A Formalism for Specification of Java API Interfaces

Davide Ancona
University of Genoa
davide.ancona@unige.it

Francesco Dagnino
University of Genoa
francesco.dagnino@dibris.unige.it

Luca Franceschini
University of Genoa
luca.franceschini@dibris.unige.it

CCS Concepts • **Software and its engineering → API languages**; **Specification languages**; *Software verification*;

Keywords Trace expressions, specification, Java, API, runtime verification

ACM Reference Format:
Davide Ancona, Francesco Dagnino, and Luca Franceschini. 2018. A Formalism for Specification of Java API Interfaces. In *Proceedings of 20th Workshop on Formal Techniques for Java-like Programs (FTfJP'18).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Introduction

The standard Java API defines a number of useful and widely used interfaces, where the flow of methods invoked on objects must follow certain patterns to ensure their correct use. Typically, such patterns are informally specified in the documentation with several rules, which often do not cover full details, and are scattered throughout the whole documentation, since they usually involve more methods which are invoked subsequently. Failing to follow such rules causes a throw of `IllegalStateException`, or similar exceptions. Let us consider, for instance, the specification of the following types defined by the standard API of Java 10 in module `java.base`.

`java.util.Iterator` The specification of `remove` states that `IllegalStateException` is thrown if the `next` method has not yet been called, or it has already been called after the last call to the `next` method. The documentation does not cover the situation when the method is called after method `next` has been called, and has thrown `NoSuchElementException`: also in this case `remove` should throw `IllegalStateException`. This example shows that rules on correct method invocation necessarily involve more methods (`remove` and `next` in this case), and often depend on the outcome of method calls (if `next` throws `NoSuchElemenetException`, then `remove` must throw `IllegalStateException` if subsequently invoked).

Another illegal flow concerns methods `hasNext` and `next`: invocation of method `next` is illegal if the previous call to `hasNext` has returned false. As in the previous case, this implicit rule depends on the outcome of a method call (`hasNext`). Finally, although the interface does not enforce it, always invoking `hasNext` before `next` is a best practice.

`java.util.regex.Matcher` Classes `Matcher` and `Pattern` provide support for regular expressions. The introductory documentation states that attempting to query a matcher before a successful match causes a throw of `IllegalStateException`; according to the same documentation, a successful match corresponds to the event "any of the method between `matches`, `lookingAt` and `find` has been called and returned true". By further reading the documentation in the methods specific sections, one can find the equivalent rule stating that, for all query methods of the matcher, `IllegalStateException` is thrown if no match has yet been attempted, or if the previous match operation failed. Finally, one can discover that some methods, as `reset` and `region`, reset the state of the matcher, and invalidate any previous successful match. Also in this case, understanding the rules for correct method call flows requires to carefully collect and suitably combine together all information scattered throughout the documentation.

Another interesting example concerns Stream API introduced in Java 8, considered in detail in the following sections.

In this work we informally present a specification language based on trace expressions which is more expressive than other formalisms [3, 4, 6] and show how it could be successfully employed to formally specify sophisticated rules on correct method invocation flows for Java API interfaces.

Trace Expressions

Trace expressions [1] are a formalism expressly devised for runtime verification purposes [5]. In this context, the system under test is observed and relevant events are collected in a trace encoding the single execution. A monitor then verifies the trace against the formal specification which defines the expected global behavior of the system. As such, a trace expression denotes the set of traces corresponding to all correct runs of the system, formally defined with a labeled transition system [2]. In this work events are method invocations on objects, with arguments and returned values (or exceptions).

The formalism features a rich set of operators: the empty trace ϵ; prefix $\theta : \tau$, denoting the set of traces starting with an event matching θ followed by a trace accepted by τ; concatenation $\tau_1 \cdot \tau_2$, intersection $\tau_1 \wedge \tau_2$ and union $\tau_1 \vee \tau_2$, with their intuitive meanings; shuffle $\tau_1 \mid \tau_2$ (a.k.a. interleaving), denoting the set of any shuffle of a trace accepted by τ_1 and

FTfJP'18, July 2018, Amsterdam, Netherlands

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

one accepted by τ_2. $\{x; \tau\}$ declares the variable x scoped in τ, allowing trace expressions to be parametric w.r.t. some value that will only be known at runtime [2]. Furthermore, in this work we use generic trace expressions, exploiting the Java-like syntax τ<x>, for the sake of reuse and readability.

A Formalization of the Java Stream API

In `java.util.stream` objects represent possibly unbounded sequences of data to be processed only once in a lazy way. The processing of a stream consists in a *pipeline* of operations specified through a functional interface. The Java specification describes two types of operations:

- **intermediate operations** transform the given stream into a new one, they describe the intermediate steps in a processing pipeline and are lazily evaluated;
- **terminal operations** trigger the whole pipeline, they usually do not produce a new stream and, once they are invoked, the stream is marked as consumed and hence it cannot be used anymore.

As for the examples presented in the previous section, also for streams the documentation contains elaborated rules to ensure that the flows of invoked methods follow certain patterns. The Java specification explicitly says that each stream should be used only once. This means that, after any operation, only the output stream can be used to specify further manipulations; in other words, branching in a processing pipeline is not allowed.

We can formalize this behaviour using trace expressions. The set $\mathcal{E}$ of events consists of three types of elements: $o(\mathsf{sid}_i, \mathsf{sid}_o)$, where o is the name of an intermediate operation, sid_i is (the identifier of) the input stream and sid_o is (the identifier of) the returned output stream; $c(\mathsf{sid})$, where c is the name of a terminal operation and sid is (the identifier of) the stream on which it is applied; $n(\mathsf{sid})$, stating that a new stream sid has been created. Accordingly, we use three event types matching, respectively: **new**(sid) all events of shape $n(\mathsf{sid})$; **intermediate**$(\mathsf{sid}_i, \mathsf{sid}_o)$ all events of shape $o(\mathsf{sid}_i, \mathsf{sid}_o)$ **terminal**(sid) all events of shape $c(\mathsf{sid})$. Correct method invocation flows for streams can be specified by the following trace expression τ_{new}:

$$\tau_{new} = \epsilon \vee (\{\mathsf{sid}; \mathbf{new}(\mathsf{sid}) : \tau_{op}\text{<sid>}\} \mid \tau_{new})$$
$$\tau_{op}\text{<sid>} = \{\mathsf{sid}'; \mathbf{intermediate}(\mathsf{sid}, \mathsf{sid}') : \tau_{op}\text{<sid}'\text{>}\} \vee (\mathbf{terminal}(\mathsf{sid}) : \epsilon) \vee \epsilon$$

τ_{new} requires all traces to begin with a creation operation. τ_{op}<sid> defines the allowed operations on a stream sid: we can either produce, with an intermediate operation, a stream sid′ starting from sid and then continue operating *only* on sid′, or consume sid and then we cannot do anything else on sid. In this way, we can only operate on the last stream of a processing pipeline, thus avoiding branching, and we can consume the stream only once. Actually, this trace expression is more in line with the specification than the real implementation; indeed, there are special kinds of operations, like `parallel` and `sequential`, classified as intermediate in the documentation but behaving differently. They are not lazy and do not return a new stream, but are used to switch the state of the whole pipeline they belong to; the only feature they share with the other intermediate operators is that they are not terminal. More importantly, in contradiction with the specification of intermediate operations, the implementation allows such switching operations to be invoked on a stream even after another intermediate operation has been already applied on it, as shown in the following code snippet.

```
IntStream s1 = IntStream.range(1, 10);
IntStream s2 = s1.filter(x -> x % 2 == 0);
s1.parallel();
s2.collect(HashSet::new, HashSet::add,
    HashSet::removeAll);
```

Operation `parallel` is invoked on `s1` immediately after the standard intermediate operation `filter`. No exception is thrown, while the trace expression τ_{new} rejects such a program, since after `s1.filter` no other operation can be invoked on `s1`. Invocation of parallel has the effect that the elements of the streams involved in the pipeline of `s1` will be processed in parallel. Hence, the combiner `HashSet::removeAll` passed as argument to the terminal operation `collect` is used to merge the partial result containers computed in parallel. Of course, the correct combiner `HashSet::addAll` should be used instead, but the aim of the example is showing the side effect of `parallel`; if we remove `s1.parallel` from the code, then the correct result `[2, 4, 6, 8]` is computed, instead of `[]`, because streams are sequential by default, and, hence, the combiner is unused in this case.

Furthermore, the implementation allows multiple invocations of switching operations on the same pipeline, even though only the last one is considered, thus favoring error-prone practices. To avoid this, we can enforce best practices with a trace expression τ'_{new} allowing no more than one switching operation per pipeline. To this aim, we add the event type **switch**(sid) matching invocations of switching operations (the definition of τ_{op}<sid> is the same as above):

$$\tau'_{new} = \epsilon \vee (\{\mathsf{sid}; \mathbf{new}(\mathsf{sid}) : \tau'_{op}\text{<sid>}\} \mid \tau'_{new})$$
$$\tau'_{op}\text{<sid>} = \{\mathsf{sid}'; \mathbf{intermediate}(\mathsf{sid}, \mathsf{sid}') : \tau'_{op}\text{<sid}'\text{>}\} \vee (\mathbf{switch}(\mathsf{sid}) : \tau_{op}\text{<sid>}) \vee (\mathbf{terminal}(\mathsf{sid}) : \epsilon) \vee \epsilon$$

Conclusion

The documentation of Java APIs often contains scattered rules for specifying complex patterns for legal method invocation flows which are difficult to follow, and sometimes are not implemented coherently. We have shown on the specific example provided by the package `java.util.stream`, how such patterns can be globally defined by compact and formal specifications, based on trace expressions, that could be employed for annotating Java interfaces. We plan to exploit such a formalism for dynamic verification of correct method invocation flows for Java API interfaces.

References

[1] Davide Ancona, Angelo Ferrando, and Viviana Mascardi. 2016. Comparing Trace Expressions and Linear Temporal Logic for Runtime Verification. In *Theory and Practice of Formal Methods: Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*, Erika Ábrahám, Marcello Bonsangue, and Einar Broch Johnsen (Eds.). Springer International Publishing, Cham, 47–64. https://doi.org/10.1007/978-3-319-30734-3_6

[2] Davide Ancona, Angelo Ferrando, and Viviana Mascardi. 2017. Parametric Runtime Verification of Multiagent Systems. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems (AAMAS '17)*. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 1457–1459. http://dl.acm.org/citation.cfm?id=3091282.3091328

[3] Simon J. Gay, Nils Gesbert, António Ravara, and Vasco Thudichum Vasconcelos. 2015. Modular Session Types for Objects. *Logical Methods in Computer Science* 11, 4 (2015). https://doi.org/10.2168/LMCS-11(4:12)2015

[4] Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. 2018. Typechecking protocols with Mungo and StMungo: A session type toolchain for Java. *Sci. Comput. Program.* 155 (2018), 52–75. https://doi.org/10.1016/j.scico.2017.10.006

[5] Martin Leucker and Christian Schallhart. 2009. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming* 78, 5 (2009), 293 – 303. https://doi.org/10.1016/j.jlap.2008.08.004 The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07).

[6] Cláudio Vasconcelos and António Ravara. 2017. From object-oriented code with assertions to behavioural types. In *Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3-7, 2017*, Ahmed Seffah, Birgit Penzenstadler, Carina Alves, and Xin Peng (Eds.). ACM, 1492–1497. https://doi.org/10.1145/3019612.3019733