Challenges for Refinement and Composition of Instrumentations: Position Paper

Danilo Ansaloni[1], Walter Binder[1], Christoph Bockisch[2], Eric Bodden[3], Kardelen Hatun[2], Lukáš Marek[4], Zhengwei Qi[5], Aibek Sarimbekov[1], Andreas Sewe[3], Petr Tůma[4], and Yudi Zheng[5]

[1] University of Lugano, Switzerland
{danilo.ansaloni,walter.binder,aibek.sarimbekov}@usi.ch
[2] University of Twente, The Netherlands
c.m.bockisch@cs.utwente.nl, k.hatun@ewi.utwente.nl
[3] Technische Universität Darmstadt, Germany
eric.bodden@ec-spride.de, andreas.sewe@cased.de
[4] Charles University, Czech Republic
{lukas.marek,petr.tuma}@d3s.mff.cuni.cz
[5] Shanghai Jiao Tong University, China
{qizhwei,zheng.yudi}@sjtu.edu.cn

Abstract. Instrumentation techniques are widely used for implementing dynamic program analysis tools like profilers or debuggers. While there are many toolkits and frameworks to support the development of such low-level instrumentations, there is little support for the refinement or composition of instrumentations. A common practice is thus to copy and paste from existing instrumentation code. This, of course, violates well-established software engineering principles, results in code duplication, and hinders maintenance. In this position paper we identify two challenges regarding the refinement and composition of instrumentations and illustrate them with a running example.

Keywords: Instrumentation, composition, aspect-oriented programming, domain-specific languages.

1 Introduction

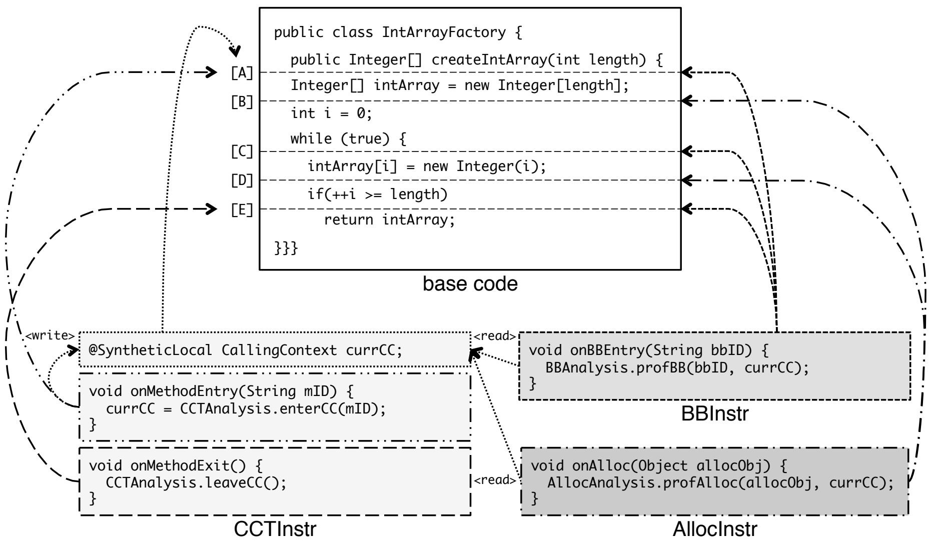
Many dynamic program analyses, including tools for profiling, debugging, testing, program comprehension, and reverse engineering, rely on code instrumentation. Such tools are usually implemented with toolkits that allow for the careful low-level optimization of the inserted code. While this low-level view is needed to keep the overhead incurred by dynamic program analyses low, it currently brings with it a lack of support for refining and composing the resulting instrumentations. Code duplication, caused either by copy and paste or by the reimplementation of common instrumentation tasks, is therefore a common code smell of many instrumentation-based tools; it is known to be error-prone and to hinder software maintenance. This is all the more problematic, as errors in low-level code are notoriously hard to find.

Before we discuss the challenges arising from the refinement and composition of instrumentations, we first define our terminology and context. An *instrumentation* selects certain sites in the code of a given base program—so-called *instrumentation sites*—and inserts code to be executed whenever the control flow reaches these sites. The *inserted code* must not change the semantics of the base program; it must complete after a finite number of instructions without throwing any exception into the base program and may read but not write any memory location accessed by the base program. Inserted code must thus resort to dedicated memory locations to pass data between different instrumentation sites. For example, local variables invisible to the base program may be used to pass data between several instrumentation sites within the same method body. Likewise, thread-local variables or global variables may be used to pass data between instrumentation sites in different methods. The inserted code typically invokes analysis methods, e.g., to update a profile. We call the classes defining those methods the *runtime classes* of the analysis.

Suitable refinement and composition mechanisms for instrumentations need to address the following two general challenges:

1. *Specification and enforcement of constraints*: Instrumentations usually fail to state important assumptions that are crucial for the instrumentations' correctness in general and when refining or composing instrumentations in particular. Such assumptions may constrain the following.
   (a) *Instrumentation sites*: The selection of sites by the different instrumentations must be consistent; different instrumentations, e.g., may need to refer to the same part of a program, regardless of whether they share common instrumentation sites or not.
   (b) *Instrumentation ordering*: Composing instrumentations often requires defining ordering constraints, not only for the instrumentations as a whole but possibly even for each instrumentation site that they target.
   (c) *Data passing*: Instrumentations may declare variables to pass data between instrumentation sites. Each of these variables has to be initialized by one instrumentation before it can be read by another.
2. *Avoiding hard-coded dependencies*: Usually, the inserted code has hard-coded dependencies on specific runtime classes. Such dependencies typically resemble invocations of static methods or constructors of runtime classes in the inserted code. When refining or composing instrumentations, these dependencies may need to be changed to use different runtime classes.

As original contribution, in this position paper we study the aforementioned challenges regarding refinement and composition of instrumentations and illustrate them with a running example (Section 2). Section 3 discusses related work and Section 4 concludes.



```
public class IntArrayFactory {
  public Integer[] createIntArray(int length) {
[A]
  Integer[] intArray = new Integer[length];
[B]
  int i = 0;
  while (true) {
[C]
    intArray[i] = new Integer(i);
[D]
    if(++i >= length)
[E]
      return intArray;
}}}
```

```
@SyntheticLocal CallingContext currCC;

void onMethodEntry(String mID) {
  currCC = CCTAnalysis.enterCC(mID);
}

void onMethodExit() {
  CCTAnalysis.leaveCC();
}
```

```
void onBBEntry(String bbID) {
  BBAnalysis.profBB(bbID, currCC);
}
```

```
void onAlloc(Object allocObj) {
  AllocAnalysis.profAlloc(allocObj, currCC);
}
```

Fig. 1. Instrumentation sites for a composition of three instrumentations: calling context profiling (*CCTInstr*), basic block profiling (*BBInstr*), and object allocation profiling (*AllocInstr*)

2 Challenges

There are several challenges that make refinement and composition of instrumentations difficult. To explain these challenges, we first introduce a running example in which three common instrumentations are composed. Next, we motivate the need for specifying and enforcing instrumentation constraints. Finally, we consider the problem of hard-coded dependencies from inserted code to specific runtime classes.

2.1 Motivating Example

Figure 1 illustrates our motivating example: a composition of three instrumentations (pseudo-code) applied to some base program. While the inserted code is intentionally kept simple, the three instrumentations have interactions that resemble those of complex, real-world analyses.

CCTInstr. The *CCTInstr* analysis maintains a Calling Context Tree (CCT) [1], i.e., a data structure that can be used to store dynamic metrics separately for each individual calling context. To efficiently expose a reference to the CCT representation of the current calling context to the code inserted into the same base-program method by the other two instrumentations, *CCTInstr* declares the synthetic local variable *currCC* that will be mapped to a local variable in each instrumented method. *CCTInstr.onMethodEntry(...)* represents the code inserted at instrumentation site [A]. We assume the instrumentation framework provides some context information; the string *mID* identifies the base-program method to be instrumented. The runtime class *CCTAnalysis* keeps track of the current calling context for each thread and maintains the CCT. *CCTAnalysis.enterCC(...)* updates the current thread's calling context on method entry and returns a reference to it, which is then stored in *currCC*. *CCTAnalysis.leaveCC()*, inserted at instrumentation site [E], updates the current thread's calling context on method completion.

BBInstr. The *BBInstr* analysis counts how often each basic blocks of code is executed. The code of *BBInstr.onBBEntry(...)* is inserted at the instrumentation sites [A], [C], and [E]. As context information provided by the instrumentation framework, *bbID* identifies the executed basic block. *BBAnalysis.profBB(...)* updates a counter corresponding to *bbID* in the current calling context (*currCC*).

AllocInstr. The *AllocInstr* analysis profiles object allocations. The code of *AllocInstr.onAlloc(...)* is inserted at the instrumentation sites [B] and [D]. As context information, *allocObj* refers to the allocated object. *AllocAnalysis.profAlloc(...)* updates an object allocation counter in the current calling context (*currCC*).

2.2 Specification and Enforcement of Constraints

We now describe the implicit constraints that must be respected to preserve correctness when refining or composing the instrumentations illustrated in Fig. 1.

Instrumentation Sites. To ensure soundness of the CCT, *CCTInstr* must be comprehensively applied to all classes. In contrast, restricting the scope of *BBInstr* and *AllocInstr* does not impair the correctness of the (subset of) collected data. To reduce the runtime overhead of the analysis, it may even be desirable to restrict expensive instrumentations like *BBInstr* to a subset of the base-program classes. To ensure the consistency of one's measurements, however, it is likewise desirable to ensure that *BBInstr* and *AllocInstr* are applied to the same selection of classes.

Instrumentation Ordering. A hard constraint of the instrumentations illustrated in Fig. 1 concerns the synthetic local variable *currCC* used to share the current calling context among the instrumentations. Even if not explicitly stated, both *BBInstr.onBBEntry(...)* and *AllocInstr.onAlloc(...)* expect this variable to be initialized by *CCTInstr.onMethodEntry(...)*. That is, if multiple instrumentations insert code on method entry, *CCTInstr.onMethodEntry(...)* must be applied first. As a consequence, at instrumentation site [A], *CCTInstr.onMethodEntry(...)* has to be inserted before *BBInstr.onBBEntry(...)*. For the other instrumentation sites, no particular ordering is required.[1] Unfortunately, ordering constraints are often

[1] Intuitively, at instrumentation site [E], *CCTInstr.onMethodExit()* should be inserted after *BBInstr.onBBEntry(...)*. But as *CCTInstr.onMethodExit()* does not modify the synthetic local variable *currCC*, the insertion order at [E] does not matter.

implicit in the implementation of complex instrumentations. Therefore, composing instrumentations usually requires in-depth knowledge of their implementation to avoid violating any implicit ordering constraint.

Data Passing. Efficient communication between different composed instrumentations is often necessary to reduce runtime overhead. However, the declaration of the name and the type of variables used for such communication is usually hard-coded in the instrumentations; thus, it is difficult to refine them without in-depth knowledge of all implementation details. In our example, both *BBInstr* and *AllocInstr* expect a variable of type *CallingContext* named *currCC*. This constraint hinders composition of instrumentations, as it is often necessary to update the code of some instrumentations to communicate through the same variables.

2.3 Hard-Coded Dependencies

Frequently, instrumentations use static method calls to access runtime classes, often in a desire to avoid the runtime overhead associated with virtual methods. For example, the instrumentations illustrated in Fig. 1 include static calls to methods in the runtime classes `CCTAnalysis`, `BBAnalysis`, and `AllocAnalysis`. These dependencies constrain refinement of runtime classes, as static methods cannot be overridden. Refactoring runtime classes to adhere to the singleton pattern helps mitigate the problem, but the static method returning the singleton instance cannot be refined so as to return an instance of a refined runtime class. To address this issue, mechanisms for dependency injection are needed.

3 Related Work

In the past, both low-level frameworks and aspect-oriented approaches have been used for various instrumentation tasks. While the former are typically more expressive and lead to faster code, the latter may offer more powerful refinement and composition mechanisms. In the following text, we compare the properties of both kinds of approaches. For brevity, we limit the discussion to solutions for the Java Virtual Machine, as it is a widely used deployment platform and has been targeted by a large body of related work.

3.1 Instrumentation Frameworks

Bytecode Manipulation Frameworks. Low level bytecode manipulation frameworks like ASM[2], BCEL[3], Javassist [5], or Soot [15] support the direct generation or transformation of arbitrary bytecode. While they offer the maximally possible control over bytecode instrumentation, composition of instrumentations is not directly supported. When instrumentations are to be developed separately,

[2] See `http://asm.ow2.org/`
[3] See `http://commons.apache.org/bcel/`

they can thus only be composed by applying them sequentially. In this case, however, each instrumentation receives the bytecode resulting from the previous instrumentations as input. Thus, later instrumentations generally cannot distinguish between the original code and code inserted by earlier instrumentations.[4] As a consequence, controlling the composition of instrumentations becomes infeasible. This is also true of the Scala library Mnemonics [11] which is slightly less low-level than the aforementioned frameworks; by exploiting Scala's type system, it ensures that only certain well-formed, type-safe bytecode sequences can be generated.

RoadRunner. Flanagan and Freund [6] propose a framework for composing different small and simple analyses for concurrent programs. Each analysis can stand on its own, but by composing them one can obtain more complex ones: each dynamic analysis is essentially a filter over event streams, and filters can be chained. Per program run, only one chain of analyses can be specified. Thus, it is generally not possible to combine arbitrary analyses; for example, two analyses that filter (e.g., suppress) events in an incompatible way cannot be combined.

DiSL. DiSL [9, 18] is a domain-specific language for instrumentation. While it offers high-level abstractions to ease the development of instrumentations, it also gives the programmer fine-grained control over the inserted code. However, DiSL lacks refinement and composition mechanisms.

3.2 Aspect-Oriented Approaches

Aspect-oriented programming (AOP) is frequently used as a high-level, language-based approach to implementing analyses. An analysis roughly maps to one or more aspects; then the instrumentations of the analysis correspond to pairs of pointcuts and advice. Pointcuts select the instrumentation sites and advice define the inserted code, in this analogy. But as aspect-oriented languages typically focus on high-level interaction with the program execution, the available instrumentation sites and context accessible in inserted code is limited. Nevertheless, the requirements for the re-use and composition of aspects are similar to those of analyses and their instrumentations. In the following text, we thus briefly discuss selected aspect-oriented languages and their features with respect to (1) constraining instrumentation sites (*scope*), to (2) specifying the *order* of aspects at shared instrumentation sites, and to (3) *sharing* structure and implementation between aspects (e.g., to realize data sharing).

AspectJ. In the past, the AspectJ language [8] (or derivatives) has been used for implementing dynamic analyses; often alternative compilers are used for this purpose, such as MAJOR [4, 16, 17] and MAJOR2 [10, 13] which allows instrumenting the Java class library unlike the standard AspectJ weaver. In AspectJ,

[4] Soot provides a way to tag statements and bytecode instructions. However, there are no guidelines that would govern or enforce a principled use of this mechanism.

aspects extend classes with pointcuts and advice.[5] Pointcuts are boolean expressions whose operands are again (possibly primitive) pointcuts. They can also be named and then referenced from multiple other pointcuts.

Scope: In an abstract aspect, pointcut expressions may also refer to *abstract* named pointcuts; e.g., the pointcut expression *scope() && call(*.new(..))* selects all constructor call sites at which also the pointcut *scope()* matches, which can be declared as an abstract pointcut. Abstract aspects can be extended whereby concrete expressions must be provided for abstract pointcuts. In this way, an analysis implemented in an aspect can be re-used while the scope for applying the analysis may be reduced, e.g., by specifying an expression for the *scope()* pointcut that only matches within a certain package.

Order: Ordering constraints between instrumentations (i.e., pointcut-advice pairs) can be imposed by explicitly declaring the precedence of entire aspects, possibly external to the aspects in question.

Sharing: Extending an abstract aspect is the only means to code re-use supported by AspectJ. The sub-aspect has to concretize abstract pointcuts and can override virtual methods.

CaesarJ. With respect to the pointcut-advice mechanism, aspects in CaesarJ [2] are very similar to those of AspectJ. Two extensions are relevant for the scoping and sharing issue of this paper, however.

Scope: Additionally, it allows to programmatically deploy and undeploy aspects and to limit their activation to certain threads or objects, thereby refining the scope of an aspect.

Order: CaesarJ provides the same mechanism for declaring aspect precedence as AspectJ.

Sharing: The CaesarJ language extends the Java type system with dependent types, i.e., types which are properties of (aspect) instances. Thus, expressions like *this* can be used in type declarations and the compiler can verify that covariant types are used together consistently.

JAsCo. The JAsCo language [12] extends Java beans with so-called hooks to aspect beans. Similar to AspectJ and CaesarJ, it offers a pointcut-advice mechanism, however, in JAsCo it is composed of several individual concepts which improve re-usability and configurability.

Scope: A hook is similar to an inner class which defines a context-independent pointcut in its constructor. Being context-independent, the pointcut expression does not refer to actual methods but rather refers to the constructor's parameters, which are made concrete upon hook instantiation. Besides its constructor and advice, a hook can contain the possibly abstract method *isApplicable*, which furthermore refines the hook's scope. Hooks are instantiated and deployed using so-called connectors which supply concrete method patterns to the hooks' constructors and implement any abstract *isApplicable* methods.

[5] The extension with inter-type declarations is out of the scope for this paper.

Order: Connectors not only specify the hooks' scope, they also fix their order either explicitly or implicitly through programmatic combination strategies. The latter can also be used to conditionally remove applicable hooks to mimic overriding.

Sharing: JAsCo allows to override hook methods on a per-advised-object basis. In this way aspect beans can be re-used and extended on different contexts.

HyperJ. The HyperJ [14] approach attempts to decompose a program along different dimensions. For each dimension, a partial program, called a hyperslice, is written which must be declaratively complete; functionality not provided but required by a hyperslice must be declared as abstract methods. A control file then governs the composition of hyperslices, which configures how the methods of the hyperslices are matched and merged.

Scope: One matching strategy is to match methods with the same names, but it is possible to compensate mismatches which especially occur when hyperslices are developed independently.

Order: The merging strategy is similarly configurable; if all except one of the definitions are abstract, the merging is trivial. For more than one concrete, matching unit, HyperJ provides merging strategies such as overriding or aggregating the result of the separate unit.

Sharing: In HyperJ, abstract methods, together with appropriate matching and merging strategies, can be used to share functionality among hyperslices.

Composition Filters. The Composition Filters Model [3] is based on the concepts of filters which are applied to method invocations.

Scope: A filter selects invocations based on the method's name and signature and it can perform additional actions or influence the execution of the target method. Filtermodules group filters and can declare data fields holding shared values. They can declare parameters to be used, e.g., for the type of fields or in the filters' expressions for selecting method invocations. A superimposition block has to be declared, possibly in a separate module, which deploys filtermodules on a set of types and provides concrete values for the parameters.

Order: For jointly superimposed filtermodules, a partial order can be specified. Other relations like overriding between filters can be declared in a similar way.

Sharing: A filtermodule can be superimposed multiple times on different type sets and with different parameter values.

Framed Aspects. In the Frames approach, so-called tags may be inserted into the code. For these tags a configuration file can then provide an application-specific replacement.

Scope: The Framed Aspects approach [7] allows the insertion of tags into aspects. Tags can be used, e.g., in place of type or method names, expressions or in patterns used by pointcuts. Thus, the scope of aspects can be refined in the configuration file.

Order: The Framed Aspects approach is independent of a concrete aspect language. The approach does not by itself offer a mechanism for specifying the order between aspects, but it inherits the mechanisms of the underlying AOP languages.

Sharing: Besides adopting the features for re-using aspect offered by the underlying language, the Framed Aspects approach itself provides re-usable aspect templates. The templates can be concretized through the configuration file.

Commonly, the presented approaches allow explicit declaration of precedence among aspects which can address ordering constraints between instrumentations. AspectJ and CaesarJ support re-use (i.e., sharing) and configuration (i.e., scoping) through inheritance and overriding; in HyperJ code can be re-used by composition while the composition specification (the control file) cannot be re-used at all. JAsCo and Composition Filters support black-box re-use and configuration through parameterization, however, the languages only support parameters for a limited set of constructs. These two approaches and CaesarJ additionally provide control over scoping by means of their programmatic deployment or superimposition features. Framed Aspects supports parameters in a more flexible way, but parameterization requires a fair amount of variability analysis to determine extension points of a planned feature. Since it requires predetermination of extension points, it limits the use of unforeseen instrumentations.

4 Conclusion

Although tools based on instrumentation techniques are in wide-spread use, the engineering of such tools often violates basic reuse principles. As efficiency of the tools is of paramount importance, low-level instrumentation frameworks, which suffer from a lack of mechanisms for refining and composing instrumentations, are commonly used. As a consequence, instrumentations are often implemented by resorting to the tedious and error-prone copy/paste anti-pattern.

In this position paper we identified two challenges that need to be addressed by future mechanisms in support of refinement and composition of instrumentations: specification and enforcement of constraints, and avoidance of hard-coded dependencies. We illustrated these challenges with a running example.

In our ongoing research, we are exploring novel refinement and composition mechanisms in the context of the domain-specific instrumentation language DiSL [9,18]. We are working on instrumentation contracts that make constraints explicit and allow for automated checks that enforce these constraints. Furthermore, we are integrating a mechanism for dependency injection to deal with the problem of hard-coded dependencies.

Acknowledgments. The research presented here was conducted while L. Marek was with the University of Lugano. It was supported by the Scientific Exchange Programme NMS–CH (project code 10.165), by a Sino-Swiss Science and Technology Cooperation (SSSTC) Institutional Partnership (project no. IP04–092010), by the Swiss National Science Foundation (project CRSII2_136225), by the Science and Technology Commission of Shanghai Municipality (project no. 11530700500), by the Czech Science Foundation (project GACR P202/10/J042), as well as by the German Federal Ministry of Education and Research (BMBF) in EC SPRIDE and by the Hessian LOEWE excellence initiative in CASED. The authors thank Éric Tanter for comments on a draft of this paper.

References

1. Ammons, G., Ball, T., Larus, J.R.: Exploiting hardware performance counters with flow and context sensitive profiling. In: Proceedings of the Conference on Programming Language Design and Implementation, pp. 85–96 (1997)
2. Aracic, I., Gasiunas, V., Mezini, M., Ostermann, K.: An Overview of CaesarJ. In: Rashid, A., Aksit, M. (eds.) Transactions on AOSD I. LNCS, vol. 3880, pp. 135–173. Springer, Heidelberg (2006)
3. Bergmans, L.M.J.: Akşit, M.: Principles and design rationale of composition filters. In: Aspect-Oriented Software Development, pp. 63–96. Addison-Wesley (2004)
4. Binder, W., Ansaloni, D., Villazón, A., Moret, P.: Flexible and efficient profiling with aspect-oriented programming. Concurrency and Computation: Practice and Experience 23(15), 1749–1773 (2011)
5. Chiba, S.: Load-Time Structural Reflection in Java. In: Bertino, E. (ed.) ECOOP 2000. LNCS, vol. 1850, pp. 313–336. Springer, Heidelberg (2000)
6. Flanagan, C., Freund, S.N.: The RoadRunner dynamic analysis framework for concurrent programs. In: Proceedings of the 9th Workshop on Program Analysis for Software Tools and Engineering, pp. 1–8 (2010)
7. Greenwood, P., Blair, L.: A Framework for Policy Driven Auto-Adaptive Systems Using Dynamic Framed Aspects. In: Rashid, A., Aksit, M. (eds.) Transactions on AOSD II. LNCS, vol. 4242, pp. 30–65. Springer, Heidelberg (2006)
8. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: Lee, S.H. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–354. Springer, Heidelberg (2001)
9. Marek, L., Villazón, A., Zheng, Y., Ansaloni, D., Binder, W., Qi, Z.: DiSL: a domain-specific language for bytecode instrumentation. In: Proceedings of the 11th International Conference on Aspect-Oriented Software Development (2012)
10. Moret, P., Binder, W., Tanter, É.: Polymorphic bytecode instrumentation. In: Proceedings of the 10th International Conference on Aspect-Oriented Software Development, pp. 129–140 (2011)
11. Rudolph, J., Thiemann, P.: Mnemonics: type-safe bytecode generation at run time. In: Proceedings of the Workshop on Partial Evaluation and Program Manipulation, pp. 15–24 (2010)
12. Suvée, D., Vanderperren, W., Jonckers, V.: JAsCo: an aspect-oriented approach tailored for component based software development. In: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development, pp. 21–29 (2003)

13. Tanter, E., Moret, P., Binder, W., Ansaloni, D.: Composition of dynamic analysis aspects. In: Proceedings of the 9th International Conference on Generative Programming and Component Engineering, pp. 113–122 (2010)
14. Tarr, P., Osher, H., Stanley, M., Sutton, J., William Harrison, W.: N degrees of separation: multi-dimensional separation of concerns. In: Aspect-Oriented Software Development, pp. 37–61. Addison-Wesley (2004)
15. Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P.: Soot - a Java optimization framework. In: Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research, pp. 214–224 (1999)
16. Villazón, A., Binder, W., Moret, P.: Flexible calling context reification for aspect-oriented programming. In: Proceedings of the 8th International Conference on Aspect-Oriented Software Development, pp. 63–74 (2009)
17. Villazón, A., Binder, W., Moret, P., Ansaloni, D.: Comprehensive aspect weaving for Java. Science of Computer Programming 76(11), 1015–1036 (2011)
18. Zheng, Y., Ansaloni, D., Marek, L., Sewe, A., Binder, W., Villazón, A., Tuma, P., Qi, Z., Mezini, M.: Turbo DiSL: partial evaluation for high-level bytecode instrumentation. In: TOOLS 2012 – Objects, Models, Components, Patterns (2012)