An Extensible AOP Framework for Runtime Monitoring

Gholamali Rahnavard
New Mexico State University
rah@nmsu.edu

Amjad Nusayr
University of Houston - Victoria
nusayra@uhv.edu

Jonathan Cook
New Mexico State University
joncook@nmsu.edu

ABSTRACT

We present a design and initial prototype for TEAMS, a new aspect oriented programming framework designed specifically for usage as an abstract instrumentation capability for runtime monitoring and dynamic analysis. Our goals for this framework are simplicity, extensibility, portability, and monitoring concept coverage. If successful, TEAMS will provide us and other researchers an easy-to-use platform for building instrumentation that will support their monitoring and analysis research, and will provide practioners an ability to craft their own analyses without needing to understand low-level instrumentation.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Measurement, Languages

Keywords

Runtime Monitoring, Aspect Oriented Programming

1. INTRODUCTION

For several years we have investigated the application of aspect oriented programming (AOP) to runtime monitoring instrumentation. Indeed if one looks at the success stories of AOP, many of them are analysis-based applications that use AOP weaving to instrument and monitor the underlying program they analyze. Examples are too numerous to list and cite. The drawback that existing AOP systems have is that none of them support all the levels of detail that dynamic analyses can require, and this limits the granularity of what a researcher can do with them. For example, virtually all AOP frameworks can only instrument down to the method level (call or execution) when it comes to code detail, and data instrumentation is usually very sparse (e.g., object fields but nothing else) or non-existent. We pointed this out in 2009 [12, 14] as we began a detailed investigation of the application of AOP to runtime monitoring, and others have noticed this before us [17] and even more recently [2]. Thus investigating the application of AOP ideas and mechanisms to the full spectrum of runtime monitoring needs is still an important research area.

We encountered severe limitations in extending existing AOP systems for the full breadth of runtime monitoring needs, and so are now attempting to build TEAMS, our own AOP framework, not for general purpose AOP usage but in particular for using AOP as a high-level abstraction for instrumentation that will be used for monitoring and analysis. TEAMS stands for The Extensible Aspect-based Monitoring System. This short paper presents our initial ideas for the framework, some early progress in prototyping the framework, and some promising experimental results that encourage us to continue in this path.

WODA '13, Houston, TX, USA.

2. MOTIVATION AND BACKGROUND

For presentation purposes, the mapping between AOP terms and runtime monitoring terms we use is: the *advice* is the instrumentation code; a *joinpoint* is a point in a program execution at which instrumentation (advice) can be inserted; a *pointcut* is a set of joinpoints, usually specified abstractly in a *pointcut expression*; and a *pointcut designator* (PCD) is specific primitive type of instrumentation point (e.g., the PCD "call" denotes the call site of a method).

In our previous work, we evaluated the breadth of needs that runtime monitoring has for its instrumentation, and devised a *dimension* based view of the types of weaving AOP would need to support to be able to create such instrumentation [13]. Our defined dimensions were *code*, traditional weaving over code (instructions); *data*, weaving over concepts in data space; *time*, weaving based on time constraints; and *sampling*, weaving to support sampling-based instrumentation. We defined these dimensions after seeing that most AOP approaches both to weaving and to the example runtime monitors they support virtually all focused on the code dimension, even though the others are useful. Using these dimensions, we began to look at how we might extend existing AOP systems to better support runtime monitoring across all of these dimensions.

The *code* dimension of weaving is well understood already, and most AOP frameworks support a variety of code-based pointcut designators (PCDs), which include method call, method execution, and even complex PCDs such as *cflow*. For monitoring purposes, however, existing AOP

frameworks are still quite limited in the features that they support. Statement level coverage analysis and many other analyses depend on being able to instrument down to the statement (or basic block) level, yet current AOP systems do not support this.

The *data* dimension weaving has limited support in most existing AOP frameworks. For example, AspectJ has pointcut designators for object field accesses, but not for accesses to local variables, array elements, or arguments. The data dimension also envisions weaving on higher level data-oriented concepts, such as when a node in a data structure is accessed, when references to objects have changed, or how much space an application has allocated.

The *time* dimension entails weaving not based on locations (points) in code or data, but on timers, either relative or absolute. An obvious example of the utility of this dimension is profiling, where a timer-based interruption of the program samples where the program is at that point in time, and constructs a statistical profile of the execution behavior of the program. Other time-based uses would be to periodically check data structure health or application progress.

The *sampling* (probability) dimension covers the notion of controlling whether or not the advice is actually executed at a joinpoint, or not. Current AOP assumes that every time a joinpoint satisfying the pointcut expression is reached, the advice will execute. However, research in runtime monitoring has shown the utility of sampling-based approaches, where instrumentation is executed probabilistically, either randomly or (more efficiently) with a counter-based approach.

In our previous experimental work we constructed both *basicblock* and *loopbackedge* PCDs [14], a time interval PCD, and several sampling PCDs, investigating both static and runtime probability-based weaving, and also non-random fixed-ratio sampling [14].

3. DESIGN

In attempting to address the full breadth of runtime monitoring needs by extending existing AOP frameworks, we found them to be very code-centric in their instrumentation (weaving) mechanisms and ultimately found it extremely difficult if not impossible to pursue all of the ideas we had. Thus we are now building our own framework, TEAMS, in order to explore applying AOP across the spectrum of runtime monitoring needs.

Figure 1 shows the high-level design of TEAMS. TEAMS will be a lightweight cross-platform system for exploring AOP mechanisms, to be used in particular for runtime monitoring. Focusing on monitoring in particular will mean that we will concentrate on features that are *observational* and *informational*, and will not be concerned with AOP features that modify the behavior of the system. This greatly simplifies some of the standard AOP concerns such as conflicting advice behavior.

The pointcut expression compiler and the actual instrumentation weaver will be the static parts of TEAMS; these will support both pointcut designators and advice execution mechanisms created using extension interfaces. We will create a suite of our own PCDs and advice execution mechanisms, but our intent is that others as well will be able to explore their own ideas in TEAMS.

A key notion in our architecture is that the *advice* portion of an aspect will be written in a *plain old programming language*, or POPL for short. In many AOP systems advice is packaged into the aspect, which then requires the AOP framework to either have a full language compiler built in, or at least a partial grammar of the language in order to translate extra-language features inside the advice into the underlying programming language. We will instead rely on well-defined interfaces and standard naming conventions to connect advice written separately in the POPL with the pointcut expressions and aspect definitions written in our lightweight aspect language.

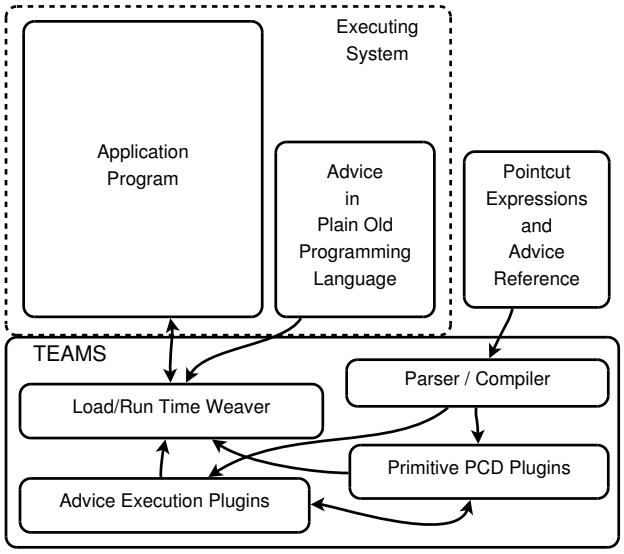

Figure 1: TEAMS overall architecture.

We will initially focus on a fully dynamic runtime engine that will perform load-time or run-time weaving, but we can later integrate static weaving if the performance payoff is large and needed.

While Section 4 presents our initial work in building TEAMS in Java, we view the architecture of TEAMS as replicable to other platforms. The weaver component of TEAMS needs to be built around an actual instrumentation mechanism; in Java we are using ASM, but in, say, a compiled environment we could use Dyninst [5] or Pin [9] or some other instrumentation tool. These choices would result in different TEAMS frameworks suited not only for different platforms but also for different purposes. For example, in a test environment setting where execution slowdown might not be an issue, we could use a very intrusive tool such as Pin to provide very low-level joinpoint types and enable a user to create fine-grained runtime monitoring instrumentation at the abstract AOP level without worrying about how to program Pin itself.

A major focus of TEAMS is extensibility: we aim to allow advanced users and researchers to develop their own monitoring mechanisms that will integrate into TEAMS. The major areas of extension are: 1) new joinpoint designators that create new fundamental monitoring mechanisms; 2) new advice execution mechanisms that control how or when the advice is executed at an instrumentation point; and 3) new information sources that create new data at an instrumentation point for advice to use. To date we have been working on the first extension mechanism, that of new joinpoint

designators, and even this work is not yet complete. We describe ideas for all three below.

3.1 Pointcut Designator Extensibility

In AOP, a pointcut designator (PCD) is the fundamental entity that embodies a specific mechanism for identifying the points in a program's execution that support advice execution. We do not intend to ourselves create every possibly useful PCD for runtime monitoring, but rather we are providing a mechanism for creating new PCDs, while also building specific PCDs that are generally useful and explore the research ideas we are particularly interested in.

We envision a modular architecture of the runtime system where each pointcut designator in the pointcut language will have its own implementation of a common interface, acting as a plugin component to the base weaving engine. In this way new pointcut designators can be added by us or by others that implement unique monitoring-oriented joinpoint types.

The pointcut expression grammar will also need to be extended when introducing a new pointcut designator for a new joinpoint type. We envision several possibilities for this issue. One would be to use a tool like Polyglot [15] that supports extensible grammars. Another is to be careful in designing the grammar and its associated actions so as to make it cleanly extensible at the pointcut designator syntax. One issue will be how to handle the syntax of the designator arguments, which can be widely varying (e.g., many existing designators takes a regular expression to match package/class/method names). A final approach is to employ a grammar merging tool, such as the Antlr *gDiff* tool, and to allow the writer of a new designator to write a grammar for their designator, following particular naming rules to avoid conflicts, and then merging that grammar with the base aspect grammar. We plan to experiment with these approaches to evaluate their viability.

3.2 Advice Execution Extensibility

As described in Section 2, we take a multi-dimensioned view of the instrumentation needs of runtime monitoring, and note that advice execution (weaving) in exiting AOP frameworks have really only supported the code dimension. Our goal will be to have advice execution to be an extensible domain where we can experiment with methods to support all dimensions.

A typical approach in existing AOP systems for deciding where to weave advice for a compound pointcut expression is to allow each pointcut designator to produce a *shadow* that describes where it matches the program, and then to find the intersection (for the && operation) of these shadows as the concrete description of the entire pointcut. Because much of runtime monitoring can be supported with code instrumentation, this basic approach is useful in Teams, with extensions.

To handle the sampling ideas, two distinct alternative mechanisms are needed. One is for a pointcut designator to base its shadow *on other shadows*; for example, with a static joinpoint selection probability we would make a probabilistic decision at each shadowed code location whether to weave advice or not. Two is to be able to insert dynamic *residue* computation that makes a runtime decision whether to execute advice or not; for example, with a dynamic joinpoint probability, each time a joinpoint occurs a dynamic probabilistic decision is made to execute the advice or not.

To handle the time domain pointcut designators, we need mechanisms where advice can be triggered completely separate from what the program is doing. We expect that with careful interface design and modular framework construction we will be able to allow new advice mechanisms to attach to the runtime framework and access the information and advice handles they need to accomplish these novel and different ideas that are crucial to supporting broad runtime monitoring needs. For example, a time-domain advice plugin will register itself, read some meta-information about the advice intervals it needs to obey for the particular pointcut expressions being used, and then it will spawn a thread to act as an alarm, sleeping the appropriate intervals and then waking up to execute the necessary advice method(s).

Data domain pointcut designators may also need mechanisms that are entirely separate from the program. For example, while in Java object field accesses are easily mapped to specific bytecode instructions, for a less memory safe language such as C++ there is no such easy reduction; in this case a mechanism such as using page protection faults to catch memory references, or even CPU watchpoint registers which can trap accesses to specific addresses, might be needed to provide an efficient data domain designator.

Because we are focusing on monitoring as the domain for our AOP framework, we are not concerned with the advice interference problem. We intend our advice methods to be observational only and not to affect the program state. However, since the advice is in the POPL of the system and the user can be in control of how it gets compiled, there is nothing immediately preventing a user from accessing and modifying parts of their system within their advice. We view such use as outside Teams's intended scope and do not plan on providing any particular support for it. Possible indirect affects from e.g., timing modifications of concurrent behavior, are also beyond the scope of our concerns.

3.3 Joinpoint Information Extensibility

A limitation of the advice-as-POPL approach is that one cannot support any special extended syntax that might provide meta-level access to information about the joinpoint that the advice was executed on. We are limited only to providing joinpoint information through advice arguments. Another consideration is when and how to generate joinpoint information; if our framework always generates as much information as possible, this would be wasteful if the advice does not actually use it.

One related work, DiSL [10], described in more detail in Section 6, has some elegant solutions to this problem, separating the static and dynamic information parts and creating efficient mechanisms to produce the static information for the advice. We imagine that a pointcut declaration can include a parameter list that names the information the advice should be provided with. Thus it would be the responsibility of the aspect creator to declare what information the advice should be provided with; this is similar to how the AspectJ *args* designator works.

Each fundamental pointcut designator in the aspect language will have a set of information types that it could provide, and the pointcut expression compiler will verify that there is at least one pointcut designator in the expression that can generate the requested information type. For example, a method execution designator could produce the

class and method name of the executing method, and a basic block designator might produce, in addition to the enclosing class and method names, a unique block ID and the source line number on which it begins. If more than one can generate the requested type then the designator that produces the most specific data of the type will be the one that generates the joinpoint data. This information will then be passed as a plain typed parameter in the advice POPL. The advice method must be declared to accept the parameters that the pointcut expression declares.

4. PROTOTYPE JAVA TEAMS

We have begun building a first instantiation of TEAMS in Java. In this prototype we are currently ignoring the possibility of static weaving, considering it an optimization which we can later revisit.

The framework is embodied as a class loader agent which initializes itself and then triggers on each Java class being loaded. It uses the ASM bytecode manipulation library to perform the necessary weaving operations on the classes being loaded [4]. ASM has built-in functionality for constructing and accessing a control flow graph of a method, and so detailed code-level weaving is nicely doable using ASM. We use the Antlr parser generator to create the pointcut expression grammar and parser.

We have created rudimentary pointcut designators for method execution, method call, field access, and basic blocks. The runtime weaver loads the advice class which is named the same as the aspect name used in the pointcut definition, with a prefix "Aspect". Each advice is a static method named as the same name of the pointcut expression name, with a prefix indicating its weaving mode: "before" or "after". The runtime weaver is a JVM class loader extension that inspects each class being loaded and uses ASM to on-the-fly transform the bytecode to include invocations to the advice methods where necessary.

To specify an aspect and its advice in Java, we rely on naming conventions to connect the two. An example aspect is

```
aspect PrintGetters {
  pointcut GetterCall(): ( within(org.app) &&
    (call(* get*()) && withincode(* doWork()))
  );
}
```

which selects all calls to methods beginning with "get" that occur inside methods named "doWork" and are in the "org.app" package. The associated advice in plain Java is

```
public class AspectPrintGetters {
  private static long i = 0;
  public static void beforeGetterCall() {
    System.out.println("Getter execution #: "+i);
    i++;
  }
}
```

The aspect class is prefixed with "Aspect" and then is named the same as the aspect. Each pointcut expression has a name, in this case "GetterCall", and the advice associated with that pointcut expression is a method named with the same name and prefixed by the execution mechanism (e.g., before). We have considered using Java annotations but do not yet see an immediate benefit or reason (e.g., is there really a benefit to unconstrained class and method names?).

Our system is a prototype in the true sense of the word, in that we are still experimenting with various capabilities and expect that we will need ongoing re-design to support the full ideas outlined in Section 3. However, each of the pointcut designators we have created are implemented in a common manner as extensions, and we describe that process here.

To create a new PCD in TEAMS-Java, three things need to be done:

1. add the grammar clause(s) that will match the designator's syntax;
2. write the code that finds and indicates the matching execution points; and
3. write the code that provides unique data for the PCD.

Since we have not started working on the data extensibility portion of TEAMS, we only describe the first two steps here.

Our pointcut expression grammar is an Antlr grammar. We currently do not have an automated mechanism to extend the grammar, but adding a new designator is straightforward and mostly mechanistic. A production rule for a generic *designator* has an OR'd set of clauses, each a production rule for one specific PCD; a clause for the new PCD is needed, which is generally just one nonterminal for the specific PCD (e.g., *call* for our method call PCD). Then a new production rule for the new PCD is needed. This rule must embody the designator plus any arguments that it might have. For the *call* example, in full Antlr syntax this looks like

```
call returns[Designator value]:
  { $value = new Designator();
    $value.setName("call"); }
  'call' '('
  e=methodSignaturePattern {$value.setArgument(e);}
  ')'
 ;
```

Curly braces surround embedded Java actions and are the same for all PCD rules; the syntax of the PCD is simply the literal "call", the literal parentheses, and the single PCD argument which is the nonterminal *methodSignaturePattern*, which matches a RegEx-style pattern describing methods to match (this is similar to AspectJ and other AOP implementations). We provide some existing nonterminals for standard AOP designator arguments, plus simple strings and numerical arguments. If a PCD needs something different, more Antlr clauses may need to be created for its arguments, but we think this will be rare. The Java actions simply create an object for TEAMS to use as it processes the expression.

Secondly, the creator of the new PCD must write code that embodies what that PCD is: to match program execution points where instrumentation can be attached. Ultimately we will be exploring multiple different views of a program execution (dimensions), but for now we focus on the most immediate and used view, that of the code being executed. TEAMS uses the ASM bytecode manipulation framework for inserting the instrumentation into the program, and so the new designator code must use some of the ASM mechanisms to discover and find the execution points that the designator matches. The code to implement a new PCD must

Table 1: Execution overhead for TEAMS.

Benchmark	No Instrumentation	TEAMS JPs Exec (count)	TEAMS Time (Sec)	TEAMS Per JP (uSec)	AspectJ JPs Exec (count)	AspectJ Time (Sec)	AspectJ Per JP (uSec)
Xalan, execution	14.89	17348	18.55	211	16114	16.91	125
Xalan, call	14.89	17.2M	18.57	0.214	21.8M	19.5	0.211
Xalan, fieldAccess	14.89	402M	24.89	0.025	253M	17.53	0.010
Xalan, basicblock	14.89	219M	21.65	0.031	n/a	n/a	n/a
H2, execution	38.59	110M	42.25	0.033	96.8M	47.08	0.088
H2, call	38.59	3.36M	41.44	0.848	3.43M	40.90	0.673
H2, fieldAccess	38.59	1.05B	46.46	0.0075	205M	42.51	0.019
H2, basicblock	38.59	4.3B	119.5	0.019	n/a	n/a	n/a

implement the interface *PCDShadowMatcher*. For now, this interface is:

```
public interface PCDShadowMatcher {
 public  Set<JoinPointShadow>  match(
        MethodNode methodNode, ArgumentList args );
}
```

The new PCD must implement just one method, *match*. This method takes as its first argument an ASM object that represents a method in a class; thus *match()* is called once for each method of each class that is loaded. The second argument are the PCD arguments from the pointcut expression. *Match()* returns a set of bytecode intervals that match the PCD, marked by the positions of the first and last instructions in the interval. Many PCDs, such as the call PCD, might have just one instruction in each interval, but some, such as a hypothetical loop body PCD, could have a longer sequence of instructions that match. Note that a PCD creator in TEAMS-Java must learn and use the ASM API to implement their PCD's functionality; TEAMS is intended to hide such complexity from the TEAMS user, but needs to expose it to the TEAMS extender.

5. EVALUATION

Table 1 shows some very preliminary execution performance evaluations results for TEAMS. The programs used, *Xalan* and *H2*, are taken from the DaCapo Java benchmark suite (9.12-bach release) [3], run within the DaCapo framework but using the Unix *time* command to get user-space execution times. Where AspectJ has an equivalent pointcut designator, we try to reproduce as closely as possible the same results as in TEAMS (our system is still in prototype stage and so may not select exactly the same joinpoints as AspectJ). We do not show the full pointcut expressions, but we used various *within* clauses to control the portions of the programs that were instrumented.

The results show the benchmark execution time without any instrumentation, and then the number of advice executions, total execution time, and per joinpoint execution time for each of TEAMS and AspectJ. Advice bodies in both contained only a simple increment of a counter variable, plus a check to print out the counter on the final execution of the advice (which we determined beforehand on a previous fully-traced execution; this was too avoid having the body optimized away).

Firstly, we should note that even in this prototype we are already able to do something AspectJ could not, that is add advice to basic blocks. While seemingly small, this moving past existing AOP systems is exactly our motivation for the entire TEAMS effort.

To compare TEAMS and AspectJ, the per-joinpoint times are most useful, since they represent the cost per individual advice execution in each framework. While in a single column the per-joinpoint times vary greatly, looking across comparable TEAMS and AspectJ joinpoint types, both systems exhibit similar overheads. This supports our conclusion that TEAMS can be an effective instrumentation framework. The per-joinpoint overheads vary greatly between the experiments mainly because the number of joinpoints executed in each experiment also varies greatly, from a low of a few thousand up to a few billion. Since the variations are similar in each of TEAMS and AspectJ, we estimate that this is due to less startup amortization in the lower joinpoint counts and also probably less dynamic optimization within the JVMs.

6. RELATED WORK

There is much recent activity and novel ideas for extending AOP in a variety of manners, and several approaches to understanding the fundamental ideas in AOP that relate to the ideas we present here. While we focus on contrasting the ideas we plan on investigating in this project with their work, we also recognize that this body of previous work has many good, existing approaches to solving particular problems that we can build upon and leverage for greater success in our project.

As noted earlier, Binder et al. [2] have discussed the need for AOP to better support runtime monitoring, and have a body of work in doing so. Their latest step is DiSL, a domain-specific language for Java bytecode instrumentation, that uses AOP ideas in its design [10]. DiSL is targeting virtually the exact same problem as we are, that of providing high level mechanisms to accomplish low-level instrumentation, and it embodies some very elegant solutions to some of the problems, such as using Java annotations for multiple purposes (advice, new join point designators, and others), allowing instrumentation to be written in Java code (annotated static methods), and providing efficient static and dynamic information to instrumentation (extensible instrumentation contexts, weave-time static information evaluation, synthetic local advice variables, etc.). DiSL is code-centric, however, whereas our hope is to move beyond code-centric weaving to support the other dimensions of runtime monitoring (data, sampling, and time). DiSL only supports before/after advice execution models (including exception throwing), whereas we will investigate further advice execu-

tion models, especially in relation to concurrency issues.

A very novel approach to creating instrumentation for a particular dynamic analysis is that of PTQL, a program trace query language [8]. The design of PTQL is based on SQL, and is intended to mimic the act of writing a database query for writing the specification of the information needed from a program execution for a particular dynamic analysis. The rich part of their work is the fact that the program instrumentation is automatically generated to produce just the information that the given query requires, and this instrumentation is richly optimized to avoid unnecessary instrumentation or computation. PTQL, as far as we can tell, is also focused on code-centric instrumentation.

Dyer and Rajan [6] have investigated new AOP infrastructure ideas, explicitly working on arguing for more extensive join point models (thus allowing more pointcut designators) and embodying those in an intermediate language and virtual machine support for weaving. Rajan has continued to press forward with typed event (EVT) based aspect oriented programming (including program instrumentation) [16].

So many of the successful applications of AOP are program instrumentation and analysis that they would be too numerous to try to list; our goal is to support more fundamental instrumentation capabilities for these applications to be pushed even further. For example, the *Monitor-Oriented Programming* [11] framework is an elegant system for creating formal analyses that provide runtime verification of particular properties, but its implementation is limited by what AspectJ supports.If we can offer MOP more extensive access to the program behavior then it becomes more useful and capable of verifying more properties of an executing program. RuleR [1] is another formalism that eventually needs underlying instrumentation support, and analysis systems like RoadRUNNER [7], even though built on their own instrumentation, could instead use, and then be extended, by taking advantage of Teams.

7. CONCLUSION

The Teams framework presented here is in its earliest prototype form, and yet it appears to have promise for supporting the directions we hope to go. Our vision is to use the elegant, formal, notations of AOP to support a broad variety of the instrumentation needs that occur in runtime monitoring. Along with some standard AOP pointcut designators, shown here was one sample pointcut designator that provided a more detailed code-based joinpoint than most AOP frameworks do, the basic block PCD. We intend to not only support further code-centric designators, but also support designators that move away from a code-centric view and incorporate such instrumentation needs as stochastic sampling and even time-domain sampling.

In the future we hope that Teams will become a useful tool for both researchers and practitioners who need to implement some custom instrumentation for their own purposes but do not desire to build it themselves from scratch.

8. REFERENCES

[1] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based Runtime Verification. In *Verification, Model Checking, and Abstract Interpretation (VMCAI'04), LNCS 2937*, pages 44–57, Jan. 2004.

[2] W. Binder, P. Moret, D. Ansaloni, A. Sarimbekov, A. Yokokawa, and E. Tanter. Towards a Domain-Specific Aspect Language for Dynamic Program Analysis: Position Paper. In *Proceedings of the sixth annual workshop on Domain-specific aspect languages*, DSAL '11, pages 9–11, New York, NY, USA, 2011. ACM.

[3] Blackburn et al. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, Oct. 2006. ACM Press.

[4] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A Code Manipulation Tool to Implement Adaptable Systems. In *Adaptable and Extensible Component Systems*, Nov. 2002.

[5] B. Buck and J. K. Hollingsworth. An API for Runtime Code Patching. *Int. J. High Perform. Comput. Appl.*, 14(4):317–329, 2000.

[6] R. Dyer and H. Rajan. Nu: A Dynamic Aspect-Oriented Intermediate Language Model and Virtual Machine for Flexible Runtime Adaptation. In *AOSD '08: Proceedings of the 7th international conference on Aspect-oriented software development*, pages 191–202, 2008.

[7] C. Flanagan and S. N. Freund. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In *Proc. 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '10, pages 1–8, New York, NY, USA, 2010. ACM.

[8] S. F. Goldsmith, R. O'Callahan, and A. Aiken. Relational Queries over Program Traces. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 385–402, New York, NY, USA, 2005. ACM.

[9] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[10] L. Marek, Y. Zheng, D. Ansaloni, W. Binder, Z. Qi, and P. Tuma. DiSL: An Extensible Language for Efficient and Comprehensive Dynamic Program Analysis. In *Proceedings of the seventh workshop on Domain-Specific Aspect Languages*, DSAL '12, pages 27–28, New York, NY, USA, 2012. ACM.

[11] P. O. Meredith, D. Jin, F. Chen, and G. Roşu. Efficient Monitoring of Parametric Context-Free Patterns. *Automated Software Engg.*, 17(2):149–180, June 2010.

[12] A. Nusayr and J. Cook. AOP for the Domain of Runtime Monitoring: Breaking Out of the Code-Based Model. In *Proc. 2009 AOSD Workshop on Domain-Specific Aspect Languages*, page 4pp, 2009.

[13] A. Nusayr and J. Cook. Extending AOP to Support Broad Runtime Monitoring Needs. In *Proc. 2009 Int'l Conf. on Software Engineering and Knowledge Engineering (SEKE)*, 2009. to appear.

[14] A. Nusayr and J. Cook. Using AOP for Detailed Runtime Monitoring Instrumentation. In *Proc. 2009 ISSTA Workshop on Dynamic Analysis (WODA)*, page 7pp, 2009.

[15] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An Extensible Compiler Framework for Java. In *Proc. 12th International Conference on Compiler Construction (LNCS 2622)*, pages 138–152, Apr. 2003.

[16] H. Rajan, G. T. Leavens, R. Dyer, and M. Bagherzadeh. Modularizing Crosscutting Concerns with Ptolemy. In *Proceedings of the tenth international conference on Aspect-oriented software development companion*, AOSD '11, pages 61–62, New York, NY, USA, 2011. ACM.

[17] H. Rajan and K. Sullivan. Aspect Language Features for Concern Coverage Profiling. In *AOSD '05: Proc. 4th international conference on Aspect-oriented software development*, pages 181–191, 2005.