Science of Computer Programming 78 (2013) 987–1009

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

A hybrid approach for resource-based comparison of adaptable Java applications[☆]

Marco Autili[*], Paolo Di Benedetto, Paola Inverardi

Dipartimento di Informatica, Università degli Studi dell'Aquila, L'Aquila, Italy

ARTICLE INFO

Article history:
Received 9 April 2011
Received in revised form 7 November 2011
Accepted 31 January 2012
Available online 11 February 2012

Keywords:
Adaptable applications
Analysis
Tool support

ABSTRACT

During the last decade, context-awareness and adaptation have been receiving significant attention in many research areas. For application developers, the heterogeneity of resource-constrained mobile terminals creates serious problems for the development of mobile applications able to run properly on a large number of different devices. Thus, resource awareness plays a crucial role when developing such applications. It identifies the capability of being aware of the resources offered by an execution environment, in order to decide whether that environment is suited to receive and execute the application. Within this line of research, we propose Chameleon, a framework that provides both an integrated development environment and a proper context-aware support to adaptable Java applications for limited devices. In this paper we present the novel hybrid (from static to dynamic) analysis approach that Chameleon uses for inspecting (adaptable) Java programs with respect to their resource consumption in a given execution environment. This analysis permits to quantitatively compare alternative versions of the same program. The analysis is based on a resource model for specifying resource provisions and consumptions, and a parametric transition system that performs the actual analysis.

1. Introduction

The technological changes we are experiencing today and the widespread use of small computing devices are rapidly evolving the way we interact and use technological infrastructures to perform everyday tasks. The future sees pervasive and invisible technology seamlessly used whenever and wherever needed. In this scenario, the need for adapting software applications becomes natural and, during the last decade, context-awareness and adaptation have been receiving significant attention in many research areas.

The development and the context-aware execution of adaptable applications is a big challenge for the research community and it is far to be solved. In the literature many valuable approaches have been proposed to this purpose, e.g., [1–10]. In particular, the work in [4] points out the little support that current mainstream programming languages and runtime environments provide for specifying adaptation and for supporting context awareness. One of the main issues to be faced is related to resource management, especially when dealing with resource-constrained devices. For application developers, the extreme heterogeneity of mobile terminals creates serious problems for the development of mobile applications able to run properly on a large number of different devices. Thus, resource awareness plays a crucial

[☆] This work is supported by the European Community's Seventh Framework Programme FP7/2007–2013 under grant agreements: number 257178 (project CHOReOS – Large Scale Choreographies for the Future Internet – www.choreos.eu), and number 231167 (project CONNECT – Emergent Connectors for Eternal Software Intensive Networked Systems – http://connect-forever.eu/).

[*] Corresponding author.
E-mail addresses: marco.autili@di.univaq.it (M. Autili), paolo.dibenedetto@di.univaq.it (P. Di Benedetto), paola.inverardi@di.univaq.it (P. Inverardi).

0167-6423/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2012.01.005

role when developing such applications. It identifies the capability of being aware of the resources offered by an execution environment, in order to decide whether that environment is suited to receive and execute an application.

In this paper we present a *hybrid* (from static to dynamic) analysis approach to quantitatively characterize the resource consumption of (adaptable) Java applications. This hybrid analysis approach has been developed in the scope of the Chameleon project[1] [11–13]. Chameleon is a framework based on an extension of the Java language that allows developers to easily implement adaptable applications in terms of *generic code* [13]. Such a generic code, opportunely preprocessed, generates a set of different *application alternatives*, i.e., different standard Java components that represent different ways of adapting the application [11,12]. Then, our analysis aims at characterizing each alternative in terms of the resources it demands to be executed in the execution environment offered by the requesting device. Specifically, the quantitative analysis technique allows for *relatively* comparing different adaptations alternatives of adaptable Java programs with respect to their resource demand, hence enabling developers to perform resource-aware programming. The approach deals with different types of resources (e.g., processor, memory, display, I/O capabilities, available radio interfaces) and does not aim at computing tight bounds for them. Rather, it aims at supporting a reasoning mechanism for selecting alternatives by performing a *relative* comparison of their resource demands. Thus, for a given execution environment, its efficacy can be assessed in terms of the ability to consistently order (and accordingly select) the different application alternatives with respect to their resource consumption.

The hybrid approach is based on a Resource Model that characterizes the notion of resources. It uses an Abstract Resource Analyzer (ARA) that *statically* scans the bytecode abstract syntax tree of Java applications and refines the associated resource sets using a transition system. The latter models the semantics of the bytecode in terms of resource consumption. ARA is parametric with respect to a Resource Consumption Profile that, dynamically sent by the Chameleon-enabled devices requesting an application, associates resource consumption to particular patterns of Java bytecode instructions. The profile permits to define the resource consumption associated to both basic instructions (e.g., `ipush`, `iload`) and to complex ones such as method calls. In particular, as it will be clear later, the *dynamic* phase of the analysis avoids to statically performs the expensive inter-procedural call-closure for library methods.

The main contributions of this paper with respect to previous works are: (i) the formalization and implementation of the novel hybrid approach that combines the static analysis performed by ARA with the dynamic analysis to derive the new Resource Consumption Profile, (ii) the Resource Model refined accordingly, (iii) a validation protocol and its supporting environment, (iv) the validation of the whole framework.

The paper is organized as follows: Section 2 gives a preamble on static analysis techniques and Section 3 briefly describes the Chameleon framework. Section 4 positions our analysis technique among related approaches. Section 5 presents the resource model, and Sections 6 and 7 present the hybrid analysis. A small example is worked out in Section 8, and the validation of the whole framework is discussed in Section 9. Section 11 concludes the paper and discusses future directions.

2. Preludium

Static program analysis [14] (and references therein) allow for predicting (precise or approximated) quantitative and qualitative properties related to the run-time behavior of a program without actually executing it. For instance, static analysis techniques allow for statically inferring cost-related properties (such as the estimation of the maximal number of loop iterations and the related worst-case execution time), as well as properties related to resource consumption (such as memory/heap usage and energy consumption). Many (often overlapping) kinds of theoretical and practical approaches exist in the literature: structural and control-flow analysis, data-flow and state-based analysis, interval analysis (used in optimizing compilers) and so on.

Practical static analysis is usually carried out by automatic tools – the analyzers – and can be performed against program models or intermediate representations, (e.g., Carmel intermediate representation of Java Card byte code), or against the actual (source or binary) program code. Models enable static analysis approaches which are independent from any (platform-constrained) technological solution. For instance, abstract models can be used for capturing (many of) the features and the structure of most object-oriented programming languages. Then, static analysis, being performed against these models, is (usually) able to compute under- or over-approximations of the actual program behavior (under the assumption that it is correctly modeled). When performed against the actual program code, static analysis allows for directly obtaining more precise values or refining their approximations; clearly, it is a matter of analysis computational cost.

Static analysis can be also combined with profiling. In fact, in most cases, cost-related properties (as well as resource-related ones) are expressed using platform-independent metrics. These metrics have been shown very useful especially for achieving (a first degree) approximation of the actual values. However, when heterogeneous platforms (having for instance different costs and resource consumption) have to be considered, it might be necessary to use parametric formulas that instantiated with platform-specific values (e.g., reported in a profile) can be transparently parsed by the same analysis technique.

Static analysis can also be used for (a priori) addressing safety issues, such as ensuring that a program will not write to a specific area of the memory or – for real-time systems – that it does not use more than a pre-established "fatal" time. To

[1] http://di.univaq.it/chameleon/.

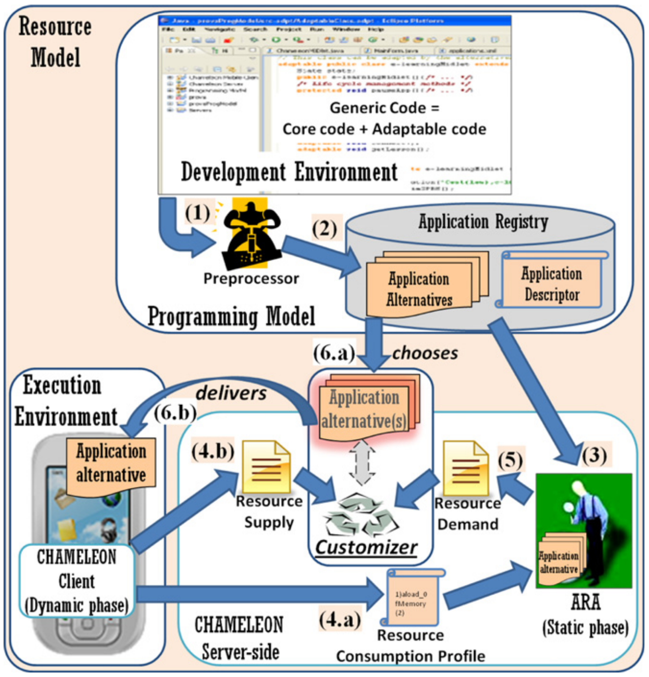

Fig. 1. CHAMELEON framework.

this end, certified static analyses based on Proof-Carrying Code (PCC) [15] play an important role; by attaching a proof to the mobile code the receiver is able to certify that the code adhere to its safety policy by checking the proof.

Abstract interpretation [16,17] is a theory that provides an approximation of the semantics of programming languages by means of mathematical domain/structures. It is able to extract information about the semantics of a program without actually computing it. This is done by abstracting on some elements of the semantics and by approximating the computation with the abstracted information. For example, instead of considering the actual values of variables it is possible to abstract these values by considering the variables type.

For the purposes of CHAMELEON, we do not consider the issues related to the PCC approach, such as trusting the proof checker and the formalism used to specify the proof. Hence, we fall within trusted domains that have been previously agreed. Moreover, we directly deal with the compiled code of the program which is more suitable for code mobility.

As it will be clear later, within CHAMELEON, part of the calculation of the resource demands is carried on by executing part of the analysis directly on the hosting execution environment. This analysis is referred to as CHAMELEON dynamic analysis[2] since, as opposed to static analysis, it examines a portion of the application code by executing it directly on the hosting device. In this setting, this paper describes a resource model and an abstract resource analyzer that combines static and dynamic analysis with a profiling technique in order to (i) infer bounds on platform-dependent resource consumption and (ii) adapt Java applications to the execution environment provided by mobile devices. As already introduced, these components are part of the CHAMELEON framework presented in the next section.

3. The CHAMELEON framework

CHAMELEON is a framework [18,11–13] to develop and deploy adaptable Java applications. The high-level architecture of CHAMELEON is shown in Fig. 1.

▶ **Programming model**. Referring to Fig. 1, the integrated *Development Environment* (DE) is based on a *Programming Model* that provides developers with a set of ad-hoc extensions to Java for easily coding adaptable applications in a flexible and declarative way. The applications' code is a *generic* code that consists of two parts: the *core* and the *adaptable* code — see in Fig. 1 the screenshot of the DE implemented as an Eclipse plugin.[3],[4] The core code represents the invariant portion of the application code. The adaptable code represents the variable portion that makes the code capable to adapt. The generic code is preprocessed by the CHAMELEON *Preprocessor* (1), also part of the DE, and a set of different standard Java application alternatives is derived and stored into the *Application Registry* together with the adaptable *Application Descriptor* (2). The

2 Dynamic software analysis is the analysis of software systems that is performed by executing programs on a real or virtual processor.
3 http://di.univaq.it/chameleon/.
4 http://sourceforge.net/projects/uda-chameleon/.

```
adaptable public class HelloWorld {
 adaptable public static void main(String[] s);
}

alternative class Alt1 adapts HelloWorld {
 public static void main(String[] s) {
    System.out.println(''Hello World'');
 }
}

alternative class Alt2 adapts HelloWorld {
 public static void main(String[] s) {
   ...
display.showAnimoticon(''Hello World'');
 }
}
```

Fig. 2. A glimpse of the programming constructs.

latter is an XML file reporting information related to the whole adaptable application, such as the number and the names of the different application alternatives, and the list of external library methods called by each one of them.

The behavioral variations of an adaptable application are specified up to the smallest unit of (object-oriented) behavioral description, i.e., up to method definition. Our programming model allows for the specification of how to adapt the application by providing a flexible and declarative way to describe which methods can be adapted, and which are the allowed adaptation alternatives. The approach is that of enriching the standard Java syntax with new constructs that allow the programmer to clearly specify behavioral variations.

In the following, we introduce basic concepts underpinning our programming model:

◦ An *adaptable method* is a method that can be adapted, i.e., its behavior can vary according to the context condition. It defines the entry-point for a behavior that can be adapted.

◦ An *adaptable class* is a class that declares one or more adaptable methods.

◦ An *alternative class* adapts an adaptable class. It contains the definition of how one or more adaptable methods can be actually adapted. It implements one of the possible actual behaviors that can be chosen when adapting a given adaptable method. An adaptable class can be adapted by one or more alternative classes. Moreover, by means of *adaptable alternative class* the Programming Model allows developers for specifying *alternatives of alternatives*, hence leading to a structured and powerful hierarchical specification of *generic* code.

◦ An *adaptable application* is an application that contains one or more adaptable classes.

◦ An *adaptation policy* is the set of all the adaptable classes, and their (hierarchies of) alternative classes, defined within the adaptable code. In other words, the adaptation policy determines the behavioral variations by defining which methods can be adapted and how.

◦ *Tags* can be used to specify conflicts among adaptable method implementations that cannot coexist in the standard classes derived by the preprocessor.

Without diving into details, in Fig. 2, we present a glimpse of how (some of) the new constructs introduced by the programming model look like. Fig. 2 shows a very basic adaptable class `HelloWorld` declaring the adaptable method `main(String[]  args)`, and an adaptation policy specifying the alternative classes `Alt1` and `Alt2`, which define different behaviors for the method `main`. After the preprocessing phase, this very simple adaptation policy will produce two different `HelloWorld` standard Java classes: one having the implementation of `main(String[]  args)` provided by the alternative class `Alt1`; the other one having the implementation provided by `Alt2`. These two classes represent two different application alternatives of the "HelloWorld" application: the first one simply displays text; the second one displays "Hello World" as an animated gif image (see Fig. 3). Obviously, the latter provides a better result but requires a graphic display and a supporting graphic library. A detailed presentation of the Programming Model and of the preprocessing phase can be found in [13].

▸ **Resource model**. The whole framework is built around the *Resource Model* described in Section 5.

▸ **Chameleon server-side**. Still referring to Fig. 1, *ARA* is an interpreter that, abstracting a standard JVM, is able to statically analyze the application alternatives (3) and derive their resource consumption (5). ARA is parametric with respect to the characteristics of the execution environment as described through the *Resource Consumptions Profile* dynamically sent by the device (4.a) upon requesting an application. This profile provides a characterization of the target execution environment, in terms of the impact that (basic and complex) Java bytecode instructions have on the resources. The transition system implemented by ARA is detailed in Section 6.

Fig. 3. Hello World adaptable application.

▶ **Chameleon client-side**. CHAMELEON-enabled devices (see left-hand side of Fig. 1) are devices deploying and running the CHAMELEON *Client* component that is able to retrieve contextual information. A CHAMELEON-enabled device provides a declarative description of the execution context in terms of the resources it supplies (i.e., *Resource Supply*) and the Resource Consumption Profile. As explained later, the list of calls to external library methods (as reported into the Application Descriptor) is used by the CHAMELEON Client during the dynamic phase of the analysis to derive the resource consumption associated to these method calls, hence avoiding ARA to statically perform the expensive inter-procedural call-closure of library methods.

▶ **Customizer**. On the server-side, the resource demands (5) of the application alternatives together with the resource supply sent by the device (4.b) are used by the *Customizer*. The latter, basing on a notion of compatibility (formalized in Section 5), is able to relatively compare and choose (6.a) the set of "best" suited alternatives, and deliver (6.b) one of them that (via the Over-The-Air (OTA) provisioning technique[5]) can be automatically deployed in the target device for execution. This means that we target devices providing their own implementation of OTA as provisioning technique, and that we rely on the different security policies as adopted by the different vendors for downloading and deploying applications.

It is worth to mention that, fully supported by the DE Eclipse plugin, the development of adaptable applications is carried out as a standard software engineering activity, and is independent from the activities performed by the other components. The server-side components might have a high computational complexity but they are executed on powerful server machines that (usually) do not suffer resource limitations. On the contrary, the client-side component is a very light-weight component and can be suitably executed on limited devices.

The CHAMELEON framework is fully implemented in and for Java. As already anticipated, the integrated development environment is realized as an Eclipse plugin and interested readers can download it together with all the other client and server modules from the CHAMELEON website[6] and the CHAMELEON source forge project.[7]

In [11,12], CHAMELEON has been used to realize a form of service adaptation in the IST PLASTIC project.[8] The goal of PLASTIC is the rapid and easy development, deployment and execution of adaptable services for the heterogeneous next-generation networks. We have used the programming model for the development of two adaptable applications (in the e-Health and e-Learning domains) for consuming and providing services targeted to mobile resource-constrained devices. In particular, in [11] we propose a mechanism that, exploiting ad-hoc methods for saving and restoring the (current) application state, enables services' evolution against context changes by dynamically un-deploying the no longer apt alternative and subsequently (re-)deploying a new alternative.

In [13] we describe how CHAMELEON has been used to implement, automatically deliver and deploy an adaptable MIDlet application that is able to suitably display a Mandelbrot fractal [19] on different limited devices. Section 9 describes how the adaptable Mandelbrot fractal has been used to validate CHAMELEON powered by the novel hybrid analysis technique.

4. Related work

ARA and the resource model can be related to other approaches to resource-oriented analysis, even though these approaches are not used for adaptation purposes.

5 http://developers.sun.com/mobility/midp/articles/ota/.
6 http://di.univaq.it/chameleon/.
7 http://sourceforge.net/projects/uda-chameleon/.
8 http://www.ist-plastic.org.

The MRG Project [20] proposes a framework (based on proof-carrying code) for giving correct guarantees that programs are free from run-time violations of resource bounds. The MRG approach is based on a resource-counting semantics that takes into account, through a resource component, the number of executed instruction and the maximum size of the stack frame. This line of research is then continued in the Mobius project [21]. A lot of work in the Mobius project is of relevance with respect to our work. For instance, in [22] the authors propose an analysis process that, at compile-time, adds cost relations for defining the cost of programs as a function on the size of input data. By inferring relations among the size of the arguments and by abstracting the recursive structure of the program, cost relations are specified in term of recursive equations. These cost relations are parametric with respect to the cost unit associated to the bytecode, and the cost unit appears as an abstract value within the equations. Indeed, the approach in [22] can be used for studying resource consumption and in [23] it is shown how a cost model for Java bytecode instructions is used for inferring the heap space consumed by each method in a program. They consider a subset of Java byte-code and, differently from us, take into account the size of input data. In principle, this allows them to be more precise with respect to the properties they prove and the guarantees they ensure. On the other hand, they do not directly handle adaptable applications and do not provide a flexible resource profile that allows for dynamically handling external library method calls.

In [24] an approach for analyzing resource consumption of bytecode is presented. The approach is restricted towards two directions. The first enables automated resource analysis of programs that iterate through data structures by translating the idea of amortized resource analysis to imperative languages. The second calculates resource bounds of iterative procedures controlled through numerical quantities by using polyhedral methods.

Focusing on the automatic computation of loops bounds, in [25] an inter-procedural technique is presented for calculating symbolic bounds on the number of statements executed by a procedure. The approach considers scalar inputs of the procedure and user-defined quantitative functions over input abstract data-structures, e.g., the length of a list. User annotations and quantitative functions can also be used for automatically deriving timing bounds.

In [26] the authors propose a path-sensitive analysis technique for inferring upper bounds of heap and stack usage under the assumption that they can be modeled as Presburger's formulas. The memory consumption expressions may have guards for capturing precise symbolic condition for each memory use/bound. An abstract definition in constraint form is built for recursive methods and loops. It is then solved using a fix point analysis. For the same purpose, the work in [27] prefers to use loop invariants for loops or user provided annotations in the place of a fix point analysis. In particular, the technique proposed in [27] computes non-linear parametric upper-bounds by calculating polynomial approximation of the dynamic memory required by a method. The technique assumes a region-based dynamic memory management by directly associating regions with the lifetime of methods. The approach "improves its performance" if a memory management that frees memory on demand is assumed.

In [28] the authors present a framework that, at both deployment- and run-time, is able to estimate the energy consumption of a distributed Java-based system. In particular, at the component level, they integrate an energy cost model and a communication cost model that allow for estimating the overall energy cost of each component. This information can then be used by software engineers in order to make decisions when adapting an application.

Tivoli [29] provides a Resource Modeler tool that enables specification and monitoring of resources by instrumenting the environment in which programs are executed.

Targeting embedded systems, in [30] an abstract resource model is proposed and attached to a process algebra that is used to formally model the system. This approach shares some similarities with us. However, the level of abstraction is different since we reason on the bytecode and not on a system model.

By exploring all the possible computation paths and by mapping JVM bytecode to a transition system, our analysis uses the same exhaustive technique of other existent tools such as Java Pathfinder (JPF) [31]. JPF checks for property violations (deadlocks or unhandled exceptions) traversing all possible execution paths. Differently from JPF, our analyzer abstracts the JVM with respect to resource consumption, i.e., the abstraction considers bytecode instructions behavior only by taking into account their effects on resources.

Finally, in the literature, the worst-case execution-time (WCET) problem has been deeply studied. An exhaustive survey of methods and tools for estimating (under precise assumptions) the WCET of hard real-time systems is given in [32]. Even though not strictly related to our work, this work provides an in-depth insight into the static and dynamic program analysis techniques used in this research area. In particular, the work in [33] also addresses the WCET problem and proposes a parametric timing analysis that instead of computing a single numeric value for WCET, as done by numeric timing analysis, derives symbolic formulas for representing the WCET. By accounting for parameters of the program, processor behavior, and by deriving parametric loop bounds, the proposed analysis allows for deriving strict WCETs, under precise assumptions.

For the sake of space, we cannot address all the enormous amount of work in static resource analysis. We are aware of this body of work and of the fact that the state of the art in resource analysis has moved far beyond the worst-case bounding that is produced by the analysis technique proposed in this paper. It can be argued that, for some resources, our analysis computes absolute bounds with less precision when compared with some of the previous approaches. This is certainly true, however for our concerns, i.e., comparing different adaptation alternatives in the same execution context, the accuracy of the computed bounds is adequate and our approach considers a larger set of resource types. More precisely, many of the previous approaches can deal with a smaller set of resource types and aim at giving a tighter absolute (over-) estimation of the consumption of each resource. Our approach instead, does not aim at establishing a punctual estimation of each resource consumption, rather it aims at supporting a reasoning mechanism that, at deployment time, allows for *relatively* comparing

```
define Energy as Natural
define Bluetooth as Boolean
define CPU as Natural
define Resolution as {low, medium, high}
```

Fig. 4. A resource definition.

and selecting adaptation alternatives. Its correctness thus is restricted to consistently reflect the ordering among resource consumptions of a set of alternatives that will run in the same execution context. In principle, being inspired by some of the previously discussed approaches, more precise bounds might be computed by taking into account input parameters and variable values. However, the different application alternatives need to be dynamically compared and selected prior deployment when input parameters are unknown, and this has to be done in a fully automatic way. To the best of our knowledge, none of the previous approaches might be easily exploited within a fully dynamic and automatic approach.

Moreover, we aim at providing developers with a practical and easy-to-use development and analysis approach of resource-adaptable applications. To this aim, our approach saves developers from the usage of complicated mechanisms and theories, which are not close to the common practice of software applications coding. Clearly, this choice has led us to reduce our expectations in terms of (theoretical) accuracy but has allowed us to fully implement our framework and experiment with the development (and the analysis) of adaptable Java applications in a way very similar to standard Java applications. However, according to the validation results discussed in Section 9, we can reasonably argue that the accuracy of CHAMELEON, in its experimental version, is adequate.

CHAMELEON also relates to many others approaches in the literature concerning context-aware adaptive systems. In [34] the authors presents the Subjective-C language. It provides dedicated abstractions to deal with context-specific method definitions, hence enabling run-time behavioral adaptation to context changes. As an extension of Objective-C, the behavior of Subjective-C objects depends not only on the messages they receive, but also on the current execution context. Indeed, the approach is based on previous work of the authors, namely the Ambience language [35]. Both languages use objects for representing particular run-time contextual situations. On the positive side, if developers a priori have a global vision of all the possible contextual situations, the approaches in [34,35] allow for the explicit specification of the adaptation logic. This logic can be specified in terms of context-specific methods to a priori treat all the possible contextual situations, hence enabling dynamic adaptation. On the contrary, the declarative and generative nature of the CHAMELEON programming model spares developers the writing of the adaptation logic and the a priori treatment of all the possible contextual situations and all the corresponding associations with behavioral variations. In fact, CHAMELEON developers separately define the smallest units of behavioral variations up to adaptable methods, and locally specify annotations on, e.g., upper bounds to iterations and recursive method calls. Then, the context-aware run-time support offered by CHAMELEON will automatically resolve the overall variability by calculating the global resource demand. Then, the adaptable application is properly customized to the current context condition. In this way, developers are not required to a priori have a global vision of all the possible contextual situations. On the negative side, the applications deployed by using CHAMELEON are customized with respect to the context at deployment time but, at run time, are frozen with respect to evolution. Section 10 better discusses this point.

Many other valuable approaches exist in the literature. For a further comparison with these other approaches we refer to our early work in [12,13].

5. Resource model

This section recalls basic concepts underlying our notion of resource, and extends the preliminary version of our resource model presented in [18,36] to meet the additional requirements of the new version of ARA as presented in Section 6.

From our perspective, a *resource* is a device related item or feature that enables the application to work correctly. CHAMELEON can deal with a predefined set of resources as defined by an ad-hoc XML file. The developer is supported by the DE Eclipse plugin (see Fig. 1) in selecting the resources of interest for the adaptable application under development. Additional resources can also be specified.

Some resources are subject to consumption (e.g., CPU power, heap space), while others, if present, are always available and never exhausted (e.g., libraries, radio interfaces). For our purposes, a resource is modeled as a typed identifier that can be associated to Natural, Boolean or Enumerated values. Natural values are used for consumable resources whose availability varies during execution. Boolean values define resources that can be present or not (i.e., non-consumable ones). Enumerated values can define non-consumable resources that provide a restricted set of admissible *ordered* values (e.g., screen resolution). Fig. 4 shows an example of a resource definition.

Definition 1 (*Resource Instance*). A resource instance is an association $res(val)$ where a resource identified by res is coupled to its value val.

Resource instances can be compared with respect to their associated values. For *natural* resources, the standard relational operators can be used; for *boolean* resources, we naturally assume that $false < true$; finally, for *enumerated* resources, the enumerated options are associated to natural numbers using a position-wise mapping according to the resource definition, and hence the specified order. Thus, in Fig. 4, we have that $low < medium < high$. We can use the notion of *resource instance* to describe the amount of a resource provided by an execution environment or needed by a program to be correctly executed. To this purpose, we introduce the following definition:

Definition 2 (*Resource Set*)**.** A resource set is a set of resource instances with no resource occurring more than once.

Definition 3 (*Resource Set Compatibility*)**.** A *resource set* $P = \{p_1, p_2, \ldots, p_n\}$ is compatible with a resource set $Q = \{q_1, q_2, \ldots, q_m\}$ if

1. **(Availability)** For every *resource instance* $p(v) \in P$ there exist a *resource instance* $q(t) \in Q$ such that $p = q$.
2. **(Wealth)** For every pair of *resource instance*s $p(v) \in P$ and $q(t) \in Q$ such that $p = q$ then $p(v) \le q(t)$. The $\le$ relation is *overloaded* in order to take into account the actual type of the resource.

We will write $P \lhd Q$ to assert that the resource set P is compatible with the resource set Q. The definition of compatibility between two resource sets is used to verify if the resource demand of an application alternatives is compatible with the resource supply provided by an execution environment, and hence if the application can run safely in that environment. The availability condition ensures that each type of demanded resource is supplied. The wealth condition ensures that for every resource demanded by an alternative a "sufficient amount" of that resource is supplied. In the following we introduce some operations over resource instances and resource sets. These operations are then used by ARA to compute the resource demand of the alternatives.

Definition 4 (*Resource Instance Sum Operator* $\oplus$)**.** Given two resource instances $r(v_1)$ and $r(v_2)$, the sum $r(v_1) \oplus r(v_2)$ is the resource instance defined as follow:

$$r(v_1) \oplus r(v_2) = \begin{cases} r(v_1 + v_2) & \text{if } \mathit{typeof}(r) = \text{Natural} \\ r(\mathit{max}(v_1, v_2)) & \text{otherwise} \end{cases}$$

where *typeof* (r) identifies the type of the resource r, and the *max* operator returns the maximum value according to the ordering relation associated to the resource type.

Definition 5 (*Resource Instance Scalar Multiplication Operator* $\odot$)**.** Given a *Natural n* and a resource instance $r(v)$, the scalar multiplication $n \odot r(v)$ is the resource instance defined as follow:

$$n \odot r(v) = \begin{cases} r(n * v) & \text{if } \mathit{typeof}(r) = \text{Natural} \\ r(v) & \text{otherwise} \end{cases}$$

The above operators are extended to *resource sets* as follows:

Definition 6 (*Resource Set Sum Operator* $\oplus$)**.** Given two resource sets P and Q, the resource set $P \oplus Q = S_1 \cup S_2 \cup S_3$ where:

- $S_1 = \{p(v) \mid \forall v, t \quad p(v) \in P \wedge p(t) \notin Q\}$
- $S_2 = \{q(v) \mid \forall v, t \quad q(v) \in Q \wedge q(t) \notin P\}$
- $S_3 = \{r(v) \oplus r(t) \mid \forall v, t \quad r(v) \in P \wedge r(t) \in Q\}$.

Definition 7 (*Resource Set Scalar Product Operator* $\odot$)**.** Given a Natural n, ad a resource set R, the resource set $n \odot R = \{n \odot r \mid r \in R\}$.

Note that, in its current version, Chameleon assumes a monotonic consumption of resources (e.g., for dynamic memory we do not consider any garbage collector policy). This clearly represents a limitation shared with many other approaches in the literature. Some of these approaches assume a memory management that frees memory on demand or at the end of methods lifetime. In Java this assumption does not hold. Moreover, as previously said, we are interested in *relatively* comparing and selecting different adaptation alternatives.

However, as future work, accounting for the results of the validation reported in Section 9, further taking inspiration from the literature (see Section 4), we plan to extend the current implementation of Chameleon to consider more sophisticated resource definitions. Then, the idea is to validate if the adoption of a consequently more expensive approach will led to a more precise selection, still considering that we perform a relative comparison of a predefined set of alternatives.

6. Abstract resource analyzer

This section introduces the notions needed for understanding the abstract resource analyzer, which will be then formalized in Section 7. As already said, ARA is an interpreter that derives the resource consumption of the application alternatives by abstracting the standard JVM. In this abstraction it considers only the effect that bytecode instructions have on the resources. ARA performs a worst-case analysis of the program and, by inspecting all the possible computation paths, returns an over-approximation of their resource demands. In order to have a more accurate and efficient approximation, the resource consumption, statically computed by ARA on the server side, is combined with the actual resource consumption of bytecode instructions and, in particular, those instructions for calling external library methods. To this purpose, the analysis is parametric with respect to the resource consumption profile dynamically computed and sent by the Chameleon Client as described in Section 6.1. Moreover, in Section 6.2 we describe how ARA exploits additional information that can be inserted at generic code level by means of annotations.

```
1) istore_1 → {CPU(2)}                    2) invoke.* → {CPU(4)}
3) .* → {CPU(1), Energy(1)}
4) invokestatic LocalDevice.getLocalDevice() → {Bluetooth(true), Energy(20)}
```

Fig. 5. A resource consumption profile.

6.1. Resource consumption profile

The impact that bytecode instructions have on the resources depends on the execution environment, since the same bytecode instruction may require different resources in different execution environments. Thus, a profile associates a resource consumption to particular patterns of Java bytecode instructions specified by means of *regular expressions*. Since the bytecode is a verbose language,[9] it is possible to define the resource consumption associated to both basic instructions (e.g., `ipush`, `iload`) and to complex ones such as method calls. In this respect, the resource consumption due to the execution of basic bytecode instructions is automatically computed by the CHAMELEON client component by benchmarking the hosting execution environment.

As already introduced in Section 3, the preprocessor is also able to derive the Application Descriptor (see Fig. 1). Among other information, it lists the calls to external library methods for each alternative. Specifically, for each external method call in the list, the programmer (supported by the CHAMELEON DE) specifies the worst-case call-context on the base of the application logic that has been coded. When a device requests an application, during the dynamic phase of the analysis, each library method is then benchmarked by the CHAMELEON Client in the target execution environment, and its actual resource consumption is computed. This consumption is used to specify into the profile the resources consumed by each library method. The advantage of combining the static and the dynamic phase is to avoid to statically perform the expensive inter-procedural call-closure for library methods. In fact, the static inter-procedural call-closure in Java is extremely expensive for even a simple "Hello World" program due to the use of standard libraries.

For both bytecode instructions and external methods, resource demands are estimated from the device characteristics by using Java system calls, such as, `System.CurrentTimeMilliseconds()`, `System.getProperty(‘‘battery level’’)`, `System.getProperty(‘‘com.nokia.mid.batterylevel’’)`. By default, the CHAMELEON benchmark comes with a set of predefined mappings between regular expressions and the resources consumed. Concerning bytecode instructions, the CHAMELEON client can be extended by coding other mappings. Concerning external methods the CHAMELEON DE allows, through a specific GUI, developers to add specific functions to be called to compute new resource demands.

Fig. 5 represents an example of a resource consumption profile for the resources defined in Fig. 4. In the last row, `Bluetooth (true)` states that a call to the `getLocalDevice()` static method of the `LocalDevice` class within the `javax.bluetooth` library requires the presence of a Bluetooth radio interface on the device. This library method call causes a consumption of Energy equal to 20 cost units. Note that the expression $.*$ matches every bytecode. In fact, CHAMELEON starts with a *default resource consumption profile* that, as better explained below, is refined by more specific expressions as computed by the CHAMELEON Client component. That is, in Fig. 5, the expression $.*$ means that by default any single bytecode instruction consumes (at least) 1 Energy cost unit and 1 CPU cost unit.

$$resBinding : ByteCodeRegExp \rightarrow ResourceSet$$

The function *resBinding* maps regular expressions to resource sets. In other words, it maps bytecode instructions to the resource consumption associated to the execution of the instructions matching the regular expression. If a bytecode instruction matches more than one regular expression that is associated to the consumption of the same resource, the *resBinding* function will return the resource consumption associated to the most specific regular expression. In fact, this resource consumption is supposed to be the most fitting resource demand. By the terms "most specific" we intend the smallest regular expression according to the following (partial) order defined over regular expressions: given two regular expressions r_1 and r_2, $r_1 \sqsubseteq r_2 \Leftrightarrow \mathcal{L}(r_1) \subseteq \mathcal{L}(r_2)$ where $\mathcal{L}(r)$ is the language generated by the regular expression r (i.e., r_1 is more specific than r_2). For example, `invokevirtualItem.getData()Ljava/lang/String;` matches the regular expressions 2 and 3 in Fig. 5. Both expressions 2 and 3 specify CPU resource consumption but, since the former has a more specific regular expression, its associated CPU consumption is considered. Thus, as expression 3 specifies also an Energy resource consumption, *resBinding*(`invokevirtual Item.getData()Ljava/lang/String;`) = $\{CPU(4), Energy(1)\}$. Dealing with partial order it can happen that two or more expressions are "equally specific". In this case ARA chooses the expression matched as first.

6.2. Annotations

Annotations are specified at the generic code level by means of calls to the "do nothing" methods of the `Annotation` class shown in Fig. 6. In this way, after compilation, annotations are encoded in the bytecode through well recognizable

[9] This is particularly true in the case of method invocation where the invoked method is uniquely identified by a fully qualified identifier in term of its base class identifier, name, and formal parameters.

```
public class Annotation {
  public static void resourceAnnotation(String ann) {};
  public static void loopAnnotation(int n) {};
  public static void callAnnotation(int n) {};
}
```

Fig. 6. Annotation class.

method calls so that they can be easily processed by ARA. The Chameleon Development Environment supports the insertion of three types of annotations:

Resource annotation: used to directly express a resource demand (i.e., a resource set) which is processed by ARA and contributes to the resource demand of the current computation. This kind of annotations can be placed everywhere in the source code, but are typically used to specify the resource consumption due to the execution of external methods. They represent an alternative to the use of resource consumption profiles for all these cases when the resource consumption is independent from the device and well known to the developer (especially for Boolean or Enumerated resources). For instance, in Fig. 7 the method call `Annotation.resourceAnnotation("CPU(20), Bluetooth(true)");` (and hence the corresponding bytecode instructions at lines 36 and 38) is used to specify the resource demand of the method `printBT(String)` of the class `PrintServices` external to the application.

Loop annotation: used to express an upper bound on the number of loop iterations and must be specified as first instruction of the loop body. For instance, in Fig. 7 the method call `Annotation.loopAnnotation(100);` (and hence the corresponding bytecode instructions at lines 10 and 12) is used to specify that the loop will be repeated no more than 100 times.

Call annotation: used to express an upper bound on the number of times a recursive method can be recursively called. This kind of annotations are placed just before recursive method calls and can be determined in the same way as loop annotations.

Note that, in many cases, the upper bounds for loop and call annotations can easily be specified at "coding-time" directly by programmers, benefiting from the deep knowledge of the application logic. However, these bounds can also be inferred by static prediction techniques [25–27,37–40] (just to cite some), or experimentally retrieved (e.g., by instrumenting the code and running ad-hoc test cases).

It is worth to mention that we do not use standard Java annotation mechanism since we need to place annotations also on loops, blocks, and simple statements. By contrast, Java SE 6 permits annotations only on declarations. Indeed, there exist more than one Java implementation that supports expression and statement annotations [41,42]. However, the JSR 308 proposal [43] extends the Java annotation mechanism to permit annotations on any occurrence of a type. There is also interest in further extending this proposal so that annotations can be placed not only on type references, but also on loops, blocks, and simple statements. Type annotations were planned to be part of the Java 7. In the future we will consider this extension.

6.3. Bytecode abstract syntax tree

As previously said, the analysis scans each possible execution path. Some of these paths may share the same blocks of bytecode instructions and, in our analysis, they will always lead to the same resource consumption. Thus, shared blocks are analyzed only once and the result suitably reused. To this purpose ARA reconstructs the Java Abstract Syntax Tree (AST) of the methods' bytecode by exploiting well known structural analysis techniques [44–46]. These techniques analyze the bytecode and construct a hierarchical tree of subgraphs (Structure Encapsulation Tree — SET) that represent source code patterns. By traversing the SET it is then possible to reconstruct the source code abstract syntax tree. In addition, the AST allows for identifying the scope of loop annotations by associating the blocks of bytecode instructions to the bodies of the (source code) loops they belong to. In particular, our framework implementation makes use of SOOT [47], a tool for structural analysis that we have modified for our analysis purposes (e.g., for annotations handling).

In the following, we customize the "general purpose" Java AST definition to the Chameleon approach:

Definition 8 (*Bytecode Abstract Syntax Tree*)**.** A BAST is a finite, node and arc labeled, directed tree where internal nodes represent Java control flow constructs (e.g., WHILE) and the children of these nodes represent meaningful components of the construct (e.g., WHILE *condition* and *body*). The role played by a child node in the construct is used to label the arc entering the child node itself. Finally, leaf nodes represent JVM instructions. Additional information related to annotations can be attached to any node.

For instance, in Fig. 8, the nodes 36 and 38 are marked to indicate that they refer to the resource annotation corresponding to the bytecode of Fig. 7, lines 36 and 38.

```java
abstract class Item {
  public String title;
  public abstract String getData();
}
class Book extends Item {
  public String getData(){
    String data="Title=" + title;
    return data;    }
}
class DVD extends Item {
  public String getData(){return "...";}
}
public class Library {
  static Item[] archive = new Item[100];
  public static void print(String title){
    int i=0;
    while (i<archive.length) {
      Annotation.loopAnnotation(100);
      if (archive[i].title==title){
        String s=archive[i].getData();
        Annotation.resourceAnnotation("CPU(20),
                                     Bluetooth(true)");
        PrintServices.printBT(s);
        break;
      } else
        i++;
    }
  }
.....
}
```

Java source code

```
0: iconst_0
1: istore_1
2: iload_1
3: getstatic Library/archive[LItem;
6: arraylength
7: if_icmpge —> 54
10: bipush  100
12: invokestatic Annotation.loopAnnotation(I)V
15: getstatic Library/archive[LItem;
18: iload_1
19: aaload
20: getfield Item/titleLjava/lang/String;
23: aload_0
24: if_acmpne —> 48
27: getstatic Library/archive[LItem;
30: iload_1
31: aaload
32: invokevirtual Item.getData()Ljava/lang/String;
35: astore_2
36: ldc "CPU(20), Bluetooth(true)"
38: invokestatic Annotation.resourceAnnotation(Ljava/lang/String;)V
41: aload_2
42: invokestatic PrintServices.printBT(Ljava/lang/String;)V
45: goto —> 54
48: iinc
51: goto —> 2
54: return
```

`print(String)` method bytecode

Fig. 7. A simple Java application and the bytecode of the `print(String)` method.

Table 1
Internal node labels.

Internal Node Label	Comment	Children Arc Labels
BLOCK	Sequence of statements	Element*
IF_ELSE	IF and IF THEN ELSE	Condition, trueBranch [falseBranch]
WHILE	WHILE and FOR	Condition, body
DOWHILE	DO_WHILE	Condition, body
SWITCH	SWITCH	Choice, case*
TRY	TRY	Try, catch*, [finally]

Table 1 reports the valid labels for an internal node, and the corresponding arc labels for the roles played by its children. The symbol * postfixed to a child arc label means that a node can have more than one child playing that role. For example, a SWITCH node has only one child representing the *choice* component and more than one child representing a *case alternative*. Children arc labels reported in square brackets represent optional components. Note that, since compilers translate `for` statements similarly to `while` statements, both of them are represented as WHILE nodes.

Fig. 7 shows a simple Java application and the bytecode generated by Sun *javac* compiler for the method `print(String)`. Fig. 8 shows the BAST for that bytecode, where bytecode instructions have been replaced by their line numbers. Note that, performing a pre-order visit, we obtain the bytecode instructions in the same order as the one resulting from the compilation process. In Section 8 we will show how the ARA transition system behaves with respect to this BAST.

Hereafter in this section, we define some functions that abstract the structural analysis process implemented within our framework and that will be used in Sections 6.4 and 7 for defining the ARA transition system.

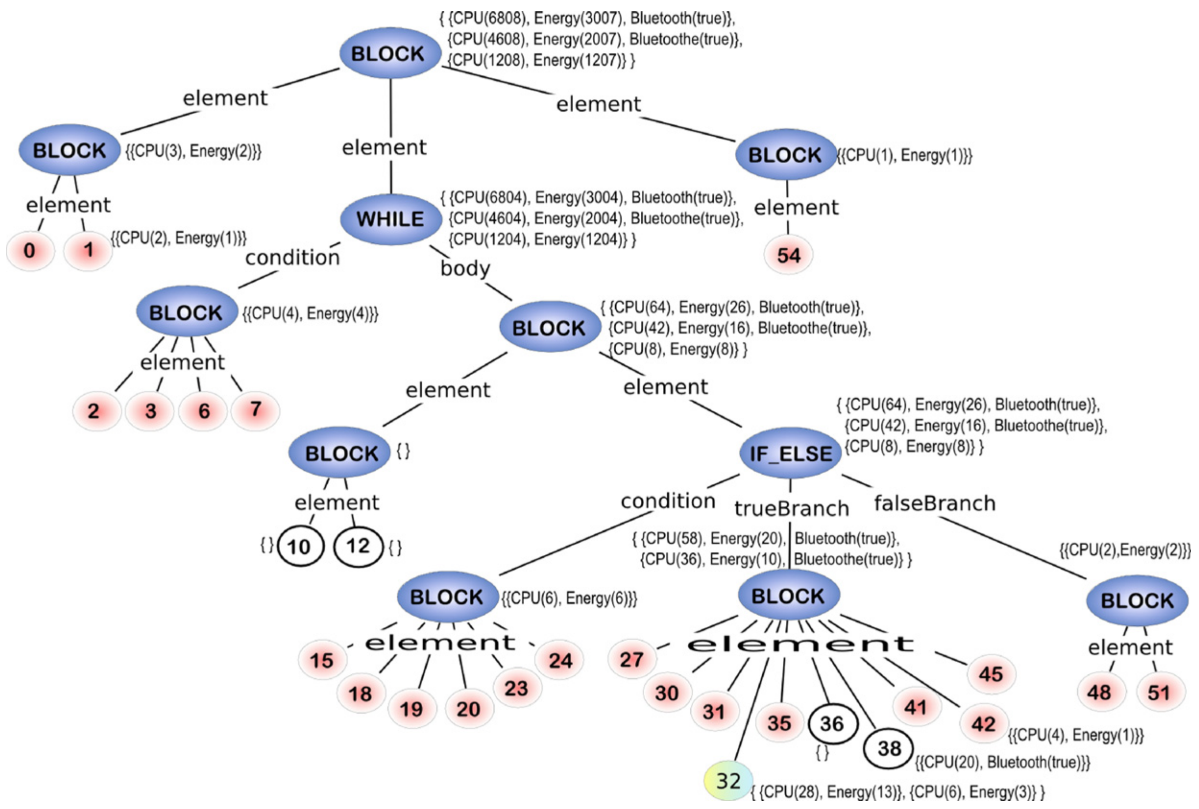
BLOCK
{ {CPU(6808), Energy(3007), Bluetooth(true)}, {CPU(4608), Energy(2007), Bluetoothe(true)}, {CPU(1208), Energy(1207)} }
element
BLOCK
{{CPU(3), Energy(2)}}
WHILE
{ {CPU(6804), Energy(3004), Bluetooth(true)}, {CPU(4604), Energy(2004), Bluetoothe(true)}, {CPU(1204), Energy(1204)} }
BLOCK
{{CPU(1), Energy(1)}}
condition
body
{{CPU(2), Energy(1)}}
{{CPU(4), Energy(4)}}
{ {CPU(64), Energy(26), Bluetooth(true)}, {CPU(42), Energy(16), Bluetoothe(true)}, {CPU(8), Energy(8)} }
IF_ELSE
trueBranch
falseBranch
{ {CPU(58), Energy(20), Bluetooth(true)}, {CPU(36), Energy(10), Bluetoothe(true)} }
{{CPU(2),Energy(2)}}
{{CPU(6), Energy(6)}}
{{CPU(4), Energy(1)}}
{{CPU(20), Bluetooth(true)}}
{ {CPU(28), Energy(13)}, {CPU(6), Energy(3)} }
0 1 54 2 3 6 7 10 12 15 18 19 20 23 24 27 30 31 32 35 36 38 41 42 45 48 51

Fig. 8. BAST of the `print(String)` method bytecode.

BuildBAST: *MethodsBytecode* $\rightarrow$ *BAST*. Given the bytecode of a method in the set *MethodsBytecode*, the *BuildBAST* function returns the corresponding BAST.

IsLeaf: *BAST* $\rightarrow$ $\{true, false\}$. Given a BAST, the *isLeaf* function returns *true* if the BAST has only one node (i.e., it is a leaf), *false* otherwise.

Label: *BAST* $\rightarrow$ *NodeLabels* where *NodeLabels* is the set of all possible node labels (see Table 1). Given a BAST, the *Label* function returns the label of the root node.

Children: *BAST* $\times$ *ArcLabel* $\rightarrow$ $\mathcal{P}(BAST) \cup \{\texttt{null}\}$ where *ArcLabel* is the set of all possible arc labels (see Table 1) and $\mathcal{P}(BAST)$ represents the powerset of *BAST*. Given a BAST and an arc label, the *Children* function returns the set of subtrees of the root node having that arc label. If the root has no subtrees with the specified arc label, then `null` is returned. For example, referring to Fig. 8, let S be the subtree representing the while condition, $Children(S, element)$ returns the set of subtrees $\{2, 3, 6, 7\}$.

The following functions are used by ARA to retrieve annotations associated to the specified BAST:

ResourceAnnotation: *BAST* $\rightarrow$ *ResourceSet*. Given a BAST, if it is a leaf labeled with an `invokestatic Annotation.resourceAnnotation(ann)V` bytecode instruction, the *ResourceAnnotation* function returns the resource set that has been specified through the parameter *ann*, the empty set otherwise (see for instance $\{CPU(20), Bluetooth(true)\}$ in the source code of Fig. 7).

LoopAnnotation: *BAST* $\rightarrow$ $\mathcal{N}$. Given a BAST, if it represents a loop body, the *LoopAnnotation* function returns the integer indicating the upper bound of loop repetitions specified through an annotation.

CallAnnotation: *BAST* $\rightarrow$ $\mathcal{N}$. Given a BAST, if it is a leaf labeled with an "`invoke`" bytecode instruction (i.e., `invokespecial`, `invokestatic`, `invokevirtual` or `invokeinterface`), the *CallAnnotation* function returns the upper bound of the number of recursive calls specified through an annotation.

Since *Annotation* method calls are introduced only for internal use, the corresponding bytecode instructions (comprising the initialization of their formal parameters) will be removed from the application code. In this way they will not contribute to the final resource demand of the application.

IsAnnotation: *BAST* $\rightarrow$ $\{true, false\}$. Given a BAST, if it is a leaf, the *IsAnnotation* function returns *true* if its label represents either (i) a call to a method of the Annotation class or (ii) the initialization of its formal parameters, *false* otherwise.

(I.1 null BAST Rule)

$$\frac{}{\langle e, b, M, \mathtt{null}\rangle \rightarrow_{ARA} \emptyset}$$

(I.2 BLOCK Node Rule)

$$\frac{\begin{array}{c} Label(n) = \mathtt{BLOCK} \\ Children(n, \mathtt{element}) = Z = \{z_1 \ldots . z_k\} \\ \forall z_i \in Z \quad \langle e, b, M, z_i\rangle \rightarrow_{ARA} C_i \\ C = \bigoplus_{i=1}^{k} C_i \end{array}}{\langle e, b, M, n\rangle \rightarrow_{ARA} C}$$

(I.3 IF_ELSE Node Rule)

$$\frac{\begin{array}{c} Label(n) = \mathtt{IF\_ELSE} \\ Children(n, \mathtt{condition}) = \{n_{cond}\} \\ Children(n, \mathtt{trueBranch}) = \{n_{true}\} \\ Children(n, \mathtt{falseBranch}) = \{n_{false}\} \\ \langle e, b, M, n_{cond}\rangle \rightarrow_{ARA} C_{cond} \\ \langle e, b, M, n_{true}\rangle \rightarrow_{ARA} C_{true} \\ \langle e, b, M, n_{false}\rangle \rightarrow_{ARA} C_{false} \\ C_1 = C_{cond} \oplus C_{true} \\ C_2 = C_{cond} \oplus C_{false} \\ C = C_1 \cup C_2 \end{array}}{\langle e, b, M, n\rangle \rightarrow_{ARA} C}$$

(I.4 WHILE Node Rule)

$$\frac{\begin{array}{c} Label(n) = \mathtt{WHILE} \\ Children(n, \mathtt{condition}) = \{n_{cond}\} \\ Children(n, \mathtt{body}) = \{n_{body}\} \\ t = LoopAnnotation(n_{body}) \\ \langle e, b, M, n_{cond}\rangle \rightarrow_{ARA} C_{cond} \\ \langle e, b, M, n_{body}\rangle \rightarrow_{ARA} C_{body} \\ C_{cond'} = (t+1) \odot C_{cond} \\ C_{body'} = t \odot C_{body} \\ C = C_{cond'} \oplus C_{body'} \end{array}}{\langle e, b, M, n\rangle \rightarrow_{ARA} C}$$

Fig. 9. BAST internal node transition system rules.

6.4. Abstracting the JVM

This section succinctly defines the formal model used by ARA to abstract the Java Virtual Machine.

$$\begin{array}{lcl} Environment & = & \{e \mid e : ClassID \rightarrow Class \cup \{\perp\}\} \\ Class & = & ClassID \times Method \\ Method & = & \{m \mid m : MethodID \rightarrow MethodInfo \cup \{\perp\}\} \\ MethodInfo & = & ClassID \times \mathcal{N} \times MethodsBytecode \cup \{\perp\} \end{array}$$

The *Environment* is the set of all functions that are used to access the classes and interfaces declared within application alternatives (see Section 3). Each application alternative has associated a function $e \in Environment$ that takes a class/interface identifier from the set *ClassID* and returns an element of type *Class*. Undefined ($\perp$) is returned if there is no class/interface information associated to the identifier.

Class is a pair whose first element is a class/interface identifier *id*. The second element is a function $m \in Method$ that (being associated to *id*) retrieves information for all the methods declared within the class/interface identified by *id*. *Method* functions take a fully qualified method identifier (i.e., `ClassName.methodName(params...)`) and returns a tuple in *MethodInfo*. The tuple contains the class identifier of the class/interface in which the method has been declared, an integer indicating the number of arguments the method expects, and the method bytecode.

$$Bast : Environment \times MethodID \rightarrow BAST \cup \{\mathtt{null}\}$$

Given an environment *e* and a fully qualified method identifier *id*, the *Bast* function returns the BAST of the method identified by *id*. Basically, it extracts the class/interface of the method and uses the environment to retrieve the function in *Method* associated to that class/interface. The function returns *null* if no implementation is provided for the method identified by *id*, e.g., it is an abstract method.

7. ARA formal definition

In this section we formalize the abstract resource analyzer by using a path-insensitive big-step transition system. The analyzer performs a scanning of the application code by traversing the BAST corresponding to each method, and by incrementally refining the associated resource sets (see Definition 2) representing the resource demands of the application.

Section 7.1 describes the configurations of the ARA transition system. Sections 7.2 and 7.3 detail some rules (shown in Figs. 9 and 10) for internal and leaf BAST nodes, respectively. For the sake of space, some rules (such as the ones for treating exceptions handling, threads and switch nodes) are not reported. Refer to [48] for a complete description of them.

7.1. Transition system configurations

The ARA configurations are defined as follow:

$$\begin{array}{lcl} \Gamma_{ARA} & : & \{\langle e, b, M, n\rangle\} \cup \mathcal{P}(ResourceSet) \\ \mathcal{T}_{ARA} & : & \mathcal{P}(ResourceSet) \end{array}$$

where $e \in Environment$, $b \in ResourceBinding$, $M \subseteq MethodID$, $n \in BAST$. $\mathcal{P}(ResourceSet)$ is the set of resource sets used by the current computation. $ResourceBinding = \{b \mid b : ByteCodeRegExp \rightarrow ResourceSet\}$ is a set of *resBinding* functions that model resource consumption profiles as defined in Section 6.1. M is a set of method identifiers and it is used to handle recursive methods. It maintains a list of all methods analyzed during the current computation in order to avoid the transition system to enter an infinite loop. Finally, n represents the BAST of the method being analyzed.

Typically, a Java application starts its execution from the method `main (String[])`, thus the typical initial configuration is $\langle e, b, \{\texttt{Main.main (String[])}\}, Bast(e, \texttt{Main.main(String[])})\rangle$ where e is the environment representing the current application and b is the binding function representing the resource consumption profile characterizing the current execution environment.

The final configurations $\mathcal{T}_{ARA}$ are sets of resource sets, each one of them representing the resource demand of a possible execution path. To properly handle such sets, we need to extend the definition of $\oplus$ and $\odot$ operators over sets of *ResourceSet*s.

Definition 9 (*Sets of ResourceSets Sum Operator* $\oplus$)**.** Given two sets of resource sets R_1 and R_2, $R_1 \oplus R_2$ is the set of resource sets $R = \{r_1 \oplus r_2 \mid r_1 \in R_1, r_2 \in R_2\}$.

Definition 10 (*set of ResourceSets Scalar Product Operator* $\odot$)**.** Given a Natural n, and a set of resource sets R, $n \odot R$ is the resource set $R' = \{n \odot r \mid r \in R\}$.

7.2. BAST internal node rules

null BAST rule. The rule states that the resource demand of a `null` BAST is the empty set (see Fig. 9).

Block node transition rule. The rule states that the resource consumption associated to a block is the sum of the resources required by each element in Z that forms the block.

IF_ELSE node transition rule. When the transition system encounters an *IF_ELSE* BAST node, the subtree n_{cond} for the condition, n_{true} for the true branch, and n_{false} for the false branch are extracted and analyzed. The resources C_{cond} required by the condition subtree are then summed to that of the true branch C_{true} by using the $\oplus$ operator, hence achieving the resource consumption C_1 of the true branch computation. The same holds for the resource consumption C_2 of the false branch. The final state C is the union of C_1 and C_2, i.e., the resource consumption of all possible computations is returned. Note that, if an *IF_ELSE* BAST node represents a Java `if` statement without `else` clause, the rule will have $n_{false} = \texttt{null}$, $C_{false} = \emptyset$ and $C_2 = C_{cond}$.

WHILE node transition rule. When the transition system encounters a *WHILE* BAST node, the subtree n_{cond} for the condition, and n_{body} for the body are extracted and analyzed. The *LoopAnnotation* function is called over the loop body to retrieve the upper bound t on the number of loop iterations (see Section 6.3). The resources C_{body}, required by the body subtree, are then multiplied by t (using the $\odot$ operator) in order to approximate the resource consumption of t execution of the loop. The condition subtree C_{cond}, instead, is multiplied by $t + 1$ since the condition in a while loop is always executed one time more with respect to the one of the body. Their sum is then returned as final configuration.

The scalar product $t \odot C_{body}$ represents the iteration. We point out that it might lead to a weak approximation. In fact, if branches are present in the body subtree, C_{body} will contain more then one resource set. By $\odot$-multiplying each one of these resource sets by t, we are implicitly assuming that in every loop iteration the same branching alternative is always chosen. This might not be the real case, of course. However, this approximation will surely contains the resource set representing the worst case, i.e., the repetition for t times of the branching with the "highest" resource demand. Since we want to perform a worst-case analysis, the provided approximation is safe.

7.3. BAST leaf node rules

Fall-Back Leaf Rule. This rule is applied when no other leaf rules of the transition system can be applied (see Fig. 10). The rule uses the function b (that represents the current resource consumption profile) in order to obtain the resource demand of the bytecode instruction `instr`. The resource set r, that describes the resource demand, is then returned. Note that, the execution of a single bytecode instruction can have only one computation, hence the returned resource demand is a singleton. The *Like* operator is used for checking if the instruction matches a given regular expression.

Resource Annotation Rule. This rule states that the resource consumption due to a leaf node labeled with a resource annotation, is the resource set specified in the annotation returned by the *ResourceAnnotation* function (see its definition in Section 6.3). Note that, no resource consumption is associated to the resource annotation instruction itself, in fact, differently from the *Fall-Back Leaf Rule*, the resource binding $b(\texttt{instr})$ is not considered.

Annotation Instructions Rule. This rule states that a leaf node labeled with an instruction related to annotation handling (i.e., $IsAnnotation(n) = \texttt{true}$), which is not a resource annotation, is not associated to any resource consumption.

The following rules detail the behavior of the transition system with respect to method calls. We recall that the JVM uses four instructions to handle method calls: *invokevirtual*, *invokeinterface*, *invokespecial* and *invokestatic*. All these instructions refer to a symbolic reference, which is a bundle of information that uniquely identifies a method. It includes the class/interface name, method name, and method descriptor. The class/interface name is that of the compile-time class/interface of the object invoking the method, as declared in the source code. The instructions `invokespecial` and

(L.1 Fall-Back Leaf Rule)

$$\frac{\begin{array}{c} \mathit{IsLeaf}(n) \quad \mathit{Label}(n) = \mathtt{instr} \\ \mathtt{instr}\ !\mathit{Like}(\texttt{"invoke*"}) \\ !\mathit{IsAnnotation}(n) \\ r = b(\mathtt{instr}) \end{array}}{\langle e, b, M, n\rangle \rightarrow_{ARA} \{r\}}$$

(L.2 Resource Annotation Rule)

$$\frac{\begin{array}{c} \mathit{IsLeaf}(n) \quad \mathit{Label}(n) = \mathtt{instr} \quad \mathit{IsAnnotation}(n) \\ \mathtt{instr}\ \mathit{Like}\ (\texttt{"invokestatic Annotation.resourceAnnotation*"}) \\ r = \mathit{ResourceAnnotation}(n) \end{array}}{\langle e, b, M, n\rangle \rightarrow_{ARA} \{r\}}$$

(L.3 Annotation Instructions Rule)

$$\frac{\begin{array}{c} \mathit{IsLeaf}(n) \quad \mathit{Label}(n) = \mathtt{instr} \quad \mathit{IsAnnotation}(n) \\ \mathtt{instr}\ !\mathit{Like}\ (\texttt{"invokestatic Annotation.resourceAnnotation*"}) \end{array}}{\langle e, b, M, n\rangle \rightarrow_{ARA} \emptyset}$$

(L.4 invokestatic and invokespecial Rule 1)

$$\frac{\begin{array}{c} \mathit{IsLeaf}(n) \quad \mathit{Label}(n) = \mathtt{instr} \\ \mathtt{instr} = \mathtt{invokestatic\ methId} \quad \mathit{OR} \quad \mathtt{instr} = \mathtt{invokespecial\ methId} \\ \mathtt{methId} \in M \\ r = b(\mathtt{instr}) \end{array}}{\langle e, b, M, n\rangle \rightarrow \{r\}}$$

(L.5 invokestatic and invokespecial Rule 2)

$$\frac{\begin{array}{c} \mathit{IsLeaf}(n) \quad \mathit{Label}(n) = \mathtt{instr} \\ \mathtt{instr} = \mathtt{invokestatic\ methId} \quad \mathit{OR} \quad \mathtt{instr} = \mathtt{invokespecial\ methId} \\ \mathtt{methId} \notin M \qquad \mathtt{instr}\ !\mathit{Like}\ (\texttt{"invokestatic Annotation*"}) \\ r = b(\mathtt{instr}) \\ \mathit{Bast}(e, \mathtt{methId}) = n' \qquad t = \mathit{CallAnnotation}(n) \\ \langle e, b, s, M \cup \mathtt{methId}, n'\rangle \rightarrow_{ARA} C \\ C' = \{r\} \oplus (t \odot C) \end{array}}{\langle e, b, M, n\rangle \rightarrow_{ARA} C'}$$

(L.6 invokevirtual Rule)

$$\frac{\begin{array}{c} \mathit{IsLeaf}(n) \quad \mathit{Label}(n) = \mathtt{instr} \\ \mathtt{instr} = \mathtt{invokevirtual\ methId} \\ r = b(\mathtt{instr}) \\ t = \mathit{CallAnnotation}(n) \\ \mathit{LookupOverrides}(\mathtt{e}, \mathtt{methId}) = S \\ Z = \{z_1 \ldots z_k\} = \{s_i | s_i \in S, s_i \notin M\} \\ \forall z_i \in Z \quad \mathit{Bast}(e, z_i) = n_i \\ \forall z_i \in Z \quad \langle e, b, s, M \cup z_i, n_i\rangle \rightarrow_{ARA} C_i \\ \forall i = 1 \ldots k \quad C_{i'} = \{r\} \oplus (t \odot C_i) \\ C = \bigcup_{i=1}^{k} C_{i'} \end{array}}{\langle e, b, M, n\rangle \rightarrow_{ARA} C}$$

Fig. 10. BAST leaf node transition system rules.

`invokestatic` are for static binding, i.e., the called method is exactly the one specified in the reference. The instructions `invokevirtual` and `invokeinterface` are for dynamic binding, i.e., the JVM will decide at runtime what method implementation to invoke based on the actual type of the object invoking the method. In fact, considering inheritance mechanisms and polymorphism:

- the actual method called by *invokevirtual* is an overriding method defined in one of the subclasses of the class contained in the *invokevirtual* reference.
- the actual method called by *invokeinterface* is a method defined in one of the implementing classes of the superinterface contained in the *invokeinterface* reference.

invokestatic and invokespecial Rule 1. This rule avoids the transition system to enter in an infinite loop when `invokestatic` or `invokespecial` (mutually) call recursive methods. In fact, if a method that has already been analyzed in the current computation is invoked (i.e., $\mathtt{methId} \in M$), the resource demand r is just the one associated to the execution of the call itself (i.e., $b(\mathtt{instr})$), without analyzing again the method (see next rule).

invokestatic and invokespecial Rule 2. If n is a leaf labeled with an `invokestatic` or `invokespecial` instruction on a method that has not been already analyzed in the current computation (i.e., $\mathtt{methId} \notin M$), the *Bast* function is used to return the BAST for the method (i.e., n'). Then, the resource demand C due to the execution of the method is computed starting from a configuration where the identifier of method is included in the set of the already analyzed methods ($M \cup \mathtt{methId}$). The *CallAnnotation* function is used to return the number of times t the method has to be executed. Recalling that call annotations statically handle (mutually) recursive methods calls (see Section 6.2), the consumption of t executions of the method is $t \odot C$. Finally, the result is summed to the resource demand r due to the execution of the method call. Note that, this rule is not applied if the invoked method belongs to the Annotation class, and the rule L.2 in Fig. 10 is applied instead. Moreover, for methods declared outside the application (e.g., external library or native ones), $\mathit{Bast}(e, \mathtt{methId}) = \mathtt{null}$, $C = \emptyset$ and consequently $C' = b(\mathtt{instr})$. In other words, the resource demand is the one specified by the resource profile, according to the dynamic analysis performed by the Chameleon Client.

It is worth to mention that, when a new object is created, the compiler explicitly adds to the generated bytecode the `invokespecial` calls to the constructors of the object class and superclasses. That is, the objects creation mechanism is well handled by this rule.

invokevirtual Rule. The main difference between this rule and the previous one concerns the usage of the *LookupOverrides* function. It is used to handle the polymorphism and inheritance mechanism previously described. Since we are in a static setting and we are looking for the worst case, ARA analyzes all the methods that can be potentially called at runtime by the `invokevirtual methId` instructions — i.e., all the methods that override `methId`. These are returned by the *LookupOverrides* function:

$$\mathit{LookupOverriding} : \mathit{Environment} \times \mathit{MethodID} \rightarrow \mathcal{P}(\mathit{MethodID})$$

Given the environment and a (fully qualified) method identifier `methId = ClassID.methodName(params...)`, the *LookupOverriding* function returns a set containing the identifiers of all the methods overriding the method `methodName(params...)` within all the subclasses of the class `ClassID`. This rule is analogous to rule `L.5` of Fig. 10. For each method s_i (returned by the *LookupOverriding* function), which has not been already analyzed ($s_i \notin M$), the resource demand C_i' is computed. Thus, C_i' represents the resource demand of those execution paths that choose s_i as overridden method implementation. The union of all the $C_{i'}$ is finally returned, hence representing all possible execution paths.

7.4. ARA properties

Proposition 1 (*Completeness of the Analysis*)**.** *ARA examines all the possible worst case execution paths of each method for collecting resource information.*

Completeness is achieved by construction since for each BAST node, that produces a branching point, parallel analyses (one for each child) are performed. The resource demands of all the branches are then collected and their union is returned as the final state of the transition system.

We recall that we assume upper bounds for loops and recursive calls. For this reason, even though a BAST encodes all the possible execution paths of the methods, our analysis does not explore all of them in the case of loops and recursive calls. However, as detailed in Section 7.2 (WHILE Node transition rule), the analysis ARA performs is complete with respect to the worst-case execution paths.

Proposition 2 (*Monotonicity*)**.** *The ARA transition relation is monotonic with respect to the construction of the resource sets.*

Each rule of ARA increases the current resource set by using the $\oplus$ and $\odot$ operators. Basically, these operators increase, at each transition, either the number of elements in the current resource set or the value of the resource instances in it contained. Therefore, at each transition, there is no way to "reduce" the amount and the number of the resources. Moreover, the resource sets that are operands of the $\oplus$ and $\odot$ operators, are always contained into the resulting sets. This implies that, at each step, the resource sets are always comparable.

The completeness and the monotonicity properties guarantee that the "highest-demanding" execution path is eventually found. The compatibility of all these resource sets against the resource supply (provided by the execution environment) guarantees that, with respect to ARA and to the notion of compatibility, the analyzed application will be correctly executed.

8. A small example

This section presents an example to show how the ARA performs the analysis of the `print(String)` method bytecode of the simple application shown in Fig. 7. The example is small but highlights many interesting features of the analysis.

For this application, the environment function e retrieves the information about the class identifiers `Item`, `Book`, `DVD` and `Library`; `PrintServices` is an external class. The analysis is intended to estimate the consumption of the resources defined in Fig. 4. The resource binding function b is derived from the resource consumption profile of Fig. 5. The BAST for the `print(String)` method is shown in Fig. 8, where each node reports the resource consumption of its whole subtree.

For the sake of space, Fig. 8 does not show the resource demand for some of the nodes. Actually, these nodes are implicitly associated to $\{\{CPU(1), Energy(1)\}\}$ that is the result of the application of the Rule `L.1`, using the binding corresponding to the $*$ instruction pattern (i.e., $b(*) = \{CPU(1), Energy(1)\}$). On the other hand, the rule `L.1` is applied to leaf 1 by selecting the binding $b(\texttt{istore\_1}) = \{CPU(2), Energy(1)\}$.

The leaves 12 and 10 represent the call to the `LoopAnnotation` method and the *push* of its formal parameter onto the operand stack, respectively. Hence, the *IsAnnotation* function applied to them returns *true*, the rule `L.3` is applied, and an empty resource demand is returned. The same holds for the leaf 36 that instantiates the formal parameter of the `resourceAnnotation` method call of leaf 38. For the latter, the rule `L.2` is applied and the resource demand specified by the parameter of the `resourceAnnotation` method is returned (i.e., $\{\{CPU(20), Bluetooth(true)\}\}$).

On the node 32, which corresponds to an `invokevirtual` instruction, the rule `L.6` is applied. Since there is no `callAnnotation` instruction, $t = 1$. $LookupOverriding(e, Item.getData()) = S = \{Item.getData(), Book.getData(), DVD.getData()\} = Z$, as none of methods is in M.

Now, since $Bast(e, Item.getData())$=*null*, the resource demand of the method *Item.getData*() is empty and rule I.1 is applied. The resource demands of *Book.getData*() and *DVD.get Data* methods are $\{\{CPU(24), Energy(12)\}\}$ and $\{\{CPU(2), Energy(2)\}\}$, respectively. Each one of them is summed with the resource demand $r = \{CPU(4), Energy(1)\}$ of the `invokevirtual` instruction and their union is returned as the resource demand of the node 32, i.e., $\{\{CPU(28), Energy(13)\}, \{CPU(6), Energy(3)\}\}$.

We now describe how ARA deals with internal nodes by starting from the root. Since the latter is a BLOCK node, rule I.2 is applied and the transition system is recursively called on the children (i.e., a BLOCK node, a WHILE node, and another BLOCK). The leftmost BLOCK has two leaf children and the outcome of their analysis is returned back to rule I.2. This allows for assigning $\{\{CPU(3), Energy(2)\}\}$ as the final configuration of the leftmost BLOCK. This set is then returned backwardly up to the root, and is combined with the outcomes the other two children. The final configuration of the root is the set on the right of the root node in Fig. 8. This is actually the set containing all the resource sets computed by following all the worst case computational paths. For example, the resource set $\{CPU(6808), Energy(3007), Bluetooth(true)\}$ corresponds to the execution path traversing the `trueBranch` of the IF_ELSE node and calling the `Book.getData()` function on node 32.

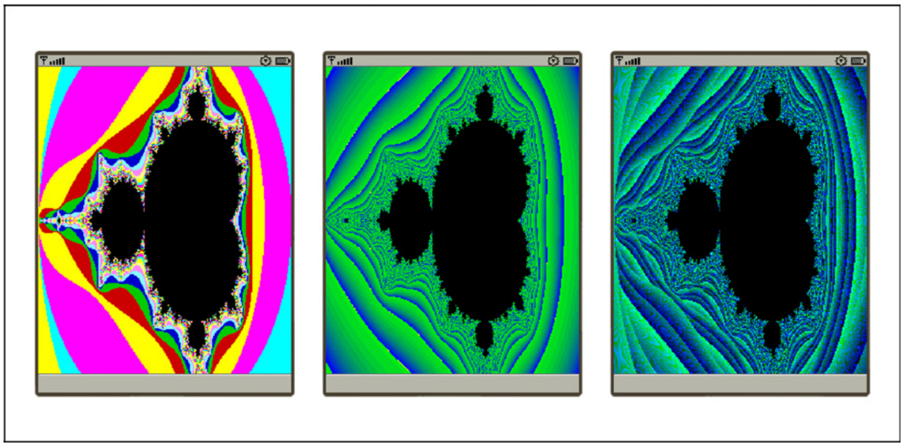

Fig. 11. Alternative coloring algorithms for the Mandelbrot fractal.

As already mentioned, when checking the compatibility of the resource demanded by the application (with respect to the resources provided by an execution environment) we need to check the compatibility for each single resource set contained in the final configuration returned by ARA. In other words, we need to check that all the resource demands for all the worst case execution paths are compatible with the resources provided by the execution environment.

9. Validation

In this section we validate our approach by using the adaptable Mandelbrot fractal case study as briefly described below. More details on the Mandelbrot fractal can be found in [13] and the complete adaptable code for it can be downloaded from.[10] In order to investigate if our hybrid analysis is sufficiently accurate, and hence if CHAMELEON is reliable, we have defined a validation protocol and we have implemented a validation environment to automatize the protocol.

Validation case study. As already anticipated in Section 3, CHAMELEON has been used to implement, automatically deliver and deploy via OTA[11] an adaptable MIDlet that is able to suitably display a Mandelbrot fractal [19] filling the screen of resource-constrained devices.

The drawing of a fractal can be a simple or a complex task. In the most cases, the more complex is the construction method, the more beautiful is the produced fractal, and the more resources are required to the device (CPU, screen characteristics, memory, ...). In order to display the finest possible fractal while accounting for the characteristics of the device, a context-aware adaptable application is auspiced. Typically, the construction of a Mandelbrot fractal is done iterating the fractal formula to determine if a canvas pixel is in the Mandelbrot set. Then, each pixel is colored according to some coloring algorithm.

From the generic code [13] of the adaptable Mandelbrot fractal application CHAMELEON derives 10 different alternatives, which use different building and coloring methods. As an example, see the outputs of three possible alternatives in Fig. 11. If the device is equipped with a very slow CPU, which is not able (or takes too long time) to compute a fractal locally, CHAMELEON provides an alternative that (if a network connection is available) connects to a remote server, downloads a fractal adapted to the device screen characteristics (sent by the device), and displays it.

We now briefly summarize how the CHAMELEON Programming Model has been used to implement the adaptable application. The excerpt of the (adaptable) class diagram in Fig. 12 shows two adaptable classes, `MandelFractalMIDlet` and `MandelCanvas`, and a standard Java class `SocketConnection`. The latter is a thread that provides a set of methods to connect to a remote server, retrieves a customized fractal image, and displays it. The adaptable MIDlet `MandelFractalMIDlet` class declares the adaptable method `startApp()`. Two possible alternatives are provided for this method. The `Local Construction` alternative uses `MandelCanvas` (as adapted by the adaptable alternative `Local Builder`) to build and display the fractal. The `RemoteConstruction` alternative uses `SocketConnection` to download a customized fractal built on a remote server, and then uses `MandelCanvas` (as adapted by the alternative `Remote Getter`) to display the fractal.

More specifically, the adaptable alternative `Local Builder` has six alternatives. The `MatrixCanvas` alternative, stores the fractal in a matrix and displays it all at a time when the process ends. `ArrayCanvas` displays a fractal column at a time storing it into an array during the building phase. `DirectCanvas` draws and displays a fractal pixel at a time. `MatrixCanvas`, `ArrayCanvas` and `DirectCanvas` implement different ways of building the fractal, hence requiring different resources and providing different QoS. In particular, they require, in the listed order, a decreasing amount of

[10] http://di.univaq.it/chameleon/.

[11] http://developers.sun.com/mobility/midp/articles/ota/.

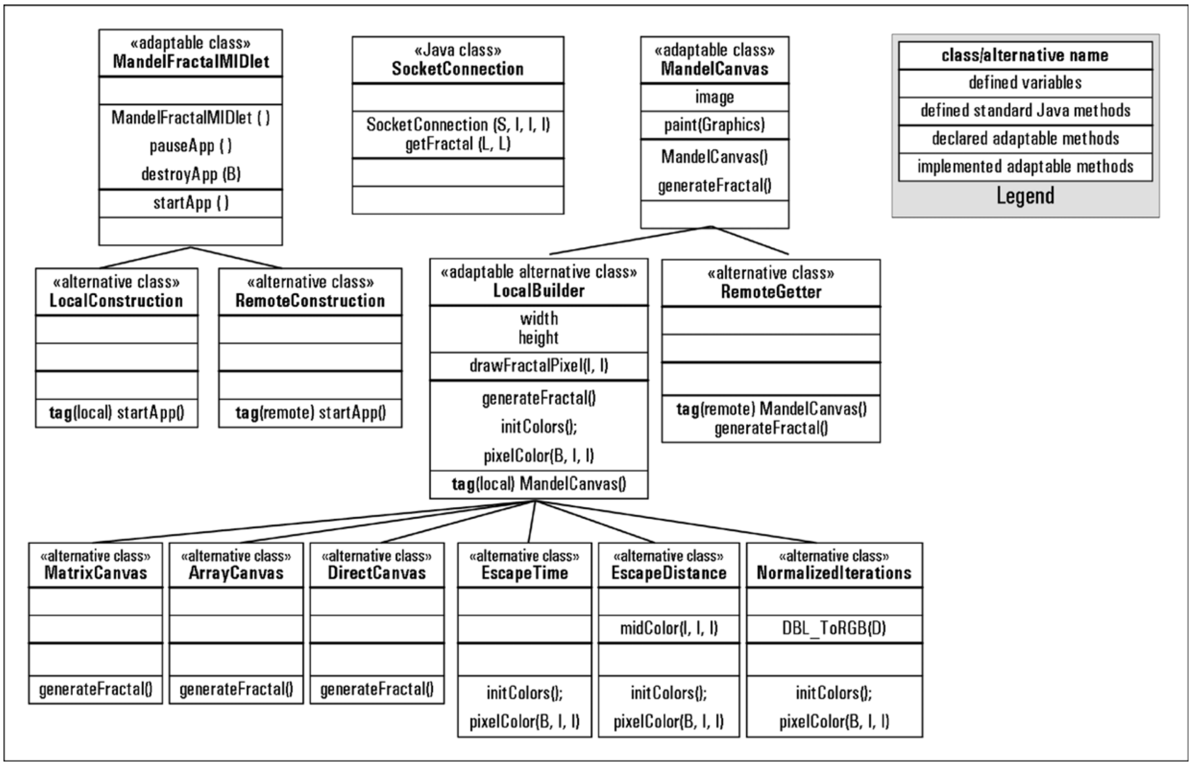

Fig. 12. Class diagram of the adaptable Mandelbrot fractal MIDlet.

memory to store the fractal and an increasing time to complete the task, mainly due to the screen refresh frequency.[12] As a consequence, the user perceives different QoSs due to the different ''promptness'' of the building methods. In the worst case (i.e., `MatrixCanvas`), the user has to contemplate a ''black screen'' until the whole drawing process ends. The three other alternatives of the `LocalBuilder` class, namely, `EscapeTime`, `EscapeDistance`, `NormalizedIterations`, provide different adaptations for the methods `initColors` and `pixelColor`. These alternatives produce, in the listed order, images of growing beauty (see Fig. 11), but require increasing resource demands (CPU power, number of screen colors, ...).

The tags `remote` and `local` attached to the methods are declared to be conflicting tags, and hence, e.g., the tagged methods `RemoteConstruction.startUp()` and `LocalBuilder.MandelCanvas()` cannot coexist within the same application alternative.

Validation protocol. The validation aims at verifying if the selected and deployed application alternative is actually the ''best'' for the target execution environment. To this end, our validation protocol (i) executes each alternative and determines which is the best one on the field (*actual best alternative*); (ii) analyzes each alternative using our hybrid analysis and determines which is the best one according to the framework (Chameleon *best alternative*); (iii) compares the two results. The best alternative is determined according to a notion of *Goodness*. Briefly, the *Goodness* expresses ''how good'' is an application alternative as a function of the resource consumption, the resource supply offered by the target execution environment, and a resource priority defined by the user. The formal definition of the *Goodness* function can be found in [48]. Clearly, for the validation to be correct, the *Goodness* function used to rank the alternatives on the field is exactly the same used by the Chameleon Customizer to select the application alternative to be deployed. The resource demand of the *actual best alternative* is the one actually computed in the field; whereas, the resource demand of the Chameleon *best alternative* is the estimated one. This allows us to validate if the estimated resource demand of an alternative is reliable, not in an absolute way, but in a *relative* way with respect to the demands of all the other alternatives.

Validation environment. Executing the validation protocol can be a time consuming and difficult activity to be performed manually. For this reason we created a validation environment that automatizes the validation process.

• *Device emulator.* The device emulator used in the validation environment is the one provided with the SUN Java Wireless Toolkit for CLDC. The toolkit includes the emulation environments, performance optimization and tuning features. We chose

[12] Many small devices have a very high screen refresh latency.

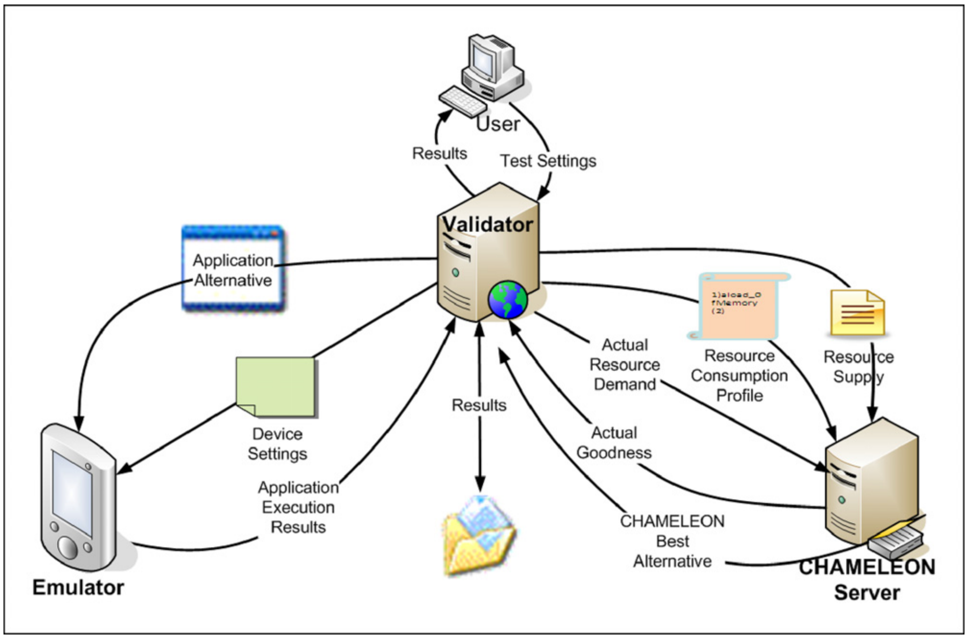

Fig. 13. Validation environment architecture.

that emulator because it is a standard for Java MIDlets, permits to adjust the resource oriented characteristics and provides OTA capabilities. In [48] we detail the resource that we have tuned for the emulation (e.g., Graphics primitives latency, Display refresh types, VM speed emulation, Network throughput emulation, Storage size).

• *Validation environment architecture and workflow.* The validation environment has been implemented on a client–server architecture as a web application (see Fig. 13). The workflow of the validation process as implemented by the validator is the following:

1. Upon user invocation, the validator repeats the specified number of times the following steps:
   (a) Randomly set the device resource characteristics.
   (b) Compute the resource supply and the resource consumption profile.
   (c) For each application alternative, deploys the alternative on the emulator via OTA, execute it, and compute the actual goodness.
   (d) Determine the actual best alternative.
   (e) Invoke CHAMELEON to determine the CHAMELEON best alternative.
   (f) Compare the results.
2. The validator displays the validation results, along with statistical information (e.g., mean and max number of execution failures in a test cycle, elapsed time) and detailed information of each single validation cycle (device skin, device characteristics, emulator outputs for each application alternative, CHAMELEON outputs).

Validation results. The Mandelbrot case study considers several boolean, enumerated and natural resources. Even though the adaptable fractal is a relatively small application, it encodes the main factors that can influence the accuracy of analysis, i.e., the resource consumption profile, resource annotations and loop annotations. The validation environment has been executed on 5 desktop PCs in parallel (at least Intel core due processor and 1 GB of RAM) and 10.000 test cycles have been run.

The test results shown in Fig. 14 reveal that the selected CHAMELEON *best alternative* is the 1st *actual best alternative* in 80% of the cases, according to the order imposed by the *Goodness* rank. At a first glance, one may think that this is a bad result, but the perspective changes if we consider the severity of the errors. In fact, in the 17.7% of the cases the severity of the error is low, which means that the CHAMELEON best alternative is either the 2nd or the 3rd *actual best alternative*. Only in the 2.3% of the cases the severity is from medium to high, which means that the CHAMELEON best alternative is between the 4th and the 10th *actual best alternative*. As a result, it can be argued that the analysis has a satisfactory accuracy. This is even more true if we consider that, when trying to execute the 10 alternatives on the emulator, we have experienced an average of 68% of application failures during all the test cycles. In other words, randomly downloading an alternative we have a 0.68 probability that the alternative fails. The detailed validation results reported in [48] reveal that using CHAMELEON this risk is reduced to 0.013.

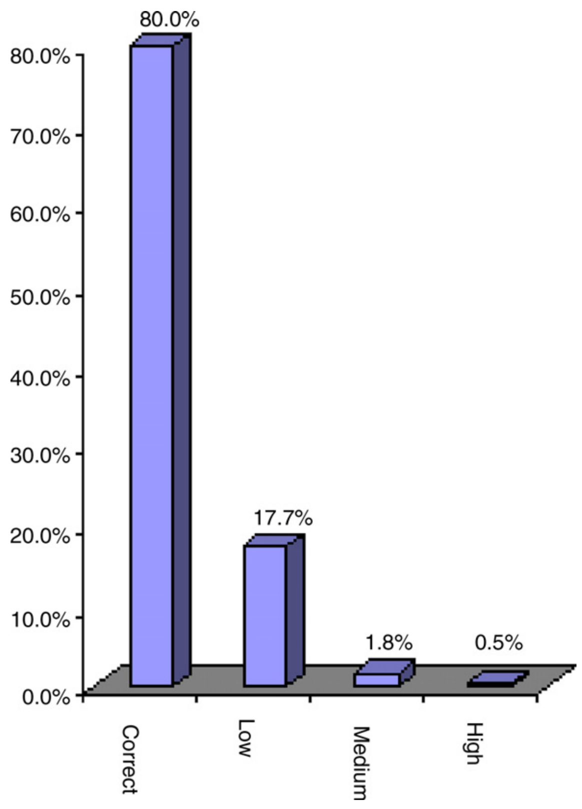

Fig. 14. Results of the validation.

10. Discussion

In this section we discuss different aspects of Chameleon trying to highlight some peculiarities together with pros and cons.

In order to understand the applicability of Chameleon to a larger scope, we discuss the portability of the approach on some of the well known platforms in the market: (i) Devices powered with Symbian Operating System (OS) can be programmed using different languages, among which Java Micro Edition. Applications are developed and deployed by using standard techniques and tools, such as the Sun Java Wireless Toolkit. Chameleon can be used as it is; (ii) The Google's Android OS uses the Dalvik virtual machine to run applications that are commonly written in a dialect of Java and compiled to the Dalvik bytecode. Even though different, the Dalvik bytecode and the J2ME bytecode present many similarities that make the portability of Chameleon straightforward; (iii) Concerning proprietary operating systems, such as the iPhone OS and the BlackBerry OS, applications are developed by following a close philosophy that does not require the flexibility of the Chameleon development process. In fact, for such OSs the exploitation of Chameleon might make no sense to address heterogeneity and resource limitation of the different execution environments. These issues can be a priori addressed since applications run on a priori known execution environments for which resource constraints can be tackled by using explicit hard-coded conditionals.

Nowadays, the use of (client-side) scripts in mobile platforms is catching a growing interest (e.g., rich web applications using JavaScript). Scripts are often interpreted from source code or bytecode. The current implementation of our framework does not consider the resource analysis of script-based languages. However, its extension to enable the computation of the resource consumption of scripts is straightforward. Scripts might be specified as adaptable code and a set of script alternatives can be derived from it. Thus, the overall framework can be used to deliver script code (e.g., within a HTML page) that adapts to the resources offered by the hosting device. To this end, script alternatives can be analyzed by using our approach, hence still exploiting its hybrid nature.

As far as dynamic variation of consumable resources is concerned, in [11] we propose a mechanism that, exploiting ad-hoc methods for saving and restoring the (current) application state, enables services' evolution to handle context changes by dynamically un-deploying the no longer apt alternative and subsequently (re-)deploying a new alternative. In this situation, Chameleon considers also the resource consumption due to the stopping/updating/restarting mechanism relying on the costs estimated by ARA for the save and restore methods. Clearly, this approach is time and resource consuming. Moreover, it does not allow for seamlessly reconfiguration steps. In this direction we are investigating the notion of quiescence state [49,50], and possible extensions to the mobile world of existing approaches, such as Javeleon[13] and Javassist,[14] to enable dynamic code changes in the running application, without losing any of the application state and without interrupting it.

Concerning the delivery and deployment of application alternatives, the availability of the CHAMELEON server is required for the duration of the downloading. OTA is used for delivering the application and an XML-based proprietary protocol is used for CHAMELEON-specific client/server interactions (e.g., resource profile and resource supply upload). However, when evolution has to be supported, our approach requires the continuous availability of the connection to the CHAMELEON server. To limit this need, the a priori knowledge of the variability of specific resources in a given context can be of help (e.g., the availability of both a WiFi connection that is battery consuming and of a GPRS connection that is more battery saving). This knowledge, together with the possibility of performing dynamic code changes, would enable the possibility of a priori delivering (more than one) "code alternatives" that are necessary to handle the variability of the specifically considered resources.

By referring to Section 3, we recall that the client-side component is a very light-weight component and its tasks can be "safely" executed on limited devices. On the contrary, ARA might have a high computational complexity but it is executed on powerful server machines that (usually) does not suffer resource limitations. Concerning the computational complexity of ARA, it is worth to note that the resource demand depends on the device characteristics, and hence it can be computed at deployment time, only after the resource consumption profile has been provided by the device. Thus, since the analysis process is computationally expensive, serving a device request may take relatively long time. However, exploiting the fact that the analysis performed by ARA is "context free" (i.e., it considers all the bytecode instructions in their isolation [36]), the analysis phase is separated from the actual resource demand calculation. By doing so, the analysis is entirely performed off-line on the CHAMELEON server, and its result is a description of the application structure that is parametric with respect to information provided by the device through the resource consumption profile. The resource demand will be then calculated upon client request, at deployment time, by simply instantiating the parameters in the extracted description.

Concerning our programming model [13], it is worth to note that it shares similarities with traditional features provided by the Java language. For example, alternatives are similar to subclasses. However, note that methods declared within alternatives are the basic unit for adaptation. As detailed in [13], the final adapted (standard Java) classes will be constructed by choosing the methods provided by some alternatives. This means that the actual implementation of different methods can be chosen from different alternatives. This gives more flexibility than subclassing. In fact, in order to obtain the same result with the subclassing mechanism the developer has to define a subclass for every possible combination of methods, constituting an application alternative. This would lead to a larger number of declared subclasses (with respect to the number of alternative classes) with a lot of redundancy, and related maintenance issues. However, to be closer to standard Java, as future work we plan to use Java annotations to express the specific semantics of our adaptation specification mechanisms. Indeed, as already anticipated in Section 6.2, we are waiting for the possibility of using standard Java annotations also for specifying CHAMELEON annotations on loops, blocks, and simple statements.

The programming model can also be related to Software Product Lines (SPLs). In particular, our generic code can be seen as a features model that represents: features through (adaptable) classes, commonalities through standard classes (core code), variation points through adaptable methods, and variation point refinements through adaptation alternatives. Furthermore, our multiple method definitions in alternative classes and our tag mechanism express *require* and *mutually exclusive* relationships between adaptable methods implementations, respectively. Moreover, our preprocessor, checking the fulfillment of the adaptation policy constraints, implements a basic form of validation technique. However, within this line of research, our goal is to propose a context-oriented programming approach to the development of context-adaptable applications, which is closer to the common practice of software applications coding. Thus, the main difference between our programming model and feature model is the abstraction level. In-fact, feature models are more user-oriented (or marketing-oriented) representations of the problem space and express end-user understandable concepts. Our programming model allows for defining code-level models and it is used by developers to express how the application classes can be adapted. For what concerns dynamicity aspects, our approach and Dynamic SPLs also share the same underlying concepts. Again, the main difference relies on the abstraction level of the adaptable classes versus the features, as units of adaptation.

The CHAMELEON framework has been developed in and for *Java*. Although adaptable applications can be coded by using a wide range of other languages and Java is a resource consuming programming language, we opted for Java since it is widely used to power many mobile devices. Moreover, it has many characteristics that fit our purposes: It is *object oriented*, and provides useful abstraction for modeling adaptable applications; It is *bytecode compiled*, and allows us to abstract by physical details of the underlying hardware platform; It is *widely used* in different heterogeneous contexts, hence providing an effective environment for experimentation.

The use of CHAMELEON requires some extra work to developers. In particular, they have to specify some configuration files, such as the resource definition schema and the adaptable application descriptor. The effort to define these configuration files is limited and is assisted by the DE. Moreover, developers have to specify additional information to instruct ARA in order to perform a better analysis. In particular, she has to specify annotations and to implement ad hoc methods of the CHAMELEON

13 http://javeleon.org/.
14 http://www.csg.is.titech.ac.jp/~chiba/javassist/.

client to be used by the dynamic analysis in order to compute the resource consumption profile for external methods (see Section 6.1). The effort to do this job can be hard and depends on both the kind of application and the degree of precision the developer wants to reach. Thus, it is clear that the more the resource consumption profile and the annotations are accurate, the more the resulting resource demand will be precise.

11. Conclusion and future work

In this paper we have presented a hybrid analysis technique for characterizing the resource consumption of (adaptable) Java programs. The core of the approach is a Resource Model that characterizes the notion of resources and a transitional semantics of the bytecode, the Abstract Resource Analyzer, parametric with a Resource Consumption Profile. Combining static and dynamic analysis, the technique allows for a relative comparison of Java programs with respect to their use of resources in the same execution context. This makes it possible to choose among multiple alternatives the ones that fit better with the available resource provisions. The hybrid analyzer is part of a bigger framework, namely CHAMELEON, that supports the development and the deployment of adaptable Java applications for heterogeneous and resource-constrained devices. CHAMELEON has been fully implemented in and for Java and it is currently being used to experiment and analyze adaptable Java applications.

The approach has been used to implement a number of medium-sized case studies. In this paper, it has been validated by using the adaptable Mandelbrot fractal case study. According to the validation results, we can reasonably argue that the accuracy of CHAMELEON, in its experimental version, is adequate for its purposes.

As future work, we will investigate if, for some resources, annotations and specific control flows, it is worth to adopt more expensive approaches in favor of more tight bounds, and hence a more precise relative comparison. We are currently extending the annotation mechanism to consider parametric values related to resources (e.g., screen size) for loop and call annotations. Then, this parameters can be instantiated at analysis time with the actual values of the resources offered by the target execution environment. This will permit a tighter estimation of resource consumptions.

We are also refining the relationships among different resources when specifying resource consumption profiles. Towards this direction, an effort on ontology specifications must be made in order to establish a common vocabulary among resource types. For instance, it is desirable to be able to match resource types, establish relations among them, and predicate about a common set of related resource types.

Finally, more expressive bindings might be introduced in the resource consumption profile in order to better model the execution environment with respect to resources. For example, a new binding mechanism might be introduced for matching resource consumption not only with patterns of single bytecode instructions, but also with patterns for recurrent blocks of instructions.

Acknowledgements

We acknowledge the anonymous reviewers for the competent comments and critic suggestions that allowed us to improve this work.

References

[1] E. Gamma, R. Helm, R. Johnson, Design Patterns. Elements of Reusable Object-Oriented Software, in: Addison-Wesley Professional Computing Series, Addison-Wesley, 1995.
[2] E.-N. Volanschi, C. Consel, G. Muller, C. Cowan, Declarative specialization of object-oriented programs, in: OOPSLA'97: Object-Oriented Programming, Systems, Languages and Applications, 1997, pp. 286–300.
[3] C. Cowan, A. Black, C. Krasic, C. Pu, J. Walpole, C. Consel, E. Volanschi, Specialization classes: an object framework for specialization, in: 5th International Workshop on Object-Orientation in Operating Systems, 1996.
[4] R. Hirschfeld, P. Costanza, O. Nierstrasz, Context-oriented programming, Journal of Object Technology 7 (3) (2008) 125–151.
[5] S. Kochan, Programming in Objective-C, Sams, 2003.
[6] R. Keays, A. Rakotonirainy, Context-oriented programming, in: MobiDe'03: 3rd ACM International Workshop on Data Engineering for Wireless and Mobile Access, 2003, pp. 9–16.
[7] K. Henricksen, J. Indulska, Developing context-aware pervasive computing applications: models and approach, Pervasive and Mobile Computing 2 (1) (2006) 37–64.
[8] É Tanter, K. Gybels, M. Denker, A. Bergel, Context-aware aspects, in: Software Composition, in: LNCS, vol. 4089, 2006, pp. 227–242.
[9] A. Villazón, W. Binder, D. Ansaloni, P. Moret, Advanced runtime adaptation for java, in: GPCE'09: 8th International Conference on Generative Programming and Component Engineering, 2009, pp. 85–94.
[10] A. Villazón, W. Binder, D. Ansaloni, P. Moret, Hotwave: creating adaptive tools with dynamic aspect-oriented programming in java, in: GPCE'09: 8th International Conference on Generative Programming and Component Engineering, 2009, pp. 95–98.
[11] M. Autili, P.D. Benedetto, P. Inverardi, D.A. Tamburri, Towards self-evolving context-aware services, in: DisCoTec '08 (CAMPUS'08): Context-aware Adaptation Mechanisms for Pervasive and Ubiquitous Services, 2008.
[12] M. Autili, P.D. Benedetto, P. Inverardi, Context-aware adaptive services: the plastic approach, in: FASE'09: 12th Fundamental Approaches to Software Engineering, in: LNCS, Springer, 2009, pp. 124–139.
[13] M. Autili, P. Di Benedetto, P. Inverardi, A programming model for adaptable java applications, in: PPPJ'10: 8th International Conference on the Principles and Practice of Programming in Java, 2010, pp. 119–128.
[14] R. Wilhelm, S. Altmeyer, C. Burguière, D. Grund, J. Herter, J. Reineke, B. Wachter, S. Wilhelm, Static timing analysis for hard real-time systems, in: VMCAI'10: 11th Verification, Model Checking, and Abstract Interpretation, 2010, pp. 3–22.
[15] G.C. Necula, Proof-carrying code, in: POPL'97: 24th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, ACM, 1997, pp. 106–119.

[16] P. Cousot, R. Cousot, Static determination of dynamic properties of programs, in: 2nd International Symposium on Programming, Paris, France, 1976.
[17] P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: POPL'77: 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1977, pp. 238–252.
[18] F. Mancinelli, P. Inverardi, Quantitative resource-oriented analysis of java (adaptable) applications, in: WOSP'07: 6th International Workshop on Software and Performance, 2007, pp. 15–25.
[19] B.B. Mandelbrot, The Fractal Geometry of Nature, W. H. Freeman, 1982, http://www.worldcat.org/isbn/0716711869.
[20] D. Aspinall, K. MacKenzie, Mobile resource guarantees and policies., in: CASSIS'06: 3rd International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, 2006, pp. 16–36.
[21] G. Barthe, Mobius, securing the next generation of java-based global computers, ERCIM News (2005).
[22] E. Albert, P. Arenas, S. Genaim, G. Puebla, D. Zanardini, Experiments in cost analysis of java bytecode, Electronic Notes Theoretical Computur Science 190 (1) (2007) 67–83.
[23] E. Albert, S. Genaim, M. Gomez-Zamalloa, Heap space analysis for java bytecode, in: ISMM'07: 6th International Symposium on Memory Management, ACM, 2007, pp. 105–116.
[24] D. Aspinall, R. Atkey, K. MacKenzie, D. Sannella, Symbolic and analytic techniques for resource analysis of java bytecode, in: Trustworthly Global Computing, in: LNCS, vol. 6084, 2010, pp. 1–22.
[25] S. Gulwani, K.K. Mehra, T. Chilimbi, SPEED: precise and efficient static estimation of program computational complexity, in: POPL'09: 36th Symposium on Principles of Programming Languages, 2009, pp. 127–139.
[26] W.-N. Chin, H.H. Nguyen, C. Popeea, S. Qin, Analysing memory resource bounds for low-level programs, in: ISMM'08: 7th International Symposium on Memory Management, 2008, pp. 151–160.
[27] V. Braberman, F. Fernández, D. Garbervetsky, S. Yovine, Parametric prediction of heap memory requirements, in: ISMM'08: 7th International Symposium on Memory Management, 2008, pp. 141–150.
[28] C. Seo, S. Malek, N. Medvidovic, An energy consumption framework for distributed java-based systems, in: ASE'07: 22th IEEE/ACM International Conference on Automated Software Engineering, ACM, 2007, pp. 421–424.
[29] M. Moeller, B. Callahan, V. Gucer, J. Hollis, S. Weber, Introducing Tivoli Distributed Monitoring Workbench 4.1, IBM Redbooks, 2002.
[30] O. Sokolsky, Resource modeling for embedded systems design., in: WSTFEUS'04: 2nd IEEE Workshop on Software Technologies for Embedded and Ubiquitous Computing Systems, 2004, pp. 99–103.
[31] W. Visser, K. Havelund, G. Brat, S. Park, F. Lerda, Model checking programs, Automated Software Engineering Journal 10 (2) (2003) 203–232.
[32] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, P. Stenström, The worst-case execution-time problem—overview of methods and survey of tools, Transactions on Embedded Computing Systems 7 (3) (2008) 1–53.
[33] S. Altmeyer, C. Hümbert, B. Lisper, R. Wilhelm, Parametric Timing Analyis for Complex Architectures, in: RTCSA'08: 14th International Conference on Embedded and Real-Time Computing Systems and Applications, 2008, pp. 367–376.
[34] S. González, N. Cardozo, K. Mens, A. Cádiz, J.-C. Libbrecht, J. Goffaux, Subjective-c: Bringing context to mobile platform programming, in: Proceedings of the Third International Conference on Software Language Engineering, in: SLE'10, 2011, pp. 246–265.
[35] S. González, K. Mens, A. Cádiz, Context-oriented programming with the ambient object system, Journal of Universal Computer Science 14 (20) (2008) 3307–3332.
[36] M. Autili, P.D. Benedetto, P. Inverardi, F. Mancinelli, A resource-oriented static analysis approach to adaptable java applications, in: COMPSAC'08: 32nd International Computer Software and Applications Conference, 2008.
[37] A. Ermedahl, J. Gustafsson, Deriving annotations for tight calculation of execution time, in: Euro-Par'97: 3rd International Euro-Par Conference on Parallel Processing, Springer-Verlag, 1997, pp. 1298–1307.
[38] F. Curatelli, L. Mangeruca, A method for computing the number of iterations in data dependent loops, Real-Time Systems 32 (1–2) (2006) 73–104.
[39] C. Healy, M. Sjödin, V. Rustagi, D. Whalley, R.V. Engelen, Supporting timing analysis by automatic bounding of loopiterations, Real-Time Systems 18 (2–3) (2000) 129–156.
[40] M.S.C. Heal, D. Whalley, Bounding loop iterations for timing analysis, in: RTAS'98: 4th IEEE Real-Time Technology and Applications Symposium, IEEE, 1998, p. 12.
[41] M. Eaddy, A. Aho, Statement annotations for fine-grained advising, in: In ECOOP Workshop on Reflection, AOP, and Meta-Data for Software Evolution, 2006.
[42] W. Cazzola, @java, a java annotation extension, 2011, http://cazzola.dico.unimi.it/atjava.html.
[43] M.D. Ernst, Type annotations specification (jsr 308), 2009, http://types.cs.washington.edu/jsr308/.
[44] C. Cifuentes, Reverse compilation technique, Master's Thesis, Queensland University of Technology, 1994.
[45] B.S. Baker, An algorithm for structuring flowgraphs, Journal of the ACM 24 (1) (1977) 98–120.
[46] J. Miecznikowski, L. Hendren, Decompiling java using staged encapsulation, in: WCRE'01: 8th Working Conference on Reverse Engineering, IEEE, 2001, p. 368.
[47] Sable research group, Soot: a java optimization framework, http://www.sable.mcgill.ca/soot/.
[48] P. Di Benedetto, A framework for context aware adaptable software applications and services, Ph.D. Thesis, University of L'Aquila, Computer Science Department, 2010, http://www.di.univaq.it/chameleon/output/download.php?fileID=20.
[49] J. Kramer, J. Magee, The evolving philosophers problem: Dynamic change management, IEEE Transactions on Software Engineering 16 (11) (1990) 1293–1306.
[50] Y. Vandewoude, P. Ebraert, Y. Berbers, T. D'Hondt, Tranquility: a low disruptive alternative to quiescence for ensuring safe dynamic updates, IEEE Transactions on Software Engineering 33 (12) (2007) 856–868.