GSV evaluator library

0.1

Generated by Doxygen 1.13.2

1 Namespace Index 1

1.1 Namespace List 1

2 Class Index 3

2.1 Class List 3

3 File Index 5

3.1 File List 5

4 Namespace Documentation 7

4.1 iif_sadaf Namespace Reference 7

4.2 iif_sadaf::talk Namespace Reference 7

4.3 iif_sadaf::talk::GSV Namespace Reference 7

4.3.1 Typedef Documentation 8

4.3.1.1 InformationState 8

4.3.2 Function Documentation 8

4.3.2.1 create() 8

4.3.2.2 extends() [1/3] 9

4.3.2.3 extends() [2/3] 9

4.3.2.4 extends() [3/3] 10

4.3.2.5 isDescendantOf() 11

4.3.2.6 operator<() 11

4.3.2.7 str() [1/3] 11

4.3.2.8 str() [2/3] 12

4.3.2.9 str() [3/3] 12

4.3.2.10 subsistsIn() [1/2] 12

4.3.2.11 subsistsIn() [2/2] 12

4.3.2.12 update() 13

5 Class Documentation 15

5.1 iif_sadaf::talk::GSV::Evaluator Struct Reference 15

5.1.1 Detailed Description 15

5.1.2 Member Function Documentation 16

5.1.2.1 operator()() [1/5] 16

5.1.2.2 operator()() [2/5] 16

5.1.2.3 operator()() [3/5] 17

5.1.2.4 operator()() [4/5] 17

5.1.2.5 operator()() [5/5] 18

5.2 iif_sadaf::talk::GSV::IModel Struct Reference 18

5.2.1 Detailed Description 19

5.2.2 Constructor & Destructor Documentation 19

5.2.2.1 ∼IModel() 19

5.2.3 Member Function Documentation 19

5.2.3.1 domain_cardinality() 19

5.2.3.2 predicateInterpretation() 19
5.2.3.3 termInterpretation() 19
5.2.3.4 world_cardinality() 20
5.3 iif_sadaf::talk::GSV::Possibility Struct Reference 20
5.3.1 Detailed Description 20
5.3.2 Constructor & Destructor Documentation 20
5.3.2.1 Possibility() [1/3] 20
5.3.2.2 Possibility() [2/3] 21
5.3.2.3 Possibility() [3/3] 21
5.3.3 Member Function Documentation 21
5.3.3.1 operator=() [1/2] 21
5.3.3.2 operator=() [2/2] 21
5.3.3.3 update() 21
5.3.4 Member Data Documentation 22
5.3.4.1 assignment 22
5.3.4.2 referentSystem 22
5.3.4.3 world 22
5.4 iif_sadaf::talk::GSV::ReferentSystem Struct Reference 22
5.4.1 Detailed Description 23
5.4.2 Constructor & Destructor Documentation 23
5.4.2.1 ReferentSystem() [1/3] 23
5.4.2.2 ReferentSystem() [2/3] 23
5.4.2.3 ReferentSystem() [3/3] 23
5.4.3 Member Function Documentation 23
5.4.3.1 operator=() [1/2] 23
5.4.3.2 operator=() [2/2] 23
5.4.3.3 value() 23
5.4.4 Member Data Documentation 24
5.4.4.1 pegs 24
5.4.4.2 variablePegAssociation 24
6 File Documentation **25**
6.1 evaluator.hpp File Reference 25
6.2 evaluator.hpp 25
6.3 imodel.hpp File Reference 26
6.4 imodel.hpp 26
6.5 information_state.hpp File Reference 26
6.6 information_state.hpp 27
6.7 possibility.hpp File Reference 27
6.8 possibility.hpp 28
6.9 referent_system.hpp File Reference 28
6.10 referent_system.hpp 29

6.11 evaluator.cpp File Reference 29
6.12 evaluator.cpp 30
6.13 information_state.cpp File Reference 33
6.14 information_state.cpp 33
6.15 possibility.cpp File Reference 34
6.16 possibility.cpp 35
6.17 referent_system.cpp File Reference 36
6.18 referent_system.cpp 37

Index **39**

Chapter 1

Namespace Index

1.1 Namespace List

Here is a list of all namespaces with brief descriptions:

iif_sadaf 7
iif_sadaf::talk 7
iif_sadaf::talk::GSV 7

Chapter 2

Class Index

2.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

iif_sadaf::talk::GSV::Evaluator
Represents an evaluator for logical expressions 15

iif_sadaf::talk::GSV::IModel
Interface for class representing a model for QML without accessiblity 18

iif_sadaf::talk::GSV::Possibility
Represents a possibility as understood in the underlying semantics 20

iif_sadaf::talk::GSV::ReferentSystem
Represents a referent system for variable assignments 22

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

evaluator.hpp 25
imodel.hpp 26
information_state.hpp 26
possibility.hpp 27
referent_system.hpp 28
evaluator.cpp 29
information_state.cpp 33
possibility.cpp 34
referent_system.cpp 36

Chapter 4

Namespace Documentation

4.1 iif_sadaf Namespace Reference

Namespaces

- namespace talk

4.2 iif_sadaf::talk Namespace Reference

Namespaces

- namespace GSV

4.3 iif_sadaf::talk::GSV Namespace Reference

Classes

- struct Evaluator

  Represents an evaluator for logical expressions.
- struct IModel

  Interface for class representing a model for QML without accessiblity.
- struct Possibility

  Represents a possibility as understood in the underlying semantics.
- struct ReferentSystem

  Represents a referent system for variable assignments.

Typedefs

- using InformationState = std::set<Possibility>

  An alias for `std::set<Possibility>`

Functions

- InformationState create (const Model &model)

  Creates an information state based on a model.
- InformationState update (const InformationState &input_state, std::string_view variable, int individual)

  Updates the information state with a new variable-individual assignment.
- bool extends (const InformationState &s2, const InformationState &s1)

  Determines if one information state extends another.
- bool isDescendantOf (const Possibility &p2, const Possibility &p1, const InformationState &s)

  Determines if one possibility is a descendant of another within an information state.
- bool subsistsIn (const Possibility &p, const InformationState &s)

  Checks if a possibility subsists in an information state.
- bool subsistsIn (const InformationState &s1, const InformationState &s2)

  Checks if an information state subsists within another.
- std::string str (const InformationState &state)
- bool extends (const Possibility &p2, const Possibility &p1)

  Determines whether one Possibility extends another.
- bool operator< (const Possibility &p1, const Possibility &p2)
- std::string str (const Possibility &p)
- bool extends (const ReferentSystem &r2, const ReferentSystem &r1)

  Determines whether one ReferentSystem extends another.
- std::string str (const ReferentSystem &r)

4.3.1 Typedef Documentation

4.3.1.1 InformationState

```
using iif_sadaf::talk::GSV::InformationState = std::set<Possibility>
```

An alias for `std::set<Possibility>`

Definition at line 15 of file information_state.hpp.

4.3.2 Function Documentation

4.3.2.1 create()

```
InformationState iif_sadaf::talk::GSV::create (
            const Model & model)
```

Creates an information state based on a model.

This function creates an InformationState object containing exactly one possibility for each possible world in the base model.

Parameters

model	The model upon which the information state is based

Returns

A new information state

Definition at line 18 of file information_state.cpp.

4.3.2.2 extends() [1/3]

```
bool iif_sadaf::talk::GSV::extends (
            const InformationState & s2,
            const InformationState & s1)
```

Determines if one information state extends another.

Checks whether every possibility in s2 extends at least one possibility in s1.

Parameters

s2	The potentially extending information state.
s1	The base information state.

Returns

True if s2 extends s1, false otherwise.

Definition at line 76 of file information_state.cpp.

4.3.2.3 extends() [2/3]

```
bool iif_sadaf::talk::GSV::extends (
            const Possibility & p2,
            const Possibility & p1)
```

Determines whether one Possibility extends another.

A Possibility `p2` extends `p1` if:

- They have the same world.
- Every peg mapped in `p1` has the same individual in `p2`.

Parameters

p2	The potential extending Possibility.
p1	The base Possibility.

Returns

True if `p2` extends `p1`, false otherwise.

Definition at line 76 of file possibility.cpp.

4.3.2.4 extends() [3/3]

```
bool iif_sadaf::talk::GSV::extends (
            const ReferentSystem & r2,
            const ReferentSystem & r1)
```

Determines whether one ReferentSystem extends another.

This function checks whether the referent system `r2` extends the referent system `r1`. A referent system `r2` extends `r1` if:

- The range of `r1` is a subset of the range of `r2`.
- The domain of `r1` is a subset of the domain of `r2`.
- Variables in `r1` retain their values in `r2`, or their values are new relative to `r1`.
- New variables in `r2` have new values relative to `r1`.

Parameters

r2	The potential extending ReferentSystem.
r1	The base ReferentSystem.

Returns

True if `r2` extends `r1`, false otherwise.

Definition at line 100 of file referent_system.cpp.

4.3.2.5 isDescendantOf()

```
bool iif_sadaf::talk::GSV::isDescendantOf (
             const Possibility & p2,
             const Possibility & p1,
             const InformationState & s)
```

Determines if one possibility is a descendant of another within an information state.

A possibility p2 is a descendant of p1 if it extends p1 and is contained in the given information state.

Parameters

p2	The potential descendant possibility.
p1	The potential ancestor possibility.
s	The information state in which the relationship is checked.

Returns

True if p2 is a descendant of p1 in s, false otherwise.

Definition at line 98 of file information_state.cpp.

4.3.2.6 operator<()

```
bool iif_sadaf::talk::GSV::operator< (
             const Possibility & p1,
             const Possibility & p2)
```

Definition at line 88 of file possibility.cpp.

4.3.2.7 str() [1/3]

```
std::string iif_sadaf::talk::GSV::str (
             const InformationState & state)
```

Definition at line 133 of file information_state.cpp.

4.3.2.8 str() [2/3]

```
std::string iif_sadaf::talk::GSV::str (
            const Possibility & p)
```

Definition at line 93 of file possibility.cpp.

4.3.2.9 str() [3/3]

```
std::string iif_sadaf::talk::GSV::str (
            const ReferentSystem & r)
```

Definition at line 69 of file referent_system.cpp.

4.3.2.10 subsistsIn() [1/2]

```
bool iif_sadaf::talk::GSV::subsistsIn (
            const InformationState & s1,
            const InformationState & s2)
```

Checks if an information state subsists within another.

An information state s1 subsists in s2 if all possibilities in s1 have corresponding possibilities in s2.

Parameters

s1	The potential subsisting state.
s2	The state in which s1 may subsist.

Returns

True if s1 subsists in s2, false otherwise.

Definition at line 127 of file information_state.cpp.

4.3.2.11 subsistsIn() [2/2]

```
bool iif_sadaf::talk::GSV::subsistsIn (
            const Possibility & p,
            const InformationState & s)
```

Checks if a possibility subsists in an information state.

A possibility subsists in an information state if at least one of its descendants exists within the state.

Parameters

p	The possibility to check.
s	The information state.

Returns

True if p subsists in s, false otherwise.

Definition at line 112 of file information_state.cpp.

4.3.2.12 update()

```
InformationState iif_sadaf::talk::GSV::update (
            const InformationState & input_state,
            std::string_view variable,
            int individual)
```

Updates the information state with a new variable-individual assignment.

Creates a new information state where each possibility has been updated with the given variable-individual assignment.

Parameters

input_state	The original information state.
variable	The variable to be added or updated.
individual	The individual assigned to the variable.

Returns

A new updated information state.

Definition at line 43 of file information_state.cpp.

Chapter 5

Class Documentation

5.1 iif_sadaf::talk::GSV::Evaluator Struct Reference

Represents an evaluator for logical expressions.

`#include <evaluator.hpp>`

Public Member Functions

- InformationState operator() (std::shared_ptr< UnaryNode > expr, std::variant< std::pair< InformationState, const IModel $*$ > > params) const

  Evaluates a unary logical expression on an InformationState.
- InformationState operator() (std::shared_ptr< BinaryNode > expr, std::variant< std::pair< InformationState, const IModel $*$ > > params) const

  Evaluates a binary logical expression on an InformationState.
- InformationState operator() (std::shared_ptr< QuantificationNode > expr, std::variant< std::pair< InformationState, const IModel $*$ > > params) const

  Evaluates a quantified expression on an InformationState.
- InformationState operator() (std::shared_ptr< IdentityNode > expr, std::variant< std::pair< InformationState, const IModel $*$ > > params) const

  Evaluates an identity expression, filtering based on variable or term equality.
- InformationState operator() (std::shared_ptr< PredicationNode > expr, std::variant< std::pair< InformationState, const IModel $*$ > > params) const

  Evaluates a predicate expression by filtering states based on predicate denotation.

5.1.1 Detailed Description

Represents an evaluator for logical expressions.

The Evaluator struct applies logical operations on `InformationState` objects using the visitor pattern. It also takes an IModel$*$. It evaluates different types of logical expressions, including unary, binary, quantification, identity, and predication nodes. The evaluation modifies or filters the given `InformationState` and `IModel*`, based on the logical rules applied.

Due to the way `std::visit` is implemented in C++, the input `InformationState` and `IModel*` must be wrapped in a `std::variant` and passed as a single argument.

The application of `GSV::EValuator()` may throw std::invalid_argument, under various circumstances (see the member functions' documentation for details).

Definition at line 23 of file evaluator.hpp.

5.1.2 Member Function Documentation

5.1.2.1 operator()() [1/5]

```
InformationState iif_sadaf::talk::GSV::Evaluator::operator() (
            std::shared_ptr< BinaryNode > expr,
            std::variant< std::pair< InformationState, const IModel * > > params) const
```

Evaluates a binary logical expression on an InformationState.

Processes logical operations such as conjunction, disjunction, and implication, modifying the state accordingly.

Parameters

expr	The binary expression to evaluate.
params	The input information state and IModel pointer

Returns

The modified InformationState after applying the operation.

Exceptions

std::invalid_argument	if the operator is invalid.

Definition at line 89 of file evaluator.cpp.

5.1.2.2 operator()() [2/5]

```
InformationState iif_sadaf::talk::GSV::Evaluator::operator() (
            std::shared_ptr< IdentityNode > expr,
            std::variant< std::pair< InformationState, const IModel * > > params) const
```

Evaluates an identity expression, filtering based on variable or term equality.

Compares the denotation of two terms or variables and retains only the possibilities where they are equal.

May throw std::out_of_range if either the LHS or the RHS of the identity lack an interpretation in the base model for the information state, or are variables without a binding quantifier or a proper anaphoric antecedent.

Parameters

expr	The identity expression to evaluate.
params	The input information state and IModel pointer

Returns

The filtered InformationState after applying identity conditions.

Exceptions

std::invalid_argument	if the quantifier is invalid.

Definition at line 224 of file evaluator.cpp.

5.1.2.3 operator()() [3/5]

```
InformationState iif_sadaf::talk::GSV::Evaluator::operator() (
            std::shared_ptr< PredicationNode > expr,
            std::variant< std::pair< InformationState, const IModel * > > params) const
```

Evaluates a predicate expression by filtering states based on predicate denotation.

Checks if a given predicate holds in the current world and filters possibilities accordingly.

May throw std::out_of_range if (i) any argument to the predicate lacks an interpretation in the base model for the information state, or is a variable without a binding quantifier or a proper anaphoric antecedent, or (ii) the predicate lacks an interpretation in the base model for the information state.

Parameters

expr	The predicate expression to evaluate.
params	The input information state and IModel pointer

Returns

The filtered InformationState after evaluating the predicate.

Exceptions

std::invalid_argument	if the quantifier is invalid.

Definition at line 256 of file evaluator.cpp.

5.1.2.4 operator()() [4/5]

```
InformationState iif_sadaf::talk::GSV::Evaluator::operator() (
            std::shared_ptr< QuantificationNode > expr,
            std::variant< std::pair< InformationState, const IModel * > > params) const
```

Evaluates a quantified expression on an InformationState.

Handles existential and universal quantifiers by iterating over possible individuals in the model and updating the state accordingly.

Parameters

expr	The quantification expression to evaluate.
params	The input information state and IModel pointer

Returns

The modified InformationState after applying the quantification.

Exceptions

std::invalid_argument	if the quantifier is invalid.

Definition at line 155 of file evaluator.cpp.

5.1.2.5 operator()() [5/5]

```
InformationState iif_sadaf::talk::GSV::Evaluator::operator() (
            std::shared_ptr< UnaryNode > expr,
            std::variant< std::pair< InformationState, const IModel * > > params) const
```

Evaluates a unary logical expression on an InformationState.

Applies an operator (such as necessity, possibility, or negation) to modify the given state accordingly.

Parameters

expr	The unary expression to evaluate.
params	The input information state and IModel pointer

Returns

The modified InformationState after applying the operation.

Exceptions

std::invalid_argument	if the operator is invalid.

Definition at line 53 of file evaluator.cpp.

The documentation for this struct was generated from the following files:

- evaluator.hpp
- evaluator.cpp

5.2 iif_sadaf::talk::GSV::IModel Struct Reference

Interface for class representing a model for QML without accessiblity.

```
#include <imodel.hpp>
```

Public Member Functions

- virtual int world_cardinality () const =0
- virtual int domain_cardinality () const =0
- virtual int termInterpretation (std::string_view term, int world) const =0
- virtual const std::set< std::vector< int > > & predicateInterpretation (std::string_view predicate, int world) const =0
- virtual ∼IModel ()

5.2.1 Detailed Description

Interface for class representing a model for QML without accessiblity.

The IModel interface defines the minimal requirements on any implementation of a QML model that works with the GSV evaluator library.

Any such implementation should contain four functions:

- a function retrieving the cardinality of the set W of worlds
- a function retrieving the cardinality of the domain of individuals
- a function that retrieves, for any possible world in W, the interpretation of a singular term at that world (represented by an `int`)
- a function that retrieves, for any possible world in W, the interpretation of a predicate at that world (represented by a `std::set<std::vector<int>>`)

Definition at line 21 of file imodel.hpp.

5.2.2 Constructor & Destructor Documentation

5.2.2.1 ∼IModel()

```
virtual iif_sadaf::talk::GSV::IModel::∼IModel ()  [inline], [virtual]
```

Definition at line 27 of file imodel.hpp.

5.2.3 Member Function Documentation

5.2.3.1 domain_cardinality()

```
virtual int iif_sadaf::talk::GSV::IModel::domain_cardinality () const  [pure virtual]
```

5.2.3.2 predicateInterpretation()

```
virtual const std::set< std::vector< int > > & iif_sadaf::talk::GSV::IModel::predicate↩
Interpretation (
            std::string_view predicate,
            int world) const  [pure virtual]
```

5.2.3.3 termInterpretation()

```
virtual int iif_sadaf::talk::GSV::IModel::termInterpretation (
            std::string_view term,
            int world) const  [pure virtual]
```

5.2.3.4 world_cardinality()

```
virtual int iif_sadaf::talk::GSV::IModel::world_cardinality () const  [pure virtual]
```

The documentation for this struct was generated from the following file:

- imodel.hpp

5.3 iif_sadaf::talk::GSV::Possibility Struct Reference

Represents a possibility as understood in the underlying semantics.

```
#include <possibility.hpp>
```

Public Member Functions

- Possibility (std::shared_ptr< ReferentSystem > r_system, int world)
- Possibility (const Possibility &other)
- Possibility & operator= (const Possibility &other)
- Possibility (Possibility &&other) noexcept
- Possibility & operator= (Possibility &&other) noexcept
- void update (std::string_view variable, int individual)

  Updates the assignment of a variable to an individual.

Public Attributes

- std::shared_ptr< ReferentSystem > referentSystem
- std::unordered_map< int, int > assignment
- int world

5.3.1 Detailed Description

Represents a possibility as understood in the underlying semantics.

Possibilities are just tuples of a referent system, an assignment of individuals to pegs, and a possible world.

The class also contains a few convenience functions for handling the first two components.

Definition at line 18 of file possibility.hpp.

5.3.2 Constructor & Destructor Documentation

5.3.2.1 Possibility() [1/3]

```
iif_sadaf::talk::GSV::Possibility::Possibility (
             std::shared_ptr< ReferentSystem > r_system,
             int world)
```

Definition at line 7 of file possibility.cpp.

5.3.2.2 Possibility() [2/3]

```
iif_sadaf::talk::GSV::Possibility::Possibility (
             const Possibility & other)
```

Definition at line 13 of file possibility.cpp.

5.3.2.3 Possibility() [3/3]

```
iif_sadaf::talk::GSV::Possibility::Possibility (
             Possibility && other)  [noexcept]
```

Definition at line 30 of file possibility.cpp.

5.3.3 Member Function Documentation

5.3.3.1 operator=() [1/2]

```
Possibility & iif_sadaf::talk::GSV::Possibility::operator= (
             const Possibility & other)
```

Definition at line 19 of file possibility.cpp.

5.3.3.2 operator=() [2/2]

```
Possibility & iif_sadaf::talk::GSV::Possibility::operator= (
             Possibility && other)  [noexcept]
```

Definition at line 36 of file possibility.cpp.

5.3.3.3 update()

```
void iif_sadaf::talk::GSV::Possibility::update (
             std::string_view variable,
             int individual)
```

Updates the assignment of a variable to an individual.

The variable is first added to or updated in the associated referent system. Then, the assignment is modified to map the peg of the variable to the new individual.

Parameters

variable	The variable to update.
individual	The new individual assigned to the variable.

Definition at line 55 of file possibility.cpp.

5.3.4 Member Data Documentation

5.3.4.1 assignment

```
std::unordered_map<int, int> iif_sadaf::talk::GSV::Possibility::assignment
```

Definition at line 29 of file possibility.hpp.

5.3.4.2 referentSystem

```
std::shared_ptr<ReferentSystem> iif_sadaf::talk::GSV::Possibility::referentSystem
```

Definition at line 28 of file possibility.hpp.

5.3.4.3 world

```
int iif_sadaf::talk::GSV::Possibility::world
```

Definition at line 30 of file possibility.hpp.

The documentation for this struct was generated from the following files:

- possibility.hpp
- possibility.cpp

5.4 iif_sadaf::talk::GSV::ReferentSystem Struct Reference

Represents a referent system for variable assignments.

```
#include <referent_system.hpp>
```

Public Member Functions

- ReferentSystem ()=default
- ReferentSystem (const ReferentSystem &other)
- ReferentSystem & operator= (const ReferentSystem &other)
- ReferentSystem (ReferentSystem &&other) noexcept
- ReferentSystem & operator= (ReferentSystem &&other) noexcept
- int value (std::string_view variable) const

  Retrieves the peg value associated with a given variable.

Public Attributes

- int pegs = 0
- std::unordered_map< std::string_view, int > variablePegAssociation = {}

5.4.1 Detailed Description

Represents a referent system for variable assignments.

The ReferentSystem class maintains associations between variables and integer pegs.

Definition at line 15 of file referent_system.hpp.

5.4.2 Constructor & Destructor Documentation

5.4.2.1 ReferentSystem() [1/3]

```
iif_sadaf::talk::GSV::ReferentSystem::ReferentSystem ()  [default]
```

5.4.2.2 ReferentSystem() [2/3]

```
iif_sadaf::talk::GSV::ReferentSystem::ReferentSystem (
             const ReferentSystem & other)
```

Definition at line 22 of file referent_system.cpp.

5.4.2.3 ReferentSystem() [3/3]

```
iif_sadaf::talk::GSV::ReferentSystem::ReferentSystem (
             ReferentSystem && other)  [noexcept]
```

Definition at line 37 of file referent_system.cpp.

5.4.3 Member Function Documentation

5.4.3.1 operator=() [1/2]

```
ReferentSystem & iif_sadaf::talk::GSV::ReferentSystem::operator= (
             const ReferentSystem & other)
```

Definition at line 27 of file referent_system.cpp.

5.4.3.2 operator=() [2/2]

```
ReferentSystem & iif_sadaf::talk::GSV::ReferentSystem::operator= (
             ReferentSystem && other)  [noexcept]
```

Definition at line 42 of file referent_system.cpp.

5.4.3.3 value()

```
int iif_sadaf::talk::GSV::ReferentSystem::value (
             std::string_view variable) const
```

Retrieves the peg value associated with a given variable.

Parameters

variable	The variable whose peg value is to be retrieved.

Returns

The peg value associated with the variable.

Exceptions

std::out_of_range	If the variable has no associated peg.

Definition at line 59 of file referent_system.cpp.

5.4.4 Member Data Documentation

5.4.4.1 pegs

```
int iif_sadaf::talk::GSV::ReferentSystem::pegs = 0
```

Definition at line 25 of file referent_system.hpp.

5.4.4.2 variablePegAssociation

```
std::unordered_map<std::string_view, int> iif_sadaf::talk::GSV::ReferentSystem::variablePeg↩
Association = {}
```

Definition at line 26 of file referent_system.hpp.

The documentation for this struct was generated from the following files:

- referent_system.hpp
- referent_system.cpp

Chapter 6

File Documentation

6.1 evaluator.hpp File Reference

```
#include "expression.hpp"
#include "information_state.hpp"
```

Classes

- struct iif_sadaf::talk::GSV::Evaluator

  Represents an evaluator for logical expressions.

Namespaces

- namespace iif_sadaf
- namespace iif_sadaf::talk
- namespace iif_sadaf::talk::GSV

6.2 evaluator.hpp

Go to the documentation of this file.

```
#pragma once

#include "expression.hpp"
#include "information_state.hpp"

namespace iif_sadaf::talk::GSV {

struct Evaluator {
    InformationState operator()(std::shared_ptr<UnaryNode> expr, std::variant<std::pair<InformationState, const IModel*» params) const;
    InformationState operator()(std::shared_ptr<BinaryNode> expr, std::variant<std::pair<InformationState, const IModel*» params) const;
    InformationState operator()(std::shared_ptr<QuantificationNode> expr, std::variant<std::pair<InformationState, const IModel*» params) const;
    InformationState operator()(std::shared_ptr<IdentityNode> expr, std::variant<std::pair<InformationState, const IModel*» params) const;
    InformationState operator()(std::shared_ptr<PredicationNode> expr, std::variant<std::pair<InformationState, const IModel*» params) const;
};

}
```

6.3 imodel.hpp File Reference

```
#include <set>
#include <string_view>
#include <vector>
```

Classes

- struct iif_sadaf::talk::GSV::IModel

  Interface for class representing a model for QML without accessiblity.

Namespaces

- namespace iif_sadaf
- namespace iif_sadaf::talk
- namespace iif_sadaf::talk::GSV

6.4 imodel.hpp

Go to the documentation of this file.

```
#pragma once

#include <set>
#include <string_view>
#include <vector>

namespace iif_sadaf::talk::GSV {

struct IModel {
public:
    virtual int world_cardinality() const = 0;
    virtual int domain_cardinality() const = 0;
    virtual int termInterpretation(std::string_view term, int world) const = 0;
    virtual const std::set<std::vector<int»& predicateInterpretation(std::string_view predicate, int world) const = 0;
    virtual ~IModel() {};
};

}
```

6.5 information_state.hpp File Reference

```
#include <set>
#include <string>
#include <string_view>
#include "model.hpp"
#include "possibility.hpp"
```

Namespaces

- namespace iif_sadaf
- namespace iif_sadaf::talk
- namespace iif_sadaf::talk::GSV

Typedefs

- using iif_sadaf::talk::GSV::InformationState = std::set<Possibility>

    An alias for `std::set<Possibility>`

Functions

- InformationState iif_sadaf::talk::GSV::create (const Model &model)

    Creates an information state based on a model.
- InformationState iif_sadaf::talk::GSV::update (const InformationState &input_state, std::string_view variable, int individual)

    Updates the information state with a new variable-individual assignment.
- bool iif_sadaf::talk::GSV::extends (const InformationState &s2, const InformationState &s1)

    Determines if one information state extends another.
- bool iif_sadaf::talk::GSV::isDescendantOf (const Possibility &p2, const Possibility &p1, const InformationState &s)

    Determines if one possibility is a descendant of another within an information state.
- bool iif_sadaf::talk::GSV::subsistsIn (const Possibility &p, const InformationState &s)

    Checks if a possibility subsists in an information state.
- bool iif_sadaf::talk::GSV::subsistsIn (const InformationState &s1, const InformationState &s2)

    Checks if an information state subsists within another.
- std::string iif_sadaf::talk::GSV::str (const InformationState &state)

6.6 information_state.hpp

Go to the documentation of this file.

```
#pragma once

#include <set>
#include <string>
#include <string_view>

#include "model.hpp"
#include "possibility.hpp"

namespace iif_sadaf::talk::GSV {

using InformationState = std::set<Possibility>;

InformationState create(const Model& model);
InformationState update(const InformationState& input_state, std::string_view variable, int individual);
bool extends(const InformationState& s2, const InformationState& s1);

bool isDescendantOf(const Possibility& p2, const Possibility& p1, const InformationState& s);
bool subsistsIn(const Possibility& p, const InformationState& s);
bool subsistsIn(const InformationState& s1, const InformationState& s2);

std::string str(const InformationState& state);
}
```

6.7 possibility.hpp File Reference

```
#include <memory>
#include <string>
#include <unordered_map>
#include "referent_system.hpp"
```

Classes

- struct iif_sadaf::talk::GSV::Possibility

  Represents a possibility as understood in the underlying semantics.

Namespaces

- namespace iif_sadaf
- namespace iif_sadaf::talk
- namespace iif_sadaf::talk::GSV

Functions

- bool iif_sadaf::talk::GSV::extends (const Possibility &p2, const Possibility &p1)

  Determines whether one Possibility extends another.
- bool iif_sadaf::talk::GSV::operator< (const Possibility &p1, const Possibility &p2)
- std::string iif_sadaf::talk::GSV::str (const Possibility &p)

6.8 possibility.hpp

Go to the documentation of this file.

```
#pragma once

#include <memory>
#include <string>
#include <unordered_map>

#include "referent_system.hpp"

namespace iif_sadaf::talk::GSV {

struct Possibility {
public:
    Possibility(std::shared_ptr<ReferentSystem> r_system, int world);
    Possibility(const Possibility& other);
    Possibility& operator=(const Possibility& other);
    Possibility(Possibility&& other) noexcept;
    Possibility& operator=(Possibility&& other) noexcept;

    void update(std::string_view variable, int individual);

    std::shared_ptr<ReferentSystem> referentSystem;
    std::unordered_map<int, int> assignment;
    int world;
};

bool extends(const Possibility& p2, const Possibility& p1);
bool operator<(const Possibility& p1, const Possibility& p2);

std::string str(const Possibility& p);

}
```

6.9 referent_system.hpp File Reference

```
#include <set>
#include <string>
#include <string_view>
#include <unordered_map>
```

Classes

- struct iif_sadaf::talk::GSV::ReferentSystem

  Represents a referent system for variable assignments.

Namespaces

- namespace iif_sadaf
- namespace iif_sadaf::talk
- namespace iif_sadaf::talk::GSV

Functions

- bool iif_sadaf::talk::GSV::extends (const ReferentSystem &r2, const ReferentSystem &r1)

  Determines whether one ReferentSystem extends another.
- std::string iif_sadaf::talk::GSV::str (const ReferentSystem &r)

6.10 referent_system.hpp

Go to the documentation of this file.

```
#pragma once

#include <set>
#include <string>
#include <string_view>
#include <unordered_map>

namespace iif_sadaf::talk::GSV {

struct ReferentSystem {
public:
    ReferentSystem() = default;
    ReferentSystem(const ReferentSystem& other);
    ReferentSystem& operator=(const ReferentSystem& other);
    ReferentSystem(ReferentSystem&& other) noexcept;
    ReferentSystem& operator=(ReferentSystem&& other) noexcept;

    int value(std::string_view variable) const;

    int pegs = 0;
    std::unordered_map<std::string_view, int> variablePegAssociation = {};
};

bool extends(const ReferentSystem& r2, const ReferentSystem& r1);
std::string str(const ReferentSystem& r);

}
```

6.11 evaluator.cpp File Reference

```
#include "evaluator.hpp"
#include <algorithm>
#include <functional>
#include <stdexcept>
#include <ranges>
#include "variable.hpp"
```

Namespaces

- namespace iif_sadaf
- namespace iif_sadaf::talk
- namespace iif_sadaf::talk::GSV

6.12 evaluator.cpp

Go to the documentation of this file.

```
#include "evaluator.hpp"

#include <algorithm>
#include <functional>
#include <stdexcept>
#include <ranges>

#include "variable.hpp"

namespace iif_sadaf::talk::GSV {

namespace {

void filter(InformationState& state, const std::function<bool(const Possibility&)>& predicate) {
    for (auto it = state.begin(); it != state.end(); ) {
        if (!predicate(*it)) {
            it = state.erase(it);
        }
        else {
            ++it;
        }
    }
}

int termDenotation(std::string_view term, int w, const IModel& m)
{
    return m.termInterpretation(term, w);
}

const std::set<std::vector<int»& predicateDenotation(std::string_view predicate, int w, const IModel& m)
{
    return m.predicateInterpretation(predicate, w);
}

int variableDenotation(std::string_view variable, const Possibility& p)
{
    return p.assignment.at(p.referentSystem->value(variable));
}

} // ANONYMOUS NAMESPACE

InformationState Evaluator::operator()(std::shared_ptr<UnaryNode> expr, std::variant<std::pair<InformationState, const IModel*» params) const
{
    InformationState hypothetical_update = std::visit(Evaluator(), expr->scope, params);
    InformationState& input_state = (std::get<std::pair<InformationState, const IModel*»(params)).first;

    if (expr->op == Operator::E_POS) {
        if (hypothetical_update.empty()) {
            input_state.clear();
        }
    }
    else if (expr->op == Operator::E_NEC) {
        if (!subsistsIn(input_state, hypothetical_update)) {
            input_state.clear();
        }
    }
    else if (expr->op == Operator::NEG) {
        filter(input_state, [&](const Possibility& p) -> bool { return !subsistsIn(p, hypothetical_update); });
    }
    else {
        throw(std::invalid_argument("Invalid operator for unary formula"));
    }

    return std::move(input_state);
}
```

```

InformationState Evaluator::operator()(std::shared_ptr<BinaryNode> expr, std::variant<std::pair<InformationState, const IModel*» params) const
{
    const IModel* model = (std::get<std::pair<InformationState, const IModel*»(params)).second;

    if (expr->op == Operator::CON) {
        return std::visit(
            Evaluator(),
            expr->rhs,
            std::variant<std::pair<InformationState, const IModel*»(std::make_pair(std::visit(Evaluator(), expr->lhs, params), model))
        );
    }

    InformationState& input_state = (std::get<std::pair<InformationState, const IModel*»(params)).first;
    InformationState hypothetical_update_lhs = std::visit(Evaluator(), expr->lhs, params);

    if (expr->op == Operator::DIS) {
        InformationState hypothetical_update_rhs = std::visit(
            Evaluator(),
            expr->rhs,
            std::variant<std::pair<InformationState, const IModel*»(std::make_pair(std::visit(Evaluator(), negate(expr->lhs), params), model))
        );

        const auto in_lhs_or_in_rhs = [&](const Possibility& p) -> bool {
            return hypothetical_update_lhs.contains(p) || hypothetical_update_rhs.contains(p);
        };

        filter(input_state, in_lhs_or_in_rhs);
    }
    else if (expr->op == Operator::IMP) {
        InformationState hypothetical_update_consequent = std::visit(
            Evaluator(),
            expr->rhs,
            std::variant<std::pair<InformationState, const IModel*»(std::make_pair(hypothetical_update_lhs, model))
        );

        auto all_descendants_subsist = [&](const Possibility& p) -> bool {
            auto not_descendant_or_subsists = [&](const Possibility& p_star) -> bool {
                return !isDescendantOf(p_star, p, hypothetical_update_lhs) || subsistsIn(p_star, hypothetical_update_consequent);
            };
            return std::ranges::all_of(hypothetical_update_lhs, not_descendant_or_subsists);
        };

        const auto if_subsists_all_descendants_do = [&](const Possibility& p) -> bool {
            return !subsistsIn(p, hypothetical_update_lhs) || all_descendants_subsist(p);
        };

        filter(input_state, if_subsists_all_descendants_do);
    }
    else {
        throw(std::invalid_argument("Invalid operator for binary formula"));
    }

    return std::move(input_state);
}

InformationState Evaluator::operator()(std::shared_ptr<QuantificationNode> expr, std::variant<std::pair<InformationState, const IModel*» params) const
{
    InformationState& input_state = (std::get<std::pair<InformationState, const IModel*»(params)).first;
    const IModel* model = (std::get<std::pair<InformationState, const IModel*»(params)).second;

    if (expr->quantifier == Quantifier::EXISTENTIAL) {
        std::vector<InformationState> all_state_variants;

        for (int i : std::views::iota(0, model->domain_cardinality())) {
            InformationState s_variant = update(input_state, expr->variable, i);
            all_state_variants.push_back(std::visit(
                Evaluator(),
                expr->scope,
                std::variant<std::pair<InformationState, const IModel*»(std::make_pair(s_variant, model)))
            );
        }

        InformationState output;
        for (const auto& state_variant : all_state_variants) {
            for (const auto& p : state_variant) {
                output.insert(p);
            }
        }

        return output;
    }
    else if (expr->quantifier == Quantifier::UNIVERSAL) {
        std::vector<InformationState> all_hypothetical_updates;

        for (int d : std::views::iota(0, model->domain_cardinality())) {
            InformationState hypothetical_update = std::visit(
                Evaluator(),
                expr->scope,
                std::variant<std::pair<InformationState, const IModel*»(std::make_pair(update(input_state, expr->variable, d), model))
            );
            all_hypothetical_updates.push_back(hypothetical_update);
        }

        const auto subsists_in_all_hyp_updates = [&](const Possibility& p) -> bool {
            const auto p_subsists_in_hyp_update = [&](const InformationState& hypothetical_update) -> bool {
                return subsistsIn(p, hypothetical_update);
            };
            return std::ranges::all_of(all_hypothetical_updates, p_subsists_in_hyp_update);
        };

        filter(input_state, subsists_in_all_hyp_updates);
    }
    else {
        throw(std::invalid_argument("Invalid quantifier"));
    }

    return std::move(input_state);
}

InformationState Evaluator::operator()(std::shared_ptr<IdentityNode> expr, std::variant<std::pair<InformationState, const IModel*» params) const
{
    InformationState& input_state = (std::get<std::pair<InformationState, const IModel*»(params)).first;
    const IModel& model = *(std::get<std::pair<InformationState, const IModel*»(params)).second;

    auto assigns_same_denotation = [&](const Possibility& p) -> bool {
        const int lhs_denotation = isVariable(expr->lhs) ? variableDenotation(expr->lhs, p) : termDenotation(expr->lhs, p.world, model);
        const int rhs_denotation = isVariable(expr->lhs) ? variableDenotation(expr->rhs, p) : termDenotation(expr->rhs, p.world, model);
        return lhs_denotation == rhs_denotation;
    };

    filter(input_state, assigns_same_denotation);

    return std::move(input_state);
}

InformationState Evaluator::operator()(std::shared_ptr<PredicationNode> expr, std::variant<std::pair<InformationState, const IModel*» params) const
{
    InformationState& input_state = (std::get<std::pair<InformationState, const IModel*»(params)).first;
    const IModel& model = *(std::get<std::pair<InformationState, const IModel*»(params)).second;

    auto tuple_in_extension = [&](const Possibility& p) -> bool {
        std::vector<int> tuple;

        for (const std::string& argument : expr->arguments) {
            const int denotation = isVariable(argument) ? variableDenotation(argument, p) : termDenotation(argument, p.world, model);
            tuple.push_back(denotation);
        }

        return predicateDenotation(expr->predicate, p.world, model).contains(tuple);
    };

    filter(input_state, tuple_in_extension);

    return std::move(input_state);
}

}
```

6.13 information_state.cpp File Reference

```
#include "information_state.hpp"
#include <algorithm>
#include <iostream>
#include <memory>
```

Namespaces

- namespace iif_sadaf
- namespace iif_sadaf::talk
- namespace iif_sadaf::talk::GSV

Functions

- InformationState iif_sadaf::talk::GSV::create (const Model &model)

  Creates an information state based on a model.
- InformationState iif_sadaf::talk::GSV::update (const InformationState &input_state, std::string_view variable, int individual)

  Updates the information state with a new variable-individual assignment.
- bool iif_sadaf::talk::GSV::extends (const InformationState &s2, const InformationState &s1)

  Determines if one information state extends another.
- bool iif_sadaf::talk::GSV::isDescendantOf (const Possibility &p2, const Possibility &p1, const InformationState &s)

  Determines if one possibility is a descendant of another within an information state.
- bool iif_sadaf::talk::GSV::subsistsIn (const Possibility &p, const InformationState &s)

  Checks if a possibility subsists in an information state.
- bool iif_sadaf::talk::GSV::subsistsIn (const InformationState &s1, const InformationState &s2)

  Checks if an information state subsists within another.
- std::string iif_sadaf::talk::GSV::str (const InformationState &state)

6.14 information_state.cpp

Go to the documentation of this file.

```
#include "information_state.hpp"

#include <algorithm>
#include <iostream>
#include <memory>

namespace iif_sadaf::talk::GSV {

InformationState create(const Model& model)
{
    std::set<Possibility> possibilities;

    auto r_system = std::make_shared<ReferentSystem>();

    const int number_of_worlds = model.world_cardinality();
    for (int i = 0; i < number_of_worlds; ++i) {
        possibilities.emplace(r_system, i);
    }

    return possibilities;
}

InformationState update(const InformationState& input_state, std::string_view variable, int individual)
{
    InformationState output_state;

    auto r_star = std::make_shared<ReferentSystem>();

    for (const auto& p : input_state) {
        Possibility p_star(r_star, p.world);
        p_star.assignment = p.assignment;
        r_star->pegs = p.referentSystem->pegs;
        for (const auto& map : p.referentSystem->variablePegAssociation) {
            auto var = map.first;
            int peg = map.second;
            r_star->variablePegAssociation[var] = peg;
        }

        p_star.update(variable, individual);

        output_state.insert(p_star);
    }

    return output_state;
}

bool extends(const InformationState& s2, const InformationState& s1)
{
    const auto extends_possibility_in_s1 = [&](const Possibility& p2) -> bool {
        const auto is_extended_by_p2 = [&](const Possibility& p1) -> bool {
            return extends(p2, p1);
        };
        return std::ranges::any_of(s1, is_extended_by_p2);
    };

    return std::ranges::all_of(s2, extends_possibility_in_s1);
}

bool isDescendantOf(const Possibility& p2, const Possibility& p1, const InformationState& s)
{
    return s.contains(p2) && (extends(p2, p1));
}

bool subsistsIn(const Possibility& p, const InformationState& s)
{
    const auto is_descendant_of_p_in_s = [&](const Possibility& p1) -> bool { return isDescendantOf(p1, p, s); };
    return std::ranges::any_of(s, is_descendant_of_p_in_s);
}

bool subsistsIn(const InformationState& s1, const InformationState& s2)
{
    const auto subsists_in_s2 = [&](const Possibility& p) -> bool { return subsistsIn(p, s2); };
    return std::ranges::all_of(s1, subsists_in_s2);
}

std::string str(const InformationState& state)
{
    std::string desc;

    desc += "--------------------\n";
    for (const Possibility& p : state) {
        desc += str(p);
        desc += "--------------------\n";
    }

    desc.pop_back();

    return desc;
}

}
```

6.15 possibility.cpp File Reference

```
#include "possibility.hpp"
#include <algorithm>
```

Namespaces

- namespace iif_sadaf
- namespace iif_sadaf::talk
- namespace iif_sadaf::talk::GSV

Functions

- bool iif_sadaf::talk::GSV::extends (const Possibility &p2, const Possibility &p1)

  Determines whether one Possibility extends another.
- bool iif_sadaf::talk::GSV::operator< (const Possibility &p1, const Possibility &p2)
- std::string iif_sadaf::talk::GSV::str (const Possibility &p)

6.16 possibility.cpp

Go to the documentation of this file.

```
#include "possibility.hpp"

#include <algorithm>

namespace iif_sadaf::talk::GSV {

Possibility::Possibility(std::shared_ptr<ReferentSystem> r_system, int world)
    : referentSystem(r_system)
    , assignment({})
    , world(world)
{ }

Possibility::Possibility(const Possibility& other)
    : referentSystem(other.referentSystem)
    , assignment(other.assignment)
    , world(other.world)
{ }

Possibility& Possibility::operator=(const Possibility& other)
{
    if (this != &other) {
        this->referentSystem = other.referentSystem;
        this->assignment = other.assignment;
        this->world = other.world;
    }

    return *this;
}

Possibility::Possibility(Possibility&& other) noexcept
    : referentSystem(std::move(other.referentSystem))
    , assignment(std::move(other.assignment))
    , world(other.world)
{ }

Possibility& Possibility::operator=(Possibility&& other) noexcept
{
    if (this != &other) {
        this->referentSystem = std::move(other.referentSystem);
        this->assignment = std::move(other.assignment);
        this->world = other.world;
    }
    return *this;
}

void Possibility::update(std::string_view variable, int individual)
{
    referentSystem->variablePegAssociation[variable] = ++(referentSystem->pegs);
    assignment[referentSystem->pegs] = individual;
}

/*
* NON-MEMBER FUNCTIONS
*/

bool extends(const Possibility& p2, const Possibility& p1)
{
    const auto peg_is_new_or_maintains_assignment = [&](const std::pair<int, int>& map) -> bool {
        int peg = map.first;
        int individual = map.second;

        return !p1.assignment.contains(peg) || (p1.assignment.at(peg) == p2.assignment.at(peg));
    };

    return (p1.world == p2.world) && std::ranges::all_of(p2.assignment, peg_is_new_or_maintains_assignment);
}

bool operator<(const Possibility& p1, const Possibility& p2)
{
    return p1.world < p2.world;
}

std::string str(const Possibility& p)
{
    std::string desc = "[ ] Referent System:\n" + str(*p.referentSystem);
    desc += "[ ] Assignment function: \n";

    if (p.assignment.empty()) {
        desc += "  [ empty ]\n";

    }
    else {
        for (const auto& [peg, individual] : p.assignment) {
            desc += "  - peg_" + std::to_string(peg) + " -> e_" + std::to_string(individual) + "\n";
        }
    }

    desc += "[ ] Possible world: w_" + std::to_string(p.world) + "\n";

    return desc;
}

}
```

6.17 referent_system.cpp File Reference

```
#include "referent_system.hpp"
#include <algorithm>
#include <stdexcept>
```

Namespaces

- namespace iif_sadaf
- namespace iif_sadaf::talk
- namespace iif_sadaf::talk::GSV

Functions

- std::string iif_sadaf::talk::GSV::str (const ReferentSystem &r)
- bool iif_sadaf::talk::GSV::extends (const ReferentSystem &r2, const ReferentSystem &r1)

  Determines whether one ReferentSystem extends another.

6.18 referent_system.cpp

Go to the documentation of this file.

```cpp
#include "referent_system.hpp"

#include <algorithm>
#include <stdexcept>

namespace iif_sadaf::talk::GSV {

namespace {

std::set<std::string_view> domain(const ReferentSystem& r)
{
    std::set<std::string_view> domain;
    for (const auto& [variable, peg] : r.variablePegAssociation) {
        domain.insert(variable);
    }

    return domain;
}

} // ANONYMOUS NAMESPACE

ReferentSystem::ReferentSystem(const ReferentSystem& other)
    : pegs(other.pegs)
    , variablePegAssociation(other.variablePegAssociation)
{ }

ReferentSystem& ReferentSystem::operator=(const ReferentSystem& other)
{
    if (this != &other) {
        this->pegs = other.pegs;
        this->variablePegAssociation = other.variablePegAssociation;
    }

    return *this;
}

ReferentSystem::ReferentSystem(ReferentSystem&& other) noexcept
    : pegs(other.pegs)
    , variablePegAssociation(std::move(other.variablePegAssociation))
{ }

ReferentSystem& ReferentSystem::operator=(ReferentSystem&& other) noexcept
{
    if (this != &other) {
        this->pegs = other.pegs;
        this->variablePegAssociation = std::move(other.variablePegAssociation);
        other.pegs = 0;
    }
    return *this;
}

int ReferentSystem::value(std::string_view variable) const
{
    if (!variablePegAssociation.contains(variable)) {
        std::string error_msg = "Variable " + std::string(variable) + " has no anaphoric antecedent of binding quantifier";
        throw(std::out_of_range(error_msg));
    }

    return variablePegAssociation.at(variable);
}

std::string str(const ReferentSystem& r)
{
    std::string desc = "Number of pegs: " + std::to_string(r.pegs) + "\n";
    desc += "Variable to peg association:\n";

    if (r.variablePegAssociation.empty()) {
        desc += "  [ empty ]\n";
        return desc;
    }

    for (const auto& [variable, peg] : r.variablePegAssociation) {
        desc += "  - " + std::string(variable) + " -> peg_" + std::to_string(peg) + "\n";
    }

    return desc;
}

bool extends(const ReferentSystem& r2, const ReferentSystem& r1)
{
    if (r1.pegs > r2.pegs) {
        return false;
    }

    std::set<std::string_view> domain_r1 = domain(r1);
    std::set<std::string_view> domain_r2 = domain(r2);

    if (!std::ranges::includes(domain_r2, domain_r1)) {
        return false;
    }

    const auto old_var_same_or_new_peg = [&](std::string_view variable) -> bool {
        return r1.value(variable) == r2.value(variable) || r1.pegs <= r2.value(variable);
    };

    if (!std::ranges::all_of(domain_r1, old_var_same_or_new_peg)) {
        return false;
    }

    const auto new_var_new_peg = [&](std::string_view variable) -> bool {
        return domain_r1.contains(variable) || r1.pegs <= r2.value(variable);
    };

    return std::ranges::all_of(domain_r2, new_var_new_peg);
}

}
```

Index

∼IModel
    iif_sadaf::talk::GSV::IModel, 19

assignment
    iif_sadaf::talk::GSV::Possibility, 22

create
    iif_sadaf::talk::GSV, 8

domain_cardinality
    iif_sadaf::talk::GSV::IModel, 19

evaluator.cpp, 29, 30
evaluator.hpp, 25
extends
    iif_sadaf::talk::GSV, 8, 9

iif_sadaf, 7
iif_sadaf::talk, 7
iif_sadaf::talk::GSV, 7
    create, 8
    extends, 8, 9
    InformationState, 8
    isDescendantOf, 11
    operator<, 11
    str, 11, 12
    subsistsIn, 12
    update, 12
iif_sadaf::talk::GSV::Evaluator, 15
    operator(), 16–18
iif_sadaf::talk::GSV::IModel, 18
    ∼IModel, 19
    domain_cardinality, 19
    predicateInterpretation, 19
    termInterpretation, 19
    world_cardinality, 19
iif_sadaf::talk::GSV::Possibility, 20
    assignment, 22
    operator=, 21
    Possibility, 20, 21
    referentSystem, 22
    update, 21
    world, 22
iif_sadaf::talk::GSV::ReferentSystem, 22
    operator=, 23
    pegs, 24
    ReferentSystem, 23
    value, 23
    variablePegAssociation, 24
imodel.hpp, 26
information_state.cpp, 33
information_state.hpp, 26, 27
InformationState
    iif_sadaf::talk::GSV, 8
isDescendantOf
    iif_sadaf::talk::GSV, 11

operator<
    iif_sadaf::talk::GSV, 11
operator()
    iif_sadaf::talk::GSV::Evaluator, 16–18
operator=
    iif_sadaf::talk::GSV::Possibility, 21
    iif_sadaf::talk::GSV::ReferentSystem, 23

pegs
    iif_sadaf::talk::GSV::ReferentSystem, 24
Possibility
    iif_sadaf::talk::GSV::Possibility, 20, 21
possibility.cpp, 34, 35
possibility.hpp, 27, 28
predicateInterpretation
    iif_sadaf::talk::GSV::IModel, 19

referent_system.cpp, 36, 37
referent_system.hpp, 28, 29
ReferentSystem
    iif_sadaf::talk::GSV::ReferentSystem, 23
referentSystem
    iif_sadaf::talk::GSV::Possibility, 22

str
    iif_sadaf::talk::GSV, 11, 12
subsistsIn
    iif_sadaf::talk::GSV, 12

termInterpretation
    iif_sadaf::talk::GSV::IModel, 19

update
    iif_sadaf::talk::GSV, 12
    iif_sadaf::talk::GSV::Possibility, 21

value
    iif_sadaf::talk::GSV::ReferentSystem, 23
variablePegAssociation
    iif_sadaf::talk::GSV::ReferentSystem, 24

world
    iif_sadaf::talk::GSV::Possibility, 22
world_cardinality
    iif_sadaf::talk::GSV::IModel, 19