# Table of Contents

CAT_Lib........................................................................................................................................... 4

io ....................................................................................................................................................... 4

github .................................................................................................................................................. 4

contractautomata .............................................................................................................................. 4

catlib ................................................................................................................................................... 4

automaton ........................................................................................................................................... 4

automaton diagram ............................................................................................................................. 4

label .................................................................................................................................................... 5

label diagram ....................................................................................................................................... 5

action .................................................................................................................................................. 5

action diagram ..................................................................................................................................... 6

Action .................................................................................................................................................. 6

Address .............................................................................................................................................. 8

AddressedOfferAction ......................................................................................................................... 10

AddressedRequestAction ..................................................................................................................... 12

IdleAction .......................................................................................................................................... 14

OfferAction ......................................................................................................................................... 14

RequestAction ..................................................................................................................................... 16

AddressedAction ............................................................................................................................... 17

CALabel ............................................................................................................................................... 18

Label .................................................................................................................................................. 20

Matchable .......................................................................................................................................... 22

state .................................................................................................................................................... 24

state diagram ....................................................................................................................................... 24

AbstractState ..................................................................................................................................... 24

BasicState .......................................................................................................................................... 25

State ................................................................................................................................................... 27

transition .......................................................................................................................................... 29

transition diagram ............................................................................................................................ 29

ModalTransition ................................................................................................................................ 29

Modality ............................................................................................................................................ 32

Transition .......................................................................................................................................... 33

Automaton ......................................................................................................................................... 35

Ranked ............................................................................................................................................... 37

converters ......................................................................................................................................... 38

converters diagram ............................................................................................................................ 38

AutoDataConverter ............................................................................................................................. 38

AutoConverter .................................................................................................................................... 40

family ................................................................................................................................................ 42

family diagram .................................................................................................................................. 42

converters ......................................................................................................................................... 42

converters diagram ............................................................................................................................ 43

DimacsFamilyConverter ...................................................................................................................... 43

FeatureIDEfamilyConverter .............................................................................................................. 44

ProdFamilyConverter ........................................................................................................................ 45

FamilyConverter ............................................................................................................................... 46
The Contract Automata Library supports the Contract Automata formalism and their operations, and can be easily extended to support similar automata-based formalisms. Currently, synchronous Communicating Machines are also supported by the library. This library is a by-product of scientific research on behavioural types/contracts, and implements results published in international conferences and journals. Using the library it is possible to create new automata, import/export them, and to perform operations on them, as for example computing a composition of contracts, or computing a refinement of a composition satisfying some property (expressed as an automaton or an invariant).

Contract automata are a dialect of Finite State Automata, with special labels and tailored composition and synthesis operations. Contract automata are composable: a composition of contracts is again a contract automaton. Contract automata support as operation the synthesis of the most permissive controller from Supervisory Control Theory for Discrete Event Systems. Contract automata are used to express behavioural contracts, which are used to specify (behavioural) services interfaces, for computing a composition of contracts and synthesise a composition enjoying well-behaving properties, specified as invariants or as automata.

Contract automata formalise behavioural service contracts in terms of service offer actions and service request actions that 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 withdrawn (permitted) in the synthesised coordination. Contract automata can be configured using a product line, where each product (or configuration) predicate on which actions are required and which are forbidden.

For more info and references on publications about Contract Automata (conferences proceedings and journals) check [https://contractautomataproject.github.io/ContractAutomataLib/](https://contractautomataproject.github.io/ContractAutomataLib/)

---

**Package in package 'Package1'**

Davide Basile (ISTI CNR Italy) created on 4/23/2022. Last modified 4/23/2022

**io**

**Package in package 'CAT_Lib'**

Davide Basile (ISTI CNR Italy) created on 4/23/2022. Last modified 4/23/2022

**github**

**Package in package 'io'**

Davide Basile (ISTI CNR Italy) created on 4/23/2022. Last modified 4/23/2022
**contractautomata**

*Package in package 'github'*

contractautomata

Davide Basile (ISTI CNR Italy) created on 4/23/2022. Last modified 4/23/2022

**catlib**

*Package in package 'contractautomata'*

catlib

Davide Basile (ISTI CNR Italy) created on 4/23/2022. Last modified 4/23/2022

**automaton**

The 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 components contained in the automaton. Contract Automata have special labels, implemented inside the package labels. Contract Automata have been introduced (and formalised) in:


*Package in package 'catlib'*

automaton

Davide Basile (ISTI CNR Italy) created on 4/23/2022. Last modified 4/23/2022

**automaton diagram**

*Class diagram in package 'automaton'*

automaton

Version 1.0

Davide Basile (ISTI CNR Italy) created on 4/23/2022. Last modified 4/23/2022
The label 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 implements the Matchable interface, to check if two actions match. CALabel extends Label to implement labels of Contract Automata. In this case labels are list of actions, with specific constraints.

**Package in package 'automaton'**

Davide Basile (ISTI CNR Italy) created on 4/23/2022. Last modified 4/23/2022
The action 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., 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 model for choreographies.

Actions not having an address are used in Contract Automata: in this case the participants are oblivious of the other partners and the model assume the presence of an orchestrator in charge of pairing offers and requests.

**Class diagram in package 'action'**

![Class diagram in package 'action'](image-url)
Action

Class in package 'action'

Class implementing an action of a label. Actions are matchable, i.e., they can match other actions.

Action


OUTGOING STRUCTURAL RELATIONSHIPS

⇨ Realization from Action to Matchable

[ Direction is 'Source -> Destination'. ]

INCOMING STRUCTURAL RELATIONSHIPS

⇨ Generalization from IdleAction to Action

[ Direction is 'Source -> Destination'. ]

⇨ Generalization from RequestAction to Action

[ Direction is 'Source -> Destination'. ]

⇨ Generalization from OfferAction to Action

[ Direction is 'Source -> Destination'. ]

ATTRIBUTES

label : String  Private  Const

the content/label of this action

[ Is static True. Containment is Not Specified. ]

OPERATIONS

Action (label : String ) : Public

Constructor for an action.


equals (o : Object ) : boolean Public

Overides the method of the object class

@return true if the two objects are equal

Properties:

annotations = @Override

OPERATIONS

- **getLabel () : String Public**
  Getter of the content of this action
  @return the label of this action
  

- **hashCode () : int Public**
  Overrides the method of the object class
  @return the hashcode of this object
  
  Properties:
  annotations = @Override
  

- **match (arg : Action ) : boolean Public**
  Implementation of the interface Matchable. True if this action is matching arg. Two actions match if they have the same content.
  @return true if this action matches arg
  
  Properties:
  annotations = @Override
  

- **toString () : String Public**
  Print a String representing this object
  @return a String representing this object
  
  Properties:
  annotations = @Override
  

Address

*Class in package ‘action’*

Class implementing the address of an action. An address is formed by a sender and a receiver. Two addresses are matching if they have the same sender and receiver. Addressed actions are using this class to represent the address of the action.

Address


OUTGOING STRUCTURAL RELATIONSHIPS
### OUTGOING STRUCTURAL RELATIONSHIPS

- Realization from Address to Matchable  
  [Direction is 'Source -> Destination'.]

### ATTRIBUTES

- **ACTION_SEPARATOR : String Public Const = "@"**  
  constant symbol used for separating the address from the action  
  [Is static True. Containment is Not Specified.]

- **ID_SEPARATOR : String Public Const = "]"**  
  constant symbol used for separating the sender from the receiver  
  [Is static True. Containment is Not Specified.]

- **receiver : String Private Const**  
  the receiver  
  [Is static True. Containment is Not Specified.]

- **sender : String Private Const**  
  the sender  
  [Is static True. Containment is Not Specified.]

### ASSOCIATIONS

- **Association (direction: Source -> Destination)**  
  Source: Public (Class) AddressedRequestAction  
  Target: Private address (Class) Address

- **Association (direction: Source -> Destination)**  
  Source: Public (Class) AddressedOfferAction  
  Target: Private address (Class) Address

### OPERATIONS

- **Address (sender : String, receiver : String ) : Public**  
  Constructor for an address  

- **equals (o : Object ) : boolean Public**  
  Overrides the method of the object class  
  @return true if the two objects are equal
## OPERATIONS

### hashCode () : int Public

Overrides the method of the object class

@return the hashcode of this object

### match (arg : Address ) : boolean Public

Two addresses are matching if they have the same sender and receiver.

@return true if the addresses are matching

### toString () : String Public

Print a String representing this object

@return a String representing this object

---

### AddressedOfferAction

*Class in package 'action'*

Class implementing an addressed offer action. It extends offer action and implements addressed action.

AddressedOfferAction


---

### OUTGOING STRUCTURAL RELATIONSHIPS

- Realization from AddressedOfferAction to AddressedAction
  - [Direction is 'Source -> Destination'.]

- Generalization from AddressedOfferAction to OfferAction
  - [Direction is 'Source -> Destination'.]
ATTRIBUTES

- **address : Address** Private Const

  the address of the action [ Is static True. Containment is Not Specified. ]

ASSOCIATIONS

- **Association (direction: Source -> Destination)**
  
  Source: Public (Class) AddressedOfferAction  
  Target: Private address (Class) Address

OPERATIONS

- **AddressedOfferAction (label : String , address : Address ) : Public**

  Constructor for an addressed offer action  

- **equals (o : Object ) : boolean Public**

  Overrides the method of the object class  
  @return true if the two objects are equal

  Properties:
  - annotations = @Override  

- **getAddress () : Address Public**

  Getter of the address of this action  
  @return the address of this action

  Properties:
  - annotations = @Override  

- **hashCode () : int Public**

  Overrides the method of the object class  
  @return the hashcode of this object

  Properties:
  - annotations = @Override  

- **match (arg : Action ) : boolean Public**

  Redefinition of the match of an action. Returns true if arg is an addressed action, the corresponding addresses are matching as well as their super classes. For example, an addressed offer action matches an addressed request action if both addresses are matching and the offer is matching the request.
AddressedRequestAction

Class in package 'action'

Class implementing an addressed request action. It extends request action and implements addressed action.


OUTGOING STRUCTURAL RELATIONSHIPS

Generalization from AddressedRequestAction to RequestAction

Realization from AddressedRequestAction to AddressedAction

ATTRIBUTES

address : Address Private Const

the address of this action

ASSOCIATIONS

Association (direction: Source -> Destination)

Source: Public (Class) AddressedRequestAction

Target: Private address (Class) Address

OPERATIONS

AddressedRequestAction (label : String , address : Address ) : Public
OPERATIONS

Constructor for an addressed request action

equals (o : Object ) : boolean Public

Overrides the method of the object class
@return true if the two objects are equal

Properties:
  annotations = @Override

getAddress () : Address Public

Getter of the address of this action
@return the address of this action

Properties:
  annotations = @Override

hashCode () : int Public

Overrides the method of the object class
@return the hashcode of this object

Properties:
  annotations = @Override

match (arg : Action ) : boolean Public

Redefinition of the match of an action. Returns true if arg is an addressed action, the corresponding addresses are matching as well as their super classes. For example, an addressed offer action matches an addressed request action if both addresses are matching and the offer is matching the request.
@return true if the two actions are matching.

Properties:
  annotations = @Override

toString () : String Public

Print a String representing this object
@return a String representing this object

Properties:
  annotations = @Override
**IdleAction**

*Class in package 'action'*

Class implementing an idle action.

IdleAction


<table>
<thead>
<tr>
<th>OUTGOING STRUCTURAL RELATIONSHIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generalization from IdleAction to Action</td>
</tr>
<tr>
<td>[ Direction is 'Source -&gt; Destination'. ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ATTRIBUTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDLE : String  Public Const = &quot;-&quot;</td>
</tr>
<tr>
<td>Constant symbol denoting an idle action</td>
</tr>
<tr>
<td>[ Is static True. Containment is Not Specified. ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPERATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>IdleAction () : Public</td>
</tr>
<tr>
<td>Constructor for an idle action</td>
</tr>
</tbody>
</table>

**OfferAction**

*Class in package 'action'*

Class implementing an offer action.

OfferAction


<table>
<thead>
<tr>
<th>OUTGOING STRUCTURAL RELATIONSHIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generalization from OfferAction to Action</td>
</tr>
<tr>
<td>[ Direction is 'Source -&gt; Destination'. ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INCOMING STRUCTURAL RELATIONSHIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generalization from AddressedOfferAction to OfferAction</td>
</tr>
<tr>
<td>[ Direction is 'Source -&gt; Destination'. ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ATTRIBUTES</th>
</tr>
</thead>
</table>
### ATTRIBUTES

- **OFFER**: String Public Const = "!"
  
  Constant symbol denoting an offer

### OPERATIONS

- **equals** (o : Object) : boolean Public
  
  Overrides the method of the object class
  @return true if the two objects are equal
  
  Properties:
  annotations = @Override
  [ Is static True. Containment is Not Specified. ]

- **hashCode** () : int Public
  
  Overrides the method of the object class
  @return the hashcode of this object
  
  Properties:
  annotations = @Override

- **match** (arg : Action) : boolean Public
  
  An offer action matches a request action with the same label.
  @return true if this actions matches arg
  
  Properties:
  annotations = @Override

- **OfferAction** (label : String) : Public
  
  Constructor for an offer action

- **toString** () : String Public
  
  Print a String representing this object
  @return a String representing this object
  
  Properties:
  annotations = @Override
**RequestAction**  
*Class in package 'action'*

Class implementing a request action.

---

**Attributes**

- **REQUEST** : String  
  Public Const  
  = "?"  
  Constant symbol denoting a request  
  [ Is static True. Containment is Not Specified. ]

---

**Operations**

- **equals** (o : Object ) : boolean  
  Public  
  Overrides the method of the object class  
  @return true if the two objects are equal  
  Properties:  
  annotations = @Override  

- **hashCode** () : int  
  Public  
  Overrides the method of the object class  
  @return the hashcode of this object  
  Properties:  
  annotations = @Override  

- **match** (arg : Action ) : boolean  
  Public  
  A request action matches an offer action with the same label.  
  @return true if this actions matches arg  
  Properties:
OPERATIONS

```
annotations = @Override
```

![RequestAction (label : String) : Public]

Constructor for a request action

```
```

![toString () : String Public]

Print a String representing this object

```
@return a String representing this object
```

Properties:
```
annotations = @Override
```

---

**AddressedAction**

*Interface in package 'action'*

Interface for an addressed action. An addressed action must provide a method to retrieve the corresponding address.


---

**INCOMING STRUCTURAL RELATIONSHIPS**

- ![Realization from AddressedOfferAction to AddressedAction](#)
  
  [ Direction is 'Source -> Destination'. ]

- ![Realization from AddressedRequestAction to AddressedAction](#)
  
  [ Direction is 'Source -> Destination'. ]

---

**OPERATIONS**

- ![getAddress () : Address Public](#)

  Returns the address of this object

  ```
  @return the address of this object
  ```

---

**CALabel**

*Class in package 'label'*

Class implementing a label of a Contract Automaton, by extending the super class `<code>Label</code>`. The content of each label is a list of actions. Contract automata labels can be of three types:
- offer: one action is an offer action and all the others are idle actions,
- request: one action is a request action and all the others are idle actions,
- match: two actions are matching (i.e., one is a request, the other an offer, and the content is the same) and all the others are idle.

### OUTGOING STRUCTURAL RELATIONSHIPS

- Generalization from CALabel to Label

### OPERATIONS

- **CALabel (rank : Integer , principal : Integer , action : Action ) : Public**
  
  Constructor only used for requests or offer actions, i.e., only one principal is moving. The action must be either a request action or an offer action. The index of the principal moving must be lower than the rank.

- **CALabel (label : List<Action> ) : Public**
  
  Constructor using a list of strings. Each element in the list is the action of the principal at that position. The label must be well-formed (see description of this class).

- **getAction () : Action Public**
  
  If the label is a request it returns the requests action, if it is an offer or match returns the offer action.

- **getCoAction () : Action Public**
  
  Returns the complementary action of the one returned by getAction(). If, for example, getAction() returns an offer, getCoAction() returns a request with the same content.

- **getOfferer () : Integer Public**
  
  Returns the index of the principal performing the offer action. There must be a principal performing an offer action.

- **getOffererIfAny () : Integer Private**
**OPERATIONS**

Returns the index of the principal performing the offer action, or -1 in case no principal is performing an offer.

@return the index of the principal performing the offer actions, or -1 in case no principal is performing an offer.


getOffererOrRequester () : Integer Public

Returns the index of the offerer or requester. The label must not be a match.

@return the index of the offerer or requester.


getRequester () : Integer Public

Returns the index of the principal performing the request action. There must be a principal performing a request action.

@return the index of the principal performing the request action. There must be a principal performing a request action.


getRequesterIfAny () : Integer Private

Returns the index of the principal performing the request action, or -1 in case no principal is performing a request

@return the index of the principal performing the request action, or -1 in case no principal is performing a request.


isMatch () : boolean Public

Returns true if the action is a match

@return true if the action is a match


isOffer () : boolean Public

Returns true if the action is an offer

@return true if the action is an offer


isRequest () : boolean Public

Returns true if the action is a request

@return true if the action is a request


match (label : Label<Action> ) : boolean Public

Implementation of the match method of interface Matchable. Two contract automata labels are matching if their corresponding actions have the same content but with complementary type (i.e., one is a request and the other an offer). The argument must be an instance of CALabel.

@return true if this action matches the label passed as argument

Properties:

annotations = @Override

Label

Class in package 'label'

Class representing a Label of a transition. Each label contains a tuple of elements of unconstrained generic type. The rank is the size of the tuple. Labels can be matched by other labels thanks to the Matchable interface.

Label


OUTGOING STRUCTURAL RELATIONSHIPS

⇒ Realization from Label to Ranked

⇒ Realization from Label to Matchable

INCOMING STRUCTURAL RELATIONSHIPS

⇒ Generalization from CALabel to Label

ATTRIBUTES

.content : List<T> Private Const

the content of the label

OPERATIONS

_equals (obj : Object ) : boolean Public

Overrides the method of the object class
@return true if the two objects are equal

Properties:

annotations = @Override

getAction () : Action Public

This method requires a label to be a list of actions, and requires the actions in the label to be either idle or not, all actions that are not idle must be equals, and at least one action must not be idle. It returns the unique action.
@return the (unique) action of the label
## OPERATIONS

<table>
<thead>
<tr>
<th>Method</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>getContent()</td>
<td>List&lt;T&gt;</td>
</tr>
<tr>
<td>getRank()</td>
<td>Integer</td>
</tr>
<tr>
<td>hashCode()</td>
<td>int</td>
</tr>
<tr>
<td>Label(content : List&lt;T&gt;)</td>
<td>Public</td>
</tr>
<tr>
<td>match(arg : Label&lt;T&gt;)</td>
<td>boolean</td>
</tr>
<tr>
<td>toString()</td>
<td>String</td>
</tr>
</tbody>
</table>

**getContent()**

Getter of the content of this label

@return the content of this label

**getRank()**

Method inherited from the interface Ranked. It returns the rank of the label.

@return the rank of the label.

**hashCode()**

Overrides the method of the object class

@return the hashcode of this object

**Label(content : List<T>)**

Constructor for a label

**match(arg : Label<T>)**

Implementation of the match method of the Matchable interface. Two labels match if their content is equal.

@return true if this label matches with arg label.

**toString()**

Print a String representing this object

@return a String representing this object
**Matchable**

*Interface in package 'label'*

Interface for a matchable element. This interface is implemented by all classes that provide a match method to match other objects of type T. 

@param <T> the type of the object to match with


<table>
<thead>
<tr>
<th>INCOMING STRUCTURAL RELATIONSHIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>➔ Realization from Action to Matchable</td>
</tr>
<tr>
<td>➔ Realization from Address to Matchable</td>
</tr>
<tr>
<td>➔ Realization from Label to Matchable</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPERATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>➩ match (arg : T) : boolean Public</td>
</tr>
<tr>
<td>Returns true if this object matches with arg</td>
</tr>
<tr>
<td>@return true if this object matches with arg</td>
</tr>
</tbody>
</table>
The state package groups the classes implementing states of automata. An 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.

**Package in package 'automaton'**

Davide Basile (ISTI CNR Italy) created on 4/23/2022. Last modified 4/23/2022

**state diagram**

**Class diagram in package 'state'**

Davide Basile (ISTI CNR Italy) created on 4/23/2022. Last modified 4/23/2022

**AbstractState**

**Class in package 'state'**

Class implementing an abstract state of an automaton. An abstract state can be either initial or final, or none, and has a label (its content). 

@param <T> generic type of the content of the state

AbstractState


**OUTGOING STRUCTURAL RELATIONSHIPS**
OUTGOING STRUCTURAL RELATIONSHIPS

Realization from AbstractState to Ranked

[ Direction is 'Source -> Destination'. ]

INCOMING STRUCTURAL RELATIONSHIPS

Generalization from State to AbstractState

[ Direction is 'Source -> Destination'. ]

Generalization from BasicState to AbstractState

[ Direction is 'Source -> Destination'. ]

ATTRIBUTES

label : T  Private  Const

the content of the state

[ Is static True. Containment is Not Specified. ]

OPERATIONS

AbstractState (label : T ) :  Protected

Constructs an abstract state from its label (content). Label must be non-null


getState () : T Public

Getter of the content (of type T) of the state

@return the content of the state


isFinalState () : boolean Public

Returns true if the state is final

@return true if the state is final


isInitial () : boolean Public

Returns true if the state is initial

@return true if the state is initial


BasicState

Class in package 'state'
Class implementing a BasicState of an Automaton. A BasicState implements an AbstractState of rank 1, i.e., it is the internal state of a single principal.

@param <T> generic type of the content of the basic state

---

**OUTGOING STRUCTURAL RELATIONSHIPS**

- Generalization from BasicState to AbstractState

[ Direction is ‘Source -> Destination’. ]

---

**ATTRIBUTES**

- fin : boolean  Private  Const
  the flag signalling if the state is final

[ Is static True. Containment is Not Specified. ]

- init : boolean  Private  Const
  the flag signalling if the state is initial

[ Is static True. Containment is Not Specified. ]

---

**OPERATIONS**

- BasicState (label : T , init : Boolean , fin : Boolean ) : Public
  Constructor for a BasicState. Label must not be a list of elements, and elements cannot be instances of abstract state. In other words, a basic state cannot contain inner states.


- getRank () : Integer Public
  Method inherited from the interface Ranked. The rank of the basic state is always one.
  @return the rank of the basic state, always one.

  Properties:
  annotations = @Override


- isFinalState () : boolean Public
  Returns true if the state is final
  @return true if the state is final

  Properties:
  annotations = @Override

**OPERATIONS**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>isInitial () : boolean Public</td>
<td>Returns true if the state is initial</td>
<td><code>@return true if the state is initial</code></td>
</tr>
<tr>
<td>toString () : String Public</td>
<td>Print a String representing this object</td>
<td><code>@return a String representing this object</code></td>
</tr>
</tbody>
</table>

**State**

*Class in package 'state'*

Class implementing a state of an Automaton. A state is a tuple (list) of basic states of principals. A state has a rank. Rank 1 is for an ensemble containing a single principal. A rank greater than one is for an ensemble of states of principals.  
`@param <T> generic type of the content the basic states`

**OUTGOING STRUCTURAL RELATIONSHIPS**

<table>
<thead>
<tr>
<th>Direction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generalization from State to AbstractState</td>
<td><code>[ Direction is 'Source -&gt; Destination'. ]</code></td>
</tr>
</tbody>
</table>

**OPERATIONS**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>getRank () : Integer Public</td>
<td>Method inherited from the interface Ranked. It returns the rank of the state.</td>
<td><code>@return the rank of the state</code></td>
</tr>
<tr>
<td>getState () : List&lt;BasicState&lt;T&gt;&gt; Public</td>
<td>Getter of the content of this state</td>
<td><code>@return the list of basic states</code></td>
</tr>
<tr>
<td>Method</td>
<td>Return Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>isFinalState () : boolean Public</td>
<td>boolean</td>
<td>Returns true if the state is final</td>
</tr>
<tr>
<td>isInitial () : boolean Public</td>
<td>boolean</td>
<td>Returns true if the state is initial</td>
</tr>
<tr>
<td>State (listState : List&lt;BasicState&lt;T&gt;&gt; ) :</td>
<td>Public</td>
<td>Constructor for a State</td>
</tr>
<tr>
<td>toString () : String Public</td>
<td>String</td>
<td>Print a String representing this object</td>
</tr>
</tbody>
</table>
**transition**

The transition package groups the transitions of an automaton. 
**Transition** is the super class, it has a source and target states and a label. 
**ModalTransition** extends **Transition** to include modalities. 
Modalities of Contract Automata are permitted and necessary. 
A necessary transition has a label that must be match in a composition whilst a permitted transition can be withdrawn. 
Necessary transitions can be further distinguished between urgent and lazy, where urgent is the classic notion of uncontrollability, whereas lazy is a novel notion introduced in 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 to.

**Package in package 'automaton'**

Davide Basile (ISTI CNR Italy) created on 4/23/2022. Last modified 4/23/2022

**transition diagram**

**Class diagram in package 'transition'**

Davide Basile (ISTI CNR Italy) created on 4/23/2022. Last modified 4/23/2022

---

**ModalTransition**

**Class in package 'transition'**

Class implementing a Modal Transition of an Automaton. 
<br> A modal transition is a transition further equipped with a modality. 
<br> Modalities are either permitted and necessary. 
<br> Necessary transitions can be either urgent (i.e., uncontrollable) or lazy. 
<br> A lazy transition can be either controllable or uncontrollable according to a controllability predicate that predicates over the set of transitions of an automaton.

@param <S1> generic type of the content of S 
@param <L1> generic type of the content of L 
@param <S> generic type of the state
@param <L> generic type of the label

ModalTransition


<table>
<thead>
<tr>
<th>ELEMENTS OWNED BY ModalTransition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modality : Enumeration</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OUTGOING STRUCTURAL RELATIONSHIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generalization from ModalTransition to Transition</td>
</tr>
<tr>
<td>Direction is 'Source -&gt; Destination'.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ATTRIBUTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAZY : String Public Const = &quot;L&quot;</td>
</tr>
<tr>
<td>Constant symbol denoting a lazy modality</td>
</tr>
<tr>
<td>[ Is static True. Containment is Not Specified. ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>mod : Modality Private Const</th>
</tr>
</thead>
<tbody>
<tr>
<td>the modality of this transition</td>
</tr>
<tr>
<td>[ Is static True. Containment is Not Specified. ]</td>
</tr>
</tbody>
</table>

| NECESSARY : String Public Const = "!" |
| Constant symbol denoting a necessary modality |
| [ Is static True. Containment is Not Specified. ] |

| URGENT : String Public Const = "U" |
| Constant symbol denoting a urgent modality |
| [ Is static True. Containment is Not Specified. ] |

<table>
<thead>
<tr>
<th>ASSOCIATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Association (direction: Source -&gt; Destination)</td>
</tr>
<tr>
<td>Source: Public (Class) ModalTransition</td>
</tr>
<tr>
<td>Target: Private mod (Enumeration) Modality</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPERATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>equals (obj : Object) : boolean Public</td>
</tr>
<tr>
<td>Overrides the method of the object class</td>
</tr>
<tr>
<td>@return true if the two objects are equal</td>
</tr>
</tbody>
</table>
### OPERATIONS

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Return Type</th>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getModality () : Modality</td>
<td>Public</td>
<td></td>
<td>Getter of modality</td>
</tr>
<tr>
<td>hashCode () : int</td>
<td>Public</td>
<td></td>
<td>Overrides the method of the object class</td>
</tr>
<tr>
<td>isLazy () : boolean</td>
<td>Public</td>
<td></td>
<td>Returns true if the transition is lazy</td>
</tr>
<tr>
<td>isNecessary () : boolean</td>
<td>Public</td>
<td></td>
<td>Returns true if the transition is necessary</td>
</tr>
<tr>
<td>isPermitted () : boolean</td>
<td>Public</td>
<td></td>
<td>Returns true if the transition is permitted</td>
</tr>
<tr>
<td>isUncontrollable (tr, badStates, controllabilityPred) : boolean</td>
<td>Public</td>
<td></td>
<td>Returns true if the transition is uncontrollable. An urgent transition is uncontrollable, a permitted transition is not uncontrollable. A lazy transition is uncontrollable if and only if none of the pairs formed by this transition and a transition t belonging to tr satisfies the controllability predicate, where t must be a match and the source state of t must not be contained in the set badStates.</td>
</tr>
<tr>
<td>isUrgent () : boolean</td>
<td>Public</td>
<td></td>
<td>Returns true if the transition is urgent</td>
</tr>
</tbody>
</table>

**Properties:**

- **annotations = @Override**
OPERATIONS

@return true if the transition is urgent


 ModalTransition (source : S , label : L , target : S , type : Modality ) : Public

Constructing a modal transition from the source, target states, the label and the modality. The modality must be non-null. Requirements of the constructor of the super-class must hold.


 toString () : String Public

Print a String representing this object

@return a String representing this object

Properties:

 annotations = @Override


Modality

Enumeration owned by 'ModalTransition', in package 'transition'

The enum of possible modalities of a transition

Modality

Davide Basile (ISTI CNR Italy) created on 4/23/2022. Last modified 4/23/2022

ATTRIBUTES

 PERMITTED : Public

the permitted modality

[ Stereotype is «enum». Is static True. Containment is Not Specified. ]

 URGENT : Public

the urgent modality

[ Stereotype is «enum». Is static True. Containment is Not Specified. ]

 LAZY : Public

the lazy modality

[ Stereotype is «enum». Is static True. Containment is Not Specified. ]

ASSOCIATIONS

Association (direction: Source -> Destination)

Source: Public (Class) ModalTransition

Target: Private mod (Enumeration) Modality
ASSOCIATIONS

Transition

Class in package 'transition'

Class implementing a Transition of an Automaton. States and Labels are generics, and must inherit from the corresponding super class.

@param <S1> generic type of the content of S
@param <L1> generic type of the content of L
@param <S> generic type of the state
@param <L> generic type of the label


INCOMING STRUCTURAL RELATIONSHIPS

Generalization from ModalTransition to Transition

[ Direction is 'Source -> Destination'. ]

ATTRIBUTES

- label : L Private Const
  the label
  [ Is static True. Containment is Not Specified. ]

- source : S Private Const
  the source state
  [ Is static True. Containment is Not Specified. ]

- target : S Private Const
  the target state
  [ Is static True. Containment is Not Specified. ]

OPERATIONS

- check (source : S, label : L, target : S) : void Private

- equals (obj : Object) : boolean Public
  Overrides the method of the object class
  @return true if the two objects are equal

  Properties:
  annotations = @Override
<table>
<thead>
<tr>
<th>Method</th>
<th>Return Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getLabel()</code> : L Public</td>
<td></td>
<td>Getter of label</td>
</tr>
<tr>
<td><code>getRank()</code> : Integer Public</td>
<td></td>
<td>Method inherited from the interface Ranked. It returns the rank of the transition.</td>
</tr>
<tr>
<td><code>getSource()</code> : S Public</td>
<td></td>
<td>Getter of source state</td>
</tr>
<tr>
<td><code>getTarget()</code> : S Public</td>
<td></td>
<td>Getter of target state</td>
</tr>
<tr>
<td><code>hashCode()</code> : int Public</td>
<td></td>
<td>Overrides the method of the object class</td>
</tr>
<tr>
<td><code>toString()</code> : String Public</td>
<td></td>
<td>Print a String representing this object</td>
</tr>
<tr>
<td><code>Transition(source : S, label : L, target : S)</code>: Public</td>
<td></td>
<td>Constructing a transition from a source and target states and a label Parameters must be non-null, and must have the same rank.</td>
</tr>
</tbody>
</table>
**Automaton**

*Class in package 'automaton'*

This class implements an automaton. An automaton has a set of transitions, a set of states, an initial state and a set of final states. The types of states, transitions, labels of transitions, are all generics and must extend the corresponding super-class. Each automaton object is ranked: it can represent either a single principal, or an ensemble of principals. States and labels are tuples whose size equals the rank of the automaton.

- @param <S1> the generic type in State<S1>, the content of a state.
- @param <L1> the generic type in Label<L1>, the content of a label.
- @param <S> the generic type of states
- @param <T> the generic type of transitions


### OUTGOING STRUCTURAL RELATIONSHIPS

- Realization from Automaton to Ranked

- ATTRIBUTES

  - tra : Set<T> Private Const

    The set of transitions of the automaton

- ASSOCIATIONS

  - Association (direction: Source -> Destination)

    Source: Public (Class) FMCA
    
    Target: Private aut (Class) Automaton

- OPERATIONS

  - Automaton (tr : Set<T> ) : Public

    This constructor builds an automaton from its set of transitions.


  - getBasicStates () : Map<Integer,Set<BasicState<S1>>> Public

    Returns a map where for each entry the key is the index of principal, and the value is its set of basic states. It is required that states are lists of basic states.

    @return a map where for each entry the key is the index of principal, and the value is its set of basic states


  - getForwardStar (source : AbstractState<?>) : Set<T> Public
## OPERATIONS

Returns the set of transitions outgoing from the state source
@return set of transitions outgoing state source

- getInitial () : S Public

Returns the unique initial state
@return the unique initial state

- getNumStates () : int Public

It returns the number of states of the automaton.
@return the number of states of the automaton

- getRank () : Integer Public

Method inherited from the interface Ranked. It returns the rank of the automaton.
@return the rank of the automaton

- getStates () : Set<S> Public

Returns the states of the automaton.
@return all states that appear in at least one transition

- getTransition () : Set<T> Public

Getter of the set of transitions
@return the set of transitions

- toString () : String Public

Print a String representing this object
@return a String representing this object

**Ranked**

*Interface in package 'automaton'*
This interface is implemented by ranked elements. An element is ranked if it has a rank. An element of rank 1 represents a principal. For ranks greater than one, the corresponding element represents an ensemble of principals.

**INCOMING STRUCTURAL RELATIONSHIPS**

- **Realization from AbstractState to Ranked**
  
  [ Direction is 'Source -> Destination'. ]

- **Realization from Automaton to Ranked**

  [ Direction is 'Source -> Destination'. ]

- **Realization from Label to Ranked**

  [ Direction is 'Source -> Destination'. ]

**OPERATIONS**

- **getRank () : Integer Public**

  Returns the rank of this object
  @return the rank of this object

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 in a textual format, with extension `.data`.

Package in package 'catlib'

converters

Davide Basile (ISTI CNR Italy) created on 04/23/2022. Last modified 04/23/2022

converters diagram

Class diagram in package 'converters'

Davide Basile (ISTI CNR Italy) created on 04/23/2022. Last modified 04/23/2022

Figure 6: converters

**AutDataConverter**

Class in package 'converters'

This class supports the conversion of an automaton into a textual format, with extension `<code>.data</code>`. `<br>`
@param `<L>` the type of the label of the automaton to import, must extend `<code>Label<Action>`

`AutDataConverter`

Davide Basile created on 04/23/2022. Last modified 04/29/2022
### OUTGOING STRUCTURAL RELATIONSHIPS

- Realization from AutDataConverter to AutConverter

[Direction is 'Source -> Destination'.]

### ATTRIBUTES

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
<th>Containment</th>
</tr>
</thead>
<tbody>
<tr>
<td>createLabel</td>
<td>Function&lt;List&lt;Action&gt;, L&gt;</td>
<td>Private Const</td>
<td>a builder of a label of type L from a list of actions</td>
</tr>
<tr>
<td>EMPTYMSG</td>
<td>String</td>
<td>Private Const = &quot;Empty file name&quot;</td>
<td>message to show in case of an empty file name</td>
</tr>
<tr>
<td>SUFFIX</td>
<td>String</td>
<td>Private Const = &quot;.data&quot;</td>
<td>suffix, the used file extension</td>
</tr>
</tbody>
</table>

### OPERATIONS

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Properties</th>
</tr>
</thead>
</table>
Operations

@return the imported automaton, where the content of each state and action is a String, labels are of type L, and transitions can have modalities

Properties:
    throws = IOException


loadTransition (str : String, rank : int, type : ModalTransition.Modality, states : Set<State<String>>, mapBasicStates : Map<Integer,Set<BasicState<String>>> , initial : String[], fin : String[][] ) : ModalTransition<String,Action,State<String>,L> Private

Properties:
    throws = IOException


readFinalState (strLine : String, rank : int ) : String Private


readInitialState (strLine : String, rank : int ) : String Private


readModality (strLine : String ) : ModalTransition.Modality Private


AutConverter

Interface in package 'converters'

The interface used to import/export automata. Each converter must implement this interface.
@param <A1> the type of the automaton to import
@param <A2> the type of the automaton to export


Incoming Structural Relationships

⇒ Realization from AutDataConverter to AutConverter

[ Direction is 'Source -> Destination'. ]

Operations

exportMSCA (filename : String, aut : A2 ) : void Public

This method is used to store an automaton into a file

Properties:
    throws = ParserConfigurationException,IOException,TransformerException

OPERATIONS

importMSCA (filename : String ) : A1 Public

This method is used to import an automaton of type A1 stored in a file
@return an automaton of type A1 loaded from the file

Properties:
throws = IOException,ParserConfigurationException,SAXException

parseAction (action : String ) : Action Public

This method provides facilities for parsing a string encoding a textual representation of an action into an object Action. If the string is not parsable a run-time exception is thrown.
@return the object Action encoded in the parameter
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. FMCA exploits the possibility of having partial products, i.e., products where 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 where all features are either assigned or not.

The extension of Contract Automata to product lines is fully specified in:


Package in package 'catlib'

Davide Basile (ISTI CNR Italy) created on 4/23/2022. Last modified 4/23/2022

family diagram

Class diagram in package 'family'

Davide Basile (ISTI CNR Italy) created on 4/23/2022. Last modified 4/23/2022
converters

The family.converters package groups the I/O operations of import/export of 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.
FeatureIDEFamilyConverter imports the products generated using the tool FeatureIDE.
DimacFamilyConverter imports all products that are models of a formula expressed in DIMAC format,
in a file with extension .dimac

Package in package 'family'

converters

Davide Basile (ISTI CNR Italy) created on 4/23/2022. Last modified 4/23/2022

converters diagram

Class diagram in package 'converters'
Figure 8: converters

**DimacsFamilyConverter**

*Class in package 'converters'*

Class for importing and exporting DIMACS CNF models as families of products. <br>The DIMACS CNF format is a textual representation of a formula in conjunctive normal form. <br>It is the standard format for SAT solvers. <br>

DimacsFamilyConverter


<table>
<thead>
<tr>
<th>OUTGOING STRUCTURAL RELATIONSHIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>← Realization from DimacsFamilyConverter to FamilyConverter</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ATTRIBUTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>gen : Function&lt;IProblem,int[]&gt; Private Const</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPERATIONS</th>
</tr>
</thead>
</table>

| exportFamily (filename : String , fam : Family ) : void Public |
| Operation not supported. |

Properties:
annotations = @Override
OPERATIONS

importProducts (filename : String ) : Set<Product> Public

Overides the FamilyConverter method.
@return a set of products generated from the DIMACS filename.

Properties:
  annotations = @Override
  throws = IOException, ParseFormatException, ContradictionException, TimeoutException

readFeatureStrings (filename : String ) : Map<Integer,String> Private

Properties:
  throws = IOException

FeatureIDEfamilyConverter

Class in package 'converters'

Class implementing import/export of products generated by FeatureIDE.

FeatureIDEfamilyConverter


OUTGOING STRUCTURAL RELATIONSHIPS

Realization from FeatureIDEfamilyConverter to FamilyConverter

[ Direction is 'Source -> Destination'. ]

OPERATIONS

detectDuplicates (doc : Document ) : String Private

reads all iff constraints (eq node) and returns a table such that forall i table[i][0] equals table[i][1]

exportFamily (filename : String , fam : Family ) : void Public

Overides the method of FamilyConverter. This operation is not supported.

Properties:
  annotations = @Override

getSafeFileName (filename : String ) : String Private

### OPERATIONS

#### importProducts (filename : String) : Set<Product> Public

Import the list of products generated through FeatureIDE.  
@return the imported set of products

Properties:
- annotations = @Override
- throws = ParserConfigurationException, SAXException, IOException
  
#### parseFeatures (doc : Document) : Set<String> Private

Properties:
- annotations = @Override
- throws = IOException

---

### ProdFamilyConverter

*Class in package 'converters'*

Class implementing import/export from the `<code>.prod</code>` textual format.

ProdFamilyConverter


#### OUTGOING STRUCTURAL RELATIONSHIPS

- Realization from ProdFamilyConverter to FamilyConverter

#### ATTRIBUTES

- EMPTYMSG : String Private Const = "Empty file name"

#### OPERATIONS

#### exportFamily (filename : String, fam : Family) : void Public

Overrides the method of FamilyConverter

Properties:
- annotations = @Override
- throws = IOException

#### importProducts (filename : String) : Set<Product> Public

Overrides the method of FamilyConverter

@return a set of products loaded from filename, representing a family of products
**FamilyConverter**

*Interface in package 'converters'*

This is the interface to be implemented for importing/exporting a family.


### INCOMING STRUCTURAL RELATIONSHIPS

- **Realization from FeatureIDEfamilyConverter to FamilyConverter**
  [Direction is 'Source -> Destination'.]

- **Realization from DimacsFamilyConverter to FamilyConverter**
  [Direction is 'Source -> Destination'.]

- **Realization from ProdFamilyConverter to FamilyConverter**
  [Direction is 'Source -> Destination'.]

### OPERATIONS

- **exportFamily (filename : String , fam : Family ) : void Public**

  Stores the content of the family fam in the file filename.

  Properties:
  - `throws = IOException`

- **importProducts (filename : String ) : Set<Product> Public**

  Returns a set of products loaded from filename, representing a family of products, imported from filename.

  @return a set of products loaded from filename, representing a family of products

  Properties:
  - `throws =`
**Family**

*Class in package 'family'*

Class implementing a family of products (i.e., a product line). A family is represented by its products (or configurations). In featured modal contract automata, partial products are also considered, also known as sub-families. In a partial product not all features are rendered as required or forbidden. The sub-products are partially ordered. The formal definitions can be found in:


---

**ATTRIBUTES**

- **areComparable**: BiPredicate<Product,Product> Private Const
  
a predicate for checking if two products are comparable.  
  
  [ Is static True. Containment is Not Specified. ]

- **compare**: BiFunction<Product, Product, Integer> Private Const
  
a predicate for comparing two comparable products.  
  
  [ Is static True. Containment is Not Specified. ]

- **po**: Map<Product,Map<Boolean,Set<Product>>> Private Const
  
  the partial order of products. A map such that for each product (key) a map is returned (value). The value is a partitioning in false/true the sub/super products of the key, where a sub product contains all the features (required/forbidden) of its super product.  
  
  [ Is static True. Containment is Not Specified. ]

- **products**: Set<Product> Private Const
  
  the set of products.  
  
  [ Is static True. Containment is Not Specified. ]

**ASSOCIATIONS**

- **Association** (direction: Source -> Destination)

  Source: Public (Class) Family  
  
  Target: Public (Class) Product
### ASSOCIATIONS

<table>
<thead>
<tr>
<th>Association (direction: Source -&gt; Destination)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source: Public (Class) FMCA</td>
</tr>
<tr>
<td>Target: Private family (Class) Family</td>
</tr>
</tbody>
</table>

### OPERATIONS

#### equals (obj : Object) : boolean Public

Overrides the method of the object class
@return true if the two objects are equal

Properties:
- annotations = @Override

#### Family (products : Set<Product>) : Public

Constructor of a family from a set of products. In this constructor, two products are comparable if one (p1) contains all required and forbidden features of the other (p2), and in this case p1 is less than p2.

#### Family (products : Set<Product>, areComparable : BiPredicate<Product,Product>, compare : BiFunction<Product, Product, Integer>) : Public

Constructor of a family from a set of products, and the predicates for the partial order.

#### getMaximalProducts () : Set<Product> Public

Returns all maximal products p s.t. there is no p′ greater than p.
@return all maximal products p s.t. there is no p′ greater than p.

#### getMaximumDepth () : int Public

Returns the maximum number of features available for a product in this product-line, i.e., the maximum depth of the partial order.
@return the maximum number of features available for a product in this product-line, i.e., the maximum depth of the partial order.

#### getPo () : Map<Product, Map<Boolean, Set<Product>>> Public

Getter of the partial order of products.
@return the partial order of products.

#### getProducts () : Set<Product> Public

Getter of the set of products.
### OPERATIONS

@return the set of products.

<table>
<thead>
<tr>
<th>Is static</th>
<th>Is abstract</th>
<th>Is return array</th>
<th>Is query</th>
<th>Is synchronized</th>
</tr>
</thead>
</table>

#### getSubProductsNotClosedTransitively

(p : Product) → Set<Product> Public

Returns the sub-products of prod not closed transitively. These are all sub-products of p such that, given two of them, it is never the case that one is a sub-product of the other.

@return the sub-products not closed transitively of prod.

<table>
<thead>
<tr>
<th>Is static</th>
<th>Is abstract</th>
<th>Is return array</th>
<th>Is query</th>
<th>Is synchronized</th>
</tr>
</thead>
</table>

#### getSubProductsOfProduct

(prod : Product) → Set<Product> Public

Returns the sub-products of prod.

@return the sub-products of prod.

<table>
<thead>
<tr>
<th>Is static</th>
<th>Is abstract</th>
<th>Is return array</th>
<th>Is query</th>
<th>Is synchronized</th>
</tr>
</thead>
</table>

#### getSuperProductsOfProduct

(prod : Product) → Set<Product> Public

Returns the super-products of prod.

@return the super-products of prod.

<table>
<thead>
<tr>
<th>Is static</th>
<th>Is abstract</th>
<th>Is return array</th>
<th>Is query</th>
<th>Is synchronized</th>
</tr>
</thead>
</table>

#### hashCode

: int Public

Overrides the method of the object class

@return the hashcode of this object

Properties:

<table>
<thead>
<tr>
<th>annotations = @Override</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Is static</th>
<th>Is abstract</th>
<th>Is return array</th>
<th>Is query</th>
<th>Is synchronized</th>
</tr>
</thead>
</table>

#### toString

: String Public

Print a representation of this object as String

@return a representation of this object as String

Properties:

<table>
<thead>
<tr>
<th>annotations = @Override</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Is static</th>
<th>Is abstract</th>
<th>Is return array</th>
<th>Is query</th>
<th>Is synchronized</th>
</tr>
</thead>
</table>

---

**Feature**

*Class in package 'family'*

Class implementing a feature of product.

Feature

ATTRIBUTES

- name : String  Private  Const
  the name of the feature
  [ Is static True. Containment is Not Specified. ]

ASSOCIATIONS

- Association (direction: Source -> Destination)
  Source: Public (Class) Product
  Target: Public (Class) Feature

OPERATIONS

- equals (obj : Object) : boolean Public
  Overrides the method of the object class
  @return true if the two objects are equal
  Properties:
  annotations = @Override

- Feature (name : String) : Public
  Constructor for a feature

- getName () : String Public
  Getter of the name of the feature
  @return the name of the feature

- hashCode () : int Public
  Overrides the method of the object class
  @return the hashcode of this object
  Properties:
  annotations = @Override

- toString () : String Public
  Print a representation of this object as String
  @return a representation of this object as String
  Properties:
  annotations = @Override
FMCA

Class implementing a Featured Modal Contract Automaton (FMCA). An FMCA pairs a modal contract automaton with a family, and provides operations on this pair.


ATTRIBUTES

- aut : Automaton<String, Action, State<String>, ModalTransition<String,Action,State<String>, CALabel>> Private Const
  the modal contract automaton.

- family : Family Private Const
  the family.

ASSOCIATIONS

- Association (direction: Source -> Destination)
  Source: Public (Class) FMCA
  Target: Private family (Class) Family

- Association (direction: Source -> Destination)
  Source: Public (Class) FMCA
  Target: Private aut (Class) Automaton

OPERATIONS

- FMCA (aut : Automaton<String,Action,State<String>,ModalTransition<String,Action,State<String>,CALabel>>, family : Family ) : Public
  Constructor for an FMCA from an automaton and a family.

- FMCA (aut : Automaton<String,Action,State<String>,ModalTransition<String,Action,State<String>,CALabel>>, products : Set<Product>) : Public
  This constructor instantiates the family of products by performing a pre-processing, to polish the set of products prod given as
Firstly, all features that are not labels of the given automaton are removed from the products. After that, redundant products are removed (those requiring features present in aut but not in its orchestration in agreement). 

```java
getAut(): Automaton<String, Action, State<String>, ModalTransition<String, Action, State<String>, CALabel>>
```

Getter of the automaton.

```java
getCanonicalProducts(): Map<Product, Automaton<String, Action, State<String>, ModalTransition<String, Action, State<String>, CALabel>>
```

Returns the canonical products of this FMCA. A canonical product represents all the maximal elements in the FMCA that have the same set of forbidden actions. It is required that the automaton does not contain transitions labelled with "dummy" (these labels are generated when computing the union of a set of automata).

```java
getFamily(): Family
```

Getter of the family.

```java
getOrchestrationOfFamily(): Automaton<String, Action, State<String>, ModalTransition<String, Action, State<String>, CALabel>>
```

Returns the orchestration of the family as the union of orchestrations of canonical products.

```java
getOrchestrationOfFamilyEnumerative(): Automaton<String, Action, State<String>, ModalTransition<String, Action, State<String>, CALabel>>
```

Returns the orchestration of the family as the union of orchestrations of total products.

```java
getTotalProductsWithNonemptyOrchestration(): Map<Product, Automaton<String, Action, State<String>, ModalTransition<String, Action, State<String>, CALabel>>
```

Returns a map pairing a product with its non-empty orchestration in agreement.

```java
productsRespectingValidity(): Set<Product>
```

Returns the set of products respecting validity. A product p is respecting validity iff all the mandatory actions in p correspond to executable transitions in the automaton and no action forbidden in p have executable counterparts in the automaton.

This method exploits the partial order so it starts from maximal products.
OPERATIONS

@return the set of products respecting validity


productsRespectingValidity (a : Automaton<String,Action,State<String>,ModalTransition<String,Action,State<String>,CALabel>> ) : Set<Product> Private


productsWithNonEmptyOrchestration () : Set<Product> Public

The set of products with non-empty orchestration in agreement.

@return the set of products with non-empty orchestration in agreement.


productsWithNonEmptyOrchestration (aut : Automaton<String,Action,State<String>,ModalTransition<String,Action,State<String>,CALabel>> ) : Set<Product> Private


selectProductsSatisfyingPredicateUsingPO (a : Automaton<String,Action,State<String>,ModalTransition<String,Action,State<String>,CALabel>> , pred : Predicate<Product> ) : Set<Product> Private


---

PartialProductGenerator

Class in package 'family'

Class implementing the partial product generation operator. <br>This operator takes in input a set of total products (with all features assigned), <br> and returns a set of products comprehending total products and partial products (not all features assigned). <br>This operator is similar to the Quine-McCluskey algorithm. <br>

PartialProductGenerator


OPERATIONS

apply (setprod : Set<Product> ) : Set<Product> Public

This operator takes in input a set of total products (with all features assigned), and returns a set of products comprehending total products and partial products (not all features assigned). <br>This operator is similar to the Quine-McCluskey algorithm. <br> Given two products p1 p2 identical but for a feature f activated in one and deactivated in the other, a super product (a.k.a. sub-family) is generated such that f is left unresolved. <br>This method generates all possible super products. <br>All generated super products are such that the corresponding feature model formula is satisfied. <br>@return the set of all total and partial products.

Properties:

annotations = @Override

**Product**

*Class in package 'family'*

A configuration/product of a product line/family, identified as set of required and forbidden features.

<table>
<thead>
<tr>
<th>ATTRIBUTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>forbidden : Set&lt;Feature&gt;  Private  Const</td>
</tr>
<tr>
<td>the set of forbidden features</td>
</tr>
<tr>
<td>[ Is static True. Containment is Not Specified. ]</td>
</tr>
<tr>
<td>required : Set&lt;Feature&gt;  Private  Const</td>
</tr>
<tr>
<td>the set of required features</td>
</tr>
<tr>
<td>[ Is static True. Containment is Not Specified. ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ASSOCIATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Association (direction: Source -&gt; Destination)</td>
</tr>
<tr>
<td>Source: Public (Class) Product</td>
</tr>
<tr>
<td>Target: Public (Class) Feature</td>
</tr>
<tr>
<td>Association (direction: Source -&gt; Destination)</td>
</tr>
<tr>
<td>Source: Public (Class) Family</td>
</tr>
<tr>
<td>Target: Public (Class) Product</td>
</tr>
<tr>
<td>Association (direction: Source -&gt; Destination)</td>
</tr>
<tr>
<td>Source: Public (Class) ProductOrchestrationSynthesisOperator</td>
</tr>
<tr>
<td>Target: Private p (Class) Product</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPERATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>checkForbidden (tr : Set&lt;? extends ModalTransition&lt;String,Action,State&lt;String&gt;,CALabel&gt;&gt; ) : boolean Public</td>
</tr>
<tr>
<td>Returns true if all forbidden actions of this product are not available in the transitions tr, i.e, all features name are not equal to any of the content of the actions of the transitions in tr.</td>
</tr>
<tr>
<td>@return true if all forbidden actions are not available in the transitions tr.</td>
</tr>
<tr>
<td>checkRequired (tr : Set&lt;? extends ModalTransition&lt;S1, Action, State&lt;S1&gt;, CALabel&gt;&gt; ) : boolean Public</td>
</tr>
<tr>
<td>Returns true if all required actions are available in the transitions tr, i.e, all features name of this product are equal to the content of some action of some transition in tr.</td>
</tr>
<tr>
<td>@param &lt;S1&gt; the type of the content of the state.</td>
</tr>
<tr>
<td>@return true if all required actions are available in the transitions tr.</td>
</tr>
</tbody>
</table>

Properties:
## OPERATIONS

<table>
<thead>
<tr>
<th>Method</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>equals(obj : Object) : boolean Public</code></td>
<td></td>
<td>Overrides the method of the object class. Returns true if the two objects are equal.</td>
</tr>
<tr>
<td><code>getForbidden() : Set&lt;Feature&gt; Public</code></td>
<td></td>
<td>Getter of the set of forbidden features. Returns the set of forbidden features.</td>
</tr>
<tr>
<td><code>getForbiddenAndRequiredNumber() : int Public</code></td>
<td></td>
<td>Returns the number of forbidden and required features of this product. Returns the number of forbidden and required features of this product.</td>
</tr>
<tr>
<td><code>getRequired() : Set&lt;Feature&gt; Public</code></td>
<td></td>
<td>Getter of the set of required features. Returns the set of required features.</td>
</tr>
<tr>
<td><code>hashCode() : int Public</code></td>
<td></td>
<td>Overrides the method of the object class. Returns the hashcode of this object.</td>
</tr>
<tr>
<td><code>isForbidden(l : CALabel) : boolean Public</code></td>
<td></td>
<td>Returns true if the action of l is equal to some name of a forbidden feature.</td>
</tr>
<tr>
<td><code>isValid(aut : Automaton&lt;String,Action,State&lt;String&gt;,ModalTransition&lt;String,Action,State&lt;String&gt;,CALabel&gt;&gt; ) : boolean Public</code></td>
<td></td>
<td>Returns true if the set of transitions of aut satisfies this.checkForbidden and this.checkRequired.</td>
</tr>
</tbody>
</table>
### OPERATIONS

<table>
<thead>
<tr>
<th>Constructor for a product from sets of features</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Product (required: Set&lt;Feature&gt;, forbidden: Set&lt;Feature&gt;): Public</code></td>
</tr>
<tr>
<td>Returns a new product where the features in <code>sf</code> have been removed (from both required and forbidden features).</td>
</tr>
<tr>
<td>@return a new product where the features in <code>sf</code> have been removed (from both required and forbidden features).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constructor for a product from arrays of Strings</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Product (r: String[], f: String[]): Public</code></td>
</tr>
<tr>
<td>Print a representation of this object as String</td>
</tr>
<tr>
<td>@return a representation of this object as String</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>annotations = @Override</code></td>
</tr>
</tbody>
</table>
**operations**

This package groups the various operations that can be performed on automata. 
*Projection* 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*, to compose automata, and *Synthesis* to refine an automaton to satisfy given predicates. 
These two classes are generics. 
*MSCACompositionFunction* instantiates the generic types to those used by a modal contract automaton. 
*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 the *MpcSynthesisOperator, OrchestrationSynthesisOperator, and ChoreographySynthesisOperator* are derived. 
*ProductOrchestrationSynthesisOperator* further specialises the orchestration synthesis for a given configuration. 
These operations are formally specified in:


---

**Package in package 'catlib'**

Davide Basile (ISTI CNR Italy) created on 4/23/2022. Last modified 4/23/2022

**operations diagram**

*Class diagram in package 'operations'*

Davide Basile (ISTI CNR Italy) created on 4/23/2022. Last modified 4/23/2022
**interfaces**

*Package in package 'operations'*

Davide Basile (ISTI CNR Italy) created on 4/23/2022. Last modified 4/23/2022

**interfaces diagram**

*Class diagram in package 'interfaces'*

Davide Basile (ISTI CNR Italy) created on 4/23/2022. Last modified 4/23/2022

Figure 9: operations

Figure 10: interfaces

**TetraFunction**

*Interface in package 'interfaces'*
A function over four arguments.

@param <T>
@param <U>
@param <V>
@param <W>
@param <Z>

TetraFunction

Davide Basile (ISTI CNR Italy) created on 4/23/2022. Last modified 4/23/2022

ASSOCIATIONS

<table>
<thead>
<tr>
<th>Source: Public (Class) ModelCheckingSynthesisOperator</th>
<th>Target: Private createTransition (Interface) TetraFunction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Association (direction: Source -&gt; Destination)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Source: Public (Class) ModelCheckingSynthesisOperator</th>
<th>Target: Private createTransitionProp (Interface) TetraFunction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Association (direction: Source -&gt; Destination)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Source: Public (Class) CompositionFunction</th>
<th>Target: Private createTransition (Interface) TetraFunction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Association (direction: Source -&gt; Destination)</td>
<td></td>
</tr>
</tbody>
</table>

OPERATIONS


TriFunction

Interface in package 'interfaces'

A function over three arguments.
@param <T> first argument
@param <U> second argument
@param <V> third argument
@param <Z> returned class

TriFunction


INCOMING STRUCTURAL RELATIONSHIPS

Realization from ProjectionFunction to TriFunction
[ Direction is 'Source -> Destination'. ]
### OPERATIONS

<table>
<thead>
<tr>
<th>Operation</th>
<th>Signature</th>
<th>Accessibility</th>
<th>Is static</th>
<th>Is abstract</th>
<th>Is return array</th>
<th>Is query</th>
<th>Is synchronized</th>
</tr>
</thead>
<tbody>
<tr>
<td>apply</td>
<td>( \text{arg1} : T, \text{arg2} : U, \text{arg3} : V ) : Z Public</td>
<td>Public</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

**TriPredicate**

*Interface in package 'interfaces'*

A predicate over three arguments. * Used in the synthesis method of MSCA for readability.

- @param <T> generic type of the first argument
- @param <U> generic type of the second argument
- @param <V> generic type of the third argument

**ASSOCIATIONS**

- **Association (direction: Source -> Destination)**
  - Source: Public (Class) SynthesisOperator
  - Target: Private pruningPred (Interface) TriPredicate

- **Association (direction: Source -> Destination)**
  - Source: Public (Class) SynthesisOperator
  - Target: Private forbiddenPred (Interface) TriPredicate

**OPERATIONS**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Signature</th>
<th>Accessibility</th>
<th>Is static</th>
<th>Is abstract</th>
<th>Is return array</th>
<th>Is query</th>
<th>Is synchronized</th>
</tr>
</thead>
<tbody>
<tr>
<td>test</td>
<td>( \text{arg1} : T, \text{arg2} : U, \text{arg3} : V ) : boolean Public</td>
<td>Public</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

**ChoreographySynthesisOperator**

*Class in package 'operations'*

Class implementing the Choreography Synthesis. The implemented algorithm is formally specified in Definition 4.4 and Theorem 5.5 of


- @param <S1> the type of the content of states

**OUTGOING STRUCTURAL RELATIONSHIPS**

OUTGOING STRUCTURAL RELATIONSHIPS

Generalization from ChoreographySynthesisOperator to ModelCheckingSynthesisOperator
[ Direction is ‘Source -> Destination’. ]

ATTRIBUTES

choice : Function<Stream<ModalTransition<S1,Action,State<S1>,CALabel>>,Optional<ModalTransition<S1,Action,State<S1>,CALabel>>> Private = Stream::findAny
[ Is static True. Containment is Not Specified. ]

req : Predicate<CALabel> Private Const
[ Is static True. Containment is Not Specified. ]

OPERATIONS

apply (arg : Automaton<S1,Action,State<S1>,ModalTransition<S1,Action,State<S1>,CALabel>> ) : Automaton<S1,Action,State<S1>,ModalTransition<S1,Action,State<S1>,CALabel>> Public
Applies the choreography synthesis operator to aut
@return the synthesised choreography, removing only one transition violating the branching condition each time no further updates are possible. The transition to remove is chosen non-deterministically in case a specific strategy was not provided in the constructor.

Properties:
annotations = @Override

ChoreographySynthesisOperator (req : Predicate<CALabel> ) : Public
Constructor for the choreography synthesis operator enforcing the requirement req.

ChoreographySynthesisOperator (req : Predicate<CALabel> , prop : Automaton<S1,Action,State<S1>,ModalTransition<S1,Action,State<S1>,Label<Action>>> ) : Public
Constructor for the choreography synthesis operator enforcing the requirement req and property prop.

ChoreographySynthesisOperator (req : Predicate<CALabel> , choice : Function<Stream<ModalTransition<S1,Action,State<S1>,CALabel>>,Optional<ModalTransition<S1,Action,State<S1>,CALabel>>> ) : Public
Constructor for the choreography synthesis operator enforcing the requirement req. <br> This constructor also takes in input a strategy for resolving the choice when pruning a transition not satisfying the branching condition.

isUncontrollableChoreography (tra : ModalTransition<S1,Action,State<S1>,CALabel> , str : Set<? extends ModalTransition<S1,Action,State<S1>,CALabel>> , badStates : Set<State<S1>> ) : boolean Private
Properties:
OPERATIONS

| generic = <S1> |


diamond satisfiesBranchingCondition (tra : ModalTransition<S1,Action,State<S1>,CALabel> , trans : 
Set<ModalTransition<S1,Action,State<S1>,CALabel>>> , bad : Set<State<S1>> ) : boolean Public

Return true if the set of transitions and bad states violate the branching condition. The requirements for ensuring that the synthesised automaton is a (form of) choreography roughly amount to the so-called branching condition requiring that principals perform their offers/outputs independently of the other principals in the composition. See Definition 4.1 in


@return true if the set of transitions and bad states violate the branching condition


CompositionFunction

Class in package 'operations'

Class implementing the Composition Function. The composition function supports an on-the-fly, bounded composition. It is possible to invoke a composition stopping at a given depth, and invoking again at a greater depth. In this case when reapplying the function, the previous states are stored and will not be generated again. This composition is a special type of synchronous product where synchronizations (called matches) are not broadcast, i.e., they only involve two principals. The arguments of the composition may be automata having rank greater than 1, i.e., representing an ensemble of composed automata. In this case, pre-existing matches inside the operands automata are preserved and are not rearranged. By changing the order in which principal automata are composed, different results can be obtained, in other words, this composition is non-associative. The associative composition is a special case where all operands are of rank 1. The formal definition of this composition is specified in Definition 5 of


@param <S1> the generic type of the content of states
@param <S> the generic type of states, must be a subtype of <code>State<S1></code>
@param <L> the generic type of the labels, must be a subtype of <code>Label<Action></code>
@param <T> the generic type of a transitions, must be a subtype of <code>ModalTransition<S1,Action,S,L></code>
@param <A> the generic type of the automata, must be a subtype of <code>Automaton<S1,Action,S,T ></code>

CompositionFunction


ELEMENTS OWNED BY CompositionFunction

- TIndex : Class

INCOMING STRUCTURAL RELATIONSHIPS

- Generalization from MSCACCompositionFunction to CompositionFunction

[ Direction is 'Source -> Destination'. ]
### INCOMING STRUCTURAL RELATIONSHIPS

- Generalization from ModelCheckingFunction to CompositionFunction  
  [ Direction is 'Source -> Destination'. ]

### ATTRIBUTES

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Visibility</th>
<th>Static</th>
<th>Containment</th>
</tr>
</thead>
<tbody>
<tr>
<td>aut : List&lt;? extends Automaton&lt;S1,Action,S,T&gt;&gt;</td>
<td>Private Const</td>
<td></td>
<td>True</td>
<td>Not Specified</td>
</tr>
<tr>
<td>createAutomaton : Function&lt;Set&lt;T&gt;,A&gt;</td>
<td>Private Const</td>
<td></td>
<td>True</td>
<td>Not Specified</td>
</tr>
<tr>
<td>createLabel : Function&lt;List&lt;Action&gt;,L&gt;</td>
<td>Private Const</td>
<td></td>
<td>True</td>
<td>Not Specified</td>
</tr>
<tr>
<td>createState : Function&lt;List&lt;BasicState&lt;S1&gt;&gt;,S&gt;</td>
<td>Private Const</td>
<td></td>
<td>True</td>
<td>Not Specified</td>
</tr>
<tr>
<td>createTransition : TetraFunction&lt;S,L,S,ModalTransition.Modality, T&gt;</td>
<td>Private Const</td>
<td></td>
<td>True</td>
<td>Not Specified</td>
</tr>
<tr>
<td>dontvisit : Queue&lt;S&gt;</td>
<td>Private Const</td>
<td></td>
<td>True</td>
<td>Not Specified</td>
</tr>
<tr>
<td>frontier : Queue&lt;Entry&lt;List&lt;S&gt;,Integer&gt;&gt;</td>
<td>Private Const</td>
<td></td>
<td>True</td>
<td>Not Specified</td>
</tr>
<tr>
<td>initialState : S</td>
<td>Private Const</td>
<td></td>
<td>True</td>
<td>Not Specified</td>
</tr>
<tr>
<td>match : BiPredicate&lt;L,L&gt;</td>
<td>Private Const</td>
<td></td>
<td>True</td>
<td>Not Specified</td>
</tr>
<tr>
<td>operandstat2compstat : ConcurrentHashMap&lt;List&lt;S&gt;, S&gt;</td>
<td>Private Const</td>
<td></td>
<td>True</td>
<td>Not Specified</td>
</tr>
<tr>
<td>pruningPred : Predicate&lt;T&gt;</td>
<td>Private Const</td>
<td></td>
<td>True</td>
<td>Not Specified</td>
</tr>
<tr>
<td>rank : int</td>
<td>Private Const</td>
<td></td>
<td>True</td>
<td>Not Specified</td>
</tr>
<tr>
<td>toVisit : Queue&lt;Entry&lt;List&lt;S&gt;,Integer&gt;&gt;</td>
<td>Private Const</td>
<td></td>
<td>True</td>
<td>Not Specified</td>
</tr>
</tbody>
</table>
ATTRIBUTES

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Access</th>
<th>Modifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>tr</td>
<td>Set&lt;T&gt;</td>
<td>Private</td>
<td>Const</td>
</tr>
<tr>
<td>visited</td>
<td>Set&lt;List&lt;S&gt;&gt;</td>
<td>Private</td>
<td>Const</td>
</tr>
</tbody>
</table>

ASSOCIATIONS

Association (direction: Source -> Destination)

Source: Public (Class) CompositionFunction
Target: Private createTransition (Interface)
TetraFunction

OPERATIONS

apply (bound : int) : A Public

This is one of the main functionalities of the library. It applies the composition function to compute the non-associative composition.

@return the composed automaton

Properties:

   annotations = @Override


Constructor for a composition function.


computeComposedForwardStar (trans2index : List<TIndex>, source : List<S>, sourcestate : S) :
Set<SimpleEntry<T,List<S>>> Private


createLabel (e1 : TIndex, e2 : TIndex) : L Private


flattenState (lstate : List<S>) : List<BasicState<S1>> Private


getPruningPred () : Predicate<T> Public

Getter of the pruning predicate.

@return the pruning predicate.
OPERATIONS

```java
isFrontierEmpty () : boolean Public

Returns true if no states are left to be generated, i.e., the whole depth of the composition has been generated. If it returns false, this composition can be reapplied with a major depth to produce a composition with the frontier further pushed onwards. When invoking again the composition, the previous information is stored to avoid recomputing the previously generated states.
```

```java
shiftLabel (lab : L , rank : Integer , shift : Integer ) : L Private

```

TIndex

**Class owned by 'CompositionFunction', in package 'operations'**

each transition of each MSCA in aut is associated with the corresponding index in aut

```java
TIndex

Davide Basile (ISTI CNR Italy) created on 4/23/2022. Last modified 4/23/2022
```

ATTRIBUTES

```java
ind : Integer Package Const

[ Is static True. Containment is Not Specified. ]
```

```java
tra : T Package Const

more readable than Entry

[ Is static True. Containment is Not Specified. ]
```

OPERATIONS

```java
TIndex (tr : T , i : Integer ) : Package

```

**ModelCheckingFunction**

**Class in package 'operations'**

Class implementing the Model Checking Function. This is implemented by instantiating the `CompositionFunction` to the case where two automata are composed: the first is a contract automaton, whilst the second is a generic automaton describing a property. The output is a synchronous product between the contract automaton and the property.

@param `<S1>` the generic type of the content of states
@param `<S>` the generic type of states, must be a subtype of `<State<S1>`
@param <L> the generic type of the labels, must be a subtype of <code>Label<Action></code>
@param <T> the generic type of a transitions, must be a subtype of <code>ModalTransition<S1,Action,S,L></code>
@param <A> the generic type of the automata, must be a subtype of <code>Automaton<S1,Action,S,T></code>

ModelCheckingFunction


OUTGOING STRUCTURAL RELATIONSHIPS

⇐ Generalization from ModelCheckingFunction to CompositionFunction
[ Direction is 'Source -> Destination'. ]

OPERATIONS


The constructor of a model checking function. <br> The match function of <code>CompositionFunction</code> is instantiated to match two labels with the same action content (in the style of a synchronous product). <br> The pruning predicate of <code>CompositionFunction</code> is instantiated to prune labels of transitions where the automaton is not synchronizing with the property (and vice-versa). <br> The rank of the property must be 1. <br>

ModelCheckingSynthesisOperator

Class in package 'operations'

This class implements a model checking operation followed by a synthesis operation. <br> In case the property to model-check is not given, the synthesis operation is applied straightforward. <br> Otherwise, the synthesis operation is applied on the result of the application of the model checking function. <br>
@param <S1> the generic type of the content of states
@param <S> the generic type of states, must be a subtype of <code>State<S1></code>
@param <L> the generic type of the labels of the automaton to check, must be a subtype of <code>L2</code>
@param <T> the generic type of the transitions of the automaton to check, must be a subtype of <code>ModalTransition<S1,Action,S,L></code>
@param <A> the generic type of the automaton to check, must be a subtype of <code>Automaton<S1,Action,S,T></code>
@param <L2> the generic type of the labels of the property, must be a subtype of <code>Label<Action></code>
@param <T2> the generic type of the transitions of the property, must be a subtype of <code>ModalTransition<S1,Action,S,L2></code>
@param <A2> the generic type of the automaton property, must be a subtype of <code>Automaton<S1,Action,S,T2></code>

ModelCheckingSynthesisOperator

Davide Basile (ISTI CNR Italy) created on 4/23/2022. Last modified 4/29/2022

OUTGOING STRUCTURAL RELATIONSHIPS

⇐ Generalization from ModelCheckingSynthesisOperator to SynthesisOperator
[ Direction is 'Source -> Destination'. ]

INCOMING STRUCTURAL RELATIONSHIPS
### INCOMING STRUCTURAL RELATIONSHIPS

1. **Generalization** from `MpcSynthesisOperator` to `ModelCheckingSynthesisOperator`  
   - Direction is 'Source -> Destination'.

2. **Generalization** from `OrchestrationSynthesisOperator` to `ModelCheckingSynthesisOperator`  
   - Direction is 'Source -> Destination'.

3. **Generalization** from `ChoreographySynthesisOperator` to `ModelCheckingSynthesisOperator`  
   - Direction is 'Source -> Destination'.

### ATTRIBUTES

- **changeLabel** : `UnaryOperator<L>` Private Const  
  - Is static True. Containment is Not Specified.

- **createAutomatonProp** : `Function<Set<T2>,A2>` Private Const  
  - Is static True. Containment is Not Specified.

- **createLabel** : `Function<List<Action>,L>` Private Const  
  - Is static True. Containment is Not Specified.

- **createLabelProp** : `Function<List<Action>,L2>` Private Const  
  - Is static True. Containment is Not Specified.

- **createState** : `Function<List<BasicState<S1>>,S>` Private Const  
  - Is static True. Containment is Not Specified.

- **createTransition** : `TetraFunction<S,L,S,ModalTransition.Modality, T>` Private Const  
  - Is static True. Containment is Not Specified.

- **createTransitionProp** : `TetraFunction<S,L2,S,ModalTransition.Modality, T2>` Private Const  
  - Is static True. Containment is Not Specified.

- **prop** : `A2` Private Const  
  - Is static True. Containment is Not Specified.

### ASSOCIATIONS

1. **Association** (direction: Source -> Destination)  
   - **Source**: Public (Class) `ModelCheckingSynthesisOperator`  
   - **Target**: Private `createTransition` (Interface) `TetraFunction`
ASSOCIATIONS

| Source: Public (Class) ModelCheckingSynthesisOperator | Target: Private createTransitionProp (Interface) TetraFunction |

OPERATIONS

apply (arg1 : A ): A Public

Applies the model checking and synthesis operator.
@return the automaton resulting from applying model checking and synthesis to arg

Properties:
annotations = @Override

getChangeLabel () : UnaryOperator<L> Public

Getter of the function changeLabel.
@return the function changeLabel


Constructor for a model checking synthesis operator, it requires also the constructors for the used generic types.


Constructor for a model checking synthesis operator, it requires also the constructors for the used generic types. In this constructor the pruning predicate is set to always return false.


Constructor for a model checking synthesis operator. This constructor sets to null the property and the related constructors.

MpcSynthesisOperator

Class in package 'operations'
Class implementing the most permissive controller synthesis operator. The implemented algorithm is formally specified in Definition 2.3 and Theorem 5.3 of


@param <S1> the type of the content of states

### MpcSynthesisOperator


<table>
<thead>
<tr>
<th>OUTGOING STRUCTURAL RELATIONSHIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generalization from MpcSynthesisOperator to ModelCheckingSynthesisOperator</td>
</tr>
<tr>
<td>[ Direction is 'Source -&gt; Destination'. ]</td>
</tr>
</tbody>
</table>

### OPERATIONS

- **apply** (aut : Automaton<S1,Action,State<S1>,ModalTransition<S1,Action,State<S1>,CALabel>> ) : Automaton<S1,Action,State<S1>,ModalTransition<S1,Action,State<S1>,CALabel>>
  
  Applies the mpc synthesis to aut. The argument must not contain lazy transitions.
  
  @return the synthesised most permissive controller

  Properties:
  
  annotations = @Override
  

- **MpcSynthesisOperator** (req : Predicate<CALabel> ) : Public

  Constructor for the mpc synthesis enforcing the requirement req.


- **MpcSynthesisOperator** (req : Predicate<CALabel> , prop : Automaton<S1,Action,State<S1>,ModalTransition<S1,Action,State<S1>,Label<Action>>> ) : Public

  Constructor for the mpc synthesis enforcing the requirement req and property prop.


### MSCACCompositionFunction

*Class in package 'operations'*

Class implementing the composition of Contract Automata. This class is auxiliary and is used to instantiate the generic types of CompositionFunction, where labels are objects of type CALabel and transitions are objects of type ModalTransition. @param <S1> the generic type of the content of states.

MSCACCompositionFunction

### OUTGOING STRUCTURAL RELATIONSHIPS

Generalization from MSCACompositionFunction to CompositionFunction

[Direction is 'Source -> Destination'.]

### OPERATIONS

```java
MSCACompositionFunction (aut : List<Automaton<S1,Action,State<S1>,ModalTransition<S1,Action,State<S1>,CALabel>>> , pruningPred : Predicate<ModalTransition<S1,Action,State<S1>,CALabel>>> ) : Public

Invokes the constructor of the superclass instantiating the generic types

```

### OrchestrationSynthesisOperator

*Class in package 'operations'*

Class implementing the orchestration synthesis operator.<br> The implemented algorithm is formally specified in Definition 3.2 and Theorem 5.4 of


@param <S1> the type of the content of states

OrchestrationSynthesisOperator


### OUTGOING STRUCTURAL RELATIONSHIPS

Generalization from OrchestrationSynthesisOperator to ModelCheckingSynthesisOperator

[Direction is 'Source -> Destination'.]

### INCOMING STRUCTURAL RELATIONSHIPS

Generalization from ProductOrchestrationSynthesisOperator to OrchestrationSynthesisOperator

[Direction is 'Source -> Destination'.]

### OPERATIONS

```java
apply (aut : Automaton<S1,Action,State<S1>,ModalTransition<S1,Action,State<S1>,CALabel>>> , Automaton<S1,Action,State<S1>,ModalTransition<S1,Action,State<S1>,CALabel>>> ) : Public

Applies the orchestration synthesis to aut. The argument must not contain necessary offers.

@return the synthesised orchestration.
```

Properties:

- annotations = @Override


isUncontrollableOrchestration (tra : ModalTransition<S1,Action,State<S1>,CALabel> , str : Set<? extends ModalTransition<S1,Action,State<S1>,CALabel>>> , badStates : Set<State<S1>>> ) : boolean

Properties:
  generic = <S1>

OrchestrationSynthesisOperator (req : Predicate<CALabel> ) : Public

Constructor for the orchestration synthesis operator enforcing the requirement req.

OrchestrationSynthesisOperator (req : Predicate<CALabel> , prop : Automaton<S1,Action,State<S1>,ModalTransition<S1,Action,State<S1>,Label<Action>>> ) : Public

Constructor for the orchestration synthesis operator enforcing the requirement req and property prop.

ProductOrchestrationSynthesisOperator

Class in package 'operations'

Class implementing the orchestration synthesis for a specific product of a product line. This is a further specialization of the orchestration synthesis where the requirement also checks that an action must not be forbidden by the product, and in the resulting synthesised automaton all required actions must be reachable (otherwise an empty orchestration is returned). This operation is formally specified in Definition 14 of Basile, D. et al., 2020. Controller synthesis of service contracts with variability. Science of Computer Programming, vol. 187, pp. 102344. (https://doi.org/10.1016/j.scico.2019.102344)

@param <S1> the type of the content of states

ProductOrchestrationSynthesisOperator


OUTGOING STRUCTURAL RELATIONSHIPS

Generalization from ProductOrchestrationSynthesisOperator to OrchestrationSynthesisOperator

[ Direction is 'Source -> Destination'. ]

ATTRIBUTES

p : Product Private Const

[ Is static True. Containment is Not Specified. ]

ASSOCIATIONS

Association (direction: Source -> Destination)

Source: Public (Class) ProductOrchestrationSynthesisOperator

Target: Private p (Class) Product
## OPERATIONS

```java
apply (aut : Automaton<S1,Action,State<S1>,ModalTransition<S1,Action,State<S1>,CALabel>>, indexprincipal : Integer, getNecessaryPrincipal : ToIntFunction<ModalTransition<S1,Action,State<S1>,CALabel>>> ) : Automaton<S1,Action,State<S1>,ModalTransition<S1,Action,State<S1>,CALabel>> Public

Apply the product orchestration synthesis operator to aut.
@return the synthesised orchestration of product p

Properties:
 annotations = @Override
```

```java
ProductOrchestrationSynthesisOperator (req : Predicate<CALabel>, p : Product) : Public

The constructor for the product orchestration synthesis operator.
```

---

### ProjectionFunction

**Class in package 'operations'**

Class implementing the projection function. <br> This function takes as arguments an automaton (of rank greater than 1) and an index, and returns the principal automaton (of rank 1) at position index. <br> The projected automaton can store information, if needed, on the principals it was interacting with in the composition. <br> In this case, the projected actions are addressed actions. <br> The projection function is formally defined in Definition 5 of

@param <S1> the generic type of the content of states.


---

### OUTGOING STRUCTURAL RELATIONSHIPS

| Realization from ProjectionFunction to TriFunction |
| [ Direction is 'Source -> Destination'. ] |

---

### ATTRIBUTES

- createAddress : boolean Private Const

[ Is static True. Containment is Not Specified. ]

---

### OPERATIONS

```java
apply (aut : Automaton<S1,Action,State<S1>,ModalTransition<S1,Action,State<S1>,CALabel>>, indexprincipal : Integer, getNecessaryPrincipal : ToIntFunction<ModalTransition<S1,Action,State<S1>,CALabel>>> ) : Automaton<S1,Action,State<S1>,ModalTransition<S1,Action,State<S1>,CALabel>> Public

Apply the projection function.
@return the projected i-th principal automaton.
```
OPERATIONS

Properties:

```java
   .annotations = @Override
```

```java
    createLabel (t : ModalTransition<S1,Action,State<S1>,CALabel> , indexprincipal : Integer ) : CALabel Private
```

```java
    ProjectionFunction (createAddress : boolean ) : Public
    Constructor for a projection function.
```

```java
    ProjectionFunction () : Public
    Constructor of a projection function. As default, no addressed actions are generated.
```

RelabelingOperator

Class in package 'operations'

Class implementing the relabeling operator.  <br> This operator can update the labels of states of an automaton, as well as initial and final states.  <br> As a side effect, in case the relabeling is the identity function, a clone of an automaton is created.  <br>

@param <S1> the generic type content of the states
@param <L> the generic type of the label, constrained to be a sub-type of Label<Action>


ATTRIBUTES

```java
    createLabel : Function<List<Action>,L> Private Const
    [ Is static True. Containment is Not Specified. ]
```

```java
    finalStatePred : Predicate<BasicState<S1>> Private Const
    [ Is static True. Containment is Not Specified. ]
```

```java
    initialStatePred : Predicate<BasicState<S1>> Private Const
    [ Is static True. Containment is Not Specified. ]
```

```java
    relabel : UnaryOperator<S1> Private Const
    [ Is static True. Containment is Not Specified. ]
```

OPERATIONS
**OPERATIONS**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>apply (aut : Automaton &lt;S1,Action,State&lt;S1&gt;,ModalTransition&lt;S1,Action,State&lt;S1&gt;,L&gt;&gt; ) : Set&lt;ModalTransition&lt;S1, Action, State&lt;S1&gt;, L&gt;&gt;</td>
<td>Public</td>
</tr>
</tbody>
</table>

This method applies the relabeling operator.

```
@return the relabeled automaton
```

Properties:
- annotations = @Override

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RelabelingOperator (createLabel : Function&lt;List&lt;Action&gt;,L&gt; , relabel : UnaryOperator&lt;S1&gt; , initialStatePred : Predicate&lt;BasicState&lt;S1&gt; &gt; , finalStatePred : Predicate&lt;BasicState&lt;S1&gt;&gt; ) :</td>
<td>Public</td>
</tr>
</tbody>
</table>

Constructor for the relabeling operator.

```
```

---

**SynthesisOperator**

*Class in package 'operations'*

The synthesis operation is an automatic refinement of an automaton to a refined one where the automata are equipped with two specific conditions: agreement and strong agreement, detailed in the package requirements. The synthesis must also take into account when an action is controllable or uncontrollable.

The synthesis is an iterative procedure that at each step i updates incrementally a set of states $R_i$ containing the bad states, i.e., those states that cannot prevent a forbidden state to be eventually reached, and refines an automaton $K_i$. The algorithm starts with an automaton $K_0$ equal to $A$ and a set $R_0$ containing all dangling states in $A$, where a state is dangling if it cannot be reached from the initial state $s$ or cannot reach a final state. At each step i, the algorithm prunes from $K_{i−1}$ in a backwards fashion transitions with target state in $R_{i−1}$ or forbidden source state. The set $R_i$ is obtained by adding to $R_{i−1}$ dangling states in $K_i$ and source states of uncontrollable transitions of $A$ with target state in $R_{i−1}$. When no more updates are possible, the algorithm terminates. Termination is ensured since $A$ is finite-state and has a finite set of transitions, and at each step, the subsets of its states $R_i$ cannot decrease while the set of its transitions $T_{K_i}$ cannot increase. At its termination the algorithm returns the pair $(K_s, R_s)$. We have that the result is empty, if the initial state of $A$ is in $R_s$; otherwise, the result is $K_s$ obtained from $K_s$ by removing the states $R_s$.

The abstract synthesis operations generalises the other synthesis operations by abstracting away the conditions under which a transition is pruned or a state is deemed bad, thus encapsulating and extrapolating the notion of controllability and safety. These two conditions, called pruning predicate ($\downarrow\phi_p$) and forbidden predicate ($\downarrow\phi_f$), are parameters to be instantiated by the corresponding instance of the synthesis algorithm (e.g., orchestration or choreography). Predicate $\downarrow\phi_p$ is used for selecting the transitions to be pruned. Depending on the specific instance, non-local information about the automaton or the set of bad states is needed by $\downarrow\phi_p$. Therefore, $\downarrow\phi_p$ takes as input the current transition and the automaton, and the set of bad states. If $\downarrow\phi_p$ evaluates to true, then the corresponding transition will be pruned. Predicate $\downarrow\phi_f$ is used for deciding whether a state becomes bad. The input parameters are the same as $\downarrow\phi_p$. However, $\downarrow\phi_f$ only inspects necessary transitions. If $\downarrow\phi_f$ evaluates to true, then the source state is deemed bad and added to the set of bad states.

The formal definition is given in Definition 5.1 of:


---

SynthesisOperator

# INCOMING STRUCTURAL RELATIONSHIPS

- Generalization from ModelCheckingSynthesisOperator to SynthesisOperator

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>createAut : Function&lt;Set&lt;T&gt;,A&gt;</td>
<td>Private Const [ Is static True. Containment is Not Specified. ]</td>
</tr>
<tr>
<td>forbiddenPred : TriPredicate&lt;T, Set&lt;T&gt;, Set&lt;S&gt;&gt;</td>
<td>Private Const [ Is static True. Containment is Not Specified. ]</td>
</tr>
<tr>
<td>pruningPred : TriPredicate&lt;T, Set&lt;T&gt;, Set&lt;S&gt;&gt;</td>
<td>Private Const [ Is static True. Containment is Not Specified. ]</td>
</tr>
<tr>
<td>reachable : Map&lt;S,Boolean&gt;</td>
<td>Private [ Is static True. Containment is Not Specified. ]</td>
</tr>
<tr>
<td>req : Predicate&lt;L&gt;</td>
<td>Private Const [ Is static True. Containment is Not Specified. ]</td>
</tr>
<tr>
<td>successful : Map&lt;S,Boolean&gt;</td>
<td>Private [ Is static True. Containment is Not Specified. ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Association</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Association (direction: Source -&gt; Destination)</td>
<td>Source: Public (Class) SynthesisOperator Target: Private pruningPred (Interface) TriPredicate</td>
</tr>
<tr>
<td>Association (direction: Source -&gt; Destination)</td>
<td>Source: Public (Class) SynthesisOperator Target: Private forbiddenPred (Interface) TriPredicate</td>
</tr>
</tbody>
</table>

## OPERATIONS

- apply (aut : A ) : A Public

This method applies the synthesis operator to aut.

@return the synthesised automaton.

Properties:

```plaintext
annotations = @Override
```

### OPERATIONS

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>getCreateAut</strong> () : Function&lt;Set&lt;T&gt;, A&gt; Public</td>
<td>Getter of the function for creating an automaton. @return the function for creating an automaton. [Is static False. Is abstract False. Is return array False. Is query False. Is synchronized False. ]</td>
</tr>
<tr>
<td><strong>getDanglingStates</strong> (tr : Set&lt;T&gt;, states : Set&lt;S&gt;, initial : S) : Set&lt;S&gt; Private</td>
<td>@return states who do not reach a final state or are unreachable [Is static False. Is abstract False. Is return array False. Is query False. Is synchronized False. ]</td>
</tr>
<tr>
<td><strong>getReq</strong> () : Predicate&lt;L&gt; Public</td>
<td>Getter of the requirement. @return the requirement. [Is static False. Is abstract False. Is return array False. Is query False. Is synchronized False. ]</td>
</tr>
</tbody>
</table>

### UnionFunction

*Class in package 'operations'*

Class implementing the union function. <br>This is the standard union operation of Finite State Automata, obtained by adding a new initial state with outgoing transitions to the initial states of the operands. <br>These new transitions have a dummy label. <br>Before being unified, the automata are relabeled to avoid having duplicate states. <br>

UnionFunction


### OPERATIONS
<table>
<thead>
<tr>
<th>OPERATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>apply (aut : List&lt;Automaton&lt;String,Action,State&lt;String&gt;,ModalTransition&lt;String,Action,State&lt;String&gt;,CALabel&gt;&gt;&gt; ) : Automaton&lt;String,Action,State&lt;String&gt;,ModalTransition&lt;String,Action,State&lt;String&gt;,CALabel&gt;&gt; Public</td>
</tr>
</tbody>
</table>

Compute the union function.
@return the automaton union of the automata in aut

Properties:
   annotations = @Override

requirements

This package groups the invariant requirements that can be enforced in a contract automaton. The Agreement requirement 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 is an invariant allowing only matches. This means that all requests and offers actions of principals are matched.

Package in package 'catlib'

requirements

Davide Basile (ISTI CNR Italy) created on 4/23/2022. Last modified 4/23/2022

requirements diagram

Class diagram in package 'requirements'

requirements

Version 1.0

Davide Basile (ISTI CNR Italy) created on 4/23/2022. Last modified 4/29/2022

Figure 11: requirements

Agreement

Class in package 'requirements'

The predicate of Agreement over CALabels. It holds if a CALabel is not a request.

Agreement


<table>
<thead>
<tr>
<th>OPERATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>test (l : CALabel) : boolean Public</td>
</tr>
<tr>
<td>Returns true if l is not a request.</td>
</tr>
<tr>
<td>@return true if l is not a request</td>
</tr>
<tr>
<td>Properties:</td>
</tr>
<tr>
<td>annotations = @Override</td>
</tr>
</tbody>
</table>

StrongAgreement
Class in package 'requirements'

The predicate of Strong Agreement over CALabels. Strong agreement holds if the label is a match.

```java
StrongAgreement

```

<table>
<thead>
<tr>
<th>OPERATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>test (l : CALabel) : boolean Public</td>
</tr>
<tr>
<td>Returns true if l is a match.</td>
</tr>
<tr>
<td>@return true if l is a match.</td>
</tr>
<tr>
<td>Properties:</td>
</tr>
<tr>
<td>annotations = @Override</td>
</tr>
</tbody>
</table>