ANALYSE DE PERFORMANCE DE L'INTERPRÉTEUR D'ALGÈBRE DE PROCESSUS EB^{3}PAI

par

Moulay El Mehdi Ettouhami

Mémoire présenté au Département d'informatique
en vue de l'obtention du grade de maître ès sciences (M.Sc.)

FACULTÉ DES SCIENCES

UNIVERSITÉ DE SHERBROOKE

Sherbrooke, Québec, Canada, 25 novembre 2009

Your file Votre référence
ISBN: 978-0-494-61434-1
Our file Notre référence
ISBN: 978-0-494-61434-1

Le 26 novembre 2009

le jury a accepté le mémoire de Monsieur Moulay El Mehdi Ettouhami dans sa version finale.

Membres du jury

Professeur Marc Frappier
Directeur de recherche
Département d'informatique

Monsieur Benoît Fraikin
Codirecteur de recherche
Département d'informatique

Professeur Bernard Colin
Membre
Département de mathématiques

Professeur Richard Egli
Président rapporteur
Département d'informatique

Sommaire

Un des intérêts d'utiliser des méthodes formelles de spécification dans le développement des systèmes d'information, est de pouvoir se concentrer sur les étapes d'analyse et de conception et de ne plus se préoccuper des détails d'implémentation. Le projet APIS utilise la méthode de spécification EB^3, basée sur une algèbre de processus, pour décrire le comportement fonctionnel des systèmes d'information. Le cœur du projet APIS est l'interpréteur d'algèbre de processus EB^{3}PAI ; cet interpréteur implémente un ensemble de règles permettant l'exécution efficace des actions d'une spécification. Un système spécifié à l'aide de cette méthode est généré automatiquement à l'aide de l'interpréteur.

Ce mémoire présente l'approche utilisée pour étudier les problèmes de performance dont souffre EB^{3}PAI. L'essentiel de cette approche, se base sur des techniques de tests de performances et sur les outils de profilage (*profiling* en anglais) pour récupérer des informations concernant les temps d'exécution et l'utilisation de mémoire. L'analyse statistique des résultats démontre que les performances de EB^{3}PAI sont conformes aux performances prévues à partir des algorithmes. Trois modèles linéaires sont validés pour estimer le temps d'éxécution de EB^{3}PAI. L'approche utilisée pour conduire cette analyse de performance pourra servir de cas d'étude et de guide pour des applications du même type.

Remerciements

Je remercie mon directeur de recherche, le Professeur Marc Frappier, pour son implication dans ce travail de maîtrise

Je remercie mon codirecteur de recherche, Benoît Fraikin, dont les remarques et le soutien m'ont aidé tout au long de ce travail.

Je remercie le Professeur Bernard Colin pour ses précieux conseils et directives.

Je remercie également mes camarades du GRIL pour leur soutien continu.

Je remercie aussi mes parents, mes frères, mes sœurs et toute la famille pour leur support moral et affectif inconditionnel.

Enfin je remercie toutes les personnes qui ont contribué de près ou de loin à l'aboutissement de ce travail.

Abréviations

APIS Automatic Production of Information Systems

AST Abstract Syntax Tree

BD Base de Données

CPU Central Processing Unit

EB^3 Entity-Based Black Box

EB^{3}TG Entity-Based Black Box Transaction Generator

EB^{3}PAI Entity-Based Black Box Process Algebra Interpreter

EP Expression de processus

ER Entité-Relation

GRIL Groupe de Recherche en Ingénierie du Logiciel

IDE Integrated Development Environment

IDG International Data Group

IEEE Institute of Electrical and Electronics Engineers

IS Information System

JVM Java Virtual Machine

JVMPI Java Virtual Machine Profiler Interface

NP-EB^{3}PAI Non Persistent EB^{3}PAI

OODBMS Object Oriented Data Base Management System

OS Operating System

P-EB^{3}PAI Persistent EB^{3}PAI

ABRÉVIATIONS

PE Process Expression

RPM Revolutions Per Minute

SGBDOO Système de Gestion de Bases de Données Orientées Objet

SI Système d'information

TC Test Case

TP Test Plan

Table des matières

Liste des figures

Liste des tableaux

Introduction

Contexte

Les systèmes d'information (SI) sont de plus en plus utilisés dans divers domaines comme les hôpitaux, les télécommunications, le transport et le commerce électronique. Ces SI se caractérisent par de larges bases de données généralement distribuées avec des schémas complexes et un nombre important de relations entre les entités. Ces bases de données vont être sujettes à des requêtes de consultation ou de modification envoyées par plusieurs utilisateurs avec des accès concurrentiels et ceci à partir d'une ou plusieurs interfaces graphiques offertes par le SI.

Le processus de développement des SI passe généralement par une approche semi-formelle ou informelle à l'aide de plusieurs outils. On peut alors créer des diagrammes de classes ou des modèles entités-relations et les traduire en langage *Java* ou SQL. Ceci dit, le comportement du SI reste à développer et à implémenter manuellement, ce qui demande des ressources importantes, augmente le risque d'erreurs et nécessite des tests exhaustifs. Le projet APIS offre une boîte à outils complète pour pouvoir construire des SI à partir de spécifications formelles écrites en EB^3. Ces spécifications décrivent le comportement attendu du SI.

L'interpréteur de l'algèbre de processus EB^3, EB^{3}PAI, représente le cœur d'APIS. Il a été réalisé entièrement en langage *Java*. Son rôle est de vérifier si les actions envoyées par l'utilisateur sont acceptées par le SI. Chaque action vérifiée par l'interpréteur provoque un changement de l'état courant du système. Cet état est mémorisé de manière persistante en utilisant un système de gestion de bases de données orientées objet (SGBDOO) appelé *ObjectStore PSE Pro*.

Problématique

L'un des enjeux majeurs du cycle de développement d'un SI est de pouvoir respecter les limites matérielles et logicielles prescrites. Ceci dit, le but est de réduire le temps d'exécution au minimum afin d'offrir une bonne expérience à l'utilisateur et, ainsi, promouvoir l'image du commerce. En plus des différents types de tests de fonctionnalités, l'analyse de performance doit faire partie des instruments de vérification pour chaque itération du cycle de vie du développement.

Une étude théorique, réalisée lors du développement d'EB^{3}PAI, soutient que les performances de celui-ci devraient être étroitement liées à la taille de la spécification utilisée. Ceci dit, ces performances ne doivent pas être influencées par le nombre de données utilisé lors de l'exécution d'EB^{3}PAI. Pour chaque action à exécuter, l'interpréteur doit récupérer l'état courant du système, exécuter l'action demandée par l'utilisateur et enfin mettre à jour l'état du système. On a pu relever, lors de quelques exécutions de l'interpréteur, que l'exécution de certaines actions prenait quelques secondes (de 2 à 4 secondes), ce qui est très problématique pour un SI. Ceci peut avoir deux causes potentielles : soit que la structure de données utilisée lors de l'exécution est assez lourde à parcourir, ou bien que *PSE Pro* prend un temps considérable pour créer, lire, modifier et mémoriser l'état du système sur le disque dur. Le ramasse-miette (*garbage-collector*) et l'implémentation des méthodes peuvent aussi être à l'origine du problème.

Il existe deux versions d'EB^{3}PAI : avec et sans persistance. La version avec persistance utilise le SGBDOO *Objectstore PSE Pro* pour mémoriser sur le disque dur les objets créés lors de l'exécution de l'interpréteur. Le rôle de *PSE Pro* est de modifier le *bytecode* des classes générées afin que la persistance des objets soit effective. Par contre, les objets créés lors de l'exécution de la version sans persistance d'EB^{3}PAI se situent au niveau de la mémoire vive seulement.

Objectifs

Le but de ce mémoire est de fournir une démarche afin d'analyser le comportement d'EB^{3}PAI ainsi qu'extraire les causes de ses pertes de performance. L'interpréteur peut aussi présenter des erreurs d'exécution dans certains cas isolés. Ceci dit, les objectifs de ce

mémoire sont les suivants :

1. Réfléchir aux sources possibles du problème : poser des hypothèses sur les origines de la perte de performance d'EB^{3}PAI.
2. Faire une analyse de performance de l'interpréteur : récupérer les temps d'exécution des méthodes ainsi que l'utilisation de la mémoire et l'activité du ramasse-miette.
3. Analyser les résultats : établir une comparaison entre les résultats issus de la version non persistante et ceux de la version avec persistance, ainsi qu'estimer, de manière statistique, la linéarité des performances d'EB^{3}PAI par rapport à la taille de la spécification.

Méthodologie

Il est important de suivre une démarche précise pour pouvoir réaliser ce projet. Premièrement, il est indispensable de se familiariser avec EB^{3}PAI. Il faut étudier de près la méthode EB^3 pour pouvoir comprendre le fonctionnement de l'interpréteur et ses résultats. Deuxièmement, il est important d'étudier les différents types de tests, en se concentrant sur les méthodologies de tests de performance ainsi que sur les techniques d'analyses des résultats. La sélection de l'outil de tests vient en troisième lieu, en comparant les *profilers* les plus utilisés et en choisissant celui qui répondra le mieux à nos besoins. En quatrième lieu, il faut délimiter les objectifs des tests, en créant des plans de tests répondant à chaque scénario possible d'exécution. En dernier lieu, faire l'analyse adéquate pour toutes les données récoltées, tout en se basant sur des approches statistiques pour évaluer et estimer les performances d'EB^{3}PAI.

Organisation du mémoire

Le reste du mémoire est organisé comme suit. Le chapitre 1 introduira la méthode EB^3, un concept important pour la compréhension de celui-ci. Ce chapitre parlera aussi du projet APIS et plus spécifiquement d'EB^{3}PAI. Le chapitre 2 présente un article intitulé « *Performance Analysis of the* EB^3 *Process Algebra Interpreter* ». Cet article présente la méthodologie utilisée pour évaluer les performances lors de l'execution d'EB^{3}PAI, ainsi

que l'approche statistique suivie pour analyser les résultats récoltés. En conclusion, on fera le point sur la contribution de ce travail ainsi que les critiques et les travaux futurs. L' annexe A présente l'organisation des données récoltées lors de l'exécution d'EB^{3}PAI, les outils utilisés pour les exploiter ainsi que quelques *scripts* écrits en PL/R qui permettent de calculer la régression.

Chapitre 1

Notions de base

1.1 Méthode EB^3

Définition

EB^3 (*entity-based black-box*) [4, 11] est une méthode formelle de spécification des SI, se basant sur des traces d'entrée-sortie structurées et a été spécialement conçue par M. Frappier et R. St-Denis, elle est composée d'une notation formelle et d'un processus systématique permettant de décrire une spécification du comportement externe d'un système d'information. Une trace valide est une succession d'événements pouvant être acceptés par le système. Une spécification en EB^3 consiste en un diagramme de classes représentant les besoins de l'utilisateur, incluant les entités, les associations et leur attributs respectifs, une expression de processus (EP), dénotée par *main*, définissant des traces d'entrée valides, des fonctions récursives, qui attribuent des valeurs aux entités et aux attributs, des associations et des règles d'entrée-sortie, pour attribuer une sortie à chaque trace d'entrée valide.

Le diagramme ER

Le diagramme entité-relation utilise le principe des modèles de données et des notations graphiques d'UML pour représenter les entités et les associations qui décrivent le SI. Chaque entité renferme les informations concernant ses propres attributs et ses fonctions.

L'algèbre de processus

L'essentiel de la syntaxe d'EB^3 repose sur une algèbre de processus, dont la notation est inspirée de CSP d'Hoare. Une expression de processus est construite à partir d'opérateurs appliqués à des actions. Les opérateurs supportés par la méthode EB^3 sont décrits comme suit :

- La séquence dénotée par ".".
- La fermeture de Kleene dénotée par "*".
- Le choix dénoté par "|".
- L'entrelacement dénoté par "|||".
- La composition parallèlle dénotée par "||".
- La garde dénotée par $\Longrightarrow$.
- Le choix et l'entrelacement quantifié sont définis respectivement par | x : [1, n] et par ||| x : [1 : n] .

Les expressions de processus se définissent en quatre formes :

- Elémentaire : λ et a, une action.
- Binaire : E1.E2, E1| E2, E1|[Δ]| E2.
- *Box* : ⊡ dénote un processus qui a complété son exécution avec succès.
- Unaire : | x : [1, n] : E, ||| x : [1 : n] : E, E^*.

Définitions d'attributs

En EB^3, la représentation d'une base de données nécessite, en plus du diagramme ER, une fonction de définition pour chaque attributs. Une définition d'attribut est une fonction récursive sur les traces valides acceptées par l'expression de processus *main*. Cette fonction calcule la valeur d'un attribut selon l'état courant du système. Plus de détails peuvent être trouvés dans [ ] .

Règles d'entrée-sortie

Les règles d'entrée-sortie attribuent une sortie à chaque trace valide du système pour y récupérer l'information demandée. En d'autres termes, ces règles permettent d'extraire les données d'affichage pour chaque action de l'utilisateur en se basant sur un langage formel de requêtes.

1.2 Projet APIS

L'objectif d'APIS (*Automated Production of Information System*) [5] est de développer un outil de génération automatique de SI exécutables à partir de spécifications formelles écrites en EB^3. Le but est aussi de libérer le programmeur des détails d'implémentation, pour mieux se concentrer sur les phases d'analyse et de spécification.

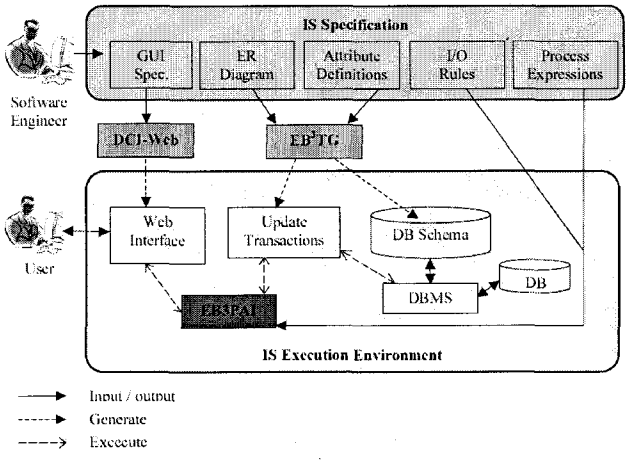

figure 1.1 – Composants du système APIS

APIS contient une composante qui permet de générer automatiquement une interface web, à travers laquelle un utilisateur peut intéragir avec le SI. L'utilisation de l'interface web produira des événements qui seront ensuite analysés par EB^{3}PAI. Le module EB^{3}TG permettra, à de son côté, de générer automatiquement des programmes *Java* qui exécuteront les transactions de BD relationnelle correspondant à la spécification des attributs du SI en EB^3.

1.3 EB^{3}PAI

Comme indiqué précédemment, le projet EB^{3}PAI (*Entity-Based Black-Box Process Algebra Interpreter*) fait partie du projet de recherche APIS ([ ]). EB^{3}PAI est un interpréteur d'expressions de processus EB^3. Il exécute une action en appliquant les règles de transitions correspondantes à une sémantique définie pour l'algèbre de processus d'EB^3. Les transitions sont du style $E \xrightarrow{b} E'$ et sont interprétées par EB^{3}PAI pour déterminer si E peut accepter l'action b et produire E'. E et E' sont, respectivement, les états du système avant et après l'exécution de l'action b.

La première version des règles de transitions définie pour EB^3 avait besoin de remaniement, puisqu'elles n'étaient pas adéquates pour une interprétation symbolique efficace. Ces règles présentaient plusieurs problèmes, parmi ces problèmes on compte le non-déterminisme, où une action peut être exécutée par plusieurs transitions provoquant des résultats différents. La solution qui a été implémentée est un ensemble de nouvelles règles plus complexes, mais proposant une interprétation plus efficace. Cette implémentation est faite en *Java*, gérant aussi la persistance de larges arbres syntaxiques abstraits qui représentent les expressions de processus en utilisant *ObjectStore PSE Pro* qui est aussi implémenté en *Java*.

1.4 Un exemple

Un exemple simple de SI est la gestion des lits dans un hôpital. Ce système permet d'ajouter ou de supprimer des lits, ainsi que d'inscrire ou désinscrire des patients pour un traitement. La figure 1.2 représente un diagramme de classe utilisé pour la construction de la spécification. On y retrouve deux entités, *bed* et *patient*, et une association *occupied* avec leurs attributs et fonctions respectives.

La figure 1.3 décrit le processus **main** qui fait appel aux expressions de processus de l'entité *bed* et de l'entité *patient*. Ces EP décrivent le comportement attendu du système. Ainsi un patient doit pouvoir s'enregistrer, puis occuper un lit, le libérer et enfin se désenregistrer. Un lit peut être créé, puis occupé, libéré et finalement supprimé. On notera que les deux expressions de processus sont synchronisées sur le processus **occupied**.

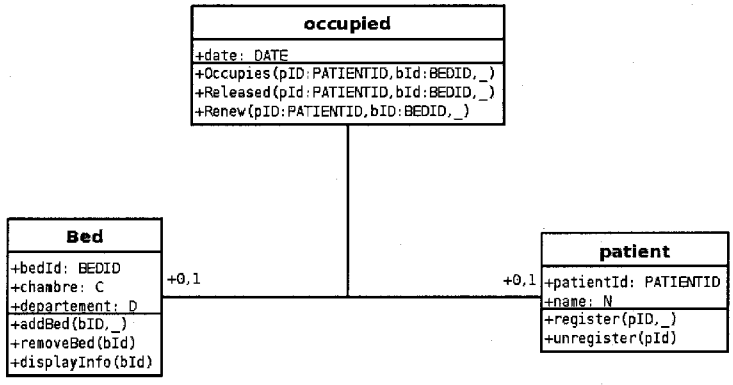

figure 1.2 – Diagramme ER de la gestion de lits

$$\textbf{main} = \; (\; |||\, bId \in \text{BEDID} \; : \; \textbf{bed}\,(bId)^*\;) \;\; ||\; (\; |||\, pId \in \text{PATIENTID} \; : \; \textbf{patient}\,(pId)^*\;)$$

$$\textbf{bed}\,(bId : \text{BEDID}) = \mathsf{AddBed}(bId, \_) \,.\, (\; (|\, pId \in \text{PATIENTID} \; : \; \textbf{occupied}\,(pId, bId))^* \;) \,.\, \mathsf{RemoveBed}(bId)$$

$$\textbf{patient}\,(pid : \text{PATIENTID}) = \mathsf{Register}(pid, \_) \,.\, (\; (\,|\, bId \in \text{BEDID} \; : \; \textbf{occupied}\,(pid, bId)^*\;) \;) \,.\, \mathsf{Unregister}(pid, \_)$$

$$\textbf{occupied}\,(pid : \text{PATIENTID}, bId : \text{BEDID}) = \mathsf{Occupies}(pid, bId, \_) \,.\, \mathsf{Renew}(pid, bId, \_)^* \,.\, \mathsf{release}(pid, bId, \_)$$

figure 1.3 – Expression de processus en EB^3 : Gestion des lits

Chapter 2

Amélioration de la gestion de persistance dans EB^{3}PAI

Résumé

Cet article développe l'approche de test appliquée à l'interpréteur EB^{3}PAI, dans le but de détecter les problèmes de performance que ce dernier rencontre. L'article présente les méthodologies, documentations ainsi que les outils nécessaires pour le processus de test de performance et explique les approche théoriques et pratiques qui répondront au mieux au cas d'EB^{3}PAI. De plus, il expose une analyse statistique complète des résultats obtenus lors du profilage d'EB^{3}PAI.

Commentaires

Ma contribution au sein de cet article consiste dans le développement d'une approche de test adaptée et applicable sur EB^{3}PAI, la création des plans de tests et leurs documentations, la réalisation des tests sur l'interpréteur et l'analyse des résultats, de plus je suis le rédacteur principal de l'article.

Performance Analysis of the EB^3 Process Algebra Interpreter

Moulay El Mehdi Ettouhami, Benoît Fraikin, Marc Frappier and Bernard Colin
Département d'informatique, Université de Sherbrooke,
Sherbrooke, Québec, Canada J1K 2R1
{Moulay.EL.Mehdi.Ettouhami,Benoit.Fraikin,Bernard.Colin,
Marc.Frappier}
@usherbrooke.ca

Abstract

This paper describes a performance analysis case study for a process algebra interpreter called EB^{3}PAI. This interpreter is used for the automatic synthesis of information systems from formal specifications. The paper proposes a methodology for conducting the performance tests and analyzing their results using statistical techniques. A very large volume of data has been collected, stored in a relational database and analyzed using PL/R, a powerful open source statistical package integrated with Postgres. A number of statistical tests have been conducted to determine if the implementation satisfies the predicted algorithmic complexity of the symbolic execution algorithm of EB^{3}PAI.

2.1 Introduction

Information systems (IS) have become largely used in several areas. However the development process remains informal or semi-formal. The behavior of the IS must be implemented manually, which requires significant resources and increases the risk of errors. Formal methods [3] use abstracts languages that cover the phases from specification to implementation. They provide notations, techniques and tools based on mathematical models that help to check that an implementation satisfies a specification. To free the programmer of the implementation details, the Automated Production of IS (APIS) project [5] was launched in 2000, providing a toolbox to automatically generate an executable IS from a formal specification written using the Entity-Based Black-Box (EB^3) method [4]. This specification describes the expected behavior of the IS, by defining the valid traces of input events that the IS must accept.

The core of the generated IS is the EB^3 Process Algebra Interpreter (EB^3PAI) [4]. Its role is to verify if a user input satisfies the specification and can be executed, if not the input is automatically rejected and the state of the IS preserved. This interpreter is coded in *Java*. Usually IS must meet the hardware and software limits on which it runs, providing acceptable performance for database access and processing queries. The IS generated using the APIS framework must also meet these criteria and provide users with performance similar to conventional IS.

For each action to be executed, the interpreter retrieves the current state of the system, executes the action if it is acceptable, and return the new updated system state. EB^3PAI is completely written in Java, the system state is represented as an abstract syntax tree (AST) and stored using an object-oriented database management system (OODBMS), Object Store PSE Pro, (which is also implemented in Java).

An algorithmic complexity analysis has concluded that the performance of EB^3PAI must be linearly related to the size of the formal specification used, and must not be influenced by the amount of data used. But according to some preliminary tests conducted on the interpreter, some performance issues have been detected. Our intended contribution in this paper is to identify in which cases these problems occurs, by proposing a structured test plan and an analysis of its execution results. The present work is based on performance testing approaches and profiling techniques, to create test plans and gather useful informa-

tions on the tests execution. This paper also proposes a technique that estimates execution time depending on the size of the specification. This will help us to identify the source of the performance costs on EB^{3}PAI.

2.2 Background

The EB^3 specification language is a formal, object-oriented, executable specification language for IS. An EB^3 specification consist of four elements; a user requirements class diagram which includes entities, associations, and their respective actions and attributes; a process expression which defines valid input traces of actions, denoted by *main*; recursive functions on traces that assign values to entity and association attributes; and input-output rules, which assign an output to each valid input trace.

2.2.1 EB^3 Process Algebra

The EB^3 process algebra is inspired from well-known process algebra like CSP, CCS, ACP and Lotos. For more details, the reader is referred to [5, 4, 9, 1]. Process expressions (PE) are divided in two types, elementary PE and compound PE. An elementary process expression is either an action $\mathsf{a}(x_1, ..., x_n)$ with a the label of the action and x_i a term, or an internal action denoted by λ. A compound PE is a combination of one or more PE using the following operators: Kleene closure E^*, sequence (also called concatenation) $E_1 \,.\, E_2$, choice $E_1 \mid E_2$, interleave $|||$, synchronization $E_1|[\Delta]|\, E_2$ (with Δ a set of action labels) and guard $p \Longrightarrow E$, where p is a formula. A PE can also include quantified versions of the choice and the interleave.

2.2.2 EB^{3}PAI

An EB^3 process expression describes the set of valid traces that represents the expected behavior of an IS. To execute an action, EB^{3}PAI uses symbolic computation on the operational semantics of the EB^3 process algebra. For instance, the PE $\mathsf{a} \,.\, (\mathsf{b} \,.\, \mathsf{c})$ can execute a and gives as a result the PE $(\mathsf{b} \,.\, \mathsf{c})$. This transition can be represented by $\mathsf{a} \,.\, (\mathsf{b} \,.\, \mathsf{c}) \xrightarrow{\mathsf{a}} (\mathsf{b} \,.\, \mathsf{c})$. It is computed with EB^{3}PAI using inference rules of the operational semantics. The current

state of a specification is the process expression obtained from computing a transition. It is represented in EB^{3}PAI by the abstract syntax tree (AST) of the process expression. EB^{3}PAI implements the persistence of ASTs using Object Store PSE Pro.

2.3 Performance Testing

According to *IDG Research* [8], eighty percent of software products developed with Java do not meet the required performance. The principal cause is that the industry gives more importance to functional testing than performance testing, and invests more money in hardware equipments to mask performance leeks. Software performance testing [6] is conducted to determine how fast some aspect of a system performs under a particular work load. This testing process can serve different purposes: it can demonstrate that the system meets performance criteria, compare two systems and find out which one performs better, or measure which component causes the system to perform badly. Load testing [8] must not be confused with performance testing. The aim of a load test is to determine if a system gives the right result even on high load scenarios, while a performance test returns time consumption and memory usage of an execution.

2.3.1 Methodology

In the industry, performance testing is usually conducted at the end of the software development life cycle. This practice increases the cost of debugging, and may delay the release of the product. The adoption of a more efficient process becomes an emergency for several organization. This methodology, known as the early performance testing [8, 2], maintains that the performance tests must be applied during the first stages of the development. The goal is to associate one or more performance tests to each functional test conducted (unit test, integration test...), in order to detect performance leeks as early as possible. Figure 2.1 is an example of performance testing integration into a V-model development process [8]. In our case, since the development process of EB^{3}PAI is already done, there is no other choice but to apply the performance tests on the current version of the interpreter.

Before starting the testing process, a detailed methodology must be described and applied in a logical way. In other words, the tester must first understand how the applica-

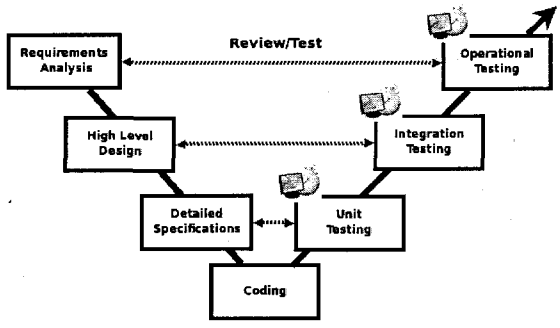

Figure 2.1: Performance testing in V-Model

tion works, and identify the roles of every external component used. After that, he must specify the overall objective of the testing process, decomposes it into several detailed sub-objectives, and provide a complete documentation, covering the theoretical and the practical aspects of the testing process.

Figure 2.2 shows the workflow representing the test execution process followed. Once the test plan and test case created, we proceed to the creation of the test specification files that represent each test case. These files are created manually respecting the EB^3 syntax and ensuring that they can be used in EB^{3}PAI. However, test inputs are generated automatically, using a script written in Java specifically for this reason. We need simply to specify action labels, their range of cardinality and the execution order to generate the desired test input file. The next step is to launch the test run using the test files specifications and their respective test inputs. This stage is divided into two parts; gathering time per action and the profiling session.

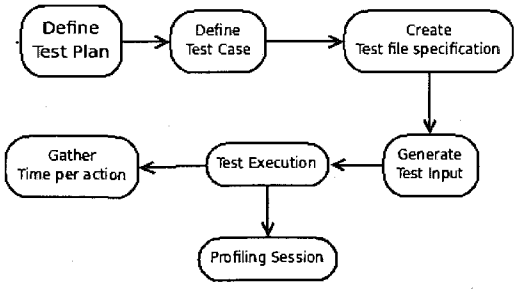

Figure 2.2: Test execution process

2.3.2 Documentation

The testing process uses and generates a lot of informations, so it is important to provide detailed documentations covering the modeling and the application of the required tests. The IEEE proposes the 829-1998 standard [10] that describes a set of basic software test documents. Their purpose is to facilitate communication between different actors by using a common frame of reference. The test manager is not forced to use all test documents described in this standard; they can be adapted to meet the desired context.

Organizing the documentation is very important, in order to follow the logic of the testing process. Fist of all, we must create the main test plan. It is a document that specifies the objective of the test, and describes also the items being tested, the features to be tested, the testing tasks to be performed, their order and in which environment they will be performed. A test plan must describe several test cases, each one defining a sub-objective to be verified. A test case must also describe how to satisfy its objective, and that includes the inputs required, the output expected and the execution procedure of the test case. After the execution of a test, we can use two kinds of documents. A test log document providing a chronological record of details about the executions of tests, and test incident reporting any event that occurs during the testing process and needs investigation.

2.3.3 Profiling

To quantify the behavior of a program during its execution we need a profiler. It helps to gather information on the frequency and duration of function calls. To complete this task, the profiler uses a variety of techniques such as hardware interrupts, code instrumentation, and a fishing system of the operating system. The output may be a record of all events occurred, or a statistical summary.

When it come to profile Java application, the majority of profilers available on the market use the Java Virtual Machine Profiler Interface (JVMPI)[1]. JVMPI is a two-way function call interface between the Java virtual machine and an in-process profiler agent that monitors the behavior of the virtual machine. In other words, with this interface it is possible to analyze all events that can occur during program execution. Analyzers that implement this interface are running at the same time as the Java application, and use the

[1] http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html

same virtual machine. This can affect the normal execution of the application, that's why these profilers must be selected according to their accuracy and effectiveness.

The *Netbeans* development environment includes, since the 6th version, an interesting profiler. Developed by *Sun Microsystems*, it has the advantage to be stable, precise and well documented. *Netbeans profiler*[2] allows the analysis of processor and memory usage with different levels of accuracy. The profiling can be applied on a Java package, a class or on a piece of source code. We can also monitor in real-time the interventions of the garbage collector.

We have selected Netbeans profiler for analyzing the execution of our test cases for many reasons: first of all, the profiler is included in *Netbeans* IDE, this allows to run profiling session and interact with the Java code easily. In comparison with other profiler (*Eclipse Profiler*[3] and *AppPerfect*[4]), Netbeans profiler imposes relatively low overhead and it's even more precise (up to one thousandth of a millisecond).

2.4 Test Plan

In our case, the objective of the test phase is to cover the behavior of EB^{3}PAI, by specifying the maximum (within the resources available) of plausible scenarios organized in test plans. The preparation of test plans first requires an understanding of the context in which the application will run.

2.4.1 Application Context

EB^{3}PAI is a process algebra interpreter for the EB^3 process algebra; it takes as an argument a formal specification describing the actions permitted by the system denoted by *main*.

$$main = |||\, x \in [1..10] \;:\; \mathsf{a}(x)\,.\,\mathsf{b}(x)\,.\,\mathsf{c}(x)$$

The above example accepts three actions, a,b and c. This system uses a quantified version of interleave that begins at 1 and ends at 10; this means that for each value of x we can execute the three actions. After being lanched, EB^{3}PAI is waiting for a user input to verify

[2] http://www.netbeans.org/kb/60/java/profiler-intro.html
[3] http://eclipsecolorer.sourceforge.net/index_profiler.html
[4] http://www.appperfect.com/products/java-profiler.html

if it is executable or not. If we execute the action a(1) on this example, the result will be:

([x=1] : b(x) . c(x))

|||

(||| $x \in [1..10] \setminus [1]$: a(x) . b(x) . c(x))

This result is the new state of the system, it is represented in Figure 2.3 as an AST. The leaves of the tree are the elementary actions of the PE and internal nodes represent operators. At the moment there are two versions of EB^{3}PAI. The first one supports the

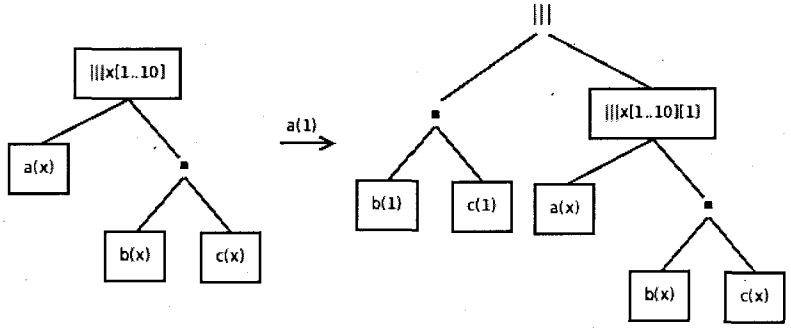

Figure 2.3: AST representation of an execution

persistence of ASTs with ObjectStore PSE Pro, which will store on the hard disk the AST. A second version that does not use persistence and keeps the AST in main memory only. The aim is to test every scenario on the two versions and retrieve the result of their profiling. This will help us in the future to determine the impact of the persistence on execution time.

2.4.2 High-Level Test Plan

We've had to take into consideration that the development of EB^{3}PAI is already done, and we can not apply an approach based on an early performance testing. In fact, we have to apply a performance testing process on the final application. We start by describing execution scenarios using the maximum of functionality supported by EB^{3}PAI. These scenarios are ordered according to their size, and organized according to common objectives in test plans. Figure 2.4 shows an overview of the testing organization used in our case. This pattern is based on the IEEE 829-1998 standard.

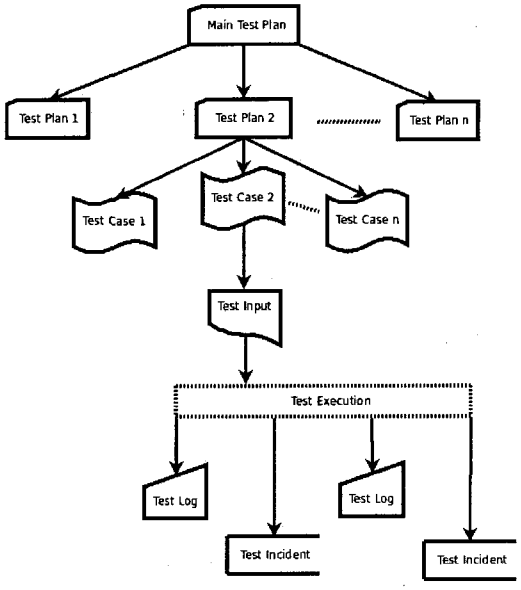

Figure 2.4: Test organisation based on IEEE 829-1998

The first step was to define the master test plan that allows us to determine the overall objective and provide some informations on the global context. This informations are detailed in the sub test plans and test cases.

Main test plan.

Test objective: Evaluate EB^{3}PAI performance, by comparing the two versions of the interpreter (with and without persistence), and analyzing its behavior when using different operators. We also have to use realistic examples to understand the impact of large specifications on the interpreter's performance, and to find the source of EB^{3}PAI performance leeks.

Feature to be tested: Analyzing the behavior of the operators supported by EB^{3}PAI will help us to understand the impact of each one on a specification. Quantification management and treatment of large PE by the interpreter must also be verified, and compared between the two available versions, in order to test the performance management in EB^{3}PAI.

Test order: The testing process should follow a logical order to meet the specified objectives. In our case, tests are conducted according to their size and algorithmic complexity. First of all, we evaluate basic operators supported by EB^{3}PAI (*, ., |, |||) and then we test quantification management of the interpreter (| $x \in z : E$ and ||| $x \in z : E$). To analyze the impact of large PE execution, we use specifications that implement entities and relations. We can then increase the complexity of the specification, by adding more entities or associations.

Environment: Hardware environment:

- CPU: Intel Quad Core Q6600 2,4GHz.
- Memory: 2Go DDR2.
- Hard disk: 500Go 7200rpm.

Software environment:

- OS: Vista Pro 32bit SP1 (Fresh install, NOD32 Antivirus).
- IDE: Netbeans 6.
- Java: JVM 1.5.
- Profiler: Netbeans Profiler.

It is necessary to separate the overall objectives of the master test plan in several objectives and order them into test plans. In our case, each test plan covers a test feature to check. The details about the proposed test plans are the following:

Test plan 1.

Objective: Evaluate the behavior of basic operators.
Features to be tested: Concatenation and choice management.
Approach: Define test cases using the features to be tested and increment the complexity of the specification. These tests will be run on both versions in order to compare performance.

Test plan 2.

Objective: Evaluate the behavior of quantification management of the interpreter.
Features to be tested: Interleave and synchronization operators, and quantified versions of ||| and |.
Approach: Define test cases using, first, simple specifications with interleave and synchronization operators. Then we will focus on their quantified versions and compare their impact on the persistent and non-persistent version of EB^{3}PAI.

Test plan 3.

Objective: Determine the impact of the number of associations and their cardinalities on the performance of EB^{3}PAI.
Features to be tested: Execution of large PE implementing associations, entities and cardinality management.
Approach: First, we use the library example, represented by two entities (book and member). Then, we increase the number of associations between these entities, and determine the impact of each case. The class diagram shown in Figure 2.5 is used in this test plan. We create four test cases, using from one to four associations between entities *book* and *member*.

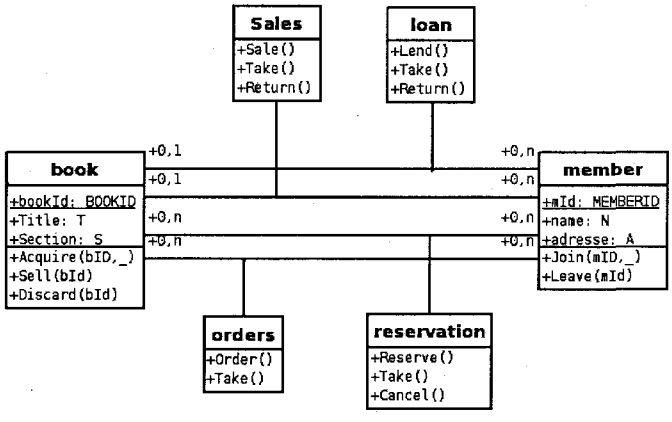

Figure 2.5: Class diagram of test plan 3

Test plan 4.

Objective: Determine the impact of the number of entities on the performance of EB^{3}PAI.

Features to be tested: Treatment of large PE implementing associations, entities and cardinality management.

Approach: We use the library example, and fix the number of association to one. The purpose is to increase the number of entities that use the same association, and determine the impact of each case. We use the ER diagram described in Figure 2.6 to create three test cases.

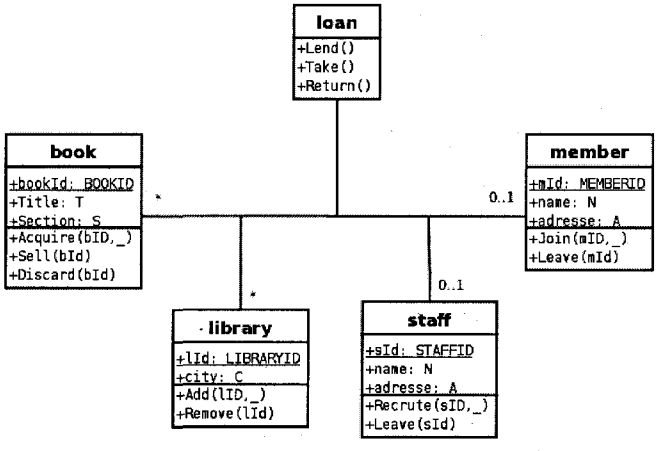

Figure 2.6: Class diagram of test plan 4

To meet the objectives described earlier, we create multiple scenarios for each test plan. These scenarios are represented as test cases. In the next section we will discuss the experimental application of performance testing, by defining test cases and test input thats correspond to each test plan.

2.4.3 Detailed Test Plan

Once the outline of the testing process is dressed, we proceed to the experimentation. We need, at this stage, to define the procedures to follow to implement the tests in a practical

way. In our case, several test cases must be extracted from the test plan that have been defined before, Each test case represents an execution scenario of the interpreter. These scenarios must be in relation with the objective to verify.

As said before, the objective of test plan 1 is to evaluate the behavior of the basic operators supported by EB^{3}PAI. Table 2.1 details the test cases required to meet this target. The Test Case column describes the EB^3 specification that implements the test case objective, and the actions that will be passed as input to the interpreter. In the first place we test the sequence operator, using a simple specification that allow the execution of a followed by b. To test the Kleene closure, we use a specification that implements a P.M.C (producer-modifier-consumer) configuration, which often occurs in IS, that allows us to execute a once, then an arbitrary number of b and finally c. To evaluate the choice operator, we can use a specification that give us the choice to execute either a or b. We add a Kleen closure to be able to execute this specification as many time as we need.

Table 2.1: Test Plan 1

ID	Test Objective	Test Case	
		Spec.	Input
1-1	Sequence operator	(a . b)*	a,b,a,b,..,a,b
2-2	P.M.C configuration	a. b*. c	a,b,b,..,b,c
3-3	Choice operator	(a | b)*	a,b,a,b,..,a,b

Table 2.2 introduces test cases that meets the objective of the second test plan. The purpose is to evaluate some scenarios where interleave(|||) and synchronization (||) are used. In test cases 2-1 and 2-2, we want to test these operators using their basic versions. Using test cases 2-3 and 2-4, we can evaluate the impact of the quantified version of choice (|) and interleave (|||). In test cases 2-5 and 2-6, we want to increase the complexity of the specification. Process P(i) is defined as a(i). b(i). c(i), and the process Q(i) is defined as d(i). b(i). e(i). In 2-5, the two processes will be synchronized on action b(i).

Most of the time, EB^{3}PAI will be used for specifications describing IS. That's why we should evaluate the behavior of the interpreter in such cases. Test plan 3 uses the library example, as an IS, that implements two entities, book and member. The purpose of the test cases described in Table 2.3, is to see if increasing the number of associations between these two entities will affect the behavior of EB^{3}PAI. The specification used in test plan 3 is described in Figure 2.7.

Table 2.2: Test Plan 2

TC	Test Objective	Test Case	
		Spec.	Input
2-1	Synchronization operator	(a || a)*	a,a,..,a
2-2	Interleave operator	(a ||| b)*	a,b,a,b,..,a,b
2-3	quantified version of |	(| i : [1..n] : a(i))*	a(1),a(2),..,a(n)
2-4	quantified version of |||	(||| i : [1..n] : a(i))*	a(1),a(2),..,a(n)
2-5	Synchronization on two quantifications	(||| i:[1..n]:P(i)) || (||| i:[1..n]: Q(i))	a(1),d(1),b(1), c(1),e(1)..a(n), d(n),b(n),c(n),e(n)

Table 2.3: Test Plan 3

ID	Test Objective	Test Case	
		Spec.	Input
3-1	Library example Two entities, **one** association	(||| bId:[1..n]:book(bId)) || (||| mId:[1..n]:member(mID))	Acquire, Join, Lend, Return
3-2	Library example Two entities, **two** associations	(||| bId:[1..n]:book(bId)) || (||| mID:[1..n]:member(mID))	Acquire, Join, Reserve, Take, Return
3-3	Library example Two entities, **three** associations	(||| bId:[1..n]:book(bId)) || (||| mId:[1..n]:member(mID))	Acquire, Join, Reserve, Sale, Take, Return

In test case 3-1 we use two entities and one association (*loan*). In test case 3-2, we add a new association called *reservation* and finally, in test case 3-3, we add a third association called *sale*. We can notice that input actions change from a test case to another; that is because these associations add new actions to the specification.

In test plan 4, we increase the number of entities in each test case, using just one association between them. Figure 2.8 presents the full specification used in test plan 4. We can notice in Table 2.4 that test case 4-1 uses two entities and one association; it's exactly the same configuration as TC 3-1. Test cases 4-2 and 4-3 use, respectively, three and four entities and one association.

Test inputs are generated automatically using a tool created specifically for this purpose. In the next section, we will present the test execution process, and the output organization.

```
main =  ( |||| bId ∈ BOOKID : book(bId)* )
      || ( |||| mId ∈ MEMBERID : member(mId)* )

book( bId : BOOKID ) = Acquire(bId, _)
                       .(
                           (| mId ∈ MEMBERID : loan(mId, bId))*
                         ||
                           (||| mId ∈ MEMBERID : reservation(mId, bId))*
                         ||
                           (||| mId ∈ MEMBERID : sales(mId, bId))*
                       )
                       .(Sell(bId) | Discard(bId));

member( mid : MEMBERID ) = Join(mid, _)
                           .(
                               (||| bId ∈ BOOKID : loan(mId, bId))*
                             ||
                               (||| bId ∈ BOOKID : reservation(mId, bId))*
                             ||
                               (||| bId ∈ BOOKID : sales(mId, bId))*
                           )
                           .Leave(mid, _) ;

loan( mid : MEMBERID, bId : BOOKID ) =
(Lend(mid, bId, _) | Take(mid, bId, _) ) . Renew(mid, bId, _)* . Return(mid, bId, _);

reservation( mid : MEMBERID, bId : BOOKID ) =
Reserve(mid, bId, _) . (Take(mid, bId, _) | Cancel(mid, bId, _) );

sales( mid : MEMBERID, bId : BOOKID ) =
(Take(mid, bId, _) | Sale(mid, bId, _) )
```

Figure 2.7: Process expression used in test plan 3

2.5 Test Execution

Once the test cases specified and the test input generated, we start the test execution phase. In this section we will detail the type of data to collect, their organization and the methodology used to analyze these results.

```
main =  ( ||| bId ∈ BOOKID : book(bId)* )
      || ( ||| mId ∈ MEMBERID : member(mId)* )

book(bId : BOOKID) = Acquire(bId, _)
                   . (
                       (| mId ∈ MEMBERID :
                         (||| sId ∈ STAFFID :
                           ||| lId ∈ LIBRARYID : loan(lId, sId, mId, bId)))*
                     )
                   . (Sell(bId) | Discard(bId));

member(mid : MEMBERID) = Join(mid, _)
                       . (
                           (| bId ∈ BOOKID :
                             (||| sId ∈ STAFFID :
                               ||| lId ∈ LIBRARYID : loan(lId, sId, mId, bId)))*
                         )
                       . Leave(mid, _) ;

staff(sid : MEMBERID) = Recruit(sid, _)
                      . (
                          (| bId ∈ BOOKID :
                            (||| mId ∈ MEMBERID :
                              ||| lId ∈ LIBRARYID : loan(lId, sId, mId, bId)))*
                        )
                      . LeaveStaff(sid, _) ;

library(lid : LIBRARYID) = Add(lid, _)
                         . (
                             (| bId ∈ BOOKID :
                               (||| mId ∈ MEMBERID :
                                 ||| sId ∈ STAFFYID : loan(lId, sId, mId, bId)))*
                           )
                         . Remove(lid, _) ;
loan(lid : LIBRARYID, sid : STAFFID, mid : MEMBERID, bId : BOOKID) =
(Lend(lid, sid, mid, bId, _) | Take(lid, sid, mid, bId, _) ) . Renew(lid, sid, mid, bId, _)* ;
```

Figure 2.8: Process expression used in test plan 4

2.5.1 Input configuration

To simulate the approximate conditions under which EB^{3}PAI will be executed, all test inputs consist of a set of twenty thousand (20,000) actions. Each test input is organized according

Table 2.4: Test Plan 4

ID	Test Objective	Test Case	
		Spec.	Input
4-1	Library example **Two** entities, one association	(||| bId:[1..n]:book(bId)) || (||| mId:[1..n]:member(mID))	Acquire, Join, Lend, Return
4-2	Library example **Three** entities, one association	(||| bId:[1..n]:book(bId)) || (||| mId:[1..n]:member(mID)) || (||| sId:[1..n]:staff(sID))	Acquire, Join, Recruit,Lend, Return
4-3	Library example **Four** entities, one association	(||| bId:[1..n]:book(bId)) || (||| mId:[1..n]:member(mID)) || (||| sId:[1..n]:staff(sID)) || (||| lId:[1..n]:library(lID))	Acquire, Join, Recruit, Add, Lend, Return

to its respective input model described above for each test case. The results of a test run may be interfering with some random factors we cannot control, like competing OS processes, disk access, etc. To reduce these factors, every test case is executed at least five (5) times.

To collect the execution of each action, we insert time collection instructions in appropriate place in the source code of EB^{3}PAI, and write these times into a text file. The test execution process can take a lot of time, depending on the test case. We have measured in average five (5) to eight (8) days to execute a single test case on the early version of EB^{3}PAI. We will see later a performance comparison after applying some improvements.

2.5.2 Output type

There are several kinds of outputs that we will be interested in after each test run, these outputs will help us to make a behavioral analysis of the interpreter. The profiling session will gather relevant informations on the execution of the interpreter. CPU profiling allows us to analyze the number of methods calls and their respective time execution. This will help discovering CPU bottlenecks. With the analysis of memory usage, we will asses the memory management of the JVM, and the impact of garbage collector on EB^{3}PAI's performance.

Since a test case input contains twenty thousand (20,000) actions and will be executed five times, we will gather up to one hundred thousand (100,000) executions time for each

test case. A good way to handle this large amount of data is to store it in a database, in order to facilitate access and manipulation.

2.5.3 Output organization

An important step after finishing collecting informations on the performance, is to organize everything in a coherent and structured way. We have decided to store all the results into a *PostgreSQL*[5] database to be able to access these data easily and quickly. The entity relationship diagram of this database is shown in Figure 2.9; this diagram is based on the test organization described in Figure 2.4. The *TestInputLine* table contains the list of actions used in every test case, and *TestRunOutputLine* contains the execution time for each one of these actions.

In order to analyze the results, we use a powerful statistical tool called *R*[6]. The *R* language is a programming language and a mathematical environment for data processing and statistical analysis. *R* also offers powerful tools for graphs generation that we will use to present our conclusions. To create a link between the *PostgreSQL* and the *R* tool we will use *PL/R*[7]. *PL/R* is a procedural language that can create *PostgreSQL* functions with the capacity to trigger functions of the R language, in other words, PL/R allows the interaction with the database (using queries) and use the results as inputs for *R* functions. Using this set of tools, we can proceed to a deep analysis of all the results available on the database.

2.6 Results analysis

In this section we will analyze the informations collected after the completion of each test. Firstly, we will begin with an overview of the EB^{3}PAI performance based on profiling data. given the large number of test cases used, we will illustrate these results using a single test case as example. The next step is to analyze overall EB^{3}PAI's performance statistically, and to find a way to predict its behavior.

[5] http://www.postgresql.org/
[6] http://www.r-project.org/
[7] http://www.joeconway.com/plr/

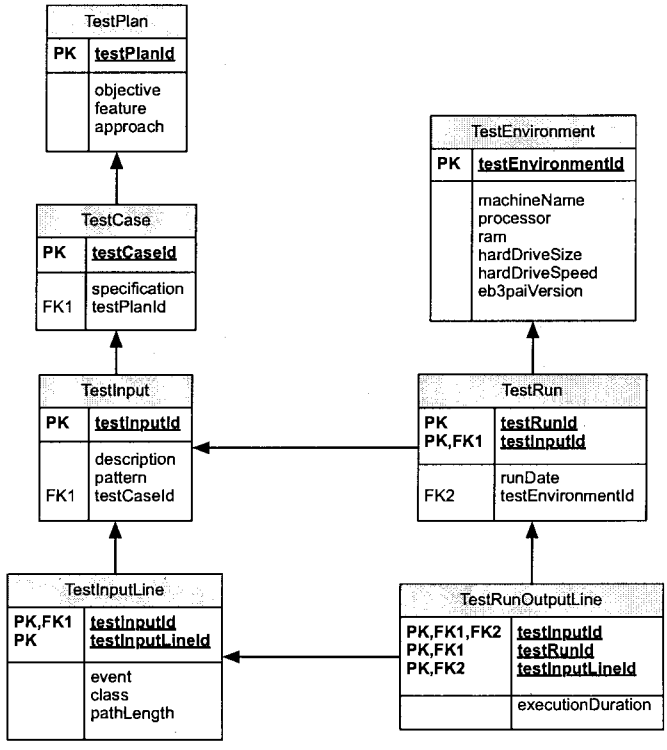

Figure 2.9: ER diagram of result data base

2.6.1 Profiling analysis

Every test case executed has been the object of a profiling session, in order to extract various information about the CPU utilization and memory management. We will use as example, the profiling data from the execution of test case 3-3 described on Table 2.3.

CPU usage

In our case, the primary purpose of applying CPU profiling was to determine the impact of persistence management on the execution of EB^{3}PAI. We need then to extract time informations about all methods invoked during an execution, and calculate their percentage. The Netbeans profiler can do all the job for us, and organize the results by methods, classes

or packages. We have used the packages sorts option to distinguish the EB^{3}PAI methods call from those of PSE Pro, responsible for the persistence capability.

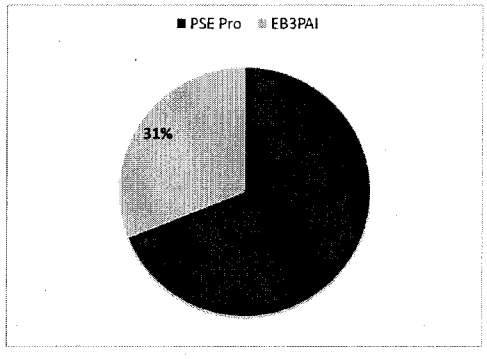

Figure 2.10: CPU profiling: methods calls

The graph in Figure 2.10 shows the percentage of time spent in methode calls when running the test cases 3-3. The package PSE Pro uses 69 per cent of total running time, this includes the initialization of the database and all disk accesses. The remaining 31 per cent concern the EB^{3}PAI package; it includes the initialization of the AST, the reading of the inputs, execution of actions and outputting the result and log files.

We have also detected, using performance testing, a major bug which affected directly the performance of EB^{3}PAI. Figure 2.11 shows execution time per action using the test case 3-3 on the first persistent version of EB^{3}PAI. We can notice that the execution time start to evolves dramatically, and go up to two seconds per action. In this case, we can see that the number of actions executed have a direct impact on the time consumption, specially when these actions are in the scope of a synchronization. Following the initial algorithmic complexity analysis in [3], this behavior is not expected. An investigation have been done, and found that the source of the problem was in a module called *processManager*. This module was supposed to optimize the overall performance by storing in a hashmap all visited nodes and use it for the following executions. The implementation was such that the same hash key was used for each node, causing a linear increase in execution time. However, after reviewing the benchmarks produced by this study, the module shows no more interest since the gain is negligible.

After optimizing the process manager module, the performance of EB^{3}PAI has been

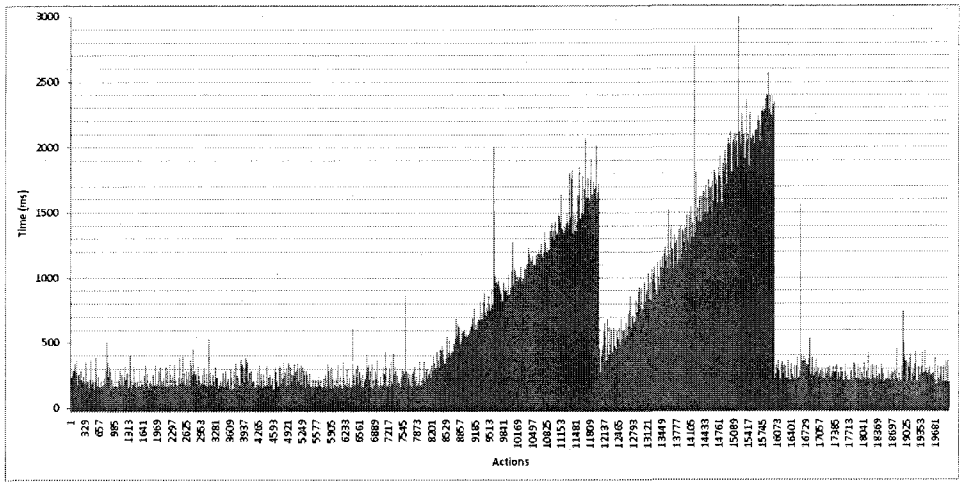

Figure 2.11: Performance leek on the early version of EB^{3}PAI

significantly improved. Figure 2.12 shows the execution of test cases 3-3 on the persistent version of the interpreter. We can notice a stable behavior comparing with the first version for the same number of actions.

Memory usage

At this stage, we will be interested in three major informations; the total heap size allocated by the *JVM* (in bytes), the heap used during the execution of EB^{3}PAI (in bytes) and the relative time spent in garbage collecting (in percentage). We will apply the memory profiling on both versions of EB^{3}PAI, with and without persistence.

Figure 2.13 shows the evolution of informations concerning the memory profiling on the non persistent version of EB^{3}PAI (NP-EB^{3}PAI). We can notice that both the allocated and the used heap size evolves almost linearly during the execution. Garbage collector remains relatively stable except some times where it uses almost 45 per cent of the JVM resources.

Figure 2.14 shows the results of the memory profiling applied on the persistent version of EB^{3}PAI (P-EB^{3}PAI). We can notice the same linear evolution as the non persistent version. Therefore the maximum heap size used on the persistent version of the interpreter for this execution goes up to fifty megabytes (50Mb), versus the non-persistent version that

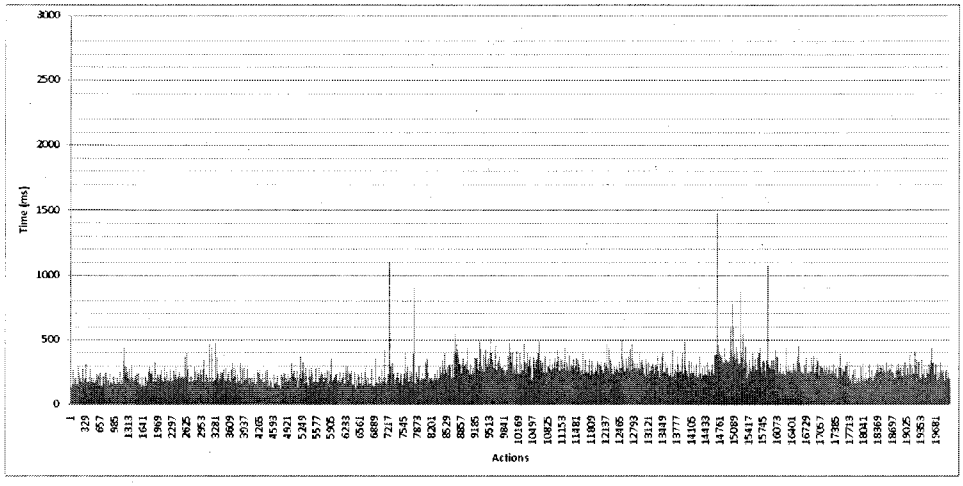

Figure 2.12: EB^{3}PAI performance after optimization

stops at twenty four megabytes (24Mb). The relative time spent on garbage collection does not exceed five percent, and it is stable throughout the execution.

In the next section, we use the data gathered for the execution of each action to understand the behavior observed by the profiling analysis. We will also discuss different statistical approaches that can serve as a basis to predict EB^{3}PAI end behavior.

2.6.2 Regression analysis

Many studies are trying to explain the relationship between a variable and other variables. The sought relation is written as follows:

$$Y=f(X_1,..,X_n)$$

with

- Y is called the explained, or response variable.
- $X_1,..,X_n$ the explanatory variables or factors.
- $f(X_1,..,X_n)$ represents the mathematical function that links the response factors.

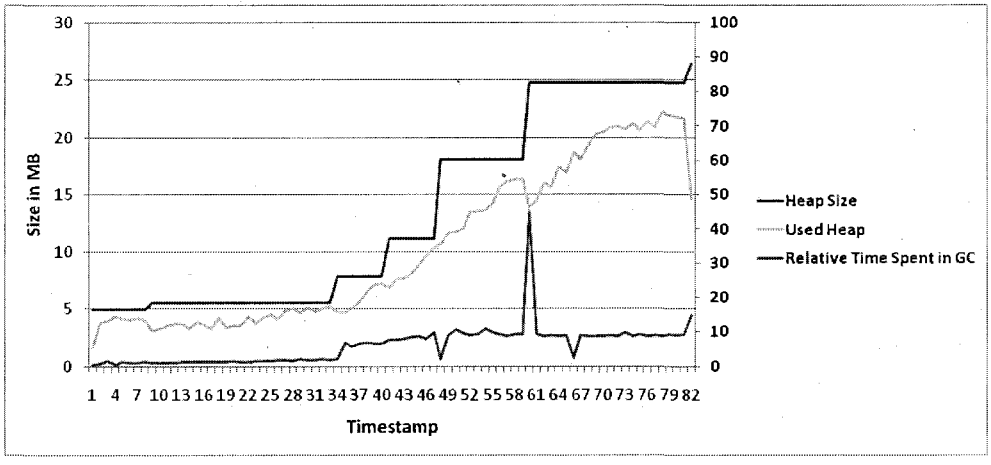

Figure 2.13: Memory usage (NP-EB^{3}PAI)

In practice, linear model [13] are largely used. For instance, if we consider the relation $Y = f(X_1, X_2)$ described as $Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2$, we can conclude that this model is a polynomial and belongs to the linear models class. The p coefficients β_i (i = 1, ..., p) are unknown parameters and their values must be assessed. If the relationship between Y, X_1 and X_2 was perfectly correct, it would suffice to know the values of Y, X_1 and X_2 factors for m gathred observations and solve a system equation with p unknowns to deliver the values of parameters β_i.

However, a relationship chosen to explain a given phenomenon is rarely accurate. First, a model is generally an approximation of a phenomenon much more complex. In addition, any experiment repeated twice under conditions believed to be identical, gives rarely the same result. The variations are usually caused by a multitude of external factors that we do not control (ex: Garbage Collector). It is therefore reasonable to attach to any model, supposed to reflect a complex phenomenon, a random term that represents the difference between the theoretical model chosen and the gathered observations. The term called random error term (ε), will be added to the model as shown by the following relationship: $Y = \beta_0 + \beta_1 X_1 \beta_2 X_2 + \varepsilon$.

In our case, the objective is to describe a model that can characterize the execution time of EB^{3}PAI. We will analyze three models, each model will try to explain the execution time using different factors. The first factor is the action class, the second is what we call the

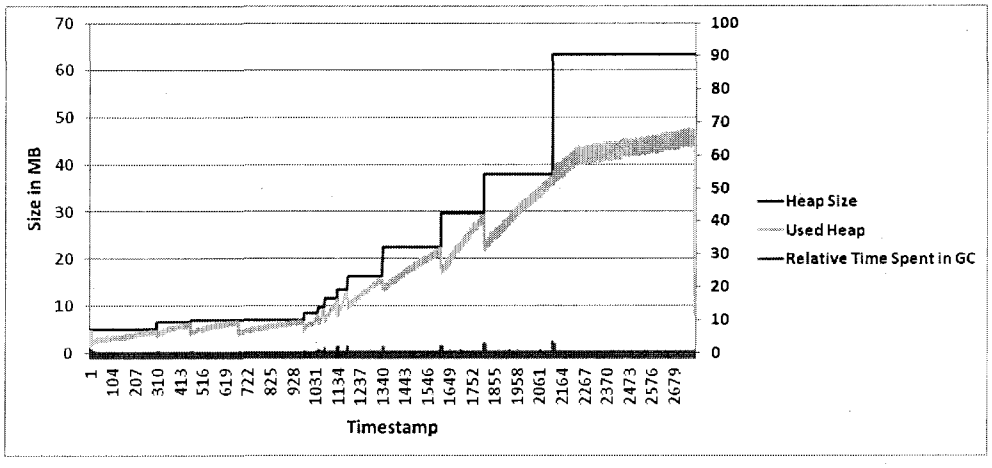

Figure 2.14: Memory usage (P-EB^{3}PAI)

path length and the third is a combination of the first and the second factor. The estimation of parameters of these models will require the use of statistical methods. The proposed technique called regression analysis [12] can calculate estimates for the parameters, using series of observations of the response and explanatory factors. We will use as observations the data gathered on the execution of EB^{3}PAI, and do the analysis for the non-persistent and the persistent version of the interpreter.

To evaluate the precision of a model, we need to look to the R-square (or the coefficient of determination) that has been calculated at the same time as the regression using this equation:

$$R^2 = \frac{\sum_i (\widehat{y}_i - \overline{y})^2}{\sum_i (y_i - \overline{y})^2}$$

with

- R^2 is the coefficient of determination.
- $\widehat{y}_i$ is a predicted value using the regression.
- y_i is a real observation.
- $\overline{y}$ is the mean of variables y_i.

This value can be interpreted in the following manner; if we have an R-square of 0.3, then we conclude that the variability of the explained values around the regression line is 1-0,3 times the original variance; this means that we have explained 30 per cent of our original variability.

$Time = f(Class)$

The purpose of this model is to analyze the impact of the action class on the execution time of EB^{3}PAI. An action belong to the class C_q if this action appears within the scope of q quantified interleaves. In the example below, the action a belong to the class C_2. An action of class C_0 means that this action is not in the scope of any quantified interleave.

$$main = |||\, x \in [1..10] \; : \; (|||\, y \in [11..20] : \mathsf{a}(x, y) \; : \;)$$

The linear model for this case is as follow:

$$T_i = \beta_0 + \beta_1 C_{j,i} + \epsilon_i \text{ with i=1,2,..,} n$$

with

- T_i is the execution time observed for the action i.
- $C_{j,i}$ is the class j of the action i.
- β_0 is the constant term, or the average value of T_i where $C_{j,i}$ is 0.
- β_1 is the coefficient of $C_{j,i}$.
- ϵ_i is the random error term on the action i.
- n is the total number of observations.

To calculate the linear regression on this model, we extract the necessary information from the database and then ask the R language to calculate all the coefficients. Since each test case is executed five times, we use the average of these five results in our calculations.

Figure 2.15 shows the regression line that explains the general approximative evolution of execution time per class of action, the observations used come from the non persistence execution of EB^{3}PAI. Instantiating the model with the calculated coefficients, we obtain the

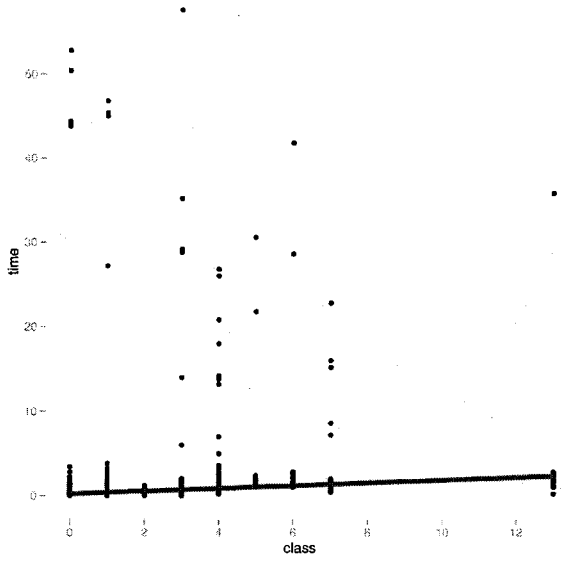

Figure 2.15: Regression line for time per class (NP-EB^{3}PAI)

following equation: $T_i = 0.21 + 0.16C_{j,i} + \epsilon_i$. In this case the coefficient of determination calculated is 0,58, this means that the regression line has covered up to 58 per cent of observations used.

The graph in Figure 2.16 shows the regression line calculated using observations of persistent executions of EB^{3}PAI. Replacing β_0 by 127,33 and β_1 by 7,77 in the model equation, will give us the regression line equation $T_i = 127,33 + 7,77C_{j,i} + \epsilon_i$ with 57 per cent of explained observations according to the R-square.

$Time = f(PathLength)$

The aim of this second model is to determine whether the size of a specification may cause a relatively high execution time. A file specification written in EB^3 is represented in EB^{3}PAI as an AST, this tree is subject to change during the execution. The variable *pathlength*, that we use, represents the number of nodes to visit before executing the desired action; in other words it is a variable describing the depth of the action to execute in the AST.

The model that we will describe is somewhat similar to the one before, instead of time

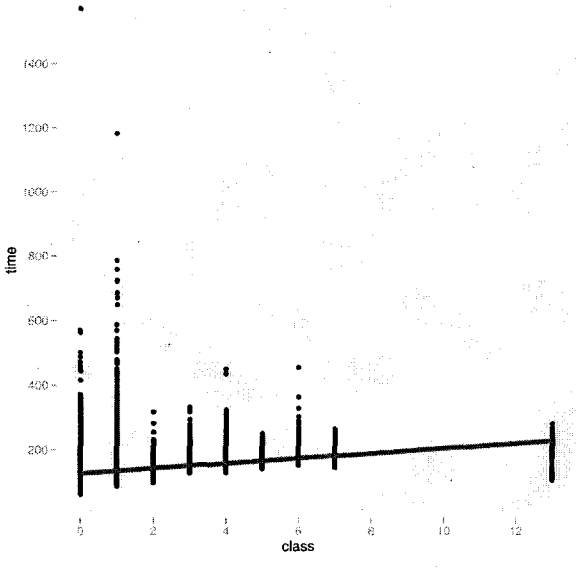

Figure 2.16: Regression line for time per class (P-EB^{3}PAI)

per class analysis it's time per path length analysis. this model is as follow:

$$T_i = \beta_0 + \beta_1 L_i + \epsilon_i \text{ with i=1,2,..,} n$$

with

- T_i is the execution time observed for the action i.
- L_i is the path length to the action i.
- β_0 is the constant term, or the average value of T_i where L_i is 0.
- β_1 is the coefficient of L_i.
- ϵ_i is the random error term on the action i.
- n is the total number of observations.

The equation $T_i = -0,0028 + 0,039 L_i + \epsilon_i$ represents the regression line showed in Figure 2.17. This result is calculated using observations of the non-persistent execution of

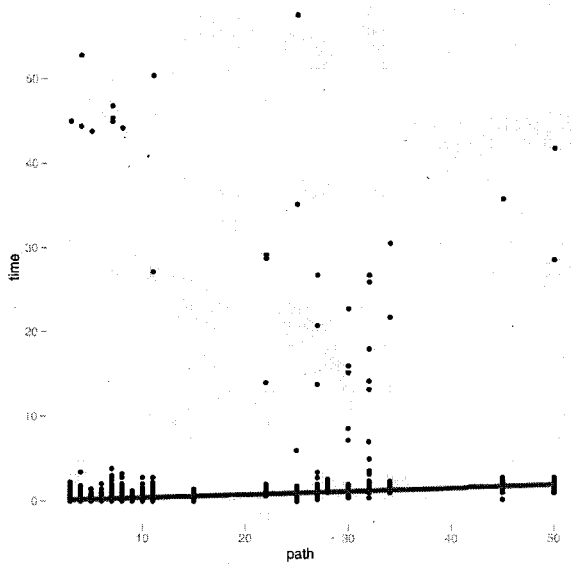

Figure 2.17: Regression line for time per path length (NP-EB^{3}PAI)

EB^{3}PAI, and explain more than 65 per cent of these observations. The value of β_1 $(0,039)$ is positive, and prove that the regression line grow slightly.

Figure 2.18 shows the regression line that represents the model equation after the coefficients resolution, the resulting equation is $T_i = 121,5+1,35L_i+\epsilon_i$ and explain more than 56 per cent observations of the persistent execution of EB^{3}PAI. According to the slope of the regression line of 1,35 (coefficient β_1), we can deduce that the evolution of the execution time is quite stable and not dramatic.

$Time = f(Class + PathLength)$

This third model describes the execution time using the class action to be performed and the path length to be discovered. This will allow us to understand the evolution of the execution time depending on the specification complexity parameters, the proposed model is as follows:

$$T_i = \beta_0 + \beta_1 C_{j,i} + \beta_2 L_i + \epsilon_i \text{ with i=1,2,..,n}$$

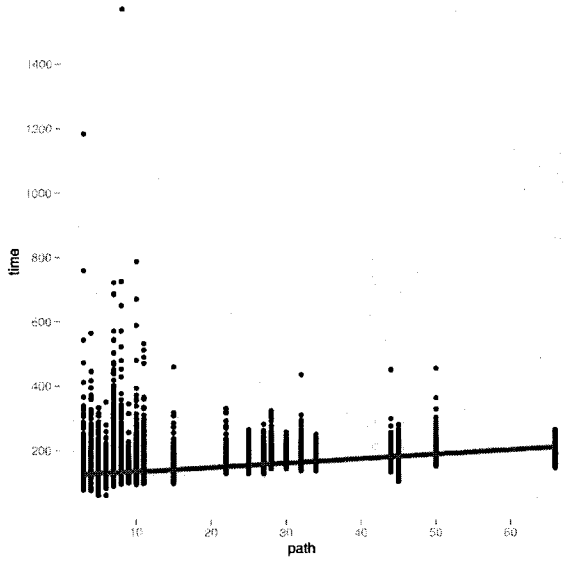

Figure 2.18: Regression line for time per path length (P-EB^{3}PAI)

with

- T_i is the execution time observed for the action i.
- $C_{j,i}$ is the class j of the action i.
- L_i is the path length to the action i.
- β_0 is the constant term, or the average value of T_i where L_i is 0.
- β_1 is the coefficient of $C_{j,i}$.
- β_2 is the coefficient of L_i.
- ϵ_i is the random error term on the action i.
- n is the total number of observations.

Graphics in Figure 2.19 and Figure 2.20 are 3D representations of time observations according to the path length and the action class. The three dimensional plane of each graph

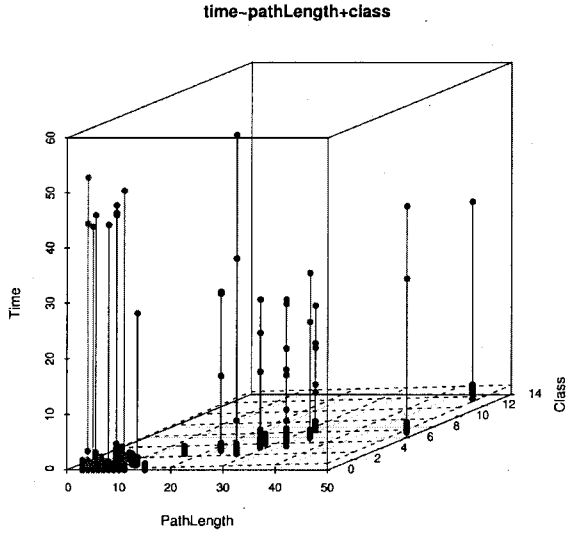

Figure 2.19: Regression plane for time per class and path length (NP-EB^{3}PAI)

represents the regression equation that have been calculated from the initial model. The first graph concerns observations gathered during the execution of non-persistent version of EB^{3}PAI, its regression equation is $T_i = 0.015 + 0.024C_{j,i} + 0.035L_i + \epsilon_i$ and explains up to 66 percent of the used observations. The second graph represents time execution of the interpreter using the persistence capability, its regression equation is $T_i = 123,6 + 4,57C_{j,i} + 0,67L_i + \epsilon_i$. This plane explains more than 59 per cent of the observations.

2.6.3 Overall time distribution

A global view on the distribution of data collected will identify the concentration of execution time. To perform such analysis, we will use on the concept of normal distribution (or Gaussian distribution). The normal distribution is used to describe data that cluster around a mean or an average, the associated graph is bell-shaped with a peak in the mean. To calculate the normal distribution, we need to extract, from the used observations, the mean and the standard deviation. A standard deviation is the variability or dispersion measure of a set of observations. In other words, the smaller standard deviation is, the closer are the

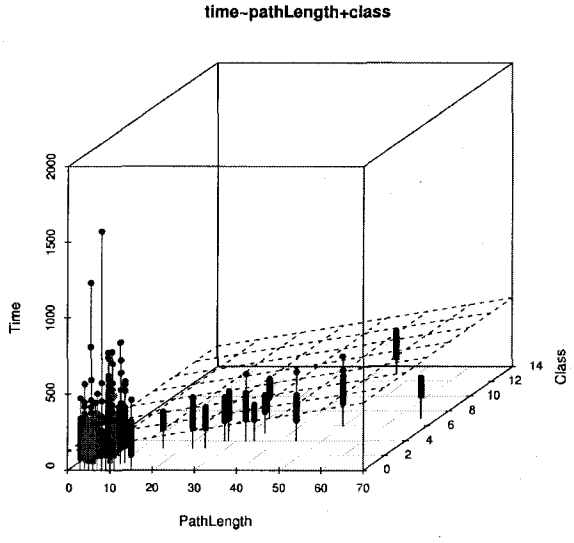

Figure 2.20: Regression plane for time per class and path length (P-EB^{3}PAI)

data from the mean, while a high standard deviation shows a large dispersion of data over a large range of values.

Figure 2.21 shows the time distribution after executing the test case 3-3 on the persistent version of EB^{3}PAI. We can notice that almost all the data gathered are between 100ms and 250ms, with as average value 167ms.

The histogram on Figure 2.22 shows the time distribution of the persistent version of EB^{3}PAI. We have also superimposed a curve representing the normal distribution according to the same data, the mean and the standard deviation are respectively 136,7 and 27,9. We have calculated also that 99,81 per cent of executions time gathered are between 50ms and 250ms.

2.6.4 Performance prediction

One of the objectives of EB^{3}PAI performance analysis is to try to predict the execution time of this interpreter in a given environment. After considering the linear model that characterize the performance of the interpreter and applying a regression analysis on each one of

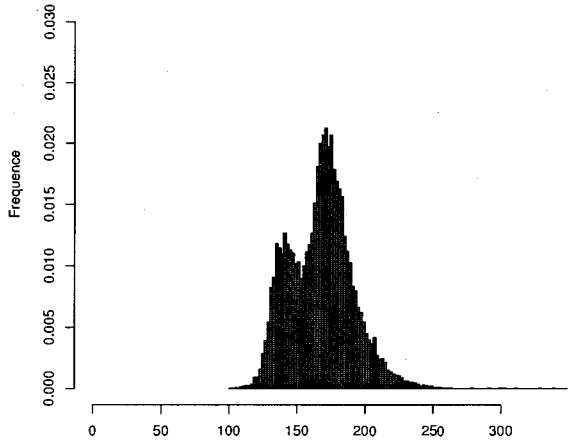

Figure 2.21: Time distribution on P-EB^{3}PAI: TC 3-3

these models, we can use the result of the regression to do an approximative estimation. The easiest way to calculate this prediction is by replacing the explicative factors in the resulting regression equations. For instance, if we consider an action e with 12 as path length and 3 as action class, we can then replace these values in the third model and predict the execution time that this action may take. For this case we will get T= 145,35ms. This way of prediction should be used to get an approximative idea only on the EB^{3}PAI performance, and can not under any circumstances be considered as the final real value.

Another way to predict the performance of EB^{3}PAI is to analyze the execution time by type of action. An action can have three different types, producer, consumer or modifier. A producer action belong generally to C_1, and allows the instantiation of entities (ex: *acquire* on Figure 2.7), a modifier action is most of the time an action of an association that links entities (ex: *lend*), and finally the consumer action is usually the last action to perform in an entity (ex: *return*). The purpose is to establish an approximative interval, for each action type, of time execution according to the observations gathered.

Figure 2.23 shows the average execution time by type of action, the data used are ob-

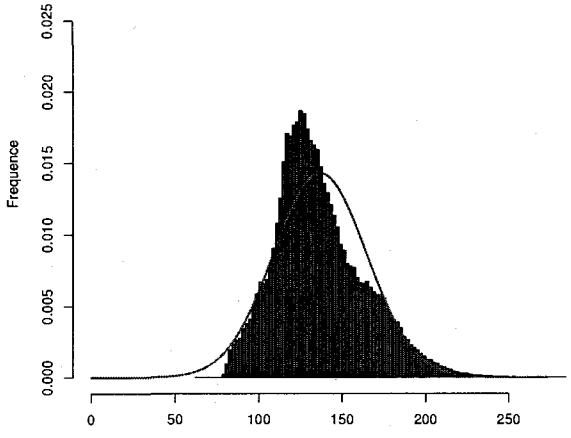

Figure 2.22: Overall time distribution on P-EB^{3}PAI

servations obtained during the testing process. According to this graph we can conclude that EB^{3}PAI takes in average 147ms to execute a producer action, 177ms to execute a modifier action and 163ms to execute a consumer action, with 30 as standard deviation. This prediction approach are less precise than the first one, but it gives a general idea on how EB^{3}PAI perform depending on the type of actions.

2.7 Conclusion

We have defined in this paper a methodology to analyze the performance of the EB^3 process algebra interpreter. We have used and adapted an IEEE standard to be able to provide a complete and organized documentation in order to analyze the testing results. Data analysis is based primarily on a statistical approach, allowing the study of the impact of many factors that may explain the final performance of the interpreter. This methodology can be used and adapted to any IS performance investigation, using concepts such as class of action or producer-consumer-modifier events.

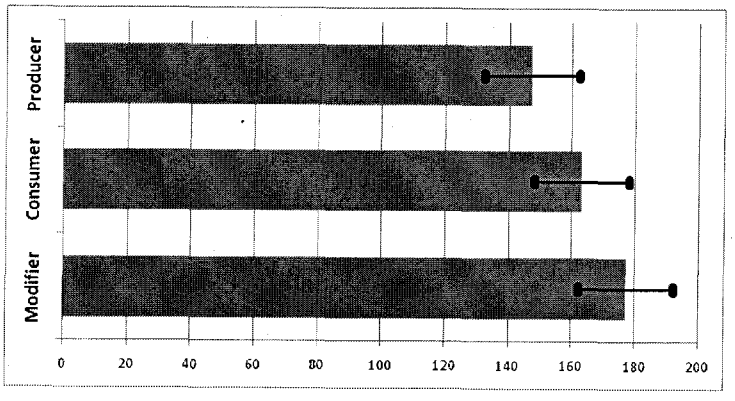

Figure 2.23: Performance prevision by action type

We have found in 2.6.1, using profiling sessions, an important performance bottlneck. The profiling process helps us to determin when and where these problem occurs. We've manage to resolve this problem, and improve considerably the performance of EB^{3}PAI. The hypothesis proposed at the development of EB^{3}PAI, is that the performance of this interpreter would be linearly related to the size of the specification. If we go back to the regression graph calculated for the three models described and look to the correspondent R-squares (greater than 50 per cent), we can deduce that our experiement confirms the hypothesis stated.

The performance of EB^{3}PAI is also largely dependent on the OO database used to store the AST of the specification. If we compare execution times on the non persistence version of EB^{3}PAI with the persistence one, we can see that the persistent version takes in average 120ms more that the non persistent one. This can be considered normal given the time required to access the hard drive when creating and modifying the AST using *ObjectStore*, versus the time to do the same thing in main memory.

EB^{3}PAI uses hash-tables to store the object representation of the specification, this hash-table is constantly resized during execution. A way to improve the performance of the interpreter is to reduce the number of resizing calls, and to optimize the AST representation.

What would be interesting as future work is to automate the performance testing process to generate different test cases according to their respective test objectives. This automated process could also test some other components of the APIS project as needed and finally

store all gathered data as described in this paper.

Conclusion

Contribution

Nous avons présenté une approche pratique permettant d'analyser les performances d'EB^{3}PAI, en définissant et documentant les plans de tests jugés pertinents pour le processus de test. Nous avons également procédé à la génération des données de tests à partir des cas de test définis, ainsi que l'exécution et la récolte des informations sur le temps d'exécution de l'interpréteur. Ces informations ont, par la suite, été organisées et mémorisées dans une base de données dans le but d'appliquer une analyse statistique adéquate. Nous avons proposé trois analyses statistiques permettant d'expliquer les temps d'exécution, ainsi que de prédire de manière approximative les performances d'EB^{3}PAI selon les différents facteurs explicatifs. Ce travail est généralisable à la plupart des SI, dans la mesure où leurs performances peuvent être linéairement expliquées, puisque les concepts modélisés dans EB^3, tels que les classes d'actions ou producteur-modificateur-consomateur, se retrouvent dans tous les SI. Seule la longueur du chemin parcouru dans la spécification est propre à EB^3.

Résultats obtenus

À l'aide de l'outil de profilage, on a pu détecter au début du processus de test une perte de performance majeure. Nous avons pu identifier la source du problème et corriger efficacement des erreurs d'implémentation. Ceci nous a permis, par la suite, d'obtenir des performances stables qui répondaient à nos attentes.

D'après l'analyse des régressions appliquées aux données récoltées, on peut confirmer

de manière statistique l'hypothèse avancée lors du développement d'EB^{3}PAI. Cette hypothèse disait que la performance d'EB^{3}PAI est linéairement liée à la taille de la spécification utilisée. Ceci est vérifié grâce aux droites de régressions calculées. Le profilage nous a permis de déceler un problème de performance important

Toutefois, les performances d'EB^{3}PAI dépendent du SGBDOO utilisé pour mémoriser l'arbre syntaxique abstrait. En comparant les temps d'exécutions de la version sans persistance avec ceux de la version avec persistance, nous avons pu constater que cette dernière prend en moyenne 120 ms en plus. Ceci peut être expliqué notamment par le temps d'accès au disque dur nécessaire pour créer et modifier l'état du système en utilisant *ObjectStore*, ainsi que l'ensemble des opérations utilisées par ce dernier. L'étude des bogues rencontrés, lors du processus de test, a permis de mettre en évidence l'importance du bon paramétrage des tables de hachage (en anglais *hashtable*). Ces tables doivent être configurées afin d'éviter un débordement de mémoire (en anglais *memory overflow*).

Le processus de test, décrit dans ce travail, a permis de récolter des informations pertinentes sur les performances réelles d'EB^{3}PAI. Si les résultats sont très satisfaisants, les données obtenues montrent que les exécutions sont soumises à certains facteurs aléatoires que nous n'avons pas pu isoler. Le fait d'exécuter chaque test cinq fois et calculer la moyenne ne fait que réduire ces facteurs aléatoires qui, parfois, provoquent des piques d'exécutions. En analysant le comportement du ramasse-miettes, l'hypothèse que celui-ci provoque ces piques est écartée. Nous en concluons que ces facteurs sont liés au système d'exploitation et peuvent se manifester de la même façon dans un environnement d'exécution final. Finalement, nous avons établit une fois pour toute que l'objectif du projet APIS est possible, à savoir la génération automatique d'un SI qui possède des performances similaires à celui développé selon des méthodes conventionnelles.

Portée

Les méthodes formelles peuvent désormais être considérées comme une alternative viable au développement des SI, tout en offrant des performances similaires à celles des SI conventionnels. De plus, en se basant sur les données d'exécution avec *PSE Pro*, nous avons pu constater que les BDOO offrent des résultats acceptables en terme de performances pour des SI.

Travaux futurs de recherche

Une extension directe de l'approche proposée serait l'automatisation du processus de test. Ceci permettra d'assurer une couverture de tests de performance du système ou il serait aussi envisageable, par une étude complémentaire, de fournir des tests fonctionnels et aussi des tests de charge. Les tests de performance et de charge seraient appliqués aisément lors de la mise en site, ce qui ferait gagner temps et argent. Par le biais de l'interpréteur, l'oracle permettant l'analyse des résultats peut aussi être généré de façon *ad hoc*. Il serait intéressant de tester l'interpréteur sur d'autres environnements, afin d'expliquer plus en détail l'impact des systèmes d'exploitation. Finalement, afin d'optimiser les performances d'EB^{3}PAI, un autre travail pourrait se porter sur le paramètrage des tables de hachage selon la taille de la spécification utilisée. Ce paramétrage devrait pouvoir lui aussi être effectué de manière complètement automatique par le système, uniquement par la lecture de la spécification, et notamment des ensembles de quantification, c'est-à-dire du nombre d'entités et de la cardinalités des relations impliqués dans le SI.

Annexe A

Organisation des données et exemples de scripts PL/R

A.1 Base de données des résultats

Les résultats d'analyse de performance sont mémorisés sur une base de données *Postgresql*, la description des tables est comme suit :

- `TestPlan` : cette table décrit l'objectif global, les fonctionnalités à tester et l'approche utilisée de chacun des quatres plans de test utilisés.
- `TestCase` : cette table contient les cas de test utilisés pour chaque plan de test, ces cas de test sont les fichiers de spécifications utilisés dans les tests de performance.
- `TestInput` : cette table décrit les schémas (en anglais *pattern*) des actions à passer en entrée lors de l'exécution d'EB^{3}PAI. Plusieurs schémas peuvent être utilisés pour un seul cas de test.
- `TestInputLine` : cette table contient les informations pertinentes (classe d'action, longueur du chemin) concernant chaque action passée en entrée lors de l'exécution d'EB^{3}PAI.
- `TestEnvironment` : cette table contient les informations concernant les environnements d'exécution sur lesquels les tests ont pu être exécutés.
- `TestRun` : cette table contient les informations concernant l'exécution de chaque test.

- `TestRunOutPutLine` : cette table contient les résultats d'exécution en milliseconde de chaque action exécutée par EB^{3}PAI.

A.2 Le langage R

Le langage R est utilisé pour pouvoir effectuer des analyses statistiques sur les données mémorisées dans la base de données.

A.2.1 Introduction

Le langage R est un langage de programmation ainsi qu'un environnement mathématique utilisé pour le traitement de données et l'analyse statistique. Le langage R offre aussi des outils puissants permettant la génération de différents types de graphiques. Ces graphiques peuvent soit être affichés directement sur l'écran ou enregistrés sous un des formats supportées (comme par exemple pdf, ps ou jpeg).

A.2.2 Exemple

L'exemple suivant illustre le fonctionnement de R, et plus spécifiquement, l'utilisation de la régression linéaire avec cet outil. Cet exemple indique l'utilisation de la fonction *lm( )* (pour *linear model*) pour analyser la relation entre le temps d'exécution d'une action et la longueur du chemin parcouru par EB^{3}PAI. La fonction `lm( )` permet de déclarer un modèle linéaire ainsi que de calculer sa régression.

```
#### ---Creation du tableau de donnees contenant deux colonne time et pathLength---###

donnee<-data.frame(time=c(125,203,143,500,460,96), pathLength=c(12,56,4,3,6,20));

### ----Declaration du model (time en fonction du pathLength) a l'aide de la fonction lm()-----###

reg<-lm(time~pathLength, data=donnee);

### ------- Affichage des resultats de regression-----###

summary(reg);

### ------- Generation des graphiques-----------------###
```

```
plot(req,witch=1:4);
```

A.3 Le langage de procédure PL/R

Pour pouvoir utiliser les données mémorisées sur la base de données pour faire des analyses de statistiques à l'aide de R. On aura besoin du langage de procédure PL/R qui est spécialement conçu pour à cette fin.

A.3.1 Introduction

Le langage PL/R est un langage de procédure qui permet de créer des fonctions *Postgres* ayant la capacité de déclencher des fonctions du langage R. La procédure d'installation est présentée ici[1].

A.3.2 Syntaxe

La syntaxe de création de fonctions sous PL/R ressemble à ceci :

```
CREATE OR REPLACE FUNCTION funcname (argument-types)
RETURNS return-type AS '
    function body
' LANGUAGE 'plr';
```

Ceci est un exemple simple de fonction qui retourne le maximum des deux arguments passés en entrée :

```
CREATE OR REPLACE FUNCTION r_max (integer, integer) RETURNS integer AS '
    if (arg1 > arg2)
       return(arg1)
    else
       return(arg2)
' LANGUAGE 'plr';
```

[1] http://www.joeconway.com/plr/doc/plr-install.html

A.3.3 Exemple

On va supposer que la base de données utilisée contient la table `Execution`, cette table a trois champs (`execId`, `execTime`, `execPathLength`).
La fonction `pg.spi.exec(query)` permet d'exécuter la requête `query` et de retourner le résultat sous forme d'un tableau, celui-ci peut être utilisé par la suite pour faire une analyse. L'exemple suivant reprend le même exemple d'analyse décrit dans la section le langage R, en utilisant PL/R. Le code est comme suit :

```
CREATE OR REPLACE FUNCTION reg_exemple()
  RETURNS text AS '
#### --Recupration des donnees-----------------------###

donnee <<- pg.spi.exec ('select  "execTime" as time,
   "execPathLength" as pathLength from "Execution"');

#### --declaration du model----------------------------###

reg<-lm(time~pathLength,data=donnee);

#### --Creation du fichier de sortie pour le graphe---###

png(''/tmp/Plot.png'');

#### --Ecrire dans le fichier---###

plot(reg,which=1);

#### --Fermer le fichier du graphe--------------------###

dev.off();

#### --Affichage d'un message-------------------------###

print('done');

' LANGUAGE 'plr'
```

A.4 Calcul de régression

Afin de calculer la régression sur les données récoltées, il nous faudra récupérer celles-ci dans la base de données et les utiliser dans R pour faire les calculs. Cette section explique

pour chaque modèle décrit, comment sa régression linéaire est calclulée, ainsi que comment générer les graphique correspondants.

A.4.1 Temps par classe d'action

Requête SQL

Ce premier modèle essait d'expliquer le temps d'exécution par rapport à la classe d'action utilisée. Les temps d'exécutions sont extraits à partir de la BD à l'aide de la requête suivante :

```
select  "classe" as class, avgExecutionDuration as time
from "TestInputLine",
          (select "testInputLineId", AVG("executionDuration") as avgExecutionDuration
        from "TestRunOutputLine" group by "testInputLineId") as avgExec
where "TestInputLine"."testInputLineId"=avgExec."testInputLineId
```

Puisque chaque action est exécutée cinq fois pour réduire les facteurs aléatoires, le `select` imbriqué sert à calculer la moyenne des temps d'exécutions pour chaque action. Le résultat de cette requête est un tableau contenant la moyenne de temps d'exécution pour chaque action et la classe d'action correspondante.

Code PL/R

Le code PL/R est la procédure utilisée à fin de calculer et générer le graphe de régression pour le modèle temps par action :

```
CREATE OR REPLACE FUNCTION f_graph()
  RETURNS text AS
'
##---Requete de recuperation de donnees--##

donnee <<- pg.spi.exec
(''select  "classe" as class, avgExecutionDuration as time
        from "TestInputLine",
           (select "testInputLineId", AVG("executionDuration") as avgExecutionDuration
        from "TestRunOutputLine" group by "testInputLineId") as avgExec
        where "TestInputLine"."testInputLineId"=avgExec."testInputLineId
 '');
```

A.4. CALCUL DE RÉGRESSION

```
##---Creation du fichier image de sortie--##
png(''/tmp/cDataPlot_CT1.png'');
##---Dessin du graphique des donnees---##
plot(donnee);
##---Calcule de la ligne de regression---##
reg<-lm(time~class,data=donnee)
myline.fit<-reg;
##---Dessin de la ligne de regression---##
abline(myline.fit,col="red");
##---Fermeture du fichier---##
dev.off();
##---Resultat des calcul sur fichier text---##
sink(file="/tmp/cpRegSummary.txt");
print(summary.lm(reg), max.levels = NULL);
sink()

'
LANGUAGE plr;

select f_graph();
```

Résultat

Le graphe dans la figure 2.16 représente le résultat du calcul de régression. Le résultat du calcul des coefficients est comme suit :

```
Call:
lm(formula = time ~ class, data = donnee)

Residuals:
     Min       1Q   Median       3Q      Max
-123.259  -12.831   -2.031   11.141 1443.369

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 127.23072    0.03222  3948.3   <2e-16 ***
class         7.77143    0.01343   578.5   <2e-16 ***
---
Signif. codes:  0 â***â 0.001 â**â 0.01 â*â 0.05 â.â 0.1 â â 1

Residual standard error: 22.89 on 679998 degrees of freedom
Multiple R-squared: 0.3298,Adjusted R-squared: 0.3298
F-statistic: 3.346e+05 on 1 and 679998 DF,  p-value: < 2.2e-16
```

A.4.2 Temps par longueur du chemin

Requête SQL

Ce deuxième modèle explique le temps d'exécution par rapport à la longueur du chemin nécessaire pour atteindre l'action utilisée. Les temps d'exécution sont extraits à partir de la BD à l'aide de la requête suivante :

```
select  "pathLength" as path, avgExecutionDuration as time
from "TestInputLine",
       (select "testInputLineId", AVG("executionDuration") as avgExecutionDuration
   from "TestRunOutputLine" group by "testInputLineId") as avgExec
where "TestInputLine"."testInputLineId"=avgExec."testInputLineId"
```

Puisque chaque action est exécutée cinq fois pour réduire les facteurs aléatoires, le `select` imbriqué sert à calculer la moyenne des temps d'exécution pour chaque action. Le résultat de cette requête est un tableau contenant la moyenne de temps d'exécution pour chaque action et la longueur du chemin utilisé correspondant.

Code PL/R

Le code PL/R est la procédure utilisée afin de calculer et générer le graphe de régression pour le modèle temps par action :

```
CREATE OR REPLACE FUNCTION f_graph()
  RETURNS text AS
'
##---Requete de recuperation de donnees--##

donnee <<- pg.spi.exec
(''select  "pathLength" as path, avgExecutionDuration as time
   from "TestInputLine",
       (select "testInputLineId", AVG("executionDuration") as avgExecutionDuration
   from "TestRunOutputLine" group by "testInputLineId") as avgExec
   where "TestInputLine"."testInputLineId"=avgExec."testInputLineId"
 '');

##---Creation du fichier image de sortie--##
png(''/tmp/pDataPlot_reg.png'');
##---Dessin du graphique des donnees---##
plot(donnee);
##---Calcule de la ligne de regression---##
reg<-lm(time~path,data=donnee)
```

```
myline.fit<-reg;
##---Dessin de la ligne de regression---##
abline(myline.fit,col="red");
##---Fermeture du fichier---##
dev.off();
##---Resultat des calcul sur fichier text---##
sink(file="/tmp/pRegSummary.txt");
print(summary.lm(reg), max.levels = NULL);
sink()

'
LANGUAGE plr;

select f_graph();
```

Résultat

Le graphe dans la figure 2.18 représente le résultat du calcul de régression. Le résultat du calcul des coefficients est comme suit :

```
Call:
lm(formula = time ~ path, data = donnee)

Residuals:
     Min       1Q   Median       3Q      Max
 -77.641  -14.782   -1.582   12.218 1438.259

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 1.215e+02  3.884e-02    3127   <2e-16 ***
path        1.359e+00  2.406e-03     565   <2e-16 ***
---
Signif. codes:  0 â***â 0.001 â**â 0.01 â*â 0.05 â.â 0.1 â â 1

Residual standard error: 23.07 on 679998 degrees of freedom
Multiple R-squared: 0.3195,Adjusted R-squared: 0.3195
F-statistic: 3.193e+05 on 1 and 679998 DF,  p-value: < 2.2e-16
```

A.4.3 Temps par classe et par longueur du chemin

Requête SQL

Ce troisième modèle explique le temps d'exécution par rapport à la classe d'action utilisée et la longueur du chemin emprunté. Les temps d'exécution sont extraits à partir de

la BD à l'aide de la requête suivante :

```
select  "pathLength" as path, "classe" as class,  avgExecutionDuration as time
from "TestInputLine",
    (select "testInputLineId", AVG("executionDuration") as avgExecutionDuration
from "TestRunOutputLine" , "TestRun"
where "TestRunOutputLine"."testRunId"="TestRun"."testRunId" and "TestRun"."testEnvironementId"=1
```

Comme précédemment, le `select` imbriqué sert à calculer la moyenne des temps d'exécution pour chaque action. Le résultat de cette requête est un tableau contenant la moyenne de temps d'exécution pour chaque action, la classe d'action correspondante et la longueur du chemin utilisé.

Code PL/R

Le code PL/R est la procédure utilisée à fin de calculer et générer le graphe de régression pour le modèle temps par classe d'action et chemin utilisé :

```
CREATE OR REPLACE FUNCTION f_graph()
  RETURNS text AS
'
##---Requete de recuperation de donnees--##
donnee <<- pg.spi.exec (''select  "pathLength" as path, "classe" as class,  avgExecutionDuration as time
from "TestInputLine",
(select "testInputLineId", AVG("executionDuration") as avgExecutionDuration
from "TestRunOutputLine" , "TestRun"
where "TestRunOutputLine"."testRunId"="TestRun"."testRunId"
and "TestRun"."testEnvironementId"=1
group by "testInputLineId")
        as avgExec
where "TestInputLine"."testInputLineId"=avgExec."testInputLineId"
'');

##---Appel a la bibliotheque 3D--##
require(scatterplot3d);
##---CrÃ©ation du fichier image de sortie--##
png(''/tmp/cpDataPlot_reg_3D.png'');
##---Dessin du graphique des donnÃ©es en 3D---##
s3d <- scatterplot3d(donnee, type="h", highlight.3d=TRUE, angle=55, scale.y=0.7, pch=16, main="scatterplot3d
##---Calcule de la ligne de regression---##
reg<- lm(time ~ class + path, data=donnee)
my.lm <- reg
##---Dessin de la ligne de regression---##
s3d$$$plane3d(my.lm)
```

```
##---Fermeture du fichier---##
dev.off();
##---Resultat des calcul sur fichier text---##
sink(file="/tmp/cpRegSummary.txt");
print(summary.lm(reg), max.levels = NULL);
sink()
'
LANGUAGE plr;

select f_graph();
```

Résultat

Le graphe de la figure 2.20 représente le résultat du calcul de régression. Le résultat du calcul des coefficients est comme suit :

```
Call:
lm(formula = time ~ class + path, data = donnee)

Residuals:
     Min       1Q   Median       3Q      Max
-108.324  -13.317   -2.043   11.023 1441.630

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 1.236e+02  3.958e-02  3122.2   <2e-16 ***
class       4.573e+00  2.466e-02   185.5   <2e-16 ***
path        6.732e-01  4.382e-03   153.6   <2e-16 ***
---
Signif. codes:  0 â***â 0.001 â**â 0.01 â*â 0.05 â.â 0.1 â â 1

Residual standard error: 22.51 on 679997 degrees of freedom
Multiple R-squared: 0.3523,Adjusted R-squared: 0.3523
F-statistic: 1.849e+05 on 2 and 679997 DF,  p-value: < 2.2e-16
```

Bibliographie

[1] Jan A. BERGSTRA et Jan W. KLOP. « Process algebra for synchronous communication ». Information and Control, 60(1-3) :109–137, 1984.

[2] Giovanni DENARO, Andrea POLINI et Wolfgang EMMERICH. « Early Performance Testing of Distributed Software Applications ». Dans Proceedings of the 4th International Workshop on Software and Performance (WOSP 2004), Redwood Shores, California, USA, 2004.

[3] Benoît FRAIKIN. « Interprétation efficace d'expression de processus EB^3 ». Thèse de doctorat, Département d'informatique, Université de Sherbrooke, Sherbrooke, Québec, Canada, avril 2006.

[4] Benoît FRAIKIN et Marc FRAPPIER. « Efficient Symbolic Computation of Process Expressions ». Science of Computer Programming, 74(9) :723–753, juillet 2009.

[5] Marc FRAPPIER, Benoît FRAIKIN, Frédéric GERVAIS, Régine LALEAU et Mario RICHARD. « Synthesizing Information Systems : the APIS Project ». Dans Colette ROLLAND, Oscar PASTOR et Jean-Louis CAVARERO, éditeurs, Proceedings of the First International Conference on Research Challenges in Information Science, RCIS 2007, pages 73–84, Ouarzazate, Morocco, avril 2007.

[6] Xiang GAN. « Software Performance Testing ». Rapport Technique, Department of Computer Science, University of Helsinki, Helsinki, Finlande, 2006, `http://www.cs.helsinki.fi/u/paakki/gan.pdf`.
Seminar Paper.

[7] Frédéric GERVAIS, Panawe BATANADO, Marc FRAPPIER et Régine LALEAU. « Génération automatique de transactions de base de données relationnelle à partir de définitions d'attributs EB3. ». Dans rapport S002 ENST PARIS, éditeur, Approches

Formelles dans l'Assistance au Développement de Logiciels, AFADL'06, pages 25–39, Paris, France, Mars 2006.

[8] Steven HAINES. « Performance Testing Methodology ». Rapport Technique, Quest Software, 2005, http://www.qanc.co.kr/4research_0402_download.htm?data_no=156&name=qperformance_testing_methodology.pdf.

[9] Robin MILNER. Communication and concurrency. Prentice-Hall, Inc., 1989.

[10] Software Engineering Technical Committee of the IEEE COMPUTER SOCIETY. « IEEE 829-1998 Standard for Software Test Documentation ». 1998. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=741968&isnumber=16010.

[11] Richard ST-DENIS et Marc FRAPPIER. « EB3 : an entity-based black-box specification method for information systems ». Software and Systems Modeling, 2(2) :134–149, juillet 2003.

[12] Alan O. SYKES. « An Introduction to Regression Analysis ». Rapport Technique, Law School, University of Chicago, 1999, http://www.law.uchicago.edu/files/files/20.Sykes_.Regression_0.pdf.

[13] Thomas D. WICKENS. « The General Linear Model ». Rapport Technique, University of California, Los Angeles, juillet 2004, http://www.ipam.ucla.edu/publications/mbi2004/mbi2004_5017.pdf.
Graduate Summer School Program.