The Nullness Analyser of JULIA

Fausto Spoto

Dipartimento di Informatica, Università di Verona, Italy
fausto.spoto@univr.it

Abstract. This experimental paper describes the implementation and evaluation of a static nullness analyser for single-threaded Java and Java bytecode programs, built inside the JULIA tool. Nullness analysis determines, at compile-time, those program points where the `null` value might be dereferenced, leading to a run-time exception. In order to improve the quality of software, it is important to prove that such situation does not occur. Our analyser is based on a denotational abstract interpretation of Java bytecode through Boolean logical formulas, strengthened with a set of denotational and constraint-based supporting analyses for locally non-`null` fields and *full* arrays and collections. The complete integration of all such analyses results in a correct system of very high precision whose time of analysis remains in the order of minutes, as we show with some examples of analysis of large software.

1 Introduction

Software is everywhere nowadays. From computers, it has subsequently been embedded in phones, home appliances, industries, aircraft, nuclear power stations, with more applications coming every day. As a consequence, its complexity is increasing and bugs spread with the software itself. While bugs are harmless in some situations, in others they have economical, human or civil consequences. Therefore, software verification is increasingly recognised as an important aspect of software technology and consumes larger and larger portions of the budget of software development houses.

Software verification aims at proving software error-free. The notion of *error* is in general very large, spanning from actual run-time errors to bad programming practices to badly-devised visual interfaces. Techniques for software verification are also manifold. It is generally accepted that the programming language of choice affects the quality of software: languages with strong static (compile-time) checks, simple syntax and simple semantics reduce the risk of errors. Good programming practices are also a key element of software quality. The reuse of trusted libraries is another. Nevertheless, bugs keep being present in modern software.

Hence, the big step in software verification should be automatic software verification tools, able to find, automatically and in reasonable time, the bugs in a program. Of course, no tool can be both correct (finding *all* bugs) and complete (finding *only* bugs), because of the well-known undecidability property of

E.M. Clarke and A. Voronkov (Eds.): LPAR-16, LNAI 6355, pp. 405–424, 2010.

software. But a tool can well be useful if it approximates the set of bugs in a precise way, as a set of *warnings*, yet leaving to the programmer the burden of determining which warnings are real bugs.

In this paper we describe our implementation of a software verifier for `null`-pointer analysis, also called *nullness* analysis, embedded inside the JULIA tool, carried out in the last three years. We show the structure of the implementation, its precision and strengths. Its goal is to spot program points in Java and Java bytecode where a `null`-pointer exception might be raised at run-time, because the value `null` is dereferenced during the access to a field, a call to a method or during synchronisation. Although JULIA works over bytecode, it can be applied to Java source code by compiling it into bytecode and we only report examples, here, over Java source code, more easily understood by the reader.

It is important to stress that we do not claim the absolute superiority of our implementation *w.r.t.* other tools for automatic software verification of Java. We only want to highlight the specific features of our tool that make it relevant in the context of software verification, possibly in coexistence with other tools. Namely, a software verification tool can be judged *w.r.t.* many orthogonal aspects, including:

correctness/completeness. A correct verifier has the advantage of definitely excluding the presence of bugs outside the list of warnings provided to the user. Nevertheless, it is understandable that some tools have decided to sacrifice correctness (and completeness), since this lets them shrink the list of warnings to the most realistic ones: a long list of warnings can only scare the user, who will not use the tool anymore. Our choice is to stay correct and restrict the list of warnings as much as possible with the help of the most advanced techniques for static analysis;

annotations/no annotations. A verifier using annotations requires the programmer to decorate the source code with invariants that can be exploited but must also be checked by the tool. Examples are method pre- and post-conditions and loop invariants. This approach burdens the programmer with an extra task, but it obliges him to reason on what he writes and clean and document the code. Moreover, a tool exploiting annotations can be extremely precise and its analysis can be modular, that is, parts of the program can be changed without having to verify everything again. Our choice has been not to use annotations;

genericity. A generic verifier performs many kinds of software verifications and is consequently more useful. Nevertheless, an implementation centered around a given verification problem can be more focused on its specific target and more optimised. Our choice is to have a general analyser (JULIA), but our limited time has been used to build advanced instantiations for nullness verification and termination analysis [20] only;

real-time/off-line. A real-time verifier runs every time the source code is modified and provides immediate feedback to the programmer. It is typically modular*, i.e.,* it performs local reasonings on each method and requires source code annotations. Sometimes (but not always) some incorrect assumptions

are made, such as assuming that methods are always called with non-`null` arguments that do not share any data structure, which is not always true. Real-time verifiers are very effective during software development. Off-line verifiers require larger resources and are consequently used at fixed milestones during software development. But they can afford the most precise verification techniques and can provide a correct and thorough report on the analysed program. Our choice is that of an off-line analyser, although its run-times are kept in the order of minutes on a standard computer.

Many different tools provide some form of nullness analysis. The ECLIPSE IDE [1] provides a real-time, very limited, incorrect, largely imprecise nullness analysis, as a plug-in. ESC/JAVA2 [4] is a generic tool, evolved from ESC/JAVA [8], that uses theorem-proving to perform an off-line nullness analysis, using annotations provided by the programmer. The tool can also work without annotations, but then the number of warnings for `null`-pointer dereferences becomes very large [15]. It is possible to infer typical nullness annotations with a tool called HOUDINI [7], which calls ESC/JAVA to validate or refuse the annotations. It does not work with ESC/JAVA2 and the latest version is from 2001. Since then, Java has largely changed. We did not manage to build HOUDINI and ESC/JAVA on our Linux machine. FINDBUGS [10] is another generic tool that performs an only syntactical off-line nullness analysis, in general incorrect (see later for an example). Nevertheless, this tool is considered very effective at spotting typical erroneous programming patterns where `null`-pointer bugs occur. JLINT [2] performs a simple, only syntactical analysis of the code to spot some possible `null`-pointer dereferences of variables. It did not compile on our Linux machine, so we could not experiment with it. According to the `README` file, it is updated to Java version 1.4 only. DAIKON [6] is a tool that infers *likely* nullness annotations, but there is no guarantee of their correctness. A comparison of those tools, beyond the case of nullness analysis, is presented in [15]. It must be stated that they do not only verify `null`-pointer dereferences, but also other properties of Java programs. NIT [11], instead, is a tool explicitly targeted at nullness analysis. It performs a provably correct, fast off-line nullness analysis of Java bytecode programs. This tool is the most similar to JULIA, since both are based on semantical static analysis through abstract interpretation [5] of Java bytecode. It is faster than JULIA, but this comes at the price of precision, since its latest version $0.5d$ is almost as precise as an old version of JULIA (see the experiments in [18]) that used only the techniques up to Subsection 3.2 of this paper.

The contribution of this paper is the presentation of the structure of the nullness analysis implemented inside JULIA, together with a brief overview of the techniques that it exploits, partially based on logical formulas. As such, it is an example of implementation of logical systems for static analysis, with strong correctness guarantees. Our goal has been precision and correctness, which entails that JULIA will not be so fast as other tools, although it is already able to analyse around 5000 methods in a few minutes on standard hardware.

The rest of the paper is organised as follows. Section 2 yields an introductory example of nullness analysis with JULIA, FINDBUGS and NIT. Section 3

describes the structure of the nullness analysis of JULIA and how each component contributes to the precision of the overall result. Section 4 describes how JULIA creates an annotation file collecting nullness information, that can later be used by other tools. Section 5 concludes the paper. The theoretical results that form the basis of our implementation have already been published elsewhere. In particular, [17] reports a formal description and proofs of correctness for the techniques of Subsections 3.1 and 3.2; [18] includes the same for Subsection 3.3 also; [19] reports definitions and proofs for the rawness analysis of Section 4 and formalises the constraint-based techniques used in JULIA. The new analyses for arrays and collections in Subsections 3.4 and 3.5 have never been published before.

2 An Introductory Example

In order to show the kind of precision that can be expected from our nullness analysis, consider the Java program in Figure 1. Methods `equals()` and `hashCode()` are reachable since they are called, indirectly, by the calls at lines 30 and 32 to the library method `java.util.HashSet.add()`. The analysis of that program through JULIA does not signal any warning. According to the correctness guarantee of JULIA, this means that the program will never dereference the `null` value at run-time. The analysis of the same program with NIT yields 8 warnings:

```
line 14: unsafe call to bool equals(Object)
line 16: unsafe call to bool equals(Object)
line 19: unsafe call to int hashCode()
line 24: unsafe call to String replace(char,char)
line 26: unsafe call to String replace(char,char)
line 31: unsafe field read of C inner
line 34: unsafe call to void println(String)
line 35: unsafe call to void println(String)
```

They are *false alarms*, since:

- at lines 14 and 19, field `name` is always non-`null` in the objects of class `C` (look at lines 24 and 26 and consider that the return value of `String.replace()` is always non-`null`). Hence the call to `equals()` does not dereference `null`;
- at line 16, the non-nullness of field `inner` has been already checked at line 15 (Java implements a short-circuit semantics for `||`);
- at lines 24 and 26, `args[i]` is non-`null`, since the Java Virtual Machine passes to `main()` an array `args` containing non-`null` values only, and there is no other call to `main()` in the program;
- at line 31, variable `t` cannot hold `null` since it iterates over the elements of the array `ts` (line 29), which is *filled* with non-`null` values by the loop at lines 23 – 27;
- at lines 34 and 35, the static field `System.out` cannot hold `null` since it is initialised to a non-`null` value by the Java Virtual Machine and is not modified after by the program in Figure 1 (it never calls `System.setOut()` with a possibly `null` argument).

```
0:  import java.util.*;
1:  public class C {
2:    private String name;
3:    private C inner;
4:    public C(String name, C inner) {
5:      this.name = name;
6:      this.inner = inner;
7:    }
8:    public String toString() {
9:      if (inner != null) return "[" + inner + "]";
10:     else return "[]";
11:   }
12:   public boolean equals(Object other) {
13:     return other instanceof C &&
14:       ((C) other).name.equals(name) &&
15:       (((C) other).inner == null ||
16:          ((C) other).inner.equals(inner));
17:   }
18:   public int hashCode() {
19:     return name.hashCode();
20:   }
21:   public static void main(String[] args) {
22:     C[] ts = new C[args.length];
23:     for (int i = 0; i < ts.length; i++) {
24:       ts[i] = new C(args[i].replace('\\','/'), null);
25:       if (i < ts.length - 1)
26:         ts[++i] = new C(args[i].replace('\\','/'), ts[i - 1]);
27:     }
28:     Set<C> s1 = new HashSet<C>(), s2 = new HashSet<C>();
29:     for (C t: ts) {
30:       s1.add(t);
31:       if (t.inner != null)
32:         s2.add(t.inner);
33:     }
34:     for (C t: s1) System.out.println(t.toString());
35:     for (C t: s2) System.out.println(t.toString());
36:   }
37: }
```

Fig. 1. A Java program exposing interesting considerations about nullness

The analysis of the same program with FINDBUGS yields no warnings. However, this result is the consequence of some optimistic (and in general incorrect) hypotheses assumed by FINDBUGS on the behaviour of the program under analysis: it cannot be taken as a proof of the fact that the program in Figure 1 never dereferences `null`. Namely, assume to comment out lines 15 and 31 in Figure 1. The program contains two bugs now: at line 16, method `equals()` is called on a possibly `null` field `inner`; at line 35, method `toString()` is called on a possibly

`null` variable `t`, since the set `s2` might contain `null` now, introduced at line 32. JULIA correctly spots these situations and issues two warnings:

```
line 16: call with possibly-null receiver to C.equals(Object)
line 35: call with possibly-null receiver to C.toString()
```

However, FINDBUGS keeps signalling no warning at all and hence is not able to find those two bugs and is incorrect. NIT keeps signalling the same 8 warnings seen above. It somehow misses the bug at line 35, since, there, no warning is signalled for the incorrect call to `toString()` but only a false alarm about the call to `println()`. However, this must be just a bug in the implementation of NIT, that might be corrected in next releases, since the underlying theory has been certified in COQ to be correct.

3 Nullness Analysis in JULIA

We describe here the phases of the nullness analyser implemented inside the JULIA analysis tool and their contribution to the overall precision. We had to use many techniques in order to achieve a very high level of precision. JULIA currently performs semantical static analyses based on denotational abstract interpretation and on constraint-based abstract interpretation. Both come in different flavors here, for inferring properties of local variables, fields, arrays of references and collections. We test each technique with all the previous techniques turned on as well, so we will see progressively increasing times and precision.

Before the actual nullness analysis starts, JULIA must of course load and parse the `.class` or `.jar` files containing the analysed Java bytecode application. Moreover, it must extract the *control flow* of the program, linking method calls with the method implementations that they actually call. We perform this through a nowadays traditional *class analysis* [13]. Type inference for local variables and Java bytecode stack elements is performed as in the official documentation [12]. These aspects of JULIA are completely independent from the actual analysis which is later performed, nullness, termination or other.

3.1 Nullness Analysis of Local Variables

Given a program point p, the number of local variables accessible at p is finite, although arbitrarily large. In particular, it is possible to access all and only the local variables declared by the method before p, which include the formal parameters of the method. This entails that it is possible to build a finite constraint expressing the nullness behaviour of the variables at p *w.r.t.* the nullness behavior of the variables at a subsequent program point p'. In [17], this constraint is defined as a Boolean logical formula whose models include all possible nullness behaviours for the local variables.

Assume for instance that p is line 22 in Figure 1. The only variable in scope at p is `args`, since `ts` is not yet declared at p. Program p' is the logically subsequent statement in that program, *i.e.,* the assignment `int i = 0` at line 23. At p',

variables `args` and `ts` are declared. The Boolean formula built by JULIA to relate the nullness of the variables at p to that of the nullness at p' uses Boolean variables of two forms: $\check{v}$ stands for the value of variable v at p; $\hat{v}$ stands for its value just after p, that is, at p'. A special variable e is used to represent exceptional states. Namely, that formula is:

$$\neg\check{e} \wedge (\neg\hat{e} \rightarrow (\neg\check{\texttt{args}} \wedge \neg\hat{\texttt{ts}})) \tag{1}$$

meaning that, if the assignment at line 22 is executed, then no exception must be raised immediately before p ($\neg\check{e}$); moreover, if no exception is raised by the assignment ($\neg\hat{e}$) then both `args` and `ts` are non-`null` at p' ($\neg\check{\texttt{args}} \wedge \neg\hat{\texttt{ts}}$). In (1), program variables of reference type have been translated into Boolean variables of two kinds: *input* variables, such as $\check{\texttt{args}}$, stand for the nullness of the corresponding program variable at the beginning of p, while *output* variables, such as $\hat{\texttt{args}}$, stand for the nullness of the corresponding variable at the end of p, *i.e.,* at beginning of p'. The special variable e stands for an exceptional situation (in the sense of a raised Java exception, implicit or explicit).

Consider now the assignment `int i = 0` at line 23. For it JULIA builds the Boolean formula:

$$\neg\check{e} \wedge \neg\hat{e} \wedge (\check{\texttt{args}} \leftrightarrow \hat{\texttt{args}}) \wedge (\check{\texttt{ts}} \leftrightarrow \hat{\texttt{ts}}) \tag{2}$$

that is, if that assignment is executed then no exception must be raised just before it ($\neg\check{e}$); no exception is ever raised by that assignment ($\neg\hat{e}$); the nullness of `args` and `ts` is not affected by the assignment to `i` ($(\check{\texttt{args}} \leftrightarrow \hat{\texttt{args}}) \wedge (\check{\texttt{ts}} \leftrightarrow \hat{\texttt{ts}})$). Nothing is said about variable `i`, since it has primitive type so its value is abstracted away.

In order to analyse the sequential execution of more statements, JULIA performs the *abstract sequential execution* of Boolean formulas, each abstracting one of the statements. For instance, the abstract sequential execution of (1) and then (2) is computed by matching the output variables of (1) with the input variables of (2). That is, those variables are renamed into the same new, fresh variables, the resulting two formulas are conjuncted ($\wedge$) and the new fresh variables are projected away through an existential quantification. The result is

$$\neg\check{e} \wedge \neg\hat{e} \wedge \neg\hat{\texttt{args}} \wedge \neg\hat{\texttt{ts}} \tag{3}$$

The latter is an abstraction of the state at the beginning of the execution of the loop at line 23, after the initialisation of `i`. It clearly states that `ts` is non-`null` there.

These Boolean formulas can be built in a methodological way as a bottom-up abstraction of the code. Their construction is based on the abstract sequential composition of formulas but also on the logical disjunction of two formulas, to abstract conditionals and virtual methods calls with multiple target implementations. Method calls are abstracted by *plugging* the analysis (*denotation*) of their body in the point of call, so that the resulting analysis is inter-procedural and completely context-sensitive. The latter means that the approximation of the final state, after a method call, is not fixed, but depends on the approximation of

the input state before the same call. Loops and recursion are modelled through a fixpoint calculation. We have used no widening, since the abstract domain of Boolean formulas has finite height (at each program point, the number of local variables in scope is fixed and finite) and we have never experienced the need of accelerating the convergence of the fixpoint calculations.

The detailed formalisation of this analysis and its proof of correctness are done in [17] by using abstract interpretation. The analysis can be used to guarantee the absence of `null` dereferences in the code. For instance, Equation (3), together with a formula built for the body of the loop at lines $23-27$ and stating that the nullness of `ts` does not change inside the loop, is enough to conclude that the dereference `ts.length` at line 23 does not raise any `null`-pointer exception.

The implementation of this analysis is quite efficient since binary decision diagrams [3] are used to represent the Boolean formulas and no aliasing information must be computed before the analysis: the abstraction into Boolean formulas abstracts away the heap memory completely and only considers the activation records of the methods, where the local variables live. This means, for instance, that we do not have to bother that the call to the constructor of `C` at line 22 modifies the nullness of `args` since this is just impossible: in Java, the callee cannot modify the value of the local variables of the caller, but only, possibly, the fields and array elements reachable from those local variables and the static fields and everything reachable from them, which are not local variables. This simplicity comes at a price: the relative imprecision of the results. For instance, the analysis of the program in Figure 1 with JULIA tuned down to use this technique only yields the following set of warnings (false alarms):

```
line 14: call with possibly-null receiver to String.equals(Object)
line 16: call with possibly-null receiver to String.equals(Object)
line 19: call with possibly-null receiver to String.hashCode()
line 24: call with possibly-null receiver to String.replace(char,char)
line 26: call with possibly-null receiver to String.replace(char,char)
line 31: read with possibly-null receiver of field inner
line 34: call with possibly-null receiver to C.toString()
line 35: call with possibly-null receiver to C.toString()
```

We report below the precision of this analysis, as the amount *derefs* of dereferences that are proved *safe* in the analysed programs, *i.e.,* having a provably non-`null` receiver. The analysed programs do not contain unsafe dereferences, as we have verified by checking, manually, the warnings issued by the most precise analysis (Subsection 3.5). Then a very precise analyser could in principle find out that 100% of their dereferences are safe. The only exception is `EJE`, that contains three bugs, that is, three dereferences that can actually happen on `null` sometimes. For `EJE`, a very precise analyser could in principle find out that 99.89% of the dereferences are safe.

We also report below the number of safe dereferences restricted to some frequent examples, namely, those related to field accesses (*access*), field modifications (*update*) and method calls (*call*). In this and the following tables, we have analysed the programs by including the whole `java.*` hierarchy in the analysis,

which is reflected in the relatively high number of methods analysed. Fewer library methods might be included, but worst-case assumptions would then be made for them by the analyser, compromising the precision of the results. The time of the analysis, in seconds, includes that for parsing the class files of the application and of the libraries. `OurTunes` is an open source cross-platform file sharing client which allows users to connect to iTunes and share music files. `EJE` is a text editor. `JFlex` is a lexical analysers generator. `utilMDE` are Michael Ernst's supporting classes for DAIKON; they include test applications, that is what we analyse. The `Annotation File Utilities` (in the following just `AFU`) are tools that allow to apply or extract annotations into Java source code (see also Section 4). The experiments have been performed on a quad-core Intel Xeon computer running at 2.66Ghz, with 8 gigabytes of RAM and Linux 2.6.27.

program	methods	time	derefs	access	update	call	warnings
`OurTunes`	3036	24.49	86.48%	92.95%	99.64%	79.12%	425
`EJE`	3077	33.31	77.04%	99.43%	100.00%	57.37%	926
`JFlex`	3735	39.05	81.29%	76.79%	98.33%	81.71%	1254
`utilMDE`	3706	43.05	92.85%	93.68%	99.06%	88.42%	252
`Annotation File Utilities`	3625	39.48	91.09%	88.45%	99.66%	86.64%	523

For verification purposes, this first technique does not have a satisfying precision. Nevertheless, it is interesting for optimisation (removal of useless nullness tests), which is less fussy about precision. Moreover, it is correct for the analysis of multi-threaded applications, while the other techniques that we are going to describe give, in special situations, incorrect results on multi-threaded programs (their adaptation to multi-threaded applications is on its way).

3.2 Globally non-`null` Fields

The warning at line 14 in Subsection 3.1 is a consequence of the fact that method `equals()` is called on field `name` of variable `other` rather than on a variable. Hence the technique of that subsection cannot prove that that call does not raise any `null`-pointer exception. The same happens for the warning at line 19. To solve this problem, we need information on the nullness of the fields also, not just of the local variables of the program. Field definitions are finite in any Java program, but an unbound number of objects can be allocated by a program, thus allowing an unbound number of field instances. Hence, it is not possible to allocate a Boolean variable for each of those instances. Even considering field definitions only, and merging their instances in the same approximation, the number of field definitions is typically too high to be reflected in the same number of Boolean variables. In [17], we have solved this problem by labelling fields as *non-*`null` whenever they are always initialised by all the constructors of their defining class, are never accessed before that initialisation and the program only stores a non-`null` values inside them (in constructors or methods). If this is the case, we are sure that these non-`null` fields always hold a non-`null` value when they are accessed. Since their identification requires nullness information about the values that are written into them, this new nullness analysis is performed in an *oracle-based* way: it is first assumed that all fields that are always initialised

by all the constructors of their defining class, before being read, are non-`null`. That is, we use an initial *oracle* that contains all such fields. A first nullness analysis is computed as in Subsection 3.1, exploiting such (optimistic) hypothesis. Then some fields are discarded from the oracle as potentially `null` whenever the last nullness analysis cannot prove that only non-`null` values are written into them. Hence a new nullness analysis is performed and the oracle further shrunk. This process is repeated until the oracle does not shrink anymore. This oracle-based technique, proved correct in [17], will be exploited also in the subsequent subsections, since the extra analyses that we will introduce there, for extra precision, need nullness information themselves.

This technique identifies *globally* non-`null` fields, since they stay non-`null`, forever, after their initialisation. Not surprisingly, it is computationally more expensive than that in Subsection 3.1. This is not only a consequence of the repeated execution of the nullness analysis, which is tamed by using caches. The actual complication is that it needs some form of definite aliasing information in order to identify the non-`null` fields. Namely, in order to spot the fields of `this` that are initialised by each constructor, it is not sufficient to look for assignments to `this.field`, since many assignments may occur, indirectly, by writing inside `x.field`, where `x` is a definite alias of `this`. This is particularly the case in Java bytecode, where, typically, the stack contains aliases of local variables (such as `this`). Moreover, helper functions are frequently used to help constructors build the state of `this` and definite aliasing is needed to track the information flow from constructors to helper functions. To that purpose, one needs to prove that the call to the helper function happens on a definite alias of `this`.

In conclusion, the cost of this analysis is higher, as well as its precision, than that of the analysis in Subsection 3.1 alone. For instance, for the program in Figure 1, JULIA reports now only 6 of the 8 warnings reported in Subsection 3.1 (we write inside square brackets the warnings that have been removed):

```
[line 14: call with possibly-null receiver to String.equals(Object)]
line 16: call with possibly-null receiver to String.equals(Object)
[line 19: call with possibly-null receiver to String.hashCode()]
line 24: call with possibly-null receiver to String.replace(char,char)
line 26: call with possibly-null receiver to String.replace(char,char)
line 31: read with possibly-null receiver of field inner
line 34: call with possibly-null receiver to C.toString()
line 35: call with possibly-null receiver to C.toString()
```

The following table shows time and precision for the analysis of the same applications analysed in Subsection 3.1 (the number of analysed methods does not change *w.r.t.* what is reported in that subsection). It reports, for comparison, inside brackets, some numbers as they were in that subsection:

program	time	derefs	access	update	call	warnings
OurTunes	31.94 (was 24.49)	95.09% (was 86.48%)	97.65%	100.00%	90.90%	173 (was 425)
EJE	43.34 (was 33.31)	98.25% (was 77.04%)	99.85%	100.00%	96.78%	74 (was 926)
JFlex	56.89 (was 39.05)	90.35% (was 81.29%)	85.23%	98.95%	93.71%	750 (was 1254)
utilMDE	64.17 (was 43.05)	95.11% (was 92.85%)	94.03%	100.00%	92.20%	195 (was 252)
AFU	52.37 (was 39.48)	96.04% (was 91.09%)	96.97%	100.00%	94.19%	231 (was 523)

This technique yields almost the same results as NIT, as the experiments in [18] show. NIT is in general faster but only slightly less precise than this technique, possibly because the analysis in Subsection 3.1 is completely context-sensitive, which is not the case for NIT.

In general, this technique, as well as those of the next subsections, is not correct for multi-threaded programs. This is because, although it assumes a field of an object o to be non-`null` only when it is always initialised by all constructors of the class of o before being read and only assigned non-`null` values in the program, it is possible, according to the Java memory model, that its initialisation is not immediately visible to other threads than that creating o. Hence, those threads might find `null` in the field [9]. NIT incurs in the same problem, since its proof of correctness considers a simplified memory model, which is not that of multi-threaded Java.

3.3 Locally non-`null` Fields

Some fields are not globally non-`null`. Namely, there are fields that are not initialised by all the constructors of their defining class, or that are accessed before being initialised, or that are assigned `null` somewhere in the program. For them, programmers often test their non-nullness before actually accessing them, with programming patterns such as that at lines 15 and 16 in Figure 1, where method `equals()` is called on field `inner` of `other` only if that field is found to contain a non-`null` value. In [18], a static analysis is coupled to that described in Subsection 3.2, which computes a set of definitely non-`null` fields *at a given program point*. This *local* non-nullness information is performed through a denotational, bottom-up analysis of the program, which is proved correct in [18]. In this analysis, the abstraction of a piece of code contains a set of definitely non-`null` fields for each variable in scope. These sets are implemented as bitmaps, for better efficiency and for keeping down the memory consumption.

Differently from Subsection 3.1, method calls might modify the approximation of the local variables of the caller here, which are sets of fields definitely non-`null` and hence possibly reset to `null` by the callee, since they are not globally non-`null` as in Subsection 3.2. This is a major complication, which requires a preliminary *sharing analysis* to infer which local variables might be affected by each method call. We perform that analysis with a denotational abstract interpretation, defined and proved correct in [16], implemented with Boolean formulas. For better precision, we couple it with a constraint-based *creation points* analysis, which provides the set of object creation statements in the program where the values bound to a given variable at a given program point or to a given field might have been created. A *constraint* is a graph whose nodes stand for the approximation of each variable at each program point and of each field and whose arcs bind those approximations, reflecting the program's information flow. A constraint-based analysis builds a large constraint for the whole program and lets information flow inside it. In our case, creation points flow along the arcs. Sets are, again, implemented as bitmaps. We use the creation points analysis at each method call to compare the creation points of each field

update instruction reachable from the code of the callee with the creation points of each local variable of the callee: if they do not intersect, the execution of the callee cannot affect the approximation for that variable. Note that sharing and creation points analysis are complementary: the former is context-sensitive (in our implementation), which is not the case for the latter that, however, lets us reason on the receiver of each single field update operations that might be performed during the execution of the callee. Moreover, constraint-based analyses allow a precise approximation of properties of the fields, which is not easy with denotational static analyses. For a formal definition of a constraint-based analysis and proof of correctness, see the case of rawness analysis in [19].

This technique increases the precision of the nullness analysis, but also its computational cost, mainly because of sharing and creation points analysis. For instance, JULIA, using this technique, reports only 5 of the 6 warnings reported in Subsection 3.2 for the program in Figure 1:

```
[line 16: call with possibly-null receiver to String.equals(Object)]
line 24: call with possibly-null receiver to String.replace(char,char)
line 26: call with possibly-null receiver to String.replace(char,char)
line 31: read with possibly-null receiver of field inner
line 34: call with possibly-null receiver to C.toString()
line 35: call with possibly-null receiver to C.toString()
```

The subsequent table shows times and precision of larger analyses with this technique:

program	time	derefs	access	update	call	warnings
`OurTunes`	102.91 (was 31.94)	98.47% (was 95.09%)	97.94%	100.00%	97.91%	54 (was 173)
`EJE`	128.66 (was 43.34)	98.66% (was 98.25%)	99.85%	100.00%	97.55%	56 (was 74)
`JFlex`	151.60 (was 56.89)	94.75% (was 90.35%)	86.44%	100.00%	97.49%	460 (was 750)
`utilMDE`	235.76 (was 64.17)	97.40% (was 95.11%)	94.03%	100.00%	96.34%	117 (was 195)
`AFU`	182.16 (was 52.37)	98.66% (was 96.04%)	97.19%	100.00%	98.27%	126 (was 231)

3.4 Full Arrays

The analyses described so far always assume that the elements of an array of references are potentially `null`. Hence, for instance, in Subsection 3.3 we still get the three warnings at lines 24, 26 and 31 of the program in Figure 1. For better precision, we need a static analysis that spots those arrays of references that only contain non-`null` elements. We call such arrays *full*. Examples of full arrays are the `args` parameter passed by the Java Virtual Machine to method `main()` or explicitly initialised arrays such as `Object[] arr = { a, b, c }`, provided it is possible to prove that `a`, `b` and `c` hold non-`null` values at run-time. Other examples are arrays iteratively initialised with loops such as that at lines $23-27$ of the program in Figure 1. Those loops must provably initialise *all* elements of the array with definitely non-`null` values. Note that this property is relatively complex to prove, since the initialisation of the array elements can proceed in many ways and orders. For instance, in Figure 1, each iteration initialises two elements of `ts` at a time. Moreover, the initialising loop might end at the final element of the array (or, downwards, at the first) and this might be expressed

through a variable rather than the `.length` notation as in Figure 1. Furthermore, the loop is often a `for` loop, but it might also be a `while` loop or a `do...while` loop (since we analyse Java bytecode, there is no real difference between those loops). In general, it is hence unrealistic to consider all possible initialisation strategies, but some analysis is needed, that captures the most frequent scenarios. Moreover, a *semantical* analysis is preferable, rather than a weak syntactical pattern matching over the analysed code. In our implementation, definite aliasing information is used to prove that `ts.length` (in Figure 1) is the size of an array that is definitely initialised inside the body of the loop at lines $23 - 27$. If this is the case, a denotational analysis based on regular expressions builds the *shape* of the body of this loop: if this shape (*i.e.,* regular expression) looks as an initialisation of some variable `i` (the same compared to `ts.length`) to 0, followed by an alternation of array stores at `i` of non-`null` values and unitary increments of `i`, then the array is assumed to hold non-`null` values at the natural exit point of the loop, but not at exceptional exit points*, i.e.,* those that end the loop if it terminates abnormally because of some exception. We are currently working at improving this analysis, by considering more iteration strategies in the initialising loop. We also plan to consider the case when the array is held in a field rather than in a local variable.

Full arrays can be copied, stored into fields or passed as parameters to methods. Hence, we use a constraint-based static analysis that tracks the flow of the arrays in the program. During this analysis, we consider that full arrays may lose their property of being full as soon as an array store operation is executed, which writes a possibly `null` value. For better precision, we exploit the available static type information and use the creation points analysis, the same of Subsection 3.3, to determine the variables, holding full arrays, that might be affected by each array store operation.

It must be said that JULIA is able to reason on single array elements as well, if they are first tested for non-nullness and then dereferenced. For instance, it knows that `args[i]` does not hold `null` immediately after line 24 in Figure 1, even if it were not able to prove that `args` is full. This is because `args[i]` has been dereferenced there, so it cannot hold `null` immediately after (or otherwise an exception would interrupt the method). Of course, this requires JULIA to prove that the constructor of `C` and method `replace()` do not write `null` into `args`, by using sharing and creation points analysis. We have embedded this local array non-nullness analysis inside the technique of Subsection 3.3, since local non-nullness of fields and of array elements can be considered in a uniform way.

The improvements induced by all these approximations of the nullness of the array elements increase the precision of the nullness analysis. For instance, the analysis with JULIA of the program in Figure 1, using these extra techniques, yields only 2 of the 5 warnings reported in Subsection 3.3 now:

```
[line 24: call with possibly-null receiver to String.replace(char,char)]
[line 26: call with possibly-null receiver to String.replace(char,char)]
[line 31: read with possibly-null receiver of field inner]
line 34: call with possibly-null receiver to C.toString()
line 35: call with possibly-null receiver to C.toString()
```

These techniques increase precision and time of the nullness analysis of larger applications as well:

program	time	derefs	access	update	call	warnings
OurTunes	160.63 (was 102.91)	98.47% (was 98.47%)	97.94%	100.00%	97.91%	54 (was 54)
EJE	164.58 (was 128.66)	98.83% (was 98.66%)	100.00%	100.00%	97.81%	47 (was 56)
JFlex	173.02 (was 151.60)	95.27% (was 94.75%)	86.44%	100.00%	97.59%	409 (was 460)
utilMDE	298.88 (was 235.76)	97.90% (was 97.40%)	94.03%	100.00%	97.24%	81 (was 117)
AFU	222.24 (was 182.16)	98.86% (was 98.66%)	97.19%	100.00%	98.63%	107 (was 126)

3.5 Collection Classes

The Java programming language comes with an extensive library. Some prominent library classes are the *collection classes*, such as `Vector`, `HashSet` and `HashMap`. They are largely used in Java programs, but complicate the nullness analysis, since most of their instances are allowed to contain `null` elements, keys or values. As a consequence, there is no syntactical guarantee, for instance, that the iterations at lines 34 and 35 in Figure 1 happen over non-`null` elements only. This is why JULIA signals two warnings there, up to the technique of Subsection 3.4. The techniques described up to now do not help here since, for instance, the elements of a hashset are stored inside a backing hashmap, which contains an array of key/value *entries*. The technique of Subsection 3.2 might be able to prove that the field holding the key or that holding the value in all entries are globally non-`null`, but this is not the case: in Figure 1 hashmaps are not only used inside the two hashsets `s1` and `s2`, but also internally by the Java libraries themselves. In some of those uses, not apparent from Figure 1, `null` is stored (or seems to JULIA to be stored) as a value or key in a hashmap. In any case, the technique of Subsection 3.2 would be very weak here because it *flattens* all collections into the same, global abstraction for the nullness of the fields of the entries inside a hashmap: a program may use more hashsets or hashmaps (as is the case in Figure 1) and it is important, for better precision, to distinguish the collections possibly having `null` among their elements from those that only contain non-`null` elements: in Section 2 we have seen that commenting out line 31 introduces a warning at line 35 but not at line 34. The technique of Subsection 3.3 does not help either. The method `get()` of a hashmap or the method `next()` of an iterator over the keys or values of a hashmap does not check for the non-nullness of the field of the hashmap entry holding the returned value (as method `equals()` in Figure 1 does for field `inner`) nor assigns it to a definitely non-`null` value before returning its value.

To prove that the two calls to `toString()` in Figure 1 happen on a non-`null` variable `t`, we use a new technique. Namely, we let JULIA prove that `s1` and `s2` are sets of non-`null` elements. To that purpose, we have developed a constraint-based analysis that approximates each local variable at each given program point and each field with a flag, stating if the value of the variable or field is an instance of a collection class that does not contain `null`. Whenever a method is called, such as `HashSet.add()`, which might modify the flag of some variable, the analysis checks if it can prove that the call adds a non-`null` value to the collection. If this is not the case, the affected variables and fields lose the flag

stating that they do not contain `null` elements. In order to over-approximate the variables and fields affected, we use sharing and creation points analysis, as in Subsection 3.3.

The property of containing non-`null` elements only is propagated by the constraint-based analysis, following variable assignments, method calls and return. Also, if an iterator is built from a collection that does not contain `null` elements, then we flag that iterator as iterating over non-`null` elements only. If a possibly `null` element is added, later, to that iterator, it will lose its flag, and this will happen also to the variables holding the backing collection.

Our implementation of this analysis currently considers around 20 collection classes but we plan to consider more in the future. In order to simplify the addition of new classes, the analysis consults some Java annotations that we have written for the methods of the collection classes. Adding more classes is hence a matter of writing new annotations. In particular, one does not need to modify the analysis itself.

By using this technique, the nullness analysis with JULIA of the program in Figure 1 issues no warning, instead of the 2 of Subsection 3.4. The following table shows that this technique improves the precision of the analysis of larger applications as well:

program	time	derefs	access	update	call	warnings
`OurTunes`	162.82 (was 160.63)	99.01% (was 98.47%)	99.11%	100.00%	98.58%	38 (was 54)
`EJE`	157.61 (was 164.58)	99.07% (was 98.83%)	100.00%	100.00%	98.26%	36 (was 36)
`JFlex`	176.61 (was 173.02)	98.66% (was 95.27%)	99.14%	100.00%	97.80%	118 (was 409)
`utilMDE`	257.01 (was 298.88)	98.76% (was 97.90%)	100.00%	100.00%	97.78%	58 (was 81)
`AFU`	231.42 (was 222.24)	99.05% (was 98.86%)	98.31%	100.00%	98.72%	91 (was 107)

The time for analysing `EJE` and `utilMDE` has actually decreased *w.r.t.* Subsection 3.4, since the extra precision has accelerated the convergence of the oracle-based nullness analysis. Three of the warnings issued for `EJE` are actual `null`-pointer bugs of that program. The others are false alarms.

This analysis and that of Subsection 3.4 are strictly intertwined. The analysis in Subsection 3.4 must first determine the program points that initialise a full array. Then, the property of being full is propagated across the program, together with the same property for collection classes. The *fullness* for arrays and collections actually interact: if a collection is full, then its `toArray()` methods return a full array.

4 Construction of the Annotation File

In Section 3, we have seen the different phases of the nullness analysis of JULIA and how they provide different levels of precision for software verification*, i.e.,* different numbers of warnings. But nullness analysis can also be used to build an *annotation* of the program under analysis, reporting which fields, parameters or return values of methods might hold `null` at run-time. This is interesting to the programmer, is useful for documentation and can also be seen as a *standard description* of the nullness behaviour of the program. As such, it can be imported and exported between different tools for software analysis.

We did not devise our own annotation language for nullness, but used one that has been developed for the *Checker Framework* for Java [14]. The latter is a generic tool for software verification, based on type checking. Types can be specified and written into Java source code as Java annotations. The system type-checks those types and reports inconsistencies. The checker framework contains a type-system for nullness and is bundled with a tool (*file annotation utilities*) that imports a succinct description of the nullness behaviour of the program (a *jaif file*, in their terminology) into Java source code. This is perfect for JULIA: since our tool analyses Java bytecode, it has no direct view of the source code, but it can generate a jaif file which is then importable into source code, by using the file annotation utilities.

```
class C:
  field inner: @Nullable
  method <init>(Ljava.lang.String;LC;)V:
    parameter #1: @Nullable
  method equals(Ljava.lang.Object;)Z:
    parameter #0: @Nullable
```

Fig. 2. Jaif file generated by JULIA for the program in Figure 1

Figure 2 reports the jaif file generated by JULIA for the program in Figure 1. The default hypothesis is that everything is non-`null`, so that only possible nullness must be explicitly reported in the file. Hence Figure 2 says, implicitly, that field `name` in Figure 1 is non-`null` and that the constructor of class `C` (method `<init>`) always receives a non-`null` value as its first parameter (numbered as 0). Jaif files report also the nullness of the elements of an array of references or of an object of a collection class. Hence, since nothing is explicitly reported in Figure 2 about method `main()`, that figure tells us that `main()` in Figure 1 is always called with a parameter which is a non-`null` array of non-`null` strings.

For another example, consider the program in Figure 3 and the corresponding jaif file generated by JULIA, shown in Figure 4. This jaif file tells us that JULIA has been able to conclude that every call to methods `main()` and `first()` happens with a non-`null` argument of non-`null` strings, as well as every call to the constructor of `Test`. The same is not stated for method `second()`, since in some cases `null` is passed to `second()` for `x`. Moreover, the calls to method `inLoop()` are reported to happen with a non-`null` argument `x`, but whose elements might be `null` (see the annotation `inner-type 0: @Nullable`). This is true, since `inLoop()` is called with an array `s` as actual parameter that is not always full at the point of call. Field `g` is reported as possibly-`null` and there are actual cases when it contains `null` at run-time. Field `map` is reported as non-`null` but its `inner-type 1` is possibly-`null`: this is the value component of the map. Instead, its key component is definitely non-`null`, since there is no annotation for `map` about `inner-type 0` and the default is non-`null`.

The Checker Framework benefits from information about which values are *raw*, that is, are objects whose non-`null` fields might not have been assigned yet. This is important since, when a value is raw, the type-checker of the Framework correctly assumes its non-`null` fields to be potentially `null`. This is the case of `this` at the beginning of the constructor of `Test`, since its fields `map` and `f` (reported to be non-`null` in the jaif file) have not been yet initialised there. Hence also `this` inside `inLoop()` and `first()` is raw. But `this` inside `second()` is *not* raw since all its non-`null` fields have been already initialised when `second()` is called. In the jaif file in Figure 4, rawness information is reported with the `@Raw` annotation, applied to the receiver `this`, although, in more complex examples, we might find it reported for fields, parameters and return types as well. JULIA never reports it for the receiver of a constructor (such as that of `Test`), since it is the default there.

JULIA computes rawness information with a constraint-based *rawness analysis*, performed after the nullness analysis of Section 3. Rawness analysis is implemented as a constraint-based static analysis, where each variable at each given program point and each field is approximated with the set of its non-`null` fields that have been definitely initialised there. Those sets flow along the constraint,

```
import java.util.*;
public class Test {
  private Map<String, Object> map;
  private String f, g;
  public static void main(String[] args) {
    new Test(args);
  }
  private Test(String[] args) {
    map = new LinkedHashMap<String, Object>();
    String[] s = new String[args.length / 2];
    for (int i = 0; i < s.length; i++) {
      map.put(s[i] = args[i * 2], null);
      inLoop(s);
    }
    System.out.println(first(s));
    f = "name";
    if (args.length > 5)
      g = "surname";
    second(g);
  }
  private void inLoop(String[] x) {}
  private String first(String[] s) {
    return s.length > 0 ? s[0] : null;
  }
  private void second(String x) {}
}
```

Fig. 3. A simple Java program

```
class Test:
  field g: @Nullable
  field map:
    inner-type 1: @Nullable
  method second(Ljava.lang.String;)V:
    parameter #0: @Nullable
  method first([Ljava.lang.String;)Ljava.lang.String;:
    return: @Nullable
    receiver: @Raw
  method inLoop([Ljava.lang.String;)V:
    parameter #0:
      inner-type 0: @Nullable
    receiver: @Raw
```

Fig. 4. The jaif file generated by JULIA for the program in Figure 3

reflecting assignments, parameter passing and method returns, and are enlarged at field assignments. A formal definition and a proof of correctness of this analysis are contained in [19].

Although the jaif files built by JULIA are correct, this does not mean that their application to the source code type-checks *w.r.t.* the type-checker of the Checker Framework. This is because the techniques of Section 3 are data-flow, much different from those applied by that type-checker. Nevertheless, we have been working at making JULIA and the Checker Framework closer.

The DAIKON tool is also able to generate jaif files, but they are only likely correct. NIT generates jaif files also and computes some rawness information during its nullness analysis. Nevertheless, it does not dump the rawness information into the jaif file. Since NIT is less precise than the full-featured nullness analysis of JULIA, the generated jaif files are less precise too.

5 Conclusion

We have described and experimented with the nullness analysis implemented inside JULIA. It is correct and very precise, with a cost in time which is still acceptable for off-line analyses (a few minutes). Note that the only other correct nullness analysis for Java is that in [11], whose precision is similar to that of Subsection 3.2, as experimentally validated in [17].

Our nullness analysis is the composition of many static analyses, denotational and constraint-based. In general, denotational analyses provide context-sensitivity, while constraint-based analyses provide better support for the analysis of fields. In terms of software engineering, JULIA contains two implementations of those classes of analysis, from which all concrete static analyses are derived by subclassing. This has simplified the development of new analyses and allowed the optimisation of the shared components and their debugging.

Our work is not finished yet. First of all, we aim at making JULIA more scalable, up to including the whole `javax.*` hierarchy in the analysis. This will

benefit the precision of the results, particularly for the analysis of those applications that make use of the Swing graphical library. We are also working at making the results correct for multi-threaded applications. In this direction, we are developing a static analysis that identifies those fields that are only accessed by the thread that has assigned them. If this is the case, all techniques from Section 3 are correct for them. For the other fields (hopefully not many) a worst-case assumption will be made. The web interface of JULIA must also be improved in order to provide better graphical effects and more feedback to the user. It is available at the address `http://julia.scienze.univr.it`, where the reader can test the tool without any local installation.

References

1. Eclipse.org Home, `http://www.eclipse.org`
2. Jlint, `http://artho.com/jlint`
3. Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers 35(8), 677–691 (1986)
4. Cok, D.R., Kiniry, J.: Esc/Java2: Uniting ESC/Java and JML. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 108–128. Springer, Heidelberg (2005)
5. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Proc. of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1977), Los Angeles, California, USA, pp. 238–252. ACM, New York (1977)
6. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon System for Dynamic Detection of Likely Invariants. Science of Computer Programming 69(1-3), 35–45 (2007)
7. Flanagan, C., Leino, K.R.M.: Houdini, an Annotation Assistant for ESC/Java. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 500–517. Springer, Heidelberg (2001)
8. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended Static Checking for Java. In: Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2002), Berlin, Germany. SIGPLAN Notices, vol. 37(5), pp. 234–245. ACM, New York (May 2002)
9. Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., Lea, D.: Java Concurrency in Practice. Addison-Wesley, Reading (2006)
10. Hovemeyer, D., Pugh, W.: Finding More Null Pointer Bugs, but Not Too Many. In: Das, M., Grossman, D. (eds.) Proc. of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2007), San Diego, California, USA, pp. 9–14. ACM, New York (June 2007)
11. Hubert, L., Jensen, T., Pichardie, D.: Semantic Foundations and Inference of non-null Annotations. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 132–149. Springer, Heidelberg (2008)
12. Lindholm, T., Yellin, F.: The Java™ Virtual Machine Specification, 2nd edn. Addison-Wesley, Reading (1999)
13. Palsberg, J., Schwartzbach, M.I.: Object-oriented Type Inference. In: Proc. of the 6th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 1991), Phoenix, Arizona, USA. ACM SIGPLAN Notices, vol. 26(11), pp. 146–161. ACM, New York (November 1991)

14. Papi, M.M., Ali, M., Correa, T.L., Perkins, J.H., Ernst, M.D.: Practical Pluggable Types for Java. In: Ryder, B.G., Zeller, A. (eds.) Proc. of the ACM/SIGSOFT 2008 International Symposium on Software Testing and Analysis (ISSTA 2008), Seattle, Washington, USA, pp. 201–212. ACM, New York (July 2008)
15. Rutar, N., Almazan, C.B., Foster, J.S.: A Comparison of Bug Finding Tools for Java. In: Proc. of the 15th International Symposium on Software Reliability Engineering (ISSRE 2004), Saint-Malo, France, pp. 245–256. IEEE Computer Society, Los Alamitos (November 2004)
16. Secci, S., Spoto, F.: Pair-Sharing Analysis of Object-Oriented Programs. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 320–335. Springer, Heidelberg (2005)
17. Spoto, F.: Nullness Analysis in Boolean Form. In: Proc. of the 6th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2008), Cape Town, South Africa, pp. 21–30. IEEE Computer Society, Los Alamitos (November 2008)
18. Spoto, F.: Precise null-Pointer Analysis. Software and Systems Modeling (to appear, 2010)
19. Spoto, F., Ernst, M.: Inference of Field Initialization. Technical Report UW-CSE-10-02-01, University of Washington, Department of Computer Science & Engineering (February 2010)
20. Spoto, F., Mesnard, F., Payet, E.: A Termination Analyser for Java Bytecode Based on Path-Length. ACM Transactions on Programming Languages and Systems (TOPLAS) 32(3) (March 2010)