Implementing Scoped Method Tracing with ContextJS

Jens Lincke Robert Krahn Robert Hirschfeld
Hasso-Plattner-Institut
Universität Potsdam, Germany
{firstname.surname}@hpi.uni-potsdam.de

ABSTRACT

Customized method tracers can be a valuable tool for debugging and program comprehension. They allow to declaratively specify what parts of the call graph should be captured and are an alternative to tedious manual debugging techniques. Method tracers are easy to implement in dynamic languages but avoiding multiple method instrumentation and recursion in the client code can become complex. In this paper we show how Context-oriented Programming (COP) can be leveraged to address such issues. Our approach is based on ContextJS, a COP implementation for JavaScript, which provides scoping mechanisms and an infrastructure for method instrumentation. These abstractions allow to separate target and tracer logic so that self-referentiality is avoided.

1. INTRODUCTION

Tracing is a debugging and program comprehension technique programmers refer to when simple *printf* style logging and manually stepping through the debugger become to tedious.

Dynamic analysis tools may help gaining insights in the dynamic behavior of programs. But when such tools are not available or programmers encounter specific needs they might employ a custom tracer. Implementing tracers in dynamic languages like JavaScript is easier than implementing them with less reflective languages like *C* or *Java*. JavaScript's reflective capabilities are expressive enough to implement a method call tracer directly by manipulating objects. Modifying source or binary code or instrumenting the virtual machine is not needed.

However, creating such dynamic and reflective meta programs can also be challenging. When the tracer is defined and executed in the same environment as the code under development it should, for example, not accidentally capture its own execution. Using Context-oriented programming (COP) [2,4] can help to avoid such issues by expressing such meta code as a separate concern.

Context-oriented Programming provides infrastructure and scoping mechanisms to directly describe such concerns in a tracer. Dynamic scoping by `withLayers` and `withoutLayers` in ContextJS [9] our COP implementation for JavaScript allows to separate target and tracer code so that the tracer does not accidentally trace itself.

Tracing behavior is a typical *homogenous crosscutting concern* [1] which means that one piece of code should be executed in several places. In contrast to *Aspect-oriented Programming* (AOP) [7], COP has no explicit support for expressing such concerns, but standard meta programming facilities in JavaScript allow to overcome such limitations.

In this paper, we make the following contributions:

- We describe how scoping of behavioral adaptations with the (de-)activation of COP layers makes the implementation of a method tracing easy
- We show how COP infrastructure can be leveraged to avoid multiple method instrumentations
- We demonstrate how in dynamic languages homogenous concerns can be represented by COP layers using simple meta programming

The remainder of the paper is structured as follows. Section 2 discusses the problems of implementing a simple method tracer in JavaScript. Section 3 gives a short introduction of COP and ContextJS and Section 4 demonstrates how ContextJS can be used to address these problems. Section 5 shows two examples of using customized tracing. Section 6 discusses related work and Section 7 concludes.

2. TRACING WITH JAVASCRIPT

In the following we demonstrate problems that occur when building a custom tracer from scratch in a dynamic language such as JavaScript. Listing 1 defines two JavaScript objects `Target` and `Transcript`. The object `Target` should be instrumented to capture its method invocations.[1]

Implementing method call tracing can be accomplished by JavaScript's build in meta programming facilities. In JavaScript everything is build out of objects and functions, so methods are defined by putting functions in object properties. The difference to normal function calls is that the `this` pseudo variable is bound to the object at call time (in Figure 1, `m2` is defined in Line 3 and called in Line 20). Listing 2 shows how adding additional behavior to methods `m1`

[1]The examples be tested out online under `http://www.lively-kernel.org/repository/webwerkstatt/demos/contextjs/SimpleObjectTracing.xhtml` *(visited 2011-04-20)*

```
Target = new Object();
Target.m1 = function(a) { return a * 2};
Target.m2 = function() { return this.m1(this.p)};
Target.toString = function() {return "Target"}
Target.print = function() {
  Transcript.show(this + ", p = " +  this.p)
};

// Definition of simple tool object
Transcript = new Object();
Transcript.toString = function() { return "Transcript" };
Transcript.items = [];
Transcript.show = function(s) {
  alert(s);
  this.items.push(s);
}

// Usage
Target.p = 3;
Target.m2()    // evaluates to 6
Target.print()  // "p = 3" is displayed
```

Listing 1: Definition of an example object

```
 1    var orgM1 = Target.m1;
 2    Target.m1 = function(a) {
 3      Transcript.show("m1 " + a);
 4      return orgM1.apply(this, arguments)
 5    }
 6    var orgM2 = Target.m2;
 7    Target.m2 = function() {
 8      Transcript.show("m2");
 9      return orgM2.apply(this, arguments)
10    }
```

Listing 2: Wrapping methods around other methods in JavaScript. This allows adding behavior without changing the original source.

and `m2` can be accomplished without changing the source code. The original functions of that method are wrapped in a function that executes new behavior (Line 3) before evaluating the original function stored in `orgM1`. Abstracting this example further leads to the method `trace` as shown in Listing 3.

The method `trace` allows to trace arbitrary methods invocations in objects shown in Listing 4. The tracer works but includes some unobvious problems. First, instrumenting method `m1` twice results in capturing that method `m1` two times in the Transcript (see Listing 5). The tracer has no knowledge that it has instrumented that method already.

Furthermore, the `Tracer` might accidentally trace itself indirectly. Listing 6 shows how the method `show` of the `Transcript` object is instrumented. But the trace code itself makes use of the Transcript (see Line 6 in Figure 3). This leads to an unexpected endless recursion when the `Tracer` calls the `Transcript` which then calls the `Tracer` and so on.

The same problems occur when the `Tracer` is further generalized to automatically trace all methods of an object. The `traceObject` method in Listing 7 implements this exemplary. Multiple objects with a common prototype may be traced at once by installing tracers in a prototype object. Since classes are implemented this way in JavaScript, discussing the tracing of objects is sufficient. The `toString` method is automatically instrumented by the `traceObject` method. Since the `toString` method is used in the `Tracer`'s `log` method

```
 1    Tracer = new Object();
 2    Tracer.log = function(object, methodName, args) {
 3      Transcript.show(object + "'s " + methodName +
 4        " was called with " + args.length + " arguments" )
 5    };
 6
 7    Tracer.trace = function(object, methodName) {
 8      var orgFunc = object[methodName];
 9      object[methodName] = function() {
10        Tracer.log(this, methodName, arguments);
11        return orgFunc.apply(this, arguments)
12      }
13    }
```

Listing 3: Definition of a simple tracer in JavaScript

```
Tracer.trace(Target, "m1")
Tracer.trace(Target, "m2")
Tracer.trace(Target, "print")
// Call methods after Tracing
Target.m2()
Target.print()
```

Listing 4: Trace individual methods

```
(A)
Target's m2 was called with 0 arguments
Target's m1 was called with 1 arguments

(B)
Target's m2 was called with 0 arguments
Target's m2 was called with 0 arguments
Target's m1 was called with 1 arguments
Target's m1 was called with 1 arguments
```

Listing 5: (A) Transcript of calling `m2` (B) after accidentally instrumenting `m1` and `m2` twice

```
Tracer.trace(Transcript, "show")
Target.print() // ERROR because of endless recursion
```

Listing 6: Error when the "Transcript" should be traced

```
Tracer.traceObject = function(object) {
  for (var name in object)  {
    if (object.hasOwnProperty(name) &&
      (typeof object[name] == "function")) {
      alert("trace " + name)
      this.trace(object, name)
    }
  }
}
Tracer.traceObject(Target)
```

Listing 7: Usage of JavaScript introspection to trace all non-inherited methods of an object.

the tracer ends up in an unexpected endless recursion again.

To address such issues tracing code has to be self-reflective to ensure that it does not trace itself. A solution is to check inside every traced method if it is used in the context of a tracer. *Context-oriented programming* provides such expressiveness.

3. CONTEXT-ORIENTED PROGRAMMING

Context-oriented programming [2, 4] (COP) extends object-oriented programming by providing dedicated language abstractions for defining and composing variations to basic program behavior. Behavioral variations are encapsulated by *layers*, modules that can crosscut classes or in the case of JavaScript objects. Layers can be dynamically de-/activated—and composed with other layers—for the dynamic extent of a code block. This mechanism allows for scoping behavioral variations to specific control flows.

ContextJS is a JavaScript language extension for Context-oriented programming. It is implemented as a library and allows for defining behavioral variations for JavaScript objects, that can be (de-)activated depending on context. ContextJS supports various approaches for scoping behavioral adaptations such as global, dynamic, instance-specific and structural scoping [9]. ContextJS changes only the execution of adapted methods in a system. This means objects and methods that no layer refines run totally unaffected.

Listing 8 shows how:

(A) a `TranscriptLayer` can be created.

(B) the behavior of the object `Target` is refined

(C) the behavior variation is activated for the execution of a code block

(D) the behavior variation is deactivated for the execution of a code block

The layer (de-)activations follows a stack like discipline [2]. Inner method deactivations cancel out outer layer activations and vice vera. Furthermore the whole layer composition can be customized by the objects themselves in ContextJS. We discussed the open implementation of such layer activation strategies like the scoping of behavioral variations to specific object structures in a previous paper [9].

```
// (A) layer creation
cop.create("TranscriptLayer")

// (B) object refinement
TranscriptLayer.refineObject(Target, {
  m1: function(a) {
    Transcript.show("m1 " + a)
    return cop.proceed(a)
  },
  m2: function(a) {
    Transcript.show("m2")
    return cop.proceed()
  },
})

// (C) layer activation
withLayers([TraceLayer], function() {
  Target.m2()
})

// (D) layer deactivation
withoutLayers([TraceLayer], function() {
  Target.m2()
})
```

Listing 8: ContextJS Syntax Example

```
1    cop.create("TraceLayer")
2    Tracer.trace = function(object, methodName) {
3      var layeredMethodDef = {};
4      layeredMethodDef[methodName] = function() {
5        var args = arguments;
6        var obj = this;
7        withoutLayers([TraceLayer], function() {
8          Tracer.log(obj, methodName, args);
9        })
10       cop.proceed.apply(this, args)
11     }
12     TraceLayer.refineObject(object, layeredMethodDef);
13   }
```

Listing 9: Implementing the trace function with ContextJS

4. CONTEXTJS TRACER

ContextJS can (de-)activate behavior variations for specific control flows. We can define tracing behavior in a layer and deactivate it when processing the method invocation in the `Tracer` itself. The definition of tracing as a separated concern is similar to the plain JavaScript version in Listing 3. Listing 9 shows how the `trace` method generates a function and stores it as partial method in the `TraceLayer`. The `cop.proceed.apply(this, args)` statement proceeds to the next layer (in most cases the original implementation) if there are no other active layers.

When using ContextJS-based `trace`, the instrumentation expression `Tracer.trace(Target, "m1")` can be called as often as needed. Since ContextJS handles method wrapping, the tracing code cannot accidentally be installed multiple times.

As a result of representing the tracing concern as a layer the tracing can be activated as needed as shown in Listing 10.

The second problem described in Section 2 is that the original tracer can accidentally use methods that trace itself. This can be taken care of by putting the instrumentation code inside a `withoutLayers` statement as shown in Line 7 of Listing 9. Listing 11 shows how `Transcript`'s `show` method

```
Tracer.traceObject(Target)

Target.m2() // without tracing

withLayers([TraceLayer], function() {
  Target.m2() // with active tracing
})
```

Listing 10: Installing the tracer and calling `m2` with and without activated TraceLayer

```
Tracer.trace(Transcript, "show")
withLayers([TraceLayer], function() {
  Target.print()
})
```

Listing 11: Tracing the Transcript without running into an endless recursion

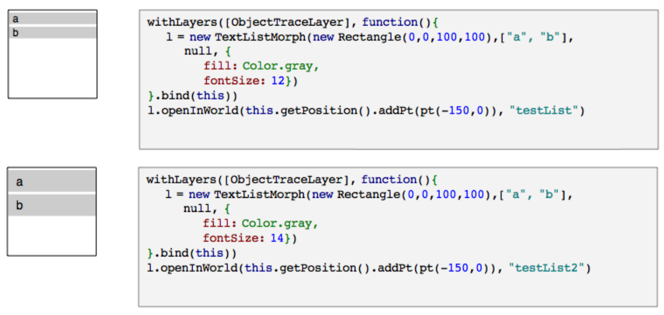
can now be instrumented without running into the endless regression. The `TracingLayer` is deactivated when called from the `Tracer`.

5. EXAMPLES

This section shows two example applications of the ContextJS tracer. The examples are taken from our work with the Web-based development environment Lively Kernel [6, 8, 10]. During our development we employ ContextJS on a regular basis and use the project as a source and test-bed for new application scenarios of COP.

5.1 Finding a Font Size Bug in Lively Kernel

The first example demonstrates how customized tracing can be used to find bugs that are not easily debuggable with standard tools. The issue we wanted to debug was that changing the font size of lists resulted in unexpected big list items as shown in Figure 1. To figure out the reason for the unexpected behavior we had to debug the Lively Kernel list abstractions. With the help of the customized tracer in Listing 12 we found out where the extent of the items changed and that the padding property was responsible as shown in Figure 2.



```
withLayers([ObjectTraceLayer], function(){
   l = new TextListMorph(new Rectangle(0,0,100,100),["a", "b"],
      null, {
         fill: Color.gray,
         fontSize: 12})
}.bind(this))
l.openInWorld(this.getPosition().addPt(pt(-150,0)), "testList")
```

```
withLayers([ObjectTraceLayer], function(){
   l = new TextListMorph(new Rectangle(0,0,100,100),["a", "b"],
      null, {
         fill: Color.gray,
         fontSize: 14})
}.bind(this))
l.openInWorld(this.getPosition().addPt(pt(-150,0)), "testList2")
```

Figure 1: Reproduction of setFontSize Bug in a Lively Kernel Web-based development environment. The list items got very big with a 14pt font compared with 12pt.

```
ObjectTracer.instrument([TextMorph]);
ObjectTracer.current().logEnterMethod =
  function (obj, methodName, args, config) {
    withoutLayers([ObjectTraceLayer], function() {
      if (this.ignoreList.include(methodName))
      return;
      var info = this.indentString() ;
      info += obj.constructor.name;
      if (config && config.category) {
        info += " [" + config.category +"] " };
      info += methodName;
      if (obj && obj.name) {
        info += " name=" + obj.name +" " };
      if (obj.getExtent) {
        try {
          info += " extent=" + obj.getExtent();
        }  catch(e) {  }
      };
      this.log(info )
      this.stackDepth ++;
    }.bind(this));
  }
```

Listing 12: A customized tracing method that produced the trace in Figure 2. It uses an `ObjectTracer` which is similar to the simpler one discussed in Section 4.

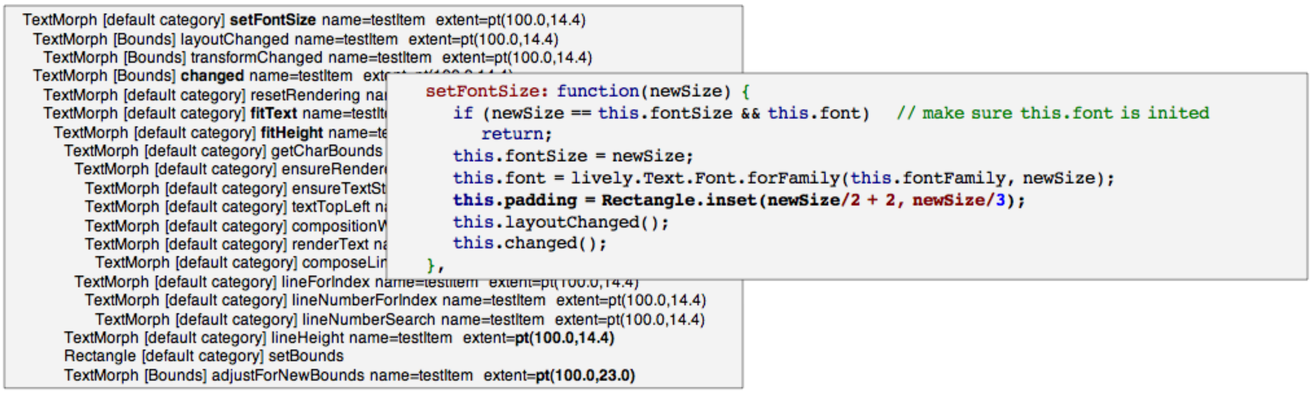
Using a debugger to step through the code would have been possible, however, using it to understand such complex behavior can be tedious. Traditional debuggers force the user to decide which methods to step over and which to step into. Without having an intimate knowledge of the code to debug this approach can require to restart time consuming debugging sessions several times. Using our customized tracing shown in Listing 12, we could declaratively adjust and refine the subject and location for what we were looking for. Repeating a run with the failing example was effortless because no manual user interaction was required.

Since the development environment Lively Kernel is self-sufficient it is crucial to separate tool logic from the source code under development. In the example above, `TextMorph` objects were instrumented. Since these objects were also used in the development environment, the ability to scope tracing to the actual execution of test snippets was important.

5.2 Understanding Layout Behavior

The second example shows how tracing was employed for understanding and profiling a new layout algorithm. We used tracing to visualize how graphical objects (`Morphs`) interact in the layouting process. We were only interested in methods of the class `Morph`, ignoring for example all methods in classes like `Point`, `Rectangle`, `Array` or `String`. The customized tracing and profiling did not only show only those methods, it further colored the lines in the trace so that morphs could be identified by their color as shown in Figure 3. Using a profiler in a meta level like the WebKit JavaScript tool suite, the profiling tools cannot be customized at runtime. Such profiles often show an overwhelming amount of information and developers cannot easily adapt them to mitigate those problems.

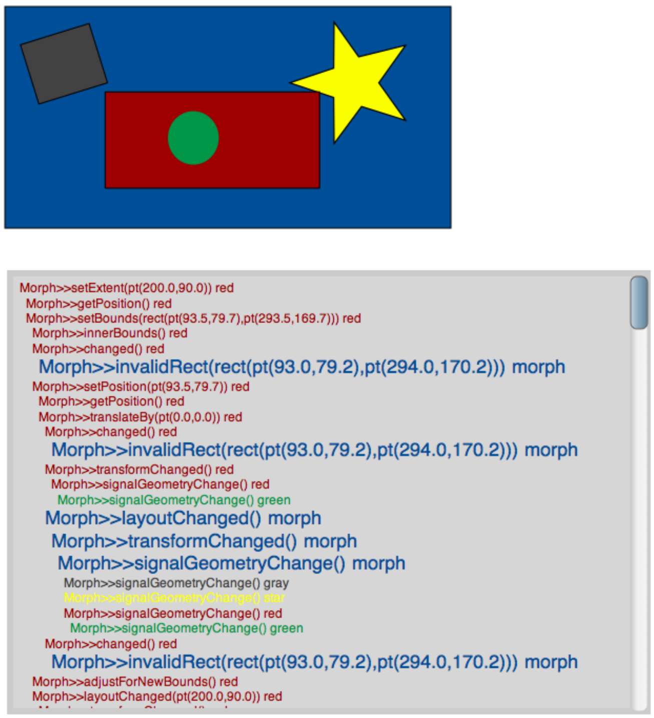
By using context specific scoping, tools can be separated from the domain objects like in external tool suites but can still be directly customizable.



```
TextMorph [default category] setFontSize name=testItem  extent=pt(100.0,14.4)
  TextMorph [Bounds] layoutChanged name=testItem  extent=pt(100.0,14.4)
    TextMorph [Bounds] transformChanged name=testItem  extent=pt(100.0,14.4)
  TextMorph [Bounds] changed name=testItem  ext
    TextMorph [default category] resetRendering na
    TextMorph [default category] fitText name=testIt
      TextMorph [default category] fitHeight name=te
        TextMorph [default category] getCharBounds
          TextMorph [default category] ensureRender
            TextMorph [default category] ensureTextSt
            TextMorph [default category] textTopLeft n
            TextMorph [default category] compositionW
            TextMorph [default category] renderText n
              TextMorph [default category] composeLin
          TextMorph [default category] lineForIndex name=testItem  extent=pt(100.0,14.4)
            TextMorph [default category] lineNumberForIndex name=testItem  extent=pt(100.0,14.4)
              TextMorph [default category] lineNumberSearch name=testItem  extent=pt(100.0,14.4)
        TextMorph [default category] lineHeight name=testItem  extent=pt(100.0,14.4)
        Rectangle [default category] setBounds
        TextMorph [Bounds] adjustForNewBounds name=testItem  extent=pt(100.0,23.0)
```

```
setFontSize: function(newSize) {
    if (newSize == this.fontSize && this.font)   // make sure this.font is inited
        return;
    this.fontSize = newSize;
    this.font = lively.Text.Font.forFamily(this.fontFamily, newSize);
    this.padding = Rectangle.inset(newSize/2 + 2, newSize/3);
    this.layoutChanged();
    this.changed();
},
```

Figure 2: Dynamic analysis and source code of the setFontSize bug in Figure 1. The interesting places where highlighted by the developer in the debugging process for documenting the issue.



```
Morph>>setExtent(pt(200.0,90.0)) red
 Morph>>getPosition() red
 Morph>>setBounds(rect(pt(93.5,79.7),pt(293.5,169.7))) red
  Morph>>innerBounds() red
  Morph>>changed() red
   Morph>>invalidRect(rect(pt(93.0,79.2),pt(294.0,170.2))) morph
  Morph>>setPosition(pt(93.5,79.7)) red
   Morph>>getPosition() red
   Morph>>translateBy(pt(0.0,0.0)) red
    Morph>>changed() red
     Morph>>invalidRect(rect(pt(93.0,79.2),pt(294.0,170.2))) morph
    Morph>>transformChanged() red
     Morph>>signalGeometryChange() red
      Morph>>signalGeometryChange() green
    Morph>>layoutChanged() morph
     Morph>>transformChanged() morph
      Morph>>signalGeometryChange() morph
       Morph>>signalGeometryChange() gray
       Morph>>signalGeometryChange() star
       Morph>>signalGeometryChange() red
        Morph>>signalGeometryChange() green
    Morph>>changed() red
     Morph>>invalidRect(rect(pt(93.0,79.2),pt(294.0,170.2))) morph
  Morph>>adjustForNewBounds() red
  Morph>>layoutChanged(pt(200.0,90.0)) red
```

Figure 3: Tracing the method setExtent of Morphs (graphical objects) in the Lively Kernel. The tracing code is customized to produce colored text that matches color of the graphical object.

6. RELATED WORK

A practical example how meta programming can be used in Web-based development environments [6] is a profiler implemented by wrapping JavaScript methods. The profiler is a Web-browser independent tool for analyzing JavaScript performance. Different to our approach presented profiler does prevent multiple instrumentations and end-less recursions when instrumenting code used in the profiler.

Tracing is a typical use case for Aspect-oriented Programming (AOP) [7]. In the reverse engineering tool ARE [3] AspectJ is used to instrument method calls. These calls are later filtered out by a pluggable system. Programs are instrumented by linking the tool as a library into the program under development. Different to our approach ARE has to instrument every method call because the scope of the tracing advice cannot be scoped at runtime.

Frameworks like Valgrind [12] and Pin [11] allow to instrument programs at runtime. This allows developers to build customized dynamic analysis tools. These frameworks instrument programs at the machine level and not at the object level. These frameworks avoid reentrance and recursion problems that might be caused from tools calling code which is under instrumentation by executing the analysis tools with a separate set of libraries [11].

7. CONCLUSION

In this paper we showed how a customized method tracer can be implemented in a dynamic language like JavaScript. As contributions we showed how COP technology can be used to scope the tracing so that the tracer does not accidentally trace its own custom code. We further showed how simple meta programming can be used to express homogenous concerns in our ContextJS COP implementation. In two examples from our work with the Lively Kernel, we showed how customizable tracers can be a valuable tool for program comprehension, debugging, and profiling.

Tracing is an excellent use case for the `withoutLayers` construct, which provides clean and reliable mechanisms to separate instrumented code from the instrumentation itself. In environments where the target system and the instrumentation code are strongly separated this is not so much an issue. In self supporting-systems [5] like the Lively Kernel, it is very important that tools like tracers are self aware so that they do not measure themselves or produce endless regressions. Putting tracing code into a layer that can be (de-)activated depending on the context solves this problem elegantly.

Tracing is a typical case of homogeneous cross-cutting concerns, which normally cannot be expressed by COP. By using standard JavaScript meta programming this could be overcome. COP implementations in less dynamic environments might not be used in this way.

In future work we will analyze how other cross-cutting concerns like an *Undo* history in Lively Kernel are easy to implement with COP.

8. REFERENCES

[1] S. Apel, D. Batory, and M. Rosenmüller. On the Structure of Crosscutting Concerns: Using Aspects or Collaborations. In *GPCE Workshop on Aspect-Oriented Product Line Engineering (AOPLE)*, 2006.

[2] P. Costanza and R. Hirschfeld. Language Constructs for Context-oriented Programming: An Overview of ContextL. In *DLS '05: Proceedings of the 2005 symposium on Dynamic languages*, pages 1–10, New York, NY, USA, 2005. ACM.

[3] T. Gschwind and J. Oberleitner. Improving Dynamic Data Analysis with Aspect-Oriented Programming. *European Conference on Software Maintenance and Reengineering*, 0:259, 2003.

[4] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented Programming. *Journal of Object Technology*, 7(3):125–151, March - April 2008.

[5] R. Hirschfeld and K. Rose, editors. *Self-Sustaining Systems, First Workshop, S3 2008, Potsdam, Germany, May 15-16, 2008, Revised Selected Papers*, volume 5146 of *Lecture Notes in Computer Science*. Springer, 2008.

[6] D. Ingalls, K. Palacz, S. Uhler, A. Taivalsaari, and T. Mikkonen. The Lively Kernel A Self-Supporting System on a Web Page. In R. Hirschfeld and K. Rose, editors, *S3 2008*, LNCS 5146. Springer-Verlag Berlin Heidelberg, 2008.

[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented Programming. In *Ecoop 1997, Proceedings 11th European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, 1997.

[8] R. Krahn, D. Ingalls, R. Hirschfeld, J. Lincke, and K. Palacz. Lively Wiki A Development Environment for Creating and Sharing Active Web Content. In *WikiSym '09*. ACM, 2009.

[9] J. Lincke, M. Appeltauer, B. Steinert, and R. Hirschfeld. An Open Implementation for Context-oriented Layer Composition in ContextJS. *Science of Computer Programming*, In Press, Corrected Proof, 2011.

[10] J. Lincke, R. Krahn, D. Ingalls, and R. Hirschfeld. Lively Fabrik - A Web-based End-user Programming Environment. In *Proceedings of the Conference on Creating, Connecting and Collaborating through Computing (C5) 2009*, Tokyo, Japan, 2009. IEEE.

[11] 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.

[12] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2):44 – 66, 2003. RV '2003, Run-time Verification (Satellite Workshop of CAV '03).