GSV evaluator library

2.0

Generated by Doxygen 1.13.2

1 Namespace Index

1.1 Namespace List

Here is a list of all namespaces with brief descriptions:

2 Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

3 Class Index

3.1 Class List

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

4 File Index

4.1 File List

Here is a list of all files with brief descriptions:

5 Namespace Documentation

5.1 iif_sadaf Namespace Reference

Namespaces

- namespace talk

5.2 iif_sadaf::talk Namespace Reference

Namespaces

- namespace GSV

5.3 iif_sadaf::talk::GSV Namespace Reference

Classes

- struct Evaluator

  Implements the GSV evaluation function for QML formulas.
- struct IModel

  Interface for class representing a model for Quantified Modal Logic.
- struct Possibility

  Represents a possibility as understood in the underlying semantics.
- class QMLModelAdapter

  Adapter class to interface with a QMLModel.
- 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 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)
- std::string repr (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::expected< int, std::string > variableDenotation (std::string_view variable, const Possibility &p)

  Retrieves the denotation of a variable within a given Possibility.
- std::string str (const Possibility &p)
- std::string repr (const Possibility &p)
- std::set< std::string_view > domain (const ReferentSystem &r)

  Retrieves the set of variables in the referent system.
- bool extends (const ReferentSystem &r2, const ReferentSystem &r1)

  Determines whether one ReferentSystem extends another.
- std::string str (const ReferentSystem &r)
- std::string repr (const ReferentSystem &r)
- std::expected< InformationState, std::string > evaluate (const QMLExpression::Expression &expr, const InformationState &input_state, const IModel &model)

  Evaluates a logical expression within a given information state, relative to a base model.
- std::expected< bool, std::string > consistent (const QMLExpression::Expression &expr, const InformationState &state, const IModel &model)

  Determines whether an expression is consistent with a given information state, relative to a base model.
- std::expected< bool, std::string > allows (const InformationState &state, const QMLExpression::Expression &expr, const IModel &model)

  Checks whether an information state allows a given expression.
- std::expected< bool, std::string > supports (const InformationState &state, const QMLExpression::↩ Expression &expr, const IModel &model)

  Determines whether an information state supports a given expression.
- std::expected< bool, std::string > isSupportedBy (const QMLExpression::Expression &expr, const InformationState &state, const IModel &model)

  Checks if an expression is supported by a given information state.
- std::expected< bool, std::string > consistent (const QMLExpression::Expression &expr, const IModel &model)

  Determines whether an expression is consistent within a given model.

- std::expected< bool, std::string > coherent (const QMLExpression::Expression &expr, const IModel &model)

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

  Determines whether a set of premises entails a conclusion, relative to a given model.
- std::expected< bool, std::string > equivalent (const QMLExpression::Expression &expr1, const QMLExpression::Expression &expr2, const IModel &model)

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

5.3.1 Typedef Documentation

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.

5.3.2 Function Documentation

allows()

```
std::expected< bool, std::string > iif_sadaf::talk::GSV::allows (
            const InformationState & state,
            const QMLExpression::Expression & expr,
            const IModel & model)
```

Checks whether an information state allows a given expression.

This function determines if the given expression is consistent with the provided information state and model. It simply delegates to `consistent()`, meaning an expression is "allowed" if it does not result in an empty information state.

Parameters

state	The initial information state.
expr	The expression to evaluate.
model	The model used for evaluation.

Returns

std::expected<bool, std::string> `true` if the expression is consistent with the state, `false` otherwise. Returns an error message if evaluation fails.

Definition at line 68 of file semantic_relations.cpp.

coherent()

```
std::expected< bool, std::string > iif_sadaf::talk::GSV::coherent (
            const QMLExpression::Expression & expr,
            const IModel & model)
```

Determines whether an expression is coherent within a given model.

This function checks if there exists at least one non-empty information state definable with respect to the base model that supports the given expression. It iterates over different possible information states and ensures that at least one state both (1) is not empty and (2) supports the expression.

Parameters

expr	The expression to check for coherence.
model	The model against which the expression is evaluated.

Returns

std::expected<bool, std::string> `true` if the expression is coherent in at least one information state, `false` otherwise. Returns an error message if evaluation fails.

Definition at line 219 of file semantic_relations.cpp.

consistent() [1/2]

```
std::expected< bool, std::string > iif_sadaf::talk::GSV::consistent (
            const QMLExpression::Expression & expr,
            const IModel & model)
```

Determines whether an expression is consistent within a given model.

This function checks if there exists at least one information state definable in terms of the base model where the given expression does not lead to an empty update. It iterates over different possible information states and ensures that at least one state allows a non-empty update of the expression.

Parameters

expr	The expression to check for consistency.
model	The model against which the expression is evaluated.

Returns

std::expected<bool, std::string> `true` if the expression is consistent in at least one information state, `false` otherwise. Returns an error message if evaluation fails.

Definition at line 182 of file semantic_relations.cpp.

consistent() [2/2]

```
std::expected< bool, std::string > iif_sadaf::talk::GSV::consistent (
            const QMLExpression::Expression & expr,
            const InformationState & state,
            const IModel & model)
```

Determines whether an expression is consistent with a given information state, relative to a base model.

This function evaluates the given expression against the provided information state and model. If the evaluation succeeds and results in a non-empty information state, the expression is considered consistent.

Parameters

expr	The expression to evaluate.
state	The initial information state.
model	The model used for evaluation.

Returns

std::expected<bool, std::string> `true` if the expression is consistent (i.e., it does not result in an empty state), `false` otherwise. Returns an error message if evaluation fails.

- If evaluation produces an empty information state, the expression is considered inconsistent.
- If an error occurs during evaluation, the error message is returned instead.

Definition at line 38 of file semantic_relations.cpp.

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.

domain()

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

Retrieves the set of variables in the referent system.

This function extracts all variables present in the given ReferentSystem instance and returns them as a set of std::string_view's.

Parameters

r	The ReferentSystem instance whose variables are being queried.

Returns

std::set<std::string_view> A set containing all variables in the system.

Definition at line 18 of file referent_system.cpp.

entails()

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

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

This function evaluates whether the conclusion follows from the premises in all possible information states. It iterates through subsets of possible worlds and applies updates from each premise to the current information state. The conclusion is then evaluated to check whether it is supported in the updated state.

Parameters

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

Returns

std::expected<bool, std::string> `true` if the conclusion is supported in all states updated by the premises, `false` otherwise. Returns an error message if evaluation fails.

Definition at line 257 of file semantic_relations.cpp.

equivalent()

```
std::expected< bool, std::string > iif_sadaf::talk::GSV::equivalent (
            const QMLExpression::Expression & expr1,
            const QMLExpression::Expression & expr2,
            const IModel & model)
```

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

This function evaluates whether the two expressions always produce similar updates to an information state across all possible subsets of worlds in the model. It iterates through these subsets, applying each expression and comparing their resulting states for similarity.

Parameters

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

Returns

std::expected<bool, std::string> `true` if the expressions always produce similar updates, `false` otherwise. Returns an error message if evaluation fails.

Definition at line 385 of file semantic_relations.cpp.

evaluate()

```
std::expected< InformationState, std::string > iif_sadaf::talk::GSV::evaluate (
            const QMLExpression::Expression & expr,
            const InformationState & input_state,
            const IModel & model)
```

Evaluates a logical expression within a given information state, relative to a base model.

This function applies an `Evaluator` visitor to the provided expression, computing an updated information state based on the evaluation result.

Parameters

expr	The logical expression to evaluate.
input_state	The initial information state in which the expression is evaluated.
model	The model providing the interpretation of terms and predicates.

Returns

std::expected<InformationState, std::string> The updated information state if evaluation is successful, or an error message if evaluation fails.

The function processes the expression using `std::visit`, dispatching to the appropriate `Evaluator` method based on the expression type. If evaluation encounters an error (e.g., an invalid operator or undefined term interpretation), an error message is returned instead of an updated state.

Definition at line 482 of file evaluator.cpp.

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.

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.

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 77 of file referent_system.cpp.

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.

isSupportedBy()

```
std::expected< bool, std::string > iif_sadaf::talk::GSV::isSupportedBy (
            const QMLExpression::Expression & expr,
            const InformationState & state,
            const IModel & model)
```

Checks if an expression is supported by a given information state.

This function is equivalent to `supports(state, expr, model)`, verifying whether the evaluation of the expression does not introduce information absent from the given information state.

Parameters

expr	The expression to evaluate.
state	The initial information state.
model	The model used for evaluation.

Returns

std::expected<bool, std::string> `true` if the expression is supported by the state, `false` otherwise. Returns an error message if evaluation fails.

Definition at line 116 of file semantic_relations.cpp.

operator<()

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

Definition at line 71 of file possibility.cpp.

repr() [1/3]

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

Definition at line 148 of file information_state.cpp.

repr() [2/3]

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

Definition at line 124 of file possibility.cpp.

repr() [3/3]

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

Definition at line 123 of file referent_system.cpp.

str() [1/3]

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

Definition at line 133 of file information_state.cpp.

str() [2/3]

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

Definition at line 104 of file possibility.cpp.

str() [3/3]

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

Definition at line 106 of file referent_system.cpp.

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.

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.

supports()

```
std::expected< bool, std::string > iif_sadaf::talk::GSV::supports (
            const InformationState & state,
            const QMLExpression::Expression & expr,
            const IModel & model)
```

Determines whether an information state supports a given expression.

This function checks if the evaluated update of the given expression subsists in the original information state. An expression is "supported" if its evaluation does not introduce information that is absent from the state.

Parameters

state	The initial information state.
expr	The expression to evaluate.
model	The model used for evaluation.

Returns

std::expected<bool, std::string> `true` if the evaluated update subsists in the initial state, `false` otherwise. Returns an error message if evaluation fails.

Definition at line 86 of file semantic_relations.cpp.

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.

variableDenotation()

```
std::expected< int, std::string > iif_sadaf::talk::GSV::variableDenotation (
            std::string_view variable,
            const Possibility & p)
```

Retrieves the denotation of a variable within a given Possibility.

This function looks up the peg associated with the given variable in the Possibility's ReferentSystem and then retrieves the corresponding individual from the assignment.

Parameters

variable	The name of the variable whose denotation is being retrieved.
p	The Possibility in which the variable is interpreted.

Returns

std::expected<int, std::string>

- If successful, returns the individual assigned to the variable.
- If the variable is not found in the ReferentSystem, returns an error message.
- If the peg retrieved from the ReferentSystem is not found in the assignment, returns an error message.

Definition at line 90 of file possibility.cpp.

6 Class Documentation

6.1 iif_sadaf::talk::GSV::Evaluator Struct Reference

Implements the GSV evaluation function for QML formulas.

```
#include <evaluator.hpp>
```

Public Member Functions

- std::expected< InformationState, std::string > operator() (const std::shared_ptr< QMLExpression::Unary↩ Node > &expr, std::variant< std::pair< InformationState, const IModel * > > params) const

  Evaluates a unary logical expression and updates the information state accordingly.
- std::expected< InformationState, std::string > operator() (const std::shared_ptr< QMLExpression::Binary↩ Node > &expr, std::variant< std::pair< InformationState, const IModel * > > params) const

  Evaluates a binary logical expression and updates the information state accordingly.
- std::expected< InformationState, std::string > operator() (const std::shared_ptr< QMLExpression::↩ QuantificationNode > &expr, std::variant< std::pair< InformationState, const IModel * > > params) const

  Evaluates a quantified logical expression and updates the information state accordingly.
- std::expected< InformationState, std::string > operator() (const std::shared_ptr< QMLExpression::Identity↩ Node > &expr, std::variant< std::pair< InformationState, const IModel * > > params) const

  Evaluates an identity expression and filters the information state accordingly.
- std::expected< InformationState, std::string > operator() (const std::shared_ptr< QMLExpression::↩ PredicationNode > &expr, std::variant< std::pair< InformationState, const IModel * > > params) const

  Evaluates a predication expression and filters the information state accordingly.

6.1.1 Detailed Description

Implements the GSV evaluation function for QML formulas.

The Evaluator struct applies logical operations on `InformationState` objects using the visitor pattern. It also takes an IModel* as parameter.

It evaluates different types of logical expressions, including unary, binary, quantification, identity, and predication nodes. The evaluation modifies or filters the given `InformationState`, based on the logical rules applied, and the semantic information provided by `IModel*`.

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.

Definition at line 24 of file evaluator.hpp.

6.1.2 Member Function Documentation

operator()() [1/5]

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

Evaluates a binary logical expression and updates the information state accordingly.

This function applies binary logical operators (such as conjunction, disjunction, and implication) to an expression and modifies the provided information state based on the result.

Parameters

expr	A shared pointer to a QMLExpression::BinaryNode representing the binary expression.
params	A variant containing a pair of the current InformationState and a pointer to the model (IModel).

Returns

std::expected<InformationState, std::string> The updated information state if evaluation is successful, or an error message if evaluation fails.

The function evaluates the left-hand side (lhs) and right-hand side (rhs) of the binary expression and modifies the information state based on the operator:

- **CONJUNCTION**: The lhs is evaluated first, and the resulting state is then used to evaluate the rhs.
- **DISJUNCTION**: The lhs is negated and evaluated separately, then the rhs is evaluated using the negated lhs state. The final state contains possibilities present in either lhs or rhs.
- **IMPLICATION**: Evaluates the lhs, then checks if every possibility in the lhs has all its descendants subsisting in the rhs update.

If an unrecognized operator is encountered, an error message is returned.

If any evaluation fails at any step, the function returns an error message indicating which part of the formula caused the failure.

Definition at line 123 of file evaluator.cpp.

operator()() [2/5]

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

Evaluates an identity expression and filters the information state accordingly.

This function determines whether two terms (variables or constants) in the given expression denote the same entity within the provided model and information state. It then filters the information state, retaining only those possibilities where the denotations match.

Parameters

expr	A shared pointer to an IdentityNode representing the identity expression.
params	A variant containing a pair of the current InformationState and a pointer to the model (IModel).

Returns

std::expected<InformationState, std::string> The updated information state if evaluation is successful, or an error message if evaluation fails.

The function follows these steps:

1. Extracts the left-hand side (lhs) and right-hand side (rhs) terms from the expression.
2. Determines the denotation of each term:
   - If the term is a variable, its denotation is obtained from the current possibility.
   - If the term is a constant, its interpretation is retrieved from the model.
3. Compares the denotations to check for identity.
4. Filters the information state, retaining only those possibilities where the lhs and rhs have the same denotation.

If a denotation is out of range (e.g., an unbound variable), an error message is returned.

Definition at line 366 of file evaluator.cpp.

operator()() [3/5]

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

Evaluates a predication expression and filters the information state accordingly.

This function checks whether a given predicate holds for a set of terms (variables or constants) in each possibility of the current information state. It retains only those possibilities where the predicate applies to the corresponding denotations.

Parameters

expr	A shared pointer to a PredicationNode representing the predication expression.
params	A variant containing a pair of the current InformationState and a pointer to the model (IModel).

Returns

std::expected<InformationState, std::string> The updated information state if evaluation is successful, or an error message if evaluation fails.

The function performs the following steps:

1. Extracts the arguments of the predicate and determines their denotations:
   - If an argument is a variable, its denotation is obtained from the current possibility.
   - If an argument is a constant, its interpretation is retrieved from the model.
2. Constructs a tuple of these denotations.
3. Checks if the tuple belongs to the extension of the predicate in the given world.
4. Filters the information state, keeping only those possibilities where the predicate holds.

If an argument's denotation is out of range (e.g., an unbound variable) or the predicate interpretation is missing, an error message is returned.

Definition at line 423 of file evaluator.cpp.

operator()() [4/5]

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

Evaluates a quantified logical expression and updates the information state accordingly.

This function processes logical quantification (existential or universal) over a variable, applying the quantifier's scope to all possible values in the model's domain.

Parameters

expr	A shared pointer to the `QuantificationNode` representing the quantified expression.
params	A variant containing the current `InformationState` and a pointer to the `IModel`.

Returns

std::expected<InformationState, std::string> The updated information state after applying quantification, or an error message if evaluation fails.

- **Existential Quantification**: Evaluates the scope of the quantifier for all values in the domain, then merges all resulting information states.
- **Universal Quantification**: Evaluates the scope of the quantifier for all values in the domain and filters the input state, keeping only those possibilities that subsist in all hypothetical updates.
- If an error occurs during evaluation (e.g., invalid quantifier or undefined term), an error message is returned instead of an updated state.

Definition at line 260 of file evaluator.cpp.

operator()() [5/5]

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

Evaluates a unary logical expression and updates the information state accordingly.

This function applies a unary operator (such as necessity, possibility, or negation) to an expression and modifies the provided information state based on the result.

Parameters

expr	A shared pointer to a QMLExpression::UnaryNode representing the unary expression.
params	A variant containing a pair of the current InformationState and a pointer to the model (IModel).

Returns

std::expected<InformationState, std::string> The updated information state if evaluation is successful, or an error message if evaluation fails.

The function first evaluates the prejacent (the inner expression of the unary operator). If evaluation fails, an error message is returned. Otherwise, it applies the appropriate modification to the information state:

- **EPISTEMIC_POSSIBILITY**: If the prejacent state is empty, the input state is cleared.
- **EPISTEMIC_NECESSITY**: If the prejacent state is not contained in the input state, the input state is cleared.
- **NEGATION**: The input state is filtered to remove elements that subsist in the prejacent update.

If an unrecognized operator is encountered, an error message is returned.

Definition at line 56 of file evaluator.cpp.

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

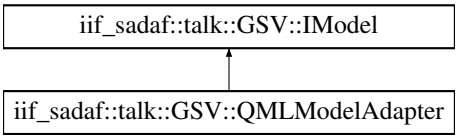
- evaluator.hpp
- evaluator.cpp

6.2 iif_sadaf::talk::GSV::IModel Struct Reference

Interface for class representing a model for Quantified Modal Logic.

`#include <imodel.hpp>`

Inheritance diagram for iif_sadaf::talk::GSV::IModel:



Public Member Functions

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

6.2.1 Detailed Description

Interface for class representing a model for Quantified Modal Logic.

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 the model, the interpretation of a singular term at that world (represented by an `int`), and returns an error message if the term is not interpreted in the model
- a function that retrieves, for any possible world in the model, the interpretation of a predicate at that world (represented by a `std::set<std::vector<int>>`), and returns an error message if the predicate is not interpreted in the model

Definition at line 23 of file imodel.hpp.

6.2.2 Constructor & Destructor Documentation

~IModel()

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

Definition at line 29 of file imodel.hpp.

6.2.3 Member Function Documentation

domain_cardinality()

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

Implemented in iif_sadaf::talk::GSV::QMLModelAdapter.

predicateInterpretation()

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

Implemented in iif_sadaf::talk::GSV::QMLModelAdapter.

termInterpretation()

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

Implemented in iif_sadaf::talk::GSV::QMLModelAdapter.

world_cardinality()

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

Implemented in iif_sadaf::talk::GSV::QMLModelAdapter.

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

- imodel.hpp

6.3 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

6.3.1 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.

Definition at line 19 of file possibility.hpp.

6.3.2 Constructor & Destructor Documentation

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.

Possibility() [2/3]

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

Possibility() [3/3]

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

Definition at line 13 of file possibility.cpp.

6.3.3 Member Function Documentation

operator=() [1/2]

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

operator=() [2/2]

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

Definition at line 19 of file possibility.cpp.

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.

6.3.4 Member Data Documentation

assignment

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

Definition at line 30 of file possibility.hpp.

referentSystem

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

Definition at line 29 of file possibility.hpp.

world

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

Definition at line 31 of file possibility.hpp.

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

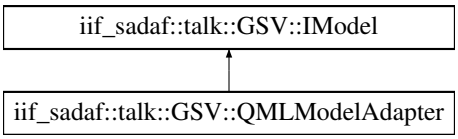
- possibility.hpp
- possibility.cpp

6.4 iif_sadaf::talk::GSV::QMLModelAdapter Class Reference

Adapter class to interface with a QMLModel.

`#include <qml_model_adapter.hpp>`

Inheritance diagram for iif_sadaf::talk::GSV::QMLModelAdapter:



Public Member Functions

- QMLModelAdapter (const QMLModel::QMLModel &qmlModel)
  Constructs an adapter for an existing QMLModel instance.
- QMLModelAdapter (std::unique_ptr< QMLModel::QMLModel > qmlModel)
  Constructs an adapter that takes ownership of a QMLModel instance.
- QMLModelAdapter (const QMLModelAdapter &)=delete
- QMLModelAdapter & operator= (const QMLModelAdapter &)=delete
- QMLModelAdapter (QMLModelAdapter &&) noexcept=default
- QMLModelAdapter & operator= (QMLModelAdapter &&) noexcept=default
- ~QMLModelAdapter () override=default
- int world_cardinality () const override
  Retrieves the cardinality of the model's world.
- int domain_cardinality () const override
  Retrieves the domain cardinality for a given model.
- std::expected< int, std::string > termInterpretation (std::string_view term, int world) const override
  Retrieves the interpretation of a given term within a specified world.
- std::expected< const std::set< std::vector< int > > *, std::string > predicateInterpretation (std::string_view predicate, int world) const override
  Interprets a given predicate within a specified world.

Public Member Functions inherited from iif_sadaf::talk::GSV::IModel

- virtual ~IModel ()

6.4.1 Detailed Description

Adapter class to interface with a QMLModel.

This class adapts a QMLModel to conform to the IModel interface, providing implementations for model-related operations such as retrieving world and domain cardinalities, and the interpretation of terms and predicates.

Definition at line 18 of file qml_model_adapter.hpp.

6.4.2 Constructor & Destructor Documentation

QMLModelAdapter() [1/4]

```
iif_sadaf::talk::GSV::QMLModelAdapter::QMLModelAdapter (
            const QMLModel::QMLModel & QMLModelModel)  [explicit]
```

Constructs an adapter for an existing QMLModel instance.

Parameters

qmlModel	Reference to an existing QMLModel object.

Definition at line 10 of file qml_model_adapter.cpp.

QMLModelAdapter() [2/4]

```
iif_sadaf::talk::GSV::QMLModelAdapter::QMLModelAdapter (
            std::unique_ptr< QMLModel::QMLModel > QMLModelModel)  [explicit]
```

Constructs an adapter that takes ownership of a QMLModel instance.

Parameters

qmlModel	A unique pointer to a QMLModel instance.

Definition at line 18 of file qml_model_adapter.cpp.

QMLModelAdapter() [3/4]

```
iif_sadaf::talk::GSV::QMLModelAdapter::QMLModelAdapter (
            const QMLModelAdapter & )  [delete]
```

QMLModelAdapter() [4/4]

```
iif_sadaf::talk::GSV::QMLModelAdapter::QMLModelAdapter (
            QMLModelAdapter && )  [default], [noexcept]
```

~QMLModelAdapter()

```
iif_sadaf::talk::GSV::QMLModelAdapter::~QMLModelAdapter ()  [override], [default]
```

6.4.3 Member Function Documentation

domain_cardinality()

```
int iif_sadaf::talk::GSV::QMLModelAdapter::domain_cardinality () const  [override], [virtual]
```

Retrieves the domain cardinality for a given model.

Returns

The number of individuals in the domain of the model.

Implements iif_sadaf::talk::GSV::IModel.

Definition at line 36 of file qml_model_adapter.cpp.

operator=() [1/2]

```
QMLModelAdapter & iif_sadaf::talk::GSV::QMLModelAdapter::operator= (
            const QMLModelAdapter & )  [delete]
```

operator=() [2/2]

```
QMLModelAdapter & iif_sadaf::talk::GSV::QMLModelAdapter::operator= (
            QMLModelAdapter && )  [default], [noexcept]
```

predicateInterpretation()

```
std::expected< const std::set< std::vector< int > > *, std::string > iif_sadaf::talk::GSV↩
::QMLModelAdapter::predicateInterpretation (
            std::string_view predicate,
            int world) const  [override], [virtual]
```

Interprets a given predicate within a specified world.

Parameters

predicate	The predicate to be interpreted.
world	The world in which the predicate is interpreted.

Returns

An expected value containing a pointer to a set of vector<int> results or an error message.

Implements iif_sadaf::talk::GSV::IModel.

Definition at line 61 of file qml_model_adapter.cpp.

termInterpretation()

```
std::expected< int, std::string > iif_sadaf::talk::GSV::QMLModelAdapter::termInterpretation (
            std::string_view term,
            int world) const  [override], [virtual]
```

Retrieves the interpretation of a given term within a specified world.

Parameters

term	The term to be interpreted.
world	The world in which the term is interpreted.

Returns

An expected value containing the interpretation result or an error message.

Implements iif_sadaf::talk::GSV::IModel.

Definition at line 48 of file qml_model_adapter.cpp.

world_cardinality()

```
int iif_sadaf::talk::GSV::QMLModelAdapter::world_cardinality () const  [override], [virtual]
```

Retrieves the cardinality of the model's world.

Returns

The number of worlds in the model.

Implements iif_sadaf::talk::GSV::IModel.

Definition at line 26 of file qml_model_adapter.cpp.

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

- qml_model_adapter.hpp
- qml_model_adapter.cpp

6.5 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
- std::expected< int, std::string > value (std::string_view variable) const

  Retrieves the referent value associated with a given variable.

Public Attributes

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

6.5.1 Detailed Description

Represents a referent system for variable assignments.

The `ReferentSystem` class provides a framework for handling variable-to-integer mappings within GSV. It allows for retrieval of variable values and tracks the number of pegs (or reference points) within the system.

Definition at line 18 of file referent_system.hpp.

6.5.2 Constructor & Destructor Documentation

ReferentSystem() [1/3]

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

ReferentSystem() [2/3]

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

ReferentSystem() [3/3]

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

Definition at line 28 of file referent_system.cpp.

6.5.3 Member Function Documentation

operator=() [1/2]

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

operator=() [2/2]

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

Definition at line 33 of file referent_system.cpp.

value()

```
std::expected< int, std::string > iif_sadaf::talk::GSV::ReferentSystem::value (
            std::string_view variable) const
```

Retrieves the referent value associated with a given variable.

This function checks whether the specified variable exists in the referent system. If the variable is found, its corresponding value is returned. Otherwise, an error message is returned indicating that the variable is not present.

Parameters

variable	The variable whose referent value is being queried.

Returns

std::expected<int, std::string> The value associated with the variable, or an error message if the variable does not exist.

Definition at line 54 of file referent_system.cpp.

6.5.4 Member Data Documentation

pegs

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

Definition at line 28 of file referent_system.hpp.

variablePegAssociation

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

Definition at line 29 of file referent_system.hpp.

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

- referent_system.hpp
- referent_system.cpp

7 File Documentation

7.1 adapters.hpp File Reference

```
#include "qml_model_adapters.hpp"
```

7.2 adapters.hpp

Go to the documentation of this file.

```
00001 #pragma once
00002
00003 #include "qml_model_adapters.hpp"
```

7.3 qml_model_adapter.hpp File Reference

```
#include <memory>
#include <QMLModel/qml-model.hpp>
#include "imodel.hpp"
```

Classes

- class iif_sadaf::talk::GSV::QMLModelAdapter

  Adapter class to interface with a QMLModel.

Namespaces

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

7.4 qml_model_adapter.hpp

Go to the documentation of this file.

```
00001 #pragma once
00002
00003 #include <memory>
00004
00005 #include <QMLModel/qml-model.hpp>
00006
00007 #include "imodel.hpp"
00008
00009 namespace iif_sadaf::talk::GSV {
00010
00018 class QMLModelAdapter : public IModel {
00019 public:
00020     explicit QMLModelAdapter(const QMLModel::QMLModel& qmlModel);
00021     explicit QMLModelAdapter(std::unique_ptr<QMLModel::QMLModel> qmlModel);
00022
00023     QMLModelAdapter(const QMLModelAdapter&) = delete;
00024     QMLModelAdapter& operator=(const QMLModelAdapter&) = delete;
00025     QMLModelAdapter(QMLModelAdapter&&) noexcept = default;
00026     QMLModelAdapter& operator=(QMLModelAdapter&&) noexcept = default;
00027     ~QMLModelAdapter() override = default;
00028
00029     int world_cardinality() const override;
00030     int domain_cardinality() const override;
00031     std::expected<int, std::string> termInterpretation(std::string_view term, int world) const
      override;
00032     std::expected<const std::set<std::vector<int>*, std::string>
      predicateInterpretation(std::string_view predicate, int world) const override;
00033
00034 private:
00035     class Impl {
00036     public:
00037         explicit Impl(const QMLModel::QMLModel& model) : ownedModel(nullptr), modelRef(&model) {}
00038         explicit Impl(std::unique_ptr<QMLModel::QMLModel> model)
00039             : ownedModel(std::move(model)), modelRef(ownedModel.get()) {
00040         }
00041         const QMLModel::QMLModel& getModel() const { return *modelRef; }
00042
00043     private:
00044         std::unique_ptr<QMLModel::QMLModel> ownedModel;
00045         const QMLModel::QMLModel* modelRef;
00046     };
00047     std::unique_ptr<Impl> pImpl;
00048 };
00049
00050 }
```

7.5 qml_model_adapter.cpp File Reference

```
#include "qml_model_adapter.hpp"
```

Namespaces

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

7.6 qml_model_adapter.cpp

Go to the documentation of this file.

```
00001 #include "qml_model_adapter.hpp"
00002
00003 namespace iif_sadaf::talk::GSV {
00004
00010 QMLModelAdapter::QMLModelAdapter(const QMLModel::QMLModel& QMLModelModel)
00011     : pImpl(std::make_unique<Impl>(QMLModelModel)) {}
00012
00018 QMLModelAdapter::QMLModelAdapter(std::unique_ptr<QMLModel::QMLModel> QMLModelModel)
00019     : pImpl(std::make_unique<Impl>(std::move(QMLModelModel))) {}
00020
00026 int QMLModelAdapter::world_cardinality() const
00027 {
00028     return pImpl->getModel().world_cardinality();
00029 }
00030
00036 int QMLModelAdapter::domain_cardinality() const
00037 {
00038     return pImpl->getModel().domain_cardinality();
00039 }
00040
00048 std::expected<int, std::string> QMLModelAdapter::termInterpretation(std::string_view term, int world)
      const
00049 {
00050     return pImpl->getModel().termInterpretation(term, world);
00051 }
00052
00061 std::expected<const std::set<std::vector<int>*, std::string>
      QMLModelAdapter::predicateInterpretation(std::string_view predicate, int world) const
00062 {
00063     return pImpl->getModel().predicateInterpretation(predicate, world);
00064 }
00065
00066 }
```

7.7 core.hpp File Reference

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

7.8 core.hpp

Go to the documentation of this file.

```
00001 #pragma once
00002
00003 #include "information_state.hpp"
00004 #include "possibility.hpp"
00005 #include "referent_system.hpp"
```

7.9 information_state.hpp File Reference

```
#include <set>
#include <string>
#include <string_view>
#include "imodel.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)
- std::string iif_sadaf::talk::GSV::repr (const InformationState &state)

7.10 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 "imodel.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 std::string repr(const InformationState& state);
00027
00028 }
```

7.11 possibility.hpp File Reference

```
#include <expected>
#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::expected< int, std::string > iif_sadaf::talk::GSV::variableDenotation (std::string_view variable, const Possibility &p)

  Retrieves the denotation of a variable within a given Possibility.
- std::string iif_sadaf::talk::GSV::str (const Possibility &p)
- std::string iif_sadaf::talk::GSV::repr (const Possibility &p)

7.12 possibility.hpp

Go to the documentation of this file.

```
00001 #pragma once
00002
00003 #include <expected>
00004 #include <memory>
00005 #include <string>
00006 #include <unordered_map>
00007
00008 #include "referent_system.hpp"
00009
00010 namespace iif_sadaf::talk::GSV {
00011
00019 struct Possibility {
00020 public:
00021     Possibility(std::shared_ptr<ReferentSystem> r_system, int world);
00022     Possibility(const Possibility& other) = default;
00023     Possibility& operator=(const Possibility& other) = default;
00024     Possibility(Possibility&& other) noexcept;
00025     Possibility& operator=(Possibility&& other) noexcept;
00026
00027     void update(std::string_view variable, int individual);
00028
00029     std::shared_ptr<ReferentSystem> referentSystem;
00030     std::unordered_map<int, int> assignment;
00031     int world;
00032 };
00033
00034 bool extends(const Possibility& p2, const Possibility& p1);
00035 bool operator<(const Possibility& p1, const Possibility& p2);
00036 std::expected<int, std::string> variableDenotation(std::string_view variable, const Possibility& p);
00037
00038 std::string str(const Possibility& p);
00039 std::string repr(const Possibility& p);
00040
00041 }
```

7.13 referent_system.hpp File Reference

```
#include <expected>
#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)

  Retrieves the set of variables in the referent system.
- 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)
- std::string iif_sadaf::talk::GSV::repr (const ReferentSystem &r)

7.14 referent_system.hpp

Go to the documentation of this file.

```
00001 #pragma once
00002
00003 #include <expected>
00004 #include <set>
00005 #include <string>
00006 #include <string_view>
00007 #include <unordered_map>
00008
00009 namespace iif_sadaf::talk::GSV {
00010
00018 struct ReferentSystem {
00019 public:
00020     ReferentSystem() = default;
00021     ReferentSystem(const ReferentSystem& other) = default;
00022     ReferentSystem& operator=(const ReferentSystem& other) = default;
00023     ReferentSystem(ReferentSystem&& other) noexcept;
00024     ReferentSystem& operator=(ReferentSystem&& other) noexcept;
00025
00026     std::expected<int, std::string> value(std::string_view variable) const;
00027
00028     int pegs = 0;
00029     std::unordered_map<std::string_view, int> variablePegAssociation = {};
00030 };
00031
00032 std::set<std::string_view> domain(const ReferentSystem& r);
00033 bool extends(const ReferentSystem& r2, const ReferentSystem& r1);
00034 std::string str(const ReferentSystem& r);
00035 std::string repr(const ReferentSystem& r);
00036
00037 }
```

7.15 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)
- std::string iif_sadaf::talk::GSV::repr (const InformationState &state)

7.16 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 std::string repr(const InformationState& state)
00149 {
00150     std::string desc = "Information State : [\n";
00151 
00152     for (const Possibility& p : state) {
00153         desc += "  " + repr(p) + "\n";
00154     }
00155 
00156     return desc + "]";
00157 }
00158 
00159 }
```

7.17 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::expected< int, std::string > iif_sadaf::talk::GSV::variableDenotation (std::string_view variable, const Possibility &p)

  Retrieves the denotation of a variable within a given Possibility.
- std::string iif_sadaf::talk::GSV::str (const Possibility &p)
- std::string iif_sadaf::talk::GSV::repr (const Possibility &p)

7.18 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 
00090 std::expected<int, std::string> variableDenotation(std::string_view variable, const Possibility& p)
00091 {
00092     const auto peg = p.referentSystem->value(variable);
00093 
00094     if (!peg.has_value()) {
00095         return std::unexpected(peg.error());
00096     }
00097 
00098     // Whenever variable exists in referent system, assignment is guaranteed to
00099     // contain the corresponding peg, so there is no need to check for existence
00100     // before returning
00101     return p.assignment.at(peg.value());
00102 }
00103 
00104 std::string str(const Possibility& p)
00105 {
00106     std::string desc = "[ ] Referent System:\n" + str(*p.referentSystem);
00107     desc += "[ ] Assignment function: \n";
00108 
00109     if (p.assignment.empty()) {
00110         desc += "   [ empty ]\n";
00111 
00112     }
00113     else {
00114         for (const auto& [peg, individual] : p.assignment) {
00115             desc += "   - peg_" + std::to_string(peg) + " -> e_" + std::to_string(individual) + "\n";
00116         }
00117     }
00118 
00119     desc += "[ ] Possible world: w_" + std::to_string(p.world) + "\n";
00120 
00121     return desc;
00122 }
00123 
00124 std::string repr(const Possibility& p)
00125 {
00126     std::string desc = "Possibility : [ " + repr(*p.referentSystem) + ", Assignment : [ ";
00127 
00128     if (p.assignment.empty()) {
00129         desc += "]";
00130     }
00131     else {
00132         for (const auto& [peg, individual] : p.assignment) {
00133             desc += "{ " + std::to_string(peg) + " : " + std::to_string(individual) + " }, ";
00134         }
00135         desc.resize(desc.size() - 2);
00136         desc += " ]";
00137     }
00138 
00139     desc += ", World : " + std::to_string(p.world) + " ]";
00140 
00141     return desc;
00142 }
00143 
00144 }
```

7.19 referent_system.cpp File Reference

```
#include "referent_system.hpp"
#include <algorithm>
#include <format>
#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)

  Retrieves the set of variables in the referent system.
- 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)
- std::string iif_sadaf::talk::GSV::repr (const ReferentSystem &r)

7.20 referent_system.cpp

Go to the documentation of this file.

```
00001 #include "referent_system.hpp"
00002
00003 #include <algorithm>
00004 #include <format>
00005 #include <stdexcept>
00006
00007 namespace iif_sadaf::talk::GSV {
00008
00018 std::set<std::string_view> domain(const ReferentSystem& r)
00019 {
00020     std::set<std::string_view> domain;
00021     for (const auto& [variable, peg] : r.variablePegAssociation) {
00022         domain.insert(variable);
00023     }
00024
00025     return domain;
00026 }
00027
00028 ReferentSystem::ReferentSystem(ReferentSystem&& other) noexcept
00029     : pegs(other.pegs)
00030     , variablePegAssociation(std::move(other.variablePegAssociation))
00031 { }
00032
00033 ReferentSystem& ReferentSystem::operator=(ReferentSystem&& other) noexcept
00034 {
00035     if (this != &other) {
00036         this->pegs = other.pegs;
00037         this->variablePegAssociation = std::move(other.variablePegAssociation);
00038         other.pegs = 0;
00039     }
00040     return *this;
00041 }
00042
00054 std::expected<int, std::string> ReferentSystem::value(std::string_view variable) const
00055 {
00056     if (!variablePegAssociation.contains(variable)) {
00057         return std::unexpected(std::format("Referent system does not contain variable {}",
      std::string(variable)));
00058     }
00059
00060     return variablePegAssociation.at(variable);
00061 }
00062
00077 bool extends(const ReferentSystem& r2, const ReferentSystem& r1)
00078 {
00079     // TODO check that these calls to value() are safe
00080     if (r1.pegs > r2.pegs) {
00081         return false;
00082     }
00083
00084     std::set<std::string_view> domain_r1 = domain(r1);
00085     std::set<std::string_view> domain_r2 = domain(r2);
00086
00087     if (!std::ranges::includes(domain_r2, domain_r1)) {
```

```
00088         return false;
00089     }
00090
00091     const auto old_var_same_or_new_peg = [&](std::string_view variable) -> bool {
00092         return r1.value(variable).value() == r2.value(variable).value() || r1.pegs <=
      r2.value(variable).value();
00093     };
00094
00095     if (!std::ranges::all_of(domain_r1, old_var_same_or_new_peg)) {
00096         return false;
00097     }
00098
00099     const auto new_var_new_peg = [&](std::string_view variable) -> bool {
00100         return domain_r1.contains(variable) || r1.pegs <= r2.value(variable).value();
00101     };
00102
00103     return std::ranges::all_of(domain_r2, new_var_new_peg);
00104 }
00105
00106 std::string str(const ReferentSystem& r)
00107 {
00108     std::string desc = "Number of pegs: " + std::to_string(r.pegs) + "\n";
00109     desc += "Variable to peg association:\n";
00110
00111     if (r.variablePegAssociation.empty()) {
00112         desc += "  [ empty ]\n";
00113         return desc;
00114     }
00115
00116     for (const auto& [variable, peg] : r.variablePegAssociation) {
00117         desc += "  - " + std::string(variable) + " -> peg_" + std::to_string(peg) + "\n";
00118     }
00119
00120     return desc;
00121 }
00122
00123 std::string repr(const ReferentSystem& r)
00124 {
00125     std::string desc = "R-System : [ ";
00126
00127     if (r.variablePegAssociation.empty()) {
00128         return desc + "]";
00129     }
00130
00131     for (const auto& [variable, peg] : r.variablePegAssociation) {
00132         desc += "{ " + std::string(variable) + " : " + std::to_string(peg) + " }, ";
00133     }
00134
00135     desc.resize(desc.size() - 2);
00136
00137     return desc + " ]";
00138 }
00139
00140 }
```

7.21 evaluator.hpp File Reference

```
#include <expected>
#include <QMLExpression/expression.hpp>
#include "information_state.hpp"
```

Classes

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

  Implements the GSV evaluation function for QML formulas.

Namespaces

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

Functions

- std::expected< InformationState, std::string > iif_sadaf::talk::GSV::evaluate (const QMLExpression::↩ Expression &expr, const InformationState &input_state, const IModel &model)

  Evaluates a logical expression within a given information state, relative to a base model.

7.22 evaluator.hpp

Go to the documentation of this file.

```
00001 #pragma once
00002
00003 #include <expected>
00004
00005 #include <QMLExpression/expression.hpp>
00006
00007 #include "information_state.hpp"
00008
00009 namespace iif_sadaf::talk::GSV {
00010
00024 struct Evaluator {
00025     std::expected<InformationState, std::string> operator()(const
      std::shared_ptr<QMLExpression::UnaryNode>& expr, std::variant<std::pair<InformationState, const
      IModel*>» params) const;
00026     std::expected<InformationState, std::string> operator()(const
      std::shared_ptr<QMLExpression::BinaryNode>& expr, std::variant<std::pair<InformationState, const
      IModel*>» params) const;
00027     std::expected<InformationState, std::string> operator()(const
      std::shared_ptr<QMLExpression::QuantificationNode>& expr, std::variant<std::pair<InformationState,
      const IModel*>» params) const;
00028     std::expected<InformationState, std::string> operator()(const
      std::shared_ptr<QMLExpression::IdentityNode>& expr, std::variant<std::pair<InformationState, const
      IModel*>» params) const;
00029     std::expected<InformationState, std::string> operator()(const
      std::shared_ptr<QMLExpression::PredicationNode>& expr, std::variant<std::pair<InformationState, const
      IModel*>» params) const;
00030 };
00031
00032 std::expected<InformationState, std::string> evaluate(const QMLExpression::Expression& expr, const
      InformationState& input_state, const IModel& model);
00033
00034 }
```

7.23 evaluator.cpp File Reference

```
#include "evaluator.hpp"
#include <algorithm>
#include <expected>
#include <format>
#include <functional>
#include <ranges>
#include <stdexcept>
#include <QMLExpression/formatter.hpp>
#include "possibility.hpp"
```

Namespaces

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

Functions

- std::expected< InformationState, std::string > iif_sadaf::talk::GSV::evaluate (const QMLExpression::↩ Expression &expr, const InformationState &input_state, const IModel &model)

  Evaluates a logical expression within a given information state, relative to a base model.

7.24 evaluator.cpp

Go to the documentation of this file.

```
00001 #include "evaluator.hpp"
00002
00003 #include <algorithm>
00004 #include <expected>
00005 #include <format>
00006 #include <functional>
00007 #include <ranges>
00008 #include <stdexcept>
00009
00010 #include <QMLExpression/formatter.hpp>
00011
00012 #include "possibility.hpp"
00013
00014 namespace iif_sadaf::talk::GSV {
00015
00016 namespace {
00017
00018 void filter(InformationState& state, const std::function<bool(const Possibility&)>& predicate) {
00019     for (auto it = state.begin(); it != state.end(); ) {
00020         if (!predicate(*it)) {
00021             it = state.erase(it);
00022         }
00023         else {
00024             ++it;
00025         }
00026     }
00027 }
00028
00029 QMLExpression::Expression negate(const QMLExpression::Expression& expr)
00030 {
00031     return std::make_shared<QMLExpression::UnaryNode>(QMLExpression::Operator::NEGATION, expr);
00032 }
00033
00034 } // ANONYMOUS NAMESPACE
00035
00056 std::expected<InformationState, std::string> Evaluator::operator()(const
      std::shared_ptr<QMLExpression::UnaryNode>& expr, std::variant<std::pair<InformationState, const
      IModel*>> params) const
00057 {
00058     const auto prejacent_update = std::visit(Evaluator(), expr->scope, params);
00059
00060     if (!prejacent_update.has_value()) {
00061         return std::unexpected(
00062             std::format(
00063                 "In evaluating formula {}:\n{}",
00064                 std::visit(QMLExpression::Formatter(), QMLExpression::Expression(expr)),
00065                 prejacent_update.error()
00066             )
00067         );
00068     }
00069
00070     InformationState& input_state = std::get<std::pair<InformationState, const IModel*>>(params).first;
00071
00072     if (expr->op == QMLExpression::Operator::EPISTEMIC_POSSIBILITY) {
00073         if (prejacent_update.value().empty()) {
00074             input_state.clear();
00075         }
00076     }
00077     else if (expr->op == QMLExpression::Operator::EPISTEMIC_NECESSITY) {
00078         if (!subsistsIn(input_state, prejacent_update.value())) {
00079             input_state.clear();
00080         }
00081     }
00082     else if (expr->op == QMLExpression::Operator::NEGATION) {
00083         filter(input_state, [&](const Possibility& p) -> bool { return !subsistsIn(p,
      prejacent_update.value()); });
00084     }
00085     else {
00086         return std::unexpected(
00087             std::format(
```

```
00088                 "In evaluating formula {}:\n{}",
00089                 std::visit(QMLExpression::Formatter(), QMLExpression::Expression(expr)),
00090                 "Invalid unary operator"
00091             )
00092         );
00093     }
00094
00095     return std::move(input_state);
00096 }
00097
00123 std::expected<InformationState, std::string> Evaluator::operator()(const
      std::shared_ptr<QMLExpression::BinaryNode>& expr, std::variant<std::pair<InformationState, const
      IModel*>> params) const
00124 {
00125     const IModel* model = (std::get<std::pair<InformationState, const IModel*>>(params)).second;
00126
00127     // Conjunction is sequential update, treated separately
00128     if (expr->op == QMLExpression::Operator::CONJUNCTION) {
00129         const auto lhs_update = std::visit(Evaluator(), expr->lhs, params);
00130
00131         if (!lhs_update.has_value()) {
00132             return std::unexpected(
00133                 std::format(
00134                     "In evaluating formula {}:\n{}",
00135                     std::visit(QMLExpression::Formatter(), QMLExpression::Expression(expr)),
00136                     lhs_update.error()
00137                 )
00138             );
00139         }
00140
00141         return std::visit(
00142                 Evaluator(),
00143                 expr->rhs,
00144                 std::variant<std::pair<InformationState, const
      IModel*>>(std::make_pair(lhs_update.value(), model)
00145             )
00146         );
00147     }
00148
00149     // All other updates are filtering updates
00150     InformationState& input_state = (std::get<std::pair<InformationState, const
      IModel*>>(params)).first;
00151     const auto hypothetical_lhs_update = std::visit(Evaluator(), expr->lhs, params);
00152
00153     if (!hypothetical_lhs_update.has_value()) {
00154         return std::unexpected(
00155             std::format(
00156                 "In evaluating formula {}:\n{}",
00157                 std::visit(QMLExpression::Formatter(), QMLExpression::Expression(expr)),
00158                 hypothetical_lhs_update.error()
00159             )
00160         );
00161     }
00162
00163     if (expr->op == QMLExpression::Operator::DISJUNCTION) {
00164         const auto negated_lhs_update = std::visit(Evaluator(), negate(expr->lhs), params);
00165
00166         if (!negated_lhs_update.has_value()) {
00167             return std::unexpected(
00168                 std::format(
00169                     "In evaluating formula {}:\n{}",
00170                     std::visit(QMLExpression::Formatter(), QMLExpression::Expression(expr)),
00171                     negated_lhs_update.error()
00172                 )
00173             );
00174         }
00175
00176         const auto hypothetical_rhs_update = std::visit(
00177             Evaluator(),
00178             expr->rhs,
00179             std::variant<std::pair<InformationState, const
      IModel*>>(std::make_pair(negated_lhs_update.value(), model))
00180         );
00181
00182         if (!hypothetical_rhs_update.has_value()) {
00183             return std::unexpected(
00184                 std::format(
00185                     "In evaluating formula {}:\n{}",
00186                     std::visit(QMLExpression::Formatter(), QMLExpression::Expression(expr)),
00187                     hypothetical_rhs_update.error()
00188                 )
00189             );
00190         }
00191
00192         const auto in_lhs_or_in_rhs = [&](const Possibility& p) -> bool {
00193             return hypothetical_lhs_update.value().contains(p) ||
      hypothetical_rhs_update.value().contains(p);
```

```
00194         };
00195
00196         filter(input_state, in_lhs_or_in_rhs);
00197     }
00198     else if (expr->op == QMLExpression::Operator::CONDITIONAL) {
00199         const auto hypothetical_consequent_update = std::visit(
00200             Evaluator(),
00201             expr->rhs,
00202             std::variant<std::pair<InformationState, const
      IModel*>>(std::make_pair(hypothetical_lhs_update.value(), model))
00203         );
00204
00205         if (!hypothetical_consequent_update.has_value()) {
00206             return std::unexpected(
00207                 std::format(
00208                     "In evaluating formula {}:\n{}",
00209                     std::visit(QMLExpression::Formatter(), QMLExpression::Expression(expr)),
00210                     hypothetical_consequent_update.error()
00211                 )
00212             );
00213         }
00214
00215         const auto all_descendants_subsist = [&](const Possibility& p) -> bool {
00216             const auto not_descendant_or_subsists = [&](const Possibility& p_star) -> bool {
00217                 return !isDescendantOf(p_star, p, hypothetical_lhs_update.value()) ||
      subsistsIn(p_star, hypothetical_consequent_update.value());
00218             };
00219             return std::ranges::all_of(hypothetical_lhs_update.value(), not_descendant_or_subsists);
00220         };
00221
00222         const auto if_subsists_all_descendants_do = [&](const Possibility& p) -> bool {
00223             return !subsistsIn(p, hypothetical_lhs_update.value()) || all_descendants_subsist(p);
00224         };
00225
00226         filter(input_state, if_subsists_all_descendants_do);
00227     }
00228     else {
00229         return std::unexpected(
00230             std::format(
00231                 "In evaluating formula {}:\n{}",
00232                 std::visit(QMLExpression::Formatter(), QMLExpression::Expression(expr)),
00233                 "Invalid operator for binary formula"
00234             )
00235         );
00236     }
00237
00238     return std::move(input_state);
00239 }
00240
00260 std::expected<InformationState, std::string> Evaluator::operator()(const
      std::shared_ptr<QMLExpression::QuantificationNode>& expr, std::variant<std::pair<InformationState,
      const IModel*>> params) const
00261 {
00262     InformationState& input_state = (std::get<std::pair<InformationState, const
      IModel*>>(params)).first;
00263     const IModel* model = (std::get<std::pair<InformationState, const IModel*>>(params)).second;
00264
00265     if (expr->quantifier == QMLExpression::Quantifier::EXISTENTIAL) {
00266         std::vector<InformationState> all_state_variants;
00267
00268         for (const int i : std::views::iota(0, model->domain_cardinality())) {
00269             const InformationState s_variant = update(input_state, expr->variable.literal, i);
00270             const auto hypothetical_s_variant_update = std::visit(
00271                 Evaluator(),
00272                 expr->scope,
00273                 std::variant<std::pair<InformationState, const IModel*>>(std::make_pair(s_variant,
      model))
00274             );
00275
00276             if (!hypothetical_s_variant_update.has_value()) {
00277                 return std::unexpected(
00278                     std::format(
00279                         "In evaluating formula {}:\n{}",
00280                         std::visit(QMLExpression::Formatter(), QMLExpression::Expression(expr)),
00281                         hypothetical_s_variant_update.error()
00282                     )
00283                 );
00284             }
00285
00286             all_state_variants.push_back(hypothetical_s_variant_update.value());
00287         }
00288
00289         InformationState output;
00290         for (const auto& state_variant : all_state_variants) {
00291             for (const auto& p : state_variant) {
00292                 output.insert(p);
00293             }
```

```
00294         }
00295
00296         return output;
00297     }
00298     if (expr->quantifier == QMLExpression::Quantifier::UNIVERSAL) {
00299         std::vector<InformationState> all_hypothetical_updates;
00300
00301         for (const int d : std::views::iota(0, model->domain_cardinality())) {
00302             const auto hypothetical_update = std::visit(
00303                 Evaluator(),
00304                 expr->scope,
00305                 std::variant<std::pair<InformationState, const
      IModel*>>(std::make_pair(update(input_state, expr->variable.literal, d), model))
00306             );
00307
00308             if (!hypothetical_update.has_value()) {
00309                 return std::unexpected(
00310                     std::format(
00311                         "In evaluating formula {}:\n{}",
00312                         std::visit(QMLExpression::Formatter(), QMLExpression::Expression(expr)),
00313                         hypothetical_update.error()
00314                     )
00315                 );
00316             }
00317
00318             all_hypothetical_updates.push_back(hypothetical_update.value());
00319         }
00320
00321         const auto subsists_in_all_hyp_updates = [&](const Possibility& p) -> bool {
00322             const auto p_subsists_in_hyp_update = [&](const InformationState& hypothetical_update) ->
      bool {
00323                 return subsistsIn(p, hypothetical_update);
00324             };
00325             return std::ranges::all_of(all_hypothetical_updates, p_subsists_in_hyp_update);
00326         };
00327
00328         filter(input_state, subsists_in_all_hyp_updates);
00329     }
00330     else {
00331         return std::unexpected(
00332             std::format(
00333                 "In evaluating formula {}:\n{}",
00334                 std::visit(QMLExpression::Formatter(), QMLExpression::Expression(expr)),
00335                 "Invalid quantifier"
00336             )
00337         );
00338     }
00339
00340     return std::move(input_state);
00341 }
00342
00366 std::expected<InformationState, std::string> Evaluator::operator()(const
      std::shared_ptr<QMLExpression::IdentityNode>& expr, std::variant<std::pair<InformationState, const
      IModel*>> params) const
00367 {
00368     InformationState& input_state = (std::get<std::pair<InformationState, const
      IModel*>>(params)).first;
00369     const IModel& model = *(std::get<std::pair<InformationState, const IModel*>>(params)).second;
00370
00371     auto assigns_same_denotation = [&](const Possibility& p) -> bool {
00372         const auto lhs_denotation = expr->lhs.type == QMLExpression::Term::Type::VARIABLE ?
      variableDenotation(expr->lhs.literal, p) : model.termInterpretation(expr->lhs.literal, p.world);
00373         const auto rhs_denotation = expr->rhs.type == QMLExpression::Term::Type::VARIABLE ?
      variableDenotation(expr->rhs.literal, p) : model.termInterpretation(expr->rhs.literal, p.world);
00374
00375         if (!lhs_denotation.has_value()) {
00376             throw std::out_of_range(lhs_denotation.error());
00377         }
00378         if (!rhs_denotation.has_value()) {
00379             throw std::out_of_range(rhs_denotation.error());
00380         }
00381
00382         return lhs_denotation.value() == rhs_denotation.value();
00383     };
00384
00385     try {
00386         filter(input_state, assigns_same_denotation);
00387         return std::move(input_state);
00388     }
00389     catch (const std::out_of_range& e) {
00390         return std::unexpected(
00391             std::format(
00392                 "In evaluating formula {}:\n{}",
00393                 std::visit(QMLExpression::Formatter(), QMLExpression::Expression(expr)),
00394                 e.what()
00395             )
00396         );
```

```
00397     }
00398 }
00399
00423 std::expected<InformationState, std::string> Evaluator::operator()(const
      std::shared_ptr<QMLExpression::PredicationNode>& expr, std::variant<std::pair<InformationState, const
      IModel*>» params) const
00424 {
00425     InformationState& input_state = (std::get<std::pair<InformationState, const
      IModel*>(params)).first;
00426     const IModel& model = *(std::get<std::pair<InformationState, const IModel*>(params)).second;
00427
00428     const auto tuple_in_extension = [&](const Possibility& p) -> bool {
00429         std::vector<int> tuple;
00430
00431         for (const QMLExpression::Term& argument : expr->arguments) {
00432             const auto denotation = argument.type == QMLExpression::Term::Type::VARIABLE ?
      variableDenotation(argument.literal, p) : model.termInterpretation(argument.literal, p.world);
00433             if (denotation.has_value()) {
00434                 tuple.push_back(denotation.value());
00435             }
00436             else {
00437                 throw std::out_of_range(denotation.error());
00438             }
00439         }
00440
00441         const auto predint = model.predicateInterpretation(expr->predicate, p.world);
00442         if (predint.has_value()) {
00443             return predint.value()->contains(tuple);
00444         }
00445         else {
00446             throw std::out_of_range(predint.error());
00447         }
00448     };
00449
00450     try {
00451         filter(input_state, tuple_in_extension);
00452         return std::move(input_state);
00453     }
00454     catch (const std::out_of_range& e) {
00455         return std::unexpected(
00456             std::format(
00457                 "In evaluating formula {}:\n{}",
00458                 std::visit(QMLExpression::Formatter(), QMLExpression::Expression(expr)),
00459                 e.what()
00460             )
00461         );
00462     }
00463 }
00464
00482 std::expected<InformationState, std::string> evaluate(const QMLExpression::Expression& expr, const
      InformationState& input_state, const IModel& model)
00483 {
00484     return std::visit(
00485         Evaluator(),
00486         expr,
00487         std::variant<std::pair<InformationState, const IModel*>(std::make_pair(input_state, &model))
00488     );
00489 }
00490
00491 }
```

7.25 semantic_relations.hpp File Reference

```
#include <expected>
#include <string>
#include <vector>
#include <QMLExpression/expression.hpp>
#include "information_state.hpp"
#include "imodel.hpp"
```

Namespaces

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

Functions

- std::expected< bool, std::string > iif_sadaf::talk::GSV::consistent (const QMLExpression::Expression &expr, const InformationState &state, const IModel &model)

  Determines whether an expression is consistent with a given information state, relative to a base model.
- std::expected< bool, std::string > iif_sadaf::talk::GSV::allows (const InformationState &state, const QMLExpression::Expression &expr, const IModel &model)

  Checks whether an information state allows a given expression.
- std::expected< bool, std::string > iif_sadaf::talk::GSV::supports (const InformationState &state, const QMLExpression::Expression &expr, const IModel &model)

  Determines whether an information state supports a given expression.
- std::expected< bool, std::string > iif_sadaf::talk::GSV::isSupportedBy (const QMLExpression::Expression &expr, const InformationState &state, const IModel &model)

  Checks if an expression is supported by a given information state.
- std::expected< bool, std::string > iif_sadaf::talk::GSV::consistent (const QMLExpression::Expression &expr, const IModel &model)

  Determines whether an expression is consistent within a given model.
- std::expected< bool, std::string > iif_sadaf::talk::GSV::coherent (const QMLExpression::Expression &expr, const IModel &model)

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

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

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

7.26 semantic_relations.hpp

Go to the documentation of this file.

```
00001 #pragma once
00002
00003 #include <expected>
00004 #include <string>
00005 #include <vector>
00006
00007 #include <QMLExpression/expression.hpp>
00008
00009 #include "information_state.hpp"
00010 #include "imodel.hpp"
00011
00012 namespace iif_sadaf::talk::GSV {
00013
00014 std::expected<bool, std::string> consistent(const QMLExpression::Expression& expr, const
      InformationState& state, const IModel& model);
00015 std::expected<bool, std::string> allows(const InformationState& state, const
      QMLExpression::Expression& expr, const IModel& model);
00016 std::expected<bool, std::string> supports(const InformationState& state, const
      QMLExpression::Expression& expr, const IModel& model);
00017 std::expected<bool, std::string> isSupportedBy(const QMLExpression::Expression& expr, const
      InformationState& state, const IModel& model);
00018
00019 std::expected<bool, std::string> consistent(const QMLExpression::Expression& expr, const IModel&
      model);
00020 std::expected<bool, std::string> coherent(const QMLExpression::Expression& expr, const IModel& model);
00021 std::expected<bool, std::string> entails(const std::vector<QMLExpression::Expression>& premises, const
      QMLExpression::Expression& conclusion, const IModel& model);
00022 std::expected<bool, std::string> equivalent(const QMLExpression::Expression& expr1, const
      QMLExpression::Expression& expr2, const IModel& model);
00023
00024 }
```

7.27 semantic_relations.cpp File Reference

```
#include "semantic_relations.hpp"
#include <algorithm>
#include <format>
#include <functional>
#include <ranges>
#include <stdexcept>
#include <variant>
#include <vector>
#include <QMLExpression/formatter.hpp>
#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

- std::expected< bool, std::string > iif_sadaf::talk::GSV::consistent (const QMLExpression::Expression &expr, const InformationState &state, const IModel &model)

  Determines whether an expression is consistent with a given information state, relative to a base model.
- std::expected< bool, std::string > iif_sadaf::talk::GSV::allows (const InformationState &state, const QMLExpression::Expression &expr, const IModel &model)

  Checks whether an information state allows a given expression.
- std::expected< bool, std::string > iif_sadaf::talk::GSV::supports (const InformationState &state, const QMLExpression::Expression &expr, const IModel &model)

  Determines whether an information state supports a given expression.
- std::expected< bool, std::string > iif_sadaf::talk::GSV::isSupportedBy (const QMLExpression::Expression &expr, const InformationState &state, const IModel &model)

  Checks if an expression is supported by a given information state.
- std::expected< bool, std::string > iif_sadaf::talk::GSV::consistent (const QMLExpression::Expression &expr, const IModel &model)

  Determines whether an expression is consistent within a given model.
- std::expected< bool, std::string > iif_sadaf::talk::GSV::coherent (const QMLExpression::Expression &expr, const IModel &model)

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

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

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

7.28 semantic_relations.cpp

Go to the documentation of this file.

```
00001 #include "semantic_relations.hpp"
00002
00003 #include <algorithm>
00004 #include <format>
00005 #include <functional>
00006 #include <ranges>
00007 #include <stdexcept>
00008 #include <variant>
00009 #include <vector>
00010
00011 #include <QMLExpression/formatter.hpp>
00012
00013 #include "evaluator.hpp"
00014 #include "imodel.hpp"
00015 #include "information_state.hpp"
00016 #include "possibility.hpp"
00017
00018 namespace iif_sadaf::talk::GSV {
00019
00038 std::expected<bool, std::string> consistent(const QMLExpression::Expression& expr, const
      InformationState& state, const IModel& model)
00039 {
00040     const auto hypothetical_update = evaluate(expr, state, model);
00041
00042     if (!hypothetical_update.has_value()) {
00043         return std::unexpected(
00044             std::format(
00045                 "In evaluating formula {}:\n{}",
00046                 std::visit(QMLExpression::Formatter(), QMLExpression::Expression(expr)),
00047                 hypothetical_update.error()
00048             )
00049         );
00050     }
00051
00052     return !hypothetical_update.value().empty();
00053 }
00054
00068 std::expected<bool, std::string> allows(const InformationState& state, const
      QMLExpression::Expression& expr, const IModel& model)
00069 {
00070     return consistent(expr, state, model);
00071 }
00072
00086 std::expected<bool, std::string> supports(const InformationState& state, const
      QMLExpression::Expression& expr, const IModel& model)
00087 {
00088     const auto hypothetical_update = evaluate(expr, state, model);
00089
00090     if (!hypothetical_update.has_value()) {
00091         return std::unexpected(
00092             std::format(
00093                 "In evaluating formula {}:\n{}",
00094                 std::visit(QMLExpression::Formatter(), QMLExpression::Expression(expr)),
00095                 hypothetical_update.error()
00096             )
00097         );
00098     }
00099
00100     return subsistsIn(state, hypothetical_update.value());
00101 }
00102
00116 std::expected<bool, std::string> isSupportedBy(const QMLExpression::Expression& expr, const
      InformationState& state, const IModel& model)
00117 {
00118     return supports(state, expr, model);
00119 }
00120
00121 namespace {
00122
00123 std::vector<InformationState> generateSubStates(int n, int k) {
00124     std::vector<InformationState> result;
00125
00126     if (k == 0) {
00127         result.push_back(InformationState());
00128         return result;
00129     }
00130
00131     if (k > n + 1) {
00132         return result;
00133     }
00134
00135     int estimate = 1;
```

```
00136     for (int i = 1; i <= k; i++) {
00137         estimate = estimate * (n + 2 - i) / i;
00138     }
00139     result.reserve(estimate);
00140 
00141     std::function<void(int, InformationState&)> backtrack =
00142         [&](int start, InformationState& current) {
00143             if (current.size() == k) {
00144                 result.push_back(current);
00145                 return;
00146             }
00147 
00148             ReferentSystem r;
00149 
00150             for (int i = start; i <= n; ++i) {
00151                 Possibility p(std::make_shared<ReferentSystem>(r), i);
00152                 current.insert(p);
00153 
00154                 backtrack(i + 1, current);
00155 
00156                 current.erase(p);
00157             }
00158         };
00159 
00160     InformationState current;
00161     backtrack(0, current);
00162 
00163     return result;
00164 }
00165 
00166 } // ANONYMOUS NAMESPACE
00167 
00182 std::expected<bool, std::string> consistent(const QMLExpression::Expression& expr, const IModel&
    model)
00183 {
00184     for (const int i : std::views::iota(0, model.world_cardinality())) {
00185         std::vector<InformationState> states = generateSubStates(model.world_cardinality() - 1, i);
00186         const auto is_consistent = [&](const InformationState& state) -> bool {
00187             const auto result = consistent(expr, state, model);
00188             if (!result.has_value()) {
00189                 throw std::runtime_error(result.error());
00190             }
00191             return result.value();
00192         };
00193         try {
00194             if (!std::ranges::any_of(states, is_consistent)) {
00195                 return false;
00196             }
00197         }
00198         catch (const std::runtime_error& e) {
00199             return std::unexpected(e.what());
00200         }
00201     }
00202     return true;
00203 }
00204 
00219 std::expected<bool, std::string> coherent(const QMLExpression::Expression& expr, const IModel& model)
00220 {
00221     for (const int i : std::views::iota(0, model.world_cardinality())) {
00222         std::vector<InformationState> states = generateSubStates(model.world_cardinality() - 1, i);
00223         const auto is_not_empty_or_supports_expression = [&](const InformationState& state) -> bool {
00224             const auto result = supports(state, expr, model);
00225             if (!result.has_value()) {
00226                 throw std::runtime_error(result.error());
00227             }
00228             return !state.empty() && result.value();
00229         };
00230         try {
00231             if (!std::ranges::any_of(states, is_not_empty_or_supports_expression)) {
00232                 return false;
00233             }
00234         }
00235         catch (const std::runtime_error& e) {
00236             return std::unexpected(e.what());
00237         }
00238     }
00239     return true;
00240 }
00241 
00257 std::expected<bool, std::string> entails(const std::vector<QMLExpression::Expression>& premises, const
    QMLExpression::Expression& conclusion, const IModel& model)
00258 {
00259     for (const int i : std::views::iota(0, model.world_cardinality())) {
00260         std::vector<InformationState> states = generateSubStates(model.world_cardinality() - 1, i);
00261         for (InformationState& input_state : states) {
00262             // Update input state with premises
00263             for (const QMLExpression::Expression& expr : premises) {
```

```
00264                 const auto update = evaluate(expr, input_state, model);
00265                 if (!update.has_value()) {
00266                     return std::unexpected(
00267                         std::format(
00268                             "In evaluating formula {}:\n{}",
00269                             std::visit(QMLExpression::Formatter(), QMLExpression::Expression(expr)),
00270                             update.error()
00271                         )
00272                     );
00273                 }
00274                 input_state = update.value();
00275             }
00276
00277             // check if update with conclusion exists
00278             const auto update = evaluate(conclusion, input_state, model);
00279
00280             // update does not exist
00281             if (!update.has_value()) {
00282                 return std::unexpected(
00283                     std::format(
00284                         "In evaluating formula {}:\n{}",
00285                         std::visit(QMLExpression::Formatter(), QMLExpression::Expression(conclusion)),
00286                         update.error()
00287                     )
00288                 );
00289             }
00290
00291             // update exists, check for support
00292             const auto does_support = supports(input_state, conclusion, model);
00293             if (!does_support.has_value()) {
00294                 return std::unexpected(
00295                     std::format(
00296                         "In evaluating formula {}:\n{}",
00297                         std::visit(QMLExpression::Formatter(), QMLExpression::Expression(conclusion)),
00298                         does_support.error()
00299                     )
00300                 );
00301             }
00302             if (!does_support.value()) {
00303                 return false;
00304             }
00305         }
00306     }
00307     return true;
00308 }
00309
00310 namespace {
00311
00312 std::expected<bool, std::string> similar(const Possibility& p1, const Possibility& p2)
00313 {
00314     const auto have_same_denotation = [&](std::string_view variable) -> bool {
00315         const auto denotation_at_p1 = variableDenotation(variable, p1);
00316         const auto denotation_at_p2 = variableDenotation(variable, p2);
00317         if (!denotation_at_p1.has_value()) {
00318             throw std::out_of_range(denotation_at_p1.error());
00319         }
00320         if (!denotation_at_p2.has_value()) {
00321             throw std::out_of_range(denotation_at_p2.error());
00322         }
00323         return denotation_at_p1.value() == denotation_at_p2.value();
00324     };
00325
00326     try {
00327         return p1.world == p2.world
00328             && domain(*p1.referentSystem) == domain(*p2.referentSystem)
00329             && std::ranges::all_of(domain(*p1.referentSystem), have_same_denotation);
00330     }
00331     catch (const std::out_of_range& e) {
00332         return std::unexpected(e.what());
00333     }
00334 }
00335
00336 std::expected<bool, std::string> similar(const InformationState& s1, const InformationState& s2)
00337 {
00338     const auto has_similar_possibility_in_s2 = [&](const Possibility p) -> bool {
00339         const auto is_similar_to_p = [&](const Possibility p_dash) -> bool {
00340             const auto comparison_result = similar(p, p_dash);
00341             if (!comparison_result.has_value()) {
00342                 throw std::out_of_range(comparison_result.error());
00343             }
00344             return comparison_result.value();
00345         };
00346         return std::ranges::any_of(s2, is_similar_to_p);
00347     };
00348
00349     const auto has_similar_possibility_in_s1 = [&](const Possibility p) -> bool {
00350         const auto is_similar_to_p = [&](const Possibility p_dash) -> bool {
```

```
00351             const auto comparison_result = similar(p, p_dash);
00352             if (!comparison_result.has_value()) {
00353                 throw std::out_of_range(comparison_result.error());
00354             }
00355             return comparison_result.value();
00356         };
00357         return std::ranges::any_of(s1, is_similar_to_p);
00358     };
00359
00360     try {
00361         return std::ranges::all_of(s1, has_similar_possibility_in_s2)
00362             && std::ranges::all_of(s2, has_similar_possibility_in_s1);
00363     }
00364     catch (const std::out_of_range& e) {
00365         return std::unexpected(e.what());
00366     }
00367 }
00368
00369 } // ANONYMOUS NAMESPACE
00370
00385 std::expected<bool, std::string> equivalent(const QMLExpression::Expression& expr1, const
    QMLExpression::Expression& expr2, const IModel& model)
00386 {
00387     for (const int i : std::views::iota(0, model.world_cardinality())) {
00388         std::vector<InformationState> states = generateSubStates(model.world_cardinality() - 1, i);
00389
00390         const auto dissimilar_updates = [&](const InformationState& state) ->bool {
00391             const auto expr1_update = evaluate(expr1, state, model);
00392             if (!expr1_update.has_value()) {
00393                 throw std::out_of_range(expr1_update.error());
00394             }
00395             const auto expr2_update = evaluate(expr2, state, model);
00396             if (!expr2_update.has_value()) {
00397                 throw std::out_of_range(expr2_update.error());
00398             }
00399
00400             return !similar(expr1_update.value(), expr2_update.value());
00401         };
00402
00403         try {
00404             if (std::ranges::any_of(states, dissimilar_updates)) {
00405                 return false;
00406             }
00407         }
00408         catch (const std::out_of_range& e) {
00409             return std::unexpected(e.what());
00410         }
00411     }
00412
00413     return true;
00414 }
00415
00416 }
```

7.29 GSV.hpp File Reference

```
#include "qml_model_adapter.hpp"
#include "information_state.hpp"
#include "possibility.hpp"
#include "referent_system.hpp"
#include "evaluator.hpp"
#include "semantic_relations.hpp"
#include "imodel.hpp"
```

7.30 GSV.hpp

Go to the documentation of this file.

```
00001 #pragma once
00002
00003 #include "qml_model_adapter.hpp"
00004
00005 #include "information_state.hpp"
```

```
00006 #include "possibility.hpp"
00007 #include "referent_system.hpp"
00008
00009 #include "evaluator.hpp"
00010 #include "semantic_relations.hpp"
00011
00012 #include "imodel.hpp"
```

7.31 imodel.hpp File Reference

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

Classes

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

  Interface for class representing a model for Quantified Modal Logic.

Namespaces

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

7.32 imodel.hpp

Go to the documentation of this file.

```
00001 #pragma once
00002
00003 #include <expected>
00004 #include <set>
00005 #include <string>
00006 #include <string_view>
00007 #include <vector>
00008
00009 namespace iif_sadaf::talk::GSV {
00010
00023 struct IModel {
00024 public:
00025     virtual int world_cardinality() const = 0;
00026     virtual int domain_cardinality() const = 0;
00027     virtual std::expected<int, std::string> termInterpretation(std::string_view term, int world) const
      = 0;
00028     virtual std::expected<const std::set<std::vector<int>*, std::string>
      predicateInterpretation(std::string_view predicate, int world) const = 0;
00029     virtual ~IModel() {};
00030 };
00031
00032 }
```

Index