TS4J: A Fluent Interface for Defining and Computing Typestate Analyses

Eric Bodden

Secure Software Engineering Group
EC SPRIDE—Fraunhofer SIT & TU Darmstadt
eric.bodden@sit.fraunhofer.de

Abstract

Typestate analyses determine whether a program's use of a given API obeys this API's usage constraints in the sense that the right methods are called on the right objects in the right order. Previously, we and others have described approaches that generate typestate analyses from textual finite-state property definitions written in specialized domain-specific languages. While such an approach is feasible, it requires a heavyweight compiler, hindering an effective integration into the programmer's development environment and thus often also into her software-development practice.

Here we explain the design of a pure-Java interface facilitating both the definition and evaluation of typestate analyses. The interface is *fluent*, a term coined by Eric Evans and Martin Fowler. Fluent interfaces provide the user with the possibility to write method-invocation chains that almost read like natural-language text, in our case allowing for a seemingly declarative style of typestate definitions. In all previously described approaches, however, fluent APIs are used to build configuration objects. In this work, for the first time we show how to design a fluent API in such a way that it also encapsulates actual computation, not just configuration.

We describe an implementation on top of Soot, Heros and Eclipse, which we are currently evaluating together with pilot customers in an industrial context at Fraunhofer SIT.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program Analysis

General Terms Algorithms, Design, Languages

Keywords Fluent interfaces, static analysis, dynamic analysis, typestate

SOAP '14, June 11 2014, Edinburgh, United Kingdom.

http://dx.doi.org/10.1145/2614628.2614634

1. Introduction

A typestate property [17] describes which operations are available on an object or a group of inter-related objects, depending on this object's or group's internal state, the typestate. For instance, programmers must not write to a connection handle that is currently in its "closed" state. Also, a programmer must refrain from further using an iterator in cases where the associated collection object was modified after the iterator has been created [7]. In past research, we and others [1, 5, 6, 8, 9] have described several domain-specific languages (DSLs) for the declarative definition of typestate properties. Listing 1 shows a definition of the pattern for fail-safe iterators mentioned above, formulated in the syntax of tracematches [1], a language extension to AspectJ [2]. The example defines three events `create_iter`, `call_next` and `update_source` along with a regular expression (line 11) that describes the set of event sequences after the tracematch body (line 12) should execute. Crucially, the tracematch executes its body only if all the matched events occurred on the same collection and iterator objects. While one could also imagine a simple, for instance Java-based, application interface (API) to define typestate properties, definitions in a domain-specific language such as tracematches have the advantage that they are relatively easy to read and comprehend, and are also fully declarative: the definition states *what* property should be analyzed for a given program, but not *how* the analysis must be performed. The user can thus abstract from implementation details unimportant to her.

Nevertheless, domain-specific languages have a number of problems on their own. First they require dedicated compiler support. A front-end must parse the typestate-property definition and, to give useful user feedback in cases of mis-specifications, it must also comprise a powerful type checker. In addition, a compiler backend must use the information of the property definition to parameterize a static or dynamic analysis approach that actually evaluates the property with respect to a given program. Building such a tool chain is a non-trivial engineering task and can easily take several person months. Another drawback is that, unless additional tool support is provided, the domain-specific lan-

```
public aspect FailSafeIter {
   pointcut collection_update(Collection c) :
    ( call(* java.util.Collection.add*(..)) ||
      call(* java.util.Collection.remove*(..)) ) &&
           target(c);
    tracematch(Collection c, Iterator i) {
      sym create_iter after returning(i) :
         call(* java.util.Collection.iterator()) &&
              target(c);
      sym call_next before:
         call(* java.util.Iterator.next()) && target(i);
      sym update_source after : collection_update(c);
     create_iter call_next* update_source+ call_next {
       throw new ConcurrentModificationException(c,i);
     }
   }
}
```

Listing 1: Tracematch definition for fail-safe iterators

guage will in all likelihood not integrate well with the software engineer's integrated development environment (IDE). The language might lack support for syntax highlighting, code completion and incremental compilation. For many developers this is a problem large enough to discourage them from using typestate specifications (or DSLs in general) in the first place.

In this paper we thus present the design of TS4J, a pure-Java API that supports typestate definitions in a seemingly declarative manner. In the scientific community, such APIs are frequently referred to as *internal DSLs*, i.e., DSLs that go without extensions to the syntax and semantics of the host language, Java in our case. Listing 3 shows the definition of a static analysis for the fail-safe iterator typestate property, written using TS4J. (We will explain the semantics later.) Because TS4J is based on pure Java, it supports syntax highlighting and Java's type checks. As we explain in more detail later, though, by following the design rules of so-called *fluent interfaces*, our architecture also allows a certain level of typechecks on the semantic level of the typestate definitions. The term "fluent interfaces" was coined by Eric Evans and Martin Fowler [10]. It describes a certain style of API design which allows users of the API to form method-call chains that almost read like natural-language sentences. By making clever use of Java interfaces, one can restrict the vocabulary and grammar of these sentences to the subset that has a well-defined semantics in the DSL.

Previous approaches to internal DSLs with fluent interfaces, however, restrict themselves to pure configuration. One of the most popular fluent interfaces, for instance, the jOOQ interface for defining type-safe SQL queries in Java [13], allows for queries such as the one shown in Listing 2. Here the calls to the methods `from`, `join`, ..., `orderBy` modify an internal configuration object which the API successfully hides from the user. The final call to `fetch` then

```
Result<Record> result =
create.select()
     .from(AUTHOR.as("a"))
     .join(BOOK.as("b")).on(a.ID.equal(b.AUTHOR_ID))
     .where(a.YEAR_OF_BIRTH.greaterThan(1920)
     .and(a.FIRST_NAME.equal("Paulo")))
     .orderBy(b.TITLE)
     .fetch();
```

Listing 2: Internal DSL for jOOQ

passes this configuration object to an SQL query evaluator that actually computes the query result. In result, the fluent interface is used for configuration but not for computation. With TS4J we show that it is possible to use a fluent interface also to define computation if that computation is written in the same host language and executes in the same process. The crucial ingredient that makes this possible is the injection of context into the objects managed by the fluent interface.

In summary, this paper describes the first fluent interface for the definition of typestate properties and, to the best of our knowledge, the first incarnation of a fluent interfaces that is used not just to implement the configuration but the actual implementation of the APIs behavior.

Section 2 further describes the concrete design and semantics of our fluent interface. Section 3 explains how we manage to have the interface implement behavior, not just configuration. We discuss related work in Section 5 and conclude in Section 6.

2. Fluent interface

Fluent interfaces are a generic way to produce internal DSLs in object-oriented programming languages such as Java. The developers of the jOOQ library mentioned above describe a generic conversion from a context-free grammar to a set of Java APIs forming an appropriate fluent interface for that grammar. The grammar is assumed to be described by a railroad diagram such as the one shown in Figure 1.

As they write [12], to generate an appropriate fluent-interface definition for any grammar given in this format one just needs to obey a set of simple rules:

- Every DSL keyword becomes a Java method.
- Every connection out of a keyword becomes an interface whose type the respective Java method returns.
- For a mandatory choice, where one cannot skip the next keyword, every keyword of that choice is a method in the given connection's interface. If only one keyword is possible, then there is only one method. In case of an "optional" keyword, the current interface extends the next one (with all its keywords / methods).

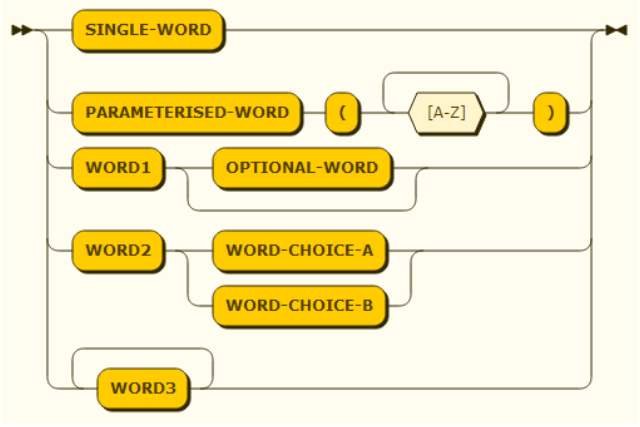

Figure 1: Railroad diagram, from [12]

- On a repetition of keywords (caused by loops in the diagram), the method representing the repeatable keyword returns the interface itself instead of the next interface.

We effectively follow this methodology to turn into Java interfaces and method declarations the grammar for TS4J, shown in Figure 2. For our first prototype we followed these conversion steps manually. We quickly realized, however, that over time one might want to adjust the DSL's grammar, which then results in the necessity to rearrange the related Java interfaces and their type hierarchy. This type of code evolution is not trivial, such that it might actually be beneficial to have the interface definitions generated automatically. The jOOQ project, in fact, does contain a code generator along those lines. However, it is specialized to SQL, using database schemas and not grammars as an input.

In addition to the generated or hand-written interface definitions, TS4J comprises a single class that implements all of the respective interfaces. Invoking a method such as `ifValueBoundTo` invokes an operation on an object of this class, and then returns the very same object, possibly then viewed through an interface different from the interface which the method was invoked on.

2.1 Interface semantics

After having shown the syntax of our fluent interface and how we construct its Java interfaces from the syntactic definition, we next describe the syntax and also its semantics in more detail. As the Grammar in Figure 2 shows, TS4J allows for the definition of a set of rules.

Location filters Each rule is inspected upon the analysis encountering a method call or return and, according to the set of defined location filters (`Loc`) will or won't apply at this call/return. In our current design, the parameters to `atCallTo` and `atReturnFrom` are method signatures in Soot's internal format (Lst. 3 lines 1–6). For the future, we envision using pointcut-like patterns [18] that would allow for a more concise selection of sets of method signatures. The special location filter `atReturnFromMethodOfStmt` will be explained later.

```
Start ::= Rule | Rule Start
Rule ::= Locs [Conds] Actions
Locs ::= Loc | Loc or Locs
Loc ::= atCallTo(String methodSig) |
        atReturnFrom(String methodSig) |
        atReturnFromMethodOfStmt(StmtId stmt)
Conds ::= Cond | Cond and Conds
Cond ::= ifValueBoundTo(Var var) Equals |
        ifInState(State state)
Equals ::= equalsThis | equalsReturnValue |
        equalsParameter(int index)
Actions ::= Action | Action and Actions
Action ::= SwitchState | TrackValue | StoreStmt | ReportError
SwitchState ::= toState(State targetState)
TrackValue ::= TrackWhat as(Var var)
TrackWhat ::= trackThis | trackReturnValue |
        trackParameter(int index)
StoreStmt ::= storeAsStmt(StmtId stmt)
ReportError ::= reportError ErrorLoc
ErrorLoc ::= here | atStmt(StmtId stmt)
```

Figure 2: Grammar for the proposed DSL

Condition filters When a rule applies because the signature matches, the system next evaluates its condition filters (`Cond`). Analysis configurations comprise three types of internal state:

a set of abstract values indexed by abstract variables,

an abstract state and

a set of statements indexed by a set of statement IDs.

Abstract variables and statement IDs both range over a user-defined `enum`. While this restricts analysis configurations to binding only a finite set of abstract values and statements, it (1) allows for a time and memory-efficient implementation using arrays, and (2) ensures termination of the induced static analysis because it guarantees a finite configuration space. The filter `ifValueBoundTo` checks whether a value previously bound to a given abstract variable equals the current call's receiver (`this`), return value or n-th parameter. In our implementation, equality is resolved using a generic taint analysis that follows assignment chains through the program and uses points-to information to resolve aliasing, but other implementations (including dynamic ones!) are possible. The filter `ifInState` retains only such configurations that are in the given abstract state.

Actions When all filters match, one or more actions are applied. The possible actions are:

switching the abstract state to a given target state,

tracking a value at the current statement by assigning it to a given abstract variable,

storing a statement for future reference, and

reporting an error condition to the user.

Tracking values and internal state is important to enable the condition filtering mentioned above. Storing statements is important for two reasons. First, when reporting an error, the developer can specify an error location (`ErrorLoc`), indexed by a statement identifier. This assumes that a statement reference has been previously stored under this identifier. (The internal state and conditions on this state can be used to assure that this is indeed the case.) Second, the user can use the special location filter `atReturnFromMethodOfStmt` which also accepts a statement identifier as an input. This mechanism is useful for being able to check bounded liveness properties. Assume that we wish to check that every object that implements the interface `java.io.Closeable` is closed eventually. Assuring this property globally would require a whole-program analysis, which is both inefficient and in most cases also too liberal for what the developer actually desires. In most situations, the developer would want to have the object closed within the scope of the execution of the method in which the object was created. In such situations the developer can use one rule to match on appropriate constructor calls, binding the call to a statement identifier such as `NEW_CLOSEABLE` and can then use a location filter `atReturnFromMethodOfStmt(NEW_CLOSEABLE)` to report an error if the respective object is still in its "open" state when the location filter matches (at the method return).

Our rigorous usage of Java interfaces and the design principles of fluent interfaces ensure that, at every given point in time, the software developer defining the rules can invoke only those methods that make sense in the current context. For instance, a rule definition cannot start with a call to `toState`, and `equalsThis` can only be called right after calls to `ifValueBoundTo(..)`. In that sense, fluent interfaces, to a certain extent, enforce typestate properties on their own.

2.2 Explanation of initial example

We next explain the effect of our initial example of the fail-safe iterator under the semantics stated above. The example rule shown in Listing 3 comprises three rules, one for the creation of new iterators (line 12), one for the modification of collections (line 16) and one for the usage of iterators (line 20). To create a rule, the user must invoke methods on a context object (here called `c` in line 11), which at the same time gives access to the fluent interface and encapsulates context information. Section 3 will explain why the latter is important. The first rule, upon encountering the creation of a new iterator through a call to `Collection.iterator()` (defined in lines 3–4), binds the receiver to `COLL` and the return value to `ITERATOR`, and also initializes the configuration's internal state to `INIT`. The second rule, at calls to `Collection.add()`, changes this state to `MODIFIED`, but only if the receiver object equals the value previously bound to `COLL`. Also it stores the current statement for future reference, under the identifier `MODIFICATION`. The last rule eventually applies at calls to `Iterator.next()`. It reports an error if the respective iterator object is in state `MODIFIED`, but only

```
1  String COLLECTION_ADD =
2  "<java.util.Collection: boolean add(java.lang.Object)>";
3  String NEW_ITERATOR =
4  "<java.util.Collection: java.util.Iterator iterator()>";
5  String ITERATOR_NEXT =
6  "<java.util.Iterator: java.lang.Object next()>";
7  enum Var { COLLECTION, ITERATOR };
8  enum State { INIT, MODIFIED };
9  enum StatementId { MODIFICATION };
10 ...
11 return c.
12   atCallTo(NEW_ITERATOR).
13     trackThis().as(COLL).
14     trackReturnValue().as(ITERATOR).
15     toState(INIT).
16   atCallTo(COLLECTION_ADD).
17     ifValueBoundTo(COLL).equalsThis().
18       toState(MODIFIED).
19       storeStmtAs(MODIFICATION).
20   atCallTo(ITERATOR_NEXT).
21     ifValueBoundTo(ITERATOR).equalsThis().and().
22     ifInState(MODIFIED).
23       reportError("Collection modified!").
24         atStmt(MODIFICATION);
```

Listing 3: The internal DSL proposed

if the iterator is the one previously bound to `ITERATOR`. The error is defined to be reported at the statement that modified the collection, i.e., the one previously stored under the identifier `MODIFICATION`. As an alternative, the developer could also simply have stated `here()` to report the error at the current location.

In this example, it is important to note that any change of the internal state of a configuration is reflected in `all` objects currently bound to that configuration. This is because states are associated with abstract configurations, not with the individual objects that they bind. In particular, note that the second rule changes the state for all configurations referring to the collection being modified. There can be many such configurations, for instance because any collection can have many live iterators at the same time, but also because any such iterators can have many static aliases, each of which will yield a separate configuration in our implementation. Changing the internal state of any of those configurations implicitly changes the state for the related iterator(s) as well, giving sense to the third rule: this rule refers only to the internal state of configurations related to iterators, abstracting from the collections they are bound to. For further reading on such multi-object properties, we refer the interested reader to related work [1, 4, 7, 15].

3. Implementing behavior

As we explained earlier, one particularity that sets our approach apart from previous approaches to fluent interfaces, is that the fluent interface is not just used for defining a con-

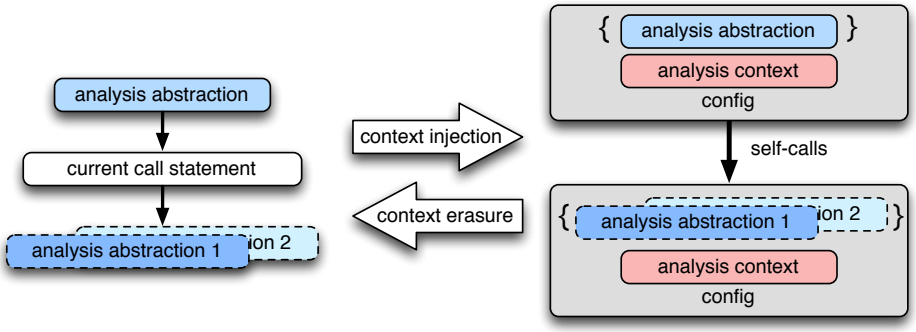

Figure 3: Workflow of the analysis implementation

figuration object, but instead is used to also implement the actual behavior of the rules that the developer defines.

As explained in the previous section, the definition of each analysis rule induces the creation of a single object that implements all possible Java interfaces constituting the fluent interface. All methods invoked on this object return the object itself, i.e., `this`, but with a declared interface type reflecting the methods that can be called next on this object, according to the grammar of the fluent interface. Figure 3 shows this object as "config", in Listing 3 it is called `c`.

At each call statement, TS4J instantiates a new config object, comprising (1) the current analysis configurations that reached this statement, and (2) an analysis context giving access to the current statement as well as context information including pointers to may-alias and must-alias analyses. The framework then passes this object to the rule definition provided by the user, for instance the rule shown in Listing 3. The self-calls that make up the rule-definitions cause the config object to filter and/or modify the abstraction that it holds. If all filters fail to match, then the config object returns itself with an unmodified abstraction, further setting an internal flag causing subsequent calls on the same config object to have no further effect. If a filter does match then the config object may modify its abstraction, depending on the semantics of the rule definitions. As shown in the figure, in the general case, a rule may cause the config to produce not just a single successor abstractions but multiple ones. In particular, this is the case when binding values to variables: in this case the config object produces a new abstraction with the value being bound, but also retains the old abstraction with the unbound value. This is necessary, for instance, in situations where a configuration referencing a collection is also bound to a specific iterator. Since the same collection can have other iterators in the future, a config with the iterator being unbound needs to be retained.

After the execution of the rule has finished, the config object is erased again, and only the resulting abstraction objects are retained. Those are then passed to a general IFDS [16] solver which takes care of propagating the abstractions further. Because our whole approach is based on IFDS, which requires separable flow functions anyway, it is known that individual abstractions can be processed separately. Hence, when multiple abstractions reach the same call statement, our implementation can simply wrap them individually into config objects one by one.

It is interesting to note that the processing of rules is really only necessary at method calls and returns, and it only needs to handle the abstract semantics of the given rules. In particular, all handling of aliasing and of the passing of call parameters and return values is performed transparently and does not need to be considered when writing or evaluating rules. In our implementation this manifests itself through the fact that only call-to-return-flow and return-flow functions trigger rule computations. In contrast, normal-flow functions handle assignments only, as do call-flow functions, which assign actual parameter names to formal parameter names.

4. Implementation

Figure 4 shows a screenshot of an example program within our current prototype, running with the analysis rule for fail-safe iterators. As explained before, the error is shown at the statement that modifies the collection. Note how the analysis handles inter-procedural flows and aliasing.

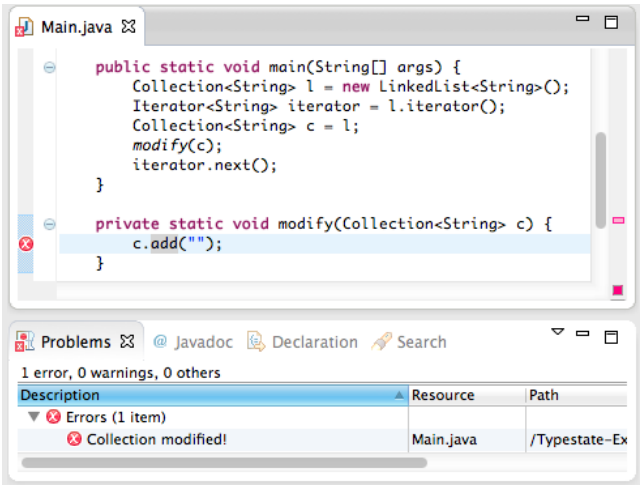

Figure 4: Screenshot of analysis implementation

The current prototype is implemented in the form of an Eclipse plugin, which itself exports an extension point for defining and plugging in novel analysis rules. To insert a new rule, the user simply extends this extension point within a separate Eclipse plugin and then implements a class with a single callback method `atCallToReturn(config)` or `atReturn(config)` respectively. The declared interface type of the parameter config is chosen such that it reflects exactly the methods defined through the first terminals reachable from the `Start` non-terminal of the grammar from Figure 2. Both methods have an interface return type implemented by all interfaces returned by all methods with which a valid rule can end. This assures that the user can only return config objects which have indeed received an appropriate set of self-calls, according to the grammar. In particular, the callback cannot simply return `config` without defining any rule at all, as `config` has a type incompatible with the return type.

5. Related Work

While fluent interfaces have found some attention in the practice of software engineering, their properties and design principles have not yet been the subject of academic study.

Existing commercial or successful open-source program-analysis tools typically follow one of two paths. Either they allow users to define rules through a graphical user interface or through a regular application interface. Both approaches have their problems. Definitions based on a user interface require good UI design to be helpful. Designing a good UI is costly. With new additions to the expressive power of rules, the UI support must be expanded as well. Further, such "visual" rule definitions cannot easily be processed with text-editing and version-control tools. Nevertheless, some existing tools such as IBM AppScan Source or the information-analysis tool Joana [11] follow this path. Definitions using regular APIs have the problem that they often lead to rule definitions that are both cumbersome to read and write. It is challenging to have the rule-definitions API stick to the essentials, such that the user is not overburdened by the size of the API. Fluent interfaces as we use them in our work allow for good ease of use because at any time they permit the user only those method invocations that make sense in the current context. Existing tools that allow rule-definitions or analysis-definitions using regular APIs include FindBugs [3] and Jlint [14].

6. Conclusion

We have shown a fluent Java interface for the definition and evaluation of typestate-analysis rules. While our implementation uses the rules for executing a static inter-procedural program analysis, we believe that also a use for dynamic analysis should be possible. The fluent interface hides all necessary implementation details from the user, and through the clever use of interface types, prevents the ill-formed definitions of rules.

References

[1] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding Trace Matching with Free Variables to AspectJ. In *OOPSLA*, pages 345–364, October 2005.

[2] AspectJ. `http://eclipse.org/aspectj/`.

[3] Nathaniel Ayewah, William Pugh, J David Morgenthaler, John Penix, and YuQian Zhou. Using findbugs on production software. In *OOPSLA companion*, pages 805–806. ACM, 2007.

[4] Eric Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *ICSE '10: International Conference on Software Engineering*, pages 5–14, New York, NY, USA, May 2010. ACM.

[5] Eric Bodden, Feng Chen, and Grigore Roşu. Dependent advice: A general approach to optimizing history-based aspects. In *AOSD*, pages 3–14, March 2009.

[6] Eric Bodden and Laurie Hendren. The clara framework for hybrid typestate analysis. *International Journal on Software Tools for Technology Transfer (STTT)*, 14:307–326, 2012. 10.1007/s10009-010-0183-5.

[7] Eric Bodden, Laurie J. Hendren, and Ondřej Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *ECOOP*, volume 4609 of *LNCS*, pages 525–549, 2007.

[8] Eric Bodden, Patrick Lam, and Laurie Hendren. Clara: a framework for statically evaluating finite-state runtime monitors. In *RV 2010*, volume 6418 of *LNCS*, pages 74–88. Springer, November 2010.

[9] Feng Chen and Grigore Roşu. MOP: an efficient and generic runtime verification framework. In *OOPSLA*, pages 569–588, October 2007.

[10] Martin Fowler. FluentInterfaces, December 2005. `http://martinfowler.com/bliki/FluentInterface.html`.

[11] Dennis Giffhorn and Christian Hammer. Precise analysis of java programs using joana. In *SCAM 2008*, pages 267–268. IEEE, 2008.

[12] jOOQ team. The Java Fluent API Designer Crash Course, January 2012. `http://blog.jooq.org/2012/01/05/the-java-fluent-api-designer-crash-course/`.

[13] jOOQ team. jOOQ: Get Back in Control of Your SQL, March 2014. `http://www.jooq.org/`.

[14] Konstantin Knizhnik and Cyrille Artho. Jlint manual. *URL http://jlint. sourceforge. net*, 2002.

[15] Nomair A. Naeem and Ondřej Lhoták. Typestate-like analysis of multiple interacting objects. In *OOPSLA*, pages 347–366, October 2008.

[16] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95*, pages 49–61, 1995.

[17] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *TSE*, 12(1):157–171, January 1986.

[18] The AspectJ Team. The AspectJ 5 Development Kit Developer's Notebook, 2004.