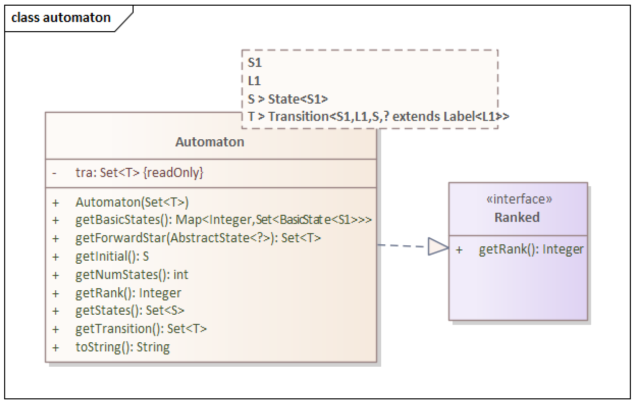

Figure 1: Class diagram of package automaton

1. Software Design

Below we list the packages of `CATLib` and the classes they contain.

io.github.contractautomata.catlib.automaton This automaton package contains the class implementing an automaton. Each `Automaton` has a set of transitions, a set of states, an initial state, and a set of final states. To be composable, an `Automaton` 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. The class diagram is depicted in Figure 1.

io.github.contractautomata.catlib.automaton.label This package groups classes related to labels of automata. `Label` is the super class having a content that is a tuple of a generic type. Labels have a rank (size of the tuple) and implement the `Matchable` interface, to check if two labels match. `CALabel` extends `Label` to implement labels of Contract Automata. Following [1, 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 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. The class diagram is depicted in Figure 2.

io.github.contractautomata.catlib.automaton.label.action This package groups the classes implementing actions of labels. `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

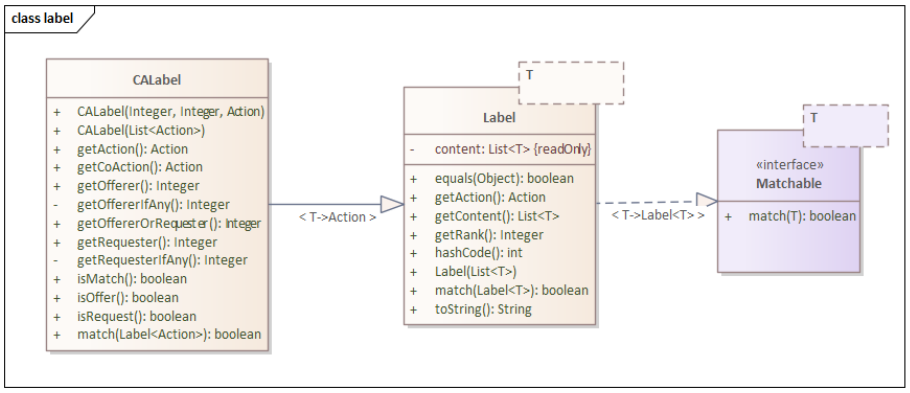

Figure 2: Class diagram of package label

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 [2]. Actions not having an address are 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 [3]. The class diagram is depicted in Figure 3.

io.github.contractautomata.catlib.automaton.state This package groups the classes implementing states of automata. `AbstractState` is the (abstract) super class, 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. The class diagram is depicted in Figure 4.

io.github.contractautomata.catlib.automaton.transition The transition package groups the transitions of an automaton. `Transition` is the super class, it has a source state, a target state and a label. `ModalTransition` extends Transition to include modalities (cf. [4, Definition 9]). 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 con-

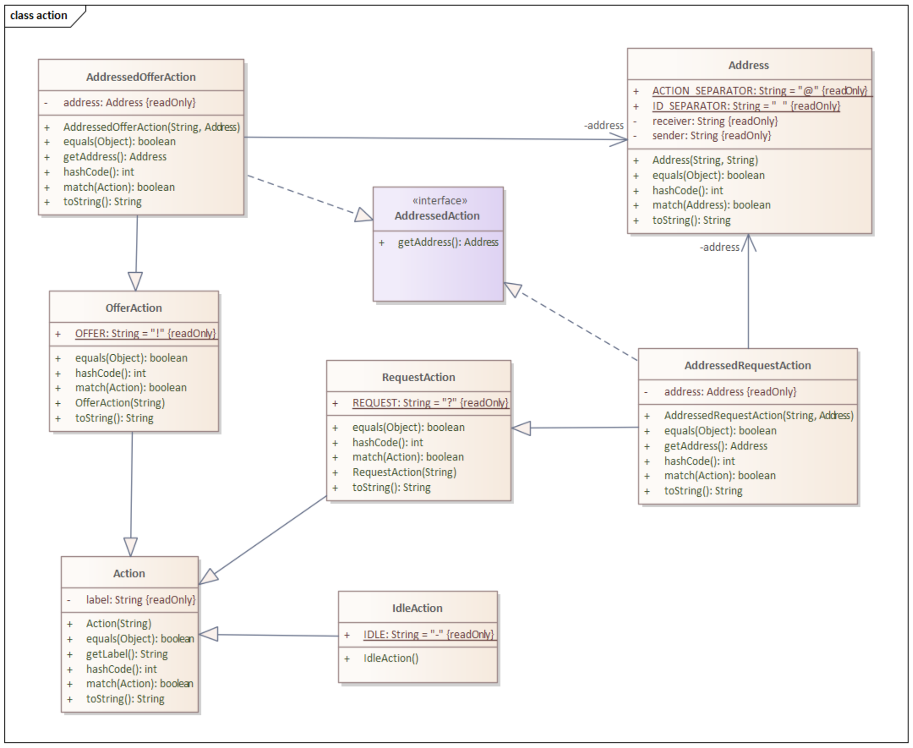

Figure 3: Class diagram of package action

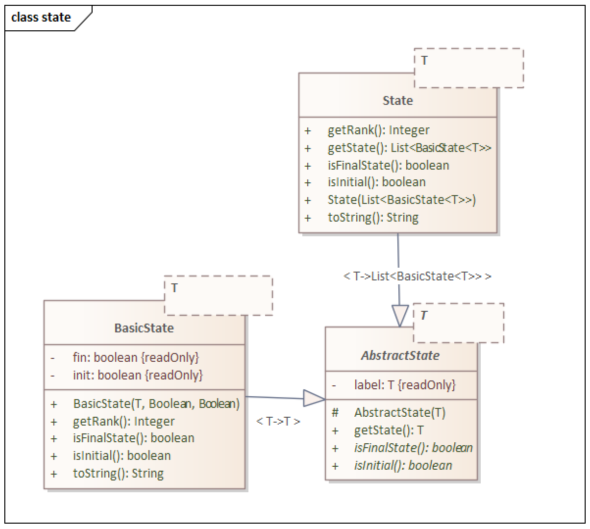

Figure 4: Class diagram of package state

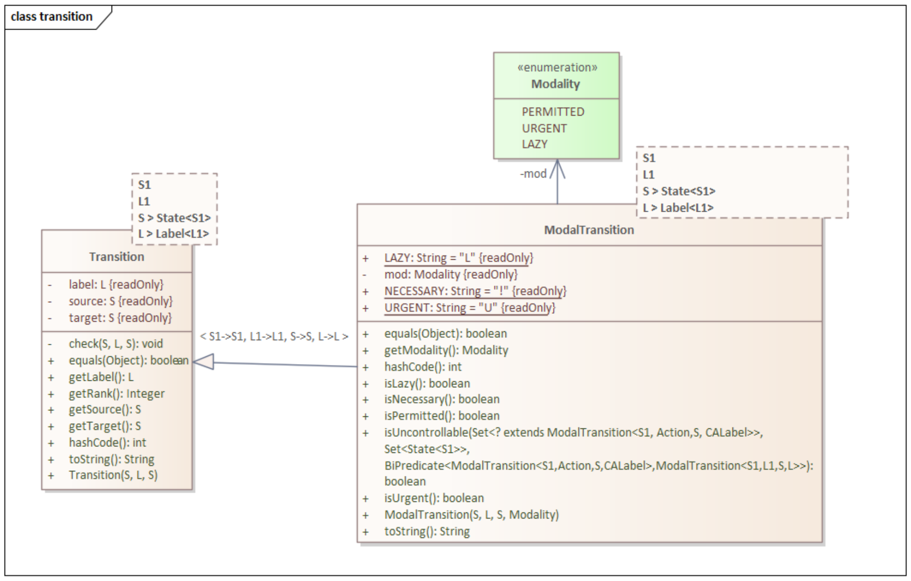

Figure 5: Class diagram of package transition

text 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 class diagram is depicted in Figure 5.

io.github.contractautomata.catlib.converters The converters package contains the classes for I/O operations (import/export). The library contains the class `AutDataConverter`, implementing the interface `AutConverter`, for converting an automaton into a textual format, with extension `.data`. The class diagram is depicted in Figure 6.

io.github.contractautomata.catlib.family The family package 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 [4, 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). FMCA admit the possibility of having partial products (cf. [4, Definition 17]), i.e., products for which the assignment of features is not completely known. The class `PartialProductGenerator` is used for generating all partial products starting from the set of total products, i.e., products in

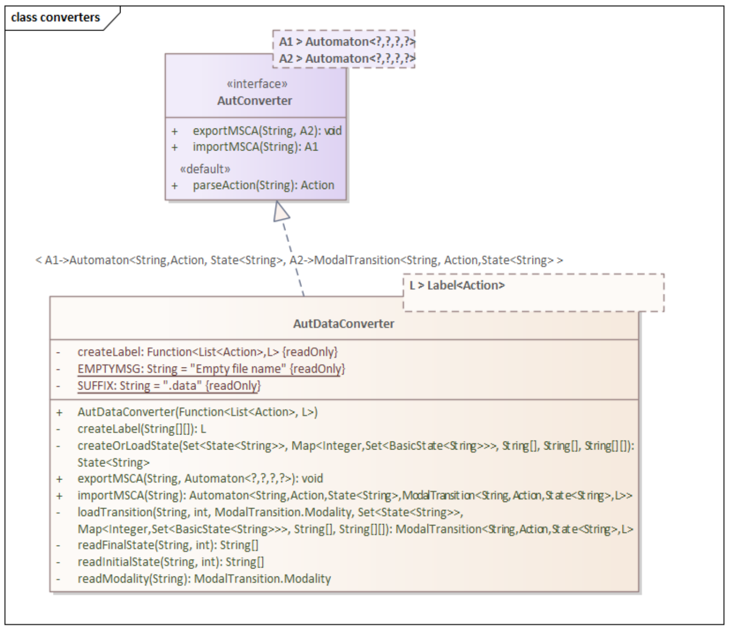
class converters
A1 > Automaton<?,?,?,?>
A2 > Automaton<?,?,?,?>
«interface»
AutConverter
+ exportMSCA(String, A2): void
+ importMSCA(String): A1
«default»
+ parseAction(String): Action
< A1->Automaton<String,Action, State<String>, A2->ModalTransition<String, Action,State<String> >
L > Label<Action>
AutDataConverter
- createLabel: Function<List<Action>,L> {readOnly}
- EMPTYMSG: String = "Empty file name" {readOnly}
- SUFFIX: String = ".data" {readOnly}
+ AutDataConverter(Function<List<Action>, L>)
- createLabel(String[][]): L
- createOrLoadState(Set<State<String>>, Map<Integer,Set<BasicState<String>>>, String[], String[], String[][]): State<String>
+ exportMSCA(String, Automaton<?,?,?,?>): void
+ importMSCA(String): Automaton<String,Action,State<String>,ModalTransition<String,Action,State<String>,L>>
- loadTransition(String, int, ModalTransition.Modality, Set<State<String>>, Map<Integer,Set<BasicState<String>>>, String[], String[][]): ModalTransition<String,Action,State<String>,L>
- readFinalState(String, int): String[]
- readInitialState(String, int): String[]
- readModality(String): ModalTransition.Modality

Figure 6: Class diagram of package converters

which all features are either required or forbidden. The class diagram is depicted in Figure 7.

io.github.contractautomata.catlib.family.converters This package groups the I/O operations for importing or exporting a product line. Each of these converters must implement the interface `FamilyConverter`, with methods for importing/exporting. `ProdFamilyConverter` converts a family to a textual representation, with extension `.prod`. The products generated using the external tool `FeatureIDE` [5, 6] can be imported using `FeatureIDEfamilyConverter`. The class `DimacFamilyConverter` is used to import all products that are models of a formula expressed in the `DIMACS` format, in a file with extension `.dimac`. To generate all models of a formula, the external library `org.ow2.sat4j` is used. The class diagram is depicted in Figure 8.

io.github.contractautomata.catlib.operations This package groups the various operations that can be performed on automata. `Projection` (cf. [4, Definition 6]) is used to extract a principal automaton from a composed automaton. `Relabeling` is used to relabel the states of an automaton. `Union` is used to compute the union of different contract automata. The main operations are `Composition` (cf. [4, Definition 5]), to compose automata, and `Synthesis` (cf. [7, Definition 5.1]), to refine an automaton to satisfy given predicates. These two classes are further specialised to implement dif-

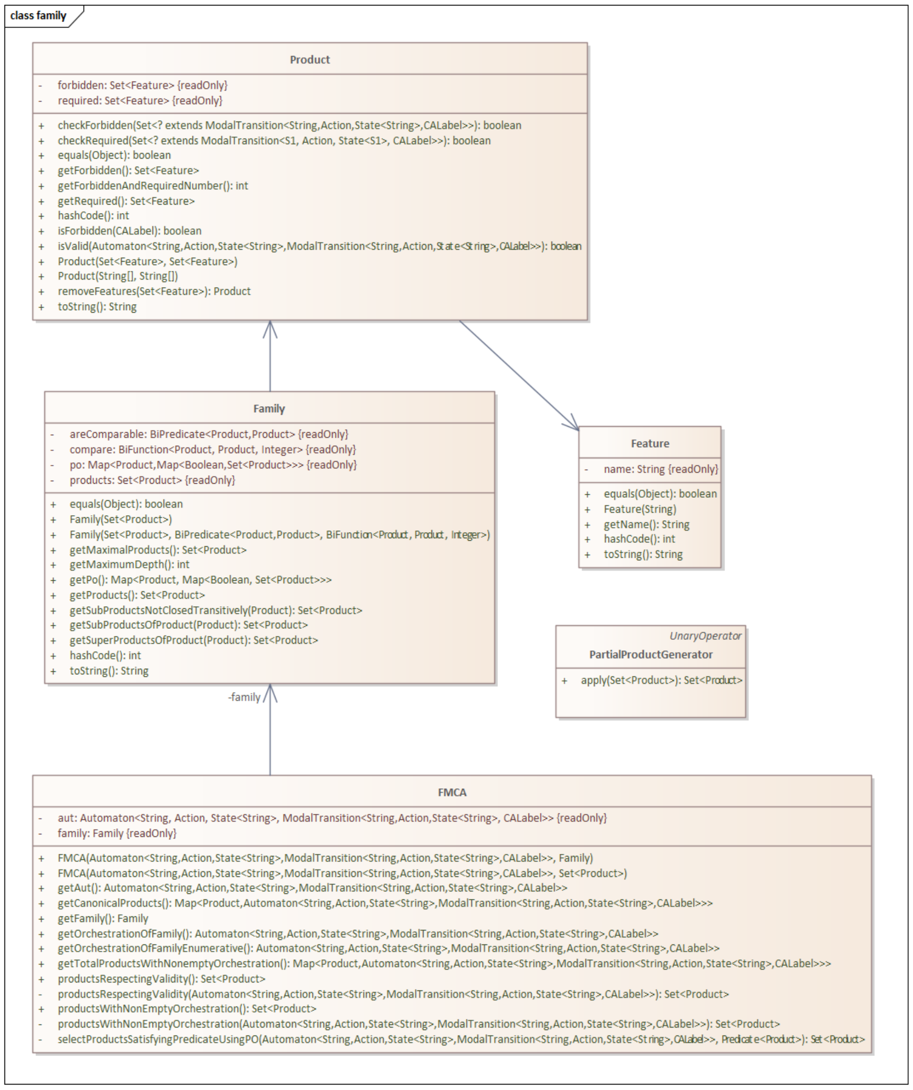
class family
Product
- forbidden: Set<Feature> {readOnly}
- required: Set<Feature> {readOnly}
+ checkForbidden(Set<? extends ModalTransition<String,Action,State<String>,CALabel>>): boolean
+ checkRequired(Set<? extends ModalTransition<S1, Action, State<S1>, CALabel>>): boolean
+ equals(Object): boolean
+ getForbidden(): Set<Feature>
+ getForbiddenAndRequiredNumber(): int
+ getRequired(): Set<Feature>
+ hashCode(): int
+ isForbidden(CALabel): boolean
+ isValid(Automaton<String,Action,State<String>,ModalTransition<String,Action,State<String>,CALabel>>): boolean
+ Product(Set<Feature>, Set<Feature>)
+ Product(String[], String[])
+ removeFeatures(Set<Feature>): Product
+ toString(): String
Family
- areComparable: BiPredicate<Product,Product> {readOnly}
- compare: BiFunction<Product, Product, Integer> {readOnly}
- po: Map<Product,Map<Boolean,Set<Product>>> {readOnly}
- products: Set<Product> {readOnly}
+ equals(Object): boolean
+ Family(Set<Product>)
+ Family(Set<Product>, BiPredicate<Product,Product>, BiFunction<Product, Product, Integer>)
+ getMaximalProducts(): Set<Product>
+ getMaximumDepth(): int
+ getPo(): Map<Product, Map<Boolean, Set<Product>>>
+ getProducts(): Set<Product>
+ getSubProductsNotClosedTransitively(Product): Set<Product>
+ getSubProductsOfProduct(Product): Set<Product>
+ getSuperProductsOfProduct(Product): Set<Product>
+ hashCode(): int
+ toString(): String
Feature
- name: String {readOnly}
+ equals(Object): boolean
+ Feature(String)
+ getName(): String
+ hashCode(): int
+ toString(): String
UnaryOperator
PartialProductGenerator
+ apply(Set<Product>): Set<Product>
-family
FMCA
- aut: Automaton<String, Action, State<String>, ModalTransition<String,Action,State<String>, CALabel>> {readOnly}
- family: Family {readOnly}
+ FMCA(Automaton<String,Action,State<String>,ModalTransition<String,Action,State<String>,CALabel>>, Family)
+ FMCA(Automaton<String,Action,State<String>,ModalTransition<String,Action,State<String>,CALabel>>, Set<Product>)
+ getAut(): Automaton<String,Action,State<String>,ModalTransition<String,Action,State<String>,CALabel>>
+ getCanonicalProducts(): Map<Product,Automaton<String,Action,State<String>,ModalTransition<String,Action,State<String>,CALabel>>>
+ getFamily(): Family
+ getOrchestrationOfFamily(): Automaton<String,Action,State<String>,ModalTransition<String,Action,State<String>,CALabel>>
+ getOrchestrationOfFamilyEnumerative(): Automaton<String,Action,State<String>,ModalTransition<String,Action,State<String>,CALabel>>
+ getTotalProductsWithNonemptyOrchestration(): Map<Product,Automaton<String,Action,State<String>,ModalTransition<String,Action,State<String>,CALabel>>>
+ productsRespectingValidity(): Set<Product>
- productsRespectingValidity(Automaton<String,Action,State<String>,ModalTransition<String,Action,State<String>,CALabel>>): Set<Product>
+ productsWithNonEmptyOrchestration(): Set<Product>
- productsWithNonEmptyOrchestration(Automaton<String,Action,State<String>,ModalTransition<String,Action,State<String>,CALabel>>): Set<Product>
- selectProductsSatisfyingPredicateUsingPO(Automaton<String,Action,State<String>,ModalTransition<String,Action,State<String>,CALabel>>, Predicate<Product>): Set<Product>

Figure 7: Class diagram of package family

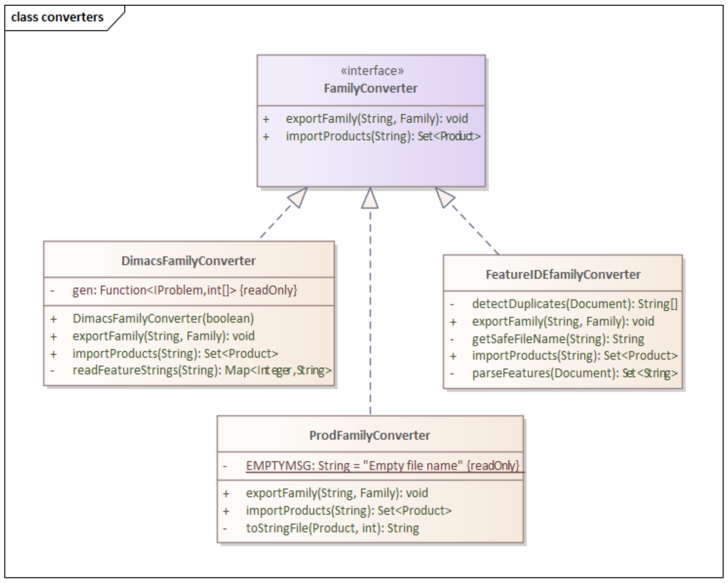
class converters
«interface»
FamilyConverter
+ exportFamily(String, Family): void
+ importProducts(String): Set<Product>
DimacsFamilyConverter
- gen: Function<IProblem,int[]> {readOnly}
+ DimacsFamilyConverter(boolean)
+ exportFamily(String, Family): void
+ importProducts(String): Set<Product>
- readFeatureStrings(String): Map<Integer,String>
FeatureIDEfamilyConverter
- detectDuplicates(Document): String[]
+ exportFamily(String, Family): void
- getSafeFileName(String): String
+ importProducts(String): Set<Product>
- parseFeatures(Document): Set<String>
ProdFamilyConverter
- EMPTYMSG: String = "Empty file name" {readOnly}
+ exportFamily(String, Family): void
+ importProducts(String): Set<Product>
- toStringFile(Product, int): String

Figure 8: Class diagram of package family converters

ferent composition and synthesis operations. `MSCACompositionFunction` instantiates the generic types to those used by a Modal Contract Automaton (MSCA). `ModelCheckingFunction` extends `CompositionFunction` to compose an automaton with a property. `ModelCheckingSynthesisOperator` is used to synthesise an automaton enforcing a given property, using both model checking and synthesis. From this last class, there are three derived operations `MpcSynthesisOperator`, `OrchestrationSynthesisOperator`, and `ChoreographySynthesisOperator` (cf. [7, Theorems 5.3-5.5]).

`ProductOrchestrationSynthesisOperator` further specialises the orchestration synthesis for a given product. Due to its size, a simplified class diagram not reporting the class members is depicted in Figure 9

io.github.contractautomata.catlib.operations.interfaces This auxiliary package groups functional interfaces not offered by the JDK.

io.github.contractautomata.catlib.requirements This package groups some invariant requirements that can be enforced in a contract automaton. The `Agreement` requirement (cf. [3, 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. [3, Definition 7]) is an invariant allowing only matches. This means that all request and offer actions of principals are matched.

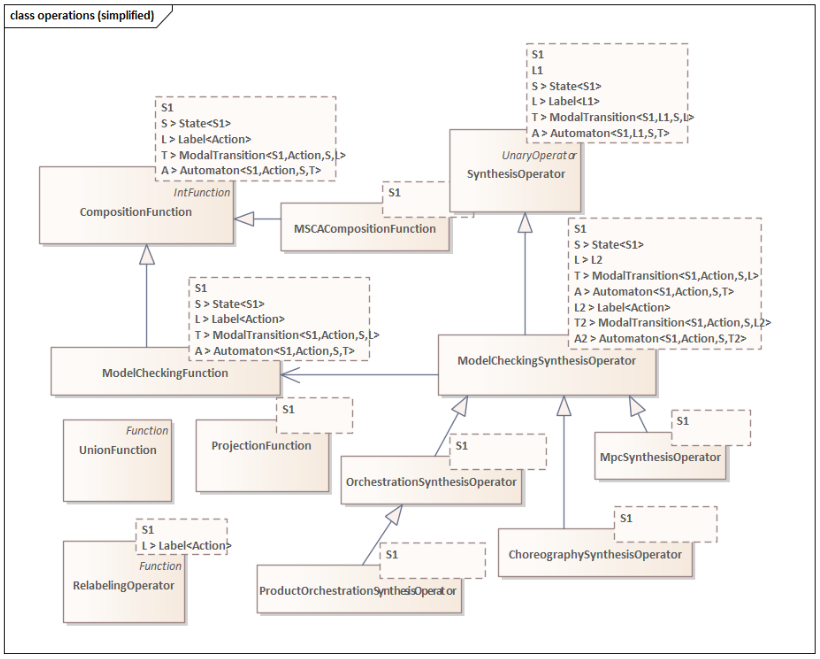
class operations (simplified)
S1
L1
S > State<S1>
L > Label<L1>
T > ModalTransition<S1,L1,S,L>
A > Automaton<S1,L1,S,T>
UnaryOperator
SynthesisOperator
S1
S > State<S1>
L > Label<Action>
T > ModalTransition<S1,Action,S,L>
A > Automaton<S1,Action,S,T>
IntFunction
CompositionFunction
S1
MSCACompositionFunction
S1
S > State<S1>
L > L2
T > ModalTransition<S1,Action,S,L>
A > Automaton<S1,Action,S,T>
L2 > Label<Action>
T2 > ModalTransition<S1,Action,S,L2>
A2 > Automaton<S1,Action,S,T2>
S1
S > State<S1>
L > Label<Action>
T > ModalTransition<S1,Action,S,L>
A > Automaton<S1,Action,S,T>
ModelCheckingFunction
ModelCheckingSynthesisOperator
Function
UnionFunction
S1
ProjectionFunction
S1
OrchestrationSynthesisOperator
S1
MpcSynthesisOperator
S1
L > Label<Action>
Function
RelabelingOperator
S1
ProductOrchestrationSynthesisOperator
S1
ChoreographySynthesisOperator

Figure 9: Simplified class diagram of package operations

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, T. Leich, FeatureIDE: An extensible framework for feature-oriented software development, Sci. Comput. Program. 79 (2014) 70–85. doi:10.1016/j.scico.2012.06.002.

[6] J. Meinicke, T. Thüm, R. Schröter, F. Benduhn, T. Leich, G. Saake, Mastering Software Variability with FeatureIDE, Springer, 2017. `doi:10.1007/978-3-319-61443-4`.

[7] 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`.