A Resource-Oriented Static Analysis Approach to Adaptable Java Applications

Conference Paper · January 2008

DOI: 10.1109/COMPSAC.2008.157 · Source: DBLP

4 authors, including:

Marco Autili
Università degli Studi dell'Aquila
118 PUBLICATIONS **1,538** CITATIONS
SEE PROFILE

Paolo Di Benedetto
Università degli Studi dell'Aquila
7 PUBLICATIONS **95** CITATIONS
SEE PROFILE

Paola Inverardi
Università degli Studi dell'Aquila
333 PUBLICATIONS **9,139** CITATIONS
SEE PROFILE

A Resource-oriented Static Analysis Approach to Adaptable Java Applications

M. Autili, P. Di Benedetto, P. Inverardi
Università dell'Aquila - Dipartimento di Informatica, Italy
{autili, paolo.dibenedetto, inverard}@di.univaq.it

Fabio Mancinelli
INRIA - Rocquencourt, France
fabio.mancinelli@inria.fr

Abstract

In this paper we present a static analysis approach for inspecting Java programs and characterizing them with respect to their resource consumption in a given execution environment. We target, in particular, resource constrained devices that are characterized by resource scarcity and limited computational power. The focus of this paper is on a parametrical Abstract Resource Analyzer that performs the actual analysis and is supported by a Resource Model that allows us to abstract application behavior in terms of its resource needs. The presented components are integrated in a larger framework that provides a complete system for reasoning and adapting Java programs with respect to heterogeneous contexts.

1 Introduction

The wide spread of small computing devices and the introduction of new communication infrastructures are rapidly changing the ways we interact and use the technology to perform everyday tasks. Today's platforms are typically made of mobile resource-constrained devices, characterized by their *heterogeneity* and limitation. Thus, applications must cope with resource scarcity and with the inherent faulty and heterogeneous nature of this environment.

In this scenario, *resource awareness* and *adaptability* have become two important aspects that must be considered when developing applications. For our purposes, (*i*) *resource awareness* identifies the capability of being aware of the resources offered by an execution environment, in order to decide whether or not it is suited to receive and execute the application; (*ii*) *adaptability* identifies the ability of "tailoring" the application for making it comply with the current resource conditions.

In this paper we present a static analysis approach to the inspection of Java programs and their characterization w.r.t. their resource consumption in a given execution environment. The approach makes use of a *Resource Model* (RM) that provides primitives for describing programs in terms of their resource needs, and of an *Abstract Resource Analyzer* (ARA) that, by using a transitional semantic of the Java bytecode, provides an abstraction of the program behavior in terms of the resources needed for its execution. Indeed, ARA is parameterized with *resource consumption profiles* to account for the different device characteristics. The resource model and ARA are integrated in a larger framework - CHAMELEON - that supports all the application's life cycle, from development to deployment, further tackling adaptation of the application to the target execution environment. The framework has been fully implemented in *Java* (although other languages are eligible) and targets resource-constrained Java-enabled devices.

The work described in this paper represents an enhancement w.r.t. the ones in [4, 5]. In particular, ARA has been split into two components for optimization purposes and the transition system it is based on has been redefined; RM has been modified accordingly and extended to introduce resource relationships for better computing resource consumption. Because of space limitation, we cannot include a related work section, however, we discuss some main related work during the presentation (see [2, 4, 5] for further references).

2 The framework

In this section we describe the server- and the client-side components of CHAMELEON [2] and briefly discuss one of the roles the framework played within the PLASTIC project [1, 6] whose goal is the agile development/deployment of lightweight and self-adapting services. Specifically, we focus on how CHAMELEON has been used for reasoning on applications and execution environments in terms of the resources they need and offer, respectively (i.e. resource demand and resource supply), and to suitably adapt the application to the environment that will host it.

PLASTIC's application code consists of two parts: the *core* and the *adaptive* code. The core code is the frozen portion of the application and represents the invariant semantics of the service. The adaptive one is a "generic code" that represents the degree of variability that makes the code capable to adapt to the execution contexts [1] . The framework is composed of the following 4 components:

• The *Development Environment* is based on a Programming Model (PM) that provides developers with a set of ad-hoc extensions to Java for easily specifying, in a flexible and declarative way, the generic code. Methods are the smallest building blocks that can be adapted.

```
public adaptable class Connection {
  public adaptable void send();
  public adaptable void connect(); ...
}
alternative UMTS adapts Connection {
 public void send(Data data) { /* via UMTS*/ ...}
 public void connect() { /* connects via UMTS*/
   Annotation.resourceAnnotation("UMTS(true)"); ... }
}
alternative BlueTooth adapts Connection {
 public void send(Data data) { /* via BlueTooth*/ ...}
 public void connect() { /* connects via BlueTooth*/
  if (isPowerOn){
    LocalDevice localDevice=LocalDevice.getLocalDevice();
  } else {
    display.out(''Please switch on Bluetooth'') }
}
```

Figure 1. An adaptable class

PM enriches the standard Java syntax by custom keywords and annotations for specifying the following elements: *adaptable classes* that are classes that contain one or more *adaptable methods*; *adaptable methods* that are the entry-points for a behavior that can be adapted; finally, *adaptation alternatives* that specify how one or more *adaptable methods* can actually be adapted. The output of this step is an extended Java program, i.e. a generic code.

▷ *Figure 1 represents a snippet of an adaptable code. The* adaptable class Connection *contains two* adaptable methods*:* send *and* connect *(introduced by the keyword* adaptable*). Adaptable methods do not have a definition in the adaptable class where they are declared but they are defined within* adaptation alternatives *(see the keywords* alternative *and* adapts*). It is possible to specify more than one alternative for a given adaptable class provided that for each adaptable method there exists at least one alternative that contains a definition for it. The* Connection *class has two alternatives: one connects and sends data via the UMTS network interface and the other one via the Bluetooth interface.*

To some extent, PM can be related to the work on Product Line Architectures (PLA) and variation in [3]. Our generic code can be seen as a PLA *features model* that represents: *commonalities* through adaptable classes, *variation points* through adaptable methods, and variation point *refinements* through adaptation alternatives.

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

Figure 2. Annotation class

Annotations may also add information about particular code instructions (see the keyword Annotation). They are specified at the source code level by means of calls to the "do nothing" methods of the Annotation class shown in Figure 2. In this way, after compilation, annotations are encoded in the bytecode through well recognizable method calls to allow for easy processing. Annotations can be of two types: (*i*) resourceAnnotations which directly express a resource demand. For instance, in Figure 1, the method call Annotation.resourceAnnotation ("UMTS(true)") demands for a UMTS interface; (*ii*) loopAnnotations express an upper bound to the number of loops. We did not use AspectJ for implementing our annotation mechanism because we wanted to give the programmer the freedom of easily inserting annotations in any place of the application code: AspectJ is very powerful but allows the definition of point cuts only in a limited set of well defined places (e.g., method calls in method's body, before and after method execution).

• The *Resource Model* (RM) is a formal model that allows for characterizing the computational resources needed to execute an application. RM is presented in Section 3.

• The *Abstract Resource Analyzer* (ARA) is the CHAMELEON server-side component that computes resource consumption. ARA is formally presented in Section 4.

• The *Execution Environment* can be any device that will host the execution of the code. Typically the Execution Environment is provided by PDAs, mobile phones, smart phones, etc. The Execution Environment is not strictly part of our framework. However, exploiting the CHAMELEON client-side component, it provides a declarative description of the resources it makes available to the application (i.e., the *resource supply*) and the *resource consumption profile*.

• The *Customizer* is the server-side component that takes in the provided resource supply and the profile, and basing on the ARA results, selects and delivers the right adaptation alternative. This step delivers standard bytecode.

Note that, both the ARA and the Customizer, are executed on the server-side component of the framework which does not suffer resource limitations. Moreover, as explained in Section 4, the heavy task of ARA is executed only once.

3 The Resource Model

We model a *resource* as a typed identifier that can be associated to *Natural*, *Boolean* or *Enumerated* values (left hand-side of Figure 3 shows an example of some resource definitions). Natural values are used for consumable resources whose availability varies during execution (e.g., energy, heap space). Boolean values define non-consumable resources that can be present or not (e.g., libraries, network radio interfaces). Enumerated values define non-consumable resources which admits a restricted set of values (e.g. screen resolution, network type).

Resource Definition:	***Resource Demand***:
`define Energy as Natural`	`{Bluetooth(true),`
`define Bluetooth as`	`Resolution(high)}`
`Boolean`	***Resource Supply***:
`define Resolution as`	`{Bluetooth(false),`
`{low, medium, high}`	`UMTS(true),Resolution(low)}`

Figure 3. Resource examples

A *resource instance* is an association $res(val)$ where a resource res is coupled to its value $val \in typeof(res)$ (e.g.

`Bluetooth(true)`). A *resource set* is a set of resource instances with no resource occurring more than once. We also define the notions of *compatibility* between two resource sets to verify if the resource set describing the *resource demand* of an alternative is compatible with the set representing the *resource supply* of an execution environment deciding if the application can there run safely.

$\triangleright$ *For instance, in Figure 3, the resource sets are clearly not compatible since the adaptation alternative demands for a bluetooth interface and high screen resolution but the execution environment does not supply any of them.*

In the remainder of this section we define some operations used by ARA for manipulating resource instances and sets that describe the application's resource demands.

• 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 follows:

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

The max operator returns the maximum value according to the ordering relation associated to the resource type.

• Given a *Natural* n and a resource instance $r(v)$, the scalar multiplication $n \odot r(v)$ is defined as follows:

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

• Given two resource sets R_1 and R_2, the resource set $R_1 \oplus R_2 = S_1 \cup S_2$ where $S_1 = (R_1 \cup R_2) \setminus (R_1 \cap R_2)$ and $S_2 = \{r(x) \oplus r(y) \mid r(x) \in R_1, r(y) \in R_2\}$.

• Given a Natural n, ad a resource set R, $n \odot R$ is the resource set $R' = \{n \odot r \mid r \in R\}$.

RM, suitably extended, also allows the specification of relationships among resources so to derive the consumption of a resource from the consumption of other resources.

$\triangleright$ *For instance, the expression $Time \doteq 2 \cdot CPU$ (as part of resource consumption profile) specifies that each consumed CPU unit will induce the consumption of 2 Time units; the more complex expression $Energy \doteq Bluetooth(true)?4 \cdot CPU : 2 \cdot CPU$ specifies that if Bluetooth is required (and hence used) by the application then each consumed CPU unit will induce the consumption of 4 Energy units, 2 Energy units otherwise.*

This mechanism permits to derive the consumption of resources not previously considered (neither by the generic code nor by the ARA analysis) without changing the generic code and without performing again the ARA analysis.

4 The Abstract Resource Analyzer

In this section we present the *Abstract Resource Analyzer* (ARA) that implements a transitional semantics for analyzing adaptable resource-aware Java applications.

ARA inspects the bytecode of the adaptation alternatives and is parametric on the *resource consumptions profile* associated to bytecode instructions in a given execution environment. Resource consumption profiles provide the description of the characteristics of a specific execution environment, in terms of the impact that Java bytecode instructions have on the resources;

1) aload_0 $\rightarrow$ {CPU(2)} **2)** .* $\rightarrow$ {CPU(1), Energy(1)}
3) invokestatic LocalDevice.getLocalDevice() $\rightarrow$ {Bluetooth(true), Energy(20)}

Figure 4. A resource consumption profile

these profiles associate resources consumption to particular patterns of bytecode instructions specified as *regular expressions*. Since the bytecode is a verbose language[1], this allows to define the resource consumption associated to both basic instructions (e.g., `ipush`, `iload`, etc.) and complex ones, e.g. method calls.

$\triangleright$ *Figure 4 represents an example of a resource consumption profile over resources defined in Figure 3. For example, the last row states that a call to the* `getLocalDevice()` *static method of the* `LocalDevice` *class of the* `javax.bluetooth` *library requires the presence of Bluetooth on the device (*`Bluetooth (true)`*), and it causes a consumption of the resource Energy equal to 20 cost units. Note that the expression ".∗" matches every bytecode.*

By means of profiles, ARA can be instantiated w.r.t. the resource-oriented characteristics of a specific execution environment. Moreover, since the *Development Environment* considers methods as the smallest adaptable entities (see Section 2), and we are interested in the worst case analysis, ARA performs a per-method static analysis and retrieves the resource demand of all its possible execution paths.

The work in [5] defines an operational semantics where each bytecode instruction is matched against the *resource consumption profile* to obtain its resource demand, and that is then opportunely combined with that of other instructions to obtain the overall resource demand. Since the resource consumption profile is provided by the device, the computation of the resource demand is dependent on the device characteristics and therefore can be performed by the CHAMELEON server only after the device's request has been received. Since the analysis process is computationally expensive, this makes serving device requests a time-consuming task. However, the analysis performed by ARA considers all the bytecode instructions in their isolation and, in this work, we take advantage of this fact in order to separate the analysis from the actual resource demand calculation, that is device dependent. By doing so, the result of the analysis will be a description of the application structure that is parametric with respect to information provided through resource consumption profiles. The final result, in fact, will be calculated by simply instantiating the parameters in the extracted description. Differently from [5], in this paper we split ARA in two components: the *Abstract Code Analyzer* (ACA), that performs the bytcode analysis,

[1]This is particularly true for method invocations where the method is uniquely identified by a fully qualified *id* (base class identifier + name + formal parameters.

and the *Abstract Resource Calculator* (ARC), that calculates the actual resource consumption of the application. By "refactoring" ARA in this way, we can perform the analysis of each application once for all device requests, thus optimizing the whole process.

Before describing these components, we introduce some definitions that will be used in the following sections.

Definition* 1*: An ***Instruction Pattern*** (*i_pattern*) is a regular expression that represents a set of Java bytecode instructions; an ***Instruction Pattern Profile*** (*i_pattern profile*) is a set of i_patterns.

An i_pattern profile ideally contains all the known i_patterns either from a standard resource consumption profile, or from previously encountered i_patterns defined in some resource consumption profile describing a device requesting an application.

▷ *For instance,* {`aload_0,.*,invokestatic LocalDevice.getLocalDevice()`} *is the i_pattern profile derived from the resource consumption profile of Figure 4.*

Definition* 2*: An ***Instruction Pattern Occurrence*** (*i_pattern occurrences*) is an association $p(n)$ where an i_pattern p is coupled to the number n of times it occurs in an execution path. An ***Instruction Pattern Set*** (*i_pattern set*) is a set of i_pattern occurrences.

▷ *For example,* `.*(3)` *is an i_pattern occurrences stating that the i_pattern* `.*` *occurred 3 times.*

4.1 The Abstract Code Analyzer

ACA performs a linear scanning of the methods' bytecode by following each possible execution path entailed by the *control flow* constructs of the bytecode itself (i.e., standard *sequential control flow*, *conditional and unconditional forward jumps*, *backward jumps* and *method calls*). Each time a fork is found (es. conditional jump), new computations are generated, one for each branch.

The analysis basically tries to match each encountered instruction against the i_pattern profile. For each possible execution path, ACA will derive an i_pattern set that associates each i_pattern in the i_pattern profile to the number of times it has been matched. Moreover, as the framework gives the possibility to insert in the source code resource annotations (see Section 2), during the scanning process, ACA has to keep trace of the encountered annotations. This is done by adding them to a resource set. The result of the analysis of an execution path is a pair *Computation-Pair=(P,R)* of an i_pattern set P and a resource set R.

ACA shares the idea of mapping JVM bytecode to a transition system by exploring all the possible computation paths with other existent tools such as Java Pathfinder (JPF) [7]. JPF checks for property violations (deadlocks or unhandled exceptions) traversing all execution paths. Differently from JPF, ARA get rid of variable values and abstracts the JVM w.r.t. resource consumption. In this abstraction ACA consider bytecode instructions behaviour only by taking into account their effects on resources.

Before presenting the transition system that formalizes ACA, we introduce some operation over i_patterns and i_pattern sets.

Definition* 3*: Given two occurrences of the same i_pattern p, we define the ***i_pattern occurrences sum operator*** $\oplus$ as follows: $p(x) \oplus p(y) = p(x + y)$.

Definition* 4*: Given an i_pattern occurrence $p(x)$ and a scalar n, we define the ***i_pattern occurrences scalar multiplication operator*** $\odot$ as follows: $n \odot p(x) = p(n * x)$.

The $\oplus$ operator is basically used when analyzing linear control flow to sum new i_pattern occurrences, while the $\odot$ operator is used to take into account loops that basically multiplies the number of occurrences of the i_patterns found in its body. Furthermore, these operators are extended to i_pattern sets in the same way as we have done for resource sets in Section 3.

4.1.1 Transition System

In the following we describe the operational semantics of the ACA by using a transition system. The configurations for the transition system are the following: $\Gamma_{ACA} : \{\langle e, o, m, pc, i, r\rangle\} \cup \mathcal{T}_{ACA}$. The structure of the JVM runtime environment is modeled by the function e. For all the application's classes, $e(\texttt{cId}) = \langle \texttt{scId}, cp, meth, statf\rangle$ maps a class identifier `cId` to a tuple of its superclass' identifier `scId`, constant pool cp, methods $meth$, static fields $statf$. o is a function that binds a bytecode instruction to the i_patterns defined in a i_pattern profile and returns the set of all matched i_patterns associated to a single occurrence. For each method $m \in meth$, $m = \langle \texttt{cid}, n_m, t_m\rangle$ is a tuple of the identifier of the method's class `cid`, the number of arguments n_m, and a function t_m that maps a program counter pc to the method bytecode. r is the resource set associated to the resource annotation of the current computation; i is the i_pattern set that stores the i_patterns occurrences of the current computation. The final configurations $\mathcal{T}_{ACA}$ are sets of $ComputationPairs$ (see Section 4.1). The reason for this choice is that branching instructions and method invocations, as shown later on, produce different flows of control: each pair describes the outcome of one of the possible execution paths. In this way, a method can be finally characterized by all the possible resource demands of all the possible execution paths. The initial configuration of ACA is $\{\langle e, o, m, 0, \phi, \phi\rangle\}$.

Figure 5 reports the transition system rules. For sake of space, we will analyze only some rules of the transition system and show the main differences with [5]. Other rules are derived by [5] in a similar manner.

The **standard control flow rule** states that from a given configuration the current instruction is matched with the function o against the i_pattern profile to construct an

instruction - standard control flow

$$\frac{\begin{array}{c} m = \langle \texttt{cid}, n_m, t_m \rangle \quad t_m(pc) = \texttt{instruction} \\ i' = i \oplus o(\texttt{instruction}) \\ r' = r \oplus ResourceAnnotation(pc) \end{array}}{\langle e, o, m, pc, i, r \rangle \rightarrow_{ACA} \langle e, o, m, pc+1, i', r' \rangle}$$

goto forward

$$\frac{\begin{array}{c} m = \langle \texttt{cid}, n_m, t_m \rangle \quad t_m(pc) = \texttt{goto}\ \mathit{addr} \\ i' = i \oplus o(\texttt{goto}\ \mathit{addr}) \quad addr > pc \end{array}}{\langle e, o, m, pc, i, r \rangle \rightarrow_{ACA} \langle e, o, m, addr, i', r \rangle}$$

if_TcmpOP - conditional forward jump

$$\frac{\begin{array}{c} m = \langle \texttt{cid}, n_m, t_m \rangle \quad t_m(pc) = \texttt{if\_TcmpOP}\ \mathit{addr} \quad addr > pc \\ i' = i \oplus o(\texttt{if\_TcmpOP}\ \mathit{addr}) \\ \langle e, o, m, pc+1, i', r \rangle \overset{*}{\rightarrow}_{ACA} R_{branch1} \\ \langle e, o, m, addr, i', r \rangle \overset{*}{\rightarrow}_{ACA} R_{branch2} \\ R = R_{branch1} \bigcup R_{branch2} \end{array}}{\langle e, o, m, pc, i, r \rangle \overset{*}{\rightarrow}_{ACA} R}$$

goto backward

$$\frac{\begin{array}{c} m = \langle \texttt{cid}, n_m, t_m \rangle \quad t_m(pc) = \texttt{goto}\ \mathit{addr} \\ i' = i \oplus o(\texttt{goto}\ \mathit{addr}) \quad addr < pc \\ n = LoopAnnotation(addr) \quad n > 1 \\ m' = BuildMethod(m, addr, pc) \\ \langle e, b, m', 0, \emptyset, \emptyset \rangle \overset{*}{\rightarrow}_{ACA} \{(i_1, r_1), \ldots (i_n, r_n)\} \\ \left. \begin{array}{c} i'_k = i' \oplus (n-1) \odot i_k \\ r'_k = (n-1) \odot r_k \\ \langle e, b, m, pc+1, i'_k, r'_k \rangle \overset{*}{\rightarrow}_{ACA} R_k \\ R = \bigcup_k R_k \end{array} \right\} \forall k = 1 \ldots n \end{array}}{\langle e, b, m, pc, i, r \rangle \overset{*}{\rightarrow}_{ACA} R}$$

invokevirtual

$$\frac{\begin{array}{c} m = \langle \texttt{cid}, n_m, t_m \rangle \quad t_m(pc) = \texttt{invokevirtual}\ \mathit{i} \\ e(\texttt{cId}) = \langle \texttt{scId}, cp, meth, statf \rangle \\ cp(\mathit{i}) = \langle \texttt{cId}_1, \texttt{mId} \rangle \\ LookupOverrides(\texttt{cId}_1, \texttt{mId}) = M = \{m_1, m_2, \ldots, m_z\} \\ i' = i \oplus o(\texttt{invokevirtual}\ \texttt{cId}_1.\texttt{mId}) \\ r' = r \oplus_{(s)} ResourceAnnotation(pc) \\ \forall m_k \in M\ \langle e, b, m_k, 0, i', r' \rangle \overset{*}{\rightarrow}_{ACA} \{(i_{k1}, r_{k1}), \ldots (i_{kn}, r_{kn})\} \\ \forall k = 1 \ldots z\ \forall j = 1 \ldots n\ \langle e, b, m, pc+1, i_{kj}, r_{kj} \rangle \overset{*}{\rightarrow}_{ACA} R_{kj} \\ R = \bigcup_{k,j} R_{kj} \end{array}}{\langle e, b, m, pc, i, r \rangle \overset{*}{\rightarrow}_{ACA} R}$$

return

$$\frac{\begin{array}{c} m = \langle \texttt{cid}, n_m, t_m \rangle \quad t_m(pc) = \texttt{return} \\ i' = i \oplus o(\texttt{return}) \end{array}}{\langle e, o, m, pc, i, r \rangle \rightarrow_{ACA} \{(i', r)\}}$$

Figure 5. Transition System Rules

i_pattern set that is added, via the $\oplus$ operator, to the current i_pattern set. If the instruction at the current program counter pc is a resource annotation (see Section 2), the corresponding resource set (passed as parameter to the resourceAnnotation method call) is computed by the function $ResourceAnnotation(pc)$ and is added to the current resource set. Finally the program counter is increased. This rule is the most general one of the transition system and it is applied when no other rule can be applied. The difference w.r.t. the analogous rule in [5] is that it updates the i_pattern set to store the occurrence of the instruction. Moreover, no reference is made to the resource consumption profile and the resource set is increased only if the instruction represents a resource annotation.

▷ *As an example, let* m *be the* `connect ()` *method whose bytecode is shown in Figure 6 and let o be the binding function built on the i_pattern profile derived from the resource consumption profile in Figure 4. Starting from the initial state, the standard rule is applied:* `aload_0` *match both the (overlapping) .* and aload_0 patterns, so, we will have:* $i' = \{.*(1), aload\_0(1)\}$ *and* $r' = \phi$*, so the new state will be* $\langle e, o, m, 1, \{.*(1), aload\_0(1)\}, \phi \rangle$*. Section 4.2 explains how ARC solves overlapping matches.*

```
 0: aload_0
 1: getfield isPowerOnZ
 2: ifeq -> 6
 3: invokestatic LocalDevice.getLocalDevice()LLocalDevice;
 4: astore_1
 5: goto -> 10
 6: aload_0
 7: getfield screenLScreen;
 8: ldc "Please switch on Bluetooth"
 9: invokevirtual display.out(Ljava/lang/String;)V
10: return
```

Figure 6. Bluetooth connect() bytecode

The **conditional forward jump transition rule** states that when a branch point is reached, the system starts two "parallel' computation analysis, one for each of the two branches. Both the computation analyses are started in a state where m is maintained as the current method to analyze. The i_pattern set is enriched with the one derived by the application of the function o to the `if` instruction, while the resource set r remain unchanged since jump instructions do not contain resource annotations. The first branch computation analysis is started at the next instruction, i.e., at *pc+1*, while the second one is started, at the jump target location *addr*.

The transition rule produces a final state R that is the union of the resource sets produced by the analysis of the two branches. This ensures that all possible execution paths are analized. In the rule we made use of $\overset{*}{\rightarrow}_{ACA}$ that is the transitive closure of the $\rightarrow_{ACA}$ relation. This rule, differently from the analogous one in [5], uses the current instruction to update the i_pattern set instead of the resource set.

▷ *For example consider the behaviour of the transition system at instruction 2 of Figure 6. The current state is* $\langle e, o, m, 2, \{.*(2), aload\_0(1)\}, \phi \rangle$*.* `ifeq->6` *matches only .*, so* $i' = \{.*(3), aload\_0(1)\}$*. The i_pattern set associated to* $branch_1$ *will be* $\{.*(8), aload\_0(2)\}$*, while the one of* $branch_2$ *will be* $\{.*(7), aload\_0(1), invokestaticLocalDevice...(1)\}$*. The resource set is always* $\emptyset$*, hence the final state (i.e., the resource demand of the connect() method) will be* $\{(\{.*(8), aload\_0(2)\}, \emptyset), (\{.*(7), aload\_0(1), invokestaticLocalDevice.getLocalDevice()(1)\}, \emptyset)\}$*.*

The **return transition rule** states that when the transition system encounter a `return`, the ComputationPair associated to the current computation is retrieved as final state (`return` is always the last instruction of a computation).

4.2 Abstract Resource Calculator

By using the resource consumption profile, ARC attaches resource informations to the $ComputationPair$ sets resulting by the analysis performed by ACA, and retrieves the resource demand of each method.

As said in Section 2, a resource consumption profile specifies the amount of resources consumed by the execution of the instructions described by an i_pattern. On the other side, the i_pattern set of a ComputationPair expresses the number of times an i_pattern has been matched in an execution path. This information is combined by ARC to obtain resource demands due to instructions execution. Moreover, it is completed with the resource set that, as part of the ComputationPair, expresses the resource consumption due to annotations. The whole process is formalized in the following definitions.

Definition* 5*: Given a resource r, an i_pattern p and a resource consumption profile C, the ***pCons function*** returns the resource set that represents the consumption of r associated to one occurrence of p in C.

Note that, during the ACA analysis process, more than one *overlapping*[2] i_pattern of the i_pattern profile can be matched by the same instruction, hence increasing all their occurrences. If those i_patterns impact over the same resources in the resource consumption profile, this would lead to an erroneous calculus of the resources' consumption, since the single instruction occurrence will be computed more times. ACA solves this problem in this way: when an i_pattern set contains overlapping i_patterns that impact over the same resources, their occurrences are decreased of the amount of occurrences of all the included patterns.

$\triangleright$ *For example, according to the resource consumption profile of Figure 4* `aload_0` *impacts only CPU resource, while* $.*$ *impacts both CPU and Energy resources. Hence, the number of real occurrences of the i_pattern set* $S{=}\{.*(8), aload\_0(2)\}$ *of the final state, will be* $\{.*(6), aload\_0(2)\}$ *with respect to* CPU *resource, and* $\{.*(8), aload\_0(2)\}$ *with respect to* $Energy$ *resource.*

The demand of a resource corresponding to an i_pattern set will be given by the sum of the resource demands associated to each i_pattern in the set multiplied for the number of the real occurrences for that resource, as specified in the following function:

Definition* 6*: Given a resource r, an i_pattern set $S = \{p_1(n_1), \ldots, p_k(n_k)\}$ and a resource consumption profile C, we define the ***demand function*** as follows: $demand(r{,}S{,}C){=}\bigoplus_i n_i{\odot}PatCons(r, p_i, C), \forall i = 1 \ldots k.$

$\triangleright$ *Let* S *and* C *be as in the previous example, demand(CPU, S, C) =* $\{$*CPU(10)*$\}$*, demand(Energy, S, C)=*$\{$*Energy(8)*$\}$*.*

The total resource demand of an i_pattern set will be the union of its demand with respect to all resources.

$\triangleright$ *In our example it will be* $\{$*CPU(10), Energy(8)*$\}$*.*

The resource demand of a computation pair will be the sum of the resource demand due to the i_pattern set element and the one expressed in the resource set element of the pair. Finally, the resource demand of a computation pair set (i.e., of a method) will be the union of the resource demand of all the computation pairs in the set, as follows:

Definition* 7*: Given a computation pair set $S = \{(P_1, R_1), \ldots (P_n, R_n)\}$ and a resource consumption profile C, the ***ARC function*** is defined as:
$ARC(S, C) = \bigcup_i demand(P_i, C) \oplus R_i \quad \forall i = 1 \ldots n$

$\triangleright$ *The resource demand of* `connect()` *method of Figure 6 is* $\{$ $\{$*CPU(10), Energy(8)*$\}$, $\{$*CPU(8), Energy(26), Bluetooth(true)*$\}$ $\}$ *that is to say that to correctly execute the method (i.e., to be compatible) we need at least 10 unit of CPU, 26 of Energy and the Bluetooth.*

Referring to Section 3, supposing that we have the expression $Time \doteq 2 \cdot CPU$ *as part of a new resource consumption profile, without performing again the analysis, we can estimate that we need 20 Time unit for executing the method.*

5 Conclusion and Future Work

In this paper we have presented an enhanced version of our approach to adaptable resource-aware Java applications for resource-constrained devices. The presented framework is more performing and computes better resource consumption w.r.t. the one in [4, 5]. As future work more expressive bindings might be introduced in order to better model the execution environment w.r.t resources (e.g., bindings matched against code patterns, not only instructions). The proposed mechanism for expressing resource relationships should be fully formalized and more expressive relationships should be investigated. On the framework side, additional static analysis techniques might be introduced in ARA to strengthen the approximation of the program execution, and the programming model might be improved both on the expressiveness side and on the tooling support.

References

[1] M. Autili, V. Cortellessa, P. D. Benedetto, and P. Inverardi. On the adptation of context-aware services. In *Proc. of SOC@Inside'07 (ICSOC'07)*, Wien, 2007.

[2] P. Inverardi, F. Mancinelli, and M. Nesi. A declarative framework for adaptable applications in heterogeneous environments. In *SAC*, pages 1177–1183, 2004.

[3] I. Jacobson, M. Griss, and P. Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success.* Addison Wesley, 1997.

[4] F. Mancinelli and P. Inverardi. A resource model for adaptable applications. In *Proc. of SEAMS'06 (ICSE'06)*, pages 9–15, NY, USA, 2006.

[5] F. Mancinelli and P. Inverardi. Quantitative resource-oriented analysis of java (adaptable) applications. In *Proc. of WOSP'07*, pages 15–25, NY, USA, 2007.

[6] PLASTIC project. http://www.ist-plastic.org.

[7] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *ASE journal*, 10(2), 2003.

[2]Recalling that i_patterns are regular expressions, let $L(p_1)$ and $L(p_2)$ be the regular languages generated by the i_patterns p_1 and p_2, respectively. We say that p_1 overlaps p_2 if $L(p_1) \cap L(p_2) \neq \phi$ and that p_1 is included in p_2 iff $L(p_1) \subseteq L(p_2)$