SAT-Based Extraction of Behavioural Models for Java Libraries with Collections

Larisa Safina, Simon Bliudze

SAT-Based Extraction of Behavioural Models for Java Libraries with Collections

Larisa Safina
larisa.safina@inria.fr
INRIA
France

Simon Bliudze
simon.bliudze@inria.fr
INRIA
France

ABSTRACT

Behavioural models are a valuable tool for software verification, testing, monitoring, publishing etc. However, they are rarely provided by the software developers and have to be extracted either from the source or from the compiled code. In the context of Java programs, a number of approaches exist for building behavioural models. Most of these approaches rely on the analysis of the compiled bytecode. Instead, we are looking to extract behavioural models—in the form of Finite State Machines (FSMs)—from the Java source code to ensure that the obtained FSMs can be easily understood by the software developers and, if necessary, updated or integrated into the original source code, e.g. in the form of annotations. Modern software systems are huge, rely on external libraries and interact with their environment. Hence, extracting useful behavioural models requires abstraction. In this paper, we present an initial approach to this problem by focusing on the extraction of FSMs modelling library APIs. We focus on the analysis of Java code involving the use of collections. To this end, we encode the operational semantics of collection operations using patterns of Boolean predicates. These patterns are instantiated based on the analysis of the source code of API implementation methods to form an encoding of the possible FSM transitions. A SAT solver is then used to determine the enabledness conditions (guards) of these transitions.

KEYWORDS

API, breaking changes, behavioral models, FSM, Java

ACM Reference Format:
Larisa Safina and Simon Bliudze. 2022. SAT-Based Extraction of Behavioural Models for Java Libraries with Collections. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Behavioral models are a valuable tool for supporting the process of building the correct software, that could be beneficial during all phases of the development lifecycle: design and implementation [8, 24, 28], testing and verification [5, 12], coordination and monitoring [18, 23], deployment [7] etc.

One of the particular problem behavioral models could help to address is maintaining the API correctness. APIs provide a great way of scoping, encapsulating and presenting library functions [6]. Reusing components with the help of APIs helps to reduce the cost of software systems development and to increase their quality [9, 10, 17]. However, API development is not free of problems. When designed badly, API offers inefficient or even incorrect code requiring writing additional code to deal with it, which leads directly to the increased development cost [21]. When documented badly and without clear explanation of the correct sequence of method usage, API can demonstrate exceptional behaviours [3]. External APIs, being a black box, can contain error-prone dependencies and make systems prone to breaking changes that have to be addressed in the client code. A new version of an external interface evolving independently can break backward compatibility and may cause runtime failures of client systems. According to studies, breaking changes are being present in the significant amount of all new releases (70% [26], 75% [15], and 80% [16]). One can argue that building a dynamic model of libraries to help the developer understand the behaviour of methods operating on collections contradicts the fundamental principle of information hiding: a developer relying on knowledge of the API implementation would produce code even more prone to the breaking changes we want to address. We believe, however, that if a signature of an API method provides a developer with the insufficient information, she will need to make her assumptions in any case whether they are supported by the extracted model or not. But comparing the two models of two different releases can show a developer if any breaking changes were introduced.

There exist various approaches based on behavioral models to deal with such problems possessing different levels of formality (from graphical representation [11] to employing behavioral types [2]) and ways of being produced and presented. In general, these models are expected to be expressive, rigourous, intuitive, well-structured and possess the adequate level of abstraction (be sufficiently detailed and precise) [20]. Such models could complement the code [27], be extracted from it [16, 25], from its binaries [26], from some intermediate representation or from other models [13, 14], from the execution traces [22] or be created to synthesize the code from them [20].

We are interested in building behavioral models that have a tight connection with API implementation source code to provide feedback to developers on the behavior of the program. In this paper, we propose an approach to semi-automatic extraction of behavioral models for APIs written in Java in the form of Finite State Machines (FSMs), focusing on the code operating on collections.

The paper is structured as follows. In Section 2, we introduce a simple running example. In Section 3, we present our approach

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
  class ExampleImpl implements ExampleAPI {
    private Set<String> idSet;

    public ExampleImpl() {
1:    idSet = new HashSet<>();
    }

    public void add(String id){
1:      idSet.add(id);       // See Section 3.2 for the line
    }                        // numbers in the left margin

    public void removeId(String idMain, String idOpt) {
1:    idSet.remove(idMain);
      if (idMain != idOpt) {
2:      idSet.remove(idOpt);
      } else {
3:    }
    }
  }
```

Figure 1: Running example source code

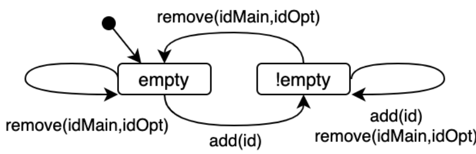

Figure 2: Behavioural model for the `HashSet` implementation

to model extraction. In Section 4, we discuss our assumptions and future work directions. Section 5 concludes the paper.

2 RUNNING EXAMPLE

Let's imagine the API created for the users of a pharmaceutical company. In particular it allows users to create reservations for the set of medications and to manipulate this collections: add or remove elements. Each medication can have two identifiers where the second is the one provided by a third-party vendor. When a detail to be added or removed both ids are checked (see the code snippet bellow)

During the last code review it was decided to migrate idSet collection from the `HashSet` to the `TreeSet` to present the order of the elements. However, this led to the unexpected exception, when some user tried to remove a medicine having only one id (`remove("LK32EJ2", null)`). This exception is not visible at the compilation but can be catched by the static code analysers, and it is clearly visible if we compare the behavioral models generated for the two versions of the code (see Figures 2 and 3).

These models show the result of calling the API methods add and remove. We model the states with the predicates we used to create this abstraction. The transitions are labeled with the API method names, some of them have the guards (values in square brackets) which are the conditions based on the methods variables

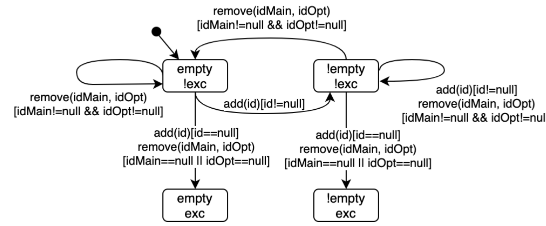

Figure 3: Behavioural model for the `TreeSet` implementation

evaluated dynamically, that have to be satisfied in order to fire the transition.

3 MODEL EXTRACTION APPROACH

3.1 Simplifying assumptions

We based our work on Java Collection Framework JDK 17—the most recent version at the time of writing [1]—and considered its General Purpose Implementations (GPI). We studied each GPI to capture its behaviour and to define its operational semantics taking into account differences in the implementation of methods for various interfaces. For the sake of brevity, we discuss only `Set` collections.

We focus on the contribution of collections to the object state. Thus, the defining property of sets, i.e. the absence of duplicates, is irrelevant. Other properties that can be taken into account are (1) whether the collection is ordered and (2) whether it permits null values. Here, we focus on the latter.

Furthermore, since the operation of enlarging the collection is called implicitly, we disregard the initial capacity and the load factor of the collection. Similarly, since element retrieval is not based on their position in the collection, we do not have to keep track of the insertion order. In particular, these assumptions allow us to process `HashSet` and `LinkedHashSet` in the same manner. For `TreeSet`, we have extra rules for adding and removing `null` elements.

In addition, we make the following assumptions that simplify the presentation. We summarise them here—further details are provided in Section 3.2. We will discuss the validity of these assumptions in Section 4. We assume that:

(1) all conditions used in branching statements (`if`, `while` etc.) are Boolean expressions referring exclusively to comparisons of values and the `contains()` and `isEmpty()` operations on collections
(2) the source code is formatted in such a manner that there is at most one operator per line, all branching statements have all their openning and closing braces and the closing braces are placed on separate lines from any other operators (cf. the example in Figure 1)
(3) there are only two types—collections and values—and any value can be stored in any collection
(4) all loops terminate.

We plan to gradually resolve these simplifications up to the limits modern model checking and other static analysis techniques will permit.

3.2 Boolean encoding of method behaviour

Our approach relies on the predicate abstraction idea from [19]. For each method in the API under consideration, we proceed as follows.

Given a method, we consider a *context* comprising a set of collection symbols C and a set of value symbols V. The symbols in V can be literal constants (e.g. "`Hello!`", `null`), variables, method parameters etc. For two symbols $v_1, v_2 \in V$ (idem for $c_1, c_2 \in C$), we write $v_1 \equiv v_2$ if they are the same symbol; $v_1 = v_2$ denotes the predicate stating that their values are equal.

The definition of the context is a parameter of the approach, determining, in particular, the degree of refinement of the resulting model. Context values are used to form the following predicates:

- $eq_{v_1,v_2} \stackrel{def}{=} (v_1 = v_2)$, for $v_1, v_2 \in V$ ($v_1 \not\equiv v_2$)
- $eq_{c_1,c_2} \stackrel{def}{=} (c_1 = c_2)$, for $c_1, c_2 \in C$ ($c_1 \not\equiv c_2$)
- $cnt_{c,v}$ holds for $v \in V$ and $c \in C$ if v is contained in c
- $empty_c$ holds for $c \in C$ if c is empty
- e holds if an exception has been thrown

We postulate the following axioms,

(1) $ax_1(c, v_1, v_2) \stackrel{def}{=} (eq_{v_1,v_2} \implies cnt_{c,v_1} = cnt_{c,v_2})$
(2) $ax_2(c_1, c_2, v) \stackrel{def}{=} (eq_{c_1,c_2} \implies cnt_{c_1,v} = cnt_{c_2,v})$
(3) $ax_3(c, v) \stackrel{def}{=} (empty_c \implies \neg cnt_{c,v})$

Let P be the set of all the predicates above and $P' \stackrel{def}{=} \{p' \mid p \in P\}$. Assume that all operators of the method are numbered sequentially from 1 to n, with the numbering of branching operators corresponding to their closing braces (cf. Figure 1). For each predicate $p \in P$ and each $i \in [0, n]$, we define a Boolean variable p^i corresponding to the valuation of p after the execution of the operator i.

For a formula $\varphi \in \mathbb{B}[P, P']$, denote $\varphi^{i,j} \stackrel{def}{=} \varphi\left[P^i/P\right]\left[P^j/P'\right]$ the formula where each predicate p (resp. p') is substituted by the variable p^i (resp. p^j). Similarly, $\varphi^i \stackrel{def}{=} \varphi[P^i/P]$, for $\varphi \in \mathbb{B}[P]$.

For a sequence of operators `s` and two indices $i, j \in [0, n]$ ($i < j$), we define the encoding function $enc^{i,j}(\texttt{s})$ as shown in Figure 4, where $\texttt{s}\!\downarrow$ denotes *the index of the last operator* in a sequence `s` and, in (4), φ_{op} is the formula encoding the semantics of the operator `op` (see Section 3.3 below). A method `meth` with the body `s` is then encoded by the formula

$$enc(\texttt{meth}) \stackrel{def}{=} axioms^0 \wedge enc^{0,\texttt{s}\downarrow}(\texttt{s}),$$

with the formula *axioms* defined by (5).

$$enc^{i,j}(\texttt{op; s}) \stackrel{def}{=} enc^{i,\texttt{op}\downarrow}(\texttt{op}) \wedge enc^{\texttt{op}\downarrow,j}(\texttt{s}) \tag{1}$$

$$enc^{i,j}(\texttt{while (cnd) s}) \stackrel{def}{=} \neg cnd^j \tag{2}$$

$$enc^{i,j}(\texttt{if (cnd) s}_1\ \texttt{else s}_2) \stackrel{def}{=} enc^{i,\texttt{s}_1\downarrow}(\texttt{s}_1) \wedge enc^{i,\texttt{s}_2\downarrow}(\texttt{s}_2) \wedge \Big(cnd^i \implies \bigwedge_{p\in P} p^j = p^{\texttt{s}_1\downarrow}\Big) \wedge \Big(\neg cnd^i \implies \bigwedge_{p\in P} p^j = p^{\texttt{s}_2\downarrow}\Big) \tag{3}$$

$$enc^{i,j}(\texttt{op}) \stackrel{def}{=} \varphi^{i,j}_{\mathsf{op}} \wedge axioms^j \tag{4}$$

$$axioms \stackrel{def}{=} \bigwedge_{\substack{c\in C\\ v_1,v_2\in V\\ v_1\not\equiv v_2}} ax_1(c, v_1, v_2) \wedge \bigwedge_{\substack{c_1,c_2\in C\\ v\in V\\ c_1\not\equiv c_2}} ax_2(c_1, c_2, v) \wedge \bigwedge_{\substack{c\in C\\ v\in V}} ax_3(c, v) \tag{5}$$

Figure 4: Definition of the encoding function $enc^{\cdot,\cdot}(\cdot)$

For example, consider the method `c.add(v)` of a `HashSet c`. Its predicate semantics is given by the formula

$$f_{\texttt{c.add(v)}} \stackrel{def}{=} cnt'_{c,v} \wedge \neg empty'_c\,.$$

Here, $C_{\texttt{c.add(v)}} = \{c\}$, $V_{\texttt{c.add(v)}} = \{v\}$, $P_{\texttt{c.add(v)}} = \{cnt_{c,v}, empty_c\}$.

Let us consider, as another example, the method `c.clear()` of a collection `c`. Its predicate semantics is given by the formula

$$f_{\texttt{c.clear()}} \stackrel{def}{=} empty'_c\,.$$

Despite v not appearing in $f_{\texttt{c.clear()}}$, we take the same $C_{\texttt{c.clear()}} = \{c\}$, $V_{\texttt{c.clear()}} = \{v\}$ and $P_{\texttt{c.clear()}} = \{cnt_{c,v}, empty_c\}$ as above. Indeed, the value of the $cnt_{c,v}$ predicate, *for all value symbols in the context*, may be modified by this method, even though ax_3 means that this does not have to be specified explicitly in $f_{\texttt{c.clear()}}$.

For each operator invocation in the source code, some of the variables in C_{op} and V_{op} will be positionally matched with the symbols in the context (C, V). E.g., for an invocation `idSet.clear()`, $c \in C_{\texttt{c.clear()}}$ is matched to $idSet \in C$ but $v \in V_{\texttt{c.clear()}}$ is not matched to any symbol in V. For the sake of clarity, let us abuse the notation as follows: we denote $C_{\mathsf{op}} \setminus C$ (resp. $V_{\mathsf{op}} \setminus V$) the set of variables that are not matched positionally.

The formula φ_{op} encoding the operator `op` (see (4) in Figure 4) is then defined as follows:

$$\varphi_{\mathsf{op}} \stackrel{def}{=} \bigwedge_{\substack{\mu: C_{\mathsf{op}}\setminus C\to C\\ \nu: V_{\mathsf{op}}\setminus V\to V}} f_{\mathsf{op}}[\mu, \nu] \wedge \bigwedge_{p\in P_{free}} p' = p \tag{6}$$

$$\text{with} \qquad P_{free} \stackrel{def}{=} P \setminus \bigcup_{\substack{\mu: C_{\mathsf{op}}\setminus C\to C\\ \nu: V_{\mathsf{op}}\setminus V\to V}} P_{\mathsf{op}}[\mu, \nu]\,,$$

where $f_{\mathsf{op}}[\mu, \nu]$ denotes the formula obtained by substituting each unmatched variable $c \in C_{\mathsf{op}} \setminus C$ (resp. $v \in V_{\mathsf{op}} \setminus V$) by the symbol $\mu(c)$ (resp. $\nu(v)$) and similarly for $P_{\mathsf{op}}[\mu, \nu]$.

Intuitively, the first conjunct in (6) means that the formula f_{op} is instantiated for all possible values of the unmatched variables. The second conjunct means that all predicates in P that do not appear in the semantics of `op` maintain their previous values.

3.3 Predicate semantics of operators

The predicate semantics of an operator `op` is given by a quadruple $(C_{\mathsf{op}}, V_{\mathsf{op}}, P_{\mathsf{op}}, f_{\mathsf{op}})$, where

- $(C_{\mathsf{op}}, V_{\mathsf{op}})$ is a context, comprising the sets C_{op} and V_{op} of, respectively, collection and value variables
- P_{op} is a subset of predicates as in Section 3.2 indexed by the variables from C_{op} and V_{op}
- $f_{\mathsf{op}} \in \mathbb{B}[P_{\mathsf{op}}, P'_{\mathsf{op}}]$ is a Boolean formula on these predicates

Intuitively, P_{op} is the set of predicates whereof the valuations *may* be affected by the operator.

Notice, finally, that a predicate p', with $p \in P_{\mathrm{op}}$, that is not fully constrained by f_{op} might take any value, representing the absence of the corresponding information. For instance, consider the method `c.remove(v)` of a `HashSet` `c`. Its predicate semantics is given by the formula

$$f_{\texttt{c.remove(v)}} \stackrel{def}{=} \neg cnt'_{c,v} \wedge (empty_c \implies empty'_c),$$

with, again, $C_{\texttt{c.remove(v)}} = \{c\}$, $V_{\texttt{c.remove(v)}} = \{v\}$ and $P_{\texttt{c.remove(v)}} = \{cnt_{c,v}, empty_c\}$. If `c` is not empty before the call to `c.remove(v)` it may or may not become empty after the call, depending on whether `v` is the only value contained therein.

3.4 Computing the FSM

Given a context (C, V) with the corresponding set of predicates P as defined in Section 3.2, we observe that the symbols can be partitioned into *state* and *indeterminacy* symbols: $C = C_{st} \uplus C_{nd}$ and $V = V_{st} \uplus V_{nd}$, such that C_{st} and V_{st} contain constants and class fields, whereas C_{nd} and V_{nd} contain method parameters, unevaluated functions etc. This induces several partitions on the set of predicates: $P = P_{st} \uplus P_{nd}$, such that P_{st} is a subset of predicates that refer only to symbols in C_{st} and V_{st}.

A *state* is a partial valuation $\sigma : P_{st} \rightharpoonup \mathbb{B}$. A *concrete state* is a total valuation $\sigma : P_{st} \to \mathbb{B}$. We say that one state σ_1 *refines* another σ_2, denoted $\sigma_1 \prec \sigma_2$ if $\mathrm{dom}(\sigma_2) \subseteq \mathrm{dom}(\sigma_1)$ and $\sigma_1(p) = \sigma_2(p)$, for all $p \in \mathrm{dom}(\sigma_2)$. We say that a set of states Σ is *complete* if every state refines some state of Σ, i.e. for every $\sigma : P_{st} \rightharpoonup \mathbb{B}$, there exists $\hat{\sigma} \in \Sigma$, such that $\sigma \prec \hat{\sigma}$. Notice that any complete set of states must comprise the empty valuation σ_0, having $\mathrm{dom}(\sigma_0) = \emptyset$. Clearly, for any given implementation of an API, there can be different complete sets of states. In particular, the set $\{\sigma_0\}$ is complete. So is the set of all concrete states $\{\sigma \,|\, \mathrm{dom}(\sigma) = P_{st}\} \cup \{\sigma_0\}$.

Given a context (C, V), a set of predicates P and a complete set of states Σ, we build a non-deterministic FSM $(\Sigma, \sigma_0, A, \to)$, where σ_0 is the initial state, A is the set of methods provided by the API under consideration and $\to \subseteq \Sigma \times A \times \mathbb{B}[P_{nd}] \times \Sigma$ is the transition relation, where the third component is the transition *guard*.

To compute the transition relation $\to$, we proceed as follows. For each method `meth` $\in A$ and each pair of states $\sigma_1, \sigma_2 \in \Sigma$, we define

$$\varphi_0 \stackrel{def}{=} \bigwedge_{p \in \mathrm{dom}\,\sigma_1} \left(p^0 = \sigma_1(p)\right) \wedge enc(\texttt{meth}) \wedge \bigwedge_{p \in \mathrm{dom}\,\sigma_2} \left(p^{\texttt{meth}\downarrow} = \sigma_2(p)\right)$$

(see Section 3.2) and submit it to a SAT solver. If φ_0 is satisfiable we obtain a model $m_1 : \{p^i \,|\, p \in P, i \in [0, \texttt{meth}\downarrow]\} \to \mathbb{B}$. We define

$$\varphi_1 \stackrel{def}{=} \varphi_0 \wedge \neg \bigwedge_{i=0}^{\texttt{meth}\downarrow} \bigwedge_{p \in P} \left(p^i = m_1(p^i)\right)$$

and submit it to the SAT solver again, repeating until we obtain some φ_n that is unsatisfiable. The transition is then

$$\sigma_1 \xrightarrow{\texttt{meth}[g]} \sigma_2, \qquad \text{with } g = \bigvee_{i=1}^{n} \bigwedge_{p \in P_{nd}} \left(p = m_i(p^0)\right).$$

Notice that, if φ_0 is unsatisfiable, we have $g = false$.

Applying this approach to the example in Figure 1 we obtain the FSMs in Figures 2 and 3. We do so using the set of all concrete states, in which each state is a combination of predicates from P (predicates forming the states could be selected to tune the level of the model's abstraction, e.g in our example we rely on the $empty_c$ predicate even though P is not limited by it. The common context for the `HashSet` version of the example is $(\{idSet\}, \emptyset)$. For methods `add` and `removeId` we take the union with, respectively, $(\emptyset, \{id\})$ and $(\emptyset, \{idMain, idOpt\})$. The predicates are the same as in Section 3.2, with $P_{st} = \{empty_{idSet}\}$ (we exclude e, since it appears neither in the axioms nor in the semantics of the operators involved).

For the `TreeSet` version, we add *null* to the common context and e to P_{st} since they appear in the semantics of the operators, e.g.

$$\begin{aligned} f_{\texttt{c.add(v)}} &\stackrel{def}{=} (eq_{v,null} = eq'_{v,null}) \\ &\wedge \neg eq_{v,null} \implies (cnt'_{c,v} \wedge \neg empty'_c \wedge (e' = e)) \\ &\wedge eq_{v,null} \implies (cnt_{c,v} = cnt'_{c,v} \wedge empty_c = empty'_c \wedge e') \end{aligned}$$

$$\begin{aligned} f_{\texttt{c.remove(v)}} &\stackrel{def}{=} (eq_{v,null} = eq'_{v,null}) \\ &\wedge \neg eq_{v,null} \implies (\neg cnt'_{c,v} \wedge (empty_c \implies empty'_c) \wedge (e' = e)) \\ &\wedge eq_{v,null} \implies (cnt_{c,v} = cnt'_{c,v} \wedge empty_c = empty'_c \wedge e') \end{aligned}$$

The full Z3 encodings of the two versions can be found in [4].

4 DISCUSSION & FUTURE WORK

The model extraction approach proposed above is based on ideas that are well-known in the Formal Methods community. The novelty lies in their application for the extraction of behavioural models from Java source code. More importantly, our main goal is not to use these models for verification but rather for assisting software engineers in the development process and, eventually, for model-based monitoring and coordination. This allows us, in particular, to aim for a less restrictive behavioural model, namely non-deterministic FSMs with guards. This change of perspective also makes the semi-automatic approach more acceptable: indeed, we believe that it is much easier to get developers to help refining an existing model than to provide one from scratch.

The semi-automatic nature of our approach arises from several levels of parametrisation among which the key one is choosing the appropriate predicates for the specification of the semantics of the basic operations. To some extent, this remark addresses the simplifying assumption (1) in Section 3.1, which can be substantially relaxed by including additional predicates to define the semantics of the terms used in the conditions. Further relaxation can, of course, be achieved by moving from SAT to SMT.

Deciding which symbols should be part of the context and which predicates should be used for the encoding is a non-trivial task, which we believe cannot be fully formalised. We see two approaches to address this question: input from domain experts, i.e. developers, and various kinds of heuristics (potentially including Machine Learning techniques). Both approaches—or combinations whereof—provides directions for future work.

Assumptions (2) and (3) are obviously non-constraining. Moreover, in any actual implementation, we would use Abstract Syntax Trees and Control Flow Graphs to explore the source code and compute the Boolean encodings. This would naturally eliminate the need for these two assumptions.

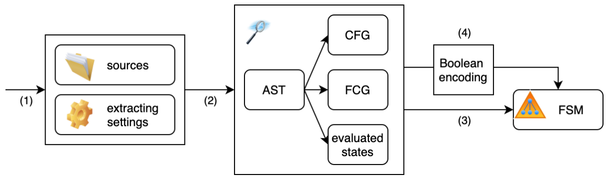

Figure 5: Intended tool architecture

Thus, the strongest assumption that we have made is the loop termination assumption (4), which would require deeper static analysis or assistance from the developers for exhibiting loop variants, which we consider beyond the near-future scope of this work.

We intend to develop a tool for extracting FSMs from Java source code based on the approach proposed in this paper. In general, the tool will proceed along the following steps (see Figure 5): collecting the information about the sources under analysis and about the fields identified as key to the object state (1); computing auxiliary data structures, and state values that can be easily evaluated (2); then using different techniques to generate the FSM. The last step differs for the basic types and collections. For instance, for simple types, such as enumerations, simple interpretative techniques could suffice (3). For complex types such as collections, we will rely on the SAT-based approach presented in this paper (4).

Finally, another point worth mentioning is the choice of the complete set of states. For the sake of simplicity, we have chosen to use the set of all concrete states. This can clearly be improved both a posteriori—e.g. reducing the FSM by bisimilarity—and a priori—e.g. building the state space on the fly to avoid unreachable states.

5 CONCLUSION

We propose an approach to the extraction of behavioural models in the form of non-deterministic FSMs with guards from Java implementations of library APIs. The approach is semi-automatic since it is parametrised by the choice of important symbols, predicates and states. This makes it highly flexible and allows the integration of expert knowledge from developers. The presented approach is fully formalised. Although the underlying ideas are well known in the Formal Methods community, we believe that their combined application for the extraction of FSMs with guards is novel and can be applied to the benefit of the Software Engineering community. We are planning to develop a tool based on the proposed approach and use it in our future projects for the purposes of monitoring and coordination of software components.

REFERENCES

[1] 2021. Java Collections Framework Overview. https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/doc-files/coll-overview.html. [Online; accessed 6-Jan-2022].

[2] Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. 2016. Behavioral Types in Programming Languages. *Found. Trends Program. Lang.* 3, 2-3 (2016), 95–230. https://doi.org/10.1561/2500000031

[3] Davide Ancona, Francesco Dagnino, and Luca Franceschini. 2018. A Formalism for Specification of Java API Interfaces. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops* (Amsterdam, Netherlands) *(ISSTA '18)*. Association for Computing Machinery, New York, NY, USA, 24–26. https://doi.org/10.1145/3236454.3236476

[4] Anoymous. 2022. Paper's example z3-encoding. https://doi.org/10.5281/zenodo.5913271

[5] Christian Bartolo Burlò, Adrian Francalanza, and Alceste Scalas. 2020. *Towards a Hybrid Verification Methodology for Communication Protocols (Short Paper)*. 227–235. https://doi.org/10.1007/978-3-030-50086-3_13

[6] Joshua Bloch. 2006. How to Design a Good API and Why It Matters. In *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 506–507. https://doi.org/10.1145/1176617.1176622

[7] Mario Bravetti, Saverio Giallorenzo, Jacopo Mauro, Iacopo Talevi, and Gianluigi Zavattaro. 2020. A Formal Approach to Microservice Architecture Deployment. In *Microservices, Science and Engineering*.

[8] Mario Bravetti and Gianluigi Zavattaro. 2019. *Relating Session Types and Behavioural Contracts: The Asynchronous Case*. 29–47. https://doi.org/10.1007/978-3-030-30446-1_2

[9] Aline Brito, Laerte Xavier, Andre Hora, and Marco Valente. 2018. Why and how Java developers break APIs. 255–265. https://doi.org/10.1109/SANER.2018.8330214

[10] Gleison Brito, Andre Hora, Marco Valente, and Romain Robbes. 2017. On the Use of Replacement Messages in API Deprecation: An Empirical Study. *Journal of Systems and Software* 137 (12 2017). https://doi.org/10.1016/j.jss.2017.12.007

[11] Steve Cook, Conrad Bock, Pete Rivett, Tom Rutt, Ed Seidewitz, Bran Selic, and Doug Tolbert. 2017. *Unified Modeling Language (UML) Version 2.5.1*. Standard. Object Management Group (OMG). https://www.omg.org/spec/UML/2.5.1

[12] James C. Corbett. 2000. Using Shape Analysis to Reduce Finite-State Models of Concurrent Java Programs. *ACM Trans. Softw. Eng. Methodol.* 9, 1 (Jan. 2000), 51–93. https://doi.org/10.1145/332740.332741

[13] Luís Cruz-Filipe, Kim S. Larsen, and Fabrizio Montesi. 2017. The Paths to Choreography Extraction. In *Foundations of Software Science and Computation Structures*, Javier Esparza and Andrzej S. Murawski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 424–440.

[14] Ornela Dardha, Simon J Gay, Dimitrios Kouzapas, Roly Perera, A Laura Voinea, and Florian Weber. 2017. Mungo and StMungo: Tools for Typechecking Protocols in Java. *Behavioural Types: from Theory to Tools* (2017), 309.

[15] Jens Dietrich, Kamil Jezek, and Premek Brada. 2014. Broken promises: An empirical study into evolution problems in Java programs caused by library upgrades. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. 64–73. https://doi.org/10.1109/CSMR-WCRE.2014.6747226

[16] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. 2006. Automated Detection of Refactorings in Evolving Components. https://doi.org/10.1007/11785477_24

[17] Tiago Espinha, Andy Zaidman, and Hans-Gerhard Groß. 2015. Web API growing pains: Loosely coupled yet strongly tied. *J. Syst. Softw.* 100 (2015), 27–43.

[18] Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. 2019. Guess Who's Coming: Runtime Inclusion of Participants in Choreographies. In *The Art of Modelling Computational Systems: A Journey from Logic and Concurrency to Security and Privacy - Essays Dedicated to Catuscia Palamidessi on the Occasion of Her 60th Birthday (Lecture Notes in Computer Science, Vol. 11760)*, Mário S. Alvim, Kostas Chatzikokolakis, Carlos Olarte, and Frank Valencia (Eds.). Springer, 118–138. https://doi.org/10.1007/978-3-030-31175-9_8

[19] Susanne Graf and Hassen Saïdi. 1997. Construction of Abstract State Graphs with PVS. In *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings (Lecture Notes in Computer Science, Vol. 1254)*, Orna Grumberg (Ed.). Springer, 72–83. https://doi.org/10.1007/3-540-63166-6_10

[20] D. Harel and E. Gery. 1997. Executable object modeling with statecharts. *Computer* 30, 7 (1997), 31–42. https://doi.org/10.1109/2.596624

[21] Michi Henning. 2007. API: Design Matters: Why Changing APIs Might Become a Criminal Offense. *Queue* 5, 4 (may 2007), 24–36. https://doi.org/10.1145/1255421.1255422

[22] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. 2008. Automatic generation of software behavioral models. *Proceedings - International Conference on Software Engineering*, 501–510. https://doi.org/10.1145/1368088.1368157

[23] Anastasia Mavridou, Valentin Rutz, and Simon Bliudze. 2017. Coordination of Dynamic Software Components with JavaBIP. https://doi.org/10.1007/978-3-319-68034-7_3

[24] Fabrizio Montesi. 2015. Kickstarting Choreographic Programming. https://doi.org/10.1007/978-3-319-33612-1_1

[25] Pavel Parízek and Ondřej Lhoták. 2012. Predicate Abstraction of Java Programs with Collections. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Tucson, Arizona, USA) *(OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 75–94. https://doi.org/10.1145/2384616.2384623

[26] S. Raemaekers, A. van Deursen, and J. Visser. 2017. Semantic versioning and impact of breaking changes in the Maven repository. *Journal of Systems and Software* 129 (2017), 140–158. https://doi.org/10.1016/j.jss.2016.04.008

[27] James Rumbaugh, Ivar Jacobson, and Grady Booch. 2004. *Unified Modeling Language Reference Manual, The (2nd Edition).* Pearson Higher Education.

[28] A. Voinea, Ornela Dardha, and Simon Gay. 2020. *Typechecking Java Protocols with [St]Mungo.* 208–224. https://doi.org/10.1007/978-3-030-50086-3_12