Contract Automata Library

Davide Basile*, Maurice H. ter Beek*

Formal Methods and Tools lab
ISTI–CNR, Pisa, Italy

Abstract

Contract automata facilitate the specification, composition, and synthesis of behavioural contracts, comprehending modalities and configurations. Contract automata are supported by a software API called Contract Automata Library. This paper accompanies the software artefact by discussing its architecture, showing some usage examples and presenting recent improvements of the software in terms of quality, availability, usability, and documentation.

Keywords: Service Composition, Controller Synthesis, Behavioural Contracts, Software Quality

1. Introduction

Behavioural contracts [1] have been introduced to specify the behaviour of services and to synthesise service compositions satisfying given formal properties [2–4]. This allows rigorous reasoning on the composite behaviour, as well as improving the modularity, adaptability, and reusability of services.

Contract automata [5] are a dialect of Finite State Automata introduced to model behavioural service contracts in terms of service offer actions and service request actions, which need to match to achieve agreement among a composition of contracts. Modalities are used to indicate when an action must be matched (necessary) and when it can be be withdrawn (permitted) in the synthesised composition [6]. Composing contracts and synthesising a well-behaving composition, usually by refining a spurious composition, are two of the main functionalities supported by contract automata. The synthesis for contract automata builds upon results from supervisory control theory [7–9] for synthesising the most permissive controller (*mpc* for short), duly revisited for synthesising orchestrations and choreographies in [10]. Contract automata and their functionalities are implemented in a software artefact, called Contract Automata Library (`CATLib`), which is under continuous development.

*First author, corresponding author
Email address: `davide.basile@isti.cnr.it` (Davide Basile)

This paper accompanies the software artefact and it presents recent improvements in the (open-source) software supporting the contract automata formalism [11], following the guidelines of *Science of Computer Programming*'s new Software Track on Original Software Publications [12] (OSP). The software's purpose is to show the feasibility of the proposed theoretical approach, to aid researchers experimenting with new developments in the theoretical framework of contract automata, and to build applications and formal tools exploiting this framework. A developer can exploit `CATLib` to instantiate contract automata and perform operations like composition and synthesis. An application developed with `CATLib` is thus formally validated *by-construction* against well-behaving properties from the theory of contract automata [5, 6, 10, 13]. An example of an application developed using the library is provided in this paper as well as details of the software's architecture. We also discuss the software's evaluation in terms of quality and correctness. Indeed, one of the goals of the research on behavioural contracts is to develop more reliable software applications. To do so, reliable implementations of these formalisms must also be made available. We tackle this aspect by detailing the techniques and external services that have been used to assess the quality and correctness of the implementation.

The software is currently used by the inventors and contributors of the contract automata formalism. The public repository hosting the software has been forked by other developers outside the organisation. The software is not intended for commercial use. Some publications that have exploited the software (including its earlier prototypes) are [5, 6, 10, 13–20]. The presented software artefact is original and has not been published elsewhere.

Related Work. This paper accompanies the software artefact `CATLib`, whose refactoring and redesign is discussed in [20]. `CATLib` was redesigned according to principles of model-based systems engineering [21, 22] and of writing clean and readable code [23, 24], which are known to improve reliability and understandability, and to facilitate maintainability and reuse. The software has moreover been refactored using lambda expressions and Java Streams as available in Java 8 [25, 26], exploiting parallelism. Experiments were conducted to show the performance improvement (cf. [20, Table 1]). There are other repositories currently under development that exploit `CATLib`. `CAT_App` [27] is a GUI front-end of `CATLib` to visualise contract automata, edit them, and use main operations (e.g., composition and synthesis). `CAT_App` was developed by specialising the `BasicGraphEditor` of the `mxGraph` library [28]. `CARE` [29] is a runtime environment to coordinate services implementing contract automata specifications.

Composition and orchestration of services via supervisory control theory is an active research field [30–34], for which decidability has been studied earlier [35]. Most of this body of work investigates fundamental aspects, without providing implementations. `CATLib` (cf. Section 2) could be extended to also support some of these proposals.

In the repository [36] of software accepted for the OSP track of *Science of Computer Programming*, two of them concern choreographies of services [37, 38], which share aspects with the choreography synthesis implemented in `CATLib`.

We note that no information about testing or code analysis is available in most of the repositories of software artefacts in [36] (31 entries at the time of writing). Instead, `CATLib` follows a thorough process of automatic building, testing, and analysis, based on [39], which is described in Section 5, providing more confidence on the correctness and quality of our implementation, which is easily accessible and verifiable through the public repository (cf. Figure 3).

Outline. This paper is organised as follows. We start by briefly describing the contribution and innovation in Section 2. The software design and background is presented in Section 3. Examples of usage of the library are described in Section 4. The criteria for the software evaluation are discussed in Section 5. Finally, Section 6 concludes the paper and discusses future work.

2. Contribution and Innovation

This software artefact is a by-product of scientific research on behavioural contracts and implements results that have been previously formally specified in several scientific publications (cf., e.g., [5, 6, 10, 13–20]).

From a recent survey in the transport domain [40] it has emerged that the majority of studies on formal methods propose specification languages, models, and their verification, whereas fewer focus on how to derive the finalised software from the verified specification. Indeed, as reported in [41], behavioural contracts/types are not yet a feature of mainstream programming languages. This software artefact allows programmers to use contract automata to develop more reliable applications. Section 4 shows a simple example of an application derived from contract automata models.

The library has been designed to be easily extendable to support similar automata-based formalisms. Currently, synchronous communicating machines [42] are also supported by the library. The relation and translation between these two formalisms was studied in [13].

This software is the first to implement innovative choreography and orchestration synthesis algorithms, different from the standard controller synthesis algorithm for supervisory control of discrete event systems [7–9]. The synthesis of a controller, an orchestration, and a choreography are all different special cases of a more abstract synthesis algorithm, formalised in [10] and implemented in `CATLib` [20] using big data-like parallel operations of Java Streams. The algorithms themselves exploit new notions of controllability [10] and compositionality [5] of behavioural contracts. The software uses established technologies for building, testing, documenting, and delivering high-quality source code.

3. Software Design

In this section, we describe the high-level structure of the library and we provide some background on the theory of contract automata [5, 6, 10]. Apart from this accompanying paper, the software artefact also includes another five

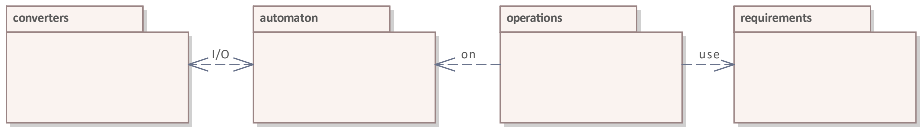

Figure 1: High level package diagram

different artefacts containing fine-grained details of the implementation: (i) a navigable interactive site visualising for all the packages their class diagram and the relative documentation of all members [43]; (ii) a diagram report containing all class diagrams of this library [44]; (iii) a library report describing all class diagrams with documentation for each class and each member of a class [45]; (iv) the JavaDoc site available online [46], reporting all class diagrams and the documentation for each class in the standard JavaDoc format (more on this in Section 5.3); (v) a package report describing all packages and classes [47]. All these technical documents, the `GitHub Page` of the library, and its current release are also permanently available at [48].

Figure 1 shows four of the eleven packages of `CATLib` and their relations. The **automaton** package and its sub-packages implement the formal model. Automata can be imported/exported using the package **converters**, operated upon using the package **operations** that in turn uses the package **requirements**. All packages are prefixed by **io.github.contractautomata.catlib**.

The implementation of an automaton is under the package **automaton** and its sub-packages contain the implementations of states, transitions, labels, and actions of an automaton. We use the typewriter font for names of classes and interfaces. The `Automaton` class implements the interface `Ranked`. The rank is the number of atomic components contained in the automaton. When several automata are composed (whose rank may be greater than one), the result is an automaton with a rank that is the sum of the rank of the operands.

`Label` is the super class of labels, having a content that is a tuple of a generic type. `CALabel` extends `Label` to implement labels of contract automata. Following [5, Definition 2.1], labels of contract automata are lists of actions of three types: (i) offer: one action is an offer action and all the others are idle actions, (ii) request: one action is a request action and all the others are idle actions, (iii) match: two actions are matching (i.e., one is a request, the other an offer, and their content is the same) and all the others are idle. `Action` is the super class from which the other actions are inheriting. In contract automata, an action can be either an `OfferAction`, a `RequestAction`, or an `IdleAction` (i.e., a 'nil' action). Actions are matchable and a request action matches an offer action (and vice versa) if both have the same label. Actions can have an `Address`, in this case implementing the interface `AddressedAction`. Actions with addresses are `AddressedOfferAction` and `AddressedRequestActions`. These actions are equipped with an address storing senders and receivers of actions. For two addressed actions to match, also their sender and receiver must be equal. Addressed actions are used to implement communicating machines, in which each participant in the composition is aware of the other participants. Communicating machines are a known model of choreographies [42]. Also actions not having an address may be used in contract automata: in this case the participants are oblivious of the other partners, and the model assumes the presence of an orchestrator in charge of pairing offers and requests [13].

The (abstract) super class of a state is `AbstractState`, where a state can be initial or final and has a label. A `BasicState` implements an `AbstractState` of a single participant, it has rank 1 and the label of the state cannot have further inner components. A `State` implements an `AbstractState` with a rank: it is a list of basic states. `Transition` is the super class for transitions, it has a source state, a target state, and a label. `ModalTransition` extends `Transition` to include modalities (cf. [6, Definition 9]). The modalities of contract automata are permitted and necessary. A necessary transition has a label that must be matched in a composition, whereas a permitted transition can be withdrawn. Necessary transitions can be further distinguished between urgent and lazy, where urgent is the classic notion of uncontrollability, while lazy is a novel notion introduced in the context of contract automata. Lazy transitions can be either controllable or uncontrollable, according to a given predicate evaluated on the whole automaton to which this transition belongs.

The various operations that can be performed on automata are grouped in the package **operations**. `Projection` (cf. [6, Definition 6]) is used to extract a principal automaton from a composed automaton. The main operations are `Composition` (cf. [6, Definition 5]), to compose automata, and `Synthesis` (cf. [10, Definition 5.1]), to refine an automaton to satisfy given predicates. These two classes are further specialised to implement different compositions and the synthesis of an orchestration, the choreography, and the most permissive controller (cf. [10, Theorems 5.3-5.5]).

The **requirements** package groups some invariant requirements that can be enforced in a contract automaton. The `Agreement` requirement (cf. [13, Definition 17]) is an invariant requiring that each transition must not be a request: only offers and matches are allowed. This means that all requests actions are matched, and an agreement is reached. The `StrongAgreement` requirement (cf. [13, Definition 7]) is an invariant allowing only matches. This means that all request and offer actions of principals are matched.

The **family** package (not displayed in Figure 1) groups together the functionalities that extend contract automata to product lines. Featured Modal Contract Automata (FMCA) is the name of this extension. The class `FMCA` implements this type of automata. The family of products is implemented by the class `Family`. Each product is implemented by the class `Product`. Each feature of a product is implemented by the class `Feature`. Following [6, Definition 11], features are identified with the actions of the automaton, and each product identifies a set of required actions (which must be reachable in the automaton) and a set of forbidden actions (which must not be reachable in the automaton).

The import/export packages for the automata and product lines are not reported, but their description is available in the cited technical documentation.

4. Examples of Usage of `CATLib`

This section shows some examples of usage of some functionalities of `CATLib`. We use as case study the game tic-tac-toe. Another available case study is the hotel reservation system we used in [6, 10]. It is described in [49], together with a description of the `.data` format used to store contract automata. The hotel reservation system features clients booking reservations from hotels with intermediary brokers, and it is one of the classical service booking examples that can be found in publications about formal methods for service-oriented computing (cf., e.g., [5, 15, 38, 50–52]). The hotel reservation system case study covers aspects of the initial modelling, while the implementation of a booking application is out-of-scope. On the other hand, the tic-tac-toe case study, being a much simpler application, also covers the implementation. It is described next.

4.1. Tic-tac-toe

The repository [53] contains the tic-tac-toe game example described in this section. It serves two purposes: (i) it provides an example of modelling the game with contract automata, and using the operations of composition and synthesis to compute the strategy for the computer to never lose a game; (ii) it also provides an example of how to use the automata to realise an application (the game) whose control flow is internally orchestrated by the synthesised contracts.

This example showcases some of the benefits of using `CATLib`: the logic of the computer playing against an opponent is synthesised automatically from some initial requirements defined as contract automata (see below). These automata graphically depict the requirements, thus enhancing the documentation (cf. Figure 2). Most importantly, the usage of this formalism guarantees that the software is formally validated by-construction against well-behaving properties (e.g., an occupied position of the grid cannot be occupied again, a user can always select a position which has not yet been occupied by the opponent, no player can perform two consecutive moves, the computer never loses a game). We note that if the implementation were programmed without exploiting the facilities provided by the library (or any similar formal method), then more programming and validation effort would be needed to guarantee such properties.

The synthesised strategy is *most permissive*: all plays in which the computer ties or wins are part of the strategy. To keep this example as simple as possi-

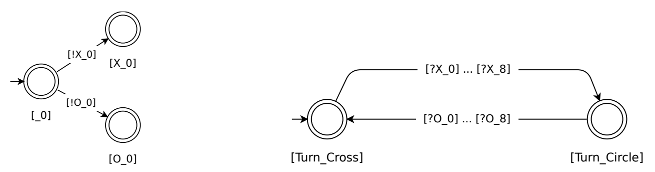

Figure 2: On the left: the automaton for filling position 0 of the grid with either `O` or `X`. On the right: the automaton modelling the turns between players (for displaying purposes, each transition represents a group of 9 transitions whose only difference is the suffix of the label, going from 0 to 8). The images have been created with `CATApp`.

ble, no optimisation is performed to reduce the state space. In particular, the total amount of configurations of the game is 5478. There are 3878 reachable configurations in the strategy of player `X` and 2094 in the strategy of player `O`.

The executable class `App.java` is implemented exploiting the synthesised strategies that are created in another executable class `AppBuildStrategy.java`. `App.java` contains the game, whilst `AppBuildStrategy.java` is used to create (offline) the strategies that are used by `App.java` to play the game. The used automata are defined or generated automatically inside `AppBuildStrategy.java`. Inside the package `grid`, the class `Grid.java` provides methods to check if a configuration is winning for some player or if it is a tie. This class also has facilities to print at console the current configuration of the game, and to import/export a configuration stored as label of a state of an automaton. Finally, inside the package `symbols`, the class `Symbol.java` and its sub-classes `Circle.java` and `Cross.java` are used to store information about the representation of each player.

Synthesis. The strategy synthesis is implemented in `AppBuildStrategy.java`. Firstly, nine automata are instantiated, each one represents the typing of `X` or `O` in a specific position. Each of these automata thus has three states (no symbol, `X`, or `O`). The code below generates the nine automata:

```
//create a list of automata, one for each position, to write either X or O in that position
List<Automaton<String,Action,State<String>,ModalTransition<String, Action, State<String>,
  CALabel>>>  aut = IntStream.range(0, size).mapToObj(i -> {
    State<String> cs_can = new State<>(List.of(new BasicState<>("_" + i, true, true)));
    State<String> cs_cross = new State<>(List.of(new BasicState<>(Cross.cross+"_"+i,false,true)));
    State<String> cs_circle = new State<>(List.of(new BasicState<>(Circle.circle+"_"+i,false,true)));
     return new Automaton<>(Map.of(Cross.cross, cs_cross, Circle.circle, cs_circle)
                .entrySet().stream().map(e -> new ModalTransition<>(cs_can,
                       new CALabel(1, 0, new OfferAction(e.getKey() + "_" + i)),
                       e.getValue(), ModalTransition.Modality.PERMITTED))
               .collect(Collectors.toSet()));}).collect(Collectors.toList());
```

The one for position 0 is displayed in Figure 2 (left side). A further automaton `turns` (depicted in Figure 2 (right side)) is necessary for enforcing turns between `X` and `O`, where `X` is the first to move. This automaton has two states, one for each turn. From each turn/state, there are nine outgoing transitions to the other state. This further automaton is created using the code below (the lists `actionsCross` and `actionsCircle` contain the action objects for `X` and `O`):

```
//creating an automaton requiring turns between X and O
State<String> cs_cross = new State<>(List.of(new BasicState<>("TurnCross", true, true)));
State<String> cs_circle = new State<>(List.of(new BasicState<>("TurnCircle", false, true)));
aut.add(new Automaton<>(Stream.concat( //add cross turn and circle turn transitions
    actionsCross.stream().map(ac -> new ModalTransition<>(cs_cross, new CALabel(1, 0, ac),
                             cs_circle, ModalTransition.Modality.PERMITTED)),
    actionsCircle.stream().map(ac -> new ModalTransition<>(cs_circle, new CALabel(1, 0, ac),
                              cs_cross, ModalTransition.Modality.PERMITTED))
    ).collect(Collectors.toSet())));
```

Next, the composition of these ten automata is computed, in which the requests of `turns` are all matched by the offers of one of the other automata in the composition, such that all transitions in the composition are matches between an offer and a request (i.e., the property of strong agreement is satisfied).

When computing the composition, the pruning predicate of the composition is used both to avoid generating transitions not enjoying strong agreement and to avoid generating transitions from states that are either winning or tying configurations. To check if a state is winning or tying, the class `Grid.java` is used: an object is instantiated by passing as argument the state, and subsequently the corresponding methods of `Grid.java` are invoked. The code below computes and returns the composition:

```
MSCACompositionFunction<String> mcf = new MSCACompositionFunction<>(aut,
    t -> { Grid m = new Grid(t.getSource().toString());
           return new StrongAgreement().negate().test(t.getLabel()) || m.win() || m.tie();});
return mcf.apply(Integer.MAX_VALUE);
```

After that, depending on which player is selected, the composed automaton is edited before starting the synthesis. Indeed, the synthesis algorithm formally guarantees that the strategy has the maximal behaviour where a final state is reachable, dangling states are never traversed, and uncontrollable transitions are never blocked [6, 10, 20]. Accordingly, the moves of the opponent are firstly changed to uncontrollable. Then, only the states where the selected player wins or ties must be marked as successful. To do so, all states of the composition are updated as non-final and a new unique final state is created. Transitions are added from states of the composition where the selected player wins or ties to this new final state. These transitions are labelled with either `tie` or `win`, respectively. The code below synthesises and returns the strategy, in which `transitions` contains the set of transitions updated as described above. We refer to the class `AppBuildStrategy.java` for the full implementation.

```
MpcSynthesisOperator<String> mso = new MpcSynthesisOperator<>(l->true);
return mso.apply(new Automaton<>(transitions));
```

Using the Automata. The class `App.java` realises the game. The application allows to play either `O` or `X`, and loads the corresponding strategy of the opponent. After the game is over, it is possible to start a new play. The control flow is orchestrated by the strategy automaton, which contains all the logic of the game and which has been synthesised automatically. Basically, `App.java` concretises the selection of one of the outgoing transitions from the current state.

The game starts from the initial state (an empty grid). Each state of the automaton contains information on each position of the grid (empty, `O`, or `X`) and whose turn is next. Indeed, these are the ten automata that have been composed. Before selecting the next move, it is checked whether there is an outgoing transition labelled with either `tie` or `win` (i.e., the game is over). If this is not the case, depending on the turn, either the user will type the next move or the computer will select the next move (one of the available outgoing transitions from the current state). During her turn, the user is constrained to insert only positions corresponding to one of the outgoing transitions from the current state. After the transition has been selected (by either the computer or the user), the current state is set to the target state of the transition and these operations are repeated until the game is over. The synthesised automaton also guarantees (by-construction) that it is not possible to assign an already

Phase	Name
Continuous integration	GitHub Actions
Build	Maven
Testing	JaCoCo, Coveralls, SonarCloud
Unit testing	Mockito
Mutation testing	PITest, Stryker
Analysis	SonarCloud, IntelliJ, CodeQL, SpotBugs, Codiga, Codacy

Table 1: Frameworks and services used for evaluating `CATLib`

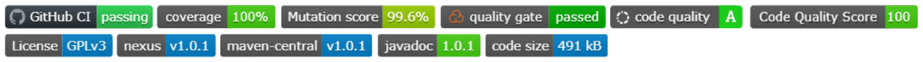

Figure 3: The `GitHub` repository badges (as of May 2022)

occupied position and that the user is never prevented from occupying some position which has not been occupied yet. The main cycle is reported below.

```
currentState = strategy.getInitial(); /* the game starts from the initial state */
while(currentState!=null){ //the forward star is the set of possible next moves in the game
        Set<ModalTransition<String,Action,State<String>,CALabel>> forwardStar =
                                        strategy.getForwardStar(currentState);
        //checking if a winning or tying state is reached, otherwise execute one turn
        if (check(forwardStar)) { currentState=null; } else {
          Symbol turn = (currentState.getState().get(9).getState().equals("TurnCross"))?
                              new Cross() : new Circle();
          if (player.getClass().equals(turn.getClass()))
                        { /* user turn */ currentState = insertPlayer(scan,forwardStar); }
          else  { /* computer turn */ currentState = insertOpponent(forwardStar); }
          System.out.println(new Grid(currentState.toString()));/* printing the grid */ } }
```

5. Software Evaluation Criteria

This section discusses the software evaluation criteria based on the guidelines of *Science of Computer Programming*'s new Software Track on OSP [12]. We detail services and frameworks that were adopted to build, test, and analyse `CATLib`, which are summarised in Table 1. These tools provided data about, e.g., code complexity and coverage, all of which are summarised in Table 2.

5.1. Building, Testing, and Code Quality

The phases of building, testing, and analysis are automatised following the approach in [39] and using state-of-the-art services, as discussed below. Up-to-date links are directly available in the repository of the software artefact as `badges`, graphically depicting (as clickable buttons) the outcome of the corresponding service (e.g., grade of code quality, compilation passed, percentage of code coverage, and mutation testing), as shown in Figure 3.

5.1.1. Continuous Integration

The repository of `CATLib` is hosted on `GitHub`. We use a continuous integration approach, exploiting the `GitHub Actions` offered by `GitHub` to automatise

Source code		Testing	
Measure	Value	Measure	Value
LOC	2519	Total unit tests	462
Total lines	5152	Total integration tests	105
Statements	947	Total tests	567
Functions	223	Unit tests (LOC)	4565
Classes	49	Integration tests (LOC)	1526
Comment lines	1139	Total tests (LOC)	6091
Comments (%)	31.1	Tests line coverage (%)	100
Lines to cover	1238	Tests branch coverage (%)	100
Conditions to cover	626	Total mutants	795
Cyclomatic complexity	630	Killed mutants	780
Cognitive complexity	287	Timed out mutants	12
		Tests ran	1173
		Tests run per mutation	1.48
		Test suite strength (%)	99.6

Table 2: Statistics of evaluating `CATLib`: source code and testing

repetitive tasks. A `.yaml` workflow is followed to automatically build, test, and analyse the software when triggered by given events (e.g., push, pull). The `CATLib` workflow file can be inspected online at [54]. In case of failures during one of the phases, the developers are notified via email and can fix the problem to restore the repository to a version that is successfully built, tested, and analysed. Currently, different services for code quality are used during the workflow (more below). The workflow produces reports that are submitted to the used services. In case of failure during one of the phases, the others will be aborted. These phases are arranged to reduce the computation time in case of failures. `Spot-bugs` analyses in a few seconds the code and thus is placed first, in order to abort the build quickly in case bugs are introduced. After this phase is completed, in parallel (i) the repository is built and tested, and (ii) mutation testing is applied (more details on testing below). These two activities are executed independently. A matrix of operative systems and Java versions is used when building and testing. In particular, `Windows`, `MacOS`, and `Ubuntu` virtual machines are used, with versions 11 and 17 of Java (the two most recent long-term support versions). A final analysis phase is carried out requiring the build and test phase to have been completed.

5.1.2. Testing

The source code is thoroughly tested using the JUnit framework. There are currently 567 tests being executed at each update, of which 462 are unit tests and the remaining 105 are integration tests. The total lines of code (LOC) of the unit tests (measured with the Statistics plugin of IntelliJ) are 4565, whilst 1526 are the LOC of the integration tests, for a total of 6091 LOC of tests, used to test the 2519 LOC of the source code (more below).

For unit testing, the `Mockito` framework is used to mock dependencies of the classes under test. The mocks are used to simulate the behaviour of real objects (e.g., it is possible to verify if a method has been invoked on a mock passed as parameter, or to specify the value to be returned when a given method of a mock is invoked, given certain parameters). In this case, a class is considered a unit to be tested. During integration tests, the mocked classes are substituted with the real classes. The integration tests use the Hotel Reservation case study discussed in Section 4.

Concerning the test coverage, without counting the lines of comments (which are around 30% of the total number of lines), the code base has 2519 LOC. Of these, 1238 are lines to cover by tests (the other lines are, e.g., import statements), and there are a total of 626 conditions to cover. The test suite has 100% line coverage and 100% branch coverage, i.e., all lines and branches are tested. All tests except one are executed on all operative systems of the matrix. The particular test that is not, covers a portion of code that is reachable by throwing an exception when opening a file. Two separate tests are needed, one for Windows and one for Unix systems. In Windows, the file is locked, whilst the POSIX file permissions are removed in the other case.

This coverage is currently documented by two services. One is `SonarCloud`, which is primarily used for code quality and is described below. The other is the service `coveralls.io` and it can be inspected online (the link is available by clicking on the corresponding badge in the repository, cf. Figure 3). `Coveralls` is a web service tracking code coverage over time. The coverage data are submitted to `Coveralls` after all tests have succeeded. The coverage report is generated using `JaCoCo` Java Code Coverage Library, a free code coverage library for Java that is integrated in the Eclipse and IntelliJ IDE (both used to develop `CATLib`). `JaCoCo` computes the line coverage and branch coverage.

Covering all branches helped in finding some left-over parts of expressions in the source code that always evaluated to either true or false (thus it was impossible to cover the other case). Removing them has simplified the code. However, code coverage is a necessary but not sufficient metric to increase confidence on the reliability of the software. Another useful metric can be derived using *mutation testing* (the authors gained prior experience with mutation testing, cf., e.g., [55]). This is a technique to measure the strength of a test suite by introducing artificial mutations in the source code that mimick bugs, so as to check whether the test suite is capable of detecting them. Example of mutations are: change the returned value of a method to null, change an expression in a condition that always evaluates to true or to false or negate it, change comparisons of two values (e.g., swapping `<=` with `<`). A mutant which causes some test to fail is called "killed", whereas those undetected are called "live".

To kill mutations, it is necessary to test many corner cases, which increases the strength of the test suite and, as a consequence, the reliability of the system under test. The strength of the test suite is measured as the number of detected mutants over the number of total mutants. We used `PITest (PIT)`, a state-of-the-art tool for mutation testing for Java. Since no external web page or badge is provided by `PIT`, we used a script to submit each newly generated report of

PIT to another mutation-testing service, called Stryker dashboard, to visualise online the various mutations of the source code and statistics, and to provide a badge for the repository (cf. Figure 3). During mutation testing, 795 mutants are generated, of which 792 are detected (780 are killed by some test and 12 are timed out), with 1173 tests ran (1.48 tests per mutation) reported by PIT.

Only three mutants survive, which are located in the family.converters package. To kill one of these three mutants, we would need a test running for more than one hour, which is not feasible. The other two are redundant mutations (not mutating the source code behaviour) and could be removed from PIT. Ignoring these redundant mutants, the strength of the test suite of CATLib is thus $\approx 100\%$.

Finally, concerning the composition function (CATLib's core functionality), we also formalised the specification of the composition in second-order logic which can be primitively expressed using Java Streams [56]. This is implemented in the class CompositionSpecValidation (under the operations folder of the integration tests), as an implementation of the BooleanSupplier functional interface. In particular, this class is instantiated by passing as arguments the operands of the composition and the computed result. The function will evaluate to true if the result satisfies the conditions for being a composition of the operands. This allows to test automatically if the implementation of the composition satisfies its specification (cf. [6, Definition 5]), and there is no need to use manually validated compositions as control in the tests (which we did anyway as a redundant check).

5.1.3. Code Quality

We used a series of analysis tools to assess and grade the code quality. These services perform automated code reviews and analysis of code quality trends over time. They are used to automatically check if good coding practices have been followed, or if known bugs or vulnerabilities are present in the code. They can also detect design-patterns smells (e.g., god class, violation of encapsulation) and code smells (e.g., too long functions). Some of these analysis tools provide badges grading the quality of the source code, as well as online pages showing the reports of the analysis with various statistics, accessible by clicking on the corresponding badge (cf. Figure 3). CATLib is currently top graded by all the used services, which are described now.

Firstly, we used the analysis tools as provided internally by the used IDEs (IntelliJ and Eclipse). These allowed to clean the code from left-overs, e.g., unused variables or imports, as well as other good practices (e.g., avoid commenting source code, use method reference when possible).

We used SpotBugs in the first phase of the build workflow. SpotBugs is a free software which uses static analysis to look for bugs in Java code. It is well integrated with Maven and supports different extensions. We also used the find-sec-bugs extension to search for potential security problems in the software. In case some bug is spotted, the build fails. This tool highlighted some external vulnerabilities that were fixed (e.g., opening a file whose name comes from an input parameter without filtering the input). `Spotbugs` currently reports zero bugs found in the implementation.

We used `CodeQL`, the code analysis engine developed by `GitHub`. Currently, the analysis is performed during the analysis phase of the workflow. When launched for the first time, a security threat was found in the `FeatureIDE` import functionality, which was due to an unrestricted XML parsing of external entities. Currently, no code scanning alerts are present in the repository.

During the workflow, also the `SonarCloud` analysis service is used, right after the testing has been completed and reports have been submitted to `Coveralls`. When first executed, the analysis reported some problems that have subsequently been fixed. For example, one problem, called catastrophic backtracking, was due to the regular expression parsing provided by Java. To fix this problem, the regular expression library provided by Google has been adopted. Some methods have been decomposed to reduce their cognitive complexity (see below). Examples of other problems highlighted were methods with many parameters, code duplication, and commented code. Currently, there are zero problems reported by this analysis tool.

`SonarCloud` also reports metrics about code coverage (described above), as well as code complexity. For example, currently the source code scores 630 as cyclomatic complexity [57] and 287 as cognitive complexity [58]. The first measures the number of independent paths in the software and it is a measure of code testability (this number is close to the number of branches to cover in the program). Cognitive complexity measures how difficult the control flow is to understand. This measure is roughly a counter incremented each time a control flow structure is encountered (e.g., `if`, `for`) and is incremented commensurated with the level of nesting of control flow structures (e.g., a first-level structure triggers an increment of 1, a second-level structure triggers an increment of 2, and so on) [58]. Currently all methods of `CATLib` are scoring low cognitive and complexity score.

Two other external services have been adopted, called `Codiga` and `Codacy`. These are analysis tools similar to `SonarCloud` whose integration is completely automatised once the rights to access the repository are provided, i.e., these tools do not require to be manually inserted in the workflow, but they download and analyse the source code when triggered by events like commits.

We note that these services may vary over time their offers, hence we refer to the online documentation for up-to-date information.

5.2. Availability, Usability

Concerning availability, the source code is open and available in a public `GitHub` repository under a GPLv3 license [59].

Regarding usability and ease of installation, since `CATLib` is an API written in Java for developing applications using contract automata, we measure this as the easiness of importing the library into a project. Examples of applications created using the library are reported in Section 4.

As stated before, `Maven` is a build automation toolkit that automatically resolves the dependencies of a Java project, and it is supported by `CATLib`.

The compiled binaries of the library (compiled in Java 11) were released in the `Maven Central Repository` [60], which is the default repository used to search for libraries by `Maven`. A release to the `Maven Central Repository` is only allowed if the project meets certain quality requirements [61]. To use the latest library version for a new project, it suffices to copy the dependency [49] to the `Maven` file `pom.xml` of the project, and `CATLib` will automatically be imported together with few dependencies, ready for use.

Concerning reproducibility, the library does not feature any specific experiment to reproduce. However, the whole workflow of building, testing, and analysing the source code is already completely automatised and reproduced externally (comprehending also the results of the examples in Section 4) each time a new update to the software is pushed. These tests are replicated in various operating systems and Java versions. This increases the confidence that all tests and analyses are replicable.

5.3. Documentation

The full `JavaDoc` documentation is accessible from the library's `GitHub` page [46] by clicking on the corresponding badge (cf. Figure 3) and from the deployed artefact. A `github.io` documentation page of the project is available online [49], explaining how to install and use the library through examples. As already stated in Section 3, various documents and sites are available from the repository providing information on the internal implementation (e.g., the class diagrams and packages). This is important to allow other developers to extend the library to their needs. Video tutorials and several examples of usage of the various functionalities of the software have been developed over the years and are available online [62].

6. Conclusion

We have presented `CATLib`, its design, usage examples, recent improvements in terms of quality, availability, usability, reproducibility, and documentation. These improvements have been assessed using established technologies from the open-source community.

Future Work. We plan to apply `CATLib` to develop other tools and examples. As mentioned in the Introduction as related work, a runtime environment called `CARE` [29] is under development using `CATLib`, which currently supports orchestrations. We also plan to implement a choreographed runtime environment, real-time support as formalised in [63], and some examples of service applications. Only the orchestration synthesis supports configurations of products in a product line. The investigation of techniques of decomposition (in the style of choreography synthesis) equipped with variability is a matter of future research. Furthermore, we plan to continue our initial investigation on the usage of `CATLib` to solve multi-agent problems [64]. Finally, it would be interesting to investigate behaviour-driven testing [65, 66] using `CATLib`.

CRediT author statement

D. Basile: (first author) Conceptualization, Visualization, Writing - Original Draft, Software, Validation, Data Curation, Investigation. **M.H. ter Beek:** Writing - Review & Editing, Visualization, Project administration, Funding Acquisition.

Acknowledgements

Research funded by the MIUR PRIN 2017FTXR7S project IT MaTTerS (Methods and Tools for Trustworthy Smart Systems). The authors would like to thank the anonymous reviewers for their excellent suggestions that have improved the paper considerably.

References

[1] M. Bartoletti, T. Cimoli, R. Zunino, Compliance in Behavioural Contracts: A Brief Survey, in: C. Bodei, G. Ferrari, C. Priami (Eds.), Programming Languages with Applications to Biology and Security, Vol. 9465 of LNCS, Springer, 2015, pp. 103–121. `doi:10.1007/978-3-319-25527-9_9`.

[2] C. Peltz, Web Services Orchestration and Choreography, IEEE Comp. 36 (10) (2003) 46–52. `doi:10.1109/MC.2003.1236471`.

[3] A. Bouguettaya, M. Singh, M. Huhns, Q. Z. Sheng, H. Dong, Q. Yu, A. G. Neiat, S. Mistry, B. Benatallah, B. Medjahed, M. Ouzzani, F. Casati, X. Liu, H. Wang, D. Georgakopoulos, L. Chen, S. Nepal, Z. Malik, A. Erradi, Y. Wang, B. Blake, S. Dustdar, F. Leymann, M. Papazoglou, A Service Computing Manifesto: The Next 10 Years, Commun. ACM 60 (4) (2017) 64–72. `doi:10.1145/2983528`.

[4] M. H. ter Beek, A. Bucchiarone, S. Gnesi, Web Service Composition Approaches: From Industrial Standards to Formal Methods, in: Proceedings of the 2nd International Conference on Internet and Web Applications and Services (ICIW'07), IEEE, 2007, pp. 15:1–15:6. `doi:10.1109/ICIW.2007.71`.

[5] D. Basile, P. Degano, G.-L. Ferrari, Automata for Specifying and Orchestrating Service Contracts, Log. Meth. Comp. Sci. 12 (4) (2016) 1–51. `doi:10.2168/LMCS-12(4:6)2016`.

[6] D. Basile, M. H. ter Beek, P. Degano, A. Legay, G.-L. Ferrari, S. Gnesi, F. Di Giandomenico, Controller synthesis of service contracts with variability, Sci. Comput. Program. 187. `doi:10.1016/j.scico.2019.102344`.

[7] P. J. Ramadge, W. M. Wonham, Supervisory Control of a Class of Discrete Event Processes, SIAM J. Control Optim. 25 (1) (1987) 206–230. `doi:10.1137/0325013`.

[8] B. Caillaud, P. Darondeau, L. Lavagno, X. Xie (Eds.), Synthesis and Control of Discrete Event Systems, Springer, 2002. `doi:10.1007/978-1-4757-6656-1`.

[9] M. A. Goorden, L. Moormann, F. F. H. Reijnen, J. J. Verbakel, D. A. van Beek, A. T. Hofkamp, J. M. van de Mortel-Fronczak, M. A. Reniers, W. J. Fokkink, J. E. Rooda, L. F. P. Etman, The Road Ahead for Supervisor Synthesis, in: J. Pang, L. Zhang (Eds.), Proceedings of the 6th International Symposium on Dependable Software Engineering: Theories, Tools, and Applications (SETTA'20), Vol. 12153 of LNCS, Springer, 2020, pp. 1–16. `doi:10.1007/978-3-030-62822-2_1`.

[10] D. Basile, M. H. ter Beek, R. Pugliese, Synthesis of Orchestrations and Choreographies: Bridging the Gap between Supervisory Control and Coordination of Services, Log. Methods Comput. Sci. 16 (2). `doi:10.23638/LMCS-16(2:9)2020`.

[11] Repository of the Contract Automata Library, `https://web.archive.org/web/20220506113357/https://github.com/contractautomataproject/ContractAutomataLib`.

[12] Guidelines of Science of Computer Programming's new Software track on Original Software Publications, `https://web.archive.org/web/20220506113513/https://www.journals.elsevier.com/science-of-computer-programming/call-for-software/a-new-software-track-on-original-software-publications-science-of-computer-programming`.

[13] D. Basile, P. Degano, G.-L. Ferrari, E. Tuosto, Relating two automata-based models of orchestration and choreography, J. Log. Algebr. Meth. Program. 85 (3) (2016) 425–446. `doi:10.1016/j.jlamp.2015.09.011`.

[14] D. Basile, P. Degano, G.-L. Ferrari, E. Tuosto, Playing with Our CAT and Communication-Centric Applications, in: E. Albert, I. Lanese (Eds.), Proceedings 36th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Objects, Components, and Systems (FORTE'16), Vol. 9688 of LNCS, Springer, 2016, pp. 62–73. `doi:10.1007/978-3-319-39570-8_5`.

[15] D. Basile, F. Di Giandomenico, S. Gnesi, P. Degano, G.-L. Ferrari, Specifying Variability in Service Contracts, in: Proceedings of the 11th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'17), ACM, 2017, pp. 20–27. `doi:10.1145/3023956.3023965`.

[16] D. Basile, F. Di Giandomenico, S. Gnesi, FMCAT: Supporting Dynamic Service-based Product Lines, in: Proceedings of the 21st International Systems and Software Product Line Conference (SPLC'17), Vol. 2, ACM, 2017, pp. 3–8. `doi:10.1145/3109729.3109760`.

[17] D. Basile, M. H. ter Beek, F. Di Giandomenico, S. Gnesi, Orchestration of Dynamic Service Product Lines with Featured Modal Contract Automata, in: Proceedings of the 21st International Systems and Software Product

Line Conference (SPLC'17), Vol. 2, ACM, 2017, pp. 117–122. doi:10.1145/3109729.3109741.

[18] D. Basile, F. Di Giandomenico, S. Gnesi, Enhancing Models Correctness through Formal Verification: A Case Study from the Railway Domain, in: L. F. Pires, S. Hammoudi, B. Selic (Eds.), Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD'17), SciTePress, 2017, pp. 679–686. doi:10.5220/0006291106790686.

[19] D. Basile, M. H. ter Beek, S. Gnesi, Modelling and Analysis with Featured Modal Contract Automata, in: Proceedings of the 22nd International Systems and Software Product Line Conference (SPLC'18), Vol. 2, ACM, 2018, pp. 11–16. doi:10.1145/3236405.3236408.

[20] D. Basile, M. H. ter Beek, A Clean and Efficient Implementation of Choreography Synthesis for Behavioural Contracts, in: F. Damiani, O. Dardha (Eds.), Proceedings of the 23rd IFIP WG 6.1 International Conference on Coordination Models and Languages (COORDINATION'21), Vol. 12717 of LNCS, Springer, 2021, pp. 225–238. doi:10.1007/978-3-030-78142-2_14.

[21] S. Tockey, How to Engineer Software: A Model-Based Approach, Wiley, 2019.

[22] K. Henderson, A. Salado, Value and benefits of model-based systems engineering (MBSE): Evidence from the literature, Syst. Eng. 24 (1) (2021) 51–66. doi:10.1002/sys.21566.

[23] R. C. Martin, Clean Code, Prentice Hall, 2008.

[24] D. Boswell, T. Foucher, The Art of Readable Code, O'Reilly, 2011.

[25] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, D. Lea, Java Concurrency in Practice, Addison-Wesley, 2006.

[26] R. Warburton, Java 8 Lambdas: Pragmatic Functional Programming, O'Reilly, 2014.

[27] Repository of the Contract Automata Application, https://web.archive.org/web/20220329023850/https://github.com/ContractAutomataProject/ContractAutomataApp.

[28] JGraphX, https://web.archive.org/web/20220302114658/https://jgraph.github.io/mxgraph/docs/manual_javavis.html.

[29] D. Basile, M. H. ter Beek, A Runtime Environment for Contract Automata (2022). arXiv:2203.14122.

[30] F. Atampore, J. Dingel, K. Rudie, A controller synthesis framework for automated service composition, Discret. Event Dyn. Syst. 29 (3) (2019) 297–365. doi:10.1007/s10626-019-00282-0.

[31] H. Farhat, Web Service Composition via Supervisory Control Theory, IEEE Access 6 (2018) 59779–59789. doi:10.1109/ACCESS.2018.2874564.

[32] M. Barati, R. St-Denis, Behavior Composition Meets Supervisory Control, in: Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics (SMC'15), IEEE, 2015, pp. 115–120. doi:10.1109/SMC.2015.33.

[33] P. Felli, N. Yadav, S. Sardina, Supervisory Control for Behavior Composition, IEEE Trans. Autom. Control 62 (2) (2017) 986–991. doi:10.1109/TAC.2016.2570748.

[34] G. D. Giacomo, F. Patrizi, S. Sardiña, Automatic behavior composition synthesis, Artif. Intell. 196 (2013) 106–142. doi:10.1016/j.artint.2012.12.001.

[35] P. Balbiani, F. Cheikh, G. Feuillade, Composition of interactive Web services based on controller synthesis, in: Proceedings of the IEEE Congress on Services, Part I (SERVICES I 2008), IEEE, 2008, pp. 521–528. doi:10.1109/SERVICES-1.2008.11.

[36] Repository of the Science of Computer Programming Original Software Publications, https://web.archive.org/web/20220506113033/https://github.com/ScienceOfComputerProgramming/.

[37] M. Autili, A. D. Salle, F. Gallo, C. Pompilio, M. Tivoli, CHOReVOLUTION: Service choreography in practice, Sci. Comput. Program. 197. doi:10.1016/j.scico.2020.102498.

[38] R. Guanciale, E. Tuosto, PomCho: A tool chain for choreographic design, Sci. Comput. Program. 202. doi:10.1016/j.scico.2020.102535.

[39] L. Bettini, Test-Driven Development, Build Automation, Continuous Integration (with Java Eclipse and friends), Leanpub, 2019.
URL https://leanpub.com/tdd-buildautomation-ci

[40] A. Ferrari, M. H. ter Beek, Formal Methods in Railways: a Systematic Mapping Study, ACM Comput. Surv.doi:10.1145/3520480.

[41] D. Kouzapas, O. Dardha, R. Perera, S. J. Gay, Typechecking protocols with Mungo and StMungo: A session type toolchain for Java, Sci. Comput. Program. 155 (2018) 52–75. doi:10.1016/j.scico.2017.10.006.

[42] J. Lange, E. Tuosto, N. Yoshida, From Communicating Machines to Graphical Choreographies, in: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'15), ACM, 2015, pp. 221–232. doi:10.1145/2676726.2676964.

[43] CATLib navigable site with packages, classes and documentation, https://contractautomataproject.github.io/ContractAutomataLib/site/index.htm, archived:https://web.archive.org/web/20220506132409/https://contractautomataproject.github.io/ContractAutomataLib/site/index.htm.

[44] CATLib technical report, https://web.archive.org/web/20220506132802/https://contractautomataproject.github.io/ContractAutomataLib/doc/CAT_Lib_doc.pdf.

[45] CATLib diagram report, https://web.archive.org/web/20220506132717/https://contractautomataproject.github.io/ContractAutomataLib/doc/CAT_Lib_diagrams.pdf.

[46] JavaDoc site of CATLib, https://web.archive.org/web/20220506114228/https://javadoc.io/doc/io.github.contractautomataproject/catlib.

[47] CATLib packages report, https://web.archive.org/web/20220623151852/https://contractautomataproject.github.io/ContractAutomataLib/doc/CATLib_Packages.pdf.

[48] CATLib Zenodo repository. doi:10.5281/zenodo.6704433.

[49] GitHub website of CATLib, https://web.archive.org/web/20220623154208/https://contractautomataproject.github.io/ContractAutomataLib/.

[50] M. H. ter Beek, S. Gnesi, N. Koch, F. Mazzanti, Formal Verification of an Automotive Scenario in Service-Oriented Computing, in: Proceedings of the 30th International Conference on Software Engineering (ICSE'08), ACM, 2008, pp. 613–622. doi:10.1145/1368088.1368173.

[51] J. Abreu, F. Mazzanti, J. L. Fiadeiro, S. Gnesi, A Model-Checking Approach for Service Component Architectures, in: D. Lee, A. Lopes, A. Poetzsch-Heffter (Eds.), Proceedings of the Joint 11th IFIP WG 6.1 International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'09) and the 29th IFIP WG 6.1 International Conference on FORmal TEchniques for Distributed Systems (FORTE'09), Vol. 5522 of LNCS, Springer, 2009, pp. 219–224. doi:10.1007/978-3-642-02138-1_15.

[52] M. H. ter Beek, F. Mazzanti, S. Gnesi, CMC–UMC: A Framework for the Verification of Abstract Service-Oriented Properties, in: Proceedings of the ACM Symposium on Applied Computing (SAC'09), ACM, 2009, pp. 2111–2117. doi:10.1145/1529282.1529751.

[53] Repository of tic-tac-toe realised with CATLib, https://web.archive.org/web/20220506112805/https://github.com/contractautomataproject/tictactoe.

[54] GitHub actions workflow of CATLib, https://web.archive.org/web/20220506112704/https://github.com/contractautomataproject/ContractAutomataLib/blob/main/.github/workflows/build.yml.

[55] D. Basile, M. H. ter Beek, S. Lazreg, M. Cordy, A. Legay, Static Detection of Equivalent Mutants in Real-Time Model-based Mutation Testing: An Empirical Evaluation, Empir. Softw. Eng. (2022) doi:10.1007/s10664-022-10149-y.

[56] Y. Cheon, Z. Cao, K. Rahad, Writing JML Specifications Using Java 8 Streams, Tech. Rep. UTEP-CS-16-83, University of Texas at El Paso, https://scholarworks.utep.edu/cs_techrep/1095/ (2016).

[57] T. J. McCabe, A complexity measure, IEEE Trans. Softw. Eng. 2 (4) (1976) 308–320. doi:10.1109/TSE.1976.233837.

[58] G. A. Campbell, Cognitive complexity: an overview and evaluation, in: Proceedings of the 2018 International Conference on Technical Debt (TechDebt'18), ACM, 2018, pp. 57–58. doi:10.1145/3194164.3194186.

[59] GPLv3 license, https://web.archive.org/web/20220411011708/https://www.gnu.org/licenses/gpl-3.0.

[60] Maven Central repository of CATLib, https://web.archive.org/web/20220506112950/https://repo1.maven.org/maven2/io/github/contractautomataproject/catlib/.

[61] Maven Central repository requirements, https://web.archive.org/web/20220116164841/https://central.sonatype.org/publish/requirements/.

[62] Contract Automata tutorials playlist, https://web.archive.org/web/20220506112602/https://www.youtube.com/playlist?list=PLory_2tIDsJvZB2eVlpji-baIz032OTwM.

[63] D. Basile, M. H. ter Beek, A. Legay, Timed service contract automata, Innovations Syst. Softw. Eng. 16 (2) (2020) 199–214. doi:10.1007/s11334-019-00353-3.

[64] D. Basile, M. H. ter Beek, V. Ciancia, An Experimental Toolchain for Strategy Synthesis with Spatial Properties, in: T. Margaria, B. Steffen (Eds.), Proceedings of the 11th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'22), LNCS, Springer, 2022.

[65] J. F. Smart, BDD in Action: Behavior-Driven Development for the whole software lifecycle, Manning, 2014.

[66] C. F. Snook, T. S. Hoang, D. Dghaym, M. J. Butler, T. Fischer, R. Schlick, K. Wang, Behaviour-Driven Formal Model Development, in: J. Sun, M. Sun (Eds.), Proceedings of the 20th International Conference on Formal Engineering Methods (ICFEM'18), Vol. 11232 of LNCS, Springer, 2018, pp. 21–36. `doi:10.1007/978-3-030-02450-5_2`.

Current code version

Nr.	**Code metadata description**	
C1	Current code version	v1.0.1
C2	Permanent link to code/repository used for this code version	`https://github.com/contractautomataproject/ContractAutomataLib`
C3	Permanent link to Reproducible Capsule	`https://doi.org/10.24433/CO.1575879.v1`
C4	Legal Code License	GPLv3
C5	Code versioning system used	Git
C6	Software code languages, tools, and services used	Java 11
C7	Compilation requirements, operating environments & dependencies	OpenJDK 11.0.12 (or compatible), Apache Maven 3.6.0
C8	Link to developer documentation/ manual	`https://contractautomataproject.github.io/ContractAutomataLib`
C9	Support email for questions	davide.basile@isti.cnr.it

Table 3: Code metadata