



HAL
open science

Bridging the Gap: A Focused DSL for RV-Oriented Instrumentation with BISM

Chukri Soueidi, Yliès Falcone

► **To cite this version:**

Chukri Soueidi, Yliès Falcone. Bridging the Gap: A Focused DSL for RV-Oriented Instrumentation with BISM. RV 2023 - 23rd International Conference on Runtime Verification, Oct 2023, Thessalokini, Greece. pp.327-338, 10.1007/978-3-031-44267-4_17 . hal-04381683

HAL Id: hal-04381683

<https://inria.hal.science/hal-04381683v1>

Submitted on 9 Jan 2024



HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License

Bridging the Gap: A Focused DSL for RV-Oriented Instrumentation with BISM

Chukri Soueidi  and Yliès Falcone 

Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France
firstname.lastname@inria.fr

Abstract. We present a novel instrumentation language for BISM, a lightweight bytecode-level instrumentation tool for JVM languages. The new DSL aims to simplify the instrumentation process, making it more accessible to a wider user base. It employs an intuitive syntax, directly mapping to the key requirements of program instrumentation for runtime verification. It enhances productivity by eliminating boilerplate code and low-level details, while also supporting code generation and collaboration. The DSL balances expressiveness, and abstraction, bridging the gap between domain experts and the complexities of instrumentation specification.

1 Introduction

Instrumentation is a fundamental aspect of Runtime Verification (RV) [3,8,7]. It entails modifying the original code within a program to extract events or insert monitors that analyze its behavior. Instrumentation languages, as well as domain-specific languages (DSLs) in general, can be primarily classified into two categories: *external* and *internal* [9]. External DSLs are self-contained languages, complete with their own parsers or compilers, and are not necessarily dependent on any host language. They typically feature customized syntax designed specifically for the target domain. In contrast, internal DSLs are implemented as an application programming interface (API) within a host language. While internal languages offer seamless integration with the host language, external languages provide a focused syntax for domain-specific concerns, making them easier to understand and learn by domain experts. From here on, we will refer to internal languages as *APIs* and external languages as *DSLs*.

For runtime verification, a widely popular instrumentation language is AspectJ [10] which implements aspect-oriented programming (AOP) for Java. It provides both an API, which extends Java with AOP constructs, and a DSL, known as the AspectJ pointcut expression language, for defining pointcuts using a domain-specific syntax. Nonetheless, AspectJ exhibits certain limitations that hinder its effectiveness. One such limitation is the absence of bytecode coverage, which affects its applicability in multithreaded programs and low-level monitoring scenarios [16]. Another limitation is the inability to inline inserted instructions. Lastly, AspectJ is also known to introduce increased overhead to the base program after instrumentation. More recently, BISM (Bytecode-Level Instrumentation for Software Monitoring) [17,18] has been presented as an instrumentation tool for Java programs addressing these limitations. It is oriented

more toward runtime verification and features an expressive and high-level instrumentation language inspired by AOP to facilitate its adoption. It has been used effectively in monitoring concurrent programs [13,16] and for combining static analysis with runtime verification [14,15].

At present, BISM employs an effective yet somewhat intricate API-based language for implementing transformers. This necessitates some degree of familiarity with Java and the BISM API, which could present challenges for the adoption of BISM as a tool. To make the process of instrumentation more approachable and user-friendly, we introduce a new DSL for BISM. Designed with user-friendliness in mind, the DSL has an intuitive syntax that maps directly to the core requirements of program instrumentation for runtime verification.

Balancing expressiveness and abstraction, our DSL offers comprehensive coverage of program aspects while reducing the complexity of instrumentation specifications. This abstraction, similar to fitting puzzle pieces together, streamlines transformer creation but comes with a moderate cost to performance and flexibility. The DSL eliminates boilerplate code and low-level implementation details, enhancing productivity and reducing the risk of errors. Its design also simplifies code maintenance, making it user-friendly for a broad range of expertise levels. The effectiveness of these enhancements is confirmed through our evaluation.

In our design, we incorporated both code generation and collaboration elements. The DSL can optionally generate equivalent API-based transformer code for more complex instrumentation tasks. It can also generate monitor interfaces from specifications, which promotes efficient collaboration between program developers and monitoring experts. These features not only simplify the instrumentation process but also ensure a clear separation of roles in runtime verification tasks. Notably, our proposed language is the only external specification language known to us that can target JVM bytecode instrumentation covering languages including Java, Scala, Kotlin, and Groovy. In summary, the new DSL for BISM aims to bridge the gap between domain experts and the complexities of implementing BISM transformers, streamlining the instrumentation process for a wide range of users.

2 DSL Design Considerations

In this section, we discuss the main design considerations for the new BISM DSL.

Intuitive syntax. Instrumenting a program generally involves specifying three key requirements: (1) program points to capture (*joinpoints*), (2) the information needed from these joinpoints, and (3) the consumption of these events. Our DSL addresses these requirements by providing exactly these three main constructs: (1) *pointcuts*, (2) *events*, and (3) *monitors* to consume the captured events.

Expressiveness & abstraction. BISM API offers remarkable expressiveness for covering all program aspects and extracting information from the executing program. We designed the DSL to retain this expressiveness while considerably simplifying the specification and providing a higher abstraction level. However, this abstraction comes with

a moderate cost to flexibility in scenarios where fine-grained control over the instrumentation process is required. For instance, when some analysis is required within the instrumentation process which can be achieved with the BISM API.

Efficiency and performance. Textual specifications for instrumentation often require parsing and compiling transformations. We optimized the DSL implementation with a focus on performance, striving to achieve reasonable execution times for parsing and applying instrumentation compared to using the native BISM API.

Usability simplification. Crafting a BISM transformer requires implementing a Java class using the provided instrumentation API. The DSL is tailored to accommodate users with diverse expertise levels by eliminating boilerplate code and low-level implementation details, ultimately enhancing productivity, minimizing errors, and simplifying code maintenance.

Code Generation. Our DSL can create API-based transformer code from user-defined textual specifications, allowing further user customizations for complex instrumentation. It can also generate monitor interfaces, promoting collaboration between developers and monitoring experts.

3 BISM Background

BISM (Bytecode-Level Instrumentation for Software Monitoring) [17,18] is a lightweight instrumentation tool for Java that is designed to provide an expressive and high-level instrumentation language for runtime verification and enforcement. The tool is inspired by aspect-oriented programming (AOP) languages, utilizing separate classes called *transformers* to encapsulate joinpoint selection and advice inlining. BISM *selectors* facilitate the selection of joinpoints by associating each selector with a well-defined region in the bytecode, such as a single bytecode instruction, control-flow branch, or method call. Listing 1.1 shows the main available selectors.

Before/AfterInstruction	OnBasicBlockEnter/Exit
Before/AfterMethodCall	OnTrue/FalseBranchEnter
OnFieldSet/Get	OnMethodEnter/Exit

Listing 1.1: BISM selectors

Static context objects allow users to extract at joinpoints information derived at compile time about instructions, basic blocks, methods, and classes. *Dynamic context objects* extract values that are only available at runtime, such as instance and static fields, local variables, method arguments, method results, executing class instances, and stack values. Additionally, BISM allows the creation of new local variables and arrays within the scope of a method to pass values required for instrumentation. *Advice methods* specify how to extract the desired information, such as invoking methods or printing. These advice methods enable the extraction of information at the identified joinpoints, based on the selected selectors and desired information extraction. A distinguishing feature of BISM is its ability to insert arbitrary bytecode instructions into the base program making it suitable for instrumenting inline monitors and enforcers.

```

public class IteratorTransformer extends Transformer {
    public void afterMethodCall(MethodCall mc, DynamicContext dc){
        // Filter to only method of interest
        if (mc.methodName.equals("iterator") &&
            mc.methodOwner.endsWith("List")) {

            // Access to dynamic data
            DynamicValue list = dc.getMethodTarget();
            DynamicValue iterator = dc.getMethodResult();

            // Create an ArrayList in the target method
            LocalArray la = dc.createLocalArray(mc.ins.basicBlock.method,
                Object.class);
            dc.addToLocalArray(la, list);
            dc.addToLocalArray(la, iterator);

            // Invoke the monitor after passing arguments
            StaticInvocation sti =
                new StaticInvocation("monitors.SafeListMonitor",
                    "receive");
            sti.addParameter("create");
            sti.addParameter(la);
            invoke(sti);
        }
    }
}

```

Listing 1.2: A BISM transformer written in Java that intercepts the creation of an iterator

Users specify the instrumentation by extending a base `Transformer` type. Listing 1.2 presents a fragment of a transformer used in a parametric monitoring setup [1,5] where we monitor the **SafeIterator** property¹. The BISM transformer, written in Java, uses the selector `afterMethodCall` to capture the return of an `Iterator` created from a `List.iterator()` method call. It uses the dynamic context object provided to retrieve the associated objects with the event. It also creates a local list to push the objects into it. Then, invokes a monitor passing an event name and the list as arguments.

4 The DSL for BISM

We here present the main abstractions and constructs provided by the DSL to address the user requirements of runtime verification and instrumentation.

4.1 Pointcuts

Pointcuts enable users to specify the joinpoints to capture from program execution. They can be denoted as follows: a pointcut *name*, a BISM selector along with a *pattern*, and an optional *guard*. Multiple selectors are chainable using the `||` operator. The pattern restricts the scope of the selector to specific methods or fields (for field selectors), applying filters such as types, method signatures, or field names. Matching can be achieved with wildcards; for example, `"* .set*(.*)" matches any method that`

¹ The property specifies that a `Collection` should not be updated when an iterator associated with it is created and being used

starts with "set". The optional guard allows the specification of a condition, essentially a Boolean expression using static context objects from the joinpoint. The guard conditions may use comparisons of booleans, numerics, and strings, and can be chained with the conjunction operator (&&) to create complex conditions. In each DSL transformer, the definition of at least one pointcut is necessary.

```
pointcut pc1 after MethodCall(* BankAccount.*(..)) with (
    getNumberOfArgs = 3 && currentClassName = "Main")
    || after MethodCall(* *.*(..)) with (instruction.
        lineNumber = 42)

pointcut pc2 before Instruction(* *.*(..))
    with (isConditionalJump = true)
```

Listing 1.3: Example of pointcuts definition.

Listing 1.3 shows a composite pointcut `pc1` that uses two selectors. The first selector captures calls to methods defined by the class `BankAccount`, but only captures calls that are invoked from the "Main" class and the called method has exactly three arguments. The second selector captures any method call occurring at line 42. This second guard showcases BISM's hierarchical context objects and how they can be accessed using dot notation. Pointcut `pc2` captures any instruction that is a conditional jump.

4.2 Events

Events encapsulate the information that needs to be extracted from a pointcut. Each event must be associated with a single pointcut along with its arguments. Multiple events can be defined for each pointcut. An event includes a name and zero or more arguments. Arguments may comprise single values or lists of values, which can include BISM static or dynamic context objects, string literals, numbers, or lists. Lists are denoted as sequences of values, separated by commas, and enclosed in brackets.

```
event e1("call", [getMethodReceiver, getMethodResult]) on pc1

event e2([opcode, getStackValues]) on pc2 to console(List)
```

Listing 1.4: Example of events definition.

Listing 1.4 shows an event named `e1`, associated with the pointcut `pc1`, that is defined with the string literal "call" and a list of dynamic context objects which extract the callee object and the result of the method. Event `e2` is defined with a list of dynamic context objects (`opcode`, `getStackValues`) and is associated with the pointcut `pc2`. The DSL also provides a construct to print an event to the console, which is particularly useful during the debugging or profiling of a program. This event is associated with the output `console(List)` which prints the event information to the console.

4.3 Monitors

Monitors listen to the occurrence of one or several events, and they define the extraction points for these events during program execution. Each monitor is identified by a unique

name, the class name and package locating its implementation, and the events it is set to listen to. Typically, the events are passed as parameters during the invocation of a monitor method. Events are mapped to the monitor method name with its argument types. Multiple monitors can be defined and each event can be listened to by more than one monitor.

```
monitor m1{
  class: com.MonitorX,
  events: [e1 to receive(String, List)]
}
```

Listing 1.5: Example of a monitor definition.

Listing 1.5 shows a monitor `m1` corresponding to a class named `com.MonitorX`. Event `e1` is mapped to the method `receive` with the argument types `String` and `List`. The specification can be simplified by directly associating the monitor with the event. However, this restricts the use of the event to only one monitor. Here is an equivalent specification without the explicit definition of a monitor.

```
event e1("create", [getMethodReceiver, getMethodResult])
  on pc1 to com.MonitorX.receive(String, List)
```

Listing 1.6: Simple monitor definition.

4.4 Code Generation

We here present the code generation facilities that bridge the gap between new and expert users of BISM, also the gap between monitoring experts and program developers.

Transformers for Complex Instrumentation Tasks The Java-based API transformers allow users to write intricate analysis logic within transformers. Equipped with the full program context provided by BISM and the Java language, the user can write any piece of analysis code that can guide the instrumentation process. For example, in [15] such analysis was used to perform residual analysis which entails checking for safe instructions statically such that they can be ignored from the instrumentation. Starting with a simple text-based DSL specification where events of interest can be identified, users can gradually move to complex instrumentation logic using a full-fledged Java transformer. To facilitate this transition, we optionally generate and compile Java transformers equivalent to the provided textual specification files. Consequently, these specification files can act as bootstrappers for implementing more complex logic.

Monitor Interfaces As discussed in Section 4.3, the process of creating an instrumentation specification often requires declaring a monitor to listen for events. In collaborative scenarios, a monitoring expert consults the developer to identify relevant events and their static and runtime context. Subsequently, the programmer creates a specification file capable of locating and extracting the necessary events, leaving the monitoring

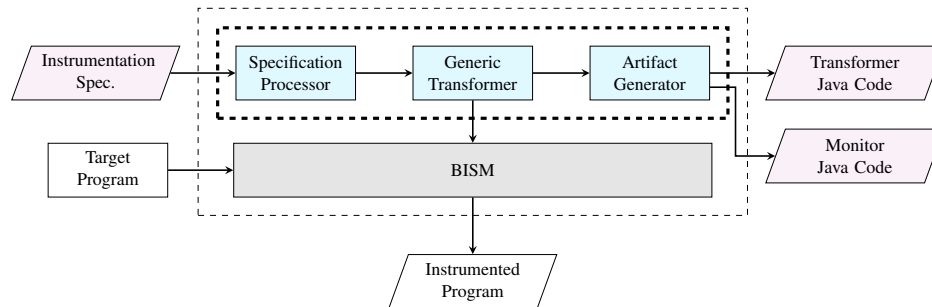


Fig. 1: Added modules on BISM (in blue) to support the external DSL.

expert solely responsible for implementing the monitors. To optimize this collaboration, our DSL incorporates a feature to generate the monitor interfaces, that need to be implemented to listen to events. This capability substantially improves interoperability between different teams, fostering a more efficient and productive workflow when addressing complex instrumentation tasks.

```

package com;
import java.util.List;

public class MonitorX{
    public static void receive(String a1, List a2) {
    }
}

```

Listing 1.7: A generated monitor class.

Listing 1.7 shows the generated interface for the monitor in Listing 1.5. The user can then implement the `receive` method to perform the desired analysis logic.

5 Implementation

We implemented the DSL as an extension to BISM with 6 KLOC of Java code². Figure 1 shows the three main modules we added to BISM which handle specification parsing, transformation application, and code generation. We provide more details on each module below.

Specification Processor. This module accepts the user’s textual specification as input and parses it into a specification object. This object is an intermediate representation that embodies the desired transformations. The module also performs a series of checks to ensure the specification’s validity, including the removal of duplicate rules and verification of well-formedness

² The tool is available at <https://gitlab.inria.fr/bism/bism-dsl>.

Generic Transformer. This module is in charge of applying user-specified transformations to the target program. It features a transformer template that extends BISM’s `Transformer` type and automatically generates the appropriate transformations. These transformations are then turned into bytecode instructions by BISM that are then weaved into the target program.

Artifact Generator. Activated upon a user’s request for code generation, this module generates an equivalent API-based transformer class utilizing BISM’s API. Additionally, it generates a monitor class that can be implemented and used to monitor the instrumented program’s behavior during runtime.

6 Evaluation

In this section, we provide an assessment of the DSL, specifically focusing on the overhead it introduces and the user experience ³.

6.1 Performance Evaluation

To evaluate the performance, we examine the overhead introduced by our DSL in comparison with both the BISM API and AspectJ. Our test case is the financial transaction system as described in [2]. We instrument this system to extract events from all method calls on `Iterator` objects, get field operations, and method executions. We test four distinct methodologies to write identical instrumentation specifications:

- **AspectJ DSL:** An aspect (`.aj`) is written using the DSL, requiring approximately 40 lines of code.
- **AspectJ API:** An API-based aspect (`.java`) is written, using annotations, requiring approximately 60 lines of code.
- **BISM DSL:** A transformer (`.spec`) is written with our proposed DSL for BISM, requiring approximately 20 lines of code.
- **BISM API:** An API-based BISM transformer (`.java`) is written using the API, requiring approximately 70 lines of code.

To ensure a fair comparison, we utilized features available across all four approaches ⁴. Each benchmark was run 20 times, with each run producing 500K events, and the mean execution times are reported in Figure 2a. The results showed that BISM API outperforms the other methods, running 1.14 times faster than BISM DSL, 1.7 times faster than AspectJ API, and 3.25 times faster than AspectJ DSL. This demonstrates that specifying instrumentation using the API generally leads to faster execution due to the extra delay incurred in DSL approaches from parsing the specification files. However, our proposed DSL outperforms AspectJ DSL and is even faster than AspectJ API, maintaining the effectiveness and efficiency of BISM in the context of software monitoring and instrumentation.

³ The full details for the experiments can be found at <https://gitlab.inria.fr/bism/bism-dsl-experiments>.

⁴ The experiment utilized AspectJ AJC 1.9.7 and JDK 11.

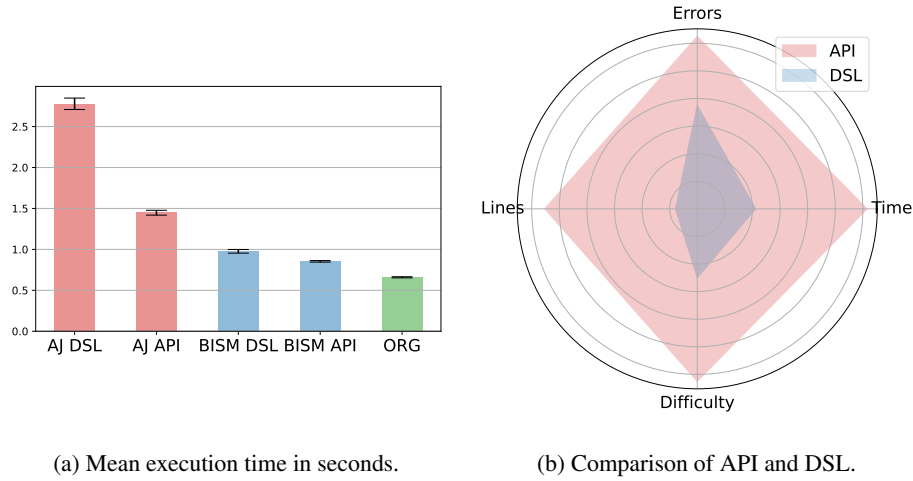


Fig. 2: Performance evaluation results.

6.2 User Experience Evaluation

Experiment design. We conducted an experiment in which 10 participants, with various experience levels in Java and runtime verification, were instructed to write transformers for monitoring four different properties sourced from [2] using both methods. The experiment used a randomized block design. The participants were divided into two equal groups: Group A and Group B. Group A used the API for Properties 1 and 3 and the DSL for Properties 2 and 4. In contrast, Group B used the DSL for Properties 1 and 3 and the API for Properties 2 and 4. This design allowed each participant to gain experience with both methods, enabling a fair comparison between the two techniques across all properties.

Collected metrics. For each property, the participants are asked to record the time they took to write the instrumentation, reported in minutes (**Time**). They also kept track of the number of mistakes they made during the process that necessitated recompilation and another run (**Errors**). In addition, the number of lines of code written for each transformer was noted, (**Lines**). Lastly, participants rated the difficulty of use on a scale of 1 to 5, where 1 signified very easy and 5 meant very hard, (**Difficulty**).

Results and analysis. In Figure 2b, we report each metric normalized to a 0-1 range based on their respective minimum and maximum values. Table 1 shows the average values for each metric. The results indicate a clear advantage of the DSL over the API for writing BISM transformers. Across all properties tested, the DSL not only needed significantly less time to implement but also resulted in fewer errors and less code. Furthermore, participants found the DSL easier to use. The good results of the DSL can be largely attributed to its concise syntax and straightforward usage. However, it is important to note that a considerable portion of errors in DSL mode were caused

by improper referencing of the monitor and its package name, and the need for full specification of return types in patterns. Future iterations of the DSL will address these pattern specification issues.

Metric	Property 1	Property 2	Property 3	Property 4
Time (API)	15	13	5	7
Time (DSL)	8	7	2	2
Errors (API)	4	3	2	1
Errors (DSL)	3	3	0	0
Lines (API)	12	16	10	24
Lines (DSL)	7	5	5	9
Difficulty (API)	4	4	3	3
Difficulty (DSL)	3	3	1	1

Table 1: Average values for each metric.

7 Related Work and Conclusion

In this section, we review some of the relevant frameworks for Java program instrumentation and compare them to our proposed DSL for BISM. It is worth noting that our proposed language is the only *external* specification language we know of that can target bytecode instrumentation thereby covering JVM languages including Java, Scala, Kotlin, and Groovy. API-based frameworks such as ASM [4], BCEL [19], Soot [20], and Javassist [6] offer a wide range of low-level bytecode transformation capabilities. However, they demand a significant understanding of bytecode and can be verbose when implementing simple instrumentation specifications. ASM, in particular, was chosen over Soot for BISM due to its performance and compact size, while Javassist and BCEL provide better levels of abstraction from bytecode details. On the other end of the spectrum, aspect-oriented programming (AOP) frameworks like AspectJ [11] provide a high-level language for specifying instrumentation. However, AspectJ incurs considerable overhead, and cannot instrument at the bytecode level, limiting its usefulness in multithreaded programs and low-level monitoring scenarios. Balancing these approaches, DiSL [12] offers an extensible API-based instrumentation language with advanced features eliminating transformation interference. DiSL offers a complex set of features making it more suitable for dynamic analysis scenarios such as profiling. In comparison, BISM reduces execution overhead and allows for arbitrary code insertion, necessary for inlining monitors and enforcers.

Our proposed BISM DSL aims to simplify BISM usage by offering a focused syntax for instrumentation and runtime verification. However, it only supports a subset of the BISM features and lacks support for inserting arbitrary bytecode instructions. The potential for DSL and API full integration is currently under investigation. Moreover, we aim to add enforcement constructs to the DSL, allowing users to specify inlined enforcers.

References

1. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.E.: Quantified event automata: Towards expressive and efficient runtime monitors. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7436, pp. 68–84. Springer (2012). https://doi.org/10.1007/978-3-642-32759-9_9
2. Bartocci, E., Falcone, Y., Bonakdarpour, B., Colombo, C., Decker, N., Havelund, K., Joshi, Y., Klaedtke, F., Milewicz, R., Reger, G., Rosu, G., Signoles, J., Thoma, D., Zalinescu, E., Zhang, Y.: First international competition on runtime verification: rules, benchmarks, tools, and final results of CRV 2014. *Int. J. Softw. Tools Technol. Transf.* **21**(1), 31–70 (2019), <https://gitlab.inria.fr/crv14/benchmarks/>
3. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Lectures on Runtime Verification - Introductory and Advanced Topics, pp. 1–33 (2018). https://doi.org/10.1007/978-3-319-75632-5_1
4. Bruneton, E., Lenglet, R., Coupaye, T.: ASM: A code manipulation tool to implement adaptable systems. In: Adaptable and extensible component systems (2002), <https://asm.ow2.io>
5. Chen, F., Roşu, G.: Parametric trace slicing and monitoring. In: Kowalewski, S., Philippou, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 246–261. Springer Berlin Heidelberg, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00768-2_23
6. Chiba, S.: Load-time structural reflection in java. In: Bertino, E. (ed.) ECOOP 2000 - Object-Oriented Programming, 14th European Conference, Sophia Antipolis and Cannes, France, June 12-16, 2000, Proceedings. Lecture Notes in Computer Science, vol. 1850, pp. 313–336. Springer (2000). https://doi.org/10.1007/3-540-45102-1_16
7. Falcone, Y., Havelund, K., Reger, G.: A tutorial on runtime verification. In: Broy, M., Peled, D.A., Kalus, G. (eds.) Engineering Dependable Software Systems, NATO Science for Peace and Security Series, D: Information and Communication Security, vol. 34, pp. 141–175. IOS Press (2013). <https://doi.org/10.3233/978-1-61499-207-3-141>, <https://doi.org/10.3233/978-1-61499-207-3-141>
8. Falcone, Y., Krstic, S., Reger, G., Traytel, D.: A taxonomy for classifying runtime verification tools. *Int. J. Softw. Tools Technol. Transf.* **23**(2), 255–284 (2021). <https://doi.org/10.1007/s10009-021-00609-z>, <https://doi.org/10.1007/s10009-021-00609-z>
9. Fowler, M., Parsons, R.: Domain-specific languages. Addison-Wesley, Upper Saddle River, NJ (2011)
10. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: Getting started with AspectJ. *Commun. ACM* **44**(10), 59–65 (2001)
11. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: Getting started with AspectJ. *Commun. ACM* **44**(10), 59–65 (2001)
12. Marek, L., Villazón, A., Zheng, Y., Ansaloni, D., Binder, W., Qi, Z.: DiSL: a domain-specific language for bytecode instrumentation. In: Hirschfeld, R., Tanter, É., Sullivan, K.J., Gabriel, R.P. (eds.) Proceedings of the 11th International Conference on Aspect-oriented Software Development, AOSD, Potsdam, Germany. pp. 239–250. ACM (2012)
13. Soueidi, C., El-Hokayem, A., Falcone, Y.: Opportunistic monitoring of multithreaded programs. In: Lambers, L., Uchitel, S. (eds.) Fundamental Approaches to Software Engineering - 26th International Conference, FASE 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings. Lecture Notes in Computer Science, vol. 13991, pp. 173–194. Springer (2023). https://doi.org/10.1007/978-3-031-30826-0_10

14. Soueidi, C., Falcone, Y.: Capturing program models with BISM. In: Hong, J., Bures, M., Park, J.W., Cerný, T. (eds.) SAC '22: The 37th ACM/SIGAPP Symposium on Applied Computing, Virtual Event, April 25 - 29, 2022. pp. 1857–1861. ACM (2022). <https://doi.org/10.1145/3477314.3507239>
15. Soueidi, C., Falcone, Y.: Residual runtime verification via reachability analysis. In: Lal, A., Tonetta, S. (eds.) Verified Software. Theories, Tools and Experiments - 14th International Conference, VSTTE 2022, Trento, Italy, October 17-18, 2022, Revised Selected Papers. Lecture Notes in Computer Science, vol. 13800, pp. 148–166. Springer (2022). https://doi.org/10.1007/978-3-031-25803-9_9
16. Soueidi, C., Falcone, Y.: Sound concurrent traces for online monitoring. In: Caltais, G., Schilling, C. (eds.) Model Checking Software - 29th International Symposium, SPIN 2023, Co-located with the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 26–27, 2023, Proceedings. pp. 59–80. Springer Nature Switzerland, Cham (2023). https://doi.org/10.1007/978-3-031-32157-3_4
17. Soueidi, C., Kassem, A., Falcone, Y.: BISM: bytecode-level instrumentation for software monitoring. In: Deshmukh, J., Nickovic, D. (eds.) Runtime Verification - 20th International Conference, RV 2020, Los Angeles, CA, USA, October 6-9, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12399, pp. 323–335. Springer (2020). https://doi.org/10.1007/978-3-030-60508-7_18
18. Soueidi, C., Monnier, M., Falcone, Y.: International Journal on Software Tools for Technology Transfer pp. 1–27 (2023). <https://doi.org/10.1007/s10009-023-00708-z>, <https://link.springer.com/article/10.1007/s10009-023-00708-z>
19. The Apache Software Foundation: Apache commons. <https://commons.apache.org>, accessed: 2020-06-18
20. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot - a java bytecode optimization framework. In: Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research. p. 13. CASCON '99, IBM Press (1999)