Program Execution Environments as Contextual Values

Markus Raab
Vienna University of Technology
Institute of Computer Languages
Austria
markus.raab@complang.tuwien.ac.at

Franz Puntigam
Vienna University of Technology
Institute of Computer Languages
Austria
franz@complang.tuwien.ac.at

ABSTRACT

Context-oriented programming (COP) provides a very intuitive way to handle run-time behavior varying in several dimensions. However, COP usually requires major language extensions and implies a considerable performance loss. To avoid language extensions we propose to specify program execution environments as contextual values in separate units. A tool translates such specifications into C++ classes usable in the rest of the program. Without the need of multiple dispatch, the performance can largely profit from simple caching. Furthermore, it is easy to support debugging and store contextual values in configuration files.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming; H.2.3 [**Languages**]: Persistent programming languages; D.2.5 [**Testing and Debugging**]: Debugging aids

General Terms

Design, Languages, Performance

Keywords

configuration specification, code generation, context oriented programming, contextual value, program execution environment, debugging, configuration file, persistence, benchmark

1. INTRODUCTION

Context-oriented programming (COP) is a technique providing multi-dimensional separation of concerns [8]. Code is executed in a dynamic scope depending on layers.

Contextual values are variables depending on the context in which they are read and modified [9]. They can be limited in their visibility and are no longer potentially global.

We propose to use program execution environments (in a broad sense) as contextual values. They include the environment variables as well as command line arguments and values retrieved from configuration files.

COP '14, Uppsala, Sweden
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2861-6/14/07...$15.00
http://dx.doi.org/10.1145/2637066.2637074 .

For example, let us use external configuration settings via `getenv` to determine the program execution environment. We have to be careful: At the presence of command line arguments or after dynamic reconfiguration the settings valid in the new context differ from those received with `getenv`. In some parts of the program we have more information and the context is more specific than in others. To reduce the danger of assuming wrong context information it is desirable to use program execution environments based on COP.

We propose to specify the values of the program execution environment in a separate unit. Such specifications contain placeholders, each representing a dimension of the context:

```
[/%language%/person/greeting]
  type=String
```

In this example, `greeting` is a contextual value of type `String` and `%language%` a placeholder to be substituted in contextual interpretations. A code generator produces the classes `Person` and `Greeting`. Through them we can activate layers (thereby changing the view, maybe by setting the language to German) and modify the value of `greeting`.

These are our contributions:

- We enable programmers and systems engineers to apply contextual values in their familiar language.
- We provide library support to use contextual values specified in configuration files, command lines, etc.
- We take care to uniquely name values with respect to their context for a better support of debugging.
- We briefly analyze the performance situation and suggest a simple, but effective optimization technique.

These contributions are of practical relevance. Many other approaches require major language extensions or depend on non-standard language features like multiple dispatch and dynamic scoping [4, 3]. It is well-known that COP often has a considerable negative impact on the performance [1] and ease of debugging. Our approach addresses these problems. Still, our approach is flexible and open for extensions. In our work we use C++, but it should be possible to do almost the same in other languages as well.

2. CONCEPTION

Our concept evolved from Elektra, a configuration management system developed and commercially applied by one of the authors [7]. Elektra is essentially a library providing access to configuration data according to several standards.

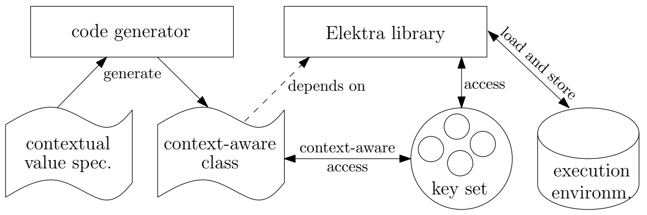

Figure 1: Integration of Elektra

It is by no means a COP system. But, the Elektra library turned out to be useful to administer contextual values.

Figure 1 shows how Elektra is integrated in our approach: A tool generates a context-aware class for each specification of a contextual value. This class uses the Elektra library to access context information stored in the key set, the data structure of Elektra. Hidden from the user, Elektra takes care that every contextual value will get its correct value derived from the program execution environment. Through the library we can also reload the environment and persistently store contextual values.

To put context-aware objects (i.e., instances of context-aware classes) into a specific context, programmers define layers and specify which contextual value depends on which layer. The activation and deactivation of layers at run-time will have direct effect to each of those contextual values.

2.1 How to use Context Information

We slightly expand the above example:

```
[/%language%/%country%/%dialect%/person/greeting]
  type=String
[/%country%/person/visits]
  type=Integer
  default=0
```

Here, `greeting` and `visits` are contextual values, but `person` is not. Classes are generated for `Person`, `Greeting` and `Visits` (class names begin with uppercase letters), where `Person` has the member variables `greeting` and `visits` of the types `Greeting` and `Visits`, respectively. Since no default value is specified for `greeting`, we have to define the value somewhere else. We do so in a configuration file containing a key/value pair for each value in each context:

```
/%/%/%/person/greeting=Hi!
/German/%/%/person/greeting=Guten Tag!
/German/Austria/%/person/greeting=Servus!
/German/Austria/traditional/person/greeting=Griaß enk!
```

Here, `%` denotes an empty name. We assume this configuration file to be loaded at the begin of the program. Next, we have to specify layers as hand-written classes like this one:

```
class CountryAustriaLayer : public Layer
{
public:
    string id() const { return "country"; }
    string operator()() const { return "Austria"; }
};
```

For simple use we overload the function call operator. Of course, the strings can be computed and need not be constant. We assume that there are similar classes for other countries, languages and dialects as well.

The following function shows the use of contextual values:

```
void visit(Person & p) {
    p.context().with<CountryAustriaLayer>()
               .with<LanguageGermanLayer>()([&] {
        cout << "visit " << ++p.visits
             << " in " << p.context()["country"]
             << ": " << p.greeting << endl;
    });
    cout << p.greeting << endl;
}
```

Every contextual value allows access to its context using the method `context()`. Each use of the member function template `with` specifies a layer for the next application of the overloaded function call operator “()” to a lambda expression. These layers are active only while executing the lambda expression. An execution of the function `visit` produces this output:

```
visit 1 in Austria: Servus!
Hi!
```

Language and country are known, but the dialect is unknown when producing the first line. No context information is available when producing the second line. As a side-effect, the value of `visits` is incremented by one, but only in the context of Austria, not in any other context.

2.2 More on Layers

Simple implementations of layers like `CountryAustriaLayer` just return a constant string. That is the most common case. In the following example, the country is determined by invoking `lookupCountry` (e.g., utilizing the current GPS position) whenever the layer is activated:

```
class CountryGPSLayer : public Layer
{
public:
    CountryGPSLayer() : m_country(lookupCountry()) {}
    string id() const { return "country"; }
    string operator()() const { return m_country; }
private:
    string m_country;
};
```

The context itself sometimes depends on contextual values. For example, we use a profile as contextual value, and a set of other contextual values depends on the profile. To some extent we can specify in the program execution environment which profile shall be active:

```
[/%application%/profile]
  type=String
  opt=p
  opt/long=profile
```

Because of `opt` and `opt/long` this contextual value is preferably taken from the command line options `-p` and `--profile` than from the configuration file. Now we can use a simple layer that will return the contextual value passed by the profile:

```
class ProfileLayer : public Layer
{
public:
    ProfileLayer(String const & profile) :
        m_profile(profile) {}
    string id() const { return "profile"; }
    string operator()() const { return m_profile; }
private:
    String const & m_profile; // contextual value
};
```

Such a profile is typically valid for the whole application. The following implementation of the `main` function shows how to set up the whole system. We have to create the context and a key set as provided by the Elektra library. The key set gets initialized with data specified in configuration files (automatically found by Elektra) and command line arguments. Our generator creates (beside the classes corresponding to the names in the execution environment) also a class `Environment` which provides access to the top-level entities like `profile` and `person`. We need an instance of `Environment` depending on the context and key set. Using the member function template `activate` of the context, we activate two layers globally. Essentially, `activate` does the same as `with`, but at the global level instead of locally.

```
int main(int argc, char**argv)
{
    KeySet ks;
    parseConfigfiles(ks);
    parseCommandline(ks, argc, argv);
    Context c;
    Environment env(ks, c);
    c.activate<MainApplicationLayer>();
    c.activate<ProfileLayer>(env.profile);
    // the rest of the program
    // e.g., visit(env.person);
}
```

2.3 Code Generation

Above code is written by the user. Let us have a look at the code generated from specifications, e.g. for `Visits`:

```
class Visits : public Integer
{
public:
    Visits(KeySet & ks, Context & context) :
        Integer(ks, context,
            Key("", KEY_VALUE,
                "/%country%/person/visits",
                KEY_META, "default", "0", KEY_END)) {}
    using Integer::operator=;
};
```

Even though such classes could be written directly by programmers, the specification gives us a much more concise and powerful way to achieve the same.

2.4 Predefined Classes

The class `Context` plays a central role in activating and deactivating layers as well as interpreting contextual values. Classes of contextual values and `Context` are organized in an observer pattern [5], where `Context` is the subject, and the contextual values are the observers. This is, the context informs contextual values about changes. Since placeholders in the specifications state which layers a contextual value depends on, the contextual values can subscribe themselves to the needed notifications.

```
class Context : public Subject
{
public:
    template <typename L> void activate(...);
    template <typename L> void deactivate(...);
    template <typename L> Context & with(...);
    template <typename L> Context & without(...);
    string evaluate(string const & spec) const;
    Context & operator()(function const & f);
    //...
};
```

As shown above, `activate` and `with` activate new layers (globally or for a specific block). Accordingly, `deactivate` and `without` deactivate layers. Arguments are forwarded to a constructor of `L`, where `L` is derived from `Layer` and gets instantiated within the function. In this context, `evaluate` interprets a name with placeholders. Thereby, a placeholder is ignored if no matching layer id is present. The function call operator `Context::operator()` calls the function given as argument in the scoped context set up by the use of `with()` and `without()`.

Each layer must implement this interface:

```
class Layer
{
public:
    virtual string id() const = 0;
    virtual string operator()() const = 0;
};
```

The return value of `id` corresponds to the name of the placeholder. This identifier must be globally unique. All instances of `Layer` with the same `id` are expected to represent the same layer, and on activation of one of them will override that of the others. The method `Layer::operator()` will be used during the context evaluation as the layers contribution.

2.5 Ambiguity

Even though we do not use the multiple dispatch techniques, the multi-dimensional value lookup still introduces a source for ambiguity. Suppose we have a specification `/%manufacturer%/%model%/serial` and the configuration values:

```
/hp/%/serial=1000
/%/EliteBook 8570/serial=1234
```

When both layers with the results `hp` and `EliteBook 8570` are active, either of the values could be taken. Because no variant is more intuitive than the other, we decided not to support such ambiguities. Hence, `%` stands only for layers with no output or empty layers.

For more flexibility, specifications can contain *placeholder groups* of the syntax `%layerid layerid ...%`. Corresponding keys can specify names of none, one, two, . . . or all layers, filling up the layer ids from left to right. As usual, in the key `%` stands for no name. One name is given by `%name`, two names by `%name%name`, and so on. This is, each name belonging to a placeholder group begins with `%`.

E.g., in the specification `/%manufacturer model%/serial`, a non-empty layer for `model` can be specified only if also a layer for `manufacturer` is specified, and no ambiguity arises.

As a further example, let us consider the specification `%manufacturer category model%` with the following configuration:

```
/%hp/serial=1XXX
/%hp%Notebook%EliteBook 8570/serial=1321
/%hp%MobileWorkstation%EliteBook 8570/serial=1234
```

When the layer `category` is inactive, we unambiguously get the serial number `1XXX`, regardless of the active layer `model`.

3. IMPLEMENTATION ISSUES

In this section we will evaluate four implementation techniques that can be used with our approach. Every technique fulfills the requirement that whenever the contextual value is accessed it will correctly deliver its value under the interpretation of the current context. In all techniques, needed

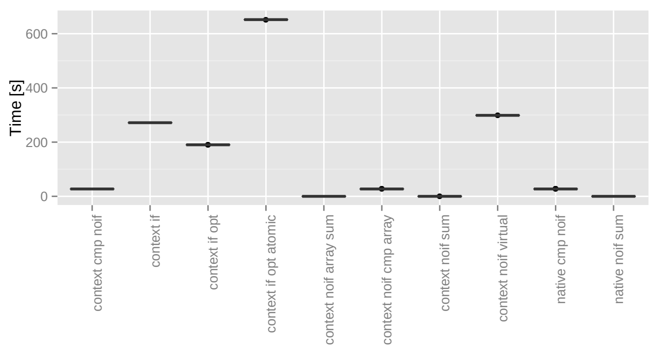

Figure 2: benchmark

updates in the event of context changes are done using the observer pattern as outlined in the Section 2.

Current techniques implementing COP features need multiple dispatch [4] or counter based control flow [3]. Usually, the costs of COP features are compared in applications using such highly dynamic concepts. Conclusions based on these comparisons are valid for many applications, but are not suitable for others, including algorithmic applications in a more static setting. We will evaluate different implementations under the assumption that accesses to contextual values occur much more often than layer activations.

We conducted the benchmarks on a hp® EliteBook 8570w using the CPU Intel® Core™ i7-3740QM @ 2.70GHz. The operating system is GNU/Linux Debian Wheezy 7.5. We used, unless noted otherwise, the gcc compiler Debian 4.7.2-5 with the options -std=c++11, -O2 and -Dopt=unlikely. We measured the time using `gettimeofday`. We executed each benchmark eleven times and show the median value (except in the graphs where all data are displayed).

Our micro-benchmark represents algorithmic and arithmetical problems that frequently access values:

```
Integer::type add_native(uint32_t const & i1,
                         uint32_t const & i2)
{ return i1+i2; }
```

This function is called 100 billion (=`iterations`) times in the following loop:

```
for (long long i=0; i<iterations; ++i)
{
    x ^= add_native(val, val);
}
```

This loop takes 27.16 seconds (see data labeled “native cmp noif” in figures 2 and 3). When the values are summed up instead of using xor, the loop takes 0.00 seconds (see “native noif sum” in Figure 2) because the compiler replaces the loop by a single arithmetical operation.

We compare this native performance with our approach, using the same loop, but using the contextual value `Integer` instead of the native value `uint32_t`:

```
Integer::type add_contextual(Integer const & i1,
                             Integer const & i2)
{ return i1+i2; }
```

3.1 (Atomic) Branches

A naïve approach to detect context changes is by checking a tidy flag on every access of the contextual value:

```
operator uint32_t() const
{ if(m_context_changed) { update(); }
  return m_cache; }
```

A *type conversation operator* in C++ allows contextual values to be used where a `uint32_t` is expected. This specific implementation adds two additional branches for each call of `add_contextual` with devastating results: The loop needs 271.62 seconds (it is one hundred times slower, see “context if” in Figure 2). The runtime can be improved to 190.13 seconds (see “context if opt” in Figure 2) by giving the compiler optimization hints to specify which conditional branch is taken more often.

The use of an `if` for every access yields additional benefits. Firstly, it makes context changes lazy and avoids updates for rarely used contextual values. Secondly, when the contextual value uses `std::atomic<bool>` instead of `bool` for `m_context_changed`, the contextual values are multi-thread safe. Unfortunately, atomicity does not come without extra costs: Needing 651.92 seconds the runtime is more than doubled when using an `atomic` type (see “context if opt atomic” in Figure 2). Surprisingly, with clang (version 3.5-1 exp1 using option -O3) the runtime is only 81.42 seconds both for `std::atomic<bool>` and `volatile bool`. But, the results are still far from desired.

Another major drawback of the `if`-solution is that the compiler cannot optimize away arithmetic loops. Therefore, we did not use it in our implementation.

3.2 Virtual Function Calls

A further implementation technique to provide contextual values is by switching classes at runtime. To call the correct class, a virtual function call is needed:

```
virtual operator uint32_t() const
{ return m_cache; }
```

Virtual function calls usually outperform switch statements. But, virtual function calls make some optimizations (especially inlining) impossible, leaving us with a runtime of 298.8 seconds (see “context noif virtual” in Figure 2). Optimizations that completely get rid of the loop are impossible. Hence, we did not use virtual function calls, too.

3.3 Member Arrays

Another way is to have an array that contains values for every context. Context switches will then change the array index. On access, the array is accessed using the given index:

```
operator uint32_t() const
{ return g_arr[m_cache]; }
```

Arrays give us very promising results: 27.16 seconds (see “context noif cmp array” in Figure 2 and 3). Additionally, optimizations can completely eliminate the loop.

But an array to cache all possibilities has an important drawback: Done in a naïve way it consumes a large amount of memory for each contextual value, because the number of layer combinations is huge. We left the exploration of this technique as a future work.

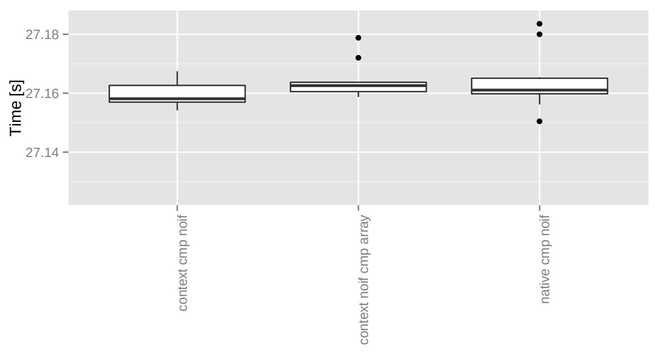

Figure 3: comparison to native performance

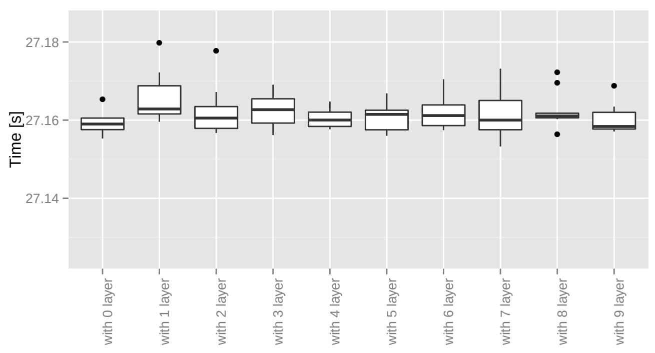

Figure 4: access with active layers

3.4 Member Variables

The simplest implementation is the use of one memory cell per contextual value and returning its content directly:

```
operator uint32_t() const
{ return m_cache; }
```

The runtime of this technique has a median of 27.16 seconds (see "context cmp noif" in figures 2 and 3). Still, the technique uses only a minimal amount of additionally memory (one native type per contextual value). So we decided to use this simple implementation technique.

Of course, returning values of member variables are not the only costs of COP. We must also compute the values when activating layers. But, we expect those costs to be rather independent of the implementation technique.

4. EVALUATION

Context-oriented programming typically has a major drawback: Performance penalties of 75% (cj and ContextL performed better in specific cases) to 99% [1] currently makes COP unattractive for some types of real world applications. Performance is one of the major advantages of our approach. There are also others like improved debugging support.

4.1 Performance

One slogan in the C++ community is "you don't pay for what you don't use". As we see in Figure 3 our implementation ("context cmp noif") has zero overhead compared with native non-contextual values although (or because) of the simple implementation based on member variables. The reason is that the compiler can perform aggressive optimizations eliminating the performance overhead. There is only small memory overhead consisting of a native value and a reference to the context for each contextual value.

We evaluate the impact of the number of active layers at runtime by activating zero to nine layers. We use the same loop as before in the scoped block of the `with` statements. For example, using two layers the setup looks as follows:

```
s.context().with<Layer1>().with<Layer2>()([&] {
    s.bm = value;
    Integer::type x = 0;
    for (long long i=0; i<iterations; ++i)
    { x ^= add_contextual(s.bm, s.bm); }
    dump << x << endl; });
```

As we see in Figure 4, increasing the number of active layers does not noticeably affect the performance. All differences are within 0.02 seconds. Because of the huge number of loop iterations (100 billion) the costs for accessing contextual values are dominant, those for activating layers are negligible.

4.2 Debugging support

Our specifications introduce names. The uniqueness of these names turned out to be very valuable for debugging.

Logging: Simple logging facilities can capture under which context a variable is accessed. We found bugs in the code easily by tracing all unique names on every access.

Backtraces: They are enriched by telling us unique names:

```
#3  0x0000000000407a56 in operator() at first.cpp:1521
    i = @0x7fffe36b69a0: { ...
      m_evaluated_name = "/german/germany/%/test" }
```

Breakpoints: They can respect the context as condition:

```
break 1520 if i.getEvaluatedName()
             .compare("/german/germany/%/test") == 0
```

Assertions: They can assure that a contextual value is in that context we think it is, e.g. the language is German:

```
assert(i.context()["language"] == "german");
assert(i.getEvaluatedName() == "/german/%/%/test");
```

The second variant is preferable. It implicitly ensures that *all* other layers affecting the contextual value are inactive. When the specification changes, the assertion will be triggered instead of pretending false safety.

Elektra 0.8.6 (see `http://www.libelektra.org`) includes the functionality proposed in the present paper. Our experience from using it shows that unique names have a positive influence on the readability of the code.

5. DISCUSSION AND FUTURE WORK

We already saw that the specification provides a short and easy-to-use syntax that need not be available in the host language. Additionally, it has the following advantages:

- Specifications can be used to generate further artifacts like configuration files and documentation.
- Language-independent specifications allows us to add support for new programming languages.

- The generation of contextual classes hides many nasty details like constructor parameters and assignment operations that would not be obvious to programmers.
- Using meta-data for contextual values as well as connecting observers is error-prone when done by hand.
- It would be tedious to write corresponding class hierarchies by hand. They typically consist of hundreds of classes, and their maintenance would be difficult.

The specification includes information about layers a contextual value depends on. It is in the hands of the programmer to design it properly. In performance critical code, unnecessary updates can be avoided. The specification allows us to declaratively describe the desired situation.

As a future work, the specification language can be improved in many ways. Maybe, it would be useful to provide a specification for layers, too.

5.1 Support of Persistence

Our approach interacts closely with the non-functional programming concern of persistence. By modifying configuration files, the user or systems engineer has complete control over initial values at startup time. The programmer can determine a default value in the specification that will be used when no persistent value is available, e.g. when a layer is active that was never used before.

As a future work, we will find a formalism to define and validate persistent data directly in the specification. More rigorous evaluations of the performance of layer switching, initial retrieval of the contextual values and real world applications also remain as further work.

6. RELATED WORK

Contextual values were proposed by Löwis et al. [10]. Tanter [9] investigated the topic in-depth. Different from our approach, those contextual values need changes in the Scheme interpreter. Tanter concentrates on call-by-value parameter passing. Although our approach also supports call-by-value parameter passing by copying values, it is only of limited interest to us because the connection to the program execution environment gets lost.

The Cartesian approach to context [6] uses a similar paradigm with a formal background. The main difference is the use of an n-dimensional table with lazy computation instead of our one-dimensional key set with eagerly computed values. While the Cartesian approach is theoretically more powerful, much beyond a declarative storage, it has the disadvantage that its contents can neither be eagerly computed nor serialized. However, serialization is necessary for handling program execution environments.

Costanza et al. [4] implemented ContextL as an extension to CLOS and rely on its features: dynamic class generation, multiple inheritance, dynamically scoped variables, and multiple dispatch. In most languages those dynamic features do not exist. From this feature list, C++ supports only multiple inheritance. The use of such features implies a significant overhead, especially because they rule out many possible optimizations that we relied upon in our work.

Dynamic aspect weaving (e.g., in the Steamloom virtual machine [2]) adds constructs for activation of partial program definitions. Different from our approach, it only works with a specific virtual machine.

7. CONCLUSION

We saw that, using our approach, the user can profit from zero overhead (without layer switches) on any number of active layers. A declarative specification of contextual values takes away the burden of writing many similar classes while ensuring static type safety. Additionally, unnecessary cache updates on layer switches are avoidable by specifying dependences between values and contexts.

Specifications in combination with active layers introduce unique names for all contextual values in each context. These names support new ways of debugging contextual values. Previously surprising behavior becomes obvious.

Elektra 0.8.6 (see `http://www.libelektra.org`) is freely available and can be used to try out the proposed approach.

8. REFERENCES

[1] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. A comparison of context-oriented programming languages. In *International Workshop on Context-Oriented Programming*, COP '09, NY, USA, 2009. ACM.

[2] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. Virtual machine support for dynamic join points. In *Proceedings of the 3rd International Conference on Aspect-oriented Software Development*, AOSD '04, pages 83–92, NY, USA, 2004. ACM.

[3] Christoph Bockisch, Sebastian Kanthak, Michael Haupt, Matthew Arnold, and Mira Mezini. Efficient control flow quantification. In *ACM SIGPLAN Notices*, volume 41, pages 125–138. ACM, 2006.

[4] Pascal Costanza, Robert Hirschfeld, and Wolfgang De Meuter. Efficient layer activation for switching context-dependent behavior. In DavidE. Lightfoot and Clemens Szyperski, editors, *Modular Programming Languages*, volume 4228 of *Lecture Notes in Computer Science*, pages 84–103. Springer, 2006.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[6] John Plaice and Blanca Mancilla. The cartesian approach to context. In *Proceedings of the 2Nd International Workshop on Context-Oriented Programming*, COP '10, NY, USA, 2010. ACM.

[7] Markus Raab. A modular approach to configuration storage. *Master's thesis, Vienna University of Technology*, 2010.

[8] Hans Schippers, Tim Molderez, and Dirk Janssens. A graph-based operational semantics for context-oriented programming. In *Proceedings of the 2Nd International Workshop on Context-Oriented Programming*, COP '10, NY, USA, 2010. ACM.

[9] Éric Tanter. Contextual values. In *Proceedings of the 2008 Symposium on Dynamic Languages*, DLS '08, pages 3:1–3:10, NY, USA, 2008. ACM.

[10] Martin von Löwis, Marcus Denker, and Oscar Nierstrasz. Context-oriented programming: Beyond layers. In *Proceedings of the 2007 International Conference on Dynamic Languages*, ICDL '07, pages 143–156, NY, USA, 2007. ACM.