GSV evaluator library

0.1

Generated by Doxygen 1.13.2

Chapter 1

Namespace Index

1.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 2

Class Index

2.1 Class List

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

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

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 evaluate (const Expression &expr, const InformationState &input_state, const IModel &model)

  Evaluates an expression within a given information state and model.
- bool consistent (const Expression &expr, const InformationState &state, const IModel &model)

  Checks if an expression is consistent with a given information state, relative to a model.
- bool allows (const InformationState &state, const Expression &expr, const IModel &model)

  Determines whether an information state allows a given expression.
- bool supports (const InformationState &state, const Expression &expr, const IModel &model)

  Checks if an information state supports a given expression.
- bool isSupportedBy (const Expression &expr, const InformationState &state, const IModel &model)

  Determines whether an expression is supported by an information state.
- bool consistent (const Expression &expr, const IModel &model)

  Determines whether an expression is consistent relative to a given model.
- bool coherent (const Expression &expr, const IModel &model)

  Determines whether an expression is coherent relative to a given model.
- bool entails (const std::vector< Expression > &premises, const Expression &conclusion, const IModel &model)

  Determines whether a set of premises entails a given conclusion, relative to a given model.
- bool equivalent (const Expression &expr1, const Expression &expr2, const IModel &model)

  Determines whether two expressions are equivalent, relative to a given model.
- InformationState create (const IModel &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)
- std::set< std::string_view > domain (const ReferentSystem &r)
- 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 allows()

```
bool iif_sadaf::talk::GSV::allows (
            const InformationState & state,
            const Expression & expr,
            const IModel & model)
```

Determines whether an information state allows a given expression.

This function is a direct alias for `consistent`, checking if the expression is compatible with the given state, relative to the provided model.

Parameters

state	The information state.
expr	The expression to check.
model	The model providing evaluation context.

Returns

`true` if the expression is allowed in the given state and model, otherwise `false`.

Exceptions

`std::invalid_argument'	if `expr` contains an invalid logical operator or quantifier.

Definition at line 54 of file semantic_relations.cpp.

4.3.2.2 coherent()

```
bool iif_sadaf::talk::GSV::coherent (
            const Expression & expr,
            const IModel & model)
```

Determines whether an expression is coherent relative to a given model.

This function iterates over all possible information states given the base model, and checks if at least one of them both supports the given expression and is non-empty. If the expression is not supported in any such information state, the function returns `false`.

Parameters

expr	The expression to check for coherence.
model	The model providing the world structure for evaluation.

Returns

`true` if the expression is coherent across all worlds, otherwise `false`.

Exceptions

`std::invalid_argument'	if `expr` contains an invalid logical operator or quantifier.

Definition at line 185 of file semantic_relations.cpp.

4.3.2.3 consistent() [1/2]

```
bool iif_sadaf::talk::GSV::consistent (
            const Expression & expr,
            const IModel & model)
```

Determines whether an expression is consistent relative to a given model.

This function iterates over all possible information states given the base model, and checks if at least one of them is consistent with the given expression. If the expression is inconsistent in all such information states, the function returns `false`.

Parameters

expr	The expression to check for consistency.
model	The model providing the world structure for evaluation.

Returns

`true` if the expression is consistent across all worlds, otherwise `false`.

Exceptions

`std::invalid_argument'	if `expr` contains an invalid logical operator or quantifier.

Definition at line 161 of file semantic_relations.cpp.

4.3.2.4 consistent() [2/2]

```
bool iif_sadaf::talk::GSV::consistent (
            const Expression & expr,
            const InformationState & state,
            const IModel & model)
```

Checks if an expression is consistent with a given information state, relative to a model.

This function evaluates the expression with respect to the provided state. If the evaluation does not result in an empty information state, the expression is considered consistent. If an out of range exception occurs during evaluation (signaling that the update does not exist, or is not defined, for the input formula), the expression is deemed inconsistent.

Parameters

expr	The expression to check for consistency.
state	The information state used for evaluation.
model	The model providing contextual information for evaluation.

Returns

`true` if the expression is consistent with the state and model, otherwise `false`.

Exceptions

`std::invalid_argument'	if `expr` contains an invalid logical operator or quantifier.

Definition at line 32 of file semantic_relations.cpp.

4.3.2.5 create()

```
InformationState iif_sadaf::talk::GSV::create (
            const IModel & 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.

4.3.2.6 domain()

```
std::set< std::string_view > iif_sadaf::talk::GSV::domain (
            const ReferentSystem & r)
```

Definition at line 8 of file referent_system.cpp.

4.3.2.7 entails()

```
bool iif_sadaf::talk::GSV::entails (
            const std::vector< Expression > & premises,
            const Expression & conclusion,
            const IModel & model)
```

Determines whether a set of premises entails a given conclusion, relative to a given model.

This function iterates over all possible information states relative to the base model, and checks whether applying the premises results in a state that supports the conclusion. If in any world the premises fail to support the conclusion, entailment fails.

Parameters

premises	A vector of expressions representing the premises.
conclusion	The expression representing the conclusion.
model	The model in which entailment is evaluated.

Returns

`true` if the premises entail the conclusion in all worlds, otherwise `false`.

Exceptions

`std::invalid_argument'	if `expr` contains an invalid logical operator or quantifier.

Definition at line 209 of file semantic_relations.cpp.

4.3.2.8 equivalent()

```
bool iif_sadaf::talk::GSV::equivalent (
            const Expression & expr1,
            const Expression & expr2,
            const IModel & model)
```

Determines whether two expressions are equivalent, relative to a given model.

Two expressions are considered equivalent if, in every possible information state, relative to the base model, their evaluation results in similar information states (under the GSV definition of similarity).

Parameters

expr1	The first expression to compare.
expr2	The second expression to compare.
model	The model in which equivalence is evaluated.

Returns

`true` if the expressions are equivalent in all worlds, otherwise `false`.

Exceptions

`std::invalid_argument'	if `expr1` or 'expr2' contain an invalid logical operator or quantifier.

Definition at line 293 of file semantic_relations.cpp.

4.3.2.9 evaluate()

```
InformationState iif_sadaf::talk::GSV::evaluate (
            const Expression & expr,
            const InformationState & input_state,
            const IModel & model)
```

Evaluates an expression within a given information state and model.

This function takes as input an 'InformationState' object, applies an `Evaluator` visitor to the input `Expression`, and returns the output `InformationState`, given the semantic values provided by the 'Model' object. It utilizes `std::visit` to dynamically apply the appropriate evaluation logic based on the type of `expr`.

Parameters

expr	The expression to evaluate.
input_state	The initial information state used during evaluation.
model	The model that provides contextual information for evaluation.

Returns

The resulting `InformationState` after evaluating the expression.

Exceptions

`std::invalid_argument',if	formula does not accord with GSV grammar
`std::out_of_range'	if interpretation is undefined

Definition at line 284 of file evaluator.cpp.

4.3.2.10 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.

4.3.2.11 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 60 of file possibility.cpp.

4.3.2.12 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 64 of file referent_system.cpp.

4.3.2.13 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.

4.3.2.14 isSupportedBy()

```
bool iif_sadaf::talk::GSV::isSupportedBy (
            const Expression & expr,
            const InformationState & state,
            const IModel & model)
```

Determines whether an expression is supported by an information state.

This function is an alias for `supports`, checking if the given state provides support for the specified expression.

Parameters

expr	The expression being tested for support.
state	The information state to check against.
model	The model providing evaluation context.

Returns

`true` if the expression is supported by the state and model, otherwise `false`.

Exceptions

`std::invalid_argument'	if `expr` contains an invalid logical operator or quantifier.

Definition at line 96 of file semantic_relations.cpp.

4.3.2.15 operator<()

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

Definition at line 71 of file possibility.cpp.

4.3.2.16 str() [1/3]

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

Definition at line 133 of file information_state.cpp.

4.3.2.17 str() [2/3]

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

Definition at line 76 of file possibility.cpp.

4.3.2.18 str() [3/3]

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

Definition at line 92 of file referent_system.cpp.

4.3.2.19 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.

4.3.2.20 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.

4.3.2.21 supports()

```
bool iif_sadaf::talk::GSV::supports (
            const InformationState & state,
            const Expression & expr,
            const IModel & model)
```

Checks if an information state supports a given expression.

The function evaluates the expression with respect to the input state, relative to the provided model. If the output information state subsists in the original state, the expression is considered supported. If an out of range exception occurs during evaluation (signaling that the update does not exist, or is not defined, for the input formula), the expression is not supported by the input state.

Parameters

state	The information state to check.
expr	The expression being tested for support.
model	The model providing evaluation context.

Returns

`true` if the expression is supported by the state and model, otherwise `false`.

Exceptions

`std::invalid_argument'	if `expr` contains an invalid logical operator or quantifier.

Definition at line 74 of file semantic_relations.cpp.

4.3.2.22 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() (const std::shared_ptr< UnaryNode > &expr, std::variant< std::pair< InformationState, const IModel * > > params) const

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

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

  Evaluates a quantified expression on an InformationState.
- InformationState operator() (const 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() (const 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() (
            const 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 79 of file evaluator.cpp.

5.1.2.2 operator()() [2/5]

```
InformationState iif_sadaf::talk::GSV::Evaluator::operator() (
            const 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 215 of file evaluator.cpp.

5.1.2.3 operator()() [3/5]

```
InformationState iif_sadaf::talk::GSV::Evaluator::operator() (
            const 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 247 of file evaluator.cpp.

5.1.2.4 operator()() [4/5]

```
InformationState iif_sadaf::talk::GSV::Evaluator::operator() (
            const 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 145 of file evaluator.cpp.

5.1.2.5 operator()() [5/5]

```
InformationState iif_sadaf::talk::GSV::Evaluator::operator() (
            const 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 43 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)=default
- Possibility & operator= (const Possibility &other)=default
- 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.

The `Possibility` class models possiblities in the GSV framework, which are defined as tuples of a referent system, an assignment if individuals to pegs, and a possible world index.

This class supports copy and move semantics, allowing for efficient duplication and transfer of instances.

Definition at line 21 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)  [default]
```

5.3.2.3 Possibility() [3/3]

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

Definition at line 13 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)  [default]
```

5.3.3.2 operator=() [2/2]

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

Definition at line 19 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 39 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 32 of file possibility.hpp.

5.3.4.2 referentSystem

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

Definition at line 31 of file possibility.hpp.

5.3.4.3 world

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

Definition at line 33 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)=default
- ReferentSystem & operator= (const ReferentSystem &other)=default
- 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 provides a framework for handling variable-to-integer mappings within GAV. It allows for retrieval of variable values and tracks the number of pegs (or reference points) within the system.

This class supports both copy and move semantics, ensuring flexibility in managing instances efficiently.

Definition at line 20 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)  [default]
```

5.4.2.3 ReferentSystem() [3/3]

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

Definition at line 18 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)  [default]
```

5.4.3.2 operator=() [2/2]

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

Definition at line 23 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 40 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 30 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 31 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

Functions

- InformationState iif_sadaf::talk::GSV::evaluate (const Expression &expr, const InformationState &input_state, const IModel &model)

  Evaluates an expression within a given information state and model.

6.2 evaluator.hpp

Go to the documentation of this file.

```
00001 #pragma once
00002
00003 #include "expression.hpp"
00004 #include "information_state.hpp"
00005
00006 namespace iif_sadaf::talk::GSV {
00007
00023 struct Evaluator {
00024     InformationState operator()(const std::shared_ptr<UnaryNode>& expr,
      std::variant<std::pair<InformationState, const IModel*>» params) const;
00025     InformationState operator()(const std::shared_ptr<BinaryNode>& expr,
      std::variant<std::pair<InformationState, const IModel*>» params) const;
00026     InformationState operator()(const std::shared_ptr<QuantificationNode>& expr,
      std::variant<std::pair<InformationState, const IModel*>» params) const;
00027     InformationState operator()(const std::shared_ptr<IdentityNode>& expr,
      std::variant<std::pair<InformationState, const IModel*>» params) const;
00028     InformationState operator()(const std::shared_ptr<PredicationNode>& expr,
      std::variant<std::pair<InformationState, const IModel*>» params) const;
00029 };
00030
00031 InformationState evaluate(const Expression& expr, const InformationState& input_state, const IModel&
      model);
00032
00033 }
```

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.

```
00001 #pragma once
00002
00003 #include <set>
00004 #include <string_view>
00005 #include <vector>
00006
00007 namespace iif_sadaf::talk::GSV {
00008
00021 struct IModel {
00022 public:
00023     virtual int world_cardinality() const = 0;
00024     virtual int domain_cardinality() const = 0;
00025     virtual int termInterpretation(std::string_view term, int world) const = 0;
00026     virtual const std::set<std::vector<int>»& predicateInterpretation(std::string_view predicate, int
      world) const = 0;
00027     virtual ~IModel() {};
00028 };
00029
00030 }
```

6.5 semantic_relations.hpp File Reference

```
#include <vector>
#include "expression.hpp"
#include "information_state.hpp"
#include "imodel.hpp"
```

Namespaces

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

Functions

- bool iif_sadaf::talk::GSV::consistent (const Expression &expr, const InformationState &state, const IModel &model)

  Checks if an expression is consistent with a given information state, relative to a model.
- bool iif_sadaf::talk::GSV::allows (const InformationState &state, const Expression &expr, const IModel &model)

  Determines whether an information state allows a given expression.
- bool iif_sadaf::talk::GSV::supports (const InformationState &state, const Expression &expr, const IModel &model)

  Checks if an information state supports a given expression.
- bool iif_sadaf::talk::GSV::isSupportedBy (const Expression &expr, const InformationState &state, const IModel &model)

  Determines whether an expression is supported by an information state.
- bool iif_sadaf::talk::GSV::consistent (const Expression &expr, const IModel &model)

  Determines whether an expression is consistent relative to a given model.
- bool iif_sadaf::talk::GSV::coherent (const Expression &expr, const IModel &model)

  Determines whether an expression is coherent relative to a given model.
- bool iif_sadaf::talk::GSV::entails (const std::vector< Expression > &premises, const Expression &conclusion, const IModel &model)

  Determines whether a set of premises entails a given conclusion, relative to a given model.
- bool iif_sadaf::talk::GSV::equivalent (const Expression &expr1, const Expression &expr2, const IModel &model)

  Determines whether two expressions are equivalent, relative to a given model.

6.6 semantic_relations.hpp

Go to the documentation of this file.

```
00001 #pragma once
00002
00003 #include <vector>
00004
00005 #include "expression.hpp"
00006 #include "information_state.hpp"
00007 #include "imodel.hpp"
00008
00009 namespace iif_sadaf::talk::GSV {
00010
00011 bool consistent(const Expression& expr, const InformationState& state, const IModel& model);
00012 bool allows(const InformationState& state, const Expression& expr, const IModel& model);
```

```
00013 bool supports(const InformationState& state, const Expression& expr, const IModel& model);
00014 bool isSupportedBy(const Expression& expr, const InformationState& state, const IModel& model);
00015
00016 bool consistent(const Expression& expr, const IModel& model);
00017 bool coherent(const Expression& expr, const IModel& model);
00018 bool entails(const std::vector<Expression>& premises, const Expression& conclusion, const IModel&
      model);
00019 bool equivalent(const Expression& expr1, const Expression& expr2, const IModel& model);
00020
00021 }
```

6.7 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 IModel &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)

6.8 information_state.hpp

Go to the documentation of this file.

```
00001 #pragma once
00002
00003 #include <set>
00004 #include <string>
00005 #include <string_view>
00006
00007 #include "model.hpp"
00008 #include "possibility.hpp"
00009
00010 namespace iif_sadaf::talk::GSV {
00011
00015 using InformationState = std::set<Possibility>;
00016
00017 InformationState create(const IModel& model);
00018 InformationState update(const InformationState& input_state, std::string_view variable, int
      individual);
00019 bool extends(const InformationState& s2, const InformationState& s1);
00020
00021 bool isDescendantOf(const Possibility& p2, const Possibility& p1, const InformationState& s);
00022 bool subsistsIn(const Possibility& p, const InformationState& s);
00023 bool subsistsIn(const InformationState& s1, const InformationState& s2);
00024
00025 std::string str(const InformationState& state);
00026 }
```

6.9 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)

6.10 possibility.hpp

Go to the documentation of this file.

```
00001 #pragma once
00002
00003 #include <memory>
00004 #include <string>
00005 #include <unordered_map>
00006
00007 #include "referent_system.hpp"
00008
00009 namespace iif_sadaf::talk::GSV {
00010
00021 struct Possibility {
00022 public:
00023     Possibility(std::shared_ptr<ReferentSystem> r_system, int world);
00024     Possibility(const Possibility& other) = default;
00025     Possibility& operator=(const Possibility& other) = default;
00026     Possibility(Possibility&& other) noexcept;
00027     Possibility& operator=(Possibility&& other) noexcept;
00028
00029     void update(std::string_view variable, int individual);
00030
00031     std::shared_ptr<ReferentSystem> referentSystem;
00032     std::unordered_map<int, int> assignment;
00033     int world;
00034 };
00035
00036 bool extends(const Possibility& p2, const Possibility& p1);
00037 bool operator<(const Possibility& p1, const Possibility& p2);
00038
00039 std::string str(const Possibility& p);
00040
00041 }
```

6.11 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

- std::set< std::string_view > iif_sadaf::talk::GSV::domain (const ReferentSystem &r)
- 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)

6.12 referent_system.hpp

Go to the documentation of this file.

```
00001 #pragma once
00002
00003 #include <set>
00004 #include <string>
00005 #include <string_view>
00006 #include <unordered_map>
00007
00008 namespace iif_sadaf::talk::GSV {
00009
00020 struct ReferentSystem {
00021 public:
00022     ReferentSystem() = default;
00023     ReferentSystem(const ReferentSystem& other) = default;
00024     ReferentSystem& operator=(const ReferentSystem& other) = default;
00025     ReferentSystem(ReferentSystem&& other) noexcept;
00026     ReferentSystem& operator=(ReferentSystem&& other) noexcept;
00027
00028     int value(std::string_view variable) const;
00029
00030     int pegs = 0;
00031     std::unordered_map<std::string_view, int> variablePegAssociation = {};
00032 };
00033
00034 std::set<std::string_view> domain(const ReferentSystem& r);
00035 bool extends(const ReferentSystem& r2, const ReferentSystem& r1);
00036 std::string str(const ReferentSystem& r);
00037
00038 }
```

6.13 evaluator.cpp File Reference

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

Namespaces

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

Functions

- InformationState iif_sadaf::talk::GSV::evaluate (const Expression &expr, const InformationState &input_state, const IModel &model)

  Evaluates an expression within a given information state and model.

6.14 evaluator.cpp

Go to the documentation of this file.

```
00001 #include "evaluator.hpp"
00002
00003 #include <algorithm>
00004 #include <functional>
00005 #include <ranges>
00006 #include <stdexcept>
00007
00008 #include "variable.hpp"
00009
00010 namespace iif_sadaf::talk::GSV {
00011
00012 namespace {
00013
00014 void filter(InformationState& state, const std::function<bool(const Possibility&)>& predicate) {
00015     for (auto it = state.begin(); it != state.end(); ) {
00016         if (!predicate(*it)) {
00017             it = state.erase(it);
00018         }
00019         else {
00020             ++it;
00021         }
00022     }
00023 }
00024
00025 int variableDenotation(std::string_view variable, const Possibility& p)
00026 {
00027     return p.assignment.at(p.referentSystem->value(variable));
00028 }
00029
00030 } // ANONYMOUS NAMESPACE
00031
00043 InformationState Evaluator::operator()(const std::shared_ptr<UnaryNode>& expr,
      std::variant<std::pair<InformationState, const IModel*>> params) const
00044 {
00045     InformationState hypothetical_update = std::visit(Evaluator(), expr->scope, params);
00046     InformationState& input_state = (std::get<std::pair<InformationState, const
      IModel*>>(params)).first;
00047
00048     if (expr->op == Operator::E_POS) {
00049         if (hypothetical_update.empty()) {
00050             input_state.clear();
00051         }
00052     }
00053     else if (expr->op == Operator::E_NEC) {
00054         if (!subsistsIn(input_state, hypothetical_update)) {
00055             input_state.clear();
00056         }
00057     }
00058     else if (expr->op == Operator::NEG) {
00059         filter(input_state, [&](const Possibility& p) -> bool { return !subsistsIn(p,
      hypothetical_update); });
00060     }
00061     else {
00062         throw(std::invalid_argument("Invalid operator for unary formula"));
00063     }
00064
00065     return std::move(input_state);
00066 }
00067
00079 InformationState Evaluator::operator()(const std::shared_ptr<BinaryNode>& expr,
      std::variant<std::pair<InformationState, const IModel*>> params) const
00080 {
00081     const IModel* model = (std::get<std::pair<InformationState, const IModel*>>(params)).second;
00082
00083     if (expr->op == Operator::CON) {
00084         return std::visit(
00085             Evaluator(),
00086             expr->rhs,
00087             std::variant<std::pair<InformationState, const
      IModel*>>(std::make_pair(std::visit(Evaluator(), expr->lhs, params), model))
00088         );
00089     }
00090
00091     InformationState& input_state = (std::get<std::pair<InformationState, const
      IModel*>>(params)).first;
00092     InformationState hypothetical_update_lhs = std::visit(Evaluator(), expr->lhs, params);
00093
00094     if (expr->op == Operator::DIS) {
00095         InformationState hypothetical_update_rhs = std::visit(
00096             Evaluator(),
00097             expr->rhs,
```

```
00098             std::variant<std::pair<InformationState, const
      IModel*»(std::make_pair(std::visit(Evaluator(), negate(expr->lhs), params), model))
00099         );
00100
00101         const auto in_lhs_or_in_rhs = [&](const Possibility& p) -> bool {
00102             return hypothetical_update_lhs.contains(p) || hypothetical_update_rhs.contains(p);
00103         };
00104
00105         filter(input_state, in_lhs_or_in_rhs);
00106     }
00107     else if (expr->op == Operator::IMP) {
00108         InformationState hypothetical_update_consequent = std::visit(
00109             Evaluator(),
00110             expr->rhs,
00111             std::variant<std::pair<InformationState, const
      IModel*»(std::make_pair(hypothetical_update_lhs, model))
00112         );
00113
00114         auto all_descendants_subsist = [&](const Possibility& p) -> bool {
00115             auto not_descendant_or_subsists = [&](const Possibility& p_star) -> bool {
00116                 return !isDescendantOf(p_star, p, hypothetical_update_lhs) || subsistsIn(p_star,
      hypothetical_update_consequent);
00117             };
00118             return std::ranges::all_of(hypothetical_update_lhs, not_descendant_or_subsists);
00119         };
00120
00121         const auto if_subsists_all_descendants_do = [&](const Possibility& p) -> bool {
00122             return !subsistsIn(p, hypothetical_update_lhs) || all_descendants_subsist(p);
00123         };
00124
00125         filter(input_state, if_subsists_all_descendants_do);
00126     }
00127     else {
00128         throw(std::invalid_argument("Invalid operator for binary formula"));
00129     }
00130
00131     return std::move(input_state);
00132 }
00133
00145 InformationState Evaluator::operator()(const std::shared_ptr<QuantificationNode>& expr,
      std::variant<std::pair<InformationState, const IModel*» params) const
00146 {
00147     InformationState& input_state = (std::get<std::pair<InformationState, const
      IModel*»(params)).first;
00148     const IModel* model = (std::get<std::pair<InformationState, const IModel*»(params)).second;
00149
00150     if (expr->quantifier == Quantifier::EXISTENTIAL) {
00151         std::vector<InformationState> all_state_variants;
00152
00153         for (const int i : std::views::iota(0, model->domain_cardinality())) {
00154             const InformationState s_variant = update(input_state, expr->variable, i);
00155             all_state_variants.push_back(std::visit(
00156                 Evaluator(),
00157                 expr->scope,
00158                 std::variant<std::pair<InformationState, const IModel*»(std::make_pair(s_variant,
      model)))
00159             );
00160         }
00161
00162         InformationState output;
00163         for (const auto& state_variant : all_state_variants) {
00164             for (const auto& p : state_variant) {
00165                 output.insert(p);
00166             }
00167         }
00168
00169         return output;
00170     }
00171
00172     if (expr->quantifier == Quantifier::UNIVERSAL) {
00173         std::vector<InformationState> all_hypothetical_updates;
00174
00175         for (const int d : std::views::iota(0, model->domain_cardinality())) {
00176             const InformationState hypothetical_update = std::visit(
00177                 Evaluator(),
00178                 expr->scope,
00179                 std::variant<std::pair<InformationState, const
      IModel*»(std::make_pair(update(input_state, expr->variable, d), model))
00180             );
00181             all_hypothetical_updates.push_back(hypothetical_update);
00182         }
00183
00184         const auto subsists_in_all_hyp_updates = [&](const Possibility& p) -> bool {
00185             const auto p_subsists_in_hyp_update = [&](const InformationState& hypothetical_update) ->
      bool {
00186                 return subsistsIn(p, hypothetical_update);
00187             };
```

```
00188             return std::ranges::all_of(all_hypothetical_updates, p_subsists_in_hyp_update);
00189         };
00190
00191         filter(input_state, subsists_in_all_hyp_updates);
00192     }
00193     else {
00194         throw(std::invalid_argument("Invalid quantifier"));
00195     }
00196
00197     return std::move(input_state);
00198 }
00199
00215 InformationState Evaluator::operator()(const std::shared_ptr<IdentityNode>& expr,
      std::variant<std::pair<InformationState, const IModel*» params) const
00216 {
00217     InformationState& input_state = (std::get<std::pair<InformationState, const
      IModel*»(params)).first;
00218     const IModel& model = *(std::get<std::pair<InformationState, const IModel*»(params)).second;
00219
00220     auto assigns_same_denotation = [&](const Possibility& p) -> bool {
00221         const int lhs_denotation = isVariable(expr->lhs) ? variableDenotation(expr->lhs, p) :
      model.termInterpretation(expr->lhs, p.world);
00222         const int rhs_denotation = isVariable(expr->lhs) ? variableDenotation(expr->rhs, p) :
      model.termInterpretation(expr->rhs, p.world);
00223         return lhs_denotation == rhs_denotation;
00224     };
00225
00226     filter(input_state, assigns_same_denotation);
00227
00228     return std::move(input_state);
00229 }
00230
00247 InformationState Evaluator::operator()(const std::shared_ptr<PredicationNode>& expr,
      std::variant<std::pair<InformationState, const IModel*» params) const
00248 {
00249     InformationState& input_state = (std::get<std::pair<InformationState, const
      IModel*»(params)).first;
00250     const IModel& model = *(std::get<std::pair<InformationState, const IModel*»(params)).second;
00251
00252     auto tuple_in_extension = [&](const Possibility& p) -> bool {
00253         std::vector<int> tuple;
00254
00255         for (const std::string& argument : expr->arguments) {
00256             const int denotation = isVariable(argument) ? variableDenotation(argument, p) :
      model.termInterpretation(argument, p.world);
00257             tuple.push_back(denotation);
00258         }
00259
00260         return model.predicateInterpretation(expr->predicate, p.world).contains(tuple);
00261     };
00262
00263     filter(input_state, tuple_in_extension);
00264
00265     return std::move(input_state);
00266 }
00267
00284 InformationState evaluate(const Expression& expr, const InformationState& input_state, const IModel&
      model)
00285 {
00286     return std::visit(
00287         Evaluator(),
00288         expr,
00289         std::variant<std::pair<InformationState, const IModel*»(std::make_pair(input_state, &model))
00290     );
00291 }
00292
00293 }
```

6.15 semantic_relations.cpp File Reference

```
#include "semantic_relations.hpp"
#include <algorithm>
#include <functional>
#include <ranges>
#include <stdexcept>
#include <variant>
#include <vector>
#include "evaluator.hpp"
```

```
#include "imodel.hpp"
#include "information_state.hpp"
#include "possibility.hpp"
```

Namespaces

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

Functions

- bool iif_sadaf::talk::GSV::consistent (const Expression &expr, const InformationState &state, const IModel &model)

  Checks if an expression is consistent with a given information state, relative to a model.
- bool iif_sadaf::talk::GSV::allows (const InformationState &state, const Expression &expr, const IModel &model)

  Determines whether an information state allows a given expression.
- bool iif_sadaf::talk::GSV::supports (const InformationState &state, const Expression &expr, const IModel &model)

  Checks if an information state supports a given expression.
- bool iif_sadaf::talk::GSV::isSupportedBy (const Expression &expr, const InformationState &state, const IModel &model)

  Determines whether an expression is supported by an information state.
- bool iif_sadaf::talk::GSV::consistent (const Expression &expr, const IModel &model)

  Determines whether an expression is consistent relative to a given model.
- bool iif_sadaf::talk::GSV::coherent (const Expression &expr, const IModel &model)

  Determines whether an expression is coherent relative to a given model.
- bool iif_sadaf::talk::GSV::entails (const std::vector< Expression > &premises, const Expression &conclusion, const IModel &model)

  Determines whether a set of premises entails a given conclusion, relative to a given model.
- bool iif_sadaf::talk::GSV::equivalent (const Expression &expr1, const Expression &expr2, const IModel &model)

  Determines whether two expressions are equivalent, relative to a given model.

6.16 semantic_relations.cpp

Go to the documentation of this file.

```
00001 #include "semantic_relations.hpp"
00002
00003 #include <algorithm>
00004 #include <functional>
00005 #include <ranges>
00006 #include <stdexcept>
00007 #include <variant>
00008 #include <vector>
00009
00010 #include "evaluator.hpp"
00011 #include "imodel.hpp"
00012 #include "information_state.hpp"
00013 #include "possibility.hpp"
00014
00015 namespace iif_sadaf::talk::GSV {
00016
00032 bool consistent(const Expression& expr, const InformationState& state, const IModel& model)
```

```
00033 {
00034     try {
00035         return !evaluate(expr, state, model).empty();
00036     }
00037     catch (const std::out_of_range&) {
00038         return false;
00039     }
00040 }
00041
00054 bool allows(const InformationState& state, const Expression& expr, const IModel& model)
00055 {
00056     return consistent(expr, state, model);
00057 }
00058
00074 bool supports(const InformationState& state, const Expression& expr, const IModel& model)
00075 {
00076     try {
00077         return subsistsIn(state, evaluate(expr, state, model));
00078     }
00079     catch (const std::out_of_range&) {
00080         return false;
00081     }
00082 }
00083
00096 bool isSupportedBy(const Expression& expr, const InformationState& state, const IModel& model)
00097 {
00098     return supports(state, expr, model);
00099 }
00100
00101 namespace {
00102
00103 std::vector<InformationState> generateSubStates(int n, int k) {
00104     std::vector<InformationState> result;
00105
00106     if (k == 0) {
00107         result.push_back(InformationState());
00108         return result;
00109     }
00110
00111     if (k > n + 1) {
00112         return result;
00113     }
00114
00115     int estimate = 1;
00116     for (int i = 1; i <= k; i++) {
00117         estimate = estimate * (n + 2 - i) / i;
00118     }
00119     result.reserve(estimate);
00120
00121     std::function<void(int, InformationState&)> backtrack =
00122         [&](int start, InformationState& current) {
00123             if (current.size() == k) {
00124                 result.push_back(current);
00125                 return;
00126             }
00127
00128             ReferentSystem r;
00129
00130             for (int i = start; i <= n; ++i) {
00131                 Possibility p(std::make_shared<ReferentSystem>(r), i);
00132                 current.insert(p);
00133
00134                 backtrack(i + 1, current);
00135
00136                 current.erase(p);
00137             }
00138         };
00139
00140     InformationState current;
00141     backtrack(0, current);
00142
00143     return result;
00144 }
00145
00146 } // ANONYMOUS NAMESPACE
00147
00161 bool consistent(const Expression& expr, const IModel& model)
00162 {
00163     for (const int i : std::views::iota(0, model.world_cardinality())) {
00164         std::vector<InformationState> states = generateSubStates(model.world_cardinality() - 1, i);
00165         if (!std::ranges::any_of(states, [&](const InformationState& state) -> bool { return
    consistent(expr, state, model); })) {
00166             return false;
00167         }
00168     }
00169     return true;
00170 }
```

```
00171
00185 bool coherent(const Expression& expr, const IModel& model)
00186 {
00187     for (const int i : std::views::iota(0, model.world_cardinality())) {
00188         std::vector<InformationState> states = generateSubStates(model.world_cardinality() - 1, i);
00189         if (!std::ranges::any_of(states, [&](const InformationState& state) -> bool { return
      !state.empty() && supports(state, expr, model); })) {
00190             return false;
00191         }
00192     }
00193     return true;
00194 }
00195
00209 bool entails(const std::vector<Expression>& premises, const Expression& conclusion, const IModel&
      model)
00210 {
00211     for (const int i : std::views::iota(0, model.world_cardinality())) {
00212         std::vector<InformationState> states = generateSubStates(model.world_cardinality() - 1, i);
00213         for (InformationState& input_state : states) {
00214             try {
00215                 for (const Expression& expr : premises) {
00216                     input_state = evaluate(expr, input_state, model);
00217                 }
00218
00219                 (void)evaluate(conclusion, input_state, model);
00220
00221                 // If we get to this point, update exists, so we check for support
00222
00223                 if (!supports(input_state, conclusion, model)) {
00224                     return false;
00225                 }
00226             }
00227             catch (const std::out_of_range&) {
00228                 ; // If update does not exist, then state does not count against entailment
00229             }
00230         }
00231     }
00232     return true;
00233 }
00234
00235 namespace {
00236
00237 bool similar(const Possibility& p1, const Possibility& p2)
00238 {
00239     const auto have_same_denotation = [&](std::string_view variable) -> bool {
00240         return p1.assignment.at(p1.referentSystem->value(variable)) ==
      p2.assignment.at(p2.referentSystem->value(variable));
00241     };
00242
00243     try {
00244         return p1.world == p2.world
00245             && domain(*p1.referentSystem) == domain(*p2.referentSystem)
00246             && std::ranges::all_of(domain(*p1.referentSystem), have_same_denotation);
00247     }
00248     catch (const std::out_of_range&) {
00249         return false;
00250     }
00251 }
00252
00253 bool similar(const InformationState& s1, const InformationState& s2)
00254 {
00255     const auto has_similar_possibility_in_s2 = [&](const Possibility p) -> bool {
00256         const auto is_similar_to_p = [&](const Possibility p_dash) -> bool {
00257             return similar(p, p_dash);
00258         };
00259         return std::ranges::any_of(s2, is_similar_to_p);
00260     };
00261
00262     const auto has_similar_possibility_in_s1 = [&](const Possibility p) -> bool {
00263         const auto is_similar_to_p = [&](const Possibility p_dash) -> bool {
00264             return similar(p, p_dash);
00265         };
00266         return std::ranges::any_of(s1, is_similar_to_p);
00267     };
00268
00269     try {
00270         return std::ranges::all_of(s1, has_similar_possibility_in_s2)
00271             && std::ranges::all_of(s2, has_similar_possibility_in_s1);
00272     }
00273     catch (const std::out_of_range&) {
00274         return false;
00275     }
00276 }
00277
00278 } // ANONYMOUS NAMESPACE
00279
00293 bool equivalent(const Expression& expr1, const Expression& expr2, const IModel& model)
```

```
00294 {
00295     for (const int i : std::views::iota(0, model.world_cardinality())) {
00296         std::vector<InformationState> states = generateSubStates(model.world_cardinality() - 1, i);
00297 
00298         const auto dissimilar_updates = [&](const InformationState& state) ->bool {
00299             return !similar(evaluate(expr1, state, model), evaluate(expr2, state, model));
00300         };
00301 
00302         try {
00303             if (std::ranges::any_of(states, dissimilar_updates)) {
00304                 return false;
00305             }
00306         }
00307         catch (const std::out_of_range&) {
00308             return false;
00309         }
00310     }
00311 
00312     return true;
00313 }
00314 
00315 }
```

6.17 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 IModel &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)

6.18 information_state.cpp

Go to the documentation of this file.

```
00001 #include "information_state.hpp"
00002
00003 #include <algorithm>
00004 #include <iostream>
00005 #include <memory>
00006
00007 namespace iif_sadaf::talk::GSV {
00008
00018 InformationState create(const IModel& model)
00019 {
00020     std::set<Possibility> possibilities;
00021
00022     auto r_system = std::make_shared<ReferentSystem>();
00023
00024     const int number_of_worlds = model.world_cardinality();
00025     for (int i = 0; i < number_of_worlds; ++i) {
00026         possibilities.emplace(r_system, i);
00027     }
00028
00029     return possibilities;
00030 }
00031
00043 InformationState update(const InformationState& input_state, std::string_view variable, int
      individual)
00044 {
00045     InformationState output_state;
00046
00047     auto r_star = std::make_shared<ReferentSystem>();
00048
00049     for (const auto& p : input_state) {
00050         Possibility p_star(r_star, p.world);
00051         p_star.assignment = p.assignment;
00052         r_star->pegs = p.referentSystem->pegs;
00053         for (const auto& map : p.referentSystem->variablePegAssociation) {
00054             const std::string_view var = map.first;
00055             const int peg = map.second;
00056             r_star->variablePegAssociation[var] = peg;
00057         }
00058
00059         p_star.update(variable, individual);
00060
00061         output_state.insert(p_star);
00062     }
00063
00064     return output_state;
00065 }
00066
00076 bool extends(const InformationState& s2, const InformationState& s1)
00077 {
00078     const auto extends_possibility_in_s1 = [&](const Possibility& p2) -> bool {
00079         const auto is_extended_by_p2 = [&](const Possibility& p1) -> bool {
00080             return extends(p2, p1);
00081         };
00082         return std::ranges::any_of(s1, is_extended_by_p2);
00083     };
00084
00085     return std::ranges::all_of(s2, extends_possibility_in_s1);
00086 }
00087
00098 bool isDescendantOf(const Possibility& p2, const Possibility& p1, const InformationState& s)
00099 {
00100     return s.contains(p2) && (extends(p2, p1));
00101 }
00102
00112 bool subsistsIn(const Possibility& p, const InformationState& s)
00113 {
00114     const auto is_descendant_of_p_in_s = [&](const Possibility& p1) -> bool { return
      isDescendantOf(p1, p, s); };
00115     return std::ranges::any_of(s, is_descendant_of_p_in_s);
00116 }
00117
00127 bool subsistsIn(const InformationState& s1, const InformationState& s2)
00128 {
00129     const auto subsists_in_s2 = [&](const Possibility& p) -> bool { return subsistsIn(p, s2); };
00130     return std::ranges::all_of(s1, subsists_in_s2);
00131 }
00132
00133 std::string str(const InformationState& state)
00134 {
00135     std::string desc;
00136
00137     desc += "--------------------\n";
```

```
00138     for (const Possibility& p : state) {
00139         desc += str(p);
00140         desc += "--------------------\n";
00141     }
00142
00143     desc.pop_back();
00144
00145     return desc;
00146 }
00147
00148 }
```

6.19 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)

6.20 possibility.cpp

Go to the documentation of this file.

```
00001 #include "possibility.hpp"
00002
00003 #include <algorithm>
00004
00005 namespace iif_sadaf::talk::GSV {
00006
00007 Possibility::Possibility(std::shared_ptr<ReferentSystem> r_system, int world)
00008     : referentSystem(r_system)
00009     , assignment({})
00010     , world(world)
00011 { }
00012
00013 Possibility::Possibility(Possibility&& other) noexcept
00014     : referentSystem(std::move(other.referentSystem))
00015     , assignment(std::move(other.assignment))
00016     , world(other.world)
00017 { }
00018
00019 Possibility& Possibility::operator=(Possibility&& other) noexcept
00020 {
00021     if (this != &other) {
00022         this->referentSystem = std::move(other.referentSystem);
00023         this->assignment.clear();
00024         this->assignment.swap(other.assignment);
00025         this->world = other.world;
00026     }
00027     return *this;
00028 }
00029
```

```
00039 void Possibility::update(std::string_view variable, int individual)
00040 {
00041     referentSystem->variablePegAssociation[variable] = ++(referentSystem->pegs);
00042     assignment[referentSystem->pegs] = individual;
00043 }
00044
00045 /*
00046 * NON-MEMBER FUNCTIONS
00047 */
00048
00060 bool extends(const Possibility& p2, const Possibility& p1)
00061 {
00062     const auto peg_is_new_or_maintains_assignment = [&](const std::pair<int, int>& map) -> bool {
00063         const int peg = map.first;
00064
00065         return !p1.assignment.contains(peg) || (p1.assignment.at(peg) == p2.assignment.at(peg));
00066     };
00067
00068     return (p1.world == p2.world) && std::ranges::all_of(p2.assignment,
      peg_is_new_or_maintains_assignment);
00069 }
00070
00071 bool operator<(const Possibility& p1, const Possibility& p2)
00072 {
00073     return p1.world < p2.world;
00074 }
00075
00076 std::string str(const Possibility& p)
00077 {
00078     std::string desc = "[ ] Referent System:\n" + str(*p.referentSystem);
00079     desc += "[ ] Assignment function: \n";
00080
00081     if (p.assignment.empty()) {
00082         desc += "   [ empty ]\n";
00083
00084     }
00085     else {
00086         for (const auto& [peg, individual] : p.assignment) {
00087             desc += "   - peg_" + std::to_string(peg) + " -> e_" + std::to_string(individual) + "\n";
00088         }
00089     }
00090
00091     desc += "[ ] Possible world: w_" + std::to_string(p.world) + "\n";
00092
00093     return desc;
00094 }
00095
00096 }
```

6.21 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::set< std::string_view > iif_sadaf::talk::GSV::domain (const ReferentSystem &r)
- 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)

6.22 referent_system.cpp

Go to the documentation of this file.

```
00001 #include "referent_system.hpp"
00002
00003 #include <algorithm>
00004 #include <stdexcept>
00005
00006 namespace iif_sadaf::talk::GSV {
00007
00008 std::set<std::string_view> domain(const ReferentSystem& r)
00009 {
00010     std::set<std::string_view> domain;
00011     for (const auto& [variable, peg] : r.variablePegAssociation) {
00012         domain.insert(variable);
00013     }
00014
00015     return domain;
00016 }
00017
00018 ReferentSystem::ReferentSystem(ReferentSystem&& other) noexcept
00019     : pegs(other.pegs)
00020     , variablePegAssociation(std::move(other.variablePegAssociation))
00021 { }
00022
00023 ReferentSystem& ReferentSystem::operator=(ReferentSystem&& other) noexcept
00024 {
00025     if (this != &other) {
00026         this->pegs = other.pegs;
00027         this->variablePegAssociation = std::move(other.variablePegAssociation);
00028         other.pegs = 0;
00029     }
00030     return *this;
00031 }
00032
00040 int ReferentSystem::value(std::string_view variable) const
00041 {
00042     if (!variablePegAssociation.contains(variable)) {
00043         const std::string error_msg = "Variable " + std::string(variable) + " has no anaphoric
      antecedent or binding quantifier";
00044         throw(std::out_of_range(error_msg));
00045     }
00046
00047     return variablePegAssociation.at(variable);
00048 }
00049
00064 bool extends(const ReferentSystem& r2, const ReferentSystem& r1)
00065 {
00066     if (r1.pegs > r2.pegs) {
00067         return false;
00068     }
00069
00070     std::set<std::string_view> domain_r1 = domain(r1);
00071     std::set<std::string_view> domain_r2 = domain(r2);
00072
00073     if (!std::ranges::includes(domain_r2, domain_r1)) {
00074         return false;
00075     }
00076
00077     const auto old_var_same_or_new_peg = [&](std::string_view variable) -> bool {
00078         return r1.value(variable) == r2.value(variable) || r1.pegs <= r2.value(variable);
00079     };
00080
00081     if (!std::ranges::all_of(domain_r1, old_var_same_or_new_peg)) {
00082         return false;
00083     }
00084
00085     const auto new_var_new_peg = [&](std::string_view variable) -> bool {
00086         return domain_r1.contains(variable) || r1.pegs <= r2.value(variable);
00087     };
00088
00089     return std::ranges::all_of(domain_r2, new_var_new_peg);
00090 }
00091
00092 std::string str(const ReferentSystem& r)
00093 {
00094     std::string desc = "Number of pegs: " + std::to_string(r.pegs) + "\n";
00095     desc += "Variable to peg association:\n";
00096
00097     if (r.variablePegAssociation.empty()) {
00098         desc += "  [ empty ]\n";
00099         return desc;
00100     }
00101
00102     for (const auto& [variable, peg] : r.variablePegAssociation) {
```

```
00103         desc += "  - " + std::string(variable) + " -> peg_" + std::to_string(peg) + "\n";
00104     }
00105
00106     return desc;
00107 }
00108
00109 }
```

Index