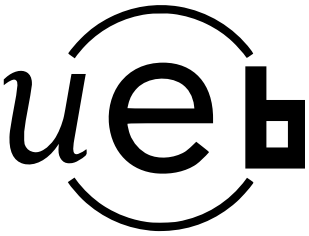
Supporting resource-awareness in managed runtime environments

Inti Gonzalez-Herrera

ANNÉE 2015



THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale Matisse

présentée par

Inti Yulien GONZALEZ HERRERA

préparée à l'unité de recherche INRIA
Rennes Bretagne Atlantique

Supporting resource awareness in managed runtime environments.

Thèse soutenue à Rennes
le 14 Décembre 2015

devant le jury composé de :

Vivien QUÉMA
LIG laboratory – Grenoble INP/ENSIMAG / *Rapporteur*

Gilles MULLER
Whisper team – LIP6 / *Rapporteur*

Gaël THOMAS
Samovar Laboratory – Telecom SudParis / *Examinateur*

Laurent RÉVEILLÈRE
LaBRI – Université de Bordeaux / *Examinateur*

Isabelle PUAUT
IRISA – Université de Rennes 1 / *Examinatrice*

Olivier BARAIS
Professor, IRISA – Université de Rennes 1 / *Directeur de thèse*

Johann BOURCIER
Maître de conférences, IRISA – Université de Rennes 1 / *Co-directeur de thèse*

Hay una historia familiar que me impactó muchísimo. Contaba mi tío Lore que una vez había intentado abandonar la escuela. Mi abuelo Lolo le autorizó, con una sola condición – mi tio debía comenzar a trabajar, pues un vago no podía ser.

Asi fue como por una semana mi tío hizo de campesino, unas once horas seguidas bajo el sol. A los pocos días no pudo más, fue y le dijo a mi abuelo – "así no puedo seguir". Lolo, tranquilito como siempre, dijo – "a mi me parece que si regresas a la escuela de seguro te aceptan". A Lore la alegría no le cupo en el cuerpo y, aunque siempre tuvo la sospecha, le tomó unos 30 años convencerse de que mi abuelo no trabajaba tanto.

A mis padres va dedicado este trabajo, pues son ellos los que – siempre usando sus modos retorcidos – me han convertido en alguien capaz de apreciar esa historia.

Acknowledgments

I would firstly like to thank the members of my jury. My examiners, Vivien Quéma and Gilles Muller were generous with their comments on my thesis, and I value their assessment and input to its improvement. Thanks are also due to Isabelle Puaut for her feedback and for agreeing to preside over my jury. I would also like to thank Gaël Thomas and Laurent Réveillère for attending as members of the jury. I was grateful for the insight of their feedback and questions.

I would also like to thank the members of the DiverSE Team. The opportunity of meeting researches who come from amazingly different places is what I appreciate the most about the last three years. It still shocks me how we share a vision despite of our differences. I would love to believe that listening all the scientific discussions has positively transformed who I am.

Finally, special thanks are due to my advisers, Olivier Barais and Johann Bourcier. First, for their useful advices, and second for offering me the opportunity of working with them; it has been – for many good reasons – an interesting and wonderful experience.

Résumé

Aujourd'hui, les systèmes logiciels sont omniprésents. Ils se trouvent dans des environnements allant des contrôleurs pour appareils ménagers à des outils complexes pour traiter les processus industriels. Les attentes de l'utilisateur final ont grandi avec le développement de l'industrie, du matériel et des logiciels. Cependant, l'industrie doit faire face à plusieurs défis pour contenter ces attentes. Parmi eux, nous trouvons des problèmes liés à la question générale de traiter efficacement les ressources informatiques pour satisfaire aux exigences non fonctionnelles. En effet, parfois, les applications doivent fonctionner sur des dispositifs à ressources limitées ou des environnements d'exécution ouverts où la gestion efficace des ressources est d'une importance primordiale pour garantir la bonne exécution des demandes. Par exemple, les appareils mobiles et les passerelles domestiques intelligentes sont des dispositifs à ressources limitées où les utilisateurs peuvent installer des applications provenant de sources différentes. Dans le cas des passerelles domestiques intelligentes, Éviter toute mauvaise conduite dans les applications est important parce que ces dispositifs contrôlent souvent un environnement physique occupé par des personnes.

Pour satisfaire à ces exigences, l'objectif est de rendre les applications et les environnements d'exécution conscient et capable de faire face à des ressources limitées. Ceci est important parce que souvent ces exigences émergent ensemble (par exemple, les smartphones sont des appareils à ressources limitées fournissant un environnement d'exécution ouvert). Quand une application inclut des fonctionnalités pour réagir et modifier son comportement suite à l'apparition d'événements liés aux ressources, on dit que l'application est «consciente des ressources». Un système logiciel nécessite un environnement d'exécution approprié pour fournir de telles caractéristiques. Aujourd'hui, les applications s'exécutant sur des environnements d'exécution gérés (MRTEs), tel que Java, font partis des systèmes qui peuvent bénéficier de cette «conscience des ressources». En effet, les MRTEs sont régulièrement utilisés pour mettre en œuvre les intergiciels, par exemple en utilisant OSGi, en raison de leur sécurité, flexibilité, et de la maturité de l'environnement de développement. Ces intergicield fournissent souvent des fonctionnalités de monde ouvert, telles que la possibilité d'ajouter de nouvelles fonctionnalités après le déploiement initial du système. Pour soutenir la capacité d'adaptation et de gestion demandée par une exécution dans un monde ouvert, il est possible d'utiliser des techniques de génie logiciel à base de composants (CBSE). Hélas, certains MRTEs, tels que Java, ont été conçus pour exécuter une seule application à la fois, de sorte qu'ils manquent d'outils pour la gestion des ressources à grain fin.

Cette thèse aborde le problème de la programmation pour créer des sys-

tèmes «conscient des ressources» supporté par des environnements d'exécution adaptés. En particulier, cette thèse vise à offrir un soutien efficace pour recueillir des données sur la consommation de ressources de calcul (par exemple, CPU, mémoire), ainsi que des mécanismes efficaces pour réserver des ressources pour des applications spécifiques. Malheureusement, les mécanismes actuels nécessaires pour permettre la programmation de ressources dépendent fortement de la technologie cible. Par exemple, réserver de la mémoire pour un processus natif Unix est différent de réserver de la mémoire pour un bundle OSGi (le premier problème consiste à créer un espace d'adressage virtuel tandis que le deuxième requiert l'utilisation conjointe d'un allocateur de mémoire et un ramasse-miettes spécifiques). En conséquence, les solutions que nous discutons dans nos recherches sont principalement ciblées sur la gestion des ressources dans le cadre des MRTEs. En particulier, nous nous concentrons sur ce genre d'environnement d'exécution lorsque nous présentons les contributions de cette thèse.

Défis

Dans les solutions existantes qui permettent de surveiller la consommation des ressources et de réserver des ressources dans les MRTEs, nous trouvons deux inconvénients importants. La lutte contre ces inconvénients, qui sont décrits ci-dessous, est l'objectif de cette thèse.

Les solutions pour la surveillance de la consommation des ressources et leur réservation imposent un impact important sur les performances à l'exécution des applications. En particulier, les mécanismes basés sur l'instrumentation, qui offrent une bonne précision, réduisent de manière significative la performance des applications. Bien que cette restriction n'impacte pas les mesures des ressources consommées, il empêche leur utilisation dans un environnement de production. En conséquence, les ingénieurs sont obligés de choisir entre deux solutions non satisfaisantes - soit des performances réduites avec une bonne précision ou des performances acceptable avec une faible précision - lorsque les applications nécessitent d'être conscientes de la consommation et de la réservation des ressources. Malgré l'utilisation répandue des MRTEs pour exécuter des applications basées sur les composants et autres abstractions, la création d'outils permettant de gérer finement les ressources pour ces abstractions est encore une tâche complexe. En effet, la création d'abstractions, comme des modèles de composants, est de plus en plus commune. Beaucoup d'outillage existe pour le faire, en particulier pour définir de nouveaux langages dédiés. En outre, bien souvent, ces abstractions ciblent les MRTEs comme technologies permettant l'exécution en raison de leur sécurité et de la maturité des environnements de développement. Cependant, ces nouvelles abstractions posent un défi aux développeurs quand ils s'intéressent à la surveillance des ressources parce que ces nouvelles abstractions ne sont pas toujours offertes avec des mécanismes de surveillance des ressources ainsi que des débogueurs personnalisés. En conséquence, les développeurs utilisent des outils traditionnels qui peuvent seulement faire face aux concepts classiques tels que des objets, des méthodes et des emplacements de mémoire, au lieu des concepts plus spécifiques. La raison pour cela est que la définition d'un outillage pour une abstraction spécifique est une tâche ardue qui doit être mise en balance

avec le public limité d'une telle abstraction.

Les défis de cette recherche peuvent être résumés dans les questions de recherche suivantes. Ces questions se posent à partir de l'analyse des inconvénients des outils actuels dans les paragraphes précédents. Il est utile de rappeler que ces questions se rapportent aux MRTEs.

QR1. Comment pouvons-nous fournir un soutien portable et efficace pour la surveillance de la consommation de ressources ?

QR2. Comment pouvons-nous choisir les mécanismes a utilisés pour garantir la réservation de ressources tout en maintenant un surcout d'exécution faible pour chaque composant logiciel ?

QR3. Comment pouvons-nous tirer profit de la connaissance de l'architecture des applications pour aider un mécanisme de gestion des ressources ?

QR4. Comment pouvons-nous faciliter la définition et la mise en place d'outils de surveillance pour de nouvelles abstractions de logiciels ?

Contributions

Les résultats de cette thèse forment trois contributions qui visent à réduire (1) le coût de calcul pour effectuer la gestion des ressources, et (2) la complexité de la création d'outils de contrôle des ressources. Deux d'entre elles ciblent exclusivement le problème de la réduction du coût de calcul pour effectuer la gestion des ressources tandis que la troisième vise également le problème de faciliter la construction d'outils de suivi de l'utilisation des ressources. Ces contributions sont brièvement décrites ci-dessous.

Contribution : un cadre de surveillance des ressources optimiste qui réduit le coût de la collecte des données de consommation de ressources. Le suivi de la consommation des ressources est le fondement de la programmation pour les systèmes conscients de leurs ressources. Dans cette recherche, une nouvelle approche construite sur l'idée d'un contrôle adaptatif est présenté. L'approche, à savoir Scapegoat, est fondée sur quatre principes : i) souvent des applications sont construites en utilisant des abstractions telles que les composants que nous pouvons utiliser pour identifier et isoler la consommation des ressources, ii) lorsque l'environnement d'exécution est en cours d'exécution sur la ressource, nous pouvons utiliser la surveillance légère optimiste et toujours être sûr que nous serons en mesure de détecter les défaillances potentielles dans le temps, iii) il est possible d'identifier rapidement le composant défectueux une fois qu'un échec potentiel est repéré, et iv) il existe des mécanismes de contrôle que nous pouvons réutiliser parce qu'ils sont échangeables à l'exécution et offrent différents compromis entre le surcoût d'exécution et la précision. Scapegoat a été mis en œuvre et évalué dans ce travail et les résultats montrent la faisabilité et l'efficacité. Cette contribution répond aux questions de recherche *QR1* et *QR3*.

Contribution : une méthodologie pour sélectionner les le support d'exécution des composants au moment du déploiement afin d'effectuer la réservation de ressources. La réservation ressources pour des applications spécifiques est

une autre préoccupation dans la programmation des systèmes conscients de leurs ressources. Dans cette recherche, nous avançons que la mise à disposition des capacités de réservation de ressources dans le cadre de l'utilisation de composants logiciels ne devrait pas seulement être considérée lors de la conception et la mise en œuvre du modèle de composant. Au lieu de cela, nous soutenons qu'il est intéressant d'utiliser un mécanisme retardé pour choisir la technique de réservation de ressources pour chaque composant, et ce choix peut être fait en regardant les besoins en ressources de chaque composant au moment du déploiement. En bref, nous suggérons que - si un modèle de composant vise à soutenir le déploiement de composants avec des contrats de garantie ressources - les besoins en ressources et les technologies disponibles devrait être des variables de décision pour déterminer comment lier des composants à des abstractions de niveau système au moment du déploiement. Dans ce travail, nous démontrons cette hypothèse à travers la mise en place d'un prototype nommé Squirrel pour montrer les bénéfices potentiels de cette méthodologie. Cette contribution est une réponse aux questions de recherche *QR2* et *QR3*.

Dans cette thèse, une approche générative pour créer des profileurs de mémoire personnalisées pour des abstractions spécifiques à un domaine, tels que les DSLs et modèles de composants, est proposée. L'approche consiste essentiellement dans un langage pour définir des profileurs et un générateur de profileur qui cible les mécanismes d'exploration de la mémoire en utilisant la technologie JVMTI. Le langage a été conçu avec des contraintes qui, même si elles réduisent son expressivité, permettent d'offrir des garanties sur le comportement et la performance des profileurs générés. Pour évaluer l'approche, nous avons comparé les profileurs générés avec cette approche, les profileurs écrits manuellement et des outils traditionnels. Les résultats montrent que les profileurs générés ont un comportement similaire à celui des solutions écrites manuellement et spécifiquement pour une abstraction donnée. Les questions de recherche *QR1* et *QR4* sont adressés par cette contribution.

Contents

Résumé en français **iii**

Table of content **ix**

Acronyms **xi**

1 Introduction **1**
1.1 Context 1
1.2 Challenges 2
1.3 Contributions 3
1.4 Plan 4

I State of the art **7**

2 Supporting resource awareness **9**
2.1 Resource-Aware Programming 10
2.2 Managed Runtime Environments 13
2.2.1 Memory Management using Garbage Collection 14
2.2.2 MRTEs hinder resource-aware programming 16
2.3 Resource awareness in MRTEs 16
2.3.1 Resource Consumption Monitoring 20
2.3.2 Resource reservation 24
2.3.3 Remarks on existing mechanisms 26
2.4 Summary 29

3 Abstraction-oriented resource awareness **31**
3.1 Developer's View versus Tooling's View 32
3.2 Dealing with abstraction-specific requirements 35
3.2.1 DSLs: land of hungry abstractions 36
3.2.2 On how software components consume resources 38
3.2.3 State of the art on dealing with abstraction-specific features . . . 43
3.2.4 Discussing the state of the art 45
3.3 Easing the construction of resource management tools 46
3.3.1 Flexible implementation of dynamic analysis tools 47

3.3.2 Discussing the limitations 50
3.4 Summary 51

II Contributions **55**

To the reader about contributions **57**

4 Scapegoat: Spotting the Faulty Component in Reconfigurable Software Systems **57**
4.1 Motivating example: open-world scenario 59
4.2 Kevoree Component Model 60
4.3 The Scapegoat framework 61
4.3.1 Specifying component contracts 61
4.3.2 An adaptive monitoring framework within the container 62
4.3.3 Leveraging Models@run.time to build an efficient monitoring framework 68
4.4 Scapegoat Performance Evaluation 69
4.4.1 Measurement Methodology 71
4.4.2 Overhead of the instrumentation solution 71
4.4.3 Overhead of Adaptive Monitoring vs Full Monitoring 73
4.4.4 Overhead from switching monitoring modes, and the need of a good heuristic 75
4.5 Scapegoat to spot faulty components in a scalable diverse web application 77
4.5.1 Use case presentation 78
4.5.2 Experimental setup 79
4.5.3 Experimentation results 80
4.5.4 Discussion of the use case 81
4.6 Threats to validity 81
4.7 Conclusions 81

5 Squirrel: Architecture Driven Resource Management **83**
5.1 Approach 84
5.1.1 Managing resources through architecture adaptations 85
5.1.2 Describing resource management requirements 86
5.1.3 Admission control 86
5.1.4 Mapping component-model concepts to system-level abstractions 87
5.2 Reference Implementation 88
5.2.1 A resource-aware container for CPU and I/O reservation 88
5.2.2 Containers for Memory reservation 89
5.3 Evaluation 91
5.3.1 Evaluating performance overhead 91
5.3.2 Evaluating starting time 92
5.3.3 Evaluating communication 93
5.3.4 Synthesis and Threats to validity 94

5.4 Conclusion 95

6 Building Efficient Domain-Specific Memory Profilers 97
6.1 Understanding the domain 98
6.2 Language to define customized memory profilers 102
6.2.1 Abstract Syntax 103
6.2.2 Concrete Syntax 107
6.2.3 Translational Semantics 107
6.2.4 Language Usage 111
6.3 Tooling 115
6.3.1 Developers of domain-specific abstractions 115
6.3.2 Users of domain-specific abstractions 117
6.3.3 Implementation Details 118
6.4 Discussion On Language Expressiveness 121
6.5 Evaluating performance of profilers 123
6.5.1 Methodology and Setup 123
6.5.2 Impact of Analysis on the Total Execution Time 124
6.5.3 Comparing Analysis Time for an Assertion 125
6.5.4 Profiling Time in Real Scenarios 126
6.6 Conclusions 128

III Conclusion and Perspectives 129

7 Conclusion and Perspectives 131
7.1 Conclusion 131
7.2 Perspectives 132

A Concrete grammar of the language 135

Bibliography 156

List of Figures 159

Résumé 161

Acronyms

CBSE Component-Based Software Engineering. 1, 34, 35, 38, 39

CLI Command-Line Interface. 123

DSL domain-specific language. 2, 32, 36

EFP extra-functional property. 42

GC garbage collector. 14–16

GPL general-purpose language. 36

IPC inter-process communication. 24

JIT just-in-time. 13, 14

JNI Java Native Interface. 116–118

JVMTI Java Virtual Machine Tool Interface. 4, 98, 116, 118

MDSD Model-Driven Software Development. 134

MRTE managed runtime environment. 1

MVM Multitasking Virtual Machine. 24

OCL Object Constraint Language. 104

OOP object-oriented programming. 98, 105

QoS Quality-Of-Service. 40

RT real-time. 24

RTSJ Real-Time Specification for Java. 24, 45

SLE Software Language Engineering. 97

SLOC Source lines of code. 121

STL Standard Template Library. 118

Chapter 1

Introduction

1.1 Context

Software systems are more pervasive than ever nowadays. They are found in environments ranging from home appliances' controllers to complex tools for dealing with industrial processes. As a side effect, end-user's expectations have grown along the development of the software/hardware industry. However, the industry faces several challenges while coping with these expectations. Among them, we find problems related to the general issue of efficiently handling computational resources to satisfy non-functional requirements. Indeed, sometimes applications must run atop resource-constrained devices or unsafe open runtime environments [BDNG06] where efficient resource management is of paramount importance to guarantee applications' proper execution. For instance, mobile devices and smart home gateways are resource constrained devices where users can install applications from different sources. In the case of smart home gateways, avoiding any misbehavior in applications is important because these devices often control a physical environment occupied by people.

To satisfy these requirements, the goal is to make applications and execution environments aware and capable of coping with resource limitations. This is important because often such requirements emerge together (e.g., *smartphones* are resource-constrained devices providing an open executing environment). When an application includes features to react and modify its behavior after resource-related events occur, it is said to be resource-aware [BCP08, PEBN07, ALG10, PCC+11, BBH+12, ADBI13]. A software system requires the appropriate runtime support to provide such features. Nowadays, applications executing atop managed runtime environments (MRTEs), such as Java, are among the systems that can benefit from this kind of support. Indeed, MRTEs are regularly used to implement middleware [BCL+06, FND+14, Gro13, Bec10, CHP06], for example using OSGi, because of their safety, flexibility, and mature development environment. These middleware often provide open world features such as the possibility of adding new functionalities after the initial system deployment. To support the adaptability and manageability demanded by an open world runtime, it is possible to use Component-Based Software Engineering (CBSE) [GMS02, DEM02, BCL+06]. Alas, some MRTEs, such as Java, were designed to execute only a single application at

a time, so they lack full support for fine-grained resource management.

This thesis addresses the problem of supporting resource-aware programming in execution environments. In particular, it aims at offering efficient support for collecting data about the consumption of computational resources (e.g., CPU, memory), as well as efficient mechanisms to reserve resources for specific applications. Unfortunately, the mechanisms needed to support resource-aware programming highly depend on the target technology. For instance, reserving memory for a native Unix process is different from reserving memory for an OSGi bundle (i.e., the former problem involves creating a virtual address space [Sta14] while the latter demands the usage of a memory allocator and a garbage collector [OSG14, AAB+00, RHM12, GTL+10]). As a consequence, the solutions we discuss in our research are mostly useful to support resource-awareness in the context of MRTEs. In particular, we focus on this kind of execution environment when we present the contributions of this thesis.

1.2 Challenges

In existing solutions that perform resource consumption monitoring and resource reservation in MRTEs, we find two important drawbacks. Tackling these drawbacks, which are described below, is the objective of the present work.

- Solutions for resource consumption monitoring and reservation impose **performance overhead** on the execution of applications [BH06b, MZA+12a, Rei08, MBKA12]. In particular, instrumentation-based mechanisms, which offer good precision, significantly reduce applications' performance [Dmi04, CvE98, BHMV09]. While this limitation does not affect the utilization of such mechanisms to, for instance, profile an application during the development phase [CvE98, BH05b, BHV01, MBEDB06, MBT11, HB08], it does prevent their use in a production environment [Dmi04]. As a consequence, engineers are forced to choose between two poor solutions – either high overhead with good precision or low overhead with low precision – when applications require being aware of resource consumption and reservation.

- Despite of the widespread utilization of MRTEs to execute applications based on components and other abstractions, **creating resource management tools** for these abstractions is still **a complex task**. Indeed, creating abstractions, such as components models, is increasingly common [VDKV00, HWRK11, WHR14]. Plenty of tooling support exists for doing so, especially to define new domain-specific languages (DSLs) [VDKV00, Fow10, EvdSV+13, Mer10, EB10]. In addition, quite often these abstractions target MRTEs as backend technologies due to their safety and mature development environments. However, new abstractions pose a challenge for developers when it comes to profiling, debugging and monitoring applications that are built using them because such new abstractions are not always shipped along customized profilers and debuggers [KVH12, WGM08, MV11, LKV11, WG05, Fai98]. As a consequence, developers find themselves using mainstream tools that are only able to cope with *"classical"* concepts such as

objects, *methods* and *memory locations*, instead of more specific concepts. The reason for this is that defining tooling support for a specific abstraction is a time consuming task that must be balanced against the limited audience of such an abstraction.

The challenges this research tackle can be summarized in the following research questions. These questions arise from the analysis of the drawbacks in the previous paragraphs. It is worthwhile remembering that these questions refer to MRTEs.

RQ1. How can we provide portable and efficient support for resource consumption monitoring?

RQ2. How can we choose what mechanisms must be used to guarantee resource reservation with low overhead for each software component?

RQ3. How can we leverage the knowledge about the architecture of applications to drive a mechanism for resource management?

RQ4. How can we ease the definition and implementation of monitoring tools for new software abstractions?

1.3 Contributions

The outcomes of this thesis are three contributions that aim at reducing the computational cost of performing resource management, and the complexity of building resource monitoring tools. Two of them target exclusively the problem of reducing the computational cost of performing resource management while the third one also targets the problem of easing the construction of resource monitoring tools. These contributions are briefly described in the rest of this section.

Contribution: an optimistic resource monitoring framework that reduces the cost of collecting resource consumption data. Resource consumption monitoring is the foundation for resource-aware programming. In this research, a new approach built upon the idea of adaptive monitoring is presented. The approach, namely Scapegoat, is based on four principles: i) often applications are built using abstractions such as components that we can use to identify and isolate the resource consumption, ii) since consumption matters when the runtime environment is running out of resource, we can use optimistic lightweight monitoring and still be sure we will be able to detect potential failures on time, iii) it is possible to *quickly identify* the faulty component once a potential failure is spotted, and iv) there are previous monitoring mechanisms that we can leverage because they are exchangeable at runtime and offer different tradeoffs between overhead and accuracy. Scapegoat was implemented and evaluated along this work and the results show its feasibility and efficiency. This contribution answers research questions *RQ1* and *RQ3*.

Contribution: a methodology to select components' bindings at deployment time in order to perform resource reservation. Reserving resource for specific applications is another concern in resource-aware programming. In this research,

we claim that providing resource reservation capabilities in a component framework should not only be considered during the design and implementation of the component model. Instead, we argue that it is worth using a lazy mechanism to choose the resource reservation technique for each component, and this choice can be made by looking at the resource requirements of each component at deployment time. In short, we suggest that – if a component model aims at supporting resource-aware component deployment – resource requirements and available technologies should be decision variables in determining how to bind components to system-level abstractions. Along the present work, evidence for such claims are provided and a prototype named Squirrel is implemented to show the potential benefices of this *methodology*. This contribution is a response to research questions *RQ2* and *RQ3*.

Contribution: a language to build customized memory profilers that can be used both during applications' development, and also in a production environment. Memory consumption monitoring and profiling are important concerns in applications that target MRTEs because even automatic memory management is not capable of guaranteeing error-free memory management. Along this thesis, a generative approach to create customized memory profilers for domain-specific abstractions, such as DSLs and component models, is proposed. The approach consists primarily in a language to define profilers and a profiler generator which targets heap memory exploration mechanisms such as the Java Virtual Machine Tool Interface (JVMTI). The language has been devised with constraints that, although reduce its expressive power, offer guaranties about the performance behavior of the generated profilers. To evaluate the approach, comparisons between profilers generated with this approach, handwritten profilers and mainstream tools are presented. The results show that the generated profilers have a behavior similar to that of handwritten solutions. Research questions *RQ1* and *RQ4* are addressed with this contribution.

1.4 Plan

The remainder of this thesis is organized as follows:

Chapter 2 first contextualizes this research, situating it in the domain of systems engineering. We show how software systems may benefit from some degree of control over computational resource consumption (resource-aware programming). Afterwards, we present the state of the art of resource consumption monitoring and reservation, putting special interest on the performance overhead of existing solutions.

Chapter 3 discusses another concern commonly found when resource-aware support is required to deal with new software abstractions. In particular, we present state of the art approaches to ease the definition of profilers and monitors of resource consumption. The discussion revolves around two kinds of abstractions that are widely used in developing software systems.

Introduction to the contributions is a short section that quickly summarizes the challenges found in reviewing the state of the art. This section them proceeds to brief the contributions of this thesis by making clear the link among the remaining chapters.

Chapter 4 presents an approach to reduce the performance overhead of monitoring resource consumption in component-based applications running on top of MRTEs; the proposed mechanism guarantees full portability. We evaluate the approach through several experiments, and discuss under which conditions it can be applied.

Chapter 5 describes a methodology to choose what mechanism must be used to guarantee resource reservations for each component instance in a particular system. A prototype that implements such a methodology is discussed. Additionally, the merits and weaknesses of our methodology are assessed by performing a set of experiment in the prototype implementation.

Chapter 6 presents the last contribution of this research. An approach for building customized memory profilers is described in this chapter. The approach is based on the definition of a domain-specific language that can be compiled into efficient platform dependent profilers. Experiments to validate the proposal are presented and discussed.

Chapter 7 concludes the thesis by summarizing the advances that it brings to supporting resource-aware programming in managed runtime environments. It also discusses the perspectives of future research related to the thesis.

Part I

State of the art

Chapter 2

Supporting resource awareness

(A corpse in a car, minus a head, in the garage)

The Wolf - That's thirty minutes away. I'll be there in ten. [...]

(He takes a sip, then, pacing as he thinks, lays out for the three men the plan of action)

The Wolf - [...] this looks to be a pretty domesticated house. That would lead me to believe that [...] you got a bunch of cleansers and cleaners [...]

Jimmie - Yeah. Exactly. Under the sink.

The Wolf - [...] what I need you two fellas **(meaning *Jules* and *Vincent*)** to do is take those cleaning products and clean the inside of the car [...] Jimmie, [...] I need blankets, need comforters, need quilts [...] need bedspreads. The thicker the better. [...]

The Wolf - If I'm curt with you, it's because time is a factor [...]

(Pulp Fiction)

In this chapter, the context of this thesis, the general problem it faces, and the limitations of existing approaches, are introduced. The ideas behind resource-aware programming are discussed in Section 2.1, where the mechanisms required to support resource awareness in a runtime environment are also presented. Since our work focuses on managed runtime environments, Section 2.2 briefly presents the features of this kind of systems that hinder resource-aware programming. Afterwards, Section 2.3 discusses the advantages and limitations of existing solutions to deal with resource consumption monitoring and reservation (two subproblems to tackle to support resource-aware programming). Finally, this chapter concludes by highlighting the lack of proper support to develop applications that can take advantage of resource awareness, and by stating what is needed to improve such support.

2.1 Resource-Aware Programming

Efficient use of computational resources is an essential concern in software systems because it can reduce the costs of the infrastructure needed to execute applications. Resource management is always a complex task; nevertheless, it is particularly challenging when many stakeholders share a platform. As a consequence of this complexity, applications access resources by using generic APIs of the runtime environment, which is then in charge of coping with resource management concerns (e.g., managing resources is traditionally a main concern of operating systems). Although there are many advantages associated to this approach, it is known that applications built upon such generic interfaces often show relative poor performance [EK+95]. The problem arises because the implementation of a general purpose mechanism for resource management must cover many use-cases; this results in complex execution paths where nothing can be assumed regarding how client applications will consume resources. It has been shown that the performance of many resource intensive systems can be improved by carefully specializing resource management at the application level [EK+95, BPK+14, MWH14].

The principle of bypassing a generic implementation in favor of a specialized one has been widely applied in computer science and software engineering [EK+95, MR96, DO09, MWH14]. Since resource usage is a key concern for any software system, specializing resource management is potentially beneficial for many applications. In this thesis, we use the term *resource-aware* to refer to those applications/systems that observe, carefully manage, and are aware of computational resources in order to improve their performance. At first, this definition might look too broad, but in reality most applications limit themselves to carefully use the resource allocations facilities offered by runtime environments. Take for example a web server that uses a pool of threads to process remote requests. By using this pool the server is in fact carefully managing resource, but with this feature alone it is not able to observe whether the pool's size should be decreased/increased. The key issue in the definition used in this thesis is that three elements must be present in an application/system in order to be classified as *resource-aware*: observation, management, and behavior modification.

Finally, our understanding of the idea of resource-aware applications/systems is in fact closely related, but not limited, to that of *Autonomic Computing* [Hor01, KC03, BMSG+09]. Figure 2.1 depicts an example of how the MAPE-K loop can be instantiated to build an autonomic manager that is aware of resource consumption. During monitoring, two variables are considered: CPU usage and memory availability. In this abstract example, the analysis phase determines whether some components are misbehaving. Next, a plan to replace the faulty components is built. The last step in the loop involves reconfiguring the system to effectively replace the faulty components. Some knowledge is needed to guide the process; in this case, information on the architecture of the system, its components, and how they are coupled, are of benefit.

Applying resource-aware techniques to develop a software system can be motivated by the need to satisfy both functional and non-functional requirements. Among the non-functional requirements we find the following:

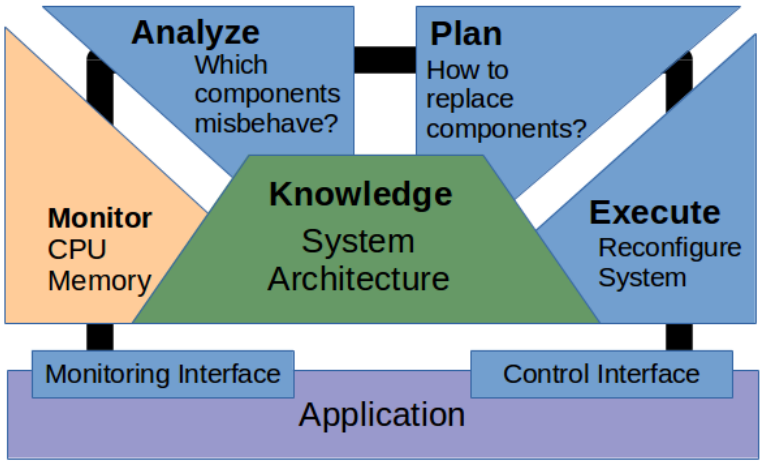


Figure 2.1 – A MAPE-K loop to support system reconfiguration based on resource consumption

- **Improve performance.** Many works show the advantages of specializing resource management to enhance system performance. Some interesting use-cases are, for instance, reducing the execution time [PCC+11], and increasing the number of requests a web server is able to handle [EK+95, BPK+14].

- **Guarantee a given Quality of Service (QoS).** Often, when the quality of a service is evaluated, we consider properties that are related either to the resources allocated to execute the service or to the mechanisms used to manage resources. For instance, misbehaviors in a property such as response time can be associated to low resource availability [ADBI09, ADBI13]. Likewise, poor QoS in multimedia systems is associated to complex resource management mechanisms in the operating system kernel [BBDS97]. Finally, resource-aware networking can be used to improve QoS properties such as data availability [BCP08] and P2P video streaming [PP07, ALG10].

- **Support per-customer resource quotas.** In Software as a Service (SaaS), it is necessary to guarantee per-tenant quotas. Although using a new application instance for each tenant is a solution, this approach leads to excessive resource consumption. On the contrary, one other solution is to design multi-tenant applications - which share most applications' code - by scheduling incoming requests in order to guarantee per-tenant quotas [KSAK14, KWK13].

- **Ensure resource isolation for critical applications.** Strong isolation among applications is often required when critical applications [Kni02] share a platform with untrusted software systems. In this context, resource usage isolation is an important concern because one application can make a second application crash

by simply monopolizing the computational resources. In this scenario, application containment [hKW00, SPF+07, MLS+15] is a useful mechanism to support resource-aware isolation. Providing specialized application containment requires extensive runtime environment support, based on approaches such as *unikernels* [MMR+13, KLC+14, MLS+15], that has been under heavy development after the widespread adoption of distributed and cloud services.

To satisfy these requirements, applications often rely on specific features offered by the runtime environment or platform (e.g., operating systems, virtual machines). In this thesis, three features that proved themselves useful to support the resource-aware programming paradigm are identified:

- **Resource consumption monitoring.** Having information regarding how an application is consuming resources is mandatory to support any form of decision making that involves the modification of applications due to resource-related issues. In this regard, it is useful to collect data about global application consumption as well as collecting such data for each application's module with clearly defined boundaries (e.g., components).

- **Resource reservation.** Ensuring resource availability for specific applications or subsystems is a way to support critical applications or other systems which exhibit, for instance, timing constraints. Reserving resources does not always imply that resources must be exclusively assigned to a process. Instead, for certain resources such as CPU time it is possible to allocate the requested resource when it is needed.

- **Observing resource availability (overcommitment).** Applications that are aware of extra resource availability are able to temporarily use such resources to improve the QoS they provide. In addition, applications can nicely modify their behavior to collaborate with other systems that share a platform if they are aware of their consumption, the resource availability and also of what other applications demand.

In this research, we focus on two of these features: *resource consumption monitoring* and *resource reservation*. In particular, this thesis devotes a significant amount of space discussing how to deal with the problem of efficiently monitoring the quantity of resources consumed by different parts of an application.

Providing support for the aforementioned features highly depends on the target abstractions provided by the runtime environment. In operating systems, resource consumption monitoring is often provided at per-process basis while virtual machine monitors (VMMs) tend to offer per virtual machine resource management. In this thesis, we focus on offering support for resource-aware programming in managed runtime environments. Hence, the next section presents a brief overview of the main properties of these systems.

2.2 Managed Runtime Environments

Applications written using a given programming language often execute in a runtime environment that implements every single built-in feature of the language and supports a specific instruction set. For a language such as *C++* the runtime environment is in charge, among others, of supporting the *throw/exception* mechanisms and the *type casting* mechanism. It is common to ship the runtime environment along the application when applications are deployed as native code because external dependencies are avoided in such a way. However, developers that use mature languages such as *C++* or *Object Pascal* lack useful built-in features that both can ease the development process, and improve applications' quality.

In 1995 the first mainstream managed runtime environment (MRTE), Java, was created. [1] It became a success even if it was not the first language providing built-in features such as automatic memory management, dynamic loading of portable code, or support for the object-oriented paradigm. This was possible due to three main reasons: the level of maturity of many technical solutions on topics such as just-in-time (JIT) compilation, the growing speed-up of hardware, and the advantages offered by the concept of *managed* languages. Since then, it has become clearer that modern languages demand features that require considerable runtime support. These features include the following [CEG+05]:

- ***Portability***: In an ideal scenario, applications should be distributed to customers in a platform independent format that is "interpreted" in the same way regardless the characteristics of concrete execution platforms. This helps to reduce the time to market of applications. To provide this feature, the application must be written in such a way that allows its execution on top of an abstract machine which has its own instruction set instead of on top of a native platform. An application can then be executed on a platform if there exists an implementation of this abstract machine that is able to run on such a platform. Having a different instruction set offers additional advantage. For instance, final applications' code can be more compact than code written using native instructions because it can represent only the high level concepts that matter to the abstract machine. To support this feature, a runtime must provide either application's interpretation, ahead-of-time compilation [MMBC97, PTB+97, WPC+11, OYM15] or JIT compilation [IHWN12, PVC01, GKS+04].

- ***Dynamic code loading***: It is also desirable to support loading new code from different sources (e.g., a network stream, a local file, internally generated code) while the application is running. Together with the reflection mechanism, this feature is useful in many scenarios such as implementing on-the-fly generation of proxy classes, implementing component frameworks, and supporting the Aspect-Oriented paradigm. Providing this feature for a MRTE is only possible using an interpreter or a JIT compiler; hence in comparison with the *portability* feature,

[1] The expression "*first mainstream*" should be interpreted here in a strictly commercial sense.

it rules out using an ahead-of-time compiler. In addition, a new requirement emerges: since code can be dynamically loaded from untrusted source, it is now mandatory to verify the correctness of such code in order to guarantee that it is safe to execute it.

- ***Automatic Memory Management***: Memory management has proven a major source of applications' crashes [NBZ08, AEPQn09]. As a consequence, automatic memory management is often a feature of modern programming languages. It is usually implemented using a *garbage collector*, a general technique that can be implemented following different approaches and often requires some compiler's assistance (e.g., mark-swept, copying, reference-counting). *Memory allocation* and *garbage collection*, which is the memory reclamation mechanism, are commonly implemented as part of the runtime environment. The memory manager is in charge, for instance, of allocating the space required by objects, and by closures.

- ***Improved error handling***: Modern programming languages tend to offer support to detect, early during the development phase and to avoid at runtime, common mistakes such as dereferencing a null pointer or accessing arrays using a wrong index. This kind of features requires extensive support for handling internal and unexpected errors as well as developer-specified faulty conditions. Implementing such features is only possible with some collaboration of the interpreter or JIT compiler.

The runtime environment support needed to execute applications written in many modern programming languages is referred as managed because the code used to execute the applications includes not only the business logic but also code to ***manage*** the memory, the possible errors, and the process of loading, verifying and compiling the code on demand.

Garbage collection, a feature present in MRTEs is relevant for this thesis. In the rest of this section we briefly describe the mentioned feature.

2.2.1 Memory Management using Garbage Collection

Automatic memory management is usually implemented using garbage collection. In this approach three entities are involved: i) the application, which is also known as the mutator in memory management jargon because it is the one modifying the memory content, ii) the allocator, which is in charge of reserving space in the heap, and iii) the garbage collector (GC), which reclaims memory that is no longer referenced by the mutator. Memory allocation follows a simple "script": a thread belonging to the mutator requests memory, in response the allocator searches for an unused block in the heap, if an unused block cannot be found then the garbage collector is invoked to reclaim blocks of memory, the allocator tries to find an unused block once again.

To understand what the garbage collector does, it is worth looking at how the memory heap is organized. The heap contains a set of used and free blocks of diverse size. A memory block can represent an *object* as in object-oriented programming or

an array, but in this description we use the generic term object as a synonymous of used memory block. Objects contain primitive values that represent the internal state of applications, but they also contain references to other objects. Due to the way in which applications allocate memory, after some time the heap contains a large set of objects that are connected by references, forming a directed-graph. In this graph, edges are references and most nodes are just objects. There is however a different class of node that are known as *roots*. Roots exist because references are not only stored within objects. Instead, references may be stored in locations such as local and global variables. If a reference is stored in a local or global variable, it is say that the referenced object O is still useful and the memory it consumes cannot be reclaimed: O is a live object. As a consequence, any object referenced by a value within O is also live. In summary, an object O is live or reachable if and only if there exists a directed path from a root node R to O. The mission of a garbage collector is to discover the set of dead objects (no longer useful) in the directed-graph and reclaim their memory.

The challenge of writing a garbage collector is reaching a good performance. A comprehensive description of different garbage collection approaches can be found in [RHM12]. In the remainder of this section, we present a brief overview of some basic techniques.

- **Mark-Sweep collectors**: In this approach, objects are allocated from a list of free blocks. Once the system decides that some memory must be reclaimed, the GC performs an initial traversal of the object graph, starting by the root nodes, following the references and marking all the visited nodes. Afterwards, the heap is traversed during a second step and every object without the mark is removed from memory.

- **Copying collectors**: In this approach, the heap is split in two spaces of equal size. Allocations are performed in one of the two spaces by simply increasing a base pointer. Once the allocation space is full, the GC is invoked to reclaim some memory. The GC proceeds by traversing the directed graph from the roots and copying every visited object to the second space. When the traversal is done, the role of both spaces is exchanged and the objects that were not copied are thus automatically reclaimed.

- **Generational collectors**: This approach tries to reduce the part of the graph that must be traversed on every GC cycle. The approach is based on the observation that most objects have a short life. As a consequence, it is often enough to traverse only a subgraph of objects that were recently allocated in order to find dead objects to dispose in a faster way. Following this idea, objects are allocated in a special space and copied to a different heap space once they "get" old enough. Generational collectors are usually combined with sophisticated variants of mark-swept and copying collectors.

There are many areas to consider in a discussion regarding garbage collection. For instance, how different approaches deal with concurrent mutators or how is the response

time improved by using parallel collectors. These topics are not mentioned here because such information is not relevant to the purposes of this work. In fact, the information to take away is that state-of-the-art GCs tend to split the heap in different spaces each of them implementing a different allocation and garbage collection mechanism. Also, GCs see the allocated objects as a directed graph where nodes that cannot be reached from the "roots" can be safely disposed.

2.2.2 MRTEs hinder resource-aware programming

MRTEs tend to offer fewer instructions and less "freedom" than native platforms. The supported concepts have a higher level of abstraction that usually favor some programming paradigms (e.g., the *invokevirtual* instruction of the JVM is used to allow method invocation as in object-oriented programming). It is common to simplify tasks such as memory management, concurrent programming, as well as resource and error handling. This greatly reduces the complexity of developing applications. On the negative side, applications may suffer some performance penalties and also lack of control. As mentioned in Section 2.1, the loss of control makes the language inappropriate in scenarios that require further assistance to deal with resources.

Among the features present in MRTEs that impact the implementation of resource-aware applications we identify:

1. Automatic memory management.
2. Dynamic code loading.
3. Support for concurrent programming.
4. The use of high-level abstractions, such as managed threads and classloaders, that do not always correspond to the concepts (process, thread) traditionally used to handle resources.

In addition, there are implementation-specific limitations that obstruct the development of resource-aware solutions. For instance, the lack of modularization in the Java HotSpot implementation of the JVM has been largely discussed [DA02, Fon04, IN05]. This lack of modularization complicates the addition of new features to the JVM. Likewise, there are many dependency relationships among different sections of code within the Java standard library that hinder resource management related tasks [BSD+08, KABM12].

2.3 Resource awareness in MRTEs

Many solutions to deal with the problems of resource consumption monitoring, control and management have been proposed. In reviewing the body of knowledge related to this topic, we are interested in solutions with specific properties. In particular, we only consider those solutions that can be applied to MRTEs. This section presents a summarized review of different approaches that we can leverage to support resource-aware programming in MRTEs. The objective is to determine how well suited are

existing approaches for supporting resource awareness in MRTEs. By focusing on a common set of properties, we are able to compare different solutions. In the rest of this section, the following properties are briefly discussed for each approach:

- **Type of resource that the approach is able to handle:** There are different types of computational resources. The mechanisms for monitoring and managing them differ considerably due to two reasons. First, the hardware/platform support for resource management varies depending on the type of resource. Second, this is also the result of intrinsic differences on the way resources are consumed. As a consequence, solutions to face resource related issues are often limited to a few types of resource. In this thesis, we are interested in the following resource types: *CPU* time, main *memory*, *network bandwidth*, and *IO throughput*. It is noteworthy that some approaches are able to deal with many resource types and others with just one type.

  Manageable type of resource - summary

  Since there is no natural order of importance among the types of resource [a], and an approach may be capable of handling many types of resource, this property is *nominal* and *multivalued*.

  [a] It may exist for specific scenarios.

- **Portability:** This property is desired on any software system. In the case of approaches that support resource awareness in MRTEs, we consider two aspects related to portability.

  The first aspect refers to whether a solution can be seamless used on a given execution environment without further modifications. For instance, some approaches rely on operating system (OS) features. Other techniques require a modified MRTEs to deliver the desired services. Finally, there are solutions that do not require features from the OS nor modifications to the MRTE. In short, we identify three values for this property: *OS Specific*, *MRTE Specific* and *Portable*. For the purpose of this thesis, we establish a partial order among these values, which is based on the superiority of the solution in terms of portability. It is clear that a solution that is both OS and MRTE independent is the most portable one. However, it can be argued whether it is better to use an OS specific solution instead of a MRTE specific approach. On the one hand, we can see it as a matter of how likely is that a solution will be adopted. Hence, it is possible that a solution based on existing OS features would be preferred over a solution based on a MRTE extended with additional features. On the other hand, MRTE specific approaches can be considered more portable due to the fact that MRTEs are themselves already ported to many OSes. In this thesis, we rely on this second criteria.

  There is a second portability aspect to consider: ease of writing a contract on resource consumption. For instance, it is hard to define a contract regarding

resource consumption if we want to execute a workload in a target platform while at the same time we control the consumed resources (e.g., 10% of CPU is probably enough to complete a workload in a development platform, but how much is the equivalent value in another platform?). The problem arises because there is a potential hardware/software mismatch between the development platform and the target platform. Hence, writing the contract using architecture dependent metrics is insufficient when dealing with heterogeneous environments [DHPW01, DDHV03]. On the contrary, using values with the same meaning in both the development and target platforms for writing the contract is a solution that eases the specification of resource consumption contracts. In this thesis, we call *fully portable* to those approaches that are OS independent, MRTE independent and support the definition of a contract using a common metric in the development and target environment. Additional advantages of using platform-independent metrics to analysis the dynamic behavior of applications are discussed thereafter in Section 3.2.2. However, it is worth mentioning that the problem of defining contracts related to resource consumption for specific platforms has been discussed elsewhere using other approaches [LP08, PHZ12].

Portability - summary

The portability is an *ordinal* property, which can take the following values: *OS Specific*, *MRTE Specific*, *Portable* and *Fully portable*.

- **Granularity:** Traditionally, operating systems have treated processes and threads as units accountable for resource consumption. More recently, virtual machines and application containers have served the same purpose. As a consequence, mature solutions exist for managing resource at process and thread levels. Unfortunately, these are coarse-grained levels that are not useful in some scenarios. In MRTEs, it is usually necessary to control and monitor resource consumption using fine-grained approaches. Four values are used while comparing different approaches in this section: *process*, *thread*, *method* and *arbitrary*. A solution provides resource awareness at the *process* if the control, monitoring and managing of resources is only possible for the whole MRTE. The granularity levels *thread* and *method* are self-explanatory. It is only necessary to highlight that if a solution is, for instance, able to handle a single managed thread then it is also capable of handling several managed threads by simply aggregating, probably with an additional performance overhead, the results of many threads. In this thesis, *arbitrary* granularity level refers to the possibility of collecting data and managing resource for any specific part of the application running on top of a MRTE. For instance, monitoring the CPU consumption of several threads plus the consumption of a few methods executed by another thread.

Granularity - summary
This is also an *ordinal* property where the values are sorted from coarser to finer granularity level.

- **Performance Overhead:** In dealing with resource awareness, a major concern is the performance overhead required to support the paradigm. The overhead is usually produced by the need of carefully monitoring and controlling how resources are used. In general, there exist trade-offs between the performance and other properties such as granularity and portability. For instance, the finer the granularity the higher the overhead.

  Despite of the importance of this property, comparing approaches based on it is nonetheless challenging due to three factors. In the first place, most approaches have been evaluated using different hardware, operating system, MRTE and benchmark. Hence, the results are not directly comparable. Second, some approaches can be applied in the context of MRTEs, but they have been evaluated in other contexts. Finally, conducting further experiments to evaluate how each approach behaves under similar conditions is not only extremely time consuming but also impractical because some approaches rely too heavily on specific platforms or they are no longer available for experimentation. Fortunately, most results have been presented as the percent of overhead produced by the addition of resource management capabilities to an already existing system. Thus, we can use these values as measurements for the comparison.

  In this section, we use the *ordinal* labels *low*, *medium*, and *high* to denote the performance overhead. These labels are used to associate a measure of quality to individual approaches. The definition of these labels is as follows: *low* overhead indicates values under 10%, *medium* overhead denotes values under 100 %, and any other value is considered *high* overhead. A reader might disagree with these definitions because, for instance, it might be argued that these definitions should be relative to the context of use (5% may be too much overhead in some domains). However, we are only using these labels to compare methods to one another. In any case, we also present the numerical value of overhead when it is available; if no numerical value is published, we discuss the reasons that led us to label a method with a given value.

  Performance Overhead - summary
  The level of measurement of this property is *scalar*. However, the scalar values should not be compared directly in this case because we are using values that were computed in different experiments. Instead, we prefer the *ordinal* values *low*, *medium*, and *high*.

In the rest of this section, different approaches related to resource management are presented. To do so, the aforementioned properties are discussed for each individual

mechanism. Section 2.3.1 covers some techniques for the monitoring of resource usage in MRTEs. A summary of approaches to reserve resource in MRTEs is presented in Section 2.3.2. Finally, Section 2.3.3 summarizes the strengths and weaknesses of each approach. This last section also highlights what are the limitations of state-of-the-art approaches.

2.3.1 Resource Consumption Monitoring

The problem of resource consumption monitoring in MRTEs is similar to that of profiling applications. After all, they both pursue the same goal - identifying parts of a system that are accountable for resource consumption. It is then unsurprising that techniques traditionally used for profiling applications had found their way as solutions for monitoring resource consumption at runtime. Likewise, approaches to collect resource usage information in OS abstractions, such as processes and threads, have been applied in the context of MRTEs. This is possible because many MRTEs rely on OS concepts for implementing concurrent programming. Among the techniques partially reused are the following:

Sampling is a technique where a separate agent is periodically executed to collect data about what an application is doing. In a common scenario, the sampler captures the value of the program counter (PC) for each thread executing within the application. Afterwards, these data and symbol table are used to identify which routines were executing most of the time. Additional information, such as the calling context, can be collected in order to build a calling-context tree. Sampling has an indisputable advantage: a low performance overhead that depends on the data collected and the sampling rate. Moreover, the data collected can exhibit a good accuracy when the sampling rate is properly chosen. Nevertheless, it is not able to collect data regarding the consumption of resources such as memory.

Instrumentation techniques are based on the idea of adding instructions to the application to collect data regarding its behavior. The instructions added, which are known as *probes*, are able to collect information about many events, such as method entry, memory allocation, execution time, system calls, and others [ASM+05, SMB+11, ABVM10]. In summary, a fundamental advantage of using instrumentation to collect resource consumption information is that the possibilities are almost countless because one has access to everything the application is doing. Alas, there is a trade-off between the number/complexity of the added probes and the resultant performance overhead: more probes implies higher overhead. As a consequence, reducing the number of probes and the complexity of each probe is of special interest for any instrumentation-based approach. In addition, it is worth mentioning that approaches exist to reduce the performance impact by temporally disabling at runtime those probes that are not necessary [Dmi04, AR01, GM11].

Reusing thread and process monitoring approaches is another common strategy. This is possible because MRTEs' implementations are usually built upon

OS abstractions. For instance, managed threads in the JVM are often implemented on top of native threads. Thus, reusing OS support for resource-aware programming in the context of MRTEs is relatively simple. Unfortunately, the abstractions used in modern operating systems are coarser than those abstractions that are present in MRTEs. As a summary, in some cases it is possible to reuse OS facilities for measuring the resource usage of a managed thread, but relying on the OS is not feasible if we are trying to monitor the resource consumption at a different granularity level.

Usually, these techniques are modified to leverage and adapt to MRTEs' features. For instance, dynamic code loading in Java greatly reduces the effort needed for instrumenting applications because the bytecode can be modified at load time without using complex patching mechanisms that are necessary elsewhere [GM11]. Another example is how *sampling*-based approaches can leverage the built-in mechanism for stack unwinding which is present in the JVM. *MRTE specific* methods have also been proposed. Since these methods require heavy modifications to existing execution environments, they are not portable. However, the performance overhead of these techniques tend to be low. Finally, there exist approaches where a mixing of different techniques is used.

Another class of approach related to measuring how applications consume resources is worth mentioning. Static analysis methods are useful to find properties of a system by just looking at its static description (i.e., source code) without dynamically monitoring its behavior. Some approaches are able to determine the worst case execution time of a Java application [SPPH10], and to calculate the worst case memory consumption of *C* applications [PHS10] (this is not applicable to MRTEs because of the garbage collector). The problem with these solutions is that developers must annotate the source code. Unfortunately, it is unlikely that such a practice will be widely adopted.

In the rest of this section, several concrete approaches are described.

Existing solutions

A solution for CPU, memory and network accounting capabilities built on top of the JVM is presented in [CvE98]. The authors propose using a mixing of *sampling*, *OS features* and *instrumentation* based on bytecode rewriting. A sampling thread simply relies on OS system calls to get the amount of CPU time used by a thread. On the contrary, a portable mechanism for memory accounting is proposed. Bytecode instructions to notify about memory consumption are inserted at each object allocation site. These instructions notify about the thread responsible for the allocation. To detect when the garbage collector deallocates an object, *finalizers* are added to non-arrays objects and a vector of *weak references* to arrays is maintained in each thread. An advantage of using bytecode rewriting for instrumentation is that no access to the application's source code is required. Network resources accounting is achieved by manually modifying the few Java classes involved on networking. Listing 2.2 shows how a method is rewritten to notify about memory allocations. A careful reader may notice two problems in the code: it is unclear how the identity of the thread allocating memory is reported, and calling *newObject* before the actual allocation may be problematic if the constructor triggers an

exception. The first problem is easy to solve: inside methods *newObject* and *newArray* there is an invocation to *Thread.currentThread*. The second issue is actually a problem of our presentation; to enhance its readability, we use Java code instead of JVM instructions to describe the transformation. However, the actual bytecode transformation is done by inserting the call to *newObject* between instructions *new* and *invokespecial*. In this way, no issue occurs when the object cannot be allocated nor when an exception is triggered by the constructor.

```
Sample createSample(int n)
{

  Sample s = new Sample(n);
  int[] a = new int[n];


  s.setA(a);
  return s;
}
```

```
Sample createSample(int n)
{
  Account.newObj(Sample.SIZE);
  Sample s = new Sample(n);
  int[] a = new int[n];
  Account.newArray(INT, n);
  Account.wrapInWeakRef(a);
  s.setA(a);
  return s;
}
```

Figure 2.2 – A method is rewritten to collect data about memory consumption.

Several approaches have been proposed to instrument applications by rewriting their bytecode; there are three reasons for the popularity of this scheme: portability, ease of use, and the capacity of monitoring arbitrary parts of an application. As a consequence, research on using bytecode rewriting for resource accounting and profiling has focus on the issues of reducing performance overhead and simplifying the development of analysis tools. On the first issue, as described in the previous paragraph, an overhead of 15% is reported by Czajkowski et al. [CvE98] when memory accounting is performed. Binder et al. [BHV01] discuss a *fully portable* approach for CPU accounting with overhead of 25%. In additional experiments conducted by Hulaas et al. [HB04, HB08], executing the SPEC JVM98 benchmark, using a framework named JRAF2 for CPU accounting, produces an overhead of 40%. Similarly, an overhead of 30% for CPU accounting is reported in [BH06c, HB08] where several optimizations are evaluated and extensive experiments are performed. Related mechanisms for writing portable profilers are presented in [BHMV09, BH06a]. In this case, the slowdown factor varies from 3.2 to 5.3 because other data about the execution context is collected in addition to CPU usage. Regarding the issue of simplifying the development of analysis tools (thoroughly discussed in Chapter 3), aspect-oriented based approaches to build profilers have been proposed and applied in different use cases [PWBK07, ABVM10]. Likewise, extensible frameworks based on bytecode rewriting have been successfully used to address the problem of code analysis [BH06b, MBEDB06, MZA+12a].

Another approach for memory accounting, presented by Price et al. [PRW03], is based on modifying the GC. In such a solution the memory heap is shared among all

tasks (i.e., thread/classloader), and a task is accountable for an object if it contributes to keep such an object alive. The marking phase of the GC is slightly modified to perform the accounting. Instead of using a unique set of roots, an additional loop over the tasks is performed. Each task contains a subset of the roots that are used to mark objects and compute the memory consumed by the task. The approach is not accurate in the sense that shared objects are not properly accounted for. The performance overhead reported by Price et al. [PRW03] is under 3%. However, in [GTM+09] an overhead up to 18% is reported using the same approach. The differences are likely the effect of using different JVMs and GC implementations. Although the overhead is low/medium, it is noteworthy that a MRTE using this approach suffers this overhead on every collection cycle. Moreover, the technique cannot be applied if reference counting is used and it remains unclear whether it is possible to integrate the approach in a concurrent collector. Similarly, an approach to trace objects in Java is proposed in [LBM15], where a modified JVM is presented. This approach is able to capture events related to the allocation and movement of objects. Relevant data, such as the thread and method responsible for the allocation, is collected after each event. Although the mechanism requires an additional phase, which is not evaluated in terms of performance, to determine when an object was deallocated, it is interesting because the performance overhead reported ranges from 2-16%.

Modifications to existing MRTEs to support lightweight instrumentation-based profilers have been proposed. In [Dmi04], an approach to instrument and (de)instrument methods on demand is proposed. The idea is to generate an additional version of each method, which includes instrumentation code. Afterwards, the runtime executes a version based on user interests. This dynamic instrumentation approach shows lower overhead than static instrumentation. Likewise, Arnold et al. [AR01] propose an approach to reduce the cost of performing instrumentation-based profiling. The insight of this approach is creating an additional version of each method where no instrumentation code is present but it is used to figure out if switching to the instrumented version is necessary. Since the switching condition can change at runtime, this approach is in fact dynamically adjusting the cost and accuracy of profiling. The results show an overhead of 6% during the profiling of CPU usage. Finally, heavy modifications to MRTEs can reduce the effort needed to perform resource accounting. For instance, due to the architecture of MRTEs such as MVM [CD01] and KaffeOS [BHL00], memory accounting in these systems shows a negligible overhead.

Operating system and hardware specific solutions have also been proposed. For instance, pooling the system to get information about how managed threads are using the CPU is proposed in [CvE98]. Performing the same task in other operating systems such as Linux and FreeBSD is also possible [SPF+07, hKW00]. Hardware performance counters is another option for CPU accounting. As shown in Overseer [PBBP11], this can be used to obtain the number of instructions executed on behalf of a managed thread. Although the overhead produces by these approaches is low, they have important drawbacks such as lack of portability and a coarse granularity level. Similarly, Banga et al. [BDM99] propose a new operating system abstraction for resource management. The authors argue that operating systems wrongly used a process (thread) as abstraction for

both, protection domain and resource management. In contrast, they propose resource containers, an abstraction for controlling at a finer level how resources are consumed by different parts of an application. These containers allow fine-grained control over the definition of independent tasks and the resources they consume. This idea is evaluated in server systems, such as web servers, showing low overhead. Although, it is not conceived for dealing with resource management in MRTEs, a middleware written atop a MRTE can benefit from such a proposal.

Resource accounting for other abstractions such as OSGi bundles have also been proposed. In [MBKA12], an adaptive approach for CPU accounting is presented. This solution aims at properly identifying the amount of CPU consumed by each bundle; the overhead is in 20-60% range. Pursuing a similar goal, a modified JVM is presented in [ATBM14] to compute the amount of memory consumed. A slowdown of 46% is produced by the proposed modifications to the GC.

2.3.2 Resource reservation

Supporting resource reservation in MRTEs has been mostly done using non-portable solutions. As a matter of fact, most approaches are based on heavy modifications to already existing MRTEs in order to add resource reservation capabilities on top of more common features. Moreover, as presented in the following paragraphs, MRTE specific solutions are only able to reserve a few types of resources. On the contrary, OS specific solutions, that are able to deal with many types of resource, are available in modern OSes. In the same way, some work have been done in devising *fully portable* resource reservation upon existing MRTEs.

KaffeOS is a modified JVM that supports the concept of isolated process at the virtual machine level [BHL00, BH05a]. It offers common operations such as process forking and inter-process communication. KaffeOS isolates the data of each process by providing a per-process memory heap where references to objects in another heap are forbidden. Since the memory is partitioned by design, reserving certain amount of memory for a particular task is a simple operation built in the execution environment. In the same way, controlling the amount of memory used by some applications is straightforward. Applications with resource requirements can be easily developed if shared memory regions are avoided. Experiments performed to evaluate KaffeOS show that it has an overhead of 11%. In a related approach, the Multitasking Virtual Machine (MVM), the authors aim at isolating many JVMs on top of a single OS process [CD01]. This approach offers advantages such as reduced memory footprint and faster initialization time for new applications. As was mentioned in the previous section, per isolate memory accounting is available in MVM; this can be used to provide memory reservation with a low overhead. Unfortunately, in the context of a middleware, there is a slowdown if communication between tasks running atop isolates is needed because –in such a case– an inter-process communication (IPC) mechanism must be used.

Deterministic response time is required to build real-time (RT) applications. Often, this requires appropriate support for CPU time reservation. However, features such as automatic memory management produce non-deterministic CPU usage that affects the

response time. The Real-Time Specification for Java (RTSJ) addresses Java limitations that prevent its use as RT environment. In particular, modifications to the memory management subsystem have been implemented because it is the biggest source of non-deterministic behavior in Java. To support systems with hard-RT constraints, RTSJ provides additional memory regions where deallocations are managed by developers. Unfortunately, the rules that define how objects in different regions interact, break the Java programming model. Support for soft-RT constraints on the other hand is provided by mean of a new GC approach, Metronome GC, that guarantees a minimum percentage of CPU time to the mutator over any interval of time [BCR03]. For instance, if a RTSJ environment is configured to ensure 80% to the mutator then of 60 seconds at least 48 seconds will be devoted to the mutator. When possible, using this mechanism is encouraged because no assumption in the programming model is broken. The features for CPU and memory reservation in the RTSJ are limited, but they show how modifications to the GC can be used to reserve computational resources.

Modern operating systems support per thread resource reservation. In Linux, control groups (CGroup) [SPF+07] is an abstraction used to specify limits on the amount of resources that threads are allowed to consume. To enforce limits on a thread, it must be attached to a cgroup. Since virtually all threads in the system can be attached to cgroups, it is possible to use the mechanism to provide resource reservation. The Linux kernel includes cgroups subsystems for resources such as CPU, memory, network bandwidth and IO throughput. Each subsystem has, however, constraints regarding how they can be used. For instance, the memory subsystem measures the consumption at per page level (i.e., 4K pages) which is of course not useful if a memory page contains objects that belong to different managed threads in a JVM. Similarly, FreeBSD offers Jails [hKW00] and resource limits that can also be used to control resource usage and per thread resource reservation.

The Java Accommodation of Mobile Untrusted Software (JAMUS) is a framework that supports the deployment of "untrusted" software components [SG02]. In particular, JAMUS provides QoS guarantees related to resources consumption. The approach follows a contractual-based paradigm for dealing with resource control. At deployment time, a component specifies what resources it requires at runtime. By *signing* this contract, the component is forced to use only those resources explicitly mentioned while the framework promises to deliver all the resources the component requests. JAMUS implements a resource broker which role is to guarantee the availability of resources for each component. Using an initial description of available resources, the broker builds and maintains a structure which represents its perception of resource availability. Before deployment, a control-admission process checks if there are enough free resources available to deploy the component. Resource reservation is done by updating the broker's perception about resources availability. Once a component is accepted on the platform, its execution is monitored to verify if its behavior is correct.

A related approach to resource control and isolation in multi-tenant applications is described in [KSAK14]. It is limited to control the CPU utilization of tenants that requests services from a remote provider. The approach applies resource demand estimation techniques in combination with a request based admission control. Resource

demand estimation is used to determine resource consumption data for individual requests. Such knowledge is used by the admission control mechanism to schedule the order in which requests are processed, which is simply done by delaying requests originating from tenants that have exceeded their CPU quota. The experiments presented show how the mechanism is able to properly isolate tenants, but little experiments related to the performance overhead is presented. The evaluation presented show an overhead in response time of 5%.

2.3.3 Remarks on existing mechanisms

Many approaches for resource accounting and reservation have been proposed; Table 2.1 summarizes the properties of those techniques presented in Sections 2.3.1 and 2.3.2. The table is split in three sections: first, a set of approaches that target solely resource accounting; second, approaches that aim at solving both resource accounting and resource reservation; and finally, techniques that only face the problem of reservation. The data provided in column *Overhead* are taken from the research papers covered by the two previous sections. This table shows a clear picture of existing trade-offs in the design of solutions to support resource aware programming. In the remainder of this section, these trade-offs and practices are discussed. Additionally, the weaknesses of different techniques (Some values in the table are marked with the symbols $\downarrow$ and $\downarrow\downarrow$, denoting weak and very weak points respectively.) are highlighted. The rest of the section also gives an insight of some good practices that help to create better solutions.

On the issue of overhead. This is a property clearly affected by both *granularity* and *portability*. For instance, low overhead is usually associated to the usage of OS specific approaches [BDM99, PBBP11, SPF+07, hKW00], which at the same time are restricted to coarse-grained abstractions such as threads and processes. It is also worth noting that MRTE specific approaches tend to show a relatively low overhead [PRW03, GTM+09, Dmi04, AR01, ATBM14, BHL00, CD01], and how these techniques often have limitations on both the type of resource and granularity level they can handle. It is in particular interesting how solutions that show low overhead have carefully crafted either their internal organization [BHL00, CD01] or operation mechanism [Dmi04, AR01] to reduce the computational footprint. On the contrary, approaches that use instrumentation based on bytecode rewriting show in general a higher overhead than other solutions. This is especially true for fully portable approaches where memory and CPU accounting are done using bytecode instrumentation [CvE98, HB04, HB08, BHV01, BH06c, HB08]. Actually, the problem with instrumentation based on bytecode rewriting is the lack of access to low-level details that can be used to reduce the overhead. As mentioned, the granularity and nature of the collected data have a profound impact on overhead. Representative cases of this issue are the techniques for monitoring of CPU and memory consumed by OSGi bundles [MBKA12, ATBM14], where despite of the proposed optimization the overhead remains at a medium level. The frameworks for portable profiling presented by Binder et al. [BH06a, BHMV09] are others scenarios where the overhead is high because of the data

Table 2.1 – Summary of discussed approaches; for each approach its properties are shown. Some values are marked with ↓ and ↓↓, denoting a weak point and a very weak point respectively.

Approach	**Resources**	**Portability**	**Granularity**	**Overhead**
Approaches for resource accounting				
Resource Containers [BDM99]	CPU Memory↓↓ Network IO Throughput	OS Specific↓↓	Thread↓	low
Overseer (HPC) [PBBP11]	CPU↓↓	OS Specific↓↓	Thread↓	low
Modified GC [PRW03, GTM+09]	Memory	MRTE Specific↓↓	Thread↓ Classloader↓	medium (3-18%)
Tracing Objects [LBM15]	Memory↓↓	MRTE Specific↓↓	Thread Method	medium (2-16%)
Dynamic Profiling [Dmi04, AR01]	CPU Memory Others*	MRTE Specific↓↓	Arbitrary*	low (6-10%)
JRes [CvE98]	CPU Memory Network	OS Specific↓↓ Fully portable Fully portable	Thread↓	medium* (18%)
J-RAF2 [HB04, HB08]	CPU	Fully Portable	Thread↓	medium (37%)
Portable CPU Accounting [BHV01, BH06c, HB08]	CPU	Fully Portable	Thread↓	medium (25-30%)
Portable profilers [BH06a, BHMV09]	CPU Memory	Fully portable	Arbitrary*	high (300-500%)↓↓
OSGi CPU profiling [MBKA12]	CPU	Portable	OSGi Bundle↓	medium (20-60%)
OSGi Memory profiling [ATBM14]	Memory	MRTE Specific↓↓	OSGi Bundle↓	medium (46%)
Approaches for resource accounting and reservation				
CGroups (Linux) [SPF+07]	CPU Memory↓↓ Network IO Throughput	OS Specific↓↓	Thread↓	low
Jails (FreeBSD) [hKW00]	CPU Memory↓↓ Network IO Throughput	OS Specific↓↓	Process↓↓	low
KaffeOS [BHL00]	Memory	MRTE Specific↓↓	Process*↓↓	medium (11%)
MVM [CD01]	Memory	MRTE Specific↓↓	Process*↓↓	low (0.5 %)
Approaches for resource reservation				
JAMUS [SG02]	CPU Memory Network IO Throughput	OS Specific↓↓	Thread↓	medium
Multi-tenant CPU isolation [KSAK14]	CPU↓	Portable	Tenant↓↓	low (5%)

being collected (calling-context tree). Finally, using adaptive monitoring and profiling techniques is a design decision that greatly reduces the overhead. In particular, using a dynamic condition to change the profiling level may have a significant impact on the running system [Dmi04, AR01]. Likewise, as shown in [MBKA12], adjusting the monitoring level using simple heuristics to reduce overhead may be decisive in a production environment.

Some limitations on the data shown in table 2.1 are worth mentioning. First, the overhead for MVM and KaffeOS do not take into account additional slowdown that may occur if these platforms are used to isolate components that are tightly coupled. In scenarios like this, the overall performance would suffer because a form of IPC would be necessary. A second limitation is the lack of good data about the performance overhead of OS specific approaches. In particular, as far as we know, there is no detailed experiments on such overhead. There are however partial experimental results. For instance, using per thread hardware performance counter can produce a 20% overhead in context-switch [Wea13]; and they can also produce considerable overhead when used in time based sampling [Wea15].

On the issue of portability. As expected, portable solutions produce higher overhead than OS and MRTE specific approaches. They are nevertheless capable of monitoring the usage of any resource type at any granularity level. Conversely, modifications to MRTEs tend to target a resource type - modified GCs are useful for memory accounting, but there is no related work applicable to CPU accounting. Non-portable approaches are also hard to apply when it is necessary to collect data at arbitrary granularity level because it is complex to anticipate how the abstractions provided by a runtime will be used by applications/systems. An example that illustrates this problem is related to how MRTEs manage memory in comparison to OSes: an OS often delivers pages of 4K while a MRTE allocates objects that are smaller than a page. As a result, memory accounting facilities as provided by OSes (cgroups) are not useful in MRTEs. In general, a solution based on a specific OS is only able to deal with a granularity level with low overhead if there are simple mappings between the concepts in the OS and the granularity level (e.g., managed threads are mapped to OS threads, which are easily monitored using OS facilities).

On the capacity to handle different resource types. CPU accounting is addressed in almost all approaches presented in Sections 2.3.1 and 2.3.2. This problem is particularly well understood; thus, solutions at all levels (OS, MRTE, application) have been proposed. However, there is limited support for CPU reservation using portable solutions, and generating low overhead. Memory accounting and reservation are available for different granularity levels, but it is still complex to craft efficient solutions. Actually, all existing mechanisms with low-overhead are MRTE specific. Network and IO throughput accounting are far less addressed problems. Even if there are portable approaches, they are limited to perform per thread accounting. Although this may seems enough, it remains an open problem.

As for the capacity of *OS specific* approaches [BDM99, hKW00, SPF+07], it is noteworthy again how they have limited use for dealing with memory is MRTEs. Likewise, an *MRTE specific* approach such as [LBM15] is not able to fully monitor memory consumption because it lacks support to quickly detect object deallocations.

Available solutions for resource reservation. In reviewing the state of the art, we realized that resource consumption monitoring in MRTEs is a problem that has been addressed far more often than resource reservation in MRTEs. Likely, this is product of a single fact: resource accounting can easily be used to implement other non-functional requirements such as dependability, QoS, and deployment of untrusted code. On the contrary, resource reservation seems a less common need because most execution environments already offer good (although less restrictive) mechanisms to handle resource. In cases where there exist strong resource requirements (such as embedded devices), engineers tend to handcraft specific solutions.

2.4 Summary

Resource-aware programming is a useful paradigm that requires extensive runtime support. We state that the scope of this thesis is limited to provide resource consumption monitoring and reservation in managed runtime environments. In our vision, resource-aware programming support must be as reusable, generic, and lightweight as possible.

In reviewing the state of the art of resource consumption monitoring and resource reservation, we realized that despite of the existence of several solutions, none of them are able to properly support resource-aware programming in MRTEs. In particular:

1. There are no portable approaches for resource monitoring that can deliver competitive performance overhead. As a consequence, customized MRTEs that support resource awareness are used when these features are needed in production environments. Due to the additional dependency, the development and deployment of applications are harmed.

2. Most resource monitoring approaches are limited to determine the resource usage at relatively coarse-grained levels such as threads, processes, and classloaders. In those techniques that are applicable in fine-grained levels, the performance of the monitored system quickly degrades.

3. Resource reservation support in MRTEs is mostly limited to using application containers. Although this approach guarantees resource usage isolation and per container reservation, applications running inside containers suffer additional performance overhead due to communication when these applications are coupled. It is our belief that in cases where strong isolation is not required, software systems would profit from lightweight resource reservation mechanisms.

Additionally, we learned important lessons that are worth considering during the design and implementation of frameworks/tools to support resource awareness:

1. Adjusting dynamically the monitoring granularity based on actual needs of the system can have a positive impact on performance overhead.

2. Reorganizing the internal architecture of execution environments to make resource awareness a first-class feature has proven a successful mean to reduce the overhead. This principle can be understood by observing, for instance, that mapping the concept of OSGi bundle to a low-level entity such as a heap region can greatly simplify memory management.

Nowadays, MRTEs are used to execute complex systems. To implement them, several engineering techniques are used – different programming paradigms, languages, and software deployment methods. This diversity hinders the development process, in particular its maintenance and evolution. To ease the development process, it is common to define and use software abstractions that address specific software concerns (e.g., components for deployment, aspects for cross-cutting concerns). However, heavily using new abstractions to develop applications, when support for resource awareness is needed, has its own disadvantages. In the next chapter, we discuss the challenges related to these disadvantages, and how they have been addressed in the literature.

Chapter 3

Abstraction-oriented resource awareness

Where is the 'any' key?

(Homer Simpson, in response to the message, "Press any key")

Defining and using software abstractions (such as classes, components and languages) are common operations when building applications. Sometimes, it is useful to consider the problem of how instances of an abstraction consume computational resources. For instance, computing the CPU consumed by all *threads* running on a system is quite helpful for system administration purposes. Another example is the necessity of knowing the number of instances of a given class when we are profiling applications.

As shown in the previous chapter, supporting resource management is a complex task that highly depends on the technology a system is running atop of. In this chapter, we show that specific features of the software abstraction, which is being targeted, also influence the way resource management support is implemented. Due to all these specific features there is considerable variability to consider when writing resource management tools; dealing with this variability is complex.

This chapter mainly discusses how the usage of software abstractions poses new challenges when we are implementing support for resource consumption monitoring and reservation. In particular, this chapter describes an abstraction – components (Section 3.2.2), which is frequently used on top of MRTEs. A comprehensive discussion on how this abstraction consumes resources is presented. Moreover, state-of-the-art approaches, for handling features specific of components and other abstractions, are presented and their limitations discussed (Section 3.2). We then present the state of the art on simplifying the construction of tooling support for resource consumption accounting and reservation (Section 3.3).

3.1 Developer's View versus Tooling's View

Building abstractions is at the core of software development. They are meant to tackle numerous problems in software engineering, ranging from providing better representation of the business logic to supporting applications' extensibility. Interestingly, abstractions are not built from scratch; instead, they are implemented upon other abstractions provided by the runtime environment. This leads to the well-known layered architecture where complex features are created using more simple concepts. In the field of operating systems, processes are built relying on low-level concepts such as hardware interrupts, context-switch, and MMU hardware. In the area of programming languages, recursive routines are implemented upon basic hardware stack manipulation.

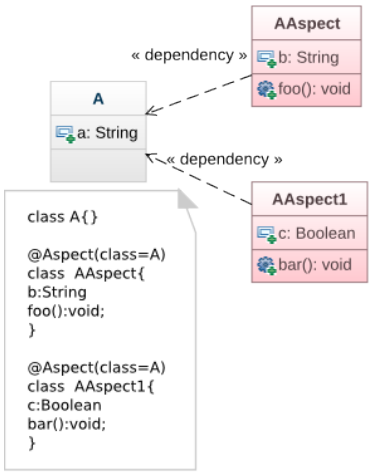
Once a new abstraction is implemented and its invariants defined, you can use it without a complete understanding of the implementation details. This has profound implications in the software development process; it now requires special tooling support because developers immediately start thinking in terms of such an abstraction. For instance, showing plain assembler instructions while debugging applications is no longer good choice once you start coding your applications in a language that includes high-level concepts such as routines, loops, and conditional-statements. In the same way, when a profiler is used to check the memory consumption of Java-based applications, the data produced is expected to reflect terms such as *object* and *class*. Ideally, tools such as editors, debuggers, and profilers must be modified to make them aware of each new abstraction introduced in the development cycle. Likewise, mechanisms to monitor resource consumption at runtime should also be modified. Unfortunately, this is not always the case. For instance, often developers lack the proper tools when they implement applications that execute in MRTEs. A couple of illustrative examples are given below; the first is associated to the usage of DSLs, the second is related to OSGi bundles.

The case of providing tools for new DSLs Many of the newly designed DSLs are built on top of existing object-oriented languages runtime such as the JVM. Therefore, people in charge of optimizing, debugging and maintaining software applications can use the existing debuggers and profilers of these platforms. However, there is a clear mismatch between classical profilers used in object-oriented systems and the newly designed languages. Indeed, the concepts introduced in these new DSLs may not exhibit a straightforward mapping to the underlying object-oriented system. As a consequence, it may be time consuming and complex to use a classical profiler to check applications that are based on these new languages.

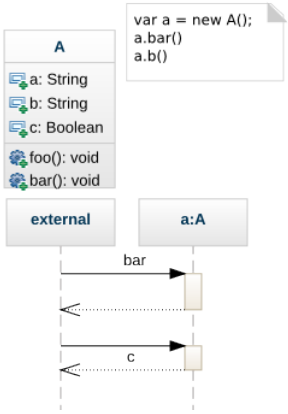
For instance, active annotations allow developers to participate in the translation process of Xtend source code to Java code via library. Such mechanism is often used directly by developers to introduce abstractions, and to define internal DSLs. In the K3-AL[1] project, active annotations are used to create an open-class mechanism on top of Java [CLCM00]. As a result, the annotation processor changes the program structure

[1] Available at `https://github.com/diverse-project/k3/wiki`

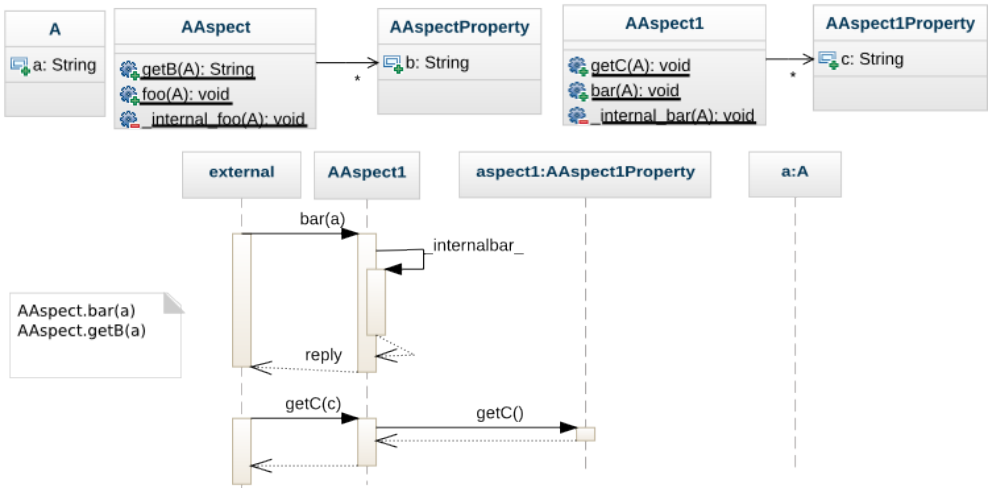
to implement this feature. It adds an indirection layer to invoke some methods (to *AASpect*), creates a new set of objects to represent part of the state of one conceptual object (*AASpectProperty*), and uses the Xtend extension method feature. Figure 3.1 shows the result of this translation process. The code written by the developer is shown in Figure 3.1a; to its right, Figure 3.1b shows the user view (the code that can be written to use the open-class mechanism); and the runtime view is depicted in Figure 3.1c. In this example, an instance of an open-class is not represented by a single JVM object; instead, it is represented by many objects.



(a) Viewpoint of library developer



(b) Viewpoint of library user



(c) Tooling's view

Figure 3.1 – Open-class mechanism in K3-AL, three views are shown: how the developer of the library see it (Fig 3.1a), how the user of the library see it (Fig 3.1b), and how it is perceived by the tools (Fig 3.1c).

The case of monitoring components CBSE is an interesting example because it is widely used. Curiously, although developers are encouraged to think in terms of high-level abstractions when software is written using components, little tooling support exists for resource awareness. Indeed, in OSGi, a component framework, many bundles [2] are deployed on top of a single JVM instance. Due to the communication mechanism used in OSGi, where objects are routinely shared, it is complex to decide which bundle should be accounted for the consumption of a particular object. A possible approach is deciding that an object O is being consumed by a bundle if O's class was loaded using the classloader C associated with such a bundle. Then, we can use this mechanism to monitor per-bundle memory consumption. However, a profiler must perform considerable – and costly – amount of processing to collect such kind of data because it is not straightforwardly available in the JVM. Given the widespread usage of OSGi, memory profilers often support collecting data regarding per-bundle memory usage. Unfortunately, similar abstractions (components models), equally implemented atop of Java, are often not properly supported by such tools because they are not as popular as OSGi. Hence, data must be manually aggregated when an application uses abstractions that are not supported by profilers; this is a considerable burden for developers.

The example of OSGi is also useful to highlight how some abstractions have specific requirements regarding resource consumption monitoring and reservation. In the particular case of determining how many resources are being consumed by a bundle, it is noteworthy that there exist two ways of doing so when a bundle requests a service from another bundle. Both approaches have pros and cons [MPH08, MBKA12]. In a first option, all resources used to satisfy a request are charged to the bundle that originally issued the request, no matter whether part of the computation is performed in other bundles. In a second option, a bundle only consumes resources when it is executing its own code. The important point is that, above the concern we present in the previous chapter regarding the mechanisms to support resource-aware programming, engineers also have to focus on features specific to each abstraction.

The problem at hand In this thesis, we argue that a *mismatch*, between the developer's view and the tooling's view, exists when the concepts managed by the developers are not clearly reflected in the tools. This mismatch may complicate the development of applications, as well as prevent the correctness of software systems. We identify in this thesis, two ways in which such a mismatch may affect software development when new software abstractions are heavily used and, at the same time, support for resource-aware programming is also required:

- The creation of new software abstractions poses challenges for software developers because abstractions may have requirements that are not addressed by generic developing tools. In particular, tools, such as profilers, and runtime monitors, may require modifications in order to reduce the gap between the user's view and

[2] A bundle is a unit of deployment in OSGi.

the tools' view. The problem is how to efficiently reuse the generic mechanisms of existing approaches to handle the specific requirements of new abstractions.

- Since new software abstractions are constantly being defined, there is an increasing pressure to ease the creation of abstraction-specific tooling support. Simplifying the definition of tools in order to support new abstractions is the issue in this case.

In the rest of this chapter, we extensively discuss both concerns.

3.2 Dealing with abstraction-specific requirements

As claimed in the previous section, when a new abstraction is defined, there is a gap between the capabilities of existing tools and users' expectations. Therefore, it is necessary to reduce this gap by modifying generic tools to make them capable of dealing with specific features of new abstractions.

In this section, we present the challenges that emerge when tooling support for resource management is being built. To do so, we discuss the topic in two ways:

1. We present how the mismatch between existing tools and users' expectation continuously emerges due to the increasing usage of DSLs to build applications. This serves both to further motivate the research, and to present in details the technical complexity an engineer may face when new abstractions are used.

2. We describe the specific features of a concrete kind of abstraction – software components. We do so because, despite of the fact that CBSE is widely used in the industry, existing approaches to support resource awareness still have limitations.

The presentation aims at highlighting the type of modifications that might be required in order to make an existing resource management tool capable of dealing with concepts defined in a new abstraction. To guide the discussion, the following elements are given for both, programs written using DSLs and component-based systems:

- A **brief description** of the main elements of the abstraction, focusing on those features that impact resource management.

- Illustrative cases of **implementations** of these abstractions **on top of MRTEs**. For instance, it briefly mentions components models that support Java.

- An analysis of how **resources are consumed** by each type of abstraction.

- **Special characteristics of these abstractions** to take into account when support for resource-aware programming is implemented.

Afterwards, the *state of the art* on providing support for resource awareness for new abstractions and, in particular, for component models is presented. This is done in the form of a summarized discussion of existing approaches. The section ends by analyzing the limitations in these approaches.

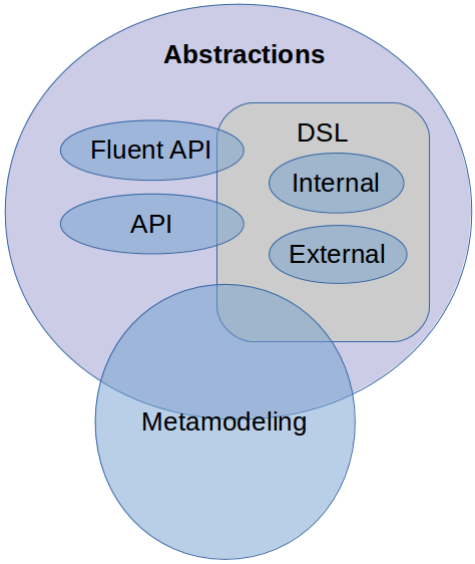


Figure 3.2 – Organizing abstractions as DSLs. This resembles the idea of "language" embraced by Czarnecki and Eisenecker [CE00].

3.2.1 DSLs: land of hungry abstractions

In their seminal work [CE00], Czarnecki and Eisenecker embrace a broad notion of "language" that encompasses what are now known as internal and external DSLs [Fow10], but it also includes libraries of routines or classes that extend a programming language because they introduce new concepts and vocabulary. A DSL describes a specific domain (e.g., state machines, interface definition language); thus, its utility is restricted to the domain it represents. This widely used notion of "language" is what we have in mind when we discuss the problems associated to resource consumption in DSLs. Figure 3.2 shows a possible organization of these abstractions.

The mechanisms used to represent concrete abstractions vary. For instance, an internal DSL is embedded into a general-purpose language (GPL). This is often done by relying on meta-programming facilities of a host language. Likewise, design patterns and reflection are used to implement some forms of internal DSLs, such as fluent APIs, and annotation-based languages. On the contrary, a "*program*" written using an external DSL requires a separated translation process (i.e., compilation) in order to produce an artifact that can be integrated as part of an application. It is then interesting that languages, regardless their representation, are always translated to concepts in a lower layer of the software stack. Developers then find the gap between their view and the tooling's view. The problem is to find out how software abstractions consume computational resources.

Support in MRTEs As mentioned, concrete "programs" written in a DSL are typically translated to some host GPL. This means that the DSL concepts are layered on

top of concepts, such as *classes*, *objects*, and *threads*. Plenty of approaches exist for writing DSL-based applications that are transformed to be executed on top of MRTEs; in particular, many solutions target the Java language and the JVM. For instance, the Xtext language workbench [EB10], the Eclipse Modeling Framework [EMF13], the meta-programming capabilities of languages such as Scala [HO10] and Clojure [Kel13] where internal DSLs can be defined and executed, the combination of annotations and annotation processor [HZS08], and intentional programming frameworks such as Meta-Programming System (MPS) [JetPS, VSBK14]. Likewise, F# applications, which execute in CLR, can use meta-programming support to write DSLs [CLW13]; and the Boo language provides constructors that are easy to use for crafting DSLs [Rah10].

Abstractions and their resources consumption Naturally, the resource consumption of a "program" written in a DSL depends on the DSL itself. If a language is only used to describe data (without any executable semantics), then a program in such a language will consume memory. On the contrary, if a language only describes behavior, a program written using such a language would use CPU to perform the computation.

An example is useful to illustrate how a DSL consumes resources. Figure 3.1 shows a program written in a state machine language that resembles a DSL described in [Voe10]. In this DSL, states, events and transitions are concepts defined by the DSL, but the guard conditions and the actions have a behavior that, as can be seen in the example, is plain imperative code. To evaluate such sections of a state machine, CPU time is required. In this case, it is necessary to know the translational semantic of the language in order to properly measure the CPU consumption.

Listing 3.1 – State machine to control the door of a bus.

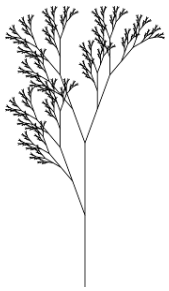
```
StateMachine BusDoorManager
  events open_door() stop_requested() bus_stops() time_elapsed()
  states (initial = DoorClosed) {
    state DoorClosed:
      on stop_requested() => ReachingStop
    state BusStopped:
      on open_door() => DoorOpen { light.on; door.open; timer.wait }
    state ReachingStop:
      on open_door() => OpeningDoor {}
      on bus_stops() => BusStopped {}
    state OpeningDoor:
      on bus_stops() => DoorOpen { light.on; door.open; timer.wait }
    state DoorOpen:
      on open_door_requested() => DoorOpen {}
      on time_elapsed() => DoorClosed { door.close; light.off }
  }
```

In other cases, the translation process of a DSL may only generate a structure for each "program", without any associated behavior. Hence, only memory is consumed by the execution environment to represent a concrete model at run time. For example, suppose we define a language to represent plants using the L-System formalism [PL90]. A plant created by Prusinkiewicz et al. [PL90] using L-Systems, is depicted in Figure 3.3.

The structures generated by this language may be simple strings of symbols (see 3.3c). The memory consumed by a L-System depends on the number of iterations, the rules defined, and the concrete data structure used to store the symbols (it can be a string of characters, a tree, a list, or an array). The important point is that users of this language should see this kind of structures as black-boxes. Hence, in this example, the *object* used to store the plant must be seen as structure of type *OL-System* instead of as a simple Java *string*.

```
plant P0
  contants: + − [ ]
  variables: X F
  initial: X
  rules:
    X −> F[+X]F[−X]+X
    F −> FF
  iterations: 7
  angle: 20 degrees
```

(a) DSL Code



(b) Graphical representation

```
String P0 = "FF[+F[+X]F[−X]+X]FF[−F[+X]F[−X]+X]+F[+X]F[−X]+X";
```

(c) Java code generated after translation (two iterations instead of seven)

Figure 3.3 – Simple OL-System to generate a plant in two dimensions. On the left, the OL-System is represented using a DSL, on the right we show the tree that can be generated using such OL-System, below is the representation in Java.

An important **feature to consider** by engineers who implement support **for resource awareness**, is that languages do not always provide well-defined boundaries. For instance, an internal DSL may be transformed into a list of statements of the host language without defining a new routine or thread. In a case like this, instrumentation is the only mechanism that can be used to achieve CPU consumption monitoring. Interestingly, interaction between parts of an application written using DSLs can also impact the way resource accounting is done. In the state machine example previously discussed, a state machine might trigger events in another state machine by simple executing some actions in one of its transitions. In a situation like that, a resource accounting framework should properly determine the change of execution context; unfortunately, implementing this can be computationally expensive.

3.2.2 On how software components consume resources

CBSE is a particularly interesting – and concrete – example of how tools that provide support for resource accounting and reservation must take into consideration features

specific of each kind of abstraction. In this section, we thoroughly discuss the concerns to address when dealing with component-based systems.

CBSE aims at developing applications by reusing independent units of software [GMML12, CSVC11]. Through the utilization of components, connectors and configurations, CBSE reduces the complexity in the development and maintenance of systems [DvdHT02, MT00, vOvdLKM00]. One of its technical advantages is that it facilitates the management of dynamic architectures [NGM+08, JBD15] because it simplifies the implementation of features, such as, self-organizing the structure of a system, and self-adapting its behavior [PLM12, JBD15, ZGC09]. Likewise, many works [GMML12] have shown the benefits of using component-based approaches in open-world environments [BDNG06, CFG10, PPMB10].

In a general sense, the concept that embodies the idea behind software components can be defined as follows [CSVC11]:

Definition 1: A *Software Component* is a software building block that conforms to a component model.

Definition 2: A *Component Model* defines standards for (i) properties that individual components must satisfy; and (ii) methods, and possibly mechanisms, for composing components.

Plenty of diversity exists in current component models and frameworks [HC01, SGM02, CSVC11]. They tend to target different technologies, aim at different use cases, provide support for different concerns, and use different design principles. Crnkovic et al. [CSVC11] propose properties that can be used to classify component models; we are interested in those models that have the following properties:

Modeling capabilities: it is common to provide a mechanism for modeling the system architecture during the development phase; this results useful to reason about the system. In addition, it is possible to support some form of reflection for querying the architecture of a system at runtime. Component models that include both features are the target of our research.

Deployment of components at runtime: since, we are dealing with the problem of supporting resource awareness in open environments, we focus on component models that allow component deployment at runtime. Many component models are able to cope with the necessity of adaptation through, for example, the deployment of new modules, the instantiation of new services, and the creation of new bindings between components [Por14, ZWK14, IFMW08, GMPLMT10].

Other properties worth mentioning in this thesis are those that describe how components communicate. For instance, whether the concept of *port* is implemented; if there exists distinction between *required* and *provided interfaces*; characteristics of the *interface language*; and what is the *communication type* (synchronous, unicast, and others). Our interest in these properties is limited to understanding what mechanisms are used to support interaction between components, how are these mechanisms implemented,

and how this interaction affects the way in which we deal with resource consumption monitoring and reservation.

Support for component-based engineering in MRTEs An important property of component models is its implementation support. The discussion in this research is limited to those that have been implemented for MRTEs. Several component models that provide support for MRTEs have been proposed in both the industry and the academy. Among others, we can mention Enterprise Java Beans (EJB) [Gro13], the Open Services Gateway Initiative (OSGi) [The12], Fractal [BCL$^+$06], SOFA 2.0 (Software Appliances) [BHP06], Palladio [Bec10], and Kevoree [MBNJ09b, LLC10]. It is interesting how these component models have different properties when it comes to *modeling capabilities*, *architecture of the system supported*, and *constructs for interaction among components*. However, they also differ in how components are represented on top of MRTE concepts; in other words, they follow different approaches to implement the component framework itself.

How components consume resources Interestingly, components provide boundaries between different software entities, which are forced to communicate through well defined interfaces; it is then possible to write Quality-Of-Service (QoS) contracts associated to these interfaces [BJPW99].

It is important to remember the difference between component type and component instance when we discuss the resource usage of these systems. Indeed, instances may have a state while component types are stateless. The distinction is important because all instances of a single component type share the same implementation. As a consequence, it is not simple to define how a component consumes resources. For example, the memory consumed by an instance includes those *objects* used for the component framework to represent the instance itself, its ports, and bindings; it also includes the state of the component. However, it cannot include the memory used to store the component's code since it is shared among many instances. Monitoring CPU and network consumption is even harder, because the code responsible for the consumption is shared. To solve this problem, a context is associated with each component in order to determine at runtime the instance responsible for the execution of a given operation that is using resources. Interestingly, the exact representation of this context depends on the component model, and it may impact the performance of component-based systems. A common way to define this context, is associating a set of threads to a component.

To summarize, the resources consumed by a component comprise (but are not limited to): its state, the time-shared resources it uses (CPU, network), the space required to store data and code shared among all instances of a component type, and the temporary space needed to execute the component.

Contracts on resource consumption QoS contracts serve, among others, to describe how components consume resources. They simply express what resources will be consumed by a component to perform some action. In writing such contracts, it is better for developers to use platform-independent metrics. Indeed, doing otherwise hinders the

interpretation of a contract when components are deployed in different platforms. For instance, if n CPU cycles are required to handle a request, this n may, in fact, represent different amount of CPU time depending on the architecture. This is the reason why, the number of bytecode instructions to be executed, which is platform-independent, offers the following advantages:

- It is easy to control the admission of components because each platform knows how many bytecode instructions it is able to execute in an interval of time.
- A portable framework for resource consumption monitoring is easier to implement.
- Different measurements are easy to compare, even across platforms, because they use the same metric.

Besides the metrics used, the exact interpretation of a contract is also a fundamental concern. Contracts may express properties for just one component and a single type of resources, but they can also express how should be the usage of resources in more complex scenarios where different components and types of resources are involved.

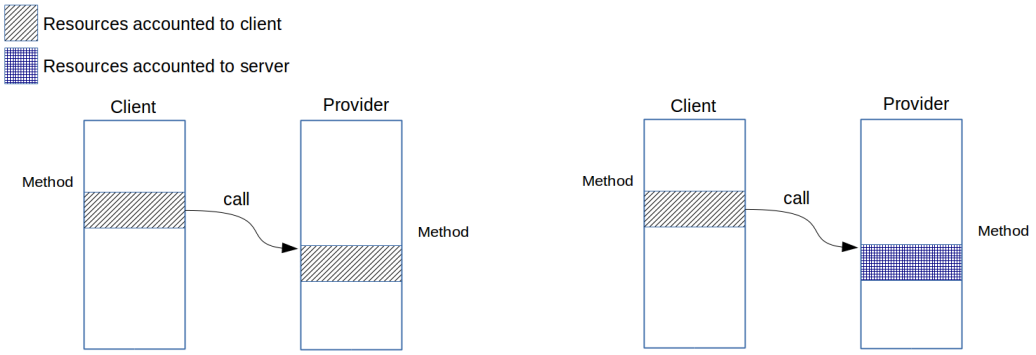
Specific features to consider A first issue to take into consideration is *how to account for resource consumption in the presence of interaction between components.* Usually, components are organized as clients and providers, where a component (provider) performs operations on behalf of other components (clients). It is then possible to account for resource consumption in two ways [MPH08, MBKA12]:

Indirect accounting: all the resources consumed to serve a request that was originated in a component A are accounted to A (See Figure 3.4a). In other words, there is no resource consumption accounted to service providers.

Direct accounting: the resources consumed during interaction are accounted to the provider (See Figure 3.4b). For instance, the CPU used by a code that belongs to component A is accounted to A, no matter if the code is executed on behalf of a client.

Both ways have advantages and disadvantages. In the case of direct accounting, if a provider is called in an endless loop, the resource usage will be accounted to the provider instead of to the client that executes such a loop. On the contrary, if a service is poorly implemented, in indirect accounting the user of the service is identified as the responsible.

Similarly, there is another problem to determine the memory consumption of components. Often, objects are created by a component (allocator) and used in other components (clients). This happens during interaction among components, when they exchange data in the form of objects (e.g., a service component creates new objects in response to clients' requests). For instance, a component may allocate space for an object, then send it to a client, and finally forget about it. In that case, it is not clear what component should be accounted for the memory consumed by that particular object; it



(a) Indirect Accounting (b) Direct Accounting

Figure 3.4 – The two mechanisms to account for resource consumption when components interact.

can be either the client or the original allocator. Both approaches have pros and cons. The intuition dictates that if a component is preventing an object of being collected it should be accounted for that particular block of memory. However, such an approach may be dangerous if a buggy (or malicious) provider creates objects unnecessarily big, sends them to clients, and forgets about them; in that case, it would be hard to identify the provider as the source of excessive consumption. Most existing solutions for memory consumption follow a fixed criterion, either account always for the allocator or for the component preventing the object's collection.

A second aspect is to decide *where should be implemented* the mechanism for resource consumption monitoring and reservation. Essentially, this consists in, selecting which actors implement the mechanism and policies to manage resources, and deciding if the actors collaborate to achieve their goal [CSVC11]. Indeed, many component models provide no facilities for managing extra-functional properties (EFPs). In these cases, the mechanism used to handle a property is left to the designers of each application. This facilitates the creation of EFP management policies that are specifically tuned towards a system, and also allows the use of multiple policies in a system. On the contrary, other approaches favor the separation of concerns between functional and non-functional aspects. Hence, components are only allowed to address functional aspects, while containers are in charge of wrapping components to guarantee EFPs.

Finally, some differences among component models impact the implementation of mechanisms for resource consumption monitoring. For instance. Kevoree and OSGi provide different methods for interaction between components. In Kevoree, components communicate with each other through ports. It is straightforward to identify in the code when a component is requesting a service because a single interface (port) is used to do so, no matter if a component is using different ports. As a consequence, an automated tool can easily instrument the code to detect when components are communicating. On the contrary, OSGi uses plain Java interfaces and objects to connect bundles. In that cases it is more complex to detect when components are communicating because

a bundle can communicate with several services using different interfaces.

3.2.3 State of the art on dealing with abstraction-specific features

This section presents a several approaches that tackle the problem of providing tooling support for developing applications using new software abstractions. In particular, it discusses some solutions for debugging domain-oriented abstractions. Furthermore, this section describes several mechanisms to monitor how components consume resources and to isolate such consumption.

Providing resource awareness support for DSLs Among the development tools that help to software maintenance, debuggers and profilers generally support mainstream concepts such as *classes*, *objects*, *methods*, and *call stack*. Using them, it is possible to determine, at some extent, how applications are consuming resources. However, we have found that limited support exists for extending the usage of these tools in order to make them understand more specific concepts (e.g., classes representing the business logic of an application, or a design pattern). In other words, they are not fully capable of dealing with resources at per DSL level. Yet, some related works do exist; they aim at reducing the gap between the developer's view and the tools' view. In particular, the problem of offering debugging support for DSL constructs have been discussed. In Xtext [EB10], languages that produce code in the base language (i.e., Java) may profit from mostly automatic debugger support. When the newly define DSL is transformed into the base language, the system keeps traces between the two models (source and destination). Using these traces, a debugging infrastructure is able to identify what constructor is being executed in the original DSL. Meanwhile, Voelter [Voe10] discusses how to add debugging capabilities to a DSL when the MPS language workbench is used. In this case, no trace model is required; instead, every concept of a language that requires debugging support must implement a set of interfaces to guide a generic debugging framework. Similar works have been conducted [vdBCOV05] for the ASF+SDF Meta-Environment [vdBvDH$^+$01], defining a generic debugging framework that can be customized for DSLs. Finally, mechanisms to build various tools for new languages are described in [HPM$^+$05]. Based on attribute grammars, and implemented using the LISA system [MLAZ02]; the tools proposed include editors, inspectors, debuggers and visualizers.

Component-based systems Many works address the issue of supporting resource consumption monitoring in component-based systems. In addition, some existing approaches present solutions for the isolation of resource consumption among components running on top of a single MRTE instance. Unfortunately, most approaches are limited to a specific component model; in particular, works exist to solve the problem for both EJB and OSGi.

EJBMemProf, a framework for profiling the memory consumption in EJB is presented in [MV05]. The main idea of this framework is instrumenting applications' code to trigger an event each time an object is allocated; in response to such an event the

framework identifies the component responsible for the allocation. A set of rules that guide this identification process is discussed by the authors, and their accuracy evaluated. Unfortunately, the overhead of the system is too high because, as part of these rules, the class name of each allocated object is compared against the package name of each component (this impacts the performance even if *hashes* are used to compare *strings*). As a consequence, using the framework in a production environment is not possible. Instead, this tool aims at supporting the development of component-based software. Similarly, an approach to measure the execution time of EJB components is proposed in [ML05]. In this solution, engineers manually select those parts of a component that may be responsible for most CPU consumption at runtime. These parts are then profiled in a development environment; the resulting data is combined with a description of the deployment platform to estimate what would be the execution time in the deployment platform. Finally, a mechanism for measuring the response time of components, as well as the invocation tree, is discussed by Meyerhoefer et al. [Mey07]. This approach uses interceptors to collect data about how components calls each other. It is intended to be use as a development tool.

Monitoring the resource consumption of OSGi bundles has also been addressed. For instance, Miettinen et al. [MPH08] present a framework to measure CPU and memory usage of OSGi bundles. This framework relies on some modifications to an existing OSGi platform in order to identify which bundle is consuming a given resource. Such a modification creates a unique *ThreadGroup* for each bundle; since each object allocation and method execution is performed by a *thread*, it is possible to figure out the bundle responsible by simple looking at the *ThreadGroup* of the thread. Alas, this approach suffers of considerable performance overhead because it extensively uses JVMTI and bytecode rewriting to detect resource usage (See previous Chapter). In [MBKA12], the authors propose an approach to reduce the overhead induced when CPU consumption is monitored; this is an adaptive monitoring system that is able to dynamically tune the accuracy of monitoring mechanisms depending on detected performance issues. This solution is built on the idea of creating proxies that are responsible for detecting invocations, and also on the usage of localized CPU sampling. The experiments show an overhead of 2% when idle (the lightweights monitoring mode) and 20% when completely active. Memory consumption monitoring in OSGi execution environments has also been discussed in [ATBM14], where the authors argue that some information regarding the *business logic* is required to properly estimate the resource consumption of interacting bundles that belong to different stakeholders. To encode that information, they propose a DSL that describes what component must be charged for a given consumption when two components interact; this effectively increases the accuracy of the monitoring framework. Unfortunately, the approach requires a modified JVM, and a persistent overhead is induced (up to 46%) because the framework cannot be deactivated.

Other approaches address the issue of providing resource isolation between OSGi bundles. In [KYK+14], the authors propose a memory isolation method for OSGi-based home gateways. The method isolates the memory consumption of bundles without the need to modify bundles or the OSGi framework and has minimal overhead costs. It does

so by modifying the JVM (object layout, allocator, and the garbage collector) in order to evaluate, after each allocation, whether the bundle responsible for the allocation is violating some developers-defined limits on resource usage. Meanwhile, I-JVM [GTM+09] is a JVM that provides isolation between OSGi bundles. In addition to avoiding unintended object sharing, the approach also tackles the issue of resource consumption monitoring for components; this is meant to, for instance, be used by administrator to avoid denial-of-service attacks. The experimental results show an overhead of 16% on inter-bundle calls. Likewise, the problem of isolating CPU consumption in OSGi execution environments in order to support real-time component software development is discussed by Richardson et al. [RWDD09]. The idea is to use the RTSJ to support CPU isolation; this also requires modifications to the OSGi framework. The authors claim that just using RTSJ is not enough to ensure real-time properties of applications when the OSGi framework is used to build them. Besides the arguments to justify such a claim, a solution to achieve CPU isolation is presented.

Profilers such as Eclipse MAT and VisualVM offer support, although limited, to perform memory consumption monitoring for mainstream component models (OSGi and EJB). They do so by providing built-in subsystems that are able to process Java memory dumps to calculate the per-component consumption. Due to the need of processing the complete memory dump, the performance overhead is considerable.

3.2.4 Discussing the state of the art

In reviewing the state of the art, we have found some limitations on how existing solutions deal with specific features of abstractions. In this section we discuss such limitations:

Limited support for resource accounting in component-based systems There are several approaches for monitoring how individual components use resources. However, they are limited and inefficient. First, most of them only target mainstream component models such as OSGi and EJB; this is a fact noteworthy because it is not clear whether the ideas behind existing approaches can be applied to other component models. The second and more important problem is the considerable overhead induced by these solutions. In the case of memory consumption accounting, the best results we found show a persistent overhead of 46% (medium). By comparison, experiments of CPU consumption accounting show a lower overhead of 20% when *sampling* is used. In many scenarios, these overheads are unacceptable.

Sub-utilization of information about the architecture of systems . Existing approaches only use rudimentary information about the architecture of the system that is running on top of the component framework. In other words, they are able to determine how each component consumes resources (regardless of the accuracy). However, as shown in [MBKA12], it is possible to use additional information on how components are connected (their dependencies) to improve the monitoring accuracy. Similarly, Attouchi et al. [ATBM14] show how to use the

knowledge about the *business logic* (in particular how components interact) to properly determine what component should be accounted for the consumption of an object. Nevertheless, we have not found results showing how to reduce performance overhead by using information about the architecture of component-based systems.

Non-portable solution for resource isolation Existent approaches to provide per-component resource consumption isolation require a modified MRTE. In practice, this prevents the adoption of these approaches in production environments; due to the cost of maintaining a customized MRTE, and the complexity of keeping it up to date, some managers with limited budget may decide that these solutions are not acceptable.

Wrongly assume that resource consumption is homogeneous Usually, components consume resources in different ways; some of them require CPU while other are memory or IO consumers. However, many approaches to resource consumption and reservation are built without taking this into consideration; in short, they manage resources in the same way for all components. This may have negative consequences in some cases. For instance, some monitoring frameworks to calculate CPU consumption induce a persistent overhead in all components, even if only one of them is consuming too much CPU. In the same way, solutions tend to use fixed mechanisms for monitoring and reserving resources, but we show in the previous chapter that some mechanisms are better for handling some kind of resources even if they are not able to manage all resources. It is our belief that adaptive mechanisms are preferable because, by analyzing the requirements and observing the status of a system, they are theoretically capable of reducing the overhead.

Limited capabilities for building tooling support Debuggers, simulators and interpreters have been proposed as tools to support the maintenance of software written using DSLs; however, as far as we know, no profiler that specifically aims at reducing the gap between DSLs and base language have been presented. It is worth mentioning that some general profiling frameworks can be used to ease the construction of profilers, even if they do not specifically address DSLs; these mechanisms as well as their limitations are discussed hereafter.

As shown in this section, reusing existing solutions for resource consumption monitoring and reservation, and adapting them to specific features of new abstractions is important to ease their adoption. We have seen in this section several approaches that target specific abstractions such as, component models. However, since defining abstraction is so common, it is impractical to manually build tools with specific features for all of them. Hence, further reducing the engineering effort required to build resource management tools is highly desirable. The next section presents several approaches that address this issue.

3.3 Easing the construction of resource management tools

As we already mentioned, the definition of new abstractions is common in software development (for example, using DSLs). Sometimes, it is useful having tools to check how instances of these abstractions consume computational resources. Unfortunately, building such tools is not a simple tasks. It is then necessary to provide a "simpler" mechanism to build tooling support for abstractions.

Resource consumption monitoring is a form of dynamic analysis, and there are many approaches that tackle the issue of simplifying the definition of dynamic analysis tools. Unfortunately, we have found fewer mechanisms to ease the construction of resource reservation tools. Due to this fact, we mostly discuss in this section the problem of supporting resource accounting.

In this section, we first present some approaches that aim at easing the construction of dynamic analysis tools. We do so by describing the following dimensions for each approach:

Generality Indicates the expressive power of the approach. We only consider two values: *arbitrary* and *limited*.

Ease of use We argue that the possible values are *already known language*, *new language*, *require specific knowledge of the MRTE*.

Performance Overhead As in the previous chapter, we use the values *low*, *medium*, and *high*. Likewise, these values are taken from the literature review.

In Section 3.3.2, we briefly discuss the limitations of such approaches.

3.3.1 Flexible implementation of dynamic analysis tools

One of the most tedious and error-prone tasks, when building tooling support for dynamic analysis, is dealing with low-level details such as, bytecode instrumentation. If dynamic analysis tools are built from scratch, developers are forced to focus on mastering details of the execution platform, when in fact, they are interested in implementing high-level ideas. To address this issue, various approaches have been proposed; they can be clustered into three categories: instrumentation frameworks, high-level APIs for bytecode manipulation, and aspect-oriented tools.

Instrumentation frameworks Several instrumentation frameworks for MRTEs have been proposed. In [LV99], the authors propose the JVM Profiling Interface (JVMPI); a set of low-level facilities built-in the JVM to trigger notifications when certain events occur during application execution. Although the events reported only provide primitive information, such as *method invoked* and *thread created*, this basic data can be used along other infrastructure to build more powerful tools. Severe limitations prevented the success of JVMPI (deprecated in favor of JVMTI): performance impact on the JVM, the relatively low-level interface provided, and the limited capabilities to detect

fine-grained events. These limitations are partially addressed in [MBEDB06] where a framework called Javana is proposed. In specific, Javana proposes further modifications to the JVM to detect more events. Additionally, it aims at easing the construction of efficient user-defined profilers by providing a generic instrumentation framework that can be adapted to specific needs. To do so, users define a profile using an aspect-inspired DSL where *pointcuts* represent the events of interest for a profiler, and advices, which are written in *C/C++*, are in charge of collecting data on the dynamic behavior of a program. Besides the problem of requiring a modified JVM, this approach also has other disadvantages such as demanding a deep understanding of the JVM.

A profiling framework that instruments Java programs at the bytecode level to build context-sensitive execution profiles at runtime is proposed in [Bin05]. The framework includes an exact profiler as well as a sampling profiler. Users can define their own profilers using a provided infrastructure for program transformation. The most interesting point is that profilers are written in pure Java; this lowers the barrier for Java developers who devise customized profiling strategies. Finally, Reiss proposes a framework [Rei08], DYPER, to organize and schedule the execution of monitoring agents. Each agent (so-called proflet) is able to obtain data, regarding some properties, using two approaches. Through sampling the data collected have poor quality, while data collected using instrumentation are very detailed. The framework schedules the execution of proflets to guarantee a bound in the overhead of monitoring resource consumption. To perform the scheduling, each proflet provides an estimate of both the expected application overhead and the time needed to set up the detailed monitoring; this information is used to dispatch the execution of a detailed collection of data. In practice, this is a form of adaptive monitoring where mechanisms with high overhead are executed only when possible. Proflets are built using either Java or C, and they can be composed in order to collect more complex data. The main limitation of this approach lies on the difficulty of properly estimating the overhead of proflets.

High-level APIs for bytecode manipulation The use of bytecode rewriting techniques to build dynamic analysis tools have leaded to the development of high-level APIs for bytecode rewriting. For instance, ASM [BLC02, Kul07] is a Java bytecode manipulation and analysis framework written itself in Java. It is useful to transform classes directly in binary form. To do so, it provides methods to traverse the binary code of Java classfile that allows users to create custom transformations. Alas, it requires considerable knowledge regarding the JVM specification. In particular, it is mandatory to understand the Java instruction set, how are they executed, and the basic structure of a classfile in Java. By comparison, Javassist [Jav99] simplifies Java bytecode manipulation. It is a class library for editing bytecodes in Java; however, unlike ASM, it provides a source level API: to transform a classfile without knowledge of the specifications of the Java bytecode. For example, you can specify what bytecode to insert in an existing class by using plain Java source code – Javassist compiles it on the fly and inserts it on the class being transformed.

Aspect-oriented tool construction The collection of data for a given dynamic analysis can be easily understood as a crosscutting concern. Because of this, researchers have been attracted by aspect-oriented solutions. Indeed, aspect-oriented frameworks already provide the mechanisms to i) specify multiple points of interest in the binary code of an application, and ii) execute handlers when the program counter reaches these locations. In other words, by simply defining aspects (*pointcuts* and *advices*), developers can focus on the high-level ideas of dynamic analysis. Nonetheless, aspect-oriented solutions are not flawless; some limitations have prevented its adoption in this field. An interesting evaluation of the positive and negatives points of using aspect-oriented dynamic analysis is presented in [PWBK07]. In addition to four dynamic analysis presented and evaluated (showing medium or high overhead), the authors also discuss how the *pointcuts* of AspectJ are not sufficient to achieve better performance nor to create any type of analysis. The issue of improving the performance is tackled by Binder et al. [BH06b]. In their work, the MAJOR [VBMA11] framework, an aspect weaver that enhances AspectJ with support for comprehensive weaving, is extended to guarantee fast sharing of values between aspects. This simple addition is enough to reduce the performance overhead of some dynamic analysis.

DISL [MZA+12a, MZA+12b] is a domain-specific aspect language for bytecode instrumentation; it uses annotations and plain Java to describe what a dynamic analysis tool must do. The novelty of this approach is that new *joinpoints* and *guard conditions* can be defined using the Java language along some annotations. It is then possible to collect data that is not accessible using a standard framework such as AspectJ. Unfortunately, to define new *jointpoints*, some knowledge of JVM internals is required. Finally, in an effort to overcome the limitations of specific aspect weavers, which prevent using them to implement arbitrary dynamic analysis tools, Achenbach et al. [AO10] propose an approach to customize aspect weavers. When a new concrete weaver is built, the developer can choose arbitrary locations in the program as *joinpoints*. In the same way, different strategies to weave *advices* can be implemented. Implemented in Ruby, the approach is evaluated through the definition of a debugger and a testing tool.

Memory profilers Other approaches focus on profiling the memory usage of applications. Memory profilers that are widely used in the industry provide languages to perform mostly arbitrary queries on the set of objects loaded in the heap. For instance, in Eclipse MAT [BEK+06] and Visual VM [OQL14], users can write queries in OQL (a SQL-like language) to retrieve information. Despite of the fact that few constructors of OQL are really implemented in Eclipse MAT, this approach would allow, in theory, collecting practically any information contained in the heap. Similarly, YourKit [you03] provides a language based on set theory to filter objects with specific properties; this language is used when no built-in memory analysis can provide the desired data. Besides providing query languages, mainstream profilers also support the development of extensions (e.g., plugins written in Java); these extensions essentially traverse the graph of objects in a heap dump to collect information. Alas, both queries and extensions require costly operations – a complete dump of the heap, and a step to preprocess the dump; only after these operations, the frameworks are capable of executing queries.

DSLs for resource monitoring and reservation In DeAl [RIS+10], the authors propose a language to compute heap assertions at garbage collection time. The design of this approach aims at guarantee a low performance overhead. To ensure such a property, the language is only able to compute boolean outputs; while resource consumption monitoring, for instance, needs to compute values of integer type. DeAl is a purely declarative language; in exchange for the declarative style and the focus on assertions, some properties about the performance of queries written in DeAl can be formally proved.

Bossa, a language that aims at easing the definition of new scheduling policies is proposed by Muller at el. [MLD05]. The language defines a set of concepts related to the domain of scheduling; they can be used to facilitate both the specification of scheduling policies and the verification of safety properties. A scheduling policy is transformed into a corresponding *C* code that can be plugged to the target OS kernel. In practical scenarios, new scheduling policies induce low performance overhead on application. The most important drawback nevertheless is that implementing a compiler for the language requires a deep understanding of low-level details of the target kernel; and it might even require modifications to a kernel. If available in a platform, this mechanism can be used by an application running on top of a MRTE to defining CPU reservation policies.

3.3.2 Discussing the limitations

Although there exist many approaches that aim at reducing the complexity of writing tools to support resource-aware programming, in reviewing the literature we have found some limitations that prevent using such solutions in production environments. In particular, the identified limitations are related to the three aforementioned dimensions, *generality*, *ease of use*, and *performance overhead*. Unsurprisingly, these three dimensions are often in contradictions. For instance, most approaches that offer low performance overhead require considerable knowledge about the target technology (not easy to use). The question is whether the trade-offs followed by different approaches are good enough. In this section, we discuss in details the limitations we find in these approaches.

Tools have high overhead Many times, the tools that we can build, using the approaches presented in the previous section, induce high overhead. This is particularly true for solutions based on bytecode rewriting, but also for other event-based approaches, such as JVMPI. This is not surprising because, as we discuss in the previous chapter, bytecode rewriting and other techniques tend to produce high overhead. The mechanisms that are able to keep a low overhead, do so by limiting the expressive power of the tools (as in DeAl [RIS+10]), and by reducing the accuracy of the tools in certain cases (as in DYPER [Rei08]). We have found that, when a solution is expressive and produce efficient tools, it generally requires extensive knowledge of low-level details. It is not clear whether this trade-off is unavoidable. Some results suggest that writing efficient tools without sacrificing generality is possible at some extent [Rei08]. Unfortunately, most approaches that

aim at simplifying the development of dynamic analysis tools, do not address the problem of efficiency with similar strength.

Often, knowledge of low-level details is required The usability of a mechanism is complex to evaluate; the approaches we have presented have not been assessed in this dimension. Instead, these approaches follow empirical evidence that suggests the benefits of some techniques. One of these evidences indicates that using already known languages may ease the development of dynamic analysis tools. Hence, some solutions encourage the use of Java to build profilers and others the utilization of a SQL-like language. Likewise, the empirical evidence suggests that paradigms, such as aspect-oriented programming, and functional programming, should be favored. The problem is that using known-languages and programming paradigms do not automatically reduce the complexity of writing tools. Actually, often the complexity is the result of having to deal with low-level details about the platform, and many techniques still require considerable knowledge of the target platform. In the same way, using aspect-oriented programming can also create additional problems. Indeed, since well-known frameworks, such as AspectJ, cannot be used to develop arbitrary profiler, new frameworks for aspect-oriented programming must be used to implement tools such as profilers; the time required to learn these frameworks hinders the development of dynamic analysis tools.

Although many techniques with low overhead can be implemented using low-level technologies (such as JVMTI), or through modifications to legacy MRTEs; we think that approaches that use low-level APIs may also complicate the developments of tools.

Figure 3.5 depicts the characteristics of each approach, and how close they are to an "ideal" solution. The area is split in four regions that represent in which conditions an approach. The *usability* axes represents how "easy" to use is an approach while the *overhead* axes shows whether the tools built with a solution are efficient. Meanwhile, the size of a circumference shows how generic is an approach. The "values" to locate each approach are obtained by carefully analyzing the respective description in the previous section. Although these are fuzzy values based on our judgment, they are useful to express a fact: every single solution has some limitations that prevent its use as mechanism to create dynamic analysis tools.

3.4 Summary

Defining and using software abstractions is at the core of systems development. Once a new abstraction is defined, the tools used in the development process and to support resource awareness should be modified to make them reflect the new concepts. When these modifications are not done, we argue that a mismatch exists between developer's view and the tooling's view. Alas, implementing this tooling support is tedious and error-prone. In this chapter, we have identified two challenges related to this complexity:

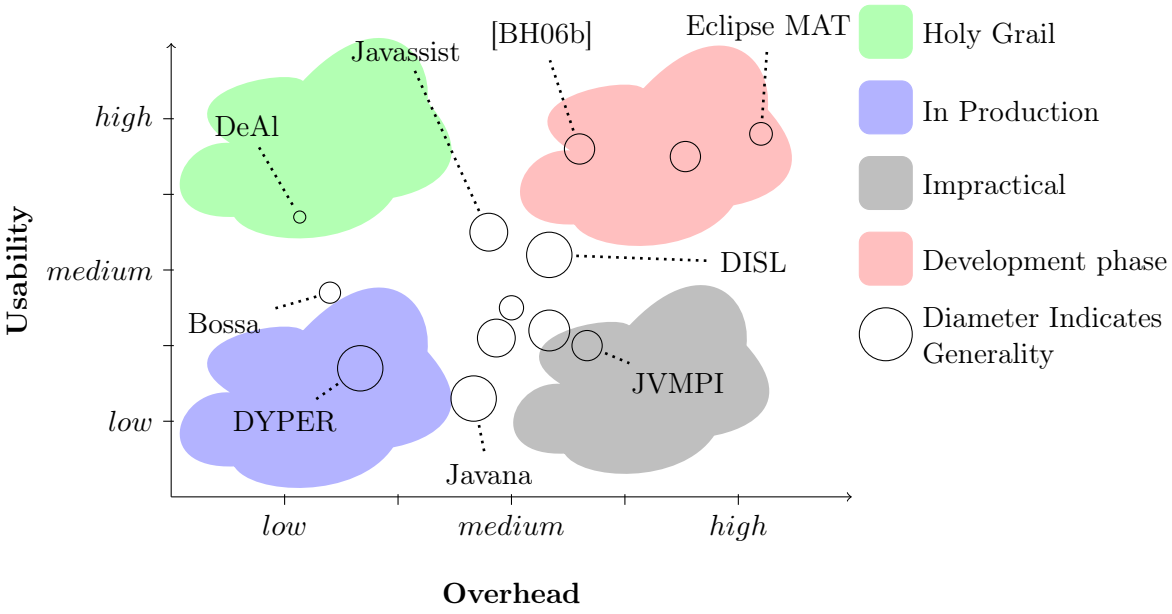


Figure 3.5 – Assessing the relevance of the aforementioned approaches. The location in terms of overhead and usability is given for each approach. Furthermore, the area of each circumference indicates how general the approach is (the larger the better).

- A concrete abstraction may have specific features to consider when implementing tooling support. By adding new requirements to the tools, these features hinder their implementation. At the same time, some features can be useful to drive the behavior of systems that require support for resource awareness.

- Since new software abstractions are defined all the time (for example, using DSLs), it is necessary to ease the task of building their tooling support. For instance, building a resource consumption monitor for a new DSL should be simple enough as to make it affordable for DSL developers.

In reviewing the state of the art of resource management for software abstractions, we realized that severe shortcomings in existing approaches prevent their usage in production environments. With regard to the matter of dealing with abstraction-specific features, the following issues exist:

- Existing solutions offer limited support for resource accounting in component-based systems. Besides targeting only mainstream component models; they induce, in general, high performance overhead.

- Most approaches that target component-based systems use little information about the architecture of the systems. In those cases where such knowledge is used, the goal is solely to improve the accuracy of resource consumption monitoring.

- Mechanisms exist to ease the construction of debuggers for new DSLs. As far as we know, no similar mechanism for profilers have been proposed.

In addition, it is our opinion that current approaches, to build tooling support, lack the required simplicity and maturity. In particular:

- A deep understanding of low-level details of the target MRTE is often required to build tools for resource awareness.
- Memory profilers offer extension capabilities. Unfortunately, such extensions suffer from high overhead because they depend on methods to collect data that offer poor performance.
- Although several approaches are able to produce complex dynamic analysis tools, most of them are not capable of delivering such functionality with low overhead. Interestingly, the mechanisms that do offer a low overhead either require extensive knowledge of the platform or are limited in the kind of analysis they can perform.

Part II

Contributions

To the Reader

In the first two chapters, we discuss the relevance of resource-aware programming in the development of software systems. It is also highlighted that resource awareness requires extensive support from the runtime environment. Severe limitations in existing approaches for resource accounting and reservation prevent the use of resource-aware programming. In particular, we show how the granularity level at which a resource accounting method can be applied is of utmost importance. This is further discussed in Chapter 3, where we highlight the fact that every time a new software abstraction is created, it can be seen as a new granularity level. Since creating abstraction is common in software development, mechanisms to easily develop support for resource-aware programming are needed.

In the rest of this thesis, we present three approaches that contribute to achieve our goal – supporting resource-aware software development. Figure 3.6 depicts a subway map to establish how the new mechanisms we propose are connected to each other, how they contribute to the common goal, and their relationship with other approaches. Although this metaphor is by no means complete, it helps to quickly summarize what is required to achieve resource awareness.

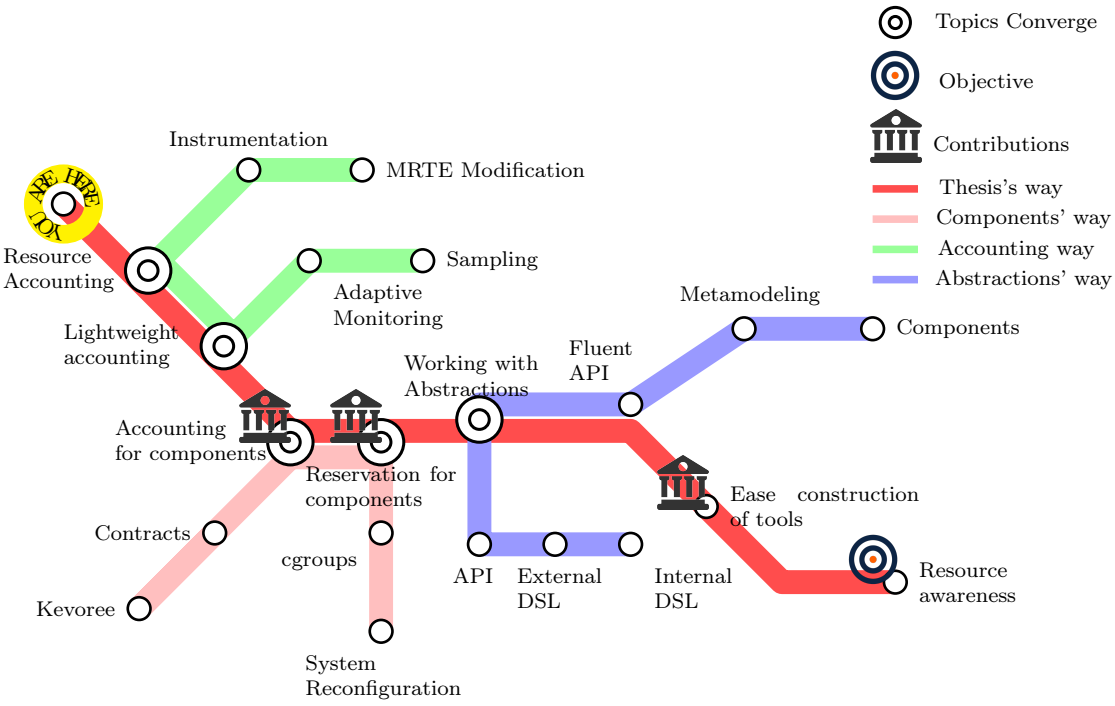


Figure 3.6 – This subway map shows how this research contributes to the state-of-the-art on supporting resource-awareness.

This thesis makes three *stops*.

Resource accounting for components The **first** *stop* is to propose an approach for resource consumption monitoring in component-based systems that run on top of MRTEs (see Chapter 4). Before it, our journey takes us through a way where we collect the basic methods needed to perform resource accounting. Afterwards, at the first *stop*, we reuse such methods to provide *efficient* accounting for components.

Resource reservation for components Resource accounting is the foundation for a **second** *stop* – providing resource reservation for components. It is then when we discuss a methodology to choose at runtime a "*good*" representation of components in the execution platform. This is done by delaying, until the deployment phase, the selection of the low-level method used to guarantee resource reservation; the idea is that only at that point in time we have information that can be useful to reduce the overhead. This serves to guarantee both low performance overhead and per component resource reservation (see Chapter 5).

Ease the constructions of tools The journey continues, and we learn how the mechanisms to define software abstractions lack support for building resource accounting tools. We also realize that this issue makes us remember our first *stop* because components are just a concrete abstraction; and it also brings memories of the second one because new methods to calculate resource consumption may influence the mechanism we choose to represent components at runtime. Then we *stop* for the **third** time; we propose an approach to ease the construction of efficient and customized memory profilers for MRTEs (see Chapter 6).

The validation of each contribution is presented in the corresponding chapter. Different experiments are used to illustrate the characteristics of each solution we present. In addition to the use of the same macro-benchmark across the evaluation of the three contributions, we also assess our work using real-world applications. The results we show in each contribution chapter are relevant to the global goal of the thesis, but also to motivate the subsequent chapters.

Chapter 4

Scapegoat: Spotting the Faulty Component in Reconfigurable Software Systems

---

(*Snot's* dead body lay down on the ground)

Witness - And like every time, *Snot*, he would fade a few shooters, you know. Play it out until the pot's deep. Then he would snatch and run, you see what I'm saying?

McNulty - Every time?

Witness - Couldn't help himself.

McNulty - [...] You let him do this?

Witness - Naw man. We catch him and kick his [...]

McNulty - I got to ask you [...] – if he did that every time – why did you even let him into the games?

Witness - Huh?

McNulty - If *Snot* always stole the money, why did you let him play?

(*Witness* looks at *McNulty* like he's an idiot)

Witness - Got to **(pause)** this America, man.

(The Wire, Season 1, Episode 1)

State of the art monitoring systems [FS04, KHW03, BH06a] collect data about the internal state of applications at runtime, such as the time spent executing a component, the amount of I/O and memory used, and the number of calls to a component. The

overhead that these monitoring systems introduce into applications is high, which makes it unlikely for them to be used in production systems. Results presented in [BHMV09] show that overhead due to fine-grain monitoring systems can be up to a factor of 4.3. The experiments presented in this chapter show that overhead grows with the size of the monitored software. Thus, overhead greatly limits the scalability and usage of monitoring systems.

In this chapter, we address excessive overhead in monitoring approaches by introducing an optimistic adaptive monitoring system - Scapegoat - that provides lightweight global monitoring under normal conditions, and precise and localized monitoring when problems are detected. Although our approach reduces the accumulated amount of overhead in the system, it also introduces a delay in finding the source of a faulty behaviour. Our objective is to provide an acceptable trade-off between the overhead and the delay to identify the source of faulty behaviour in the system.

Our optimistic adaptive monitoring system is based on the following principles:

- **Contract-based resource usage.** The monitoring system follows component-based software engineering principles. Each component is augmented with a contract that specifies their expected or previously calculated resource usage [BJPW99]. The contracts specify how a component uses memory, I/O and CPU resources.

- **Localized just-in-time injection and activation of monitoring probes.** Under normal conditions our monitoring system performs a lightweight global monitoring of the system. When a problem is detected at the global level, our system activates local monitoring probes on specific components in order to identify the source of the faulty behaviour. The probes are specifically synthesized according to the component's contract to limit their overhead. Thus, only the required data are monitored (e.g., only memory usage is monitored when a memory problem is detected), and only when needed.

- **Heuristic-guided search of the problem source.** We use a heuristic to reduce the delay of locating a faulty component while maintaining an acceptable overhead. This heuristic is used to inject and activate monitoring probes on the suspected components. However, overhead and latency in finding the faulty component are greatly impacted by the precision of the heuristic. A heuristic that quickly locates faulty components will reduce both delays and the accumulated overhead of the monitoring system. We propose using Models@run.time techniques in order to build an efficient heuristic.

The evaluation of our optimistic adaptive monitoring system shows that, in comparison to other state-of-the-art approaches, the overhead of the monitoring system is reduced by up to 92.98%. Regarding latency, our heuristic reduces the delay to identify the faulty component when changing from global, lightweight monitoring to localized, just-in-time monitoring. We also present a use case to highlight the possibility of using Scapegoat on a real application, that shows how to automatically find buggy components on a scalable modular web application.

The remainder of this chapter is organized as follows. Section 4.1 motivates our work through a case study which is used to validate the approach. A brief description of Kevoree, a platform for distributed component-based software development, is presented in Section 4.2. Section 4.3 provides an overview of the Scapegoat framework. It highlights how the component contracts are specified, how monitoring probes are injected and activated on-demand, how the Scapegoat framework enables the definition of heuristics to detect faulty components without activating all the probes, and how we benefit from Models@run.time to build efficient heuristics. Section 4.4 evaluates the approach through a comparison of detection precision and detection speed with other approaches. Section 4.5 presents a use case based on an online web application[1] that leverages software diversity for safety and security purposes. Finally, Section 4.7 discusses the approach and presents the conclusions of this chapter.

4.1 Motivating example: open-world scenario

During a dangerous event, many firefighters are present and need to collaborate to achieve common goals. Firefighters have to coordinate among themselves and commanding officers need to have an accurate real-time view of the system.

The Daum project[2] provides a software application that supports firefighters in these situations. The application runs on devices with limited computational resources because it must be mobile and taken on-site. It provides numerous services for firefighters depending on their role in the crisis. In this chapter, we focus on the two following roles:

- A collaborative functionality that allows commanding officers to follow and edit tactical operations. The firefighters' equipment include communicating sensors that report on their current conditions.

- A drone control system which automatically launches a drone equipped with sensors and a camera to provide a different point-of-view on the current situation.

As is common in many software applications, the firefighter application may have a potentially infinite number of configurations. These configurations depend on the number of firefighters involved, the type of crisis, the available devices and equipment, among other parameters. Thus, it is generally not possible to test all configurations to guarantee that the software will always function properly. Consequently, instead of testing all configurations, there is a need to monitor the software's execution to detect faulty behaviours and prevent system crashes. However, fine-grained monitoring of the application can have excessive overhead that makes it unsuitable with the application and the devices used in our example. Thus, there is a need for an accurate monitoring system that can find faulty components while reducing overhead.

[1] `http://cloud.diversify-project.eu/`
[2] `https://github.com/daumproject`

The Daum project has implemented the firefighter application using a Component Based Software Architecture. The application makes extensive use of the Kevoree[3] component model and runtime presented in chapter 3.

4.2 Kevoree Component Model

Kevoree [4] is an example of framework for building distributed and reconfigurable applications. It is built around a component model, and it leverages the Models@run.time approach to ease the construction of reconfigurable systems.

Built on top of dynamic component frameworks, Models@run.time denote model-driven approaches that aim at taming the complexity of dynamic adaptation. It basically pushes the idea of reflection [MBNJ09b] one step further by considering the reflection-layer as a real model: "something simpler, safer or cheaper than reality to avoid the complexity, danger and irreversibility of reality". In practice, component-based and service-based platforms offer reflection APIs that allow instrospecting the application (e.g., which components and bindings are currently in place in the system) and dynamic adaptation (e.g., changing the current components and bindings). While some of these platforms offer rollback mechanisms to recover after an erroneous adaptation [LLC10], the purpose of Models@run.time is to prevent the system from actually enacting an erroneous adaptation. In other words, the "model at runtime" is a reflection model that can be decoupled from the application (for reasoning, validation, and simulation purposes) and then automatically resynchronized. This model can not only manage the application's structural information (i.e., the architecture), but can also be populated with behavioral information from the specification or the runtime monitoring data.

Kevoree provides multiple concepts that are used to create a distributed application that allows dynamic adaptation. The *Node* concept is used to model the infrastructure topology and the *Group* concept is used to model the semantics of inter-node communication, particularly when synchronizing the reflection model among nodes. Kevoree includes a *Channel* concept to allow for different communication semantics between remote *Components* deployed on heterogeneous nodes. All Kevoree concepts (*Component*, *Channel*, *Node*, *Group*) obey the object type design pattern [JW97] in order to separate deployment artifacts from running artifacts.

Kevoree supports multiple execution platforms (e.g., Java, Android, MiniCloud, FreeBSD, Arduino). For each target platform it provides a specific runtime container. Moreover, Kevoree comes with a set of tools for building dynamic applications (a graphical editor to visualize and edit configurations, a textual language to express reconfigurations, several checkers to valid configurations).

As a result, Kevoree provides a promising environment by facilitating the implementation of dynamically reconfigurable applications in the context of an open-world environment. Because our goal is to design and implement an adaptive monitoring

[3] `http://www.kevoree.org`
[4] `http://kevoree.org/`

system, the introspection and the dynamic reconfiguration facilities offered by Kevoree suit the needs of the ScapeGoat framework.

4.3 The Scapegoat framework

Our optimistic adaptive monitoring system extends the Kevoree platform with the following principles: i) component contracts that define per-component resource usage, ii) localized and just-in-time injection and activation of monitoring probes, iii) heuristic-guided faulty component detection. The following subsections present an overview of these three principles in action.

4.3.1 Specifying component contracts

In Scapegoat, we follow the contract-aware component classification [BJPW99], which applies B. Meyer's Design-by-Contract principles [Mey92] to components. In fact, Scapegoat provides Kevoree with *Quality of Service* contract extensions that specify the worst-case values of the resources the component uses. The resources specified are memory, CPU, I/O and the time to service a request. The exact semantic of a contract in Scapegoat is: *the component will consume at most X resource if it receives at most N requests on its provided ports.*

For example, for a simple Web server component we can define a contract on the number of instructions per second it may execute [BH06a] and the maximum amount of memory it can consume. The number of messages can be specified per component or per component-port. In this way, the information can be used to tune the usage of the component roughly or detailedly. An example is shown in Listing 4.1.[5] This contract extension follows the component interface principle [AH01], and allows us to detect if the problem comes from the component implementation or from a component interaction. That is, we can distinguish between a component that is using excessive resources because it is faulty, or because other components are calling it excessively.

4.3.2 An adaptive monitoring framework within the container

Scapegoat provides a monitoring framework that adapts its overhead to current execution conditions and leverages the architectural information provided by Kevoree to guide the search for faulty components. The monitoring mechanism is mainly injected within the component container.

Each Kevoree node/container is in charge of managing the component's execution and adaptation. Following the Models@run.time approach, each node can be sent a new architecture model that corresponds to a system evolution. In this case, the node compares its current configuration with the configuration required by the new architectural model and computes the list of individual adaptations it must perform. Among these adaptations, the node is in charge of downloading all the component packages and their

[5] Examples of contract for the architecture presented in section 4.1 can be found at `http://goo.gl/uCZ2Mv`.

```
add node0.WsServer650 : WsServer

//Specify that this component can use 2580323 CPU
//instructions per second
set WsServer650.cpu_wall_time = 2580323 intr/sec

//Specify that this component can consume a maximum of 15000
//bytes of memory
set WsServer650.memory_max_size = 15000 bytes

//Specify that the contract is guaranteed under the assumption that
//we do not receive more than 10k messages on the component and
//10k messages on the port named service
//(this component has only one port)
set WsServer650.throughput_all_ports = 10000 msg/sec
set WsServer650.throughput_ports.service = 10000 msg/sec
```

Listing 4.1 – Component contract specification example

dependencies, and loading them into memory. During this process, Scapegoat provides the existing container with (i) checks to verify that the system has enough resources to manage the new component, (ii) instrumentation for the component's classes in order to add bytecode for the monitoring probes, and iii) communication with a native agent that provide information about heap utilization. Scapegoat uses the components' contracts to check if the new configuration will not exceed the amount of resources available on the device. It also instruments the components' bytecode to monitor object creation (to compute memory usage), to compute each statement (for calculating CPU usage), and to monitor calls to classes that wrap I/O access such as the network or file-system. In addition, Scapegoat provides a mechanism to explore the Java heap and to account for memory consumption with an alternative mechanism.

We provide several instrumentation levels that vary in the information they obtain and in the degree they impact the application's performance:

- **Global monitoring** does not instrument any components, it simply uses information provided directly by the JVM.
- **Memory instrumentation** or memory accounting, which monitors the components' memory usage.
- **Instruction instrumentation** or instruction accounting, which monitors the number of instructions executed by the components.
- **Memory and instruction instrumentation**, which monitors both memory usage and the number of instructions executed.

Probes are synthesized according to the components' contracts. For example, a component whose contract does not specify I/O usage will not be instrumented for I/O resource monitoring. All probes can be dynamically activated or deactivated. Note that due to a technical limitation, one of the two probes implemented to check memory consumption must be always activated. This memory consumption probes, based on

bytecode instrumentation must, remain activated to guarantee that all memory usage is properly accounted for, from the component's creation to the component's destruction. Indeed, deactivating this memory probes would cause object allocations to remain unaccounted for. However, probes for CPU, I/O usage and the second probe for memory can be activated on-demand to check for component contract compliance.

We propose two different mechanisms to deal with memory consumption. The first mechanism is based on bytecode instrumentation and accounts for each object created. As mentioned previously, this mechanism cannot be disabled. The second mechanism is a just-in-time exploration of the JVM heap, performed on demand. These two mechanisms differ in i) when the computation to account for consumption is done, ii) how intensive it is, and iii) in the way the objects are accounted for. Computations in the first mechanism are spread throughout the execution of the application, short and lightweight operations are executed every time a new object instance is created or destroyed. Objects are always accounted to the component that creates them. Computations in the second mechanism occur only on demand but are intensive because they involve traversing the graph of living objects in the heap. The accounting policy follows the paradigm of assigning objects to the component that is holding them and, if an object is reachable from more than one component, it is accounted to either one randomly, as suggested in [PRW03, GTM+09]; we call this second mechanism **Heap Exploration**.

We minimize the overhead of the monitoring system by activating selected probes only when a problem is detected at the global level. We estimate the most likely faulty components and then activate the relevant monitoring probes. Following this technique, we only activate fine-grain monitoring on components suspected of misbehavior. After monitoring the subset of suspected components, if any of them are found to be the source of the problem, the monitoring system terminates. However, if the subset of components is determined to be healthy, the system starts monitoring the next most likely faulty subset. This occurs until the faulty component is found. If no components are found to be faulty, we fallback to global monitoring. If the problem still exists the entire process is restarted. This can occur in cases where, for example, the faulty behavior is transient or inconsistent. The monitoring mechanism implemented in Scapegoat is summarized in Listing 4.2.

As a result, we consider that applications are executing under the conditions of one of the following monitoring modes:

- **No monitoring.** The software is executed without any monitoring probes or modifications.
- **Global monitoring.** Only global resource usage is being monitored, such as the CPU and memory usage at the Java Virtual Machine (JVM) level.
- **Full monitoring.** All components are being monitored for all types of resource usage. This is equivalent to current state-of-the-art approaches.
- **Localized monitoring.** Only a subset of the components are monitored.

```
1  monitor(C: Set<Component>, heuristic : Set<Component>→Set<Component>)
2    init memory probes (c | c ∈ C ∧ c.memory_contract ≠ ∅)
3    while container is running
4      wait violation in global monitoring
5      checked = ∅
6      faulty = ∅
7      while checked ≠ C ∧ faulty = ∅
8        subsetToCheck = heuristic ( C \ checked )
9        instrument for adding probes ( subsetToCheck )
10       faulty = fine−grain monitoring( subsetToCheck )
11       instrument for removing probes ( subsetToCheck )
12       checked = checked ∪ subsetToCheck
13     if faulty ≠ ∅
14       adapt the system (faulty, C)
15
16 fine−grain monitoring( C : Set<Component> )
17   wait few milliseconds // to obtain good information
18   faulty = {c | c ∈ C ∧ c.consumption > c.contract}
19   return faulty
```

Listing 4.2 – The main monitoring loop implemented in Scapegoat

- **Adaptive monitoring.** The monitoring system changes from Global monitoring to Full or Localized monitoring if a faulty behaviour is detected.

For the rest of this chapter we use the term **all components** for the adaptive monitoring policy that indicates that the system changes from *global monitoring* mode to *full monitoring* mode if and when a faulty behaviour is detected.

4.3.2.1 Scapegoat's architecture

The Scapegoat framework is built using the Kevoree component framework. Scapegoat extends Kevoree by providing a new Node Type and three new Component Types:

- **Monitored Node.** Handles the admission of new components by storing information about resource availability. Before admission, it checks the security policies and registers components with a contract in the monitoring framework. Moreover, it intercepts and wraps class loading mechanisms to record a component type's loaded classes. Such information is used later to (de)activate the probes.

- **Monitoring Component.** This component type is in charge of checking component contracts. Basically, it implements a complex variant of the algorithm in Listing 4.2. It communicates with other components to identify suspected components.

- **Ranking Component.** This is an abstract Component Type; therefore it is user customizable. It is in charge of implementing the heuristic that ranks the components from the most likely to be faulty to the least likely.

- **Adaptation component.** This component type is in charge of dealing with the adaptation of the application when a contract violation is detected. It is also a customizable component. The adaptation strategy whenever a faulty component is

discovered is out os scope of this thesis. Nevertheless, several strategies may be implemented in Scapegoat, such as removing faulty components or slowing down communication between components when the failure is due to a violation in the way one component is using another.

4.3.2.2 Extensibility of the Scapegoat Framework

The Scapegoat framework has been built with the idea of being as generic as possible, thus supporting various extensions and specializations. In this section we discuss the extension points provided by the Scapegoat framework.

Heuristics used to rank suspected faulty components can be highly specialized and, as we show in section 4.4, have a remarkable impact on the behavior of Scapegoat. A new heuristic is created by defining a component that implements an interface to provide a ranking of the suspected components. To do so, a context is sent with each ranking request on this component. This context is composed of three elements, i) a model that describes the components and links of the deployed application, ii) a history that contains all the models that have been deployed on the platform, and iii) a history of failures composed of metadata regarding what components have failed as well as why and when it happened. In this thesis, we present three heuristics. The first heuristic is proposed in section 4.3.3 and shows how we can leverage the Models@Run.time paradigm to guide the framework in finding the component that is behaving abnormally. Due to their simplicity, the other two heuristics are presented in section 4.4 where we use them to evaluate the behavior of Scapegoat.

The mechanism for creating new heuristics is based on the strategy design pattern. Figures 4.1 and 4.2 illustrates this extension point.

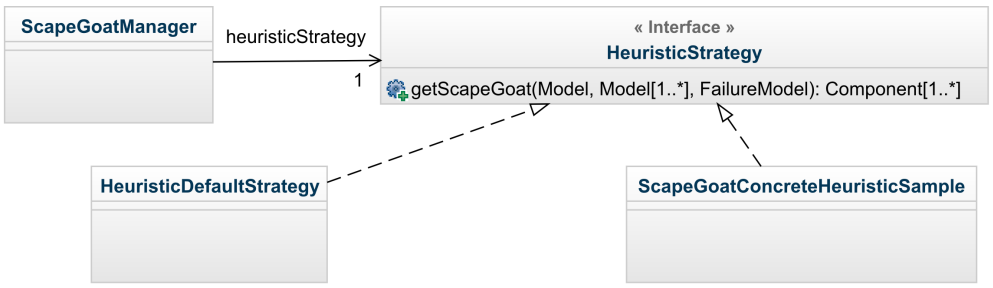


Figure 4.1 – Heuristic extension point in Scapegoat. This illustrates the class diagram.

A second extensible aspect of the framework is the admission control system. The framework provides an API to hook user-defined actions when new components are submitted for deployment. Basic data describing the execution platform in terms of resource availability, information about the already deployed components and the new component's contract are sent to the user-defined admission control system. On each request, the admission control system has to accept or refuse the new component. We

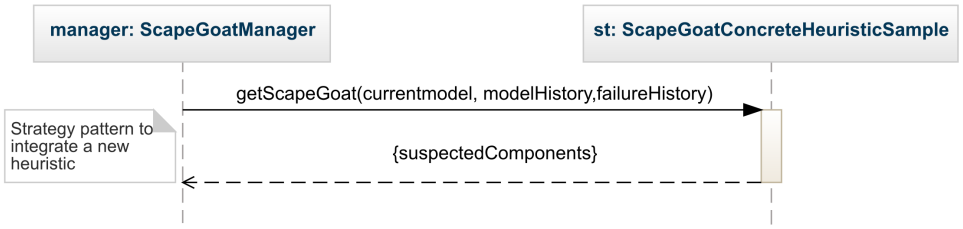


Figure 4.2 – A sequence diagram showing how the extension point to define heuristics in Scapegoat is used.

are using an approach that checks the theoretical availability of resources whenever a component is deployed, and accepts the new component if the contract can fit in the remaining available resources. Scapegoat is meant to support other policies as, for instance, overcommitment.

A last element that can be specialized to user needs is the contracts semantic. In section 4.3.1 we describe how we interpret the contract in this work. However, it is possible to define other contract semantics, for instance, accepting values that are closed to the limit defined in the contract, or using fuzzy values instead of sharp values. It is worth noting that modifying the semantic of the contract would likely involved redefining the domain-specific language to describe contract and also modifying the admission control system.

4.3.2.3 Implementation strategy

Scapegoat aims at minimizing monitoring overhead when the framework is monitoring the global behavior of the JVM. To achieve this, Scapegoat uses as few probes as possible when executing in global monitoring mode. Only when it is necessary, the framework activates the required probes. This features are implemented in the framework in three modules that are in charge of different concerns: a module to activate/deactivate the probes, a module to collect the resource usage, and a module to compute what components should be carefully monitored. In this section we focus on the modules for activating/deactivating probes and for collecting information of resource usage because they required considerable engineering effort. Notice, however, that this module is executed on demand when the framework already decides the monitoring mode to use and what components to monitor.

Module to activate/deactivate probes In Scapegoat we use bytecode instrumentation to perform localized monitoring. However, instead of doing as previous approaches that manipulate the bytecode that defines components just when the component's code is executed for the first time, we modify the bytecode many times during components' life. Every time the monitoring mode is changed we either activate or deactivate the probes by simply inserting them in the bytecode or by removing them.

Implementing this mechanism at per-component basis requires knowing all the classes that have been loaded for a component. This information is kept using a dictionary in which we treat a component's id as a key and a set of class names as a value. The dictionary is filled using the *traditional* classloader mechanism of Java. In short, when a class is loaded on behalf of a component, we detect the class name and the thread that is loading the class. Using the thread's id we are able to identify the component because we use special naming conventions for each thread executing the initial code of a component. When probes are activated/deactivated on a component, iterating over the set of class names allows the re-instrumentation of each involved class.

The probes perform two actions: collecting data about the local usage of resources (e.g., objects recently allocated, instructions executed in the current basic block, bytes sent through the network), and notifying to the resource consumption monitor about the collected data. Some data we collect is computed statically when the bytecode is loaded. This includes the size of each basic block and the size of each object allocated when the size of each instance of the class is already known. Other data, such as bytes sent through the network or the size of allocated arrays, can only by collected dynamically when the code is running. To notify about the collected data we use simple method calls to a proxy class in charge of forwarding the data to the monitoring module. Probes to detect CPU consumption are inserted at the end of each basic block. These probes collect the size in number of instructions of its container basic block. Probes for IO throughput and network bandwidth are added in a few selected method defined in classes of the Java Development Kit (JDK). These probes take the needed information from local variables (e.g., number of bytes) and call the proxy class.

Our implementation, which is built using the ASM library [6] for bytecode manipulation and a Java agent to get access to and transform the classes, is based on previous approaches to deal with resource accounting and profiling in Java [Bin06, BH06a, CvE98]. As in previous approaches, we compute the length of each basic block to count the number of executed instructions and we try to keep a cache of *known* methods with a single basic block. Moreover, we compute the size of each object once it is allocated and we use *weak* references instead of *finalizers* to deal with deallocation.

Module to collect information regarding resource usage In Scapegoat, there are two mechanisms to collect information about how components consume resources.

The first mechanism is able to capture the usage of CPU, IO throughput, network bandwidth and memory. Every time a probe that was inserted in the code of a component is executed, the proxy class forwards the local resource usage to the module in charge of collecting the resource usage. Along with the local resource usage, probes also notify the id of the components consuming resource. Such data is then used to aggregate the global consumption of each component. It is worth noting that, when this first mechanism is used to collect memory consumption, on object is always accounted as consumed for the component responsible for its initial allocation. In short, no matter whether the initial component C that allocates the object no longer held a reference

[6] asm.ow2.org

to an object O, as long as O remains in memory, C is accounted for its consumption. Moreover, as was already mentioned, using this mechanism is not possible to deactivate the probes related to memory consumption.

On the contrary, the second mechanism is only useful to collect information about memory consumption. The advantages of this method are: we can leverage the proposed optimistic monitoring because it executes only on demand, and it has no impact on the number of objects allocated in memory because no *weak references* are used. However, in this method an object O is consumed not for the component that allocates it but for those components that held references to it. As a consequence, in certain occasions the framework states that an object is being consumed for many components at the same time. We built this solution on top of JVMTI by implementing the algorithm proposed in [PRW03, GTM+09], with the main difference being that our solution works without modifying the garbage collector. In summary, this algorithm simply try to find those objects that are reachable from the references of each component. It does so by traversing the graph of live object using as the component instance and its threads as roots of the traversal. Since our approach does not require a modification to the garbage collector, it is portable and works with different garbage collector implementations.

4.3.3 Leveraging Models@run.time to build an efficient monitoring framework

As presented in section 4.3.2, our approach offers a dynamic way to activate and deactivate fine-grain localized monitoring. We use a heuristic to determine which components are more likely to be faulty. Suspected components are the first to be monitored.

Our framework supports the definition of different heuristics, which can be application or domain-specific. In this chapter we propose a heuristic that leverages the use of the Models@run.time approach to infer the faulty components. The heuristic is based on the assumption that the cause of newly detected misbehavior in an application is likely to come from the most recent changes in the application. This can be better understood as follows:

- recently added or updated components are more likely to be the source of a faulty behaviour;
- components that directly interact with recently added or updated components are also suspected.

We argue that when a problem is detected it is probable that recent changes have led to this problem, or else, it would have likely occurred earlier. If recently changed components are monitored and determined to be healthy, it is probable that the problem comes from direct interactions with those components. Indeed, changes to interactions can reveal dormant issues with the components. The algorithm used for ranking the components is presented in more detail in Listing 4.3. In practice, we leverage the architectural-based history of evolutions of the application, which is provided by the Models@run.time approach.

```
1  ranker() : list <Component>
2    // used to avoid adding duplicated elements to the list
3    visited = ∅
4    // this list will contain the result of calling the routine
5    ranking = {}
6    for each model M ∈ History
7      // adding components that were added in this model
8      N = {c | c was added in M}
9      ranking.add N\visited
10     visited = visited ∪ N
11     // finding neighbors
12     Neighbors = ⋃_{c∈N} c.neighbors
13     SortedNeighbors = sort (Neighbors \ visited, History)
14     // adding neighbors
15     ranking.add SortedNeighbors
16     visited = visited ∪ Neighbors
17   // return the built ranking
18   return ranking
19
20 // this routine recursively sort a set of components using the following criteria :
21 // components are sorted by the timestamp that indicates when they were installed
22 private sort (S : Set<Component>, H : History) : list<Component>
23   r = {}
24   if S ≠ ∅
25     choose b | b ∈ S ∧ b is newer with respect to H than any other element in S
26     r.add b, sort (S\{b}, H)
27   return r
```

Listing 4.3 – The ranking algorithm (uses the model history for ranking).

Listing 4.3 shows two routines, but only routine *ranker* is public. It can be called by the monitoring system when it is necessary to figure out in what order components must be carefully monitored. After initializing an empty list which will hold the rank, the algorithm starts to iterate in line 6 over the history of models that have been installed in the system. As mentioned, this history contains a sorted set of models that describe what components have been installed in the system. Within each iteration, the algorithm first computes in line 8 the set of components that were installed at such a point in time. Afterwards, these components are added to the result. The next step, executed at lines 12 and 13, is finding those components that are directly connected to components that were added to the application at this point in time. Finally, these *neighbors* are added to the rank after being sorted. Routine *sort* simply sorts a set of components using as criteria the time at which components where installed in the system.

4.4 Scapegoat Performance Evaluation

In this section we present a first series of experiments and discuss the usability of our approach. We formulate the following questions to assess the quality and the efficiency of Scapegoat:

- **What is the impact of the various levels of instrumentation on the**

application? Our approach assumes high overhead for full monitoring and low overhead for a lightweight global monitoring system. The experiments presented in section 4.4.2 show the overhead for each instrumentation level.

- **What is the performance cost of using instrumentation-based and heap-exploration-based memory monitoring?** Since both mechanisms have by design different features, the experiments in section 4.4.2 show the overhead each mechanism produces.

- **Does our adaptive monitoring approach have better performance than state-of-the-art monitoring solutions?** The experiment presented in section 4.4.3 highlights the performances benefits of our approach considering a real-world scenario.

- **What is the impact of using a heuristic in our adaptive monitoring approach?** The experiment presented in section 4.4.4 highlights the impact of the application and component sizes, and the need of a good heuristic to quickly identify faulty components.

The efficiency of our monitoring solution is evaluated on two dimensions: the overhead on the system and the delay to detect failures. We show there is a trade-off between the two dimensions and that Scapegoat provides a valuable solution that increases the delay to detect a faulty component but reduces accumulated overhead. This evaluation has been conducted on a Cyber Physical System case study. It corresponds to a concrete application that leverage the Kevoree framework for dyamic adaptation purpose.

We have built several use cases based on a template application from our motivating example in section 4.1. We reused an open-source crisis-management application for firefighters that has been built with Kevoree components. We use two functionalities of the crisis-management application. The first one is for managing firefighters. The equipment given to each firefighter contains a set of sensors that provides data for the firefighter's current location, his heartbeat, his body temperature, his acceleration movements, the environmental temperature, and the concentration of toxic gases. These data are collected and displayed in the crisis-management application, which provides a global-view of the situation. The second functionality uses drones to capture real-time video from an advantageous point-of-view.

Figure 4.3 shows the set of components that are involved in our use-case, including components for firefighters, drones and the crisis-management application[7]. The components in the crisis-management application are used in our experiments, but the physical devices (drones and sensors) are simulated through the use of mock components. The application presents two components: the first one is a web browser that shows information about each firefighter in the terrain, and the second one allows to watch the video being recorded by any drone in the field. A Redis database is used to store the data that is consumed for the application's GUI.

[7]More information about these components is given in `http://goo.gl/x64wHG`

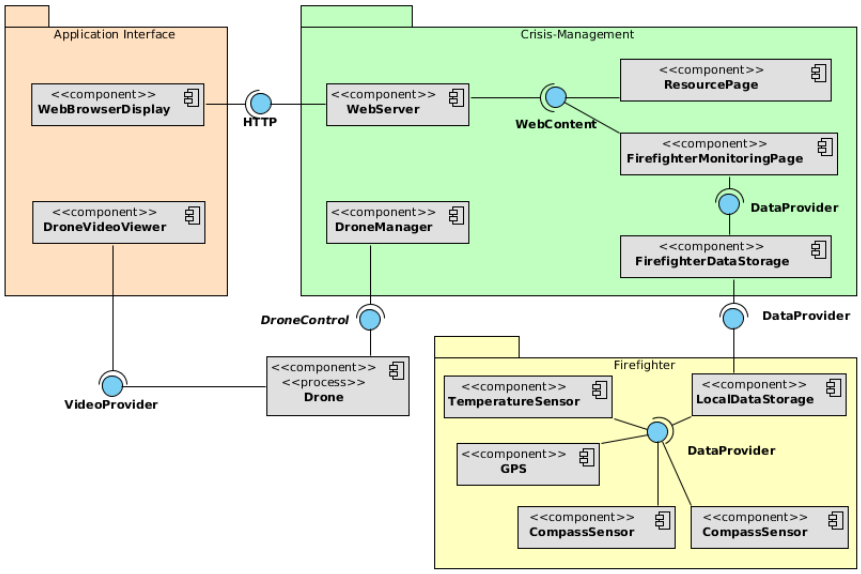


Figure 4.3 – The component configuration for our crisis-management use-case.

Every use case we present extends the crisis-management base application by any one of the following possibilities: adding new or redundant components, adding external Java applications with wrapper components (e.g., Weka, DaCapo), or modifying existing components (e.g., to introduce a fault into them). Using this template in the experiments allow us to measure the behavior of our proposal in a more realistic environment where many components with different features co-exist.

4.4.1 Measurement Methodology

To obtain comparable and reproducible results, we used the same hardware across all experiments: a laptop with a 2.90GHz Intel(R) i7-3520M processor, running Fedora 19 with a 64 bit kernel and 8GiB of system memory. We used the HotSpot Java Virtual Machine version 1.7.0_67, and Kevoree framework version 5.0.1. Each measurement presented in the experiment is the average of ten different runs under the same conditions.

The evaluation of our approach is tightly coupled with the quality of the resource consumption contracts attached to each component. We built the contracts following classic profiling techniques. The contracts were built by performing several runs of our use cases, without inserting any faulty components into the execution. Firstly, we executed the use cases in an environment with global monitoring activated to get information for the global contract. Secondly, per-component contracts were created by running the use cases in an environment with full monitoring.

4.4.2 Overhead of the instrumentation solution

Our first experiment compares the various instrumentation levels to show the overhead of each one. In this section, *Memory instrumentation* refers to the technique for accounting memory which leverage bytecode instrumentation, while *Heap Exploration* refers to the memory accouting technique which leverage on-demand heap exploration. In this experiment, we compare the following instrumentation levels: *No monitoring*, *Global monitoring*, *Memory instrumentation*, *Instructions instrumentation*, *Memory and instructions instrumentation* (i.e., Full monitoring). We also evaluate the impact on performance of the two fine-grain memory monitoring approaches we proposed: instrumentation-based and heap-dump-based.

In this set of experiments we used the DaCapo 2006 benchmark suite [Bea06]. We developed a Kevoree component to execute this benchmark [8]. The container was configured to use full monitoring and the parameters in the contract are upper bounds of the real consumption[9].

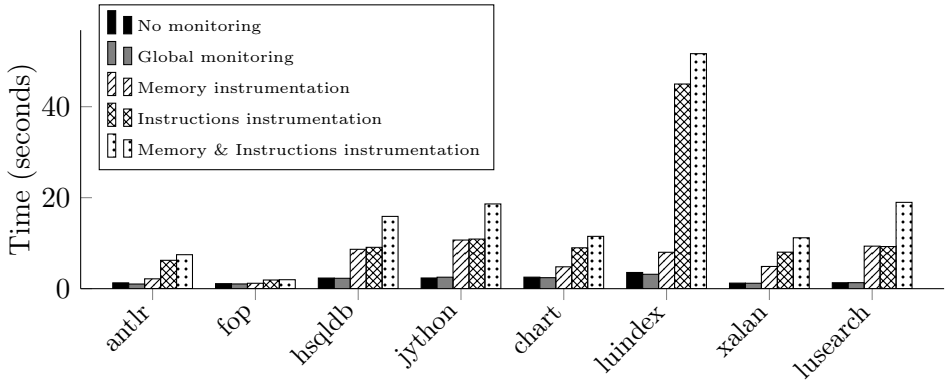


Figure 4.4 – Execution time for tests using the DaCapo Benchmark

Figure 4.4 shows the execution time of several DaCapo tests under different scenarios when only instrumentation is used to provide fine-grain monitoring. First, we wish to highlight that *Global monitoring* introduces no overhead compared with the *No monitoring* mode. Second, the overhead due to memory accounting is lower than the overhead due to instruction accounting. This is very important because, as we described in section 4.3.2, memory probes cannot be deactivated dynamically.

To perform the comparison, we evaluate the overhead produced for each monitoring mode. We calculated the overhead as:

$$overhead = \frac{WithInstrumentation}{GlobalMonitoring}$$

The average overhead due to instruction accounting is 5.62, while the value for memory accounting depends on the monitoring mechanism. If bytecode instrumentation

[8] `http://goo.gl/V5T6De`

[9] Scripts are generated from those available at `http://goo.gl/FR8LC7`.

is used, the average overhead is 3.29 which is close to the values reported in [BHMV09]. In the case of instruction accounting, these values are not as good as the values reported in [BHMV09]; because they obtain a better value between 3.2 and 4.3 for instructions accounting. The performance difference comes from a specific optimization that we chose not to apply. The optimization provides fast access to the execution context by adding a new parameter to each method. Nevertheless, this solution needs to keep a version of the method without the new parameter because native calls cannot be instrumented like that. We decided to avoid such an optimization because duplication of methods increases the size of the applications, and with it, the memory used by the heap. In short, our solution can reach similar values if we include the mentioned optimization, but at the cost of using more memory. On the other hand, the values we report are far lower than the values reported in [BHMV09] for hprof. Hence, we consider that our solution is comparable to state of the art approaches in the literature.

In Figure 4.5 we compare the execution time of the same benchmarks but using different memory monitoring approaches. This comparison is important because, as explained in section 4.3.2, the two approaches have different CPU footprint. These are controlled experiments where, in order to stress the technique, we demand the execution of a *heap exploration* step every two seconds, which is not the expected usage pattern. On the contrary, the memory instrumentation technique is executed with the expected usage pattern. In comparison to using memory instrumentation where the average execution time is 3.29, the average overhead in execution time decreases to 1.79 if the *Heap Exploration* monitoring mechanism is used. This value is better than the value reported in [BHMV09]. These results suggest that this technique has less impact on the behavior of applications being monitored.

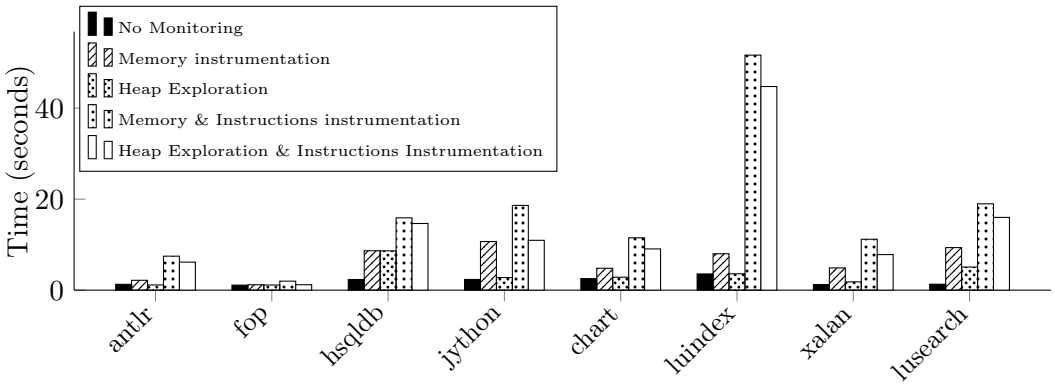


Figure 4.5 – Comparison of execution time for tests using two different memory monitoring techniques

The results of our experiment shown in Figures 4.4 and 4.5 demonstrate the extensive impact of the *Full monitoring* mode, which uses either *Memory instrumentation* or *CPU instrumentation*, has on the application. Thus, our *Adaptive monitoring* mode, which uses *Global monitoring* and switches to *Full monitoring* or *localized monitoring*, has the potential to reduce this accumulated overhead due to the fact that *Global monitoring*

has no appreciable overhead.

4.4.3 Overhead of Adaptive Monitoring vs Full Monitoring

The previous experiment highlights the potential of using *Adaptive monitoring*. However, switching from *Global monitoring* to either *Full* or *Localized monitoring* introduces an additional overhead due to having to instrument components and activate monitoring probes. Our second experiment compares the overhead introduced by the adaptive monitoring with the overhead of *Full monitoring* as used in state-of-the-art monitoring approaches.

Table 4.1 shows the tests we built for the experiment. We developed the tests by extending the template application. Faults were introduced by modifying an existing component to break compliance with its resource consumption contract. We reproduce each execution repetitively; thus, the faulty behaviour is triggered many times during the execution of the application. The application is not restarted.

Table 4.1 – Features of use cases.

Test Name	Monitored Resource	Faulty Resource	Heuristic	External Task
UC1	CPU, Memory	CPU	number of failures	Weka, training neural network
UC2	CPU, Memory	CPU	number of failures	dacapo, antlr
UC3	CPU, Memory	CPU	number of failures	dacapo, chart
UC4	CPU	CPU	number of failures	dacapo, xalan
UC5	CPU, Memory	CPU	less number of failures first	dacapo, chart
UC6	Memory	CPU	number of failures	Weka, training neural network

Figure 4.6 shows the execution time of running the use cases with different scenarios. Each scenario uses a specific monitoring policy (*Full monitoring*, *Adaptive monitoring with All Components*, *Adaptive monitoring with Localized monitoring*, *Global monitoring*). All these scenarios were executed with the heap explorer memory monitoring policy. This Figure shows that the overhead differences between *Full monitoring* and *Adaptive monitoring with All Components* is clearly impacted by scenarios that cause the system to transition too frequently between a lightweight Global and a fine-grain *Adaptive monitoring*. Such is the case for use cases UC3 and UC4 because the faulty component is inserted and never removed. Using *Adaptive monitoring* is beneficial if the overhead of *Global monitoring* plus the overhead of switching back and forth to *All Components monitoring* is less than the overhead of the *Full monitoring* for the same

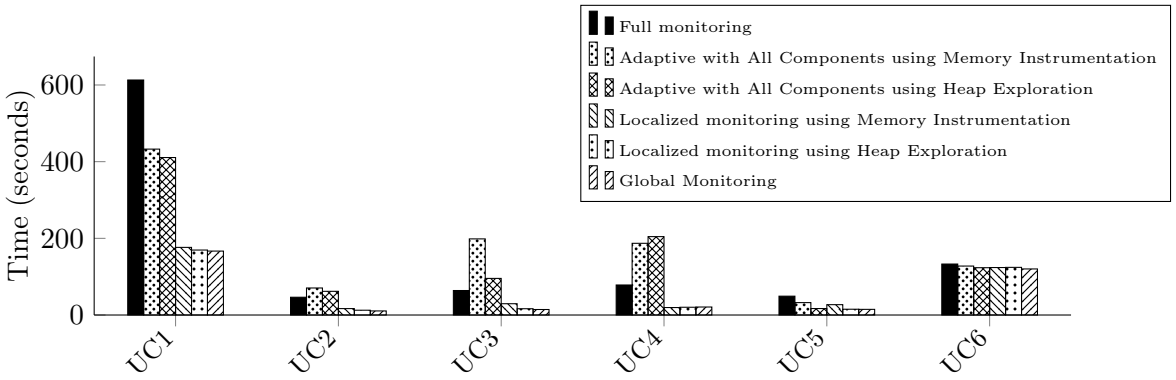


Figure 4.6 – Execution time for some use cases under different monitoring policies.

execution period. If the application switches between monitoring modes too often then the benefits of adaptive monitoring are lost.

The overhead of switching from *Global monitoring* to *full components* or *Localized monitoring* comes from the fact that the framework must reload and instrument classes to activate the monitoring probes. Therefore, using *Localized monitoring* reduces the number of classes that must be reloaded. This is shown in the third use-case of Figure 4.6, which uses a heuristic based on the number of failures. Because we execute the faulty component many times, the heuristic is able to select, monitor and identify the faulty component quickly. This reduces overhead by 93%. We use the following equation to calculate overhead:

$$Gain = 100 - \frac{OurApproach - GlobalMonitoring}{FullMonitoring - GlobalMonitoring} * 100$$

We also evaluate the execution time for each use case using the instrumentation-based memory monitoring mode. The average gain in that case is 81.49% and, as shown in previous section, in average it behaves worse than the *Heap Exploration* mechanism. However, it is worth noting that the difference between using memory monitoring based on instrumentation and heap exploration is less remarkable than in the previous experiment. Observe how in test UC4, using a combination of heap exploration and adaptive monitoring with all components behaves worse than using plain instrumentation-based memory monitoring. In this particular test, activating and deactivating the monitoring probes dominate the execution time. Alas, adding a heap exploration step right after the probes are activated, just add some extra overhead. On the contrary, there is no additional step executed when we use instrumentation to measure the memory usage. Apparently, what matter when the all components strategy is guiding the adaptive monitoring is the ratio among the amount of allocations performed by components and the size of those components.

4.4.4 Overhead from switching monitoring modes, and the need of a good heuristic

As we explain in the previous experiments, even if using *Localized monitoring* is able to reduce the overhead of the monitoring system, the switch between *Global* and *Localized monitoring* introduces additional overhead. If this overhead is too high, the benefits of adaptive monitoring are lost.

In this experiment we show the impact of the application's size, in terms of number of components, and the impact of the component's size, in terms of number of classes, on adaptive monitoring. We also show that the choice of the heuristic to select suspected components for monitoring is important to minimize the overhead caused from repeated instrumentation and probe activation processes.

For the use case, we created two components and we introduced them into the template application separately. Both components perform the same task, which is performing a *primality test* on a random number and sending the number to another component. However, one of the components causes 115 classes to be loaded, while the other only loads 4 classes.

We used the same basic scenario with a varying number of *primality testing's* components and component sizes. In this way, we were able to simulate the two dimensions of application size. The exact settings, leading to 12 experiments, are defined by the composition of the following constraints:

- $N_{comp} = \{4, 8, 16, 32, 64, 128\}$ which defines the number of components for the application
- $Size_{comp} = \{4, 115\}$ which defines the number of classes for a component

With these use cases, we measured the delay to find the faulty component and the execution-time overhead caused by monitoring. Figures 4.7 and 4.8 show the delay to detect the faulty component with regards to the size of the application. In the first Figure, the component size is 115 classes, and in the second Figure, the component size is four classes.

4.4.4.1 Impact of the application size

Figures 4.8 and 4.10 show the size of the application has an impact on the delay to detect faulty components, and also on the monitoring overhead. We also calculated the time needed to find the faulty component with the *All components* mode after its initialization (the time needed to switch from *Global monitoring*). This time is around 2 seconds no matter the size of the application. That is the reason the switch from *Global monitoring* to *All components* has such a large effect on overhead.

These figures also show that using *Localized monitoring* instead of *All components* when switching from *Global monitoring* helps reduce the impact of the application's size by reducing the number of components to monitor and the number of classes to instrument. However, we also see that using a sub-optimal heuristic may have negatively impacted the delay to detect faulty components. This can be explained by the multiple switches that the Random heuristic may often require to locate the faulty component.

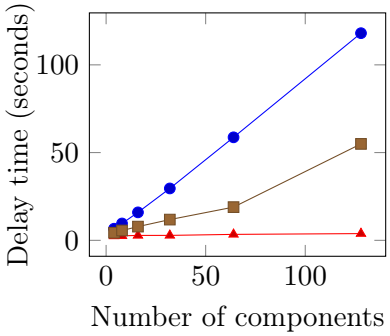


Figure 4.7 – Delay time to detect fault with a component size of 115 classes.

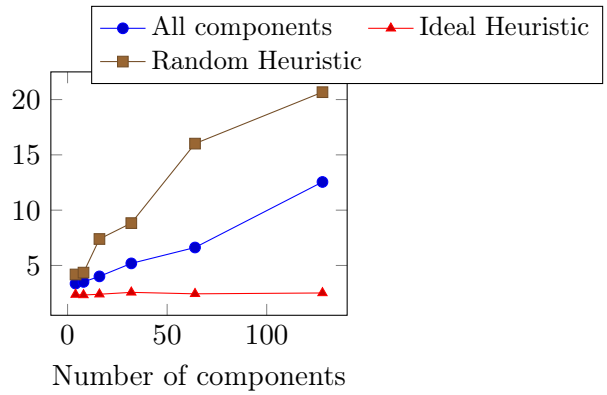


Figure 4.8 – Delay time to detect fault with a component size of four classes.

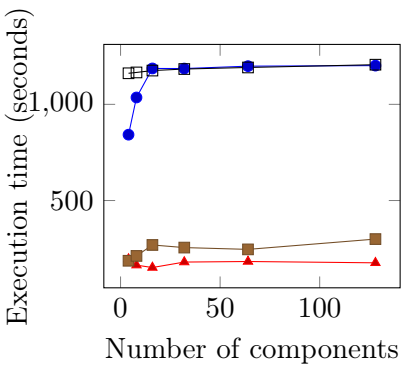


Figure 4.9 – Execution time of main task with a component size of 115 classes.

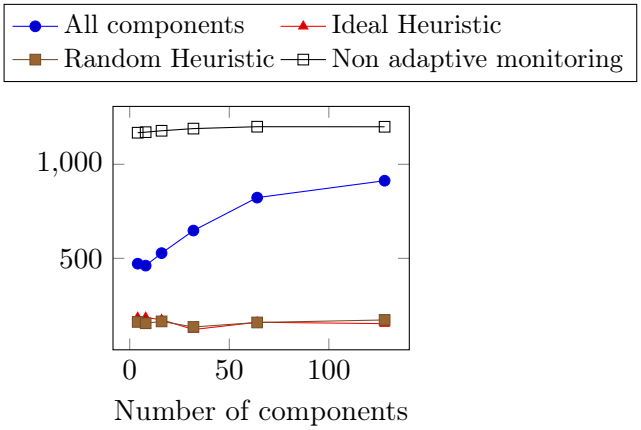


Figure 4.10 – Execution time of main task with a component size of four classes.

4.4.4.2 Impact of the component size

In Figures 4.7 and 4.8 we can observe that the component size greatly impact the performance and the delay for ScapeGoat to find the faulty component. Similar to the explanation for the application's size, component size impacts the switch from *Global monitoring* to *Localized monitoring*, because of the class reloading and instrumentation. A good heuristic drastically reduces the number of transitions; thus, it has a huge impact on the delay. When the components size increase, the choice of a good heuristics becomes even more important, because the cost of dynamic monitoring probes injection increase with the size of the components.

4.5 Scapegoat to spot faulty components in a scalable diverse web application

In this section, we present another application that benefits from the Scapegoat approach. Although the general goal of spotting components that behave abnormally regarding resource consumption remains the same, with this use case we highlight the possibility of using Scapegoat to automatically find buggy components on a scalable modular web application. The section 4.5.1 presents an introduction to the application use case, while the remainder of the section deals with the experimental setup and the results.

4.5.1 Use case presentation

We are applying the Scapegoat approach to check resource consumption contracts on a web application called MdMS.[10] This application offers a web Content Management System based on the Markdown language for editing posts. MdMS uses a typical architecture (as shown in Figure 4.11) for scalable web applications: a load-balancer connected to a set of workers (called MdMS Sosie in Figure 4.11), which are themselves connected to a distributed database to retrieve the application specific content. The worker layer of this application can be duplicated across various machines to support a growing number of clients. The web application is currently online[11].

The main characteristic of MdMS is that workers are not pure clones but diverse implementations of the MdMS server stack [ABB+14]. This proactive diversification of MdMS targets safety [Avi85] and security [FSA97] purposes. In particular, we have used a recent technique for the automatic synthesis of *sosie* programs [BAM14] in order to automatically diversify the workers. A *sosie* is a variant of a program that exhibits the same functionality (passes the same test suite) and a diverse computation (different control or data flow). *Sosie* synthesis is based on the transformation of the original program through statement deletion, addition or replacement.

While the construction of *sosies* focuses on preserving functional correctness, it ignores all the non-functional aspects of the program. Consequently, a *sosie* offers no

[10] `https://github.com/maxleiko/mdms-ringojs`
[11] `http://cloud.diversify-project.eu/`

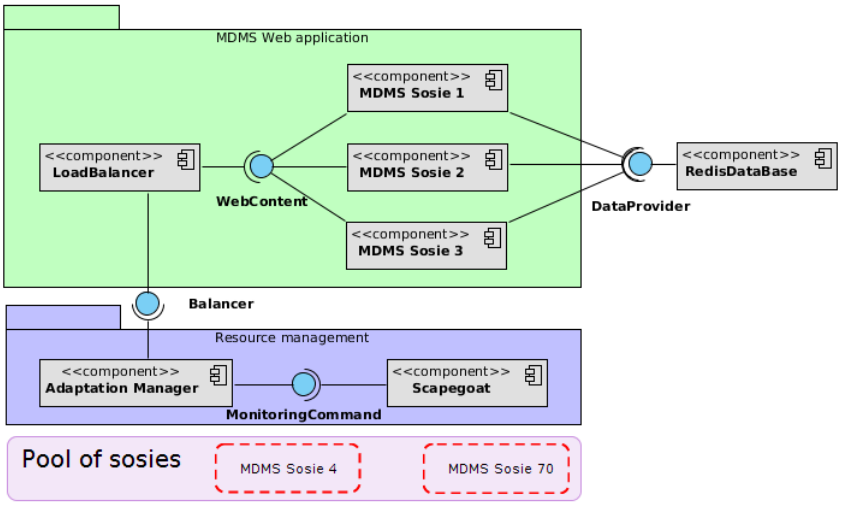


Figure 4.11 – Architecture of MdMS along with Scapegoat and additional components to adapt the system.

guarantee regarding its resource consumption and may contain memory leaks or other overhead on resource consumption that can significantly impact the performance of MdMS.

In this experiment, we use Scapegoat to monitor the resource consumption of the various *sosies* of the MdMS workers. This technique enables us to identify *sosies* in a production environment that do not behave according to the resource consumption contracts, allowing the system to remove these workers and use other *sosies*. Our goal in this experiment is to answer the following question:

- Does Scapegoat correctly identify the faulty components in a system which includes many variants of the same component?

4.5.2 Experimental setup

We devised this experiment as a scenario where many clients interact with the web application at the same time by adding and removing articles. The stress produced by these requests increases the resource consumption on the server side which is running on top of Kevoree components. Figure 4.11 depicts the server side's configuration. Since MdMS is a web application developed on top of RingoJS [12], a JavaScript runtime written in Java, our *sosies* include the RingoJS framework and the application that has been wrapped into Kevoree components.

In this experiment, we deploy many of these components as back-end servers of the web application and we use Scapegoat to monitor the consumption of each server. Their contracts regarding resource consumption were built using the mechanism described in section 4.4.1 but with the original MdMS worker as a reference component. The

[12] `https://github.com/ringo/ringojs`

application also contains a component acting as a front-end that evenly distributes the requests among back-end servers. This load balancer implements a plain round robin policy.

To produce a realistic load on the web server we have recorded a set of standard activities on the MdMS web site using Selenium [13]. We then use the Selenium facilities to replay these activities many times in parallel to provide the required work load on the server. Our experimental settings feature 120 clients which are scheduled by a pool of 7 concurrent Selenium workers. Each client adds 10 articles to the database through the Website GUI, which represents 16 requests per article, for a total of 19200 requests to the MdMS workers sent through the load balancer. In this experiment, the Selenium workers are executed on the same physical device as the web server, with the same testing platform described in section 4.4.1.

The experiment is configured as follows. Using the diversification technique described in [BAM14], we synthesized 20 *sosies* of the MdMS workers. These *sosies* are used to execute the application with a varying number of back-ends (from 4 to 10). One particular *sosie* has been modified by hand to ensure that it violates the original component's contract. We execute all the described components as well as the Scapegoat components on a single instance of Kevoree.

4.5.3 Experimentation results

Figure 4.12 shows the time required on the server side to reply to all the requests sent by Selenium. Although the values might look surprisingly high at first, they are in fact the result of a heavily loaded system. Selenium is actually rendering a couple of web pages for each added article; hence at least 2400 pages are rendered. Moreover, both clients and servers are sharing resources because they run on the same physical device. This leads to very stable execution times when monitoring is not activated because the number of requests does not change between experiments, the load balancer distributes these requests evenly, and we are using the same physical device to execute all back-end servers. In the *local monitoring* series, the global time to execute decreases until reaching 9 *sosies*. Although counterintuitive, it is caused by the effect of having *localized monitoring* and *load balancing* at the same time. For instance, when four *sosies* are used, the monitoring probes are periodically injected into one component out of four, hence roughly a quarter of the requests are handled by a slower *sosie*. However, with eight components the slow execution path is only taken by around 12.5% of the requests. The overhead of *Localized monitoring* when ten *sosies* are deployed increases because the physical machine reaches its limit and begins thrashing. As a consequence, low-level interactions with the hardware (e.g. cache misses), the operating system and the JVM slow down the execution. On average, the overhead due to monitoring with both instruction instrumentation and memory instrumentation is 1.59, which is lower than the values shown in section 4.4.2 for full monitoring despite only one of the instrumentation mechanisms being enabled in those experiments. The values in this section are better even if we are monitoring both resources because we are using the adaptive approach.

[13] `http://www.seleniumhq.org/`

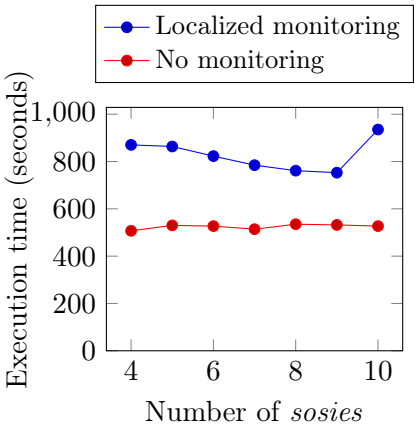


Figure 4.12 – Time to obtain the reply to all requests.

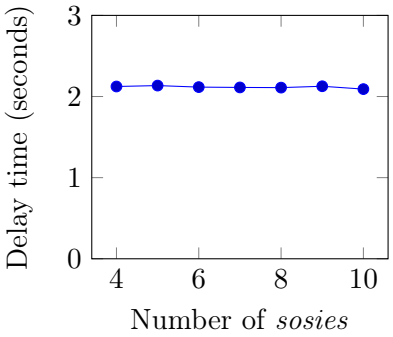


Figure 4.13 – Average delay time to detect a faulty *sosie*.

In these experiments, we evaluate the accuracy of the output and its quality in terms of the time needed to find the faulty component. Scapegoat always spots the correct *sosie*. It does so because it is an iterative process that continues until finding the faulty component. In addition, Scapegoat does not output false positives during these experiments. The delay to detect faulty components is shown in Figure 4.13. In this case, the values remain close to 2 seconds no matter the number of *sosies* used nor the execution time. This behavior is consistent with the experiments in section 4.4 because we are also using a good heuristic for the use case. It shows that Scapegoat can spot faulty components with an acceptable delay in a real application.

4.5.4 Discussion of the use case

This use case shows that Scapegoat is able to provide useful information in real applications. It also highlights how the framework can help select software variants at runtime in the context of software diversity. Or, more generally, in the field of software oriented architectures where many stakeholders may provide the same services, Scapegoat can help to choose services. Moreover, this use case leads to a distributed usage of Scapegoat, where the policies for admission control and resource consumption monitoring can be coordinated among distributed devices.

Finally, in systems where there are many variants of the same component or service, Scapegoat provides essential information to drive application reconfiguration. For example, the adaptation component in Figure 4.11 may use Scapegoat's faulty component selection to replace a faulty *sosie* or to modify the scheduling policy in the load balancer.

4.6 Threats to validity

Our experiments show the benefits of using adaptive monitoring instead of state-of-the-art monitoring approaches. As in every experimental protocol, our evaluation has some bias which we have tried to mitigate. The experiments are based on a few cases of study. We have tried to mitigate this issue by using available real cases study; we have also used different settings across our experiments. Thus, our experiments limit the validity of the approach to applications with the same characteristics of the presented case study. New experiments with other use cases are needed to broaden the validation scope of our approach.

The evaluation of the heuristic mainly shows the potential impact of using an ideal heuristic. More cases of study and experiments are needed to fully validate the value of our Models@run.time based heuristic.

4.7 Conclusions

In this chapter we presented Scapegoat, an adaptive monitoring framework to perform lightweight yet efficient monitoring of Component-Based Systems. In Scapegoat, each component is augmented with a contract that specifies its resource usage, such as peak CPU and memory consumption. Scapegoat uses a global monitoring mode that has low overhead on the system, and an on-demand fine-grained localized monitoring mode that performs extensive checking of the components' contracts. The system switches from the global monitoring mode to the localized monitoring mode whenever a problem is detected at the global level in order to identify the faulty component. Furthermore, we proposed a heuristic that leverages information produced by the Models@run.time approach to quickly predict the faulty components.

Scapegoat has been implemented on top of the Kevoree component framework which uses the Models@run.time approach to tame the complexity of distributed dynamic adaptations. The evaluation of Scapegoat shows that the monitoring system's overhead is reduced by up to 93% in comparison with state-of-the-art full monitoring systems. The evaluation also presents the benefits of using a heuristic to predict the faulty component. In the second part of the evaluation, we highlighted the benefit of Scapegoat on a classical web server architecture to dynamically determine faulty components. This second example also exposes the capacity of Scapegoat to be applied to different application domains and confirms its relatively low overhead on the running system. Scapegoat contributes to the state of the art by providing a monitoring framework which adapts its overhead depending on current execution conditions and leverages the architectural information provided by Models@run.time to drive the search for the faulty component.

The approach proposed in this chapter contributes to answer two research questions that were presented in the introduction of this thesis (see Section 1.2). In particular, it answers *RQ1* (*How can we provide portable and efficient support for resource consumption monitoring?*) by describing a monitoring framework that produces low performance overhead and is fully portable. Likewise, our proposal partially answers *RQ3* (*How can we leverage the knowledge about the architecture of applications to drive*

a mechanism for resource management?) by using knowledge about the structure of applications to drive the behavior of the framework.

Chapter 5

Squirrel: Architecture Driven Resource Management

Resource management is critical for domains where software components share an execution environment but belong to different stakeholders. For instance, in multi-tenant systems resource management is used to guarantee safety, reliability and per-stakeholder Quality of Service (QoS). These applications essentially require the isolation of tenants in terms of resource consumption [KWK13]. This enables, for example, *Software-as-a-Service* layers for cloud systems, allowing innovative pricing policies based on user requirements. Since these services are often implemented on paradigms such as component models (in the form of micro-services), the design of resource management techniques dedicated to component models is an important issue.

Component model implementations represent high level concepts, such as component instances, by means of mapping them to system-level abstractions like objects, threads, processes or virtual machines. Each mapping has unique features in terms of performance, memory footprint, etc. However, these mappings are often done in a once-size-fits-all manner, allowing some choices to optimize for memory use while others might, for example, improve inter-component communication. Interestingly, system abstractions offer varying resource management capabilities that differ in how they impact the application. Although we can hard-code the resource management concern during the design of the component model, we argue that this leads to sub-optimal systems with high overhead [Bin06, CvE98, MBKA12] because components have different resource requirements.

To address this issue, we propose Squirrel, an approach to resource management for component models that aims at reducing overhead. In Squirrel, the application is deployed with a model containing resource usage contracts for each component and a detailed view of the system. These metadata are used to choose at deployment-time the best way of representing each component in terms of system abstractions. This contrasts with the *traditional* approach of binding the representation during the design of the component model and results in the final runtime representation of the system only being known after deployment.

In this chapter we discuss an approach to resource management applicable to any

component model. To validate the feasibility of our proposal, we present a reference implementation for a Java-based component model. A set of experiments validate its feasibility and show various aspects of its overhead. The results demonstrate that choosing the right *component-to-system* mappings at deployment-time can reduce performance overhead and/or memory footprint. The contributions of this chapter are as follow:

- An approach for architecture driven resource management that leverages structural information to guide the mapping of component model concepts onto system-level abstractions.
- A reference implementation of the Squirrel framework for a Java-Based component platform.
- A performance comparison showing how different mappings can impact the overhead of the system and how the approach behaves in comparison to state-of-practice approaches for resource management.

The remainder of this chapter is organized as follows. Section 5.1 describes the Squirrel approach and presents how we leverage metadata to drive resource management. In section 5.2 we propose a reference implementation of Squirrel for a Java-based component platform. A validation of the implementation through a set of experiments is presented in section 5.3.

5.1 Approach

The main concept in Squirrel is the *resource-aware container*. Such containers are logical entities that take care of the resource management concern. By logical we mean that it is not important, from a functional point of view, how a container achieves resource management. Instead, a resource-aware container is an entity that *wraps* a set of components and offers the following properties:

- **Resource consumption monitoring** refers to the ability to assess the quantity of resources used by a component.
- **Resource reservation** is the capacity to ensure a given amount of resources will be available whenever a component demands it.
- **Resource isolation** guarantees that a component's behavior in terms of resource usage does not interfere with the behavior of another component.

Wrapping a set of components can be considered a soft definition because the *membrane* of a resource-aware container limits the behavior of the contained components only when it is relevant to the resource management concern. For instance, components within different containers can still communicate directly with each other through their interfaces without intervention of their containers as long as such communication does not affect the resource under management.

In Squirrel we propose to automatically select, deploy and configure resource containers to manage resource usage. The novelty is that we delay the selection of the container's implementation till deployment-time in order to have knowledge about the exact conditions of the system and thus minimize the overhead of the resource management system. This idea is supported by the claim that components often require disjoint sets of resource types. Our framework is composed of three essential elements: i) a mechanism to describe the management requirements of an application, ii) an admission control scheme in the middleware to handle the global view of resource availability, and iii) mechanisms to map component model concepts to system-level abstractions. In the following subsections we describe our framework and its elements.

5.1.1 Managing resources through architecture adaptations

Modern application development models, such as component-based systems, promote the usage of Architecture Description Languages (ADL) or configuration models to check properties on the system's structure and to drive system deployment. In Squirrel, we propose to enhance this layer with metadata regarding resource reservation and to use these metadata to efficiently drive resource reservation offered at the system level. The idea is to follow a gray-box approach where we automatically adapt a component-based application by applying an architecture pattern to isolate a component within a resource-aware container.

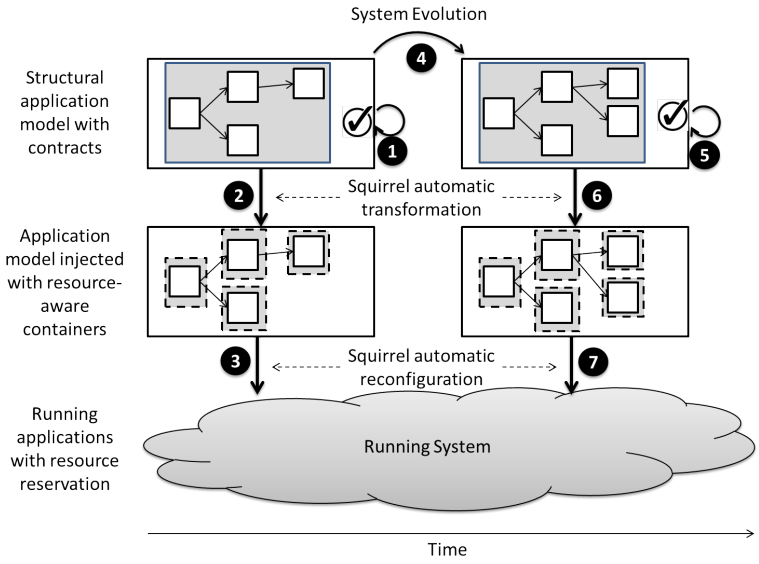


Figure 5.1 – Squirrel approach for resources reservation

As illustrated in figure 5.1, Squirrel follows an automatic process to manage resources. Squirrel receives an application's model, enhanced with contracts on resource reservation. Squirrel performs admission control to check the validity of the contracts on resource usage with respect to the available resources in the execution environment. If the contracts are consistent with available resources, the process continues, if not, the

application's model is refused. Then, as depicted by arrow 2, Squirrel automatically transforms components or the configuration/architecture model by isolating components in resource-aware containers that can be finely configured to decrease the resource management overhead. Finally, as depicted by arrow 3, Squirrel reconfigures the running system. When the application evolves (arrow 4), Squirrel attempts to preserve resource reservation properties while processing the new model (arrow 5, 6 and 7).

5.1.2 Describing resource management requirements

Beugnard et al. discuss the extending Meyer's *Design-By-Contract* idea to software components [BJPW99]. They classify component contracts into four categories: syntactic (level 1), semantic (level 2), synchronization (level 3), and Quality of Service (level 4). There is no de-facto standard to describe component contracts, but many domain specific interface description languages contain such metadata. This chapter assumes that components have contracts to deal with resource reservation (level 4). A contract in Squirrel defines component resource requirements written in terms of resource types, quotas and expected component usage.

- **Definition 1** A resource type indicates any class of computational resource that is useful to a component. Its consumption must be susceptible to monitoring and reservation. In this chapter, we consider CPU, Memory, Network Bandwidth and IO Throughput.

- **Definition 2** Expected component usage describes the expected number of external invocations of each method of the component interface. In short, let C be a component instance, then $\forall I \in C_{Interfaces}, \forall M \in I_{methods}$
  EU_{IM} is the number of expected invocations of method M, per second.

A **contract in Squirrel** is a set of tuples with the form $\langle RT, N, MU \rangle$ where RT is a resource type, N the maximum amount of resources to reserve, and MU is the measuring unit used for this resource type. Optionally, Squirrel supports the definition of a set of tuples with the form $\langle I, M, EU_{IM} \rangle$ where I is a component interface, M is a method of the interface, and EU_{IM} the expected usage. Implementations of the Squirrel approach must provide a way to define contracts with these concepts. We use a domain-specific language to describe contracts.

5.1.3 Admission control

Providing resource reservation in a component based framework requires checking if components' resource-aware contracts are compatible with the resources available in the execution environment. By checking the availability of resources, the platform controls component admissions.

To support resource management at runtime, Squirrel takes into account two events: i) component deployment, and ii) component removal. Whenever the application is modified, the system automatically recalculates the aggregated resources required by the application and compares it to the available resources in the execution environment.

If the available resources are greater than those required by the application, the reconfiguration is accepted, else, the application model is refused and the reconfigurations are discarded.

5.1.4 Mapping component-model concepts to system-level abstractions

Squirrel defines steps to map component-model concepts to system-level abstractions that allow for resource management. Mappings can be applied during the design and implementation of the framework, or at deployment-time. During framework design/implementation, developers identify system abstractions that are suitable to represent each concept and implement the respective mappings. As a second step, resource management methods for each abstraction are implemented and evaluated. This evaluation is used to determine the management methods with lowest overhead for each pair of system abstraction and resource type. Later on, at deployment-time, the platform selects a component-to-system mapping using optimization techniques and the data obtained at design-time. In this section, we briefly explain each step.

As we have mentioned, components can be represented through different system abstractions. This requires *identifying possible mappings from components to system-level abstractions*. Mappings must respect the semantics of the component model, and must provide resource management capabilities. A key problem is that different mappings have different non-functional properties, and optimizations are often needed to make the mappings attractive. Additionally, an extensible design of the component platform, where it is easy to accommodate new mappings, greatly facilitates the co-existence of different mappings to represent a component. The set SA of system abstractions that are available to represent a concept, along with the recommended optimizations for each abstraction, are defined in this step.

During the design/implementation of the platform it is necessary to *define methods to manage resources* for each pair of system abstraction and resource type. Developers must devise resource management methods for each mapping and identify the least costly. If we consider different abstractions and resource types, we can define the matrices M and C where $\forall sa \in SA, rt \in RT$ the values $M_{sa,rt}$ and $C_{sa,rt}$ indicate the method that minimizes the cost of managing the resource rt when the abstraction sa is used to represent a component. We make two assumptions about the resource management mechanisms: i) mechanisms are always composable if they manage different resource types, and ii) the costs of any pair of management mechanisms are independent.

At deployment-time, *the platform selects the mapping* to use for each component in the application. To do so, the platform uses the information contained in the matrices M and C, the set of possible optimizations for each mapping, and the resource requirements of the application. At this stage, the only data needed regarding resource requirements is the type of resource. Using this data allows selecting the best mapping candidate. Although we only use a single cost matrix that contains the overhead of each management mechanism, we think it is easy to generalize the approach to handle multi-objective optimizations with more than one cost matrix. Others refinement to

evaluate the cost of a mapping are possible. For instance, we can consider the cost of using a specific binding to connect two components that use a given mapping. Finally, there are many optimization methods that can compute the mappings, we do not propose any particular method in the approach. However, the results shown in section 4.4 suggest that very simple heuristics can lead to good performance.

5.2 Reference Implementation

Squirrel's reference implementation exploits the modesl@run.time approach and provides resource-awareness capabilities to the Kevoree component framework [FMF+12]. Models@run.time denotes model-driven approaches to tame the complexity of dynamic adaptation [MBNJ09a]. The "model at runtime" is a reflection model that can be decoupled from the application (for reasoning, validation, and simulation purposes) and then automatically resynchronized. Models can manage not only the application's structural information (i.e., the architecture), but can also be populated with other information, such as runtime monitoring data.

Kevoree is a component framework for distributed systems that builds and maintains a structural model of the system, following the models@run.time paradigm. Kevoree is mainly used because: (i) it presents a snapshot of the distributed system, and (ii) it provides a language to drive the reconfigurations. *Component* and *Channel* are two of the concepts used in Kevoree models. The former represents software units that provide the business value. The latter, with the same role as connectors, are in charge of inter-component communication. Channels encapsulate communication semantics (e.g., synchronous, unicast).

5.2.1 A resource-aware container for CPU and I/O reservation

Our implementation leverages resource-reservation mechanisms at the system-level to provide containers for CPU and I/O reservation. More specifically, it maps the concept of *container* onto a *cgroup*. Both *containers* and *Kevoree components* are hierarchical structures that are easy to map onto cgroups' hierarchy. Indeed, a container deploys components and a component runs threads. To configure the container we setup a hierarchy of cgroups using the following rules:

1. The Kevoree framework is started under a cgroup, using a fixed amount of resources that will be divided among the system's components.

2. Each component gets a new resource-aware container, also represented by a cgroup. The component's contract is translated into a slice S_c of the initial resource allotment, and the result is passed to the cgroup as configuration parameters. A slice represents the resources the component has available.

3. Since the scheduling unit for cgroups is a thread, we assign the component's threads to the cgroup to enforce resource reservation.

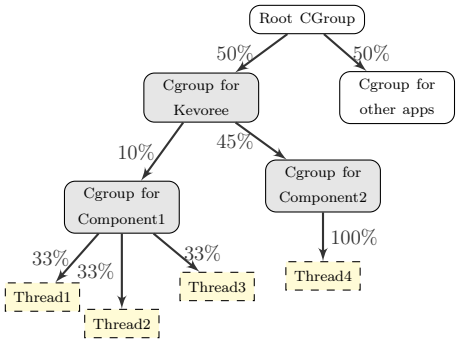


Figure 5.2 – Reserving CPU by mapping components to cgroups

This scheme is used for CPU, I/O throughput and network bandwidth. Each type of resource requires a different *container type*. Figure 5.2 shows an example using cgroups to reserve CPU for a system with two components. In the tree, every edge is labeled with the cgroup's CPU slice. A slice is set for Kevoree, while unmanaged apps are maintained in separate containers. Applying rule 2, CPU slices are assigned to *Component 1* and *Component 2* according to their resource contracts. Following rule 3, every thread in *Component1* receives 33% of the component's slice.

5.2.2 Containers for Memory reservation

Memory reservation poses a unique challenge. Although there is a cgroup-hierarchy for memory, it is not well suited for shared memory because the subsystem cannot precisely account for the consumption of each thread. As a result, if we use cgroups to deal with memory, accounting would depend on the behavior of the garbage collector, which is hard to predict. That makes cgroups inadequate to check or enforce component contracts in a single JVM process. We have devised two mechanisms to serve as memory containers. In the first mechanism, all containers exist in a single process and resource limits are enforced by leveraging previous approaches that use bytecode instrumentation to account for consumption. The second mechanism isolates components into new processes and then uses cgroups. The rest of the section describes both solutions.

5.2.2.1 Memory management based on monitoring.

A memory reservation container ensures that its components have access to the memory they require. Memory requests should only fail if a component violates its contract. Implementing such a container is simple if memory monitoring is available at the application-level and memory requests are intercepted. We use Scapegoat [GHBD+14] for memory consumption monitoring by defining multiple memory-aware containers within a single JVM. In short, each container registers its component in ScapeGoat and receives a notification if a component violates its contract. This introduces CPU and memory overhead for each component because of the instrumentation code. The main advantages are portability and simplicity.

5.2.2.2 Isolation of components in separate processes.

The second approach maps each container onto a separate process. Reservation is achieved using cgroups as described in section 5.2.1.[1] To do so, we start from an extended deployment model as shown in figure 5.1. The model is then transformed using the following rules:

1. Component isolation: each *set of components* with a shared memory contract is isolated in a separate *JVM node* within the same *physical device*.

2. Channel adjustment: channels that connect isolated components are updated to reflect the semantics of the source model. This includes changing the channel type and modifying some of its properties.

The resulting model can be deployed.

Runtime initialization through cloning. The approach to memory reservation based on isolation deploys each *set of components* into separate processes. This involves two steps: creating new instances of the runtime, and deploying a *set of components* on each instance. To reduce deployment time, instead of starting processes from scratch, new instances are cloned from a base runtime. The base runtime is created offline. Both steps, base runtime creation and cloning, are based on CRIU.[2] This tool allows snapshoting a process and starting any number of clones from the snapshot. In essence this *forks* the process.

Channel for intra-node communication. Channels are meant to send arbitrary POJO structures from one component to another. When components are isolated into separate processes, a channel must marshal and unmarshal the POJO using a representation suitable for inter-process communication. In practice, a channel must copy data at least twice, no matter what IPC mechanism is used.

In this chapter we propose a new channel for intra-node communication. Communication is performed through a message queue built on top of shared memory using an alternative high-performance framework to serialize objects. Each channel is mapped to a shared-memory region that hosts a synchronized queue of messages. This region contains three sections: an array of blocks to store message chunks, a set of free blocks, and a circular queue in which an element points to a list of chunks. We use the procedure described in [UK98] to synchronize senders and receivers, but we also support broadcast semantics without unnecessary additional copies. Our approach copies the POJO from the sender's heap to shared-memory during data marshaling, then every receiver makes a copy from shared-memory. The implementation uses a high-performance serialization framework[3] instead of Java's built-in serialization mechanism and is able to serialize arbitrary objects with better performance.

[1] In practice, we use cgroups to reserve memory, but we also bound the Java Heap to limit the consumption in Java code.

[2] `criu.org`

[3] `https://github.com/RuedigerMoeller/fast-serialization`

5.3 Evaluation

This section presents experiments that determine the performance of our reference implementation. We evaluate performance and overhead using systematic, full isolation of components using resource containers. We also compare different design decisions presented in the previous section. The experiments include:

- Measuring the CPU and memory overhead introduced by *Scapegoat* and *component isolation*.
- Determining how isolation affects deployment time and to what extent process cloning and our high-performance IPC alleviate it.
- Evaluating the extent to which known high-performance IPC techniques reduce communication overhead due to component isolation.

We used the same hardware across all experiments: a laptop with a 2.90GHz Intel(R) i7-3520M processor, running Linux with a 64 bit kernel 3.13.5 and 8GiB of RAM. We used the HotSpot JVM v1.7.0-55, and Kevoree framework v5.0.1.

5.3.1 Evaluating performance overhead

In section 5.2.2, we describe two approaches for memory reservation: 1) Scapegoat [GHBD+14], an instrumentation-based resource management container, and 2) using isolated processes with cgroups. To compare the overhead produced by these approaches, we devised use cases that contain two components that execute a test from the Dacapo Benchmark Suite [Bea06]. These components run in parallel to simulate realistic conditions where components demand resources simultaneously. The use cases are executed with different settings, as follows:

1. Using cgroups to assign 50% of the CPU time to each component (no memory monitoring). Both components run on a single Kevoree instance with 2GiB maximum heap size. This is the baseline because there is no CPU nor memory overhead. Nevertheless, execution time is affected due to the CPU allotment. We call this setting *Memory unaware*.
2. Using *Scapegoat* to monitor per-component memory consumption and bounding CPU consumption to 50%. Again, both components run on the same Kevoree instance with 2GiB maximum heap size.
3. Isolating each component in a new process, allotting 256MiB of memory to each process, and bounding its CPU consumption to 50%.

To measure CPU overhead, we use the result reported by each *Dacapo test* and we keep the maximum value. Measuring memory overhead is more complex because the garbage collector hides the real consumption. We address this by approximating the consumption with the usage after each major garbage collection. As there are many

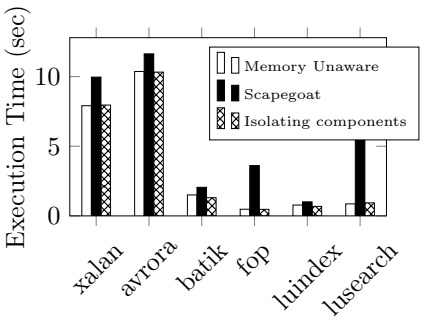


Figure 5.3 – CPU overhead caused by resource management during the execution of Dacapo benchmarks.

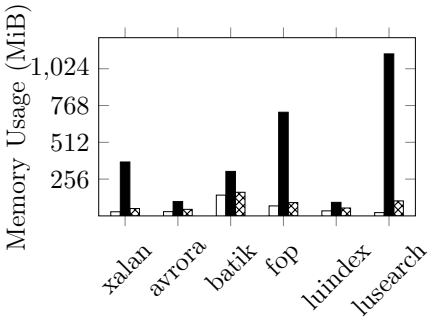


Figure 5.4 – Memory overhead caused by resource management during the execution of Dacapo benchmarks.

collection cycles during the use case's execution, and because we may have more than one runtime involved, we define a scheme to aggregate the values: the consumption MC_i of every Kevoree instance is the maximum among all the usages reported after each collection, while the final consumption is defined as $\sum_{i \in Isolates} MC_i$.

Figure 5.3 depicts the CPU overhead for *Scapegoat* and *Isolating components*. The overhead of *Scapegoat* was expected because of the instrumentation. On average, it performs 2.27 times worse than *Memory Unaware*, which is consistent with [GHBD+14]. In contrast, isolating components produces no appreciable CPU overhead for these use cases (small differences are likely due to environment fluctuations) because the components do not interact.

Both *ScapeGoat* and *Isolating components* cause memory overhead. As shown in Figure 5.4, *ScapeGoat's* is higher than when using component isolation. *Isolating components*'s overhead is the result of JVM duplication and is, on average, 99% over baseline. Meanwhile, *ScapeGoat's* overhead is due to tagging objects with the identifier of the component that owns it. Tagging adds either a field and a finalization method to an object, or wraps the object with a *weak reference* held by the framework, resulting in overhead in the *Permanent Space*. As seen in the experiments, memory overhead due to tagging is more important than CPU.

In synthesis, *Isolating components* produces no CPU overhead, and low memory overhead in comparison to the performance of the same component model without resource management features.

5.3.2 Evaluating starting time

We compare the performance of three methods to deploy components: 1) in a single JVM (the baseline), 2) in isolated JVMs, starting processes from scratch, and 3) in isolated JVMs using CRIU to clone processes. We study the scalability of each method by deploying many components. To do so, we created a template architectural model with two component types: *Component A* runs in the *management runtime* and deploys a new architectural model with resource-aware contracts; and *Components B* records

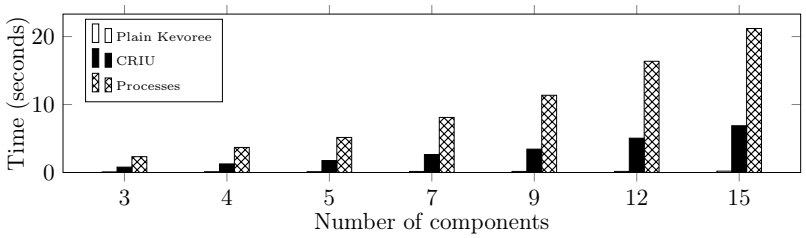


Figure 5.5 – Average deployment time per component using different strategies

the timestamp after initialization is completed. The experiment is as follows:

1. Component A is deployed in the *management runtime*. Afterwards, it forces the deployment of a new model with components of type B.
2. After deployment, each component $c \in List_B$ sends A the timestamp T_c.
3. Component A collects $T_c - T_0, \forall c \in List_B$, where T_0 is the timestamp before deployment.

Figure 5.5 shows the results for a varying number of components. As expected, using plain Kevoree is faster than deploying with other methods. Leveraging isolation with CRIU's process cloning is, on average, 19.75 times worse than plain Kevoree, and this overhead grows with the number of components. This is because CRIU-based deployment still spawns new threads in order to clone and create new instances. Nevertheless, using process cloning instead of starting processes from scratch reduces the isolation overhead by a factor of 41.79.

5.3.3 Evaluating communication

We present two experiments that highlight how we mitigate the impact of isolation on communication performance. First, we use a micro-benchmark to compare the performance of a shared-memory based IPC queue and a socket-based IPC queue. Then we benchmark the performance gain of using a specialized serialization framework that uses POJO structures, which is a common way to encode the business logic in real-life applications.

We evaluate our channel by comparing its performance against built-in TCP communication, which is a widely used IPC mechanism in Java. To measure latency and bandwidth, the metrics we use for comparison, we developed a Netpipe clone [SMG96] for Java[4]. The clone is delivered with three protocols: 1) socket-based, 2) our channel, and 3) a protocol for *named pipes*. The first uses simple TCP sockets with synchronous operation in its implementation. The second requires two channels, configured to use a queue with 128 chunks of 512Kb, because our channel is unidirectional. Figure 5.6 shows the latency for messages shorter than 128 bytes. Memory-based communication outperforms NIO-sockets for all the values in the range. Likewise, Figure 5.7 shows the throughput of both mechanisms. In general, memory-based communication behaves

[4]Source code at: `http://goo.gl/h7OVm5`

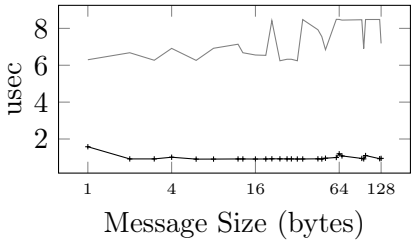


Figure 5.6 – Comparing latency of different IPC mechanisms

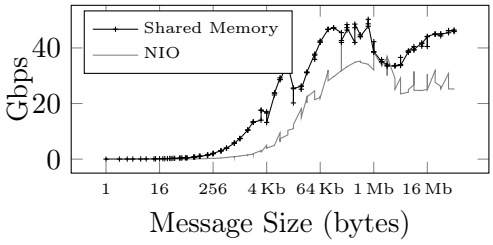


Figure 5.7 – Comparing bandwidth of different IPC mechanisms

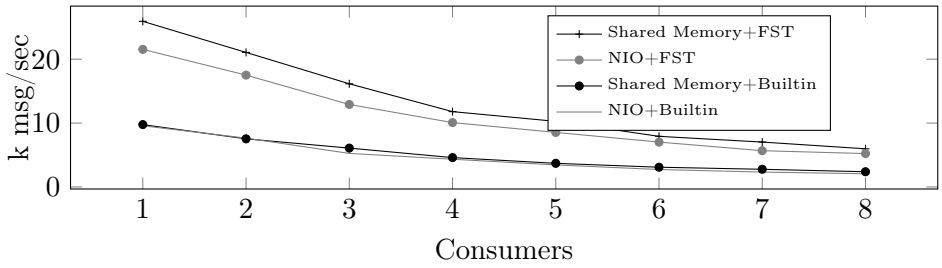


Figure 5.8 – Communication throughput for different channels.

better than TCP sockets. Our approach outperforms sockets by an average of 652.36% for messages shorter than 512 bytes. Meanwhile, it behaves on average 46.26% better for messages between 1 Mb and 64 Mb, which is the range where the benefits of large copies surpass the disadvantages of having a synchronized queue.

Latency between Java isolates is also affected by the time spent marshaling and unmarshaling messages. To evaluate the benefits of *fast Serialization*, we designed a micro-benchmark that sends a POJO structure, 16 bytes long, back and forth a million times. [5] We then measure the effect of marshaling for different numbers of consumers. Figure 5.8 shows the results of using built-in serialization against *fast Serialization* for up to 8 components. We chose this value because in component-based systems it is unlikely to find more interconnections within a single node. The figure depicts the results in *messages per second* because different serialization methods flatten the same POJO structure into buffers with varying sizes. The comparison includes two serialization and two IPC mechanisms. However, the effect of the IPC mechanism is low due to the fact that serialization dominates the execution times. As a result, curves with the same serialization mechanism are close no matter what IPC mechanism we use.

5.3.4 Synthesis and Threats to validity

We evaluated our implementation of Squirrel regarding three aspects: overall performance overhead, starting time and communication performance. We believe that the performance of our reference implementation is good enough to enable resource manage-

[5] Source code at: `http://goo.gl/FXuUxc`

ment, while not excessively affecting the application's behavior. Although some metrics exhibit high overhead, we think that the trade-off given the new features offered in Squirrel is worth considering. Memory overhead is the biggest concern. Nevertheless, isolating components within separate processes greatly reduces the memory overhead and eliminates CPU overhead in comparison to the instrumentation-based solution from Scapegoat.

A threat to validity of our experimental protocol is that we evaluate different features as orthogonal concerns. Our experiments do not study the impact of all of Squirrel's features together in a real scenario, although the assumption of orthogonality of each concern is reasonable, particularly because Squirrel mainly relies on the well tested cgroups to enable CPU and I/O reservation.

5.4 Conclusion

In this chapter, we advocate for a methodology to provide resource management capabilities to dynamic component-based frameworks. This methodology and its implementation, Squirrel, propose choosing component-to-system mappings at deployment time for better resource management. This strategy is performed automatically by checking the resource availability and transforming the application's structure to run the application on resource-aware containers. Containers describe how to map components to system abstractions allowing for different trade-offs in resource management.

The implementation we present is able to manage CPU, I/O and memory, and provide performance analyses and a comparison of different design decisions. The experiments show that choosing the right component-to-system mappings at deployment-time reduces CPU overhead and/or memory use. They also highlight that optimizing mappings is essential to reducing isolation and communication overhead to acceptable levels.

The approach proposed in this chapter contributes to answer two research questions *RQ2* (*How can we choose what mechanisms must be used to guarantee resource reservation with low overhead for each component?*) and *RQ3* (*How can we leverage the knowledge about the architecture of applications to drive a mechanism for resource management?*). Instead of selecting a fixed resource reservation mechanism for all components, we delays this selection until deployment time when we know exactly what resources a component require, and how components interact. At this point, we can specialize the approach used to reserve resources in such a way that performance overhead is kept at a low level.

Chapter 6

Building Efficient Domain-Specific Memory Profilers

Jack (V.O.) – Babies don't sleep this well.

(Jack's bedroom – night – Jack lies sound asleep)

Jack (V.O.) – I became addicted [...]

Jack (V.O.) – If I didn't say anything, people assumed the worst. They cried harder. I cried harder [...]

Jack (V.O.) – [...] the guys with cancer [...] "Free and Clear", my blood parasites group Thursdays [...] "Seize The Day", my tuberculosis Friday night.

(Fight Club)

The Software Language Engineering (SLE) community aims at reducing the effort required in engineering new languages and their corresponding development tools, thus improving the efficiency of both people in charge of designing new languages and their users [Spr14]. However, as far as we know, they do not take into account profiling tools, which are essentials for software maintenance and optimization. Indeed, although specific tools are needed to monitor running systems in order to detect defects or abnormal behaviors [DB00, JAH11], little support exists to ease their creation.

In this chapter, we focus on the problem of easing the creation of efficient memory profilers for domain-specific software abstractions that are designed to be executed on top of MRTEs. We first propose a metalanguage to specific what data about the memory use is of interest in a domain (see Section 6.2). A profiler is then generated to collect the data and present it in terms of concepts of that language. In addition, we present a tooled DSL based on such a metalanguage, which generates profilers for the JVM (see

Section 6.3). An important point of our approach is the low overhead induced by these profilers; this makes them usable in production environments (see Sections 6.4 and 6.5).

The contributions of this chapter are as follows:

- A metalanguage to describe the information that a profiler must collect. In addition, programs written using this metalanguage also define how to collect the information. Although knowledge of the underline execution model is required, the procedure to obtain data is mostly defined without using low-level details.

- A concrete implementation of this metalanguage that targets the JVM. In particular, by using the JVMTI, we can generate memory profilers with low overhead. Concrete profilers already generated are portable to any implementation of the JVM that supports JVMTI.

- A discussion of the metalanguage's expressiveness, and an evaluation of the performance overhead induced by three profilers in real-world use cases.

6.1 Understanding the domain

The purpose of memory profilers is to collect information regarding how applications use memory. To write profilers, a clear understanding about their nature is required. Informally, the term *memory profiling* refers to any kind of **process to collect** some **data about memory usage**. In an object-oriented runtime environment, such data may be as simple as the number of objects of a specific class, but it may also be as complex as the list of possible memory leak's sources. Other examples include computing the number of objects reachable from a specific class object; finding out if there is an instance of class A which is referencing an object of class B; and computing, for each *String*, its length and the number of objects that make direct reference to the *string*. The data collected by a profiler may have an arbitrary complexity; it may be a simple natural number, a boolean value, a list of values, or a composed value. For instance, an integer value is required to describe one of the aforementioned examples, the number of objects reachable from a specific class object. In the same way, a pair $\langle l, r \rangle$, where both l and r are integers, is required to store the result of determining for a *string*, its length and the number of objects referencing it.

Formally, we use a set of concepts to properly define our understanding of the term profiler, and how we address their construction in this chapter. The following concepts are used in the rest of the chapter:

An object is an atomic entity that consumes memory to store the value of its attributes. Although we mean *object* as in object-oriented programming (OOP), the operations that we can perform on these objects are restricted to accessing attributes, obtaining the amount of memory used to represent an object, and accessing meta-data such as the class name. Reusing the concept is "natural" since we are targeting MRTEs, which often support the object-oriented paradigm.

Memory Heap As mentioned, this is the region of memory used to store dynamically allocated *objects* that are connected through references – forming a directed graph. It is also the universe $\mathbb{U}$ of objects.

Structure is a subset of *objects* in the *memory heap*. The objects in a structure don't need to be directly related by references; instead, they can be arbitrarily composed. For instance, the smallest non-empty *structure* we can consider, is a structure containing a single object. Only one property is required in properly formed structures; $S_1 \bigcap S_2 = \emptyset$ for any pair of structures S_1 and S_2, which means that they are disjoint sets.

Memory Profile is a value that can be computed using information from the *objects* (and their references) included in a *structure*. An example of useful value is the total size of a structure, which can be easily calculated using the function $total\_size(S) = \sum_{o \in S} sizeof(o)$. A common pattern of use is to identify many *structures* in the heap, and then to compute a value – not necessarily the same – for each structure.

Memory Profiling consists in identifying *structures*, and computing some values associated to these structures.

Structure types provide a mean to identify several structures using a single description. In other words, since the heap can be partitioned in many structures, it is hard to manually enumerate them. We need a procedural way to describe what are the structures we are considering (i.e., their membership functions), and what data we want to compute on each one of them; *structure types* provide such facilities.

In particular, they provide (i) functions to evaluate whether an object is member of a *structure*, (ii) ways to define the values corresponding to the *memory profile* of a *structure*, and (iii) factories to identify all *structures* in the *heap*.

As mentioned, the **objects in the heap form a directed graph; a profiler must iterate over such objects to identify whether they belong to a structure**. *Our approach is a mechanism to specify what to do while the graph is being traversed.* The following examples show what kind of operations must be executed during graph exploration.

Objects reachable from a specific class object Figure 6.1 depicts a diagram of objects. Suppose we want a profiler to compute the number of objects that can be reached from the object of class *java.lang.Class* whose *classname* is "Client" (see Figure 6.1). Informally, this profiler must execute three steps to obtain the desired data. First, it iterates over the objects to find all the instances of class *java.lang.Class*. These objects are then explored to get the value of attribute *classname*, and see if it is equal to "Client"; in this way we can locate the instance of interest. Finally, the profiler traverses the graph (as in Depth First Search) using the node found in the previous

step as root, and counting the number of traversed nodes. In Figure 6.1 the objects to count are highlighted.

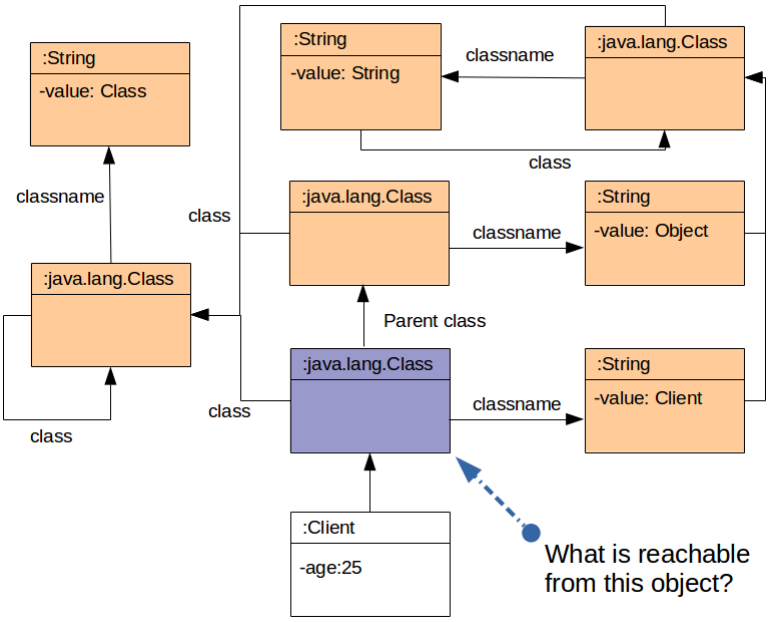


Figure 6.1 – Objects reachable from the Client class. Observe that only one object is not reachable.

The aforementioned informal steps show that a profiler should be capable of:

- **Collecting meta-data of objects** such as the class of an object (*java.lang.Class*), and the size of an object.
- **Reading the value of attributes** is also helpful to filter out elements of the heap.
- **Traversing references** is necessary because often some objects belong to a structure only because they are referenced by an object that is already a member.

To summarize, a function to determine whether an object is member of a structure may use: properties of the object itself, and properties about its relationships.

Nodes in each singly linked list In the second example, we want to compute how many nodes has each singly linked list in the heap. Figure 6.2 shows a heap with two singly linked list and one double linked list. The first list has three nodes while the second one has four; these are the values we want to obtain.

Informally, to compute such data a profiler must traverse the heap, checks whether an object's class is *NodeEntry* (membership function), and increases a counter every time an instance is found (function to compute memory profile). Unfortunately, these

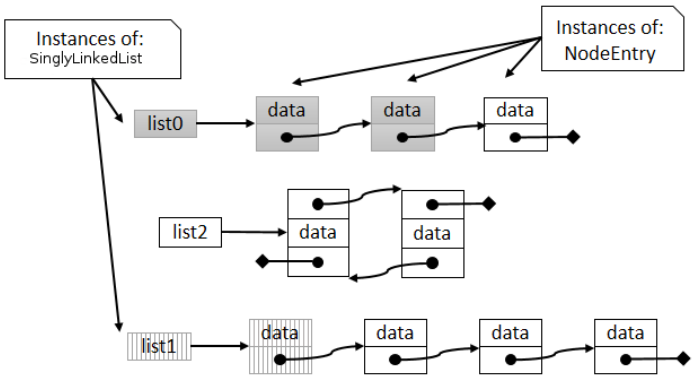


Figure 6.2 – Memory snapshot with three linked lists

two steps are insufficient to correctly compute the fact that there exist two lists. Indeed, instead of the two values 3 and 4, a single value – seven – is obtained when this procedure is used. The problem is that we have two structures instead of one. However, using the aforementioned membership function, it is not possible to know whether a *NodeEntry* belongs to a given structure. In other words, we need additional information to distinguish the two structures.

A method to solve this particular problem is to use the following recursive membership function: an object is member of a structure S if it is an instance of *NodeEntry* and it is being referenced by a member of S. For the non-recursive case, we can see how each singly linked list starts with an instance of *SinglyLinkedList*. This is easily represented with a parametrized and recursive membership function $f_{head} : Objects \mapsto bool$:

$$f_{head}(O) = \begin{cases} head = O & O\ is\ \text{SinglyLinkedList} \\ \exists x \in \text{Objects}, \quad x \text{ references } o \wedge f_{head}(x) & O\ is\ \text{NodeEntry} \\ \text{false} & \text{otherwise} \end{cases} \tag{6.1}$$

The parameter *head* represents the structure of interest, and it is an instance of *SinglyLinkedList*. In this example, we have two functions, in which only the parameter varies, because there are two instances of class *SinglyLinkedList*. Similarly, we need two counters to store the number of nodes while we are traversing the graph of objects; once again, they (and the function to compute their values) are parametrized by the *head* of the list. Figure 6.2 depicts a snapshot in time of the heap while it is being explored; the shaded nodes are those that were already explored.

This example shows that in defining memory profilers, we often need:

- To use the same functions for many structures, varying only some parameters. This is necessary for both membership functions and to compute the memory profile.

- To identify structures in the heap by using a parameter. For instance, in this example, we know that there are two structures because there are two instances of class *SinglyLinkedList*.

To summarize, we are frequently interested in calculating values for many structures that have the same "characteristics"; this is what we call *structure type*. To identify such structures we use a *parameter*; the process of associating a particular parameter value to a structure type is done by a *factory of structures*.

6.2 Language to define customized memory profilers

A global view of our approach is shown in Figure 6.3. Since our goal is to support resource-aware programming, in this architecture an application can collect data on its own memory consumption. To provide this feature, we add a layer (data collection layer) to MRTEs to take care of:

- Providing a set of interfaces for collecting meta-data of objects in the heap, iterating over such objects, and reading the value of their fields. These interfaces are used by memory profilers.
- Providing a set of interface for accessing memory profilers from applications. In other words, an application may trigger the execution of a memory profiler, and it should be able to analyze the values computed by the profiler.
- Supporting dynamic loading of memory profilers. This is helpful in production environments for loading new dynamic analysis tools without having to stop the MRTE.

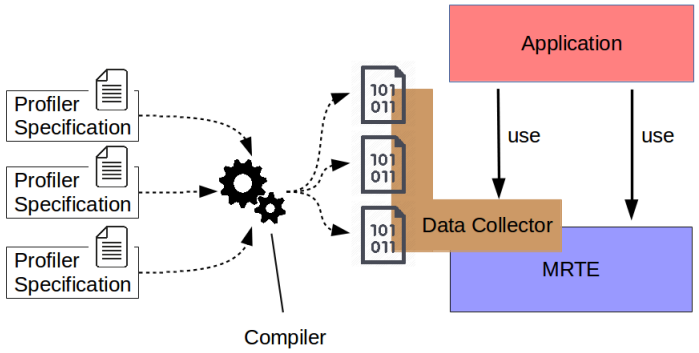


Figure 6.3 – Global view of the system. In this case, three memory profilers are defined.

In Figure 6.3, three memory profilers are plugged to the data collection layer. The exact mechanism to collect raw data from the MRTE, as well as the mechanisms to dynamically load profilers, are platform specific.

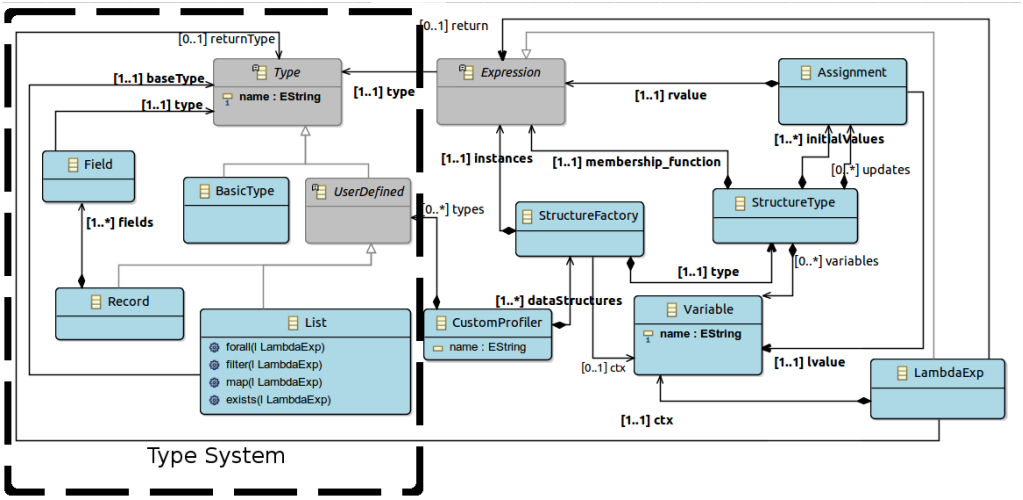


Figure 6.4 – Meta-model for representing customized memory profilers

A memory profiler can be handcrafted and plugged in this architecture; this is, nevertheless, error-prone. Instead, we favor a generative approach where profilers are written using a metalanguage that hides low-level details. A compiler is then used to transform such definitions into a binary form that can be executed in the runtime environment. This compiler is also in charge of providing interoperability between the data format provided by low-level facilities of the MRTE, and the high-level data format expected by applications running on top of such execution environments.

In the rest of this section, we describe the elements of this architecture. First, the abstract syntax of this metalanguage is discussed; we present it using a meta-model with its main concepts as well as some snips of code, written in a concrete syntax, to ease the presentation (see Subsection 6.2.1). The full concrete syntax is then introduced in Subsection 6.2.2, followed by the translational semantic of the language (see Subsection 6.2.3). Finally, in 6.2.4 we present some examples to illustrate the use of the language.

6.2.1 Abstract Syntax

The meta-model shown in Figure 6.4 describes the abstract syntax of our language. The main concept of this meta-model is a *CustomProfiler* which is composed of *UserDefined* types and *StructureFactories*. This means that a developer must focus on declaring the types to store information about memory usage, and defining what are the *structures* of interest by mean of *factories*.

The concepts related to *UserDefined* types are depicted on the left part of the metamodel, while the right part describes the *StructureFactory* which represents both the set of structures to identify and the value to compute on these structures.

User-defined types

The language support three basic types – *Integer*, *String* and *Boolean*. They are not shown in the meta-model for the sake of clarity, and because these types behave mostly as in any other language. It is however noteworthy that the language intentionally lacks support for implicit casts, and explicit casts exist in the form of built-in functions.

In addition, the language supports the declaration of both *Records* and *Lists*. A *record* is a compound type that contains *fields* to hold values of previously defined types. On the contrary, all the members of a list must be of the same type; hence, a *list* refers to a *base type*. In a profiler, *UserDefined* types can be composed in any desired way as long as a single property is respected – no recursive types are allowed. We can formalize such a constraint using the Object Constraint Language (OCL) [1]):

```
context List inv:
    not self.baseType->closure(t:Type|
        if t.oclIsKindOf(List) then
            t.oclAsType(List).baseType
        else
            if t.oclIsTypeOf(BasicType) then
                t
            else
                t.oclAsType(Record).fields.type
            endif
        endif
    )->exists(t | t = self)
```

```
context Record inv:
    not self. fields .type->closure(t:Type|
        if t.oclIsKindOf(List) then
            t.oclAsType(List).baseType
        else
            if t.oclIsTypeOf(BasicType) then
                t
            else
                t.oclAsType(Record).fields.type
            endif
        endif
    )->exists(t | t = self);
```

In this OCL code (written in Xtext OCL), the closure function computes the set of types that can be reached from a given type definition when the relationships among types are followed. We enforce this constraint for two reasons. First, it simplifies the type system, and second because it is a way to know that no data structure other than the built-in will be used to represent, for instance, a *list*. In this way, we can guarantee the performance of operations on lists. The *List* type provides operations to manipulate list values. In general, these operations correspond to a subset of *standard* operations one can find in any implementation of the list data type in functional languages. Table 6.1 shows the name of these operations as well as their signature.

filter	('a → boolean) → 'a list → 'a list
forall	('a → boolean) → 'a list → boolean
exists	('a → boolean) → 'a list → boolean
map	('a → 'b) → 'a list → 'b list
add	'a → 'a list → 'a list
first	'a list → 'a

Table 6.1 – Operations on lists

[1] `http://www.omg.org/spec/OCL/`

In Listing 6.1, a data structure to store information about a heap structure is declared. First, a list (tableOf) of string is declared, it may be used to store the class name of objects. Afterwards, a *record* with two integer fields and a list field is declared in line 2. This record may be used to hold values during profiling.

```
1 names: tableOf string
2 data: struct {
3     total_size: int
4     nr_objects: int
5     classnames : names
6 }
```

Listing 6.1 – Declaring types to store the number of object in a structure, its total size, and the class name of each object.

Defining structures to profile

Defining *StructureFactories* is at the core of writing a customized profiler. A *StructureFactory* contains an *Expression* through the *instances* relationship, which indicates a pattern to identify structures in the memory heap. Notice that a single instance of *StructureFactory* identifies many structures in memory; thus, the *Expression* is a list – a new structure must be instantiated for each element of the list.

Listing 6.2 shows a snip of code that defines a structure for each *SinglyLinkedList* in the heap. This is useful to solve one of the example we mention in Section 6.1. Notice that '*e*' is the *value* used to parametrize the structures. In this case, '*e*' will take the values in the list defined after ':'; such a list is composed by instances of class *SinglyLinkedList*. In this example, we use the operation *filter* on a built-in list – *objects*. The actual parameter for *filter* is a lambda expression that only return true if its parameter is an instance of the expected class.

```
1 create structure foreach e: objects.filter ([
2         o |
3           ret o is SinglyLinkedList
4         ]) using
```

Listing 6.2 – Defining a factory that instantiates a structure for each instance of the class SinglyLinkedList that it can find in the heap.

Defining a new *StructureFactory* implies defining a *StructureType*. This concept describes the mechanism used to identify a structure and compute some values. A *StructureType* is composed of *Assignments* that are used as *initialValues* for each *Variable* holding a value of the structure – similar to a constructor in OOP. In addition, a *StructureType* contains a boolean *expression* which is the *membership function* used to decide whether an object is member of the structure. Finally, it also contains *assignments* to update the value of each *variable* every time an object is added to the structure. The major constraint regarding these *updates* is that they must refer to already initialized *variables*, and the new assigned values must match the previous types. We formalize such a constraint using OCL:

```
context StructureType inv: updates->forAll(a: Assignament |
    self. initialValues ->exists(aa : Assignament |
      aa.lvalue = a.lvalue and aa.rvalue.type = a.rvalue.type
    ))
```

In Listing 6.3, we show how to compute the length of each *SinglyLinkedList* in the memory snapshot depicted in Figure 6.2. We use the fact that each *NodeEntry* points to the next element in the list. Since we only want to count the number of nodes in each list, we simply define an *integer* variable in the structure type – '*n*'; in line 7, this variable is updated every time an object is added to the structure. The variable '*e*' is used to parametrize the structure; observe how, in line 3, a list of members of the structure is initialized ('*e*' is its only member). This line corresponds to defining the non-recursive case of the membership function. Line 1 was already discussed, it is just worth highlighting that only two structures will be built because there are two singly linked lists in the memory snapshot. The first member of these structures are in one case *list0*, and *list1* in the other case.

```
1 create structure foreach e:objects.filter([o| ret o is SinglyLinkedList]) using
2   constructor
3     initialObjects = #[e] // a list literal with one element: e
4     n = 0
5   membership (this is NodeEntry) and (referrer in this_structure)
6   updates
7     n = n + 1
```

Listing 6.3 – Calculating the length of each singly linked list.

The recursive case of the membership function is in line 5. It looks like the definition in function 6.1; there are however differences that are due to the evaluation semantic of our language. Although the semantic is discussed hereafter, we can understand the definition by simply knowing that this boolean expression is evaluated for every edge of the directed graph. Before the evaluation, three variables are created: **this**, **referrer**, and **this_structure**; they are, respectively, the target of the edge, the source, and a structure. When we evaluated the membership function we are checking whether **this** is a member of **this_structure**, and we are passing additional information to support the definition of recursive functions.

Finally, there is a built-in variable in each *StructureType* (line 3) that is only accessible during initialization. Its type is predetermined as part of the language specification. We force the use of the proper type using OCL:

```
context StructureType inv: initialValues->exists(a: Assignment |
    a.lvalue.name = 'initialObjects' and a.rvalue.type.oclIsTypeOf(List)
  )
```

Expressions and Types

For the sake of readability, Figure 6.4 only shows a few concepts related to expressions. In addition to *arithmetic*, *boolean* and *literals* for basic types, the language includes

lambda expressions, literal for records and lists.

Observe that *variables* do not refer to a *Type*; indeed, the language is strongly typed, and the *type* of each user-defined variable is inferred from its initial value. To support type-inference, the type of each expression is clearly defined.

Built-in rvalues, which are nothing but expressions initialized by the runtime within a specific scope, and their types are also defined. There are two types of built-in rvalues – platform independent and dependent. Among the firsts, we have the list of ***objects***, the list of *loaded classes* (**classes**), a reference to the *current data structure* (**this_structure**), and a reference to the *current object* (**this**). Target dependent *rvalues* include the list of *threads* (**threads**) and the type of reference (**reference_kind**).

Built-in rvalues are only valid in specific contexts. For instance, **this**, **referrer**, **this_structure**, and **reference_kind**, exist either during the evaluation of the membership function or when the variables are being updated. On the contrary, the values **objects**, **classes**, and **threads**, are valid only during the creation of structures and their initialization. Since building rvalues of type list is costly in terms of performance overhead, we decide avoiding their use in membership functions and updates.

6.2.2 Concrete Syntax

A textual concrete syntax has been defined for our language. It can be utilized by a domain expert to define a customized memory profiler using a text editor. A grammar for this textual representation is depicted in Figure 6.5. Notice how the rules for expressions make such a grammar ambiguous; we decide to use this form to ease the presentation. However, a complete LL(*) grammar [PF11] can be found in Appendix A. One interesting aspect of this concrete language is its relative verbosity. For instance, it uses very explicit keywords such as "create structure foreach", "constructor", and "membership".

An interesting aspect of this grammar is how **list** and **struct** values are defined. In particular, they can be created from non-constant values in the right hand of the evaluation. For instance, in Listing 6.4, three values are created; line 1 instantiates a list of integers with two elements, line 2 defines a value of type *Point*, and line 3 creates an empty list of *strings* that is immediately populated with one element. Notice that line three requires writing the type of the list because it is not possible, in general, to infer the list type when it is empty.

```
1 l = #[ 4, n + 12]
2 s = struct Point { x , 14 }
3 m = #String[].add("first element")
```

Listing 6.4 – Creating list and struct values.

6.2.3 Translational Semantics

A profiler written in this language is compiled to produce a customized memory profiler for a specific target platform. For instance, in our reference implementation, the

⟨*program*⟩ ::= '`name`' ⟨*string-literal*⟩ ⟨*types*⟩ ⟨*structures*⟩ (1)

⟨*types*⟩ ::= ⟨*type*⟩ ⟨*types*⟩ | ⟨*empty*⟩ (2, 3)

⟨*type*⟩ ::= ⟨*id*⟩ '`:`' '`tableOf`' ⟨*id*⟩ | ⟨*id*⟩ '`:`' '`struct`' '`{`' ⟨*fields*⟩ '`}`' (4, 5)

⟨*fields*⟩ ::= ⟨*id*⟩ '`:`' ⟨*id*⟩ ⟨*fields*⟩ | ⟨*id*⟩ '`:`' ⟨*id*⟩ (6, 7)

⟨*structures*⟩ ::= ⟨*factory*⟩ ⟨*structures*⟩ | ⟨*factory*⟩ (8, 9)

⟨*factory*⟩ ::= '`create structure foreach`' ⟨*id*⟩'`:`'⟨*e*⟩ '`using`' ⟨*body*⟩ (10)

⟨*body*⟩ ::= '`constructor`' ⟨*s*⟩ '`membership`' ⟨*expr*⟩ '`updates`' ⟨*s*⟩ (11)

⟨*s*⟩ ::= ⟨*a*⟩ ⟨*s*⟩ | ⟨*empty*⟩ (12)

⟨*a*⟩ ::= ⟨*id*⟩ '`=`' ⟨*e*⟩ (13)

⟨*e*⟩ ::= ⟨*e*⟩ ⟨*op*⟩ ⟨*e*⟩ | ⟨*u-op*⟩ ⟨*e*⟩ | ⟨*e*⟩ '`in`' ⟨*id*⟩ | ⟨*e*⟩ '`is`' ⟨*id*⟩ (14, 15, 16, 17)
| ⟨*e*⟩ '`.`' ⟨*id*⟩ '`(`' ⟨*expr-list*⟩ '`)`' | ⟨*e*⟩ '`.`' ⟨*id*⟩ (18, 20)
| '`#`' ⟨*l-type*⟩ '`[`' ⟨*expr-list*⟩ '`]`' | '`struct`' ⟨*id*⟩ '`{`' ⟨*expr-list*⟩ '`}`' (21, 22)
| ⟨*int-literal*⟩ | ⟨*string-literal*⟩ | ⟨*bool-literal*⟩ (23, 24, 25)
| ⟨*id*⟩ | '`(`' ⟨*e*⟩ '`)`' | '`[`' ⟨*id*⟩ '`|`' ⟨*s*⟩ '`ret`' ⟨*e*⟩ '`]`' (26, 27)
| ⟨*e*⟩ '`?`' ⟨*e*⟩ '`:`' ⟨*e*⟩

⟨*expr-list*⟩ ::= ⟨*e*⟩ '`,`' ⟨*expr-list*⟩ | ⟨*empty*⟩ (28)

⟨*l-type*⟩ ::= ⟨*id*⟩ | ⟨*empty*⟩

⟨*op*⟩ ::= '`+`' | '`*`' | '`-`' | '`/`' | '`and`' | '`or`' | '`>`' | '`<`' | '`>=`' | '`<=`' | '`==`' | '`!=`'

⟨*u-op*⟩ ::= '`-`' | '`not`'

Figure 6.5 – Concrete grammar of the language. For the sake of clarity, we are using an ambiguous grammar to describe the expression language.

compiler produces a library written in *C++*, which is in charge of collecting the desired information from the runtime environment. The generated source code is a set of classes. In particular, for each *StructureType* in the model, the compiler generates a subclass of *AbstractStructure*, which is shown below. Every subclass contains attributes to store the variables used in the associated *StructureType*. In the listing below, the class *Context* holds built-in rvalues such as **objects**, **this**, and **referrer**.

```
1 class AbstractStructure {
2 public:
```

```
3       virtual void  initialize (const Context& ctx) = 0; // correspond to constructor
4       virtual bool membership(const Context& ctx) = 0; // membership function
5       virtual void update(const Context& ctx) = 0; // update variables
6   }
```

Listing 6.5 shows the code generated by the compiler for the singly linked lists example (see Listing 6.3). Notice the use of three functions provided by the runtime, *markAsMembers*, *isInstance*, and *isMember*. Lines 6-8 correspond to the creation and assignment of the built-in value *initialObjects*. Meanwhile, line 9 is simply the initialization of the value n associated to each structure. In lines 12 and 13, the membership function is evaluated. The translation process does not optimize the code for expressions; instead, it produces *triples* and relies on the *C++* compiler to optimize expressions. The method to update is a simple translation from the concrete syntax.

```
1   class SinglyLinkedListStructure1 : public AbstractStructure {
2   private:
3       int n;
4   public:
5       void  initialize (const Context& ctx) {
6           vector<DSL_Object> l;
7           l.push_back(ctx.e);
8           markAsMembers(l, ctx.this_structure); // part of the language runtime support
9           n = 0;
10      }
11      bool membership(const Context& ctx) {
12          bool b0 = isInstance(ctx.this, "NodeEntry"); // part of the language runtime support
13          bool b1 &= isMember(ctx.referrer, ctx.this_structure); // part of the runtime support
14          return b1;
15      }
16      void update(const Context& ctx) {
17          int i0 = n + 1;
18          n = i0;
19      }
20      static void createStructures(const Context& ctx, vector<DSL_Object>& instances) {
21          vector<DSL_Object> r;
22          auto f = [](DSL_Object o) { return isInstance(o, "SinglyLinkedList")};
23          ctx.objects. filter (f, r); // add valid elements to r
24          instances.insert(instances.end(),r.begin(),r.end());
25      }
26  }
```

Listing 6.5 – To represent a structure type, we declare a subclass of AbstractStructure.

The last function, *createStructures*, is used to identify structures of this type. In other words, it implements the factory. Such a function is invoked by a template initialization routine that expects as generic parameter a subclass of *AbstractStructure* (T in Listing 6.6). Formally, the signature and behavior of this routine are as follows:

```
1   template <typename T> void
2    initializationRoutine (Context& ctx, std::vector<AbstractStructure*>& s){
3
4      vector<DSL_Object> instances;
5      T::createStructures(ctx, instances);
6
```

```
7     for (DSL_Object obj : instances) {
8       AbstractStructure* ns = new T();
9
10      ctx.e = obj;
11      ctx.this_structure = defineStructure(ns); // part of the runtime support
12
13      ns->initialize(ctx);
14      s.push_back(ns);
15    }
16  }
```

Listing 6.6 – Routine to initialize structures in the heap.

In line 5 the factory is invoked to identify the set of structures in the heap. This simply add an element to the list for each structure. Afterwards, the internal format used to represent such structures are created and the structures initialized.

Assembling a profiler The final profiler is built using both the generated code and a template algorithm. This algorithm is target dependent, but in general we use the underline target facilities to collect meta-data, access fields in certain steps, traverse the objects in memory and also to populate the built-in rvalues. Informally, the idea is traversing a graph in which nodes and edges have properties, and executing *callbacks* to collect data about the graph. A simplified version of this algorithm is shown in Listing 6.7. The model of execution of profilers written using our approach is summarized in such an algorithm.

```
1  values:
2      structures: vector<AbstractStructureType*>
3  routine:
4      foreach (StructureType ST)
5          create context
6          call initializationRoutine <ST>(context, structures)
7
8      foreach (r: references among objects)
9          if (r.target has no membership)
10             create context // context.this = r.target
11             S = structures. findfirst (s | s.membership(context))
12             make context.this a member of S
13             S.update(context)
14     return structures
```

Listing 6.7 – Template algorithm to collect data with in a memory profiler. This summarizes the model of execution of profilers written using our approach

There are two loops in the algorithm. The first loop is in charge of creating the set of structures the program is intended to collect information about. The creation of a context in line 5 depends on the target platform. It basically creates values such as

the list of *objects* in memory or the list of loaded *classes*. The second loop traverses all the references among objects in memory. During each iteration, the algorithm finds the "*first*" structure for which the membership function is true (the if in line 9). Thereafter, the information for such a structure is updated. Notice that we only select one because this is a simple way to guarantee that structures are disjoints sets of objects. For instance, suppose the graph of object is $G = (\{A, B, C\}, \{\langle A, C\rangle, \langle B, C\rangle\})$, and we want to identify the members of two structures S_A and S_B using the following membership functions (parameter T is either A or B):

$$f_T(O) = \begin{cases} T = O & O \; is \; \text{Class0} \\ \exists x \in \text{Objects}, \quad x \,\text{references}\, o \wedge f_T(x) & O \; is \; \text{Class1} \\ \text{false} & \text{otherwise} \end{cases}$$

In this case, C is member of both structures, S_A and S_B. Since we want to guarantee that structures are disjoint set, we need an additional mechanism to achieve this. In our implementation, we simply consider that an object O is member of the "*first*" structure fo which evaluating the membership function produces true. In general, a structure S_0 is evaluated first than S_1 if its *StructureFactory* is defined earlier in the source code of the program; this is deterministic. Also, a structure is evaluated first if the parameter to identify the structure appears earlier in the list of instances produced by the factory; this case is non-deterministic.

To summarize, we need to enforce additional execution rules to guarantee that structures are disjoint sets. As a consequence, every membership function $f(o)$ one defines is actually evaluated as $f(o) \wedge \nexists S_T \in \mathit{Structures}, f_T(o)$.

6.2.4 Language Usage

There are several possibilities for using our DSL in the various stage of an application lifecycle. For instance, checking invariants of data structures, computing memory consumption of different software abstractions, and checking reachability properties. Some of these uses may help to support resource-aware programming, others may be useful to support additional tasks in software development. In this section, we discuss some examples to highlight possible uses of our language.

Example 1: Evaluating an assertion The first example shows how to assess the existence of a value satisfying a property, independently of which object contains it. Specifically, we want to check whether exists an object with an attribute named "data" whose value is between *3* and *5*. Since the answer we expect is a simple boolean value, this example can be seen as a kind of assertion regarding the memory content.

We can write a memory profiler to compute this value. A complete code to implement it is depicted in Listing 6.8. Since we want to compute a single boolean value that depends on all the objects in the heap, this profiler only needs to create one structure. In the code, this is shown in line 1, where the list of instances has one element. Notice that we use the string value "whole-jvm", this has been arbitrarily chosen, any value

is acceptable in this case as long as the list's length is one. The initialization section simply states that, before the graph of object is traversed, we assume that no object meets the property. The non-recursive membership function accepts every object is member of the structure. Finally, to update the variable, we assess the property for the current object (**this**), and aggregate the result to the previous value.

```
1 create structure foreach e:#["whole-jvm"] using
2   constructor
3     initialObjects = #Object[]
4     existValue = false
5   membership true
6   updates
7     existValue = existValue or (this.data >= 3 and this.data <= 5)
```

Listing 6.8 – Assessing the existence of an object with a given property.

Example 2: Counting instances of specific classes A common requirement in memory profilers is calculating the number of instances of a class. We can easily express this in our language; Listing 6.9 illustrates this usage scenario. In particular, it counts how many *Strings* and *Integers* exist. Since we are interested in two different classes, we declare two factories with their respective membership function.

```
1  create structure foreach e:#["instances of Integer"] using
2    constructor
3      initialObjects = #Object[]
4      n = 0
5    membership (this is java.lang.Integer)
6    updates
7      n = n + 1
8
9  create structure foreach e:#["instances of String"] using
10   constructor
11     initialObjects = #Object[]
12     n = 0
13   membership (this is java.lang.String)
14   updates
15     n = n + 1
```

Listing 6.9 – Calculating number of instances of two classes.

Example 3: Data about large objects Memory profilers also provide mechanisms to collect data on many objects at the same. In this example (see Listing 6.10), we show how to collect the class name and the size of very large objects. Only a structure is needed to define this profiler; both the membership function and the initialization of the structure are simple. To store the information, an empty list is declared in line 9. Every time a new object is identified as large, its class and size are added to the list as a new entry.

```
1 SingleObject: struct {
2     classname : string
```

```
3       size : int
4   }
5   listObjects : tableOf SingleObject
6   create structure foreach e:#["big objects"] using
7     constructor
8       initialObjects = #Object[]
9       data = #SingleObject[] // empty list of SingleObjects
10    membership (this.size > 1024)
11    updates
12      data = data.add( struct SingleObject { this.classname, this.size } )
```

Listing 6.10 – Collecting class and size of large objects (size > 1024 bytes).

Example 4: Consumption of threads The example in Listing 6.11 calculates the number of objects that are reachable from the threads. It also computes how much memory these objects consume.

The membership function used in this structure type is as follows:

$$f(o) = \begin{cases} true & o \text{ is } Thread \\ \exists r \in \text{Objects}, (r \text{ reference } o) \wedge f(r) & otherwise \end{cases}$$

In this case, a recursive membership function is used to discard those objects that are not referenced by an existing member of the structure. A variable, defined as a record, holds the calculated values; it is however possible to achieve the same goal using two separated integer values.

```
1   Info : struct {
2       nbObjects : int
3       size : int
4   }
5   create structure foreach e:#["whole-jvm"] using
6     constructor
7       initialObjects = threads
8       data = struct Info { 0, 0 }
9     membership ( referrer in this_structure )
10    updates
11      data = struct Info { data.nbObjects + 1 , data.size + this.size }
```

Listing 6.11 – Calculating objects reachable from threads.

A different result is obtained if we slightly modified this listing. Figure 6.12 shows a snip of code of a profiler that calculates the number of objects reachable for each thread instead of for the whole MRTE.

```
1   ...
2   create structure foreach t:threads using
3     constructor
4       initialObjects = #[t]
5       data = struct Info { 0, 0 }
6     membership ( referrer in this_structure )
7     updates
8       data = struct Info { data.nbObjects + 1 , data.size + this.size }
```

Listing 6.12 – Counting the memory consumed for each thread.

In comparison to the previous listing, only two lines are modified. However, these lines modify the non-recursive case of the membership function, as well as the number of structures identified in the heap. As a consequence, the result computed has type *tableOf Info*, where each element of the list corresponds to a thread.

Example 5: Memory consumption of K3Objects We can also identify other complex structures. For instance, to find the consumption of each K3-Al object namely *K3Object* as described in Section 3.1, we must find instances of *HashMap.Entry* that have a *K3Object* as *key*. These entries should be added to the consumption of the object *K3Object* as well as all the objects reachable from the *HashMap.Entry.value*. Figure 6.6 depicts a memory snapshot with a K3Object and two HashMap.Entries pointing to it.

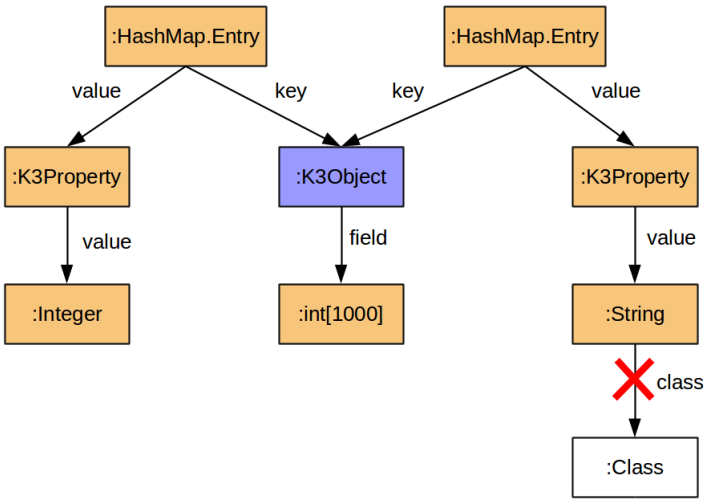


Figure 6.6 – Snapshot of memory with one K3 Object. The idea is to compute the memory used by the shaded objects.

To define this profiler, a good understanding of the implementation of K3-Objects, and their aspects, is needed. We define a factory where a list of *K3Objects* is used to identify the structures (see line 1 in Listing 6.13). Hence, there will be as many structures as instances of class *K3Objects*. In line 3, the initial members of each structure are defined; notice that the membership function is parametrized by a K3Object instance, namely '*e*'. The recursive case is then similar to previous examples, the difference is that we don't count instances of Class; this is shown in Figure 6.6).

```
1 create structure foreach e:objects.filter([it|it is K3Object]) using
2   constructor
3     initialObjects = objects.filter ([
4                        o| ret (o is HashMap.Entry) and (o.key == e)
5                      ]).add(e)
6     size = 0
7   // the second part can be more precisely written as reference_kind != class_ref
8   membership (referrer in this_structure) and not (this is java.lang.Class);
9   updates
```

```
10 ||      size = size + this.size
```

Listing 6.13 – Computing the consumption of each K3-Al Object along with its aspects.

6.3 Tooling

To validate our approach, we have implemented a tool chain to ease the definition of customized memory profilers for Java-based systems. [2] These profilers can be executed in any JVM as long as it provides support for the JVMTI.

In this section, we present tools built to support the definition of memory profilers using our language; this is done by taking into account how engineers in different roles may interact with these tools and with the resultant profilers. Indeed, in dealing with memory profilers, we have to take into consideration the two usual roles – developers of profilers and their users; after all, profilers built using our language are themselves software abstractions. A developer must know how the target domain-specific abstractions are represented on top of the JVM, and she/he must also have a clear understanding of how our language is executed. On the contrary, users only need to be aware of the interface provided by our framework, and the structure of the data collected by a profiler. In the rest of this section, we discuss details that are important to these roles.

In this section, we also present low-level details regarding how the language is implemented on top of the JMVTI. The decision of implementing our approach by relying on JVMTI has advantages and disadvantages. On the one hand, the obvious advantage lies on the portability of this solution, which makes it more valuable from a practical point of view. On the other hand, building profilers on top of the JVMTI; instead of directly modifying the JVM, impacts the performance of the generated profilers and, unfortunately, hinders (in extreme case it even prevents) the implementation of some language constructs. Nonetheless, it is our belief that guaranteeing profilers' portability should be of maximum priority. Moreover, in writing this implementation, we have found that the limitations in the JVMTI preventing the construction of better profilers can be overcome with, at most, a few additions to the API.

6.3.1 Developers of domain-specific abstractions

In our vision, developers of software libraries and component frameworks, as well as software language engineers may use our approach to define customized memory profilers for the abstractions they create. This is, in addition to delivering artifacts such as libraries, source code, simulators, text editors for DSLs, and compilers for these DSLs; engineers would also ship profilers to simplify the use of these abstractions. For instance, the developers of the Spring framework [3] may create a set of specific profilers to reduce the cost of maintaining applications written using the framework. These profilers can

[2] Available at: `https://github.com/intigonzalez/heapexplorer_language`

[3] `https://spring.io/`

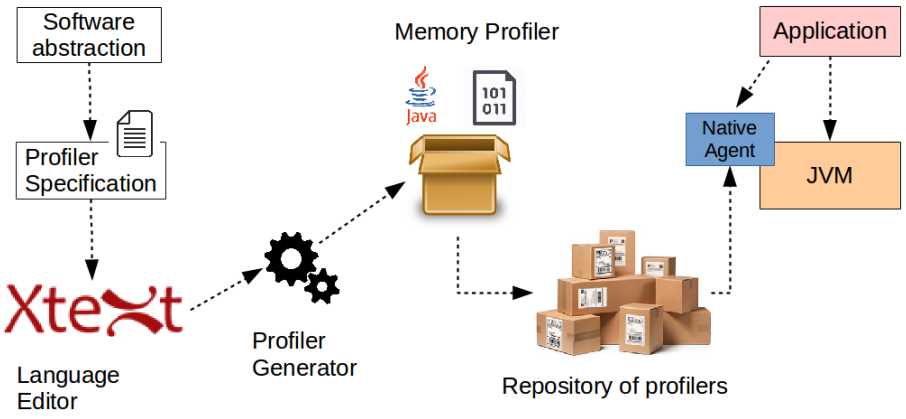


Figure 6.7 – Developer viewpoint. Memory profilers are built from the description of software abstractions.

serve as both internal tools to help in the development of abstractions, and mechanisms allowing users to better use abstractions.

Figure 6.7 summarizes the viewpoint of developers of domain-specific abstractions. To write a profiler, they use knowledge about the abstraction and the tool chain to generate the executable profiler. Our implementation of the language is built using Xtext [EB10]; it provides a textual editor that is able to handle the proposed concrete syntax. This editor provides syntax highlighting, error detection during editing, auto-completion, and compilation to native Java agents written in *C++*.

To perform low-level tasks related to memory profiling, we use JVMTI [4] and JNI. These APIs are used by both profilers and the core memory profiling library, so-called Native Agent in Figure 6.7. In this native agent, a *plugins* system, which allows users to load/unload profiles without shutting down the JVM, is implemented. Given a profiler definition, the compiler output is a package that contains the native binary code for the profiler, and a Java library you can use to access the collected data using plain Java objects.

To reduce the overhead of profilers, developers must be aware of the details of the abstraction for which the profiler is being built, the semantic of our language, and the details of its implementation. In particular, it is advisable reducing the usage of *lists* and the evaluation of nested lambda expressions. Likewise, heavily using the built-in rvalue *objects* is especially discouraged because it can easily contain many elements. It is also discouraged because, in order to reduce memory consumption, we rely on an iterator built on top of JVMTI operations that can be costly to use in terms of CPU time.

Finally, we added some built-in rvalues in this implementation because they are both useful in the context of Java and easy to obtain using the JVMTI. These values are *classes*, *classloaders*, *threads* and *objects*; they are lists of anonymous built-in record types. The relation among these types and their operations are depicted in Figure 6.8.

[4] `http://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html`

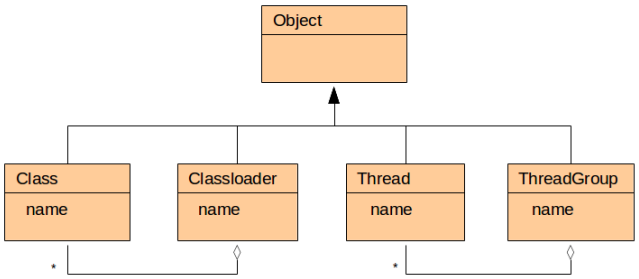


Figure 6.8 – Viewpoint of developers. Memory profilers are built from the description of software abstractions.

6.3.2 Users of domain-specific abstractions

We envision that a set of memory profilers can be shipped in addition to other "classic" deployment artifacts that users of a software abstraction receive. These profilers would support the use of the corresponding software abstraction. For instance, a user who is relying on a new extension of the Xtend language to build a system, may use specific profilers written in our language to understand the memory consumption, and in general, the behavior of the system.

The generated profilers can be used in two different ways, either as development tools or as mechanisms to support resource awareness at runtime. Due to the scope of this thesis, the reference implementation we provide is biased towards the second scenario, but it should be relatively simple to adapt it to support the software development process. To access memory profilers, a JVM must be launched with a native Java agent loaded, and a library to collect profiling data in its classpath. Once the application is running, it can trigger profiling by simply issuing a few method calls using the profiling API. Figure 6.9 illustrates the process of collecting memory profiles, the software components involved, and the APIs that must be used. Observe how the profiling framework issues a call to a handler once it is done, a parameter contains the data computed. These data are encoded in a *list*, in which elements correspond to the data computed for each identified structure in the heap.

The output of a profiler is a list of Java objects containing the collected information; and the type of these objects depend on the profiler definition. Indeed, as part of our implementation, the profiler generator creates a set of Java classes to represent the data collected in a form that is easy to digest at runtime by a Java application. Once a profiler collects the information in an internal format, it populates a representation in Java using the Java Native Interface (JNI); the code to do so is also generated by the compiler of our language. In Figure 6.10, the classes generated for a profiler are shown. Notice that a class is created for each *record* declared, and also for each *StructureType*. It can be seen how 'lists' are directly represented in Java by mean of generic Java lists. The *id* field in both *MemoryProfile1* and *MemoryProfile2* is the value used to parametrize each structure. In this particular example, where two structures are identified, the value of *MemoryProfile1.id* is "lists" and the value of *MemoryProfile2.id* is "otherObjects".

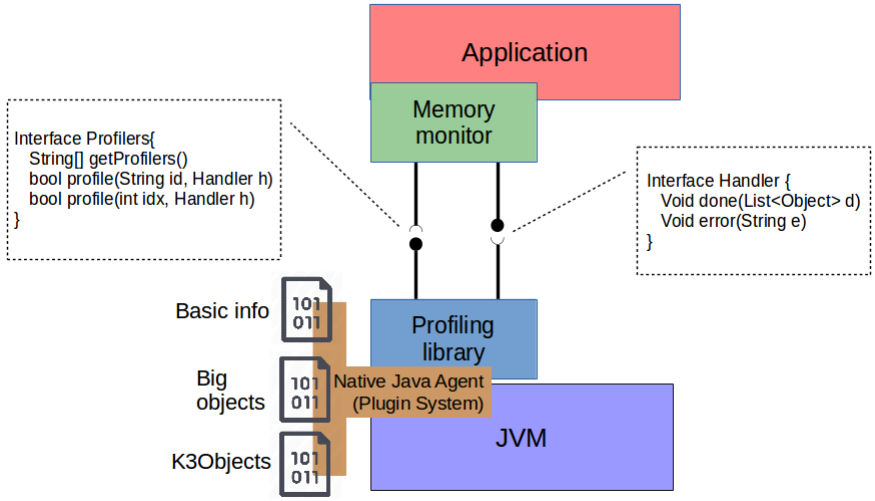


Figure 6.9 – Viewpoint of users. Memory profilers are black-boxes accessed through Java interfaces. Data collected is in the form of plain Java objects.

Given the fact that the data computed by a profiler is returned as a list of objects, and their layout is unclear, the remaining problem is how to process such data; there are two options. First, users can make the application code depends on the Java code created by the profiler generator. In this way, your application has a new dependency, but you can profit from knowing at development time the types used in the code. A second approach is using the reflection capabilities of Java to explore the data. In the evaluation, we use such an approach to log the result of an arbitrary profiler, printing all the information it has computed. Using reflection, it is also possible to build a user interface to explore the results in a customized way.

6.3.3 Implementation Details

Using JVMTI and JNI to create the built-in values *threads*, *threadgroups*, *classes*, and *classloaders* is simple. These APIs provide routines that one can use to obtain the information from the JVM. Since the number of threads (and classes) is relatively small, we simply store the data in a *vector* (included in the *C++* Standard Template Library (STL)) to reuse it every time we need it. On the contrary, creating the built-in value *objects* is challenging. Indeed, keeping a *vector* with references to all objects is not acceptable in terms of memory consumption because of the large number of objects in the heap. Fortunately, we can overcome this problem by noticing that iterating over the objects is usually enough to implement the language. For instance, in Listing 6.3 we need to iterate over the objects, and filter an object out if its type is not *SinglyLinkedList*. When this listing is executed in the memory snapshot shown in Figure 6.2, the result is a *vector* with only two elements.

To iterate over objects, we use the routine "*FollowReferences*". This function basically traverses the reference graph invoking a set of callback functions every time a

```
name "basic info"
T : struct {
    name : String
    size : int
}
create structeres for e:#[" lists "]
using
    constructor
        initialObjects = #Object[]
        data1 = #T[];
    membership (this is String) or (this is Array)
    updates
        data1 = data.add(struct T { this.name, this.size})

create structeres for e:#["otherObjects"]
using
    constructor
        initialObjects = #Object[]
        data2 = #T[];
    membership true
    updates
        data2 = data.add(struct T { this.name, this.size})
```

```
class T {
    final String name;
    final int size;
}

class MemoryProfile1 {
    final Object id;
    final List<T> data1;
}

class MemoryProfile2 {
    final Object id;
    final List<T> data2;
}
```

Figure 6.10 – Representation of profiling data in Java, as users of profilers see it. Accessing these structures is useful to support resource awareness.

new reference is found. In these callbacks, we generate the code to filter objects. For examples, Listing 6.14 illustrates how is the code generated for the expression:

```
objects. filter ([o | ret o is SinglyLinkedList])
```

Notice that accessing an object from within the callback is not possible. Instead, one can only access a **tag** associated to the object. This is the major limitation we find in implementing our language using the JVMTI. [5] Likewise, these callbacks offer metadata, such as the class and reference info, that we can leverage.

```

jint references(jvmtiHeapReferenceKind ref_kind, const jvmtiHeapReferenceInfo* ref_info,
      jlong class_tag, jlong referrer_class_tag,
      jlong size, jlong* tag_ptr, jlong* referrer_tag_ptr,
      jint length, void* user_data) {
  ...
  vector<DSL_TAG>* tags = (vector<DSL_TAG>*)(user_data);
  DSL_Class cl = getFromTag(class_tag);
  if (cl.name == "SinglyLinkedList")
      tags->push_back((DSL_TAG)*tag_ptr);
  ...
}

vector<DSL_OBJECT> v;
```

[5] This constraint is imposed in the implementation of the JVMTI function "*FollowReferences*" because no mutator code can be executed when the reference graph is being traversed. By using tags instead of objects, the JVM guarantees that no JNI function is invoked.

```
15 vector<DSL_TAG> tags;
16 callbacks.heap_reference_callback = references;
17 jvmtiEnv->FollowReferences(NO_FILTER, NULL, NULL, callbacks,&tags);
18 for (auto f : tags) {
19     jlong* a_tags = {f};
20     jint count_ptr;
21     jobject* object_result_ptr;
22     jlong* tag_result_ptr;
23     jvmtiEnv->GetObjectsWithTags(1, a_tags, &count_ptr, &object_result_ptr, &tag_result_ptr);
24     v.push_back(jobject2DSLObject(object_result_ptr[0]));
25 }
```

Listing 6.14 – Code generated for the expression *objects.filter*. This is a simplified version in C++14

An additional mechanism is required when a lambda expression for filtering *objects* is accessing attributes of an object. For instance, in Listing 6.8 a primitive attribute, "*data*", is accessed, while a reference attribute "*key*" is read in Listing 6.13. Reading primitive attributes is supported by the JVMTI function *FollowReferences*; by simply supplying the appropriates callbacks, you can get the value of every primitive field. However, this is not enough. The problem is that – to completely evaluate a lambda expression – the values involved in the expression must be collected in different execution contexts.

The solution we propose is based on using the functional library of *C++11* to implement continuations. In other words, we build *functions* that contain parts of the

```
1 [ o | o.data > 3 and
2       (referrer is String) and
3       (o.key is HashMap.Entry)]
```

```
1  references_callback() {
2      obj_tag = (DSL_TAG)*tag_ptr;
3      t0 = referrer_class.name == "String";
4      if (!t0) return;
5      obj_tag->fun = [=](String a, jvalue v) {
6        if (a == "data")
7          return t0 && v > 3;
8        return false;
9      };
10     vector<DSL_TAG>* tags =
           (vector<DSL_TAG>*)(user_data);
11     tags->push_back(obj_tag);
12 }
13
14 primitive_callback() {
15     obj_tag = (DSL_TAG)*tag_ptr;
16     t2 = obj_tag->fun(field_name, field_value);
17     obj_tag->fun = [=]() {
18       jobject obj = getFromTag(obj_tag);
19       jobject key = obj.key; // JNI
20       t3 = isInstance(key, "HashMap.Entry");
21       return t3;
22     };
23 }
```

Figure 6.11 – Evaluating an expression through partial evaluation.

evaluation of a lambda expression, the values already computed are stored in the closure associated to the *function*. Figure 6.11 shows the transformation of an expression. Observe how it requires collecting data in two different places, and partially evaluating the expression in three places.

Unfortunately, our solution has limitations. For instance, it cannot handle nested uses of the built-in value *objects* nor access primitive attributes of the referrer object. Indeed, so far we have discussed how to evaluate expressions that involve the list of *objects*, but evaluating the membership and update functions are problem with a similar solution. In other words, we also use the function *FollowReferences* to identify the *structures* in the heap.

We perform some optimizations related to the construction of the built-in values *threads*, *threadgroups*, *classes*, etc. Since not all memory profilers depends on such values, we selectively skip the construction of them. We also extend this to other cases. For instance, we do not find the class of each object when it is not required, and we avoid processing classes to obtain field names when they are not used in an expression. To implement these optimizations, we used a parametrized code template; therefore, the generated code depends on the values of generic parameters. We can tune them to satisfy our needs. Another optimization is reducing the number of nodes that must be traversed. As an illustration, we only produce code to explore primitive fields of each object, which are represented as leaf nodes in the graph, if there exists an expression accessing a field.

6.4 Discussion On Language Expressiveness

In our language, the mechanism used to collect data is explicit to the user – traversing a graph of objects. Hence, it is possible to estimate the overhead of a specific profiler. In other words, this language follows an imperative paradigm to obtain derived values. On the contrary, most query languages provide a declarative style because it *simplifies* the process of writing new queries.

We acknowledge that our approach limits the kind of memory analysis that users can express. First, it is not possible to recover all the information contained in the graph of live objects in linear time on the number of objects. Second, an imperative style forces the users to understand the underlying execution model, which is not required with declarative query languages. Nonetheless, we claim that getting rid of some expressiveness is a trade-off worth considering in order to guarantee efficient memory analysis. The empirical and theoretical evidence suggest that, in our language, *reducing the capabilities to collect data* has a bigger impact on performance gain than *generating efficient native code* to collect data.

At this point, it is worth noting why declarative approaches fail to deliver the adequate performance in production. Listings 6.15 and 6.16 show possible solutions, in OQL and Cypher/Neo4j, to the K3-Al example presented in Section 3.1. A naive comparison to the solution written in our language (see Listing 6.13) shows that the number of Source lines of code (SLOC) is similar in the three cases.

```
SELECT id, sum(size) as s
FROM (
    SELECT
        e.key.@objectId AS id,
        e.@usedHeapSize + e.value.@retainedHeapSize AS size
    FROM java.util.HashMap$Entry e
    WHERE (classof(e.key).@name = "K3Object")
    UNION ALL
    SELECT
        k3.@objectId AS id, k3.@retainedHeapSize AS size
    FROM K3Object k3
)
GROUP BY id
```

Listing 6.15 – Using OQL to compute the consumption of each K3-Al object. Actually, this query cannot be executed in Eclipse Mat nor in VisualVM since they do not provide a full OQL implementation.

There are two aspects that affects the performance of this kind of queries: the "natural" complexity of many queries, and the impossibility of applying optimizations due to the type of data. In the first place, many queries are intrinsically complex to answer. For instance, it is known that answering SPARQL queries - which was used as inspiration for Cypher/Neo4j, is PSPACE-complete [SML10, PAG09]. Second, the performance of declarative queries for dynamic memory analysis is also affected by the nature of data to process. In particular, even if some queries can be executed efficiently, the optimization steps required are in most cases impossible to execute for the type of data we are considering – a graph of objects that constantly changes. Indeed, these optimizations often require access to indexes, additional storage and multiples passes on the data [EGLGJ07, DZ02] that are not accessible on the graph of objects, and computing them may be costly by itself. On the contrary, our language makes explicit both the time and space complexities of analysis.

```
MATCH
    (key:K3Object)<-[:key]-(entry:HashMap$Entry)-[:value]->value
WITH entry, key, value
MATCH
    key-[:1..1]->fieldK
WITH entry, key, value, fieldK
MATCH
    value-[:1..1]->fieldV
RETURN key, entry.size + key.size + fieldK.size + sum(value.size) + sum(fieldV.size);
```

Listing 6.16 – Using Cypher to compute the consumption of each K3-A1 object.

A threat to validity of our approach is that we do not evaluate the usability of the language. Approaches based on existing languages, such as OQL, and CYPHER, suffer this problem far less because they are widely used in other areas. Nonetheless, our language (and its concrete syntax) is not entirely new. It resembles the "think as a vertex" paradigm of Pregel, which has proven to be successful [MAB+10]. In this paradigm, an algorithm on graph is described from the point of view of each vertex. In our case the *membership function* and the *update section* are also executed using a limited context, which only includes a few built-in rvalues. Likewise, the language is

largely inspired by the API of graph libraries; in particular, the idea of providing hooks, which are executed while a graph is traversed, has been borrowed from the Boost Graph Library and its API for visitors [6].

6.5 Evaluating performance of profilers

In this section, we evaluate the implementation of our approach. We present experiments that measure the performance overhead induced by memory profilers built using the proposed approach. This section aims at assessing whether our approach induces low overhead across different applications and types of analysis. Indeed, using memory profilers that have different levels of complexity, makes our evaluation closer to the expected use in real-world scenarios.

The goal of this section is answering the following research questions:

1. **RQ1. Does our approach produce profilers with lower overhead than state-of-the-art tools when used to perform many iterations of memory analysis at runtime?** To answer this question, we assess the overhead on total execution time produced by the periodic computation of a specific analysis. In this experiment, we measure and compare the overhead of our approach against the overhead produced by other solutions.

2. **RQ2. Is significant the difference between the time needed to execute a single analysis with our approach in comparison to previous solutions?** In a second experiment, we measure the execution time needed to perform a single memory analysis step instead of focusing on the total application execution time.

3. **RQ3. Does the advantage of our approach remain for real applications?** Finally, we perform memory analysis on actual applications, including *Eclipse*, *NetBean*, and others, to assess the overhead of profiling in "real-life" scenarios.

In general, these experiments show that our language produces specific profilers with lower overhead for applications running in production environments than well-known memory profilers.

6.5.1 Methodology and Setup

Our system is implemented on top of the JVMTI; thus, we compute our results using the HotSpot JVM version 1.7.0_76, with a heap size of 2GiB for all the experiments. Across this section, we use Eclipse Memory Analyzer 1.4.0 (Eclipse MAT), a production ready memory profiler, to perform several experiments. We use this tool in Command-Line Interface (CLI) mode; this executes the desired analysis in a separate process. In other words, in performing a memory analysis on a JVM instance A, we dump its heap and invoke Eclipse MAT in a separate JVM instance to collect profiling data.

[6] `http://www.boost.org/doc/libs/1_59_0/libs/graph/doc/`

We use DaCapo benchmarks version 2006-10-MR2 [Bea06] in the first two experiments, large input size in the first experiment, and default input sizes in the second one. In the third experiment, we use a set of actual applications based on OSGi, these applications are listed in the relevant section [7]. Although the details are specific to each experiment, in general, each measurement presented is the average of several runs under the same conditions.

To obtain comparable and reproducible results, we used the same hardware across all experiments: a 2.90GHz Intel(R) i7-3520M processor, running Linux with a 64 bit kernel version 3.17.3 and 8GiB of system memory.

6.5.2 Impact of Analysis on the Total Execution Time

In this experiment, we assess how much our approach affects the execution time of applications. To do so, we compare the time reported by the execution of DaCapo benchmarks without any kind of memory analysis against the execution time when our language is used to perform the analysis in Listing 6.11. In addition, we check how our approach behaves in comparison to other approaches for memory analysis. In this case, the profiler finds the number of objects, and their total size, when threads are used as only roots to traverse the graph of live objects.

The experiment was configured as follows: within a JVM instance, we wrap the execution of the DaCapo Benchmark. Each DaCapo test is configured to execute 20 warm-up iterations before the final test execution. This number of warm-ups is used to guarantee a long enough execution time. A separate thread periodically performs a *memory consumption monitoring step* every 2 seconds by using one of the methods we want to compare:

No analysis In this case, we simply execute the DaCapo Benchmarks without any additional task affecting its performance. This is the baseline for the comparison.

Handwritten JVMTI In this solution, we traverse all references in the graph of live objects starting on the threads, during this process the JVM is fully halted, impacting the total application's execution time.

Our approach We use our language to define the profiler in Listing 6.11. It is compiled and used at runtime.

Heap Dump + Eclipse MAT This method uses the approach described in Section 6.5.1; when an analysis is required, the JVM dumps the heap and executes Eclipse MAT in a separate process in CLI mode.

In this experiment, we measure the total time needed to complete the 20 warm-up iterations plus the time required to execute the final test. The idea is to check how much the performance is affected by each method. We repeat this process 10 times for each test in the DaCapo Benchmark suite, and take the average as final measurement.

[7] Links are available at `https://en.wikipedia.org/wiki/OSGi`

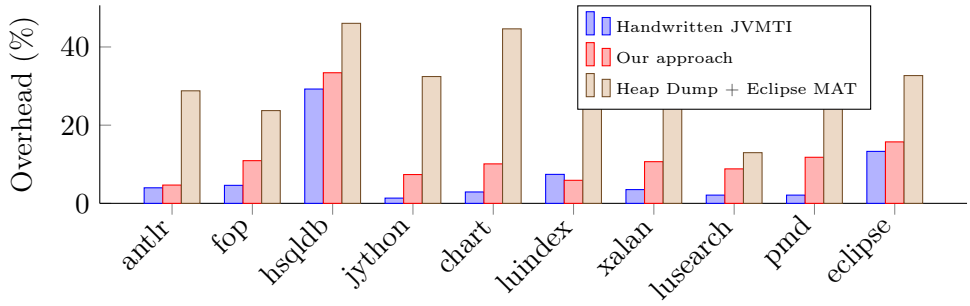


Figure 6.12 – Overhead on execution time compared to the execution without memory analysis for different tests in the DaCapo Benchmark

It is useful to discuss how varies the number of times the analysis is performed. As we mentioned, profilers run periodically in this set of experiments; thus, the number of invocations to a profiler depends on the benchmark, and the overhead produced by the profiler itself. For instance, using our approach, the memory analysis is executed a minimum of 10 times in the *fop* benchmark, and a maximum of 366 times in the *eclipse* benchmark.

Figure 6.12 depicts the overhead in total execution time for different profiling strategies and Dacapo tests. The values are shown as the percentage with respect to the baseline, which in this case is obtained when *no analysis* is executed. It is noteworthy that our approach performs close to the handwritten solution. Moreover, our solution outperforms the *Heap Dump + Eclipse MAT* approach even when the latter is executing mostly on a separate process without halting the JVM during profiling. The overhead in our approach remains between 4-33%, and it is 11.93% in average.

6.5.3 Comparing Analysis Time for an Assertion

In the previous section, we show the performance overhead on total execution time for different profiling mechanisms. However, these mechanisms are not executed under the same conditions. For instance, as we mention in Section 6.3, our implementation suspends the execution of the application while it performs the analysis. On the contrary, the *Heap Dump + Eclipse MAT* approach only suspends the application while dumping the heap, but the analysis is done in a separate process; hence, it likely runs in parallel. Therefore, in this experiment, we measure only the **analysis time**, which is the amount of elapsed time from the beginning of analysis to its end. To perform these experiments, we use again the Dacapo benchmarks. Since the analysis time depends on the number of objects visited during the computation, in this experiment, we assess the behavior of our approach using a memory profiler that must iterate over all objects to complete. For the same reason, we repeat the analysis using different input size for each benchmark; this implies that a different number of objects is found in memory.

The **assertion** used in this experiment checks **whether an instance of a specific class exists in the heap**. The following listing shows how to implement such an

assertion using our language. By defining the membership function as the *true* constant, we guarantee that all objects are visited.

```
1 create structure foreach e:#["jvm"], using
2     constructor
3         initialObjects = #Object[]
4         exists  = false
5     membership true
6     updates
7         exists  = exists or (this is UnusedClass)
```

Listing 6.17 – Detecting if there exists an instance of a specific class.

The setting of the experiment is as follows. The DaCapo benchmark suite is used with two different input sizes, default and large. Before the final test, twenty warm-ups are executed in order to ensure long enough execution time. A separate thread periodically checks the assertion and records the analysis time. The average analysis time along the complete execution of a benchmark (i.e., xalan, fop, ...) is used as data point. Ten of these data points are obtained through repetition of the previous step and used as final measurement for a pair of benchmark and analysis approach. As in the previous experiment, we use a handwritten JVMTI agents and an Eclipse MAT extension to check the assertion with those tools.

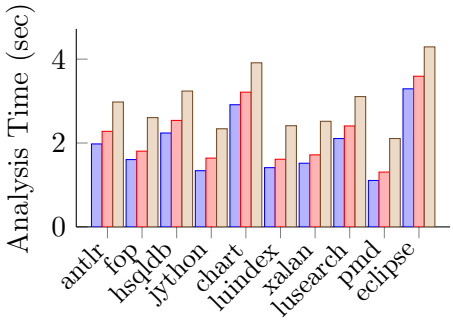


Figure 6.13 – Analysis time with default input size

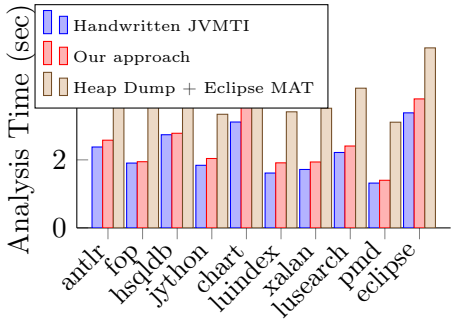


Figure 6.14 – Analysis time with large input size

Figures 6.13 and 6.14 present the results of the experiments. In both cases, default and large input size, our approach is in between the handwritten JVMTI agent and the Eclipse MAT approach. In comparison to Eclipse MAT, our approach reduces the analysis time by 25% and 39% for default and large input size respectively. As expected, the analysis time increases with the number of objects, the slowdown shown between default and large input size is of 8.42%.

6.5.4 Profiling Time in Real Scenarios

To evaluate the overhead of our approach in actual applications, we compute the memory consumption of bundles in real OSGi-based systems. Since OSGi is a widely used

framework, we chose applications built on top of OSGi or supporting it. The custom profiler definition is based on the idea that bundle consumption is the consumption of a Java classloader. Such a strategy is common when measuring memory consumption for Java-based component frameworks because modules are often isolated and represented through classloaders. The complete profiler's definition is shown below:

```
1 create structure foreach e:classloaders using
2   constructor
3     initialObjects = #[e]
4     size  = 0
5   membership ((ref_kind == root and this.class.classloader in this_structure) or
6     (ref_kind != root and referrer in this_structure))
7   updates
8     size  = size  + this.size
```

Listing 6.18 – Calculating the consumption of top components.

This experiment aims at evaluating the profiling time for each application using our approach and *Heap Dump + Eclipse MAT*. In this experiment, each application is executed, once it is initialized, the memory profiler is invoked, and its execution time measured. This process is repeated ten times for each application and analysis approach in order to use the average as final measurement. We use *Heap Dump + Eclipse MAT* to compute the memory retained for top level classloaders using a standard analysis named *top components* reports.

To execute the memory analysis from within the applications, we implemented extensions for each application (e.g., an Eclipse plugin, a NetBean module). [8].

These extensions are in charge of triggering the analysis. It was necessary because in our approach the analysis must be executed by the JVM that is being profiled. In this experiment, we perform the analysis on the following systems: Eclipse Luna [lun14], NetBeans 8.0[net15], dotCMS 3.1 [dot15], Cytoscape 3.2.1 [cyt01], Glassfish 4.1 [gla14], Liferay 6.2.2 [lif15], and WildFly 8.2 [wil13].

Figure 6.15 presents the analysis time for several applications and two analysis approaches. Our approach outperforms Eclipse MAT in all applications; the gain is 3x-19x with an average of 8x. Two factors influence the measurements. First , Eclipse MAT invests some time parsing the dump file, and creating the internal indexes to accelerate queries' response time. Second, the *top components* report in Eclipse MAT can only be implemented, using its query language, in terms of the function *retainedHeapSize*, which calculates the amount of memory retained for a given object. Since this function is costly to compute, Eclipse MAT spends a considerable amount of time on it while building the *top components* report.

[8] The evaluation code is available online: `https://github.com/intigonzalez/heapexplorer_language`

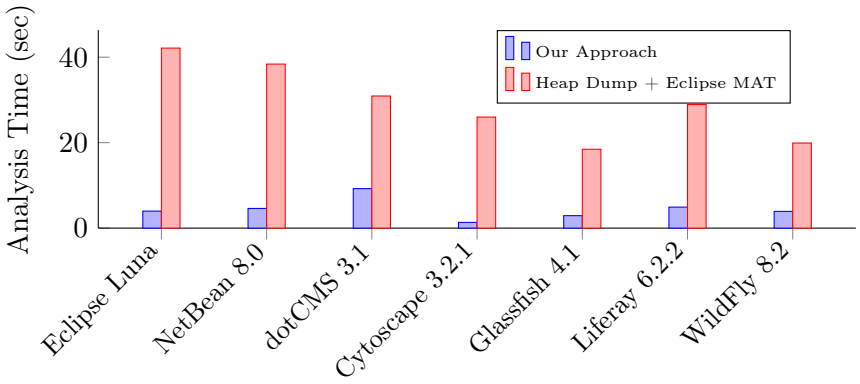


Figure 6.15 – Analysis time for real applications. It shows the time needed to compute an analysis just once. The analysis aims at finding the consumption of the top components

6.6 Conclusions

In this chapter, we propose a Domain Specific Language for expressing the mapping between abstractions and runtime data structure to collect information about the memory heap in production. This language provides an abstraction that is useful to reason about the heap and is, at the same time, easy to translate into a set of low-level routines to efficiently collect the desired information. In our opinion, this approach is a step forward in the creation of resource-aware software systems for two reasons. First, it reduces the complexity of defining customized queries; hence, developers and operators are able to use this feature to solve new problems without the need of high expertise on runtime internals. Second, such customized queries can be used in a production environment since they have a limited impact on the system's performance.

The approach proposed in this chapter contributes to answer two research questions presented in the introduction of this thesis (see Section 1.2). In particular, it answers *RQ1* (*How can we provide portable and efficient support for resource consumption monitoring?*) and *RQ4* (*How can we ease the definition and implementation of monitoring tools for new software abstractions?*) by defining a metalanguage to describe the behavior of customized memory profilers. These profilers are useful to efficiently calculate at runtime how components and other domain-specific abstractions consume resources.

Part III

Conclusion and Perspectives

Chapter 7

Conclusion and Perspectives

7.1 Conclusion

Resource-aware programming encompasses a set of techniques where applications modify their behavior based on resource availability; this is useful, for instance, if applications run under open-world conditions. Throughout this thesis, we have highlighted that a considerable runtime support is required for a software system to implement resource-aware methods. Specifically, facilities for resource accounting and reservation can be used to observe and change the consumption. Unfortunately, providing such a support is challenging in MRTEs because they often favor automatic resource management in order to ease software development.

In reviewing the state of the art, we have found shortcomings that prevent the use of existing techniques in production environments. Relatively high performance overhead in portable solutions and limited capacity to deal with arbitrary granularity levels are the most overwhelming constraints of existing approaches. The issue of handling different granularity levels is important because managed runtime environments are used to represent a wide variety of software abstractions, ranging from component models, to domain-specific languages and simple class libraries . Finally, some existing approaches show how resource accounting solutions may benefit from specializing their behavior to the monitored application.

Component-based software engineering is a good candidate to implement systems capable of coping with open-world conditions; however, the ability to handle non-functional properties is often limited because runtime environments lack support for monitoring resource consumption per components. Our first contribution is a framework, named Scapegoat, to efficiently compute per component resource utilization. In Scapegoat, we make two assumptions: there are previous monitoring mechanisms with different trade-off between overhead and accuracy, and it is possible to activate/deactivate such mechanisms at runtime. The framework follows an adaptive monitoring approach based on two principles: i) since consumption matters when the runtime environment is running out of resource, we can use optimistic lightweight monitoring and still be sure to detect potential failures on time by switching to a more precise monitoring technique; and ii) it is possible to *quickly* identify faulty components once a

potential failure is spotted. Scapegoat is capable of calculating resource usage with a lower overhead than other portable state-of-the-art approaches (overhead reduced by 92%). Moreover, since by construction Scapegoat leverages other techniques, it may benefit from new mechanisms introduced to perform monitoring as long as they can be switched on/off at runtime.

Reserving resource for specific applications is another concern in resource-aware programming. Our second contribution is a methodology to select a representation of each component in the runtime environment in such a way that resource reservations can be guaranteed with low performance overhead. We claim that the technique used to provide reservation capabilities should not be selected during the design of the component model. Instead, the resource reservation technique for each component must be chosen at deployment time, when the requirements of an application are known. In other words, we propose a methodology where both resource requirements and available technologies are decision variables to consider when we are binding components to runtime abstractions. Through this thesis, evidences for such claim are provided and a prototype, Squirrel, is implemented to show the potential benefit of this methodology.

Easing the construction of dynamic analysis tools such as resource monitoring frameworks is of utmost importance because it supports the adoption of new software abstractions. In particular, developers using component models, DSLs, and class libraries may take advantage of customized profilers. The third contribution of this thesis is a language to define customized memory profilers that can be used both during the development of applications and also in production environments. The language has been devised with constraints that, although reduce its expressive power, offer guaranties about the performance behavior of the generated profilers. To evaluate this approach, we have implemented a profiler generator that targets the JVM and uses the JVMTI to explore the content of the memory heap. Using such an implementation, we have compared generated profilers, handwritten profilers and mainstream tools. The results show that the generated profilers exhibit similar performance to the one of handwritten solutions.

7.2 Perspectives

The work presented in this thesis represents a step towards proving support for resource aware programming. This work presents many perspectives which are presented below.

Reducing overhead of instruction accounting Instrumentation by bytecode rewriting induces high performance overhead, especially when used for instruction accounting. Despite the use of adaptive monitoring which reduces the performance overhead, there are occasions when the overhead imposed is still high: while doing localized monitoring, and while probes are being activated. A way to reduce both overheads is by identifying sections of code that do not need to be instrumented.

We can learn at runtime how many instructions a method executes for a given input. This way, we only need to instrument some methods a few times until we find a predictive model (as in machine learning) that is able to predict the number of executed

instructions [TJDB06]. Afterwards, no instrumentation code is added to such methods and their consumption is measured by evaluating a prediction model when they are called.

Response to misbehavior Resource accounting only provides a single step to support the reconfiguration of a system when events about resource consumption are triggered. Handling such events in the proper way, eliminating the source of misbehavior and guaranteeing consistency of the system, is of utmost importance. There are approaches to face similar issues, for instance, replacing a service for an alternative implementation when the response time of the former is high. Since many responses are possible, it is worth considering a systematic approach to select them using a heuristic instead of a hard coded policy. This heuristic may choose among a set of reconfiguration policies that include limiting the resources available to a component, replacement of components, slowing down a component by simply delaying the access to its interface, and moving components across the distributed infrastructure

Applying the Squirrel methodology to domains with strong safety and security concerns Choosing the mapping from high-level concepts to low-level system abstractions is one of the step in providing a concrete implementation of a component model. As this mapping may affect the performance overhead of a system and its capacity to provide resource reservation, it can also affect other properties, such as security. In [GD10], the authors proposed an approach to execute components in a sandbox when it is not possible to trust in their origin; using these sandboxes also has an impact on the overall performance of the system. As a consequence, we can consider that the appropriate mechanism to put a component in a sandbox should be selected at deployment time.

A language to manipulate the graph of objects Exploring the graph of objects may be useful to reveal bugs in a system. In particular, to detect memory leaks and excessive memory consumption. It has also been discussed how to evaluate assertions on some data structures in memory by simply traversing the objects in the heap [RIS+10]. The idea is that exploring the heap may be considered a cross-cutting concern. In this context, it is interesting to study whether modifying the graph of objects may help to solve problems that can be identified by looking at objects and how they are connected.

One interesting example is eliminating memory leaks; in [ATM+15], the authors proposed a mechanism to remove stale references in dynamic OSGi applications, the approach is largely based on eliminating references between objects once the JVM detects the stale references. Since this is simply a modification to the object graph, it is worth considering the use of a language to express this kind of *automatic* bug fixing without having to hard code them using a low-level language nor modifying the JVM. The question is to what extent the same goals can be achieved without modifying the JVM by simply using profilers API such as JVMTI.

Generate the specification of memory profilers from models that describe a domain-specific abstraction Automating the construction of memory profilers for new software abstractions may ease software maintenance. Using our approach, engineers have to implement both the software abstraction and the memory profilers. A way to further reduce the development effort is by using high-level descriptions of the abstraction to generate the definition of a profiler in our language. This way, engineers could focus on the definition of the abstraction.

For instance, this is the approach followed by Xtext [EB10] to automatically generate debuggers for languages that inherit from the base language (Java). The idea is using models, as in Model-Driven Software Development (MDSD) [SVC06, Fow10], to describe the software abstraction and also the way a concrete instance is mapped to a low-level technology. In particular, this can be done using the metamodel of the abstraction (as in a DSL) and a traceability model to see how a concrete model is transformed to, for instance, Java.

Use declarative languages to define profilers Declarative languages are often preferred for solving tasks such as querying a data structure. In the case of querying a graph, there are languages able to express complex requests in concise ways, for instance, CYPHER. As we already discuss, the problem is how to efficiently schedule the execution of queries written in such languages.

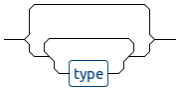
One interesting path to explore is reusing a subset of these languages that can be efficiently implemented despite the MRTEs' constraints. The main challenges in implementing an alternative like this will be to create a scheduler to optimize the execution of such queries. This is important because the graph of objects is not explicitly represented in a MRTE.

Appendix A

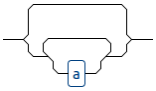
Concrete grammar of the language

Program:

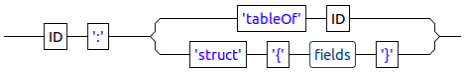
types:



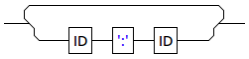
statements:



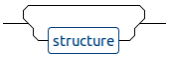
type:



fields:

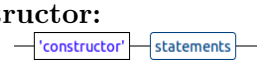


structures:



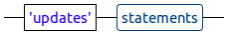
structure:

constructor:

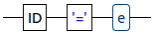


membership:

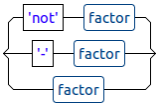
updates:



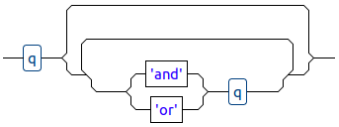
a:



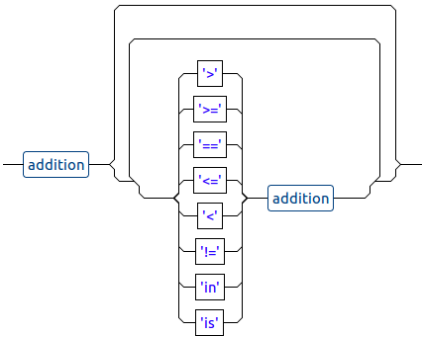
prefixed:



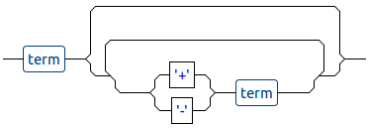
e:



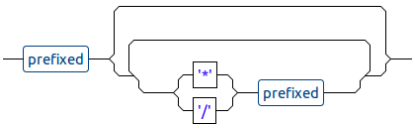
q:



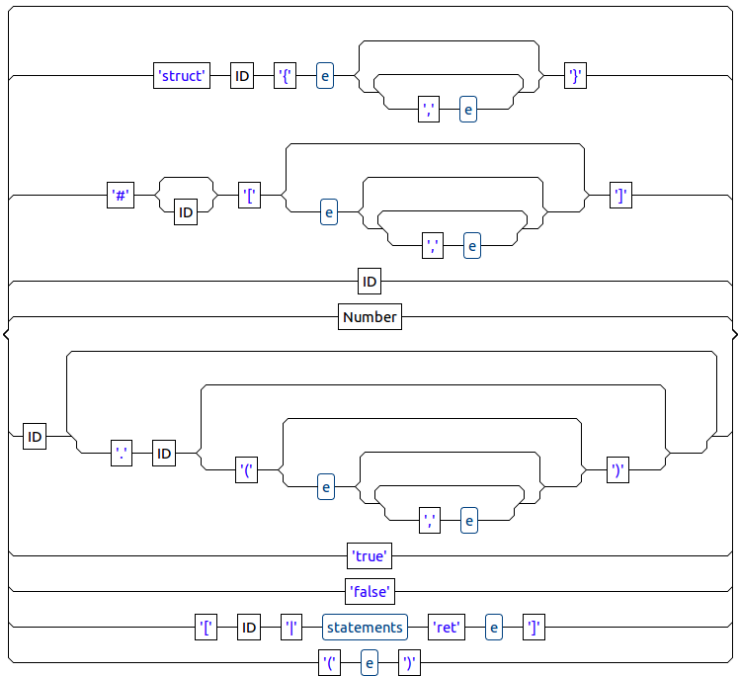
addition:



term:



factor:



Bibliography

[AAB+00] Bowen Alpern, C Richard Attanasio, John J Barton, Michael G Burke, Perry Cheng, J-D Choi, Anthony Cocchi, Stephen J Fink, David Grove, Michael Hind, et al. The jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[ABB+14] Simon Allier, Olivier Barais, Benoit Baudry, Johann Bourcier, Erwan Daubert, Franck Fleurey, Martin Monperrus, Hui Song, and Maxime Tricoire. Multi-tier diversification in internet-based software applications. *Software*, 2014.

[ABVM10] Danilo Ansaloni, Walter Binder, Alex Villazón, and Philippe Moret. Rapid development of extensible profilers for the java virtual machine with aspect-oriented programming. In *Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering*, WOSP/SIPEW '10, pages 57–62, New York, NY, USA, 2010. ACM.

[ADBI09] Marco Autili, Paolo Di Benedetto, and Paola Inverardi. Context-aware adaptive services: The plastic approach. In Marsha Chechik and Martin Wirsing, editors, *Fundamental Approaches to Software Engineering*, volume 5503 of *Lecture Notes in Computer Science*, pages 124–139. Springer Berlin Heidelberg, 2009.

[ADBI13] Marco Autili, Paolo Di Benedetto, and Paola Inverardi. A hybrid approach for resource-based comparison of adaptable java applications. *Science of Computer Programming*, 78 Issue 78:987–1009, August 2013.

[AEPQn09] Anthony Allevato, Stephen H. Edwards, and Manuel A. Pérez-Quiñones. Dereferee: Exploring pointer mismanagement in student code. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, SIGCSE '09, pages 173–177, New York, NY, USA, 2009. ACM.

[AH01] Luca Alfaro and ThomasA. Henzinger. Interface theories for component-based design. In ThomasA. Henzinger and ChristophM. Kirsch, editors, *Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*, pages 148–165. Springer Berlin Heidelberg, 2001.

[ALG10] Majed Alhaisoni, Antonio Liotta, and Mohammed Ghanbari. Resource-awareness and trade-off optimisation in p2p video streaming. *Int. J. Adv. Media Commun.*, 4(1):59–77, December 2010.

[AO10] Michael Achenbach and Klaus Ostermann. A meta-aspect protocol for developing dynamic analyses. In *Proceedings of the First International Conference on Runtime Verification*, RV'10, pages 153–167, Berlin, Heidelberg, 2010. Springer-Verlag.

[AR01] Matthew Arnold and Barbara G. Ryder. A Framework for Reducing the Cost of Instrumented Code. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 168–179, 2001.

[ASM+05] Andrew Ayers, Richard Schooler, Chris Metcalf, Anant Agarwal, Junghwan Rhee, and Emmett Witchel. Traceback: First fault diagnosis by reconstruction of distributed control flow. *SIGPLAN Not.*, 40(6):201–212, June 2005.

[ATBM14] Koutheir Attouchi, Gaël Thomas, André Bottaro, and Gilles Muller. Memory monitoring in a multi-tenant osgi execution environment. In *Proceedings of the 17th International ACM Sigsoft Symposium on Component-based Software Engineering*, CBSE '14, pages 107–116, New York, NY, USA, 2014. ACM.

[ATM+15] Koutheir Attouchi, Gaël Thomas, Gilles Muller, Julia Lawall, and André Bottaro. Incinerator - eliminating stale references in dynamic osgi applications. In *Proceedings of the international conference on Dependable Systems and Networks, DSN '15*, page 11. IEEE Computer Society, 2015.

[Avi85] A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, (12):1491–1501, 1985.

[BAM14] Benoit Baudry, Simon Allier, and Martin Monperrus. Tailored source code transformations to synthesize computationally diverse program variants. In *International Symposium on Software Testing and Analysis*, 2014.

[BBDS97] R. Black, P. Barham, A. Donnelly, and N. Stratford. Protocol implementation in a vertically structured operating system. In *Local Computer Networks, 1997. Proceedings., 22nd Annual Conference on*, pages 179–188, Nov 1997.

[BBH+12] Lubomir Bulej, Tomas Bures, Vojtech Horky, Jaroslav Keznikl, and Petr Tuma. Performance awareness in component systems: Vision paper. In *Proceedings of the 2012 IEEE 36th Annual Computer Software and Applications Conference Workshops*, COMPSACW '12, pages 514–519, Washington, DC, USA, 2012. IEEE Computer Society.

[BCL+06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, September 2006.

[BCP08] Chiara Boldrini, Marco Conti, and Andrea Passarella. Context and resource awareness in opportunistic network data dissemination. In *Proceedings of the 2008 International Symposium on a World of Wireless, Mobile and Multimedia Networks*, WOWMOM '08, pages 1–6, Washington, DC, USA, 2008. IEEE Computer Society.

[BCR03] David F. Bacon, Perry Cheng, and V.T. Rajan. The metronome: A simpler approach to garbage collection in real-time systems. In *In Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES), OTM Workshops*, pages 466–478, 2003.

[BDM99] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: a new facility for resource management in server systems. In *Proceedings of the third symposium on Operating systems design and implementation*, OSDI '99, pages 45–58, Berkeley, CA, USA, 1999. USENIX Association.

[BDNG06] Luciano Baresi, Elisabetta Di Nitto, and Carlo Ghezzi. Toward open-world software: Issue and challenges. *Computer*, 39(10):36–43, October 2006.

[Bea06] Stephen M. Blackburn and et al. The dacapo benchmarks: Java benchmarking development and analysis. In *Proc. of the 21st Annual Conf. on OO Prog. Systems, Lang., and Appl.*, OOPSLA '06, pages 169–190, NY, USA, 2006. ACM.

[Bec10] Steffen Becker. The palladio component model. In *Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering*, WOSP/SIPEW '10, pages 257–258, New York, NY, USA, 2010. ACM.

[BEK+06] Enrico Biermann, Karsten Ehrig, Christian Köhler, Günter Kuhns, Gabriele Taentzer, and Eduard Weiss. Graphical definition of in-place transformations in the eclipse modeling framework. In *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems*, MoDELS'06, pages 425–439, Berlin, Heidelberg, 2006. Springer-Verlag.

[BH05a] Godmar Back and Wilson C. Hsieh. The kaffeos java runtime system. *ACM Trans. Program. Lang. Syst.*, 27(4):583–630, July 2005.

[BH05b] Walter Binder and Jarle Hulaas. Extending standard java runtime systems for resource management. In *Proceedings of the 4th international conference on Software Engineering and Middleware*, SEM'04, pages 154–169, Berlin, Heidelberg, 2005. Springer-Verlag.

[BH06a] Walter Binder and Jarle Hulaas. Exact and portable profiling for the {JVM} using bytecode instruction counting. *Electronic Notes in Theoretical Computer Science*, 164(3):45–64, 2006. Proceedings of the 4th International Workshop on Quantitative Aspects of Programming Languages (QAPL 2006).

[BH06b] Walter Binder and Jarle Hulaas. Flexible and efficient measurement of dynamic bytecode metrics. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, GPCE '06, pages 171–180, New York, NY, USA, 2006. ACM.

[BH06c] Walter Binder and Jarle Hulaas. Using bytecode instruction counting as portable {CPU} consumption metric. *Electronic Notes in Theoretical Computer Science*, 153(2):57 – 77, 2006. Proceedings of the Third Workshop on Quantitative Aspects of Programming Languages (QAPL 2005)Quantitative Aspects of Programming Languages 2005.

[BHL00] Godmar Back, Wilson C. Hsieh, and Jay Lepreau. Processes in KaffeOS: isolation, resource management, and sharing in java. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 23–23, Berkeley, CA, USA, 2000. USENIX Association.

[BHMV09] Walter Binder, Jarle Hulaas, Philippe Moret, and Alex Villazón. Platform-independent profiling in a virtual execution environment. *Softw. Pract. Exper.*, 39(1):47–79, January 2009.

[BHP06] T. Bures, P. Hnetynka, and F. Plasil. Sofa 2.0: Balancing advanced features in a hierarchical component model. In *Software Engineering Research, Management and Applications, 2006. Fourth International Conference on*, pages 40–48, Aug 2006.

[BHV01] Walter Binder, Jane G. Hulaas, and Alex Villazon. Portable resource control in java. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, pages 139–155, New York, NY, USA, 2001. ACM.

[Bin05] Walter Binder. A portable and customizable profiling framework for java based on bytecode instruction counting. In Kwangkeun Yi, editor, *Programming Languages and Systems*, volume 3780 of *Lecture Notes in Computer Science*, pages 178–194. Springer Berlin Heidelberg, 2005.

[Bin06] Walter Binder. Portable and accurate sampling profiling for java. *Softw. Pract. Exper.*, 36(6):615–650, May 2006.

[BJPW99] A. Beugnard, J.-M. Jezequel, N. Plouzeau, and D. Watkins. Making components contract aware. *Computer*, 32(7):38–45, July 1999.

[BLC02] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.

[BMSG+09] Yuriy Brun, Giovanna Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè, and Mary Shaw. Software engineering for self-adaptive systems. chapter Engineering Self-Adaptive Systems Through Feedback Loops, pages 48–70. Springer-Verlag, Berlin, Heidelberg, 2009.

[BPK+14] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. Ix: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 49–65, Berkeley, CA, USA, 2014. USENIX Association.

[BSD+08] Stephen M. Blackburn, Sergey I. Salishev, Mikhail Danilov, Oleg A. Mokhovikov, Anton A. Nashatyrev, and Peter A. Novodvorsky. The moxie jvm experience. tech. rep. tr-cs-08-01. Technical report, Department of Computer Science, The Australian National University, 2008.

[CD01] Grzegorz Czajkowski and Laurent Daynés. Multitasking without compromise: a virtual machine evolution. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, pages 125–138, New York, NY, USA, 2001. ACM.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: MMethod. Tools and Applications.* Addison-Wesley, 2000.

[CEG+05] Michal Cierniak, Marsha Eng, Neal Glew, Brian T. Lewis, and James M. Stichnoth. The Open Runtime Platform: a flexible high-performance managed runtime environment. *Concurrency and Computation: Practice and Experience*, 17:617–637, 2005.

[CFG10] Mauro Caporuscio, Marco Funaro, and Carlo Ghezzi. Architectural issues of adaptive pervasive systems. In *Graph Transformations and Model-Driven Engineering - Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday*, pages 492–511. 2010.

[CHP06] Jan Carlson, John Håkansson, and Paul Pettersson. Saveccm: An analysable component model for real-time systems. *Electronic Notes*

in Theoretical Computer Science, 160:127 – 140, 2006. Proceedings of the International Workshop on Formal Aspects of Component Software (FACS 2005)Proceedings of the International Workshop on Formal Aspects of Component Software (FACS 2005).

[CLCM00] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. Multijava: Modular open classes and symmetric multiple dispatch for java. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 130–145, New York, NY, USA, 2000. ACM.

[CLW13] James Cheney, Sam Lindley, and Philip Wadler. A practical theory of language-integrated query. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 403–416, New York, NY, USA, 2013. ACM.

[CSVC11] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M.R.V. Chaudron. A classification framework for software component models. *Software Engineering, IEEE Transactions on*, 37(5):593–615, Sept 2011.

[CvE98] Grzegorz Czajkowski and Thorsten von Eicken. JRes: a resource accounting interface for java. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '98, pages 21–35, New York, NY, USA, 1998. ACM.

[cyt01] The cytoscape website, 2001.

[DA02] Patrick Doyle and Tarek S. Abdelrahman. A modular and extensible jvm infrastructure. In *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium (JVM '02)*, 2002.

[DB00] Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: less is more. *ACM SIGARCH Computer Architecture News*, 28(5):202–211, 2000.

[DDHV03] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. Dynamic metrics for java. *SIGPLAN Not.*, 38(11):149–168, October 2003.

[DEM02] Frédéric Duclos, Jacky Estublier, and Philippe Morat. Describing and using non functional aspects in component based applications. In *Proceedings of the 1st International Conference on Aspect-oriented Software Development*, AOSD '02, pages 65–75, New York, NY, USA, 2002. ACM.

[DHPW01] Charles Daly, Jane Horgan, James Power, and John Waldron. Platform independent dynamic java virtual machine analysis: The java grande forum benchmark suite. In *Proceedings of the ACM 2001 Java Grande/ISCOPE Conference; Palo Alto, CA; United States; 2 June 2001 through 4 June 2001; Code 60419*, pages 106–115, 2001.

[Dmi04] Mikhail Dmitriev. Profiling java applications using code hotswapping and dynamic call graph revelation. *SIGSOFT Softw. Eng. Notes*, 29(1):139–150, January 2004.

[DO09] Iulian Dragos and Martin Odersky. Compiling generics through user-directed type specialization. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 42–47, New York, NY, USA, 2009. ACM.

[dot15] The dotcms website, 2015.

[DvdHT02] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. An infrastructure for the rapid development of XML-based architecture description languages. In *24th International Conference on Software Engineering*, ICSE '02, pages 266–276, New York, NY, USA, 2002. ACM.

[DZ02] Benoît Dageville and Mohamed Zait. SQL memory management in oracle9i. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, pages 962–973. VLDB Endowment, 2002.

[EB10] Moritz Eysholdt and Heiko Behrens. Xtext: Implement your language faster than the quick and dirty way. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '10, pages 307–309, New York, NY, USA, 2010. ACM.

[EGLGJ07] Mostafa Elhemali, César A. Galindo-Legaria, Torsten Grabs, and Milind M. Joshi. Execution strategies for SQL subqueries. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 993–1004, New York, NY, USA, 2007. ACM.

[EK+95] Dawson R Engler, M Frans Kaashoek, et al. *Exokernel: An operating system architecture for application-level resource management*, volume 29. ACM, 1995.

[EMF13] Eclipse modeling framework, 2013.

[EvdSV+13] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, WilliamR. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, GabriëlD.P. Konat, PedroJ. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, VladA. Vergu, Eelco Visser, Kevin van der Vlist, GuidoH. Wachsmuth, and Jimi van der Woning. The state of the art in language workbenches. In Martin Erwig, RichardF. Paige, and Eric Van Wyk, editors, *Software Language Engineering*, volume 8225 of *Lecture Notes in Computer Science*, pages 197–217. Springer International Publishing, 2013.

[Fai98] Rickard Edward Faith. *Debugging Programs After Structure-Changing Transformation*. PhD thesis, Department of Computer Science, University of North Carolina at Chapel Hill, 1998.

[FMF+12] Francois Fouquet, Brice Morin, Franck Fleurey, Olivier Barais, Noel Plouzeau, and Jean-Marc Jezequel. A dynamic component model for cyber physical systems. In *Proc. of the 15th Symp. on Component Based Soft. Eng.*, CBSE '12, pages 135–144, New York, NY, 2012.

[FND+14] François Fouquet, Grégory Nain, Erwan Daubert, Johann Bourcier, Olivier Barais, Noel Plouzeau, and Brice Morin. Designing and evolving distributed architecture using kevoree. In *Proceedings of the 10th International ACM Sigsoft Conference on Quality of Software Architectures*, QoSA '14, pages 147–148, New York, NY, USA, 2014. ACM.

[Fon04] Philip W. L. Fong. Pluggable verification modules: An extensible protection mechanism for the jvm. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 404–418, New York, NY, USA, 2004. ACM.

[Fow10] Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010.

[FS04] Stéphane Frénot and Dan Stefan. Open-service-platform instrumentation: Jmx management over osgi. In *Proceedings of the 1st French-speaking conference on Mobility and ubiquity computing*, UbiMob '04, pages 199–202, New York, NY, USA, 2004. ACM.

[FSA97] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS-VI)*, HOTOS '97, pages 67–, Washington, DC, USA, 1997. IEEE Computer Society.

[GD10] Kiev Gama and Didier Donsez. A self-healing component sandbox for untrustworthy third party code execution. In *Proceedings of the 13th international conference on Component-Based Software Engineering*, CBSE'10, pages 130–149, Berlin, Heidelberg, 2010. Springer-Verlag.

[GHBD+14] Inti Gonzalez-Herrera, Johann Bourcier, Erwan Daubert, Walter Rudametkin, Olivier Barais, François Fouquet, and Jean-Marc Jézéquel. Scapegoat: an Adaptive monitoring framework for Component-based systems. In *Proc. of WICSA*, Australie, 2014. IEEE/IFIP.

[GKS+04] Nikola Grcevski, Allan Kielstra, Kevin Stoodley, Mark Stoodley, and Vijay Sundaresan. Javatm just-in-time compiler and virtual machine improvements for server and middleware applications. In *Proceedings of*

the 3rd Conference on Virtual Machine Research And Technology Symposium - Volume 3, VM'04, pages 12–12, Berkeley, CA, USA, 2004. USENIX Association.

[gla14] The glassfish website, 2014.

[GM11] Brendan Gregg and Jim Mauro. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1st edition, 2011.

[GMML12] Vincenzo Grassi, Raffaela Mirandola, Nenad Medvidovic, and Magnus Larsson, editors. *Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering, CBSE 2012, part of Comparch '12 Federated Events on Component-Based Software Engineering and Software Architecture, Bertinoro, Italy, June 25-28, 2012*. ACM, 2012.

[GMPLMT10] Carlo Ghezzi, Alfredo Motta, Valerio Panzica La Manna, and Giordano Tamburrelli. Qos driven dynamic binding in-the-many. In *Proceedings of the 6th International Conference on Quality of Software Architectures: Research into Practice - Reality and Gaps*, QoSA'10, pages 68–83, Berlin, Heidelberg, 2010. Springer-Verlag.

[GMS02] Dominik Gruntz, Stephan Murer, and C Szyperski. Component software: Beyond object-oriented programming. *Massachusetts: Addison-Wesley*, 2002.

[Gro13] EJB 3.2 Expert Group. Jsr 345. enterprise javabeanstm,version 3.2 ejb core contracts andrequirements version 3.2, May 2013.

[GTL+10] Nicolas Geoffray, Gaël Thomas, Julia Lawall, Gilles Muller, and Bertil Folliot. Vmkit: a substrate for managed runtime environments. *SIGPLAN Not.*, 45(7):51–62, March 2010.

[GTM+09] Nicolas Geoffray, Gaël Thomas, Gilles Muller, Pierre Parrend, Stéphane Frénot, and Bertil Folliot. I-JVM: a Java virtual machine for component isolation in OSGi. In *Proceedings of the international conference on Dependable Systems and Networks, DSN '09*, pages 544–553, Estoril, Portugal, 2009. IEEE Computer Society.

[HB04] Jarle Hulaas and Walter Binder. Program transformations for portable cpu accounting and control in java. In *Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '04, pages 169–177, New York, NY, USA, 2004. ACM.

[HB08] Jarle Hulaas and Walter Binder. Program transformations for light-weight CPU accounting and control in the jvm. *Higher Order Symbol. Comput.*, 21(1-2):119–146, June 2008.

[HC01] George T. Heineman and William T. Councill. *Component-based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.

[hKW00] Poul henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *In Proc. 2nd Intl. SANE Conference*, 2000.

[HO10] Christian Hofer and Klaus Ostermann. Modular domain-specific language components in scala. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, pages 83–92, New York, NY, USA, 2010. ACM.

[Hor01] Paul Horn. Autonomic computing: Ibm's perspective on the state of information technology, 2001.

[HPM+05] P.R. Henriques, M.J.V. Pereira, M. Mernik, M. Lenic, J. Gray, and H. Wu. Automatic generation of language-based tools using the lisa system. *Software, IEE Proceedings* -, 152(2):54–69, April 2005.

[HWRK11] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical assessment of mde in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 471–480. ACM, 2011.

[HZS08] Shan Shan Huang, David Zook, and Yannis Smaragdakis. Domain-specific languages and program generation with meta-aspectj. *ACM TRANSACTIONS ON SOFTWARE ENGINEERING AND METHODOLOGY*, (18), November 2008.

[IFMW08] Florian Irmert, Thomas Fischer, and Klaus Meyer-Wegener. Runtime adaptation in a service-oriented component model. In *Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-managing Systems*, SEAMS '08, pages 97–104, New York, NY, USA, 2008. ACM.

[IHWN12] Hiroshi Inoue, Hiroshige Hayashizaki, Peng Wu, and Toshio Nakatani. Adaptive multi-level compilation in a trace-based java jit compiler. *SIGPLAN Not.*, 47(10):179–194, October 2012.

[IN05] H. Ishikawa and T. Nakajima. Earlgray: a component-based java virtual machine for embedded systems. In *Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on*, pages 403–409, May 2005.

[JAH11] Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. Catch me if you can: Performance bug detection in the wild. *SIGPLAN Not.*, 46(10):155–170, October 2011.

[Jav99] Javassist, 1999.

[JBD15] Taylor T. Johnson, Stanley Bak, and Steven Drager. Cyber-physical specification mismatch identification with dynamic analysis. In *Proceedings of the ACM/IEEE Sixth International Conference on Cyber-Physical Systems*, ICCPS '15, pages 208–217, New York, NY, USA, 2015. ACM.

[JetPS] JetBrains, Meta Programming System (MPS).

[JW97] Ralph Johnson and Bobby Woolf. The Type Object Pattern, 1997.

[KABM12] Stephen Kell, Danilo Ansaloni, Walter Binder, and Lukáš Marek. The jvm is not observable enough (and what to do about it). In *Proceedings of the Sixth ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '12, pages 33–38, New York, NY, USA, 2012. ACM.

[KC03] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.

[Kel13] Ryan D. Kelker. *Clojure for Domain-specific Languages*. Packt Publishing, December 2013.

[KHW03] Heather Kreger, Ward Harold, and Leigh Williamson. *Java and JMX: Building Manageable Systems*. Addison-Wesley, Boston, MA, 2003.

[KLC+14] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. Osv: Optimizing the operating system for virtual machines. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 61–72, Berkeley, CA, USA, 2014. USENIX Association.

[Kni02] John C. Knight. Safety critical systems: challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 547–550, New York, NY, USA, 2002. ACM.

[KSAK14] Rouven Krebs, Simon Spinner, Nadia Ahmed, and Samuel Kounev. Resource Usage Control In Multi-Tenant Applications. In *Proceedings of the 14th IEEE/ACM Int. Symposium on Cluster, Cloud and Grid Computing (CCGrid 2014)*. IEEE/ACM, May 2014.

[Kul07] Eugene Kuleshov. Using the asm framework to implement common java bytecode transformation patterns. In *AOSD.07*, March 2007.

[KVH12] Kostas Kolomvatsos, George Valkanas, and Stathes Hadjiefthymiades. Debugging applications created by a domain specific language: The ipac case. *J. Syst. Softw.*, 85(4):932–943, April 2012.

[KWK13] Rouven Krebs, Alexander Wert, and Samuel Kounev. Multi-Tenancy Performance Benchmark for Web Application Platforms. In *Proceedings*

of the 13th Int. Conference on Web Engineering (ICWE 2013). Springer-Verlag, July 2013.

[KYK+14] Yushi Kuroda, Ikuo Yamasaki, Shigekuni Kondo, Yukihisa Katayama, and Osamu Mizuno. A memory isolation method for osgi-based home gateways. In *Proceedings of the 17th International ACM Sigsoft Symposium on Component-based Software Engineering*, CBSE '14, pages 117–122, New York, NY, USA, 2014. ACM.

[LBM15] Philipp Lengauer, Verena Bitto, and Hanspeter Mössenböck. Accurate and efficient object tracing for java applications. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ICPE '15, pages 51–62, New York, NY, USA, 2015. ACM.

[lif15] The liferay website, 2015.

[LKV11] Ricky T. Lindeman, Lennart C.L. Kats, and Eelco Visser. Declaratively defining domain-specific language debuggers. In *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering*, GPCE '11, pages 127–136, New York, NY, USA, 2011. ACM.

[LLC10] Marc Léger, Thomas Ledoux, and Thierry Coupaye. Reliable dynamic reconfigurations in a reflective component model. *Component-Based Software Engineering*, pages 74–92, 2010.

[LP08] Jonathan M. Lambert and James F. Power. Platform independent timing of java virtual machine bytecode instructions. *Electronic Notes in Theoretical Computer Science*, 220(3):97 – 113, 2008. Proceedings of the Sixth Workshop on Quantitative Aspects of Programming Languages (QAPL 2008).

[lun14] The luna eclipse website, 2014.

[LV99] Sheng Liang and Deepa Viswanathan. Comprehensive profiling support in the java virtual machine. In *5th USENIX Conference on Object Oriented Technologies*, 1999.

[MAB+10] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[MBEDB06] Jonas Maebe, Dries Buytaert, Lieven Eeckhout, and Koen De Bosschere. Javana: A system for building customized java program analysis tools. *SIGPLAN Not.*, 41(10):153–168, October 2006.

[MBKA12] Yoann Maurel, André Bottaro, Radu Kopetz, and Koutheir Attouchi. Adaptive monitoring of end-user osgi-based home boxes. In *Proceedings of the 15th Symposium on Component Based Software Engineering*, CBSE '12, pages 157–166, USA, 2012. ACM.

[MBNJ09a] Brice Morin, Olivier Barais, Gregory Nain, and Jean-Marc Jezequel. Taming dynamically adaptive systems using models and aspects. In *Proceedings of the 31st Int. Conference on Software Engineering*, ICSE '09, pages 122–132, USA, 2009.

[MBNJ09b] Brice Morin, Olivier Barais, Gregory Nain, and Jean-Marc Jezequel. Taming Dynamically Adaptive Systems with Models and Aspects. In *ICSE?09: 31st International Conference on Software Engineering*, Vancouver, Canada, May 2009.

[MBT11] Philippe Moret, Walter Binder, and Éric Tanter. Polymorphic bytecode instrumentation. In *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*, AOSD '11, pages 129–140, New York, NY, USA, 2011. ACM.

[Mer10] Bernhard Merkle. Textual modeling tools: Overview and comparison of language workbenches. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '10, pages 139–148, New York, NY, USA, 2010. ACM.

[Mey92] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[Mey07] Marcus Meyerhöfer. Testejb: Response time measurement and call dependency tracing for ejbs. In *Proceedings of the 1st Workshop on Middleware-application Interaction: In Conjunction with Euro-Sys 2007*, MAI '07, pages 55–60, New York, NY, USA, 2007. ACM.

[ML05] Marcus Meyerhöfer and Frank Lauterwald. Towards platform-independent component measurement. In *in Tenth International Workshop on Component-Oriented Programming*, 2005.

[MLAZ02] Marjan Mernik, Mitja Lenic, Enis Avdicausevic, and Viljem Zumer. Lisa: An interactive environment for programming language development. In R.Nigel Horspool, editor, *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 1–4. Springer Berlin Heidelberg, 2002.

[MLD05] Gilles Muller, Julia L. Lawall, and Hervé Duchesne. A framework for simplifying the development of kernel schedulers: Design and performance evaluation. In *HASE 2005 - High Assurance Systems Engineering Conference*, pages 56–65, Heidelberg, Germany, October 2005.

[MLS+15] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, Jon Crowcroft, and Ian Leslie. Jitsu: Just-in-time summoning of unikernels. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 559–573, Berkeley, CA, USA, 2015. USENIX Association.

[MMBC97] Gilles Muller, Bárbara Moura, Fabrice Bellard, and Charles Consel. Harissa: A flexible and efficient java environment mixing bytecode and compiled code. In *Proceedings of the 3rd Conference on USENIX Conference on Object-Oriented Technologies (COOTS) - Volume 3*, COOTS'97, pages 1–1, Berkeley, CA, USA, 1997. USENIX Association.

[MMR+13] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. *SIGPLAN Not.*, 48(4):461–472, March 2013.

[MPH08] T. Miettinen, D. Pakkala, and M. Hongisto. A method for the resource monitoring of osgi-based software components. In *Software Engineering and Advanced Applications, 2008. SEAA '08. 34th Euromicro Conference*, pages 100–107, Sept 2008.

[MR96] J.I. Munro and V. Raman. Fast stable in-place sorting witho(n) data moves. *Algorithmica*, 16(2):151–160, 1996.

[MT00] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. Softw. Eng.*, 26:70–93, January 2000.

[MV05] Marcus Meyerhöfer and Bernhard Volz. EJBMemProf — A Memory Profiling Framework for Enterprise JavaBeans. In *Proceedings of the Eighth International Symposium on Component-Based Software Engineering*, volume 3489 of *Lecture Notes in Computer Science*, pages 17–32. Springer International Publishing, 2005.

[MV11] Raphael Mannadiar and Hans Vangheluwe. Debugging in domain-specific modelling. In *Proceedings of the Third International Conference on Software Language Engineering*, SLE'10, pages 276–285, Berlin, Heidelberg, 2011. Springer-Verlag.

[MWH14] Ilias Marinos, Robert N.M. Watson, and Mark Handley. Network stack specialization for performance. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 175–186, New York, NY, USA, 2014. ACM.

[MZA+12a] Lukáš Marek, Yudi Zheng, Danilo Ansaloni, Walter Binder, Zhengwei Qi, and Petr Tuma. Disl: An extensible language for efficient and comprehensive dynamic program analysis. In *Proceedings of the Seventh Workshop on Domain-Specific Aspect Languages*, DSAL '12, pages 27–28, New York, NY, USA, 2012. ACM.

[MZA+12b] Lukáš Marek, Yudi Zheng, Danilo Ansaloni, Aibek Sarimbekov, Walter Binder, Petr Tůma, and Zhengwei Qi. Java bytecode instrumentation made easy: The disl framework for dynamic program analysis. In Ranjit Jhala and Atsushi Igarashi, editors, *Programming Languages and Systems*, volume 7705 of *Lecture Notes in Computer Science*, pages 256–263. Springer Berlin Heidelberg, 2012.

[NBZ08] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Exterminator: Automatically correcting memory errors with high probability. *Commun. ACM*, 51(12):87–95, December 2008.

[net15] The netbeans website, 2015.

[NGM+08] Elisabetta Di Nitto, Carlo Ghezzi, Andreas Metzger, Mike P. Papazoglou, and Klaus Pohl. A journey to highly dynamic, self-adaptive service-based applications. *Autom. Softw. Eng.*, 15(3-4):313–341, 2008.

[OQL14] VisualVM. `http://visualvm.java.net/`, 2008–2014.

[OSG14] OSGi. Osgi specification, release 6, June 2014.

[OYM15] Hyeong-Seok Oh, Ji Hwan Yeo, and Soo-Mook Moon. Bytecode-to-c ahead-of-time compilation for android dalvik virtual machine. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, pages 1048–1053, San Jose, CA, USA, 2015. EDA Consortium.

[PAG09] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of sparql. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, September 2009.

[PBBP11] Achille Peternier, Daniele Bonetta, Walter Binder, and Cesare Pautasso. Overseer: low-level hardware monitoring and management for java. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java, PPPJ 2011, Kongens Lyngby, Denmark, August 24-26, 2011*, pages 143–146, 2011.

[PCC+11] Jordà Polo, Claris Castillo, David Carrera, Yolanda Becerra, Ian Whalley, Malgorzata Steinder, Jordi Torres, and Eduard Ayguadé. Resource-aware adaptive scheduling for mapreduce clusters. In *Proceedings of the 12th International Middleware Conference*, Middleware '11, pages

180–199, Laxenburg, Austria, Austria, 2011. International Federation for Information Processing.

[PEBN07] Arjan Peddemors, Henk Eertink, Mortaza Bargh, and Ignas Niemegeers. Network resource awareness and control in mobile applications. *IEEE Internet Computing*, 11(2):34–43, March 2007.

[PF11] Terence Parr and Kathleen Fisher. Ll(*): The foundation of the antlr parser generator. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 425–436, New York, NY, USA, 2011. ACM.

[PHS10] Wolfgang Puffitsch, Benedikt Huber, and Martin Schoeberl. Worst-case analysis of heap allocations. In *Proceedings of the 4th international conference on Leveraging applications of formal methods, verification, and validation - Volume Part II*, ISoLA'10, pages 464–478, Berlin, Heidelberg, 2010. Springer-Verlag.

[PHZ12] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 29–42, New York, NY, USA, 2012. ACM.

[PL90] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer-Verlag, 1990.

[PLM12] Valerio Panzica La Manna. Local dynamic update for component-based distributed systems. In *Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering*, CBSE '12, pages 167–176, New York, NY, USA, 2012. ACM.

[Por14] Barry Porter. Runtime modularity in complex structures: A component model for fine grained runtime adaptation. In *Proceedings of the 17th International ACM Sigsoft Symposium on Component-based Software Engineering*, CBSE '14, pages 29–34, New York, NY, USA, 2014. ACM.

[PP07] Fabio Pianese and Diego Perino. Resource and locality awareness in an incentive-based p2p live streaming system. In *Proceedings of the 2007 Workshop on Peer-to-peer Streaming and IP-TV*, P2P-TV '07, pages 317–322, New York, NY, USA, 2007. ACM.

[PPMB10] Diego Perez-Palacin, José Merseguer, and Simona Bernardi. Performance aware open-world software in a 3-layer architecture. In *Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering*, WOSP/SIPEW '10, pages 49–56, New York, NY, USA, 2010. ACM.

[PRW03] David W. Price, Algis Rudys, and Dan S. Wallach. Garbage collector memory accounting in language-based systems. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, SP '03, pages 263–, Washington, DC, USA, 2003. IEEE Computer Society.

[PTB+97] Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, and Scott A. Watterson. Toba: Java for applications a way ahead of time (wat) compiler. In *Proceedings of the 3rd Conference on USENIX Conference on Object-Oriented Technologies (COOTS) - Volume 3*, COOTS'97, pages 3–3, Berkeley, CA, USA, 1997. USENIX Association.

[PVC01] Michael Paleczny, Christopher Vick, and Cliff Click. The java hotspottm server compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, pages 1–1, Berkeley, CA, USA, 2001. USENIX Association.

[PWBK07] David J. Pearce, Matthew Webster, Robert Berry, and Paul H. J. Kelly. Profiling with aspectj. *Softw. Pract. Exper.*, 37(7):747–777, June 2007.

[Rah10] Ayende Rahien. *DSLs in Boo: Domain Specific Languages in .NET*. Manning Publications Co., 2010.

[Rei08] Steven P. Reiss. Controlled dynamic performance analysis. In *Proceedings of the 7th International Workshop on Software and Performance*, WOSP '08, pages 43–54, New York, NY, USA, 2008. ACM.

[RHM12] Jones Richard, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. CRC Press, 2012.

[RIS+10] Christoph Reichenbach, Neil Immerman, Yannis Smaragdakis, Edward E. Aftandilian, and Samuel Z. Guyer. What can the gc compute efficiently?: A language for heap assertions at gc time. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 256–269, New York, NY, USA, 2010. ACM.

[RWDD09] T. Richardson, A. J. Wellings, J. A. Dianes, and M. Díaz. Providing temporal isolation in the osgi framework. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '09, pages 1–10, New York, NY, USA, 2009. ACM.

[SG02] Nicolas Le Sommer and Frédéric Guidec. JAMUS: Java Accommodation of Mobile Untrusted Software. 2002.

[SGM02] CLEMENS SZYPERSKI, DOMINIK GRUNTZ, and STEPHAN MURER. *Component Software: Beyond Object-Oriented programming*. Addison-Wesley, second edition edition, 2002.

[SMB+11] Aibek Sarimbekov, Philippe Moret, Walter Binder, Andreas Sewe, and Mira Mezini. Complete and platform-independent calling context profiling for the java virtual machine. *Electronic Notes in Theoretical Computer Science*, 279(1):61 – 74, 2011. Proceedings of the Bytecode 2011 workshop, the Sixth Workshop on Bytecode Semantics, Verification, Analysis and Transformation.

[SMG96] Quinn O. Snell, Armin R. Mikler, and John L. Gustafson. Netpipe: A network protocol independent performance evaluator. In *IASTED Int. Conference on Intelligent Information Management and Systems*, 1996.

[SML10] Michael Schmidt, Michael Meier, and Georg Lausen. Foundations of sparql query optimization. In *Proceedings of the 13th International Conference on Database Theory*, ICDT '10, pages 4–33, New York, NY, USA, 2010. ACM.

[SPF+07] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, 41(3):275–287, March 2007.

[SPPH10] Martin Schoeberl, Wolfgang Puffitsch, Rasmus Ulslev Pedersen, and Benedikt Huber. Worst-case execution time analysis for a java processor. *Softw. Pract. Exper.*, 40(6):507–542, May 2010.

[Spr14] Springer, editor. *Software Language Engineering*, 2008–2014.

[Sta14] William Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall, 2014.

[SVC06] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

[The12] The OSGi Alliance. OSGi Service Platform Core Specification, Release 5.0, June 2012. http://www.osgi.org/Specifications/.

[TJDB06] Gerald Tesauro, Nicholas K Jong, Rajarshi Das, and Mohamed N Bennani. A hybrid reinforcement learning approach to autonomic resource allocation. In *Autonomic Computing, 2006. ICAC'06. IEEE International Conference on*, pages 65–73. IEEE, 2006.

[UK98] Ronald C. Unrau and Orran Krieger. Efficient sleep/wake-up protocols for user-level IPC. In *Int. Conf. on Parallel Processing (ICPP '98), Minnesota*, pages 560–569, 1998.

[VBMA11] Alex Villazón, Walter Binder, Philippe Moret, and Danilo Ansaloni. Comprehensive aspect weaving for java. *Science of Computer Programming*, 76(11):1015 – 1036, 2011. Special Issue on Principles and Practice of Programming in Java (PPPJ 2008).

[vdBCOV05] M. G. J. van den Brand, B. Cornelissen, P. A. Olivier, and J. J. Vinju. Tide: A generic debugging framework — tool demonstration —. *Electron. Notes Theor. Comput. Sci.*, 141(4):161–165, December 2005.

[vdBvDH+01] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The asf+sdf meta-environment: A component-based language development environment. *Electronic Notes in Theoretical Computer Science*, 44(2):3 – 8, 2001. LDTA'01, First Workshop on Language Descriptions, Tools and Applications (a Satellite Event of {ETAPS} 2001).

[VDKV00] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *Sigplan Notices*, 35(6):26–36, 2000.

[Voe10] Markus Voelter. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2010.

[vOvdLKM00] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The Koala Component Model for Consumer Electronics Software. *Computer*, 33(3):78–85, 2000.

[VSBK14] Markus Voelter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. Towards user-friendly projectional editors. In Benoît Combemale, DavidJ. Pearce, Olivier Barais, and JurgenJ. Vinju, editors, *Software Language Engineering*, volume 8706 of *Lecture Notes in Computer Science*, pages 41–61. Springer International Publishing, 2014.

[Wea13] Vincent M. Weaver. Linux perf_event features and overhead. In *FastPath Workshop*, 2013.

[Wea15] V.M. Weaver. Self-monitoring overhead of the linux perf_event performance counter interface. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, pages 102–111, March 2015.

[WG05] Hui Wu and Jeff Gray. Testing domain-specific languages in eclipse. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 173–174, New York, NY, USA, 2005. ACM.

[WGM08] Hui Wu, Jeff Gray, and Marjan Mernik. Grammar-driven generation of domain-specific language debuggers. *Softw. Pract. Exper.*, 38(10):1073–1103, August 2008.

[WHR14] J. Whittle, J. Hutchinson, and M. Rouncefield. The state of practice in model-driven engineering. *Software, IEEE*, 31(3):79–85, May 2014.

[wil13] The wildfly website, 2013.

[WPC+11] Chih-Sheng Wang, Guillermo Perez, Yeh-Ching Chung, Wei-Chung Hsu, Wei-Kuan Shih, and Hong-Rong Hsu. A method-based ahead-of-time compiler for android applications. In *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '11, pages 15–24, New York, NY, USA, 2011. ACM.

[you03] Yourkit, 2003.

[ZGC09] Ji Zhang, Heather J. Goldsby, and Betty H.C. Cheng. Modular verification of dynamically adaptive systems. In *Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development*, AOSD '09, pages 161–172, New York, NY, USA, 2009. ACM.

[ZWK14] Di Zheng, Jun Wang, and Ben Kerong. Research of context-aware component adaptation model in pervasive environment. In *Proceedings of the 2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, DASC '14, pages 496–501, Washington, DC, USA, 2014. IEEE Computer Society.

List of Figures

2.1 A MAPE-K loop to support system reconfiguration based on resource consumption . . . 11
2.2 A method is rewritten to collect data about memory consumption. . . . 22

3.1 Open-class mechanism in K3-AL, three views are shown: how the developer of the library see it (Fig 3.1a), how the user of the library see it (Fig 3.1b), and how it is perceived by the tools (Fig 3.1c). . . . 33
3.2 Organizing abstractions as DSLs. This resembles the idea of "language" embraced by Czarnecki and Eisenecker [CE00]. . . . 36
3.3 Simple OL-System to generate a plant in two dimensions. On the left, the OL-System is represented using a DSL, on the right we show the tree that can be generated using such OL-System, below is the representation in Java. . . . 38
3.4 The two mechanisms to account for resource consumption when components interact. . . . 42
3.5 Assessing the relevance of the aforementioned approaches. The location in terms of overhead and usability is given for each approach. Furthermore, the area of each circumference indicates how general the approach is (the larger the better). . . . 52
3.6 This subway map shows how this research contributes to the state-of-the-art on supporting resource-awareness. . . . 57

4.1 Heuristic extension point in Scapegoat. This illustrates the class diagram. 65
4.2 A sequence diagram showing how the extension point to define heuristics in Scapegoat is used. . . . 66
4.3 The component configuration for our crisis-management use-case. . . . 71
4.4 Execution time for tests using the DaCapo Benchmark . . . 72
4.5 Comparison of execution time for tests using two different memory monitoring techniques . . . 73
4.6 Execution time for some use cases under different monitoring policies. . 75
4.7 Delay time to detect fault with a component size of 115 classes. . . . 76
4.8 Delay time to detect fault with a component size of four classes. . . . 76

4.9 Execution time of main task with a component size of 115 classes. . . . 77
4.10 Execution time of main task with a component size of four classes. . . . 77
4.11 Architecture of MdMS along with Scapegoat and additional components to adapt the system. . . . 78
4.12 Time to obtain the reply to all requests. . . . 80
4.13 Average delay time to detect a faulty *sosie*. . . . 80

5.1 Squirrel approach for resources reservation . . . 85
5.2 Reserving CPU by mapping components to cgroups . . . 89
5.3 CPU overhead caused by resource management during the execution of Dacapo benchmarks. . . . 92
5.4 Memory overhead caused by resource management during the execution of Dacapo benchmarks. . . . 92
5.5 Average deployment time per component using different strategies . . . 93
5.6 Comparing latency of different IPC mechanisms . . . 94
5.7 Comparing bandwidth of different IPC mechanisms . . . 94
5.8 Communication throughput for different channels. . . . 94

6.1 Objects reachable from the Client class. Observe that only one object is not reachable. . . . 100
6.2 Memory snapshot with three linked lists . . . 101
6.3 Global view of the system. In this case, three memory profilers are defined. 102
6.4 Meta-model for representing customized memory profilers . . . 103
6.5 Concrete grammar of the language. For the sake of clarity, we are using an ambiguous grammar to describe the expression language. . . . 108
6.6 Snapshot of memory with one K3 Object. The idea is to compute the memory used by the shaded objects. . . . 114
6.7 Developer viewpoint. Memory profilers are built from the description of software abstractions. . . . 116
6.8 Viewpoint of developers. Memory profilers are built from the description of software abstractions. . . . 117
6.9 Viewpoint of users. Memory profilers are black-boxes accessed through Java interfaces. Data collected is in the form of plain Java objects. . . . 118
6.10 Representation of profiling data in Java, as users of profilers see it. Accessing these structures is useful to support resource awareness. . . . 119
6.11 Evaluating an expression through partial evaluation. . . . 120
6.12 Overhead on execution time compared to the execution without memory analysis for different tests in the DaCapo Benchmark . . . 125
6.13 Analysis time with default input size . . . 126
6.14 Analysis time with large input size . . . 126

6.15 Analysis time for real applications. It shows the time needed to compute an analysis just once. The analysis aims at finding the consumption of the top components 128

Résumé

Aujourd'hui, les systèmes logiciels sont omniprésents. Parfois, les applications doivent fonctionner sur des dispositifs à ressources limitées. Toutefois, les applications nécessitent un support d'exécution de faire face à de telles limitations. Cette thèse aborde le problème de la programmation pour créer des systèmes «conscient des ressources» supporté par des environnements d'exécution adaptés (MRTEs). En particulier, cette thèse vise à offrir un soutien efficace pour recueillir des données sur la consommation de ressources de calcul (par exemple, CPU, mémoire), ainsi que des mécanismes efficaces pour réserver des ressources pour des applications spécifiques. Dans les solutions existantes, nous trouvons deux inconvénients importants. Les solutions imposent un impact important sur les performances à l'exécution des applications. La création d'outils permettant de gérer finement les ressources pour ces abstractions est encore une tâche complexe. Les résultats de cette thèse forment trois contributions:

- Un cadre de surveillance des ressources optimiste qui réduit le coût de la collecte des données de consommation de ressources.
- Une méthodologie pour sélectionner les le support d'exécution des composants au moment du déploiement afin d'effectuer la réservation de ressources.
- Un langage pour construire des profileurs de mémoire personnalisées qui peuvent être utilisés à la fois au cours du développement des applications, ainsi que dans un environnement de production.

Abstract

Software systems are more pervasive than ever nowadays. Occasionally, applications run on top of resource-constrained devices where efficient resource management is required; hence, they must be capable of coping with such limitations. However, applications require support from the runtime environment to properly deal with resource limitations. This thesis addresses the problem of supporting resource-aware programming in execution environments. In particular, it aims at offering efficient support for collecting data about the consumption of computational resources (e.g., CPU, memory), as well as efficient mechanisms to reserve resources for specific applications. In existing solutions we find two important drawbacks. First, they impose performance overhead on the execution of applications. Second, creating resource management tools for these abstractions is still a daunting task. The outcomes of this thesis are three contributions:

- An optimistic resource monitoring framework that reduces the cost of collecting resource consumption data.
- A methodology to select components' bindings at deployment time in order to perform resource reservation.
- A language to build customized memory profilers that can be used both during applications' development, and also in a production environment.