visuels représentant efficacement l'information. Il faut donc déterminer les attributs visuels qui sont les plus facilement compréhensibles pour les utilisateurs, et ceux qui facilitent la tâche à accomplir. Par exemple, la taille, la couleur ou la forme sont des attributs visuels qui permettent de comparer facilement différentes entités représentées. Nos décisions quant à la représentation des entités visuelles sont basées sur les travaux de Healey et Enns [14] sur le *processus préattentif*, qui décrit la façon dont les humains perçoivent et catégorisent des images en régions et propriétés. De plus, il est important de ne pas surcharger les entités avec beaucoup d'attributs visuels, afin que l'utilisateur puisse rapidement identifier l'information recherchée. Par exemple, la figure 4.1 démontre qu'il est facile d'identifier un disque rouge dans un environnement comprenant des disques bleus (figure 4.1(a,b)). De même, il est facile de retrouver un carré rouge dans un environnement comprenant des disques rouges (figure 4.1(c,d)). Par contre, il est plus difficile d'identifier un disque rouge dans un environnement comprenant à la fois des disques bleus et des carrés rouges (figure 4.1(e,f)).

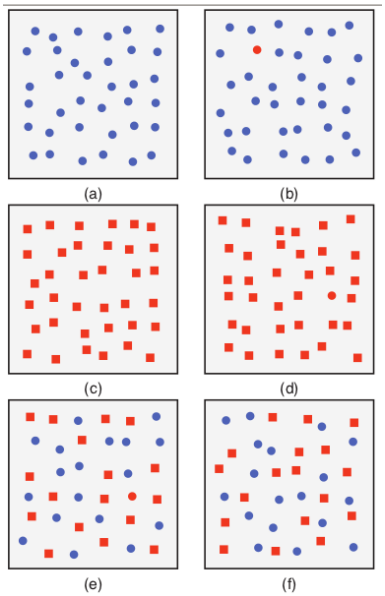


Figure 4.1 – Détection de cibles : (a) cible disque rouge absente ;(b) cible présente ; (c) cible disque rouge absente ; (d) cible présente ; (e) cible disque rouge présente ; (f) cible absente, tiré de [14]

La figure 4.2 est la liste des attributs importants entrant en compte dans le *"preattentive processus"* que les auteurs ont mis en évidence.

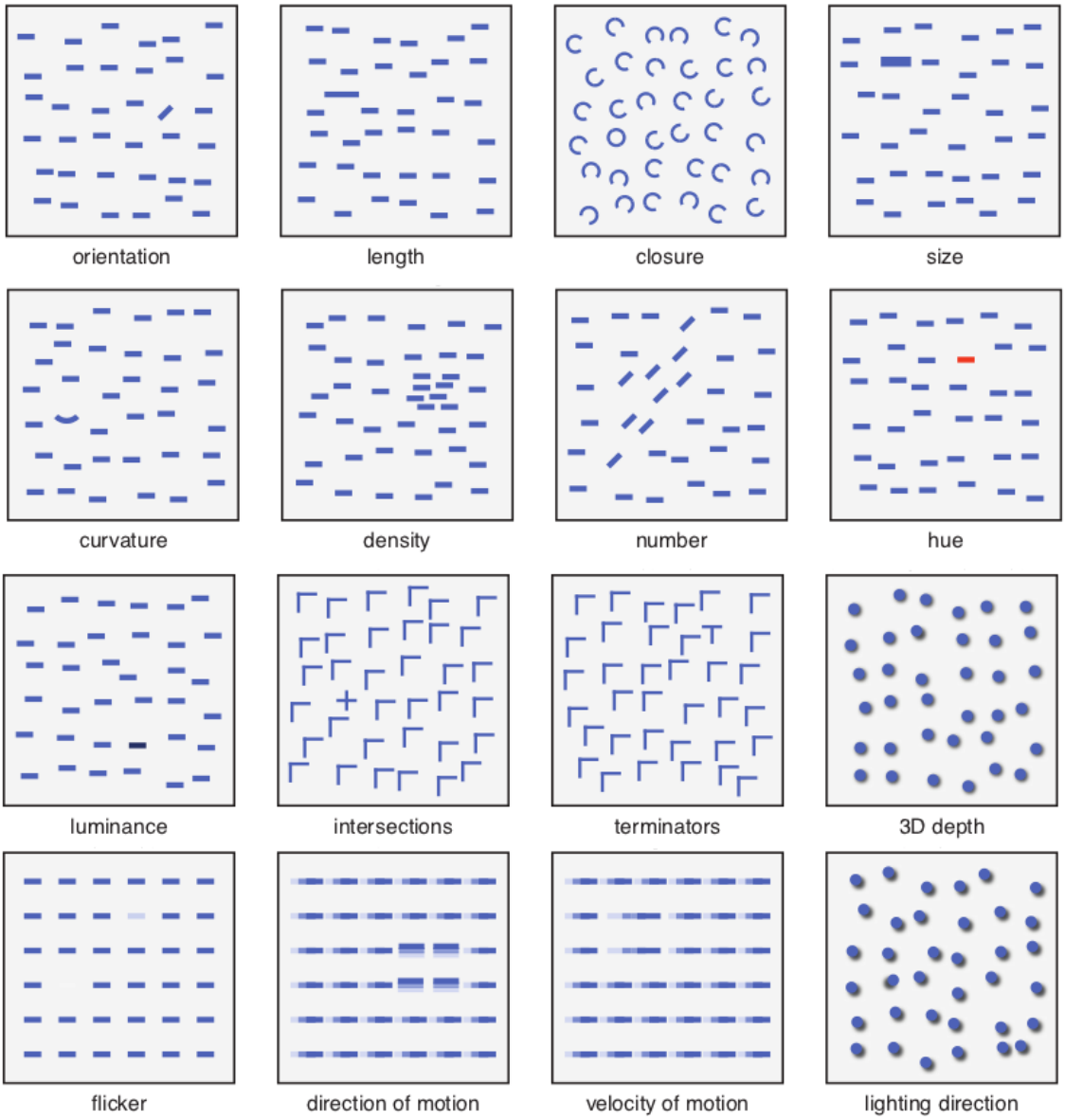


Figure 4.2 – Liste de Healey et al. des attributs visuels étant “preattentive” tirée de [14]

Fluidité des transitions et interactions. Les changements qui affectent la vue courante et les interactions avec l’outil ne doivent pas perturber la concentration de l’utilisateur. Il est donc nécessaire de minimiser leur impact. Par exemple, l’information consultée fréquemment devrait apparaître le plus près possible de la vue principale sans toutefois l’encombrer. L’information plus rarement requise peut être déplacée plus loin de la vue principale. Organiser l’information affichée de façon à la rendre facilement accessible

permet de limiter ou même éliminer la nécessité pour un utilisateur de détourner son attention durant le processus de visualisation. Afin de déterminer où et comment afficher l'information pour une tâche de visualisation donnée, il faut d'abord déterminer la pertinence de cette information pour la réalisation de la tâche. À l'aide de cette information, il est possible de choisir une représentation efficace de l'information requise.

De plus, un outil de visualisation doit généralement s'assurer qu'un utilisateur peut facilement comprendre le contexte associé à la vue courante durant ses interactions avec l'outil. Il est donc important, par exemple, d'utiliser des indices visuels à la suite d'une modification de la vue. Ces indications visuelles agissent comme des points de repère durant les transitions entre les différents états de la visualisation. Cette représentation explicite du contexte libère donc l'utilisateur de la nécessité de se souvenir des interactions précédentes avec l'outil.

4.2 Métaphore visuelle

La métaphore du Sunburst a été proposée par Stasko et al. [29] pour permettre de visualiser efficacement une arborescence. Comme les CCTs acycliques générés par nos outils d'analyse sont constitués d'un ou plusieurs arbres, ils peuvent facilement être visualisés avec cette métaphore. Afin de garantir une racine unique, un nœud racine synthétique est ajouté au CCT. Ce nœud a pour enfants chaque racine du CCT d'origine.

Le Sunburst représente un arbre sous forme de cercles concentriques placés autour de la racine au centre. Chacun des cercles représente un niveau de l'arbre. Chaque nœud de l'arbre est représenté visuellement par un *arc* (une portion d'un des cercles). Chaque arc est défini par son niveau à partir du centre ainsi que la portion du cercle qu'il occupe. L'angle occupé par un arc dépend généralement des propriétés des données visualisées, et est typiquement proportionnel à l'importance d'un nœud dans le contexte de son parent. L'angle disponible pour les enfants d'un nœud est restreint par l'angle de son arc : les enfants sont disposés à l'intérieur du même secteur que le parent, et occupent collectivement un angle qui n'excède pas celui de leur parent.

Comme vu précédemment, la figure 2.14 montre un exemple de représentation d'un

arbre par la métaphore du Sunburst. Dans cet exemple, le nœud A est la racine de l'arbre et est donc représenté au centre. Ses enfants B et C sont disposés autour du cercle correspondant à A. D et E sont enfants du nœud B, ils sont donc placés en dessous de celui-ci. La même importance est donnée ici à chacun des enfants, ils ont donc tous le même angle.

Le Sunburst permet une utilisation efficace de l'espace sans toute fois augmenter l'effort cognitif nécessaire à la compréhension. Grâce à sa représentation implicite des liens entre les nœuds, cette métaphore évite le sacrifice inutile de l'espace et permet d'éviter l'encombrement de la visualisation. Cette représentation offre aussi une métaphore très intuitive pour des régions d'une exécution (c'est à dire, un sous-arbre dans le CCT). Contrairement à d'autres métaphores telles que le TreeMap, qui utilise des boîtes imbriquées pour représenter les relations entre les nœuds d'un arbre, le Sunburst permet plus facilement de représenter les séquences d'appels d'une façon plus intuitive. Puisqu'une des actions les plus fréquentes pour l'identification des objets temporaires est l'inspection d'une longue séquence d'appels, par exemple de l'allocation d'un objet à sa capture, cette propriété est particulièrement importante. Le positionnement intuitif des arcs par le Sunburst permet donc de réduire l'effort cognitif nécessaire à la tâche d'identification des objets temporaires.

4.3 Métriques

En plus de représenter les relations hiérarchiques entre les nœuds, la métaphore du Sunburst permet de représenter des propriétés pour chacun des nœuds. Pour éviter de surcharger la visualisation, nous utilisons à cette fin seulement deux attributs visuels : la couleur et l'angle de chaque arc. Ces deux attributs visuels sont les plus faciles à distinguer par un utilisateur. Chaque attribut visuel représente la valeur d'une *métrique* associée à un nœud. Une métrique est une mesure issue des propriétés techniques ou fonctionnelles du logiciel. Durant le processus de visualisation, l'utilisateur peut associer différentes métriques à chaque attribut visuel de façon interactive.

Le choix des métriques est important. Celles-ci doivent permettre d'accomplir la tâche demandée facilement. Elles doivent donc à la fois représenter les informations pertinentes à la tâche et être rapidement comprises par l'utilisateur.

Nous avons défini quatre métriques pour l'identification des objets temporaires :

Nombre de types créés le nombre de types distincts de tous les objets alloués par un ensemble d'invocations. Cette métrique vise à mesurer un aspect de la complexité des structures de données créées par un nœud.

Nombre d'objets créés le nombre total d'objets alloués par un ensemble d'invocations. Cette métrique vise à mesurer la contribution d'un nœud à la création d'objets dans le programme.

Nombre d'objets capturés le nombre total d'objets capturés par ensemble d'invocations. Cette métrique vise à quantifier l'importance d'un nœud en fonction du nombre d'objets temporaires qu'il permet d'expliquer.

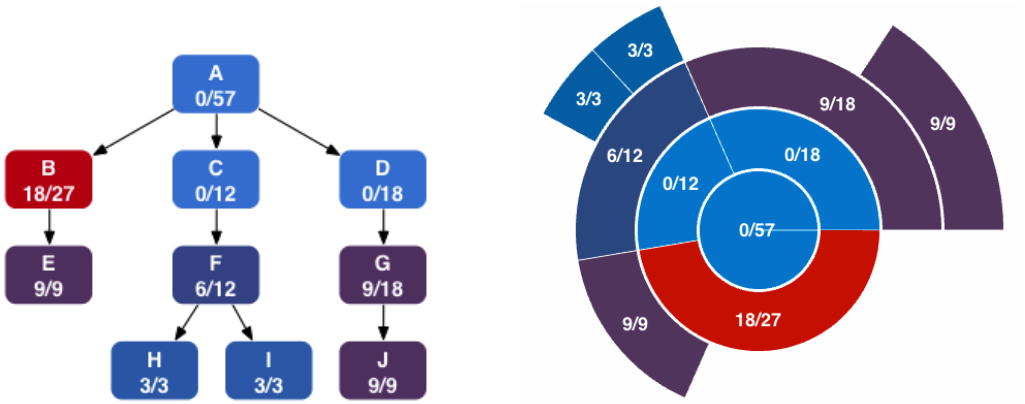
Churn le nombre d'objets créés par un ensemble d'invocations qui sont capturés par la suite. Cette métrique vise à identifier les sources les plus probables d'objets temporaires.

À l'exception de la métrique *"Churn"*, chacune de ces métriques peut être calculée sur l'ensemble des invocations correspondant à un seul nœud du CCT ou la région qu'il définit (c'est à dire, le sous-arbre ayant ce nœud comme racine). Pour chaque nœud, il est donc possible de calculer deux valeurs pour chaque métrique : une valeur individuelle et une cumulative. Chaque valeur est associée à un élément visuel différent des arcs.

La valeur individuelle est représentée visuellement par la couleur de l'arc. La gamme complète des valeurs associées à la couleur est représentée visuellement par un dégradé. Par défaut, ce gradient va du bleu, qui représente la plus faible valeur de la métrique, au rouge, pour la valeur la plus élevée pour tous les nœuds du CCT.

La valeur cumulative est représentée par l'angle occupé par l'arc. La valeur cumulative d'une métrique pour un nœud donné est définie comme la somme de la valeur individuelle de ce nœud et des valeurs cumulatives de tous ses enfants. La valeur maximale est donc celle de la racine du CCT. Le choix de représenter cette métrique par

l'angle d'un arc est donc intuitif et naturel, puisqu'il garantit que la taille d'un nœud ne peut être inférieure à la somme des tailles de ses enfants. Il est important de noter que dans les cas où la valeur individuelle d'une métrique pour un nœud est différente de zéro, le nombre de degrés disponibles pour les enfants de ce nœud sera inférieur au nombre de degrés occupés par le parent.



(a) CCT avec valeurs de métriques (b) Représentation du CCT en Sunburst

Figure 4.3 – Exemple de représentation d'un CCT (a) et des métriques par une métaphore Sunburst (b). Les métriques sont indiquées pour chaque nœud dans le format `individuel/cumulatif`.

Par exemple, la figure 4.3 montre la visualisation d'un CCT à l'aide du Sunburst. Chaque nœud possède deux valeurs de métrique : une valeur individuelle et une valeur cumulative. Les valeurs affichées ici sont à titre illustratif seulement, en pratique, notre outil ne représente pas ces informations sur les arcs.

Dans la figure, la racine (nœud A) a trois enfants. Les 360 degrés disponibles pour la racine sont donc divisés proportionnellement entre eux. La région sous le nœud B possède une valeur de 27 alors que son parent, A, possède une valeur totale de 57. B occupe donc un angle correspondant à 27/57 de l'espace disponible, soit 171 degrés. Similairement, les nœuds C et D occupent 76 (12/57 * 360) et 113 (18/57 * 360) degrés, respectivement. Au prochain niveau, le nœud E est une feuille. Sa valeur individuelle est donc égale à sa valeur cumulative, soit une valeur de 9. Ce nœud occupe donc 9/27 des 171 degrés alloués à son parent. Les nœuds F et G occupent chacun le même nombre de

degrés que leurs parents respectifs, puisque les nœuds C et D ont des valeurs individuelles de 0. Finalement, les enfants de F et G occupent la moitié de l'angle alloué à leurs parents, soit (3+3)/12 pour F et 9/18 pour G.

Dans le cas de la couleur, le bleu représente une valeur individuelle de 0 alors que le rouge représente la valeur maximale de 18. Les nœuds A, C et D ont des valeurs de 0, et sont donc représentés en bleu. B possède la valeur maximale, et est donc représenté par un rouge clair. Les nœuds restants possèdent des valeurs intermédiaires, et sont donc représentés à l'aide d'un mélange de couleurs intermédiaires entre le bleu et le rouge. Les nœuds H, I et F possèdent des valeurs près du minimum, et sont donc représentés par des teintes de bleu plus foncées. Les nœuds E, G, et J sont représentés en violet, ce qui correspond à une quantité égale de bleu et de rouge.

Dans certains cas, les nœuds représentés à la périphérie du Sunburst occupent un angle trop petit pour que l'utilisateur puisse les distinguer correctement (figure 4.4). Nous définissons donc un seuil correspondant au nombre de degrés nécessaires pour qu'un arc soit représenté. Notre visualisation rassemble tous les enfants d'un même parent pour lesquels l'angle occupé serait en dessous de ce seuil (par défaut, cette valeur est fixée à 3 degrés). Ces arcs sont alors représentés collectivement par un seul arc avec la couleur grise, car c'est une couleur qui n'attire pas l'attention (figure 4.5). Les enfants des nœuds inclus dans le groupe, s'ils existent, ne sont pas représentés dans la visualisation. Ce choix est motivé par le fait que les angles sous le seuil correspondent à des nœuds qui n'ont qu'une importance marginale dans l'analyse. Ne pas les représenter ne cause alors pas de perte importante de précision.

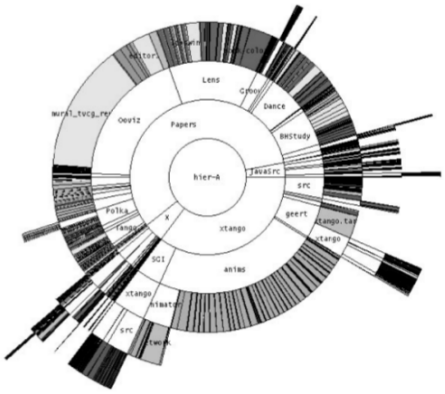


Figure 4.4 – Exemple d'arcs avec des petits angles à la périphérie du Sunburst tiré de [30]

.

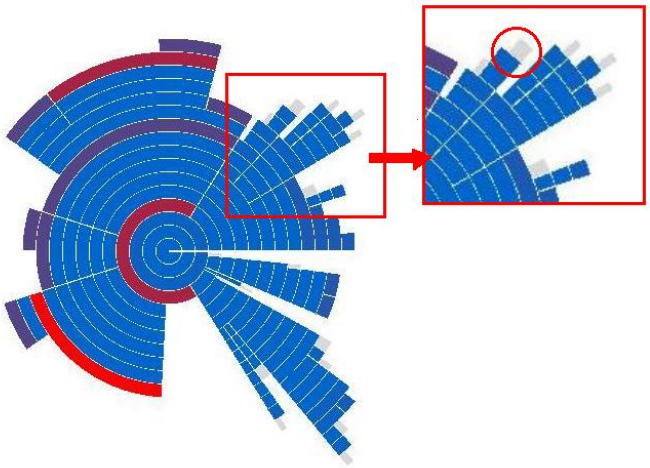


Figure 4.5 – Regroupement des méthodes

4.4 Interactions

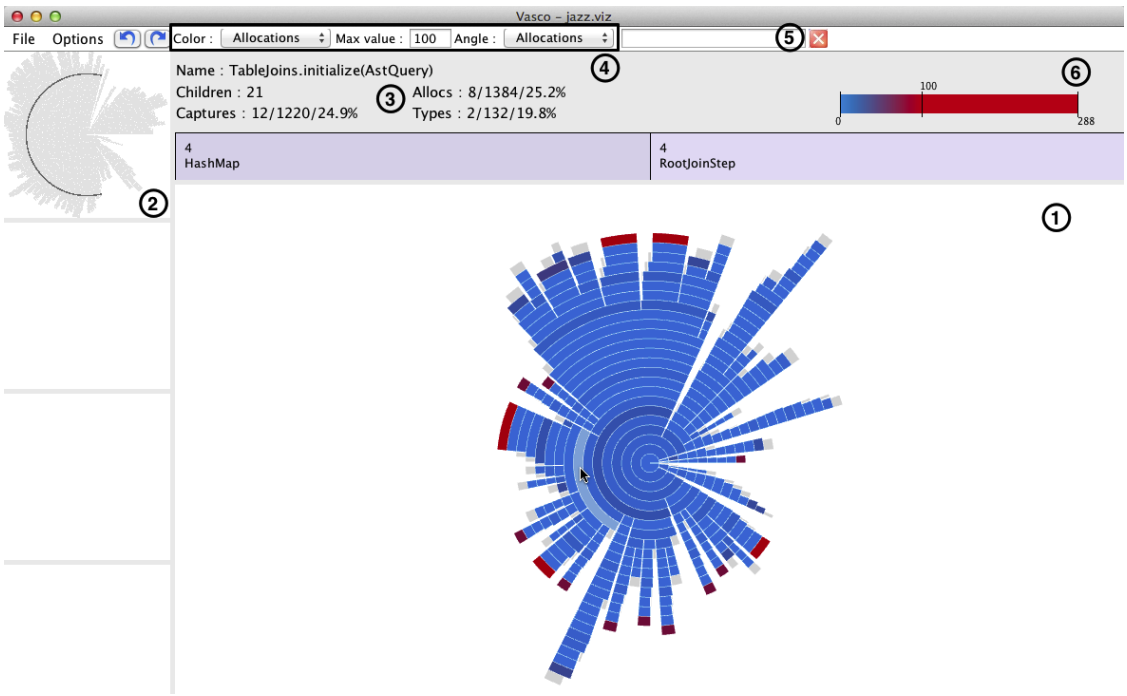


Figure 4.6 – Interface de l'outil de visualisation

Le scénario typique pour notre approche de visualisation est un processus itératif qui débute par la recherche de la source principale d'objets temporaires à l'aide de la visualisation interactive. Cette exploration nécessite des informations précises sur les régions ou les méthodes représentées. Une fois la région identifiée, l'utilisateur peut la retirer de la vue et recommencer le processus jusqu'à ce que toutes les sources d'objets temporaires aient été identifiées. Une séance de visualisation alternera donc typiquement entre une phase de recherche, dans laquelle l'utilisateur tente d'obtenir une meilleure compréhension du comportement de l'application étudiée, et une phase de réduction permettant de retirer certains éléments de la vue. Le reste de cette section présente les différentes interactions supportées par Vasco de façon à simplifier le processus de visualisation.

4.4.1 Sélection des métriques

L'utilisateur doit rapidement être dirigé vers des zones importantes de la vue. Les métriques présentent différents aspects des données visualisées. Le changement des métriques à afficher constitue donc une interaction utilisée fréquemment. L'outil permet à l'utilisateur de sélectionner rapidement les métriques qui correspondent aux caractéristiques recherchées à l'aide d'un menu déroulant se trouvant dans la barre de menu principale (figure 4.6 zone ④). La figure 4.7 montre un exemple de cette interaction. Le menu déroulant présente un choix des quatre métriques supportées par Vasco.

Les métriques sont associées à la couleur ainsi qu'à la taille des arcs et peuvent être sélectionnées indépendamment. Dans le cas de la couleur, la valeur correspondant à la couleur maximale peut être aussi redéfinie par l'utilisateur. Par défaut, cette valeur est la valeur maximale parmi tous les nœuds représentés dans la vue actuelle. Or, ce choix entraîne parfois une disposition de couleurs non optimale. Par exemple, une méthode *m* peut posséder une valeur de métrique beaucoup plus élevée que celles des autres méthodes. Les couleurs dans la visualisation dans ce cas ne seront donc pas dégradées : la méthode *m* sera de couleur maximale alors que la majorité des autres méthodes auront une couleur près de la couleur minimale. Vasco permet donc à l'utilisateur de redéfinir la valeur correspondant à la couleur maximale, soit en entrant une valeur directement dans une boîte de texte (figure 4.6 zone ④) ou par un glissement de la souris sur la zone affichant le dégradé de couleurs (figure 4.6 zone ⑥). Dans les deux cas, la sélection est reflétée en temps réel par l'outil, ce qui permet à l'utilisateur d'affiner le seuil rapidement par tâtonnement.

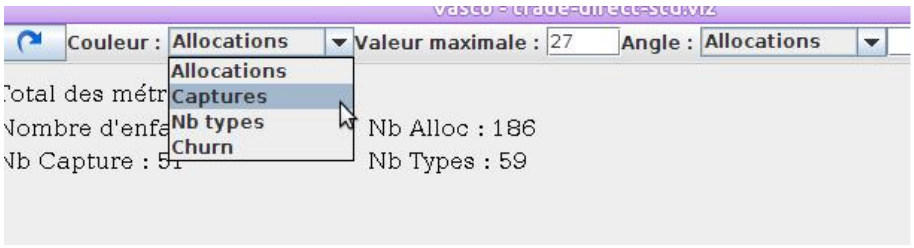


Figure 4.7 – Changement de métriques

4.4.2 Affichage de l'information contextuelle

Il est nécessaire d'avoir des informations détaillées sur les régions représentées afin de comprendre comment ces régions utilisent les objets temporaires et quelles méthodes en sont responsables. Par exemple, le nom d'une méthode peut être intéressant pour la retrouver dans le code source du programme ou le nombre et le type des allocations et captures d'une méthode pour comprendre son comportement. De plus, une des actions souvent appliquées est d'inspecter une longue séquence d'appels à partir de l'allocation d'un objet jusqu'à sa capture. Il est donc souhaitable que l'information soit complète et facilement accessible. Elle doit donc se trouver aussi proche que possible de la vue principale sans l'encombrer. Elle doit aussi pouvoir changer rapidement en fonction de la méthode inspectée.

Le panneau des informations est toujours visible juste au-dessus de la vue principale. Un simple mouvement de la souris au-dessus d'une méthode permet d'afficher les informations la concernant. Pour plus de clarté, cette méthode est mise en surbrillance. Lorsque le curseur de la souris ne se trouve sur aucune méthode, les informations agrégées s'affichent dans le panneau, c'est-à-dire les valeurs cumulatives des métriques de toutes les méthodes de la vue courante. De plus, l'outil permet la sélection multiple pour enquêter sur un groupe de méthodes sélectionnées. Quand plusieurs nœuds sont sélectionnés, le panneau d'information affiche les valeurs agrégées. Ce panel affiche plusieurs informations sur la méthode sélectionnée : le nom de la méthode sur laquelle pointe le curseur de la souris, son nombre d'enfants, et les valeurs des métriques. Trois valeurs s'affichent pour chaque métrique : la valeur pour la méthode (valeur utilisée pour la couleur), la valeur totale pour la région qui a pour racine cette méthode (utilisée pour les angles) et le pourcentage de cette valeur totale par rapport à la valeur maximale de l'ensemble des méthodes représentées. De plus, un graphe montre les différents types d'objets alloués ou capturés ainsi que leur proportion relative en nombre d'instances. L'information affichée par ce graphe varie en fonction de la métrique choisie de façon à montrer l'information pertinente dans le contexte d'analyse sélectionné par l'utilisateur.

L'information détaillée sur le comportement d'une méthode (comme le graphe de

connexion réduit généré par Churni) est aussi disponible par un menu contextuel qui apparaît sur demande (clic droit sur un nœud). Cette information détaillée est affichée dans une nouvelle fenêtre qui ne ferme pas la vue principale. La figure 4.6(3) montre un exemple du panneau des informations.

4.4.3 Recherche d'invocations de méthodes

Pour faciliter l'exploration de la vue, Vasco possède deux types de recherche d'invocations de méthodes. La première recherche toutes les invocations d'une méthode. Un clic droit sur une invocation permet le déroulement d'un menu, le choix de l'option "Autre(s) invocation(s) de cette méthode" (figure 4.8) lance la recherche de toutes les invocations de la méthode sélectionnée. Toutes les invocations de la méthode prennent alors la couleur bleu clair. Un mouvement de la souris sur l'une des invocations sélectionnées permet l'affichage des informations cumulatives de toutes les invocations de la méthode.

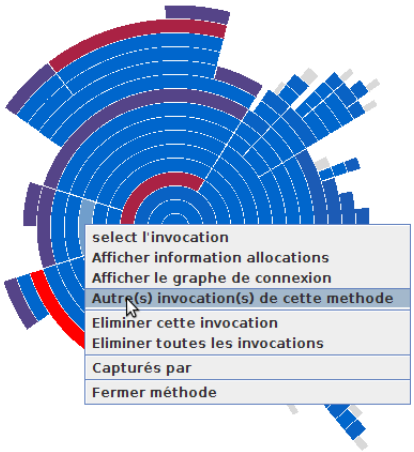


Figure 4.8 – Autres invocations de la méthode sélectionnée

De plus, nous permettons la recherche par nom de méthodes. Une barre de recherche se situe en haut à droite de l'outil (figure 4.6 zone ⑤), l'utilisateur peut entrer le nom d'une méthode (ou une partie du nom). La figure 4.9 montre un exemple de recherche. Toutes les invocations des méthodes contenant le nom tapé apparaissent en couleur bleu

clair. Pour annuler la sélection, il suffit de cliquer sur une zone vide de la vue principale ou sur le bouton situé à droite de la barre de recherche.

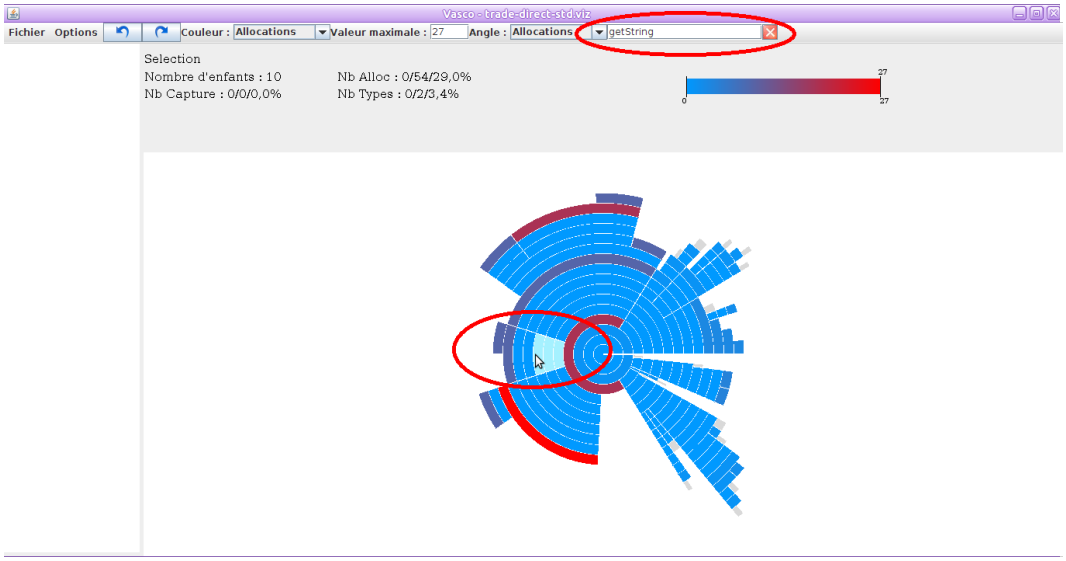


Figure 4.9 – Recherche

4.4.4 Filtrage des données

La vue initiale de l'outil montre le CCT en entier, donnant ainsi une vue d'ensemble. L'utilisateur peut naviguer et filtrer les données au travers de diverses interactions visant à réduire les données visualisées. Filtrer les données représentées permet de réduire l'effort cognitif requis de l'utilisateur.

Par exemple, l'outil permet de sélectionner une nouvelle racine. Vasco affiche alors une nouvelle vue avec seulement un sous-arbre du CCT. Cette fonctionnalité est la plus courante dans notre étude de la trace, il faut donc qu'elle soit rapide à effectuer. Un clic gauche sur la méthode choisie permet un changement de vue, cette méthode devient la racine et seulement son sous-arbre est affiché. Toutes les données sont alors recalculées pour refléter ce changement. L'exemple sur la figure 4.10 montre un changement de racine. Notons que ce changement peut faire apparaître des groupes de nœuds qui étaient précédemment cachés du fait de leur faible importance.

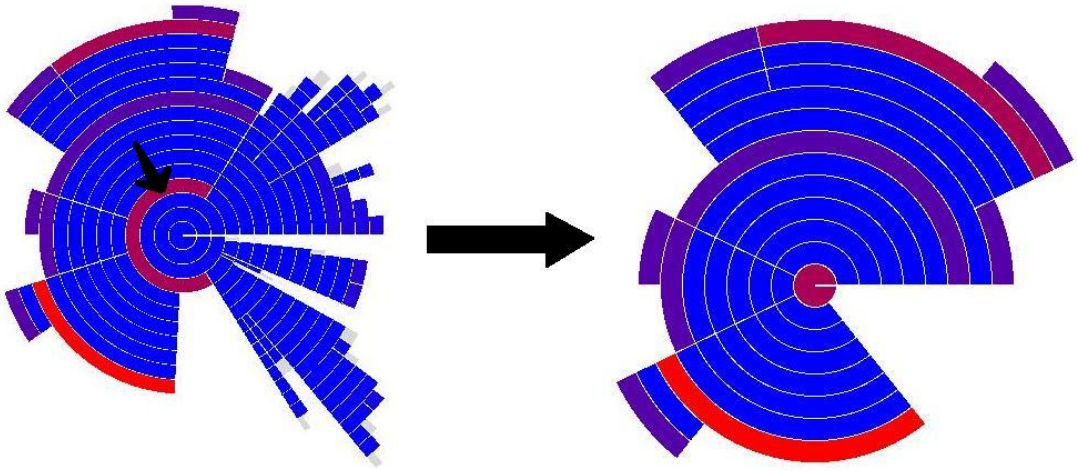


Figure 4.10 – Exemple de changement de vue : La méthode pointée par la flèche devient la racine du nouveau Sunburst

Il est fondamental que l'utilisateur ait un retour d'information sur l'interaction produite pour s'adapter rapidement à la nouvelle vue. Une représentation désaturée de la vue avant la sélection d'une nouvelle racine est ajoutée à un panneau se trouvant sur la gauche (figure 4.6 (zone ②) montre ce panneau). La méthode qui a été sélectionnée s'affiche en noir. Les images s'entassent les unes en dessous des autres. L'utilisateur peut ainsi retracer le chemin qui l'a mené à la vue courante. À tout moment, il peut cliquer sur une des images pour retourner à la vue représentée. La figure 4.6 (zone ③) montre un exemple du panneau contextuel. Un clic dans une image permet de revenir à la visualisation correspondante.

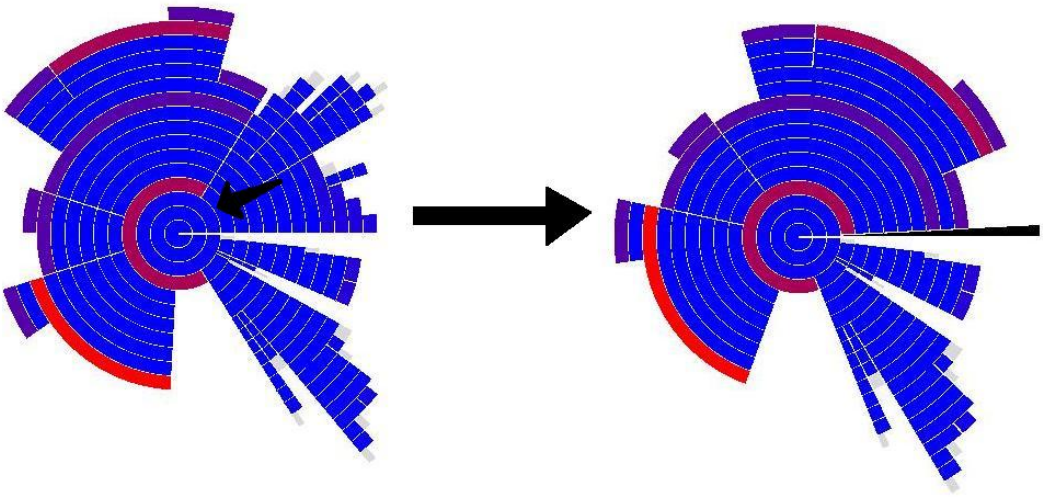


Figure 4.11 – Exemple de fermeture d'un sous-arbre : Le sous-arbre dont la racine est pointée par la flèche se ferme

L'outil supporte aussi l'opération opposée : cacher une seule région dans la vue courante. Pour cela, on choisit l'option "Fermer méthode" dans le menu apparaissant au clic droit sur une méthode *m*. La région sous la méthode *m* se cache. Cette région reste partiellement visible dans la vue courante (figure 4.12). Elle est représentée par un arc de couleur noir avec un petit arc et s'étend sur autant de niveaux que la région entière. Il est important de laisser un indice indiquant la présence de ce nœud pour deux raisons, cela permet aux utilisateurs d'inverser l'opération et donc d'ouvrir de nouveau cette région en cliquant sur la partie restée visible. Cet indice visuel aide les utilisateurs à s'adapter rapidement au résultat de la nouvelle vue. Comme pour l'opération précédente, toutes les données sont recalculées en ne prenant pas en compte la région fermée. L'exemple sur la figure 4.11 montre une fermeture d'un sous-arbre.

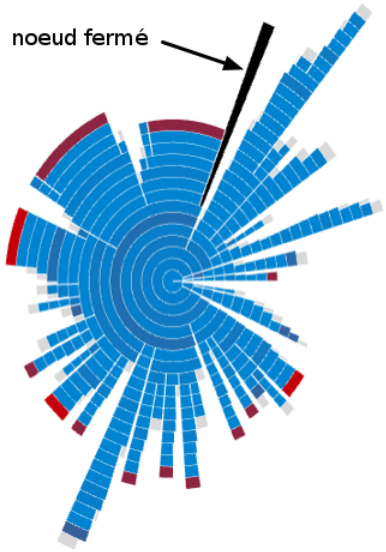


Figure 4.12 – Représentation d'une région fermée

4.4.5 Paramétrisation des données

Pour que l'outil soit facile à utiliser et que le choix initial des paramètres (exemple, le choix de couleurs) ne ralentisse pas l'utilisateur dans l'accomplissement de sa tâche, Vasco permet une certaine liberté à l'utilisateur dans la paramétrisation des données. Plusieurs options sont alors disponibles dans la barre de menu. Tout d'abord, l'utilisateur peut afficher les noms de méthodes et types d'objets en abrégé afin de les lire plus facilement, ce qui peut être difficile lorsque les noms complets sont très longs.

De plus, l'option "paramétrer les couleurs" permet à l'utilisateur de choisir les couleurs lui convenant. Une fenêtre de paramétrisation des couleurs (figure 4.13 à gauche) présente les deux couleurs de l'échelle nécessaires à la visualisation. Ces couleurs peuvent être changées au gré de l'utilisateur à l'aide d'un simple clic sur le rectangle coloré voulu, faisant apparaître une palette de choix de couleurs (figure 4.13 (à droite)).

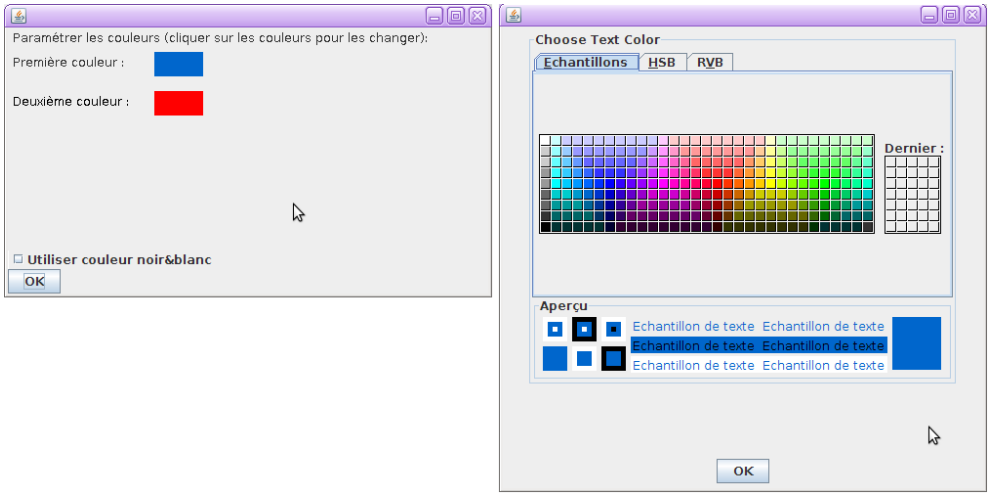


Figure 4.13 – Fenêtres de paramétrisation et du choix des couleurs

CHAPITRE 5

ÉTUDES DE CAS

Les chapitres précédents ont présenté notre approche de visualisation interactive. Afin d'évaluer l'efficacité de notre approche, nous l'appliquons à trois programmes réels de type *framework-intensive*. Dans ce chapitre, nous décrivons tout d'abord les applications étudiées, puis nous décrivons comment Vasco permet d'identifier rapidement des problèmes réels dans chacune de ces applications.

5.1 Cadre expérimental

Nous utilisons trois applications de type *framework-intensive* afin d'évaluer notre approche :

Trade Trade est une application de test pour la plate-forme WebSphere qui simule des transactions financières. Nous récupérons une trace qui a été produite à partir de la version 6.0.1 de Trade fonctionnant sur WebSphere 6.0.0.1 et DB2 8.2.0. Lors de l'exécution, l'application Trade était configurée pour utiliser une connexion directe à la base de données et les services web étaient activés.

Serveur Jazz Le serveur Jazz est une plate-forme de développement collaborative. Elle utilise une architecture client-serveur. Les clients sont intégrés à l'environnement de développement (IDEs) basé sur Eclipse. Le serveur est basé sur une WebSphere. Nous récupérons une trace faite à partir d'une seule requête pour récupérer les éléments de travail affectés à un seul développeur avec la version 1.0M5a du serveur de Jazz.

Serveur commercial de gestion de documents (CDMS) CDMS est une plate-forme commerciale qui assure la gestion et le flot de travail pour les documents. Nous avons tracé une partie de la suite de tests chargés de stocker dix documents dans la base de données. Comme ce logiciel est propriétaire, il n'est pas possible de divulguer plus d'informations à ce sujet.

Malgré le traçage d'une seule transaction pour chaque application, le tableau 5.I montre la complexité du travail produit par chacune d'entre elles. Les colonnes 2 et 3 montrent respectivement le nombre total de types distincts qui ont été alloués et le nombre total d'instances créées. Les quatre dernières colonnes montrent le nombre total de méthodes distinctes exécutées, le nombre total d'invocations de méthodes, le nombre de noeuds dans le CCT, et la profondeur maximale de la séquence d'appels durant l'exécution. Notons que le nombre de noeuds dans le CCT est plus faible dans les trois cas que le nombre des appels, allant de 19% des appels pour Jazz à seulement 1.6% pour CDMS. Cette réduction significative dans le nombre des noeuds joue un rôle clé pour mettre l'approche de visualisation à la bonne échelle. Pour chaque application, nous avons utilisé notre approche pour analyser son comportement concernant les objets temporaires. Nous décrivons les résultats de cette étude dans ce chapitre. Ces études de cas ont pour but de démontrer l'efficacité de notre approche à identifier les plus importantes sources d'*object churn*, plutôt qu'une analyse complète et approfondie du *churn* trouvé dans les applications que nous étudions.

Applications	**Types alloués**	**Instances allouées**	**Méthodes distinctes exécutées**	**Appels**	**Noeuds du CCT**	**Profo p**
TRADE	166	5,522	3,308	127,794	22,558	
JAZZ	181	9,470	2,547	170,311	32,478	
CDMS	254	62,066	3,144	1,495,192	24,285	

Tableau 5.I – Caractéristiques des applications *framework-intensive*

Dans nos études, le code source n'est pas disponible, il est donc impossible de calculer la taille des programmes en lignes de code (LOC). Toutes les classes des *frameworks* ne sont pas forcément utilisées dans l'exécution des programmes, on ne peut donc pas prendre en compte le nombre de classes. Nos statistiques dynamiques visent à donner une idée de la taille réelle du code exécuté.

5.2 Trade

À l'ouverture de l'outil, on associe la métrique *churn* à la couleur des arcs et le nombre d'allocations à leurs tailles. La figure 5.1 montre la vue avec cette association. Nous cherchons les arcs qui ont une couleur rouge, car ils représentent les méthodes allouant la plus grande proportion d'objets temporaires. Nous trouvons deux méthodes, que nous avons marquées ① et ② sur la figure. Avec un déplacement de la souris sur ①, on apprend qu'il s'agit de la méthode `DecimalFormat.parse` et qu'elle crée 216 objets sur 54 invocations différentes. Une inspection du code révèle que la plupart de ces objets sont utilisés pour la comptabilité. Leur optimisation serait difficile, car elle nécessiterait une modification importante de l'algorithme implémenté.

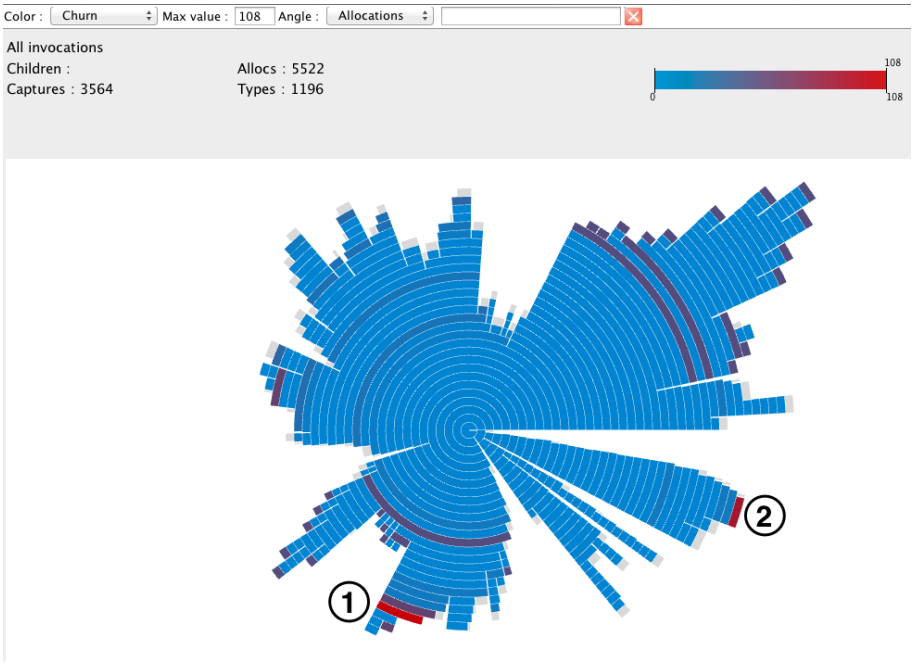


Figure 5.1 – Vue initiale de la trace d'exécution provenant de Trade

La méthode ② correspond au constructeur de la classe `Calendar`, qui crée 99 objets sur neuf invocations. Beaucoup de ces objets sont des tableaux qui enregistrent l'état du calendrier. La sélection de l'option "capturés par" pour ce nœud indique que ces objets sont capturés par la méthode `DateSerializer.getValueAsString` (figure 5.2). Dans des travaux précédents, cette méthode a été identifiée comme responsable

du *churn* dans Trade car elle recrée inutilement des objets calendrier pour chaque valeur devant être désérialisée. Une inspection manuelle du code a montré que les calendriers pourraient être mis en cache et réutilisés par cette méthode, évitant ainsi l'initialisation onéreuse de nombreux objets à chaque invocation de la méthode.

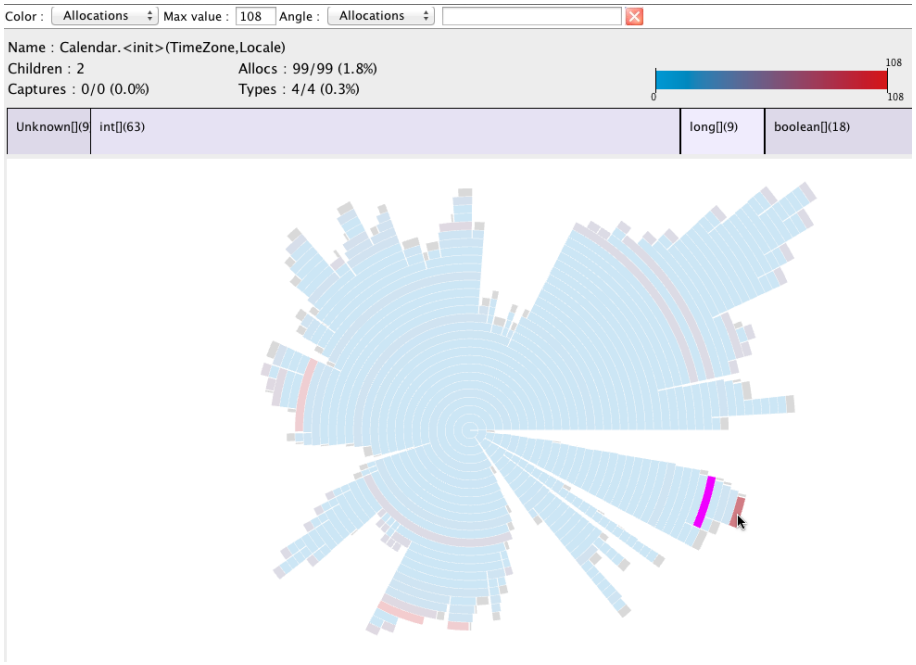


Figure 5.2 – Vue après sélection de l'option "capturés par"

Nous pouvons finalement totalement éliminer toutes les invocations de `DateSerializer.getValueAsString` et obtenir une nouvelle vue (figure 5.3) où toutes les métriques sont recalculées en ne prenant plus en compte `DateSerializer. getValueAsString`. Nous pouvons donc chercher de nouvelles régions contenant du *churn*.

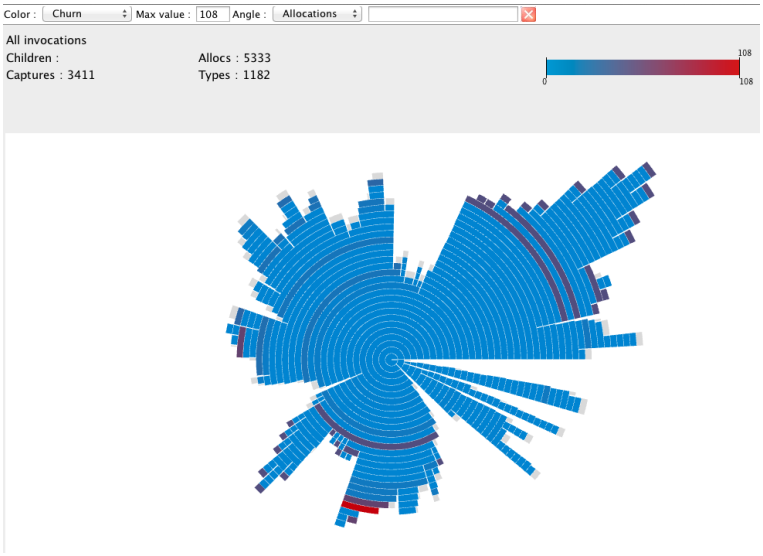


Figure 5.3 – Vue après élimination de la méthode `DateSerializer.getValue-AsString`

5.3 Serveur Jazz

La figure 5.4 montre la vue initiale pour la trace récupérée. Nous avons configuré la métrique des couleurs pour que les nœuds allouant au moins 100 objets capturés apparaissent en rouge (la métrique *churn* est associée à la couleur et la valeur maximale est fixée à 100).

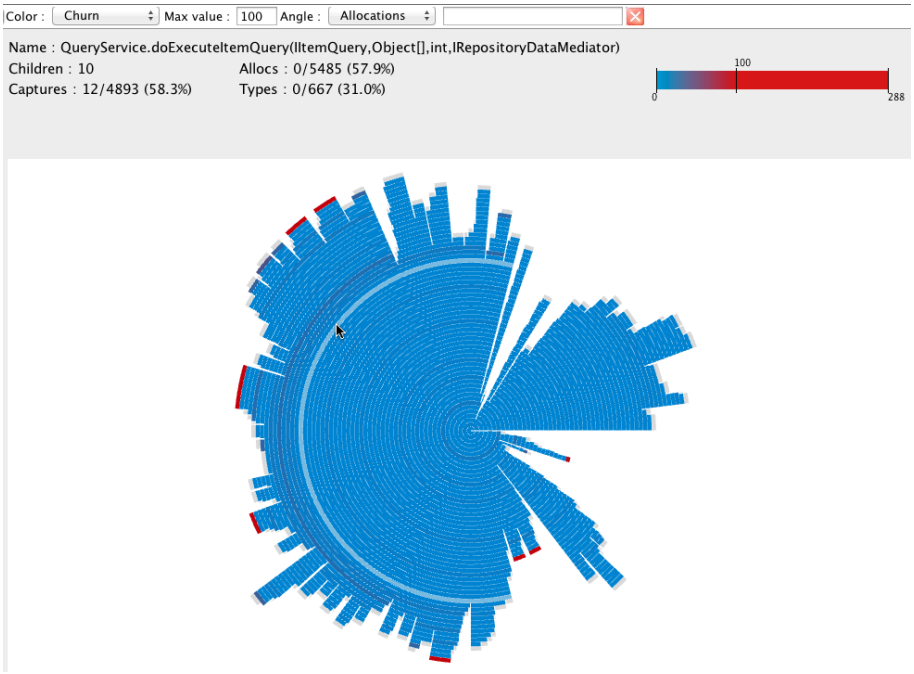


Figure 5.4 – Jazz : vue initiale pour la trace récupérée

Afin d'étudier la portion de l'exécution responsable de la majorité des allocations, nous sélectionnons une nouvelle racine qui correspond à la méthode sous le curseur de la souris dans la figure 5.4 (à l'aide d'un clic gauche sur la méthode). Grâce au panneau des informations, nous remarquons que la région est responsable de 57% des allocations et de 58% des captures du programme. La figure 5.5 montre le résultat de ce changement de racine. Beaucoup de nœuds apparaissent en rouge dans cette vue.

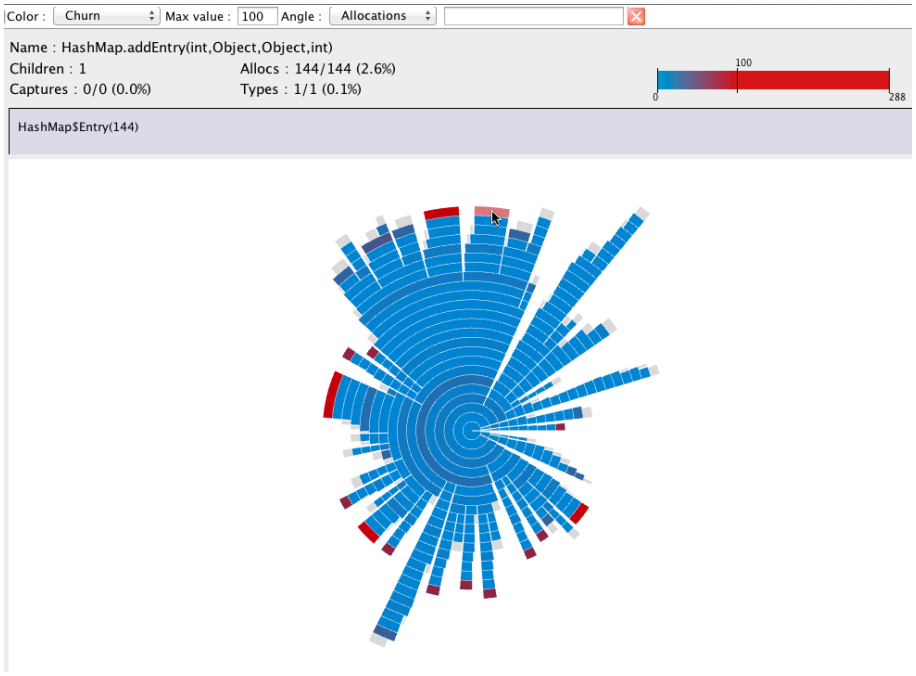


Figure 5.5 – Jazz : vue après changement de racine

Une inspection révèle qu'ils correspondent à différentes invocations de la méthode `HashMap.addEntry`. Tel qu'illustré par la figure 5.5, les invocations sélectionnées de cette méthode allouent 144 objets de type `HashMap$Entry`. Après sélection de l'option "capturés par", on remarque qu'ils sont tous capturés par la méthode `ComponentRegistry.getItemType` (Figure 5.6). En regardant l'unique méthode visible appelée par `getItemType` dans cette représentation, nous constatons qu'elle est responsable de la création de huit objets de type `HashSet`, tous sont également capturés par `getItemType`.

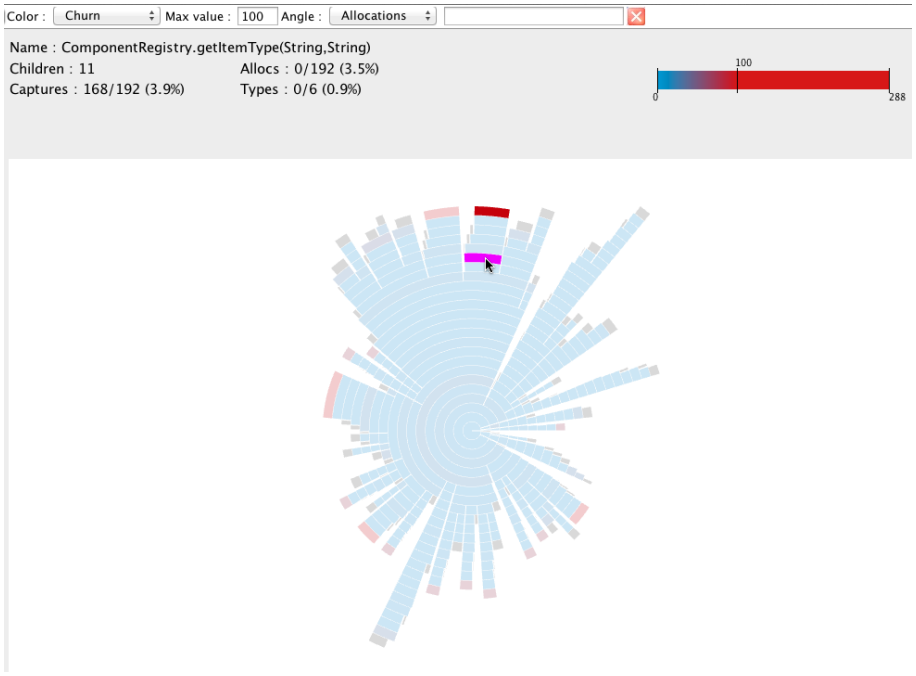


Figure 5.6 – Jazz : vue montrant la méthode qui capture les objets créés par `HashMap.addEntry`

Une inspection rapide de la séquence d'appels de `getItemType` à `HashMap.addEntry` montre que `getItemType` appelle la méthode `getAllModelURIs` dans la même classe, ce qui crée et propage un nouvel objet `HashSet` à chaque allocation. En Java, la classe `HashSet` est implémentée en utilisant une `HashMap`, ce qui explique le comportement observé. Dans cette trace, l'ensemble retourné par cette méthode contient toujours les mêmes éléments, et pourrait donc être facilement mis en cache. Pour évaluer l'impact que cette stratégie aurait sur le programme, nous sélectionnons toutes les invocations de la méthode `getAllModelURIs` et étudions les statistiques pour la sélection (figure 5.7). Cette opération révèle que cette méthode est responsable de la création de 3197 objets au total, tous sont temporaires. La mise en cache serait donc particulièrement utile dans ce cas.

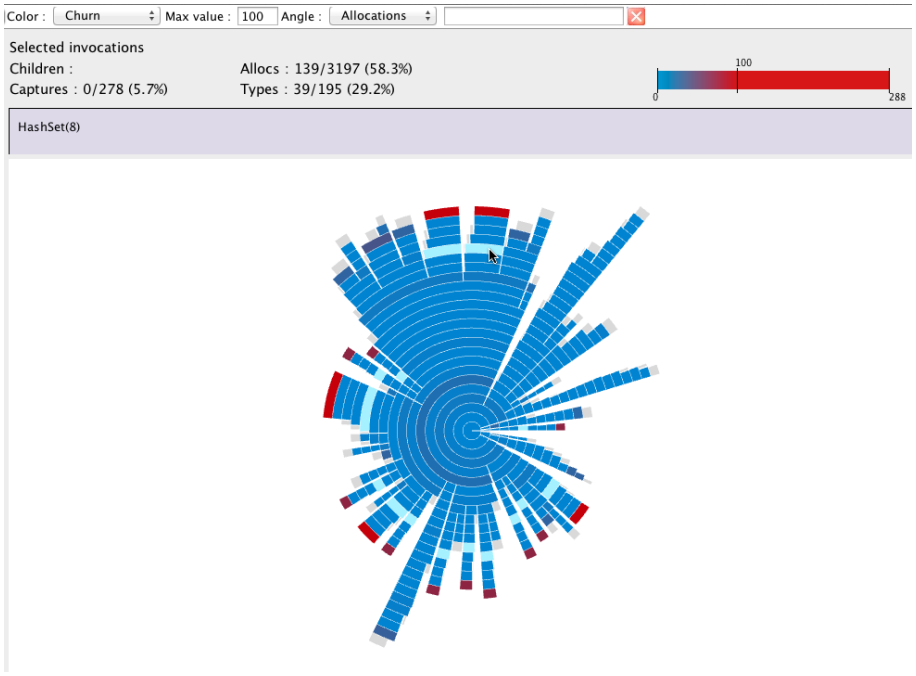


Figure 5.7 – Jazz : vue montrant les informations sur la sélection de toutes les invocations de la méthode `getAllModelURIs`

Aussi, notons que la méthode responsable du *churn* dans ce cas, `getAllModelURIs`, n'est ni une méthode allouant ni une méthode capturant. Cela illustre le besoin d'explorer la séquence d'appels afin de correctement diagnostiquer le problème.

5.4 CDMS

La figure 5.8 montre la vue initiale pour CDMS avec le *churn* visualisé sur la couleur des arcs et le nombre des allocations sur leurs tailles. Nous remarquons immédiatement plusieurs sources du *churn* qui se détachent du reste de la trace (les méthodes rouges dans la figure). Toutes ces sources sont liées aux chaînes de caractères. Pour mieux comprendre le comportement de cette application, nous changeons la métrique de couleurs en visualisant les captures (figure 5.9).

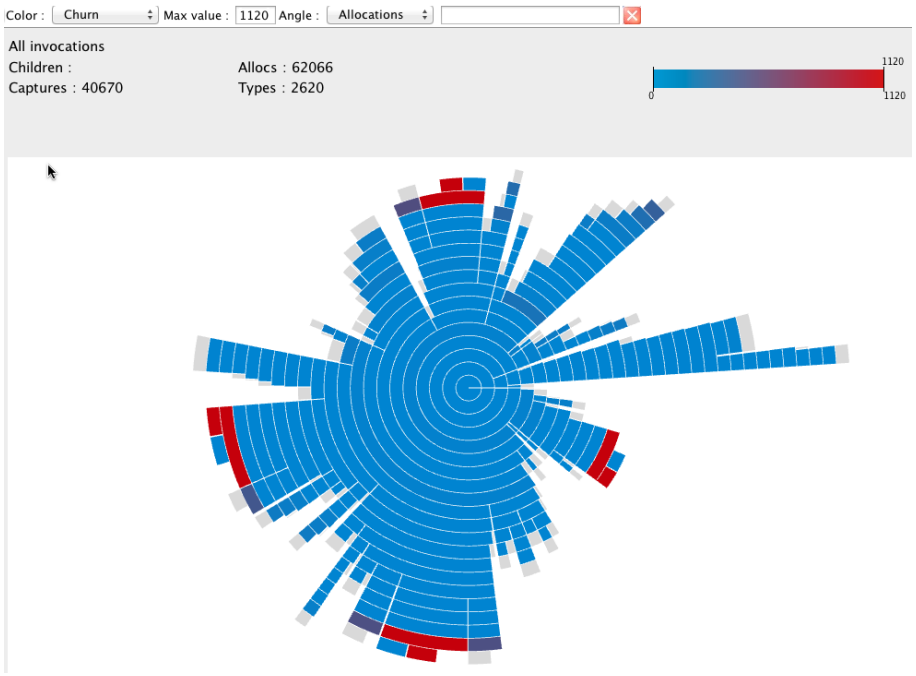


Figure 5.8 – Vue initiale de la trace de l'application CDMS

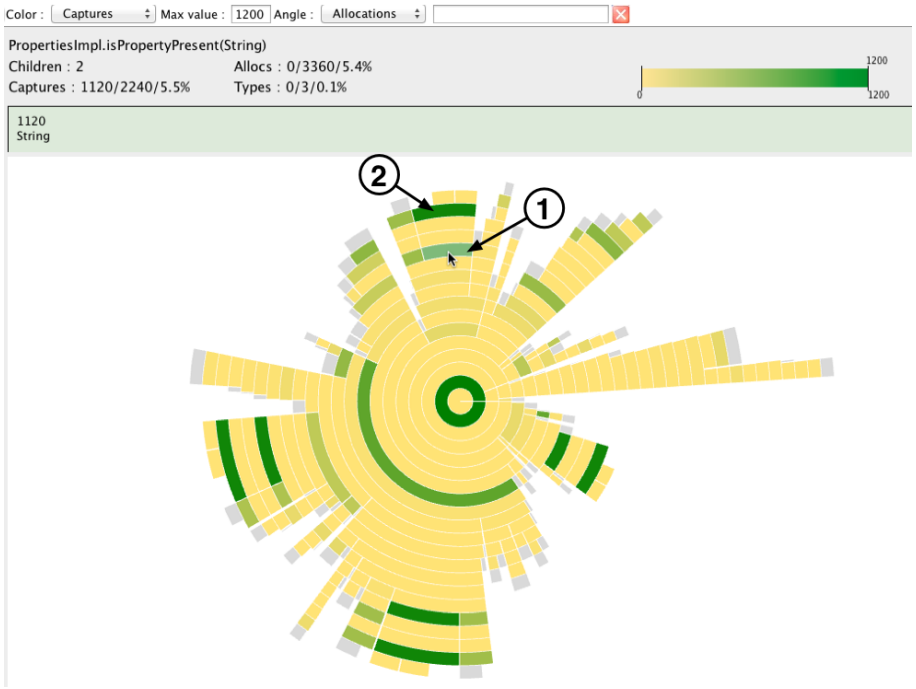


Figure 5.9 – Vue après changement de la métrique des couleurs pour mapper les captures

Cette vue indique que les 1120 chaînes de caractères allouées sont capturées par la méthode `PropertiesImpl.isPropertyPresent` (la méthode avec le label ① sur la figure 5.9). Les régions sous une invocation de cette méthode ont une deuxième source de *churn* (label ② sur la figure 5.9) capturant de nombreuses instances de `StringBuilder` suite à des appels de la méthode `String.toLowerCase`. La méthode `isPropertyPresent` est appelée au total 5370 fois dans la trace. Les quatre contextes d'appels les plus importants comptent à eux seuls 8960 objets temporaires. Des milliers d'objets temporaires sont créés pour s'assurer que les descripteurs de propriétés ne contiennent pas de lettres majuscules. Il serait beaucoup plus efficace pour faire respecter cet invariant pour toutes les entrées une seule fois avant qu'elles soient utilisées dans le système entier.

Nous pouvons retirer toutes les invocations de cette méthode de la vue (figure 5.10) et ainsi continuer notre étude de la trace. Pour cela, on sélectionne l'option "Éliminer toutes les invocations".

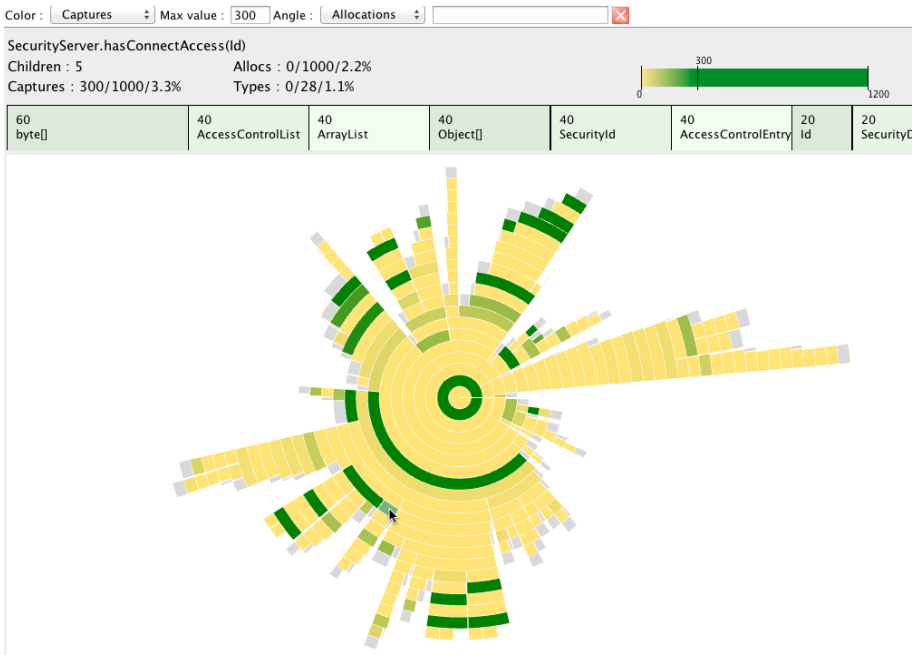


Figure 5.10 – Vue après élimination de toutes les invocations de la méthode `PropertiesImpl.isPropertyPresent`

Nous ajustons le seuil de la métrique à 302 afin de trouver d'autres régions où il

y a un fort taux de captures. Dans la figure 5.10, la souris se trouve sur la méthode `SecurityServer.hasConnectAccess`, qui capture un grand nombre d'objets. Nous pouvons lire sur le panneau d'informations que cette méthode capture 300 objets de huit types différents. La région se trouvant sous cette méthode capture 700 objets. La méthode `hasConnectAccess` est invoquée 20 fois dans la trace et elle crée 20 structures de `SecurityDescriptor` (une par invocation). Pour chaque `SecurityDescriptor` créé et initialisé, il en résulte 1000 objets temporaires qui sont créés.

Une inspection manuelle du code révèle que les instances de `SecurityDescriptor` sont créées pour des instances spécifiques de `Id`. Mettre en cache les objets `SecurityDescriptor` avec leurs `Id` associés pourrait éviter la plupart de ces créations d'objets temporaires.

CHAPITRE 6

CONCLUSION

Dans ce mémoire, nous avons présenté une nouvelle technique de visualisation permettant d'explorer les applications *framework-intensive* afin d'identifier les régions qui sont sources d'*object churn*. La métaphore du Sunburst a été utilisée pour visualiser les résultats de l'analyse d'échappement combinée effectuée sur les données de la trace d'exécution des applications *framework-intensive*. Cette technique a été développée en faisant un compromis entre la précision des données visualisées et la mise à l'échelle de celles-ci. Ainsi, l'analyse d'échappement combinée a permis de garder les données nécessaires à la recherche des sources d'*object churn* et de pouvoir les visualiser dans une unique vue.

Nous avons implémenté un outil, Vasco, qui supporte cette technique de visualisation. Notre outil a été construit en suivant trois principes de conception cités dans le chapitre 4. La **quantité de données représentées** est prise en compte en permettant la présence des données dans une vue unique tout en gardant une bonne clarté. Les données des traces d'exécution sont réduites à partir d'une analyse d'échappement combinée afin de garder ou de calculer les informations nécessaires à l'accomplissement de notre tâche qui est de trouver les sources d'*object churn*. Le Sunburst permet une représentation simple et intuitive tout en disposant efficacement de l'espace disponible dans la vue. L'**effort cognitif** est réduit grâce, entre autres, à une représentation en Sunburst intuitive. De plus, l'information sur les entités de la visualisation est présentée de façon à ne pas distraire l'utilisateur. Ainsi, elles se trouvent dans la fenêtre principale à côté de la visualisation. Finalement, nous gardons une **fluidité des transitions et des interactions**. En effet, l'outil sauvegarde et montre visuellement une trace des transitions ou interactions produites (par exemple, la fermeture de sous-arbres ou le panneau contextuel pour le changement de racine).

Trois études de cas sur des programmes représentatifs de grande taille, dont un système commercial, montrent l'efficacité de Vasco. Ces études ont montré qu'il était facile

de trouver les sources d'*object churn*. De plus, plusieurs modifications ont été proposées afin d'améliorer la performance de ces programmes. Finalement, ces études montrent que notre approche est une voie prometteuse pour explorer le comportement dynamique de systèmes complexes de type *framework-intensive*.

Pour finir, nous avons identifié de nombreux travaux futurs. Dans un premier temps, nous voulons développer un ensemble plus riche d'interactions. Par exemple, il peut être utile de rechercher des invocations par type d'instances qu'elles allouent ou capturent. Nous pouvons également améliorer la fluidité des transitions grâce entre autres à des animations. Par exemple, au moment d'un changement de racine, il serait intéressant de voir l'évolution en montrant plusieurs vues de transitions. D'autres études de cas peuvent être effectuées pour confirmer que notre outil permet une amélioration significative de la performance des programmes. Dans le cas où une trace d'exécution ne suffirait pas à analyser le programme, nous pouvons imaginer étendre notre outil pour qu'il fonctionne avec plusieurs traces d'exécution. Finalement, nous planifions d'étendre notre approche à d'autres problèmes qui nécessitent la compréhension du comportement des programmes. Par exemple, notre approche serait utile afin de permettre la visualisation du flot de données à travers une application.

BIBLIOGRAPHIE

[1] Andrea Adamoli et Matthias Hauswirth. Trevis : A context tree visualization & analysis framework and its use for classifying performance failure reports. Dans *SOFTVIS '10 : ACM Symposium on Software Visualization*, 2010.

[2] Glenn Ammons, Thomas Ball et James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *SIGPLAN Notices*, 32:85–96, 1997.

[3] Glenn Ammons, Jong-Doek Choi, Manish Gupta et N. Swamy. Finding and removing performance bottlenecks in large systems. Dans *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2004.

[4] C. Bennett, D. Myers, M.-A. Storey, D. M. German, D. Ouellet, M. Salois et P. Charland. A survey and evaluation of tool features for understanding reverse-engineered sequence diagrams. *Journal of Software Maintenance and Evolution : Research and Practice*, 20(4):291–315, 2008.

[5] Suparna Bhattacharya, Mangala Gowri Nanda, K Gopinath et Manish Gupta. Reuse, recycle to de-bloat software. Dans *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2011.

[6] Adriana Chis, Nick Mitchell, Edith Schonberg, Gary Sevitsky, Patrick O'Sullivan et Trevor Parsons. Patterns of Memory Inefficiency. Dans *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2011.

[7] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar et Samuel P. Midkiff. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(6):876–910, 2003.

[8] Bas Cornelissen, Danny Holten, Andy Zaidman, Leon Moonen, Jarke J. van Wijk et Arie van Deursen. Understanding execution traces using massive sequence and

circular bundle views. Dans *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 49–58, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[9] Wim DePauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John Vlissides et Jeana Yang. Visualizing the execution of Java programs. Dans *Software Visualization : State of the Art Survey, LNCS 2269*, 2002.

[10] Bruno Dufour, Barbara G. Ryder et Gary Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications. Dans *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*, 2008.

[11] P. Dugerdil et S. Alam. Execution trace visualization in a 3D space. Dans *Information Technology : New Generations, 2008. ITNG 2008. Fifth International Conference on*, pages 38 –43, 2008.

[12] Marc Fisher, II, Luke Marrs et Barbara G. Ryder. HI-C : diagnosing object churn in framework-based applications. Dans *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*, pages 379–380, New York, NY, USA, 2010. ACM.

[13] Marc Fisher II, Bruno Dufour, Shrutarshi Basu et Barbara G. Ryder. Exploring the impact of context sensitivity on blended analysis. Dans *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[14] Christopher G. Healey et James T. Enns. Attention and visual memory in visualization and computer graphics. *IEEE Transactions on Visualization and Computer Graphics*, 2011.

[15] D. Jerding et S. Rugaber. Using visualization for architectural localization and extraction. Dans *Proceedings of the Working Conference on Reverse Engineering (WCRE'97).*, pages 56 –65, 1997.

[16] D.F. Jerding et J.T. Stasko. The information mural : a technique for displaying and navigating large information spaces. *Visualization and Computer Graphics, IEEE Transactions on*, 4(3):257 –271, 1998.

[17] B. Johnson et B. Shneiderman. Tree-maps : a space-filling approach to the visualization of hierarchical information structures. Dans *Proceedings of the Visualization*, pages 284 –291, 1991.

[18] D.B. Lange et Y. Nakamura. Object-oriented program tracing and visualization. *Computer*, 30(5):63 –70, 1997.

[19] G. Langelier et K. Dhambri. Visual analysis of Azureus using Verso. Dans *Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007. 4th IEEE International Workshop on*, pages 163 –164, 2007.

[20] Guillaume Langelier, Houari Sahraoui et Pierre Poulin. Visualization-based analysis of quality for large-scale software systems. Dans *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 214–223, 2005.

[21] Nick Mitchell, Edith Schonberg et Gary Sevitsky. Making sense of large heaps. Dans *ECOOP 2009 ? Object-Oriented Programming*. 2009.

[22] Nick Mitchell et Gary Sevitsky. LeakBot : An automated and lightweight tool for diagnosing memory leaks in large Java applications. Dans *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2003.

[23] Nick Mitchell, Gary Sevitsky et Harini Srinivasan. Modeling runtime behavior in framework-based applications. Dans *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2006.

[24] Philippe Moret, Walter Binder, Alex Villazòn, Danilo Ansaloni et Abbas Heydarnoori. Visualizing and exploring profiles with calling context ring charts. *Software : Practice and Experience*, 40(9):825–847, 2010.

[25] A.J. Pretorius et J.J. van Wijk. Multiple views on system traces. Dans *Visualization Symposium, 2008. PacificVIS '08. IEEE Pacific*, pages 95 –102, 2008.

[26] Manos Renieris et Steven P. Reiss. Almost : exploring program traces. Dans *Proceedings of the 1999 workshop on new paradigms in information visualization and manipulation*, NPIVM '99, pages 70–77, New York, USA, 1999.

[27] Ajeet Shankar, Matthew Arnold et Rastislav Bodik. Jolt : lightweight dynamic analysis and removal of object churn. Dans *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 127–142, New York, New York, USA, 2008. ACM Press.

[28] Kavitha Srinivas et Harini Srinivasan. Summarizing application performance from a components perspective. Dans *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*, pages 136–145, 2005.

[29] J. Stasko et E. Zhang. Focus+context display and navigation techniques for enhancing radial, space-filling hierarchy visualizations. Dans *Information Visualization, 2000. InfoVis 2000. IEEE Symposium on*, pages 57 –65, 2000.

[30] John Stasko, Richard Catrambone, Mark Guzdial et Kevin McDonald. An evaluation of space-filling information visualizations for depicting hierarchical structures. *International Journal of Human-Computer Studies*, 53(5):663 – 694, 2000.

[31] TPTP. TPTP - test and performance tools platform. URL `http://www.eclipse.org/tptp`.

[32] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan et Omri Weisman. TAJ : effective taint analysis of web applications. Dans *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 87–97, 2009.

[33] Qin Wang, Wei Wang, Rhodes Brown, Karel Driesen, Bruno Dufour, Laurie Hendren et Clark Verbrugge. EVolve : an open extensible software visualization frame-

work. Dans *Proceedings of the Symposium on Software Visualisation (SoftVis)*. ACM Press, 2003.

[34] Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev et Gary Sevitsky. Go with the flow : profiling copies to find runtime bloat. Dans *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[35] Guoqing Xu, Michael D. Bond, Feng Qin et Atanas Rountev. LeakChaser : helping programmers narrow down causes of memory leaks. Dans *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 270–282. ACM, 2011.

[36] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg et Gary Sevitsky. Finding low-utility data structures. Dans *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 174–186, New York, NY, USA, 2010. ACM.

[37] Guoqing Xu et Atanas Rountev. Detecting inefficiently-used containers to avoid bloat. Dans *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 160–173, New York, NY, USA, 2010. ACM.

[38] Yi Zhao, Jin Shi, Kai Zheng, Haichuan Wang, Haibo Lin et Ling Shao. Allocation wall : a limiting factor of java applications on emerging multi-core platforms. 2009.