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::UnaryNode > &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::BinaryNode > &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::IdentityNode > &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:

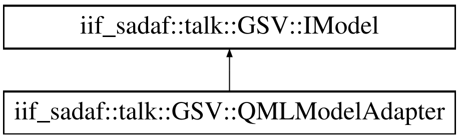
- evaluator.hpp
- evaluator.cpp

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:


iif_sadaf::talk::GSV::IModel
iif_sadaf::talk::GSV::QMLModelAdapter

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:

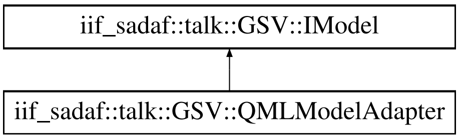
- possibility.hpp
- possibility.cpp

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:


iif_sadaf::talk::GSV::IModel
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::variablePegAssociation = {}
```

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.

```
#pragma once

#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.

```
#pragma once

#include <memory>

#include <QMLModel/qml-model.hpp>

#include "imodel.hpp"

namespace iif_sadaf::talk::GSV {

class QMLModelAdapter : public IModel {
public:
    explicit QMLModelAdapter(const QMLModel::QMLModel& qmlModel);
    explicit QMLModelAdapter(std::unique_ptr<QMLModel::QMLModel> qmlModel);

    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;
    int domain_cardinality() const override;
    std::expected<int, std::string> termInterpretation(std::string_view term, int world) const override;
    std::expected<const std::set<std::vector<int»*, std::string> predicateInterpretation(std::string_view predicate, int world) const override;

private:
    class Impl {
    public:
        explicit Impl(const QMLModel::QMLModel& model) : ownedModel(nullptr), modelRef(&model) {}
        explicit Impl(std::unique_ptr<QMLModel::QMLModel> model)
            : ownedModel(std::move(model)), modelRef(ownedModel.get()) {
        }
        const QMLModel::QMLModel& getModel() const { return *modelRef; }

    private:
        std::unique_ptr<QMLModel::QMLModel> ownedModel;
        const QMLModel::QMLModel* modelRef;
    };
    std::unique_ptr<Impl> pImpl;
};

}
```

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.

```
#include "qml_model_adapter.hpp"

namespace iif_sadaf::talk::GSV {

QMLModelAdapter::QMLModelAdapter(const QMLModel::QMLModel& QMLModelModel)
    : pImpl(std::make_unique<Impl>(QMLModelModel)) {}

QMLModelAdapter::QMLModelAdapter(std::unique_ptr<QMLModel::QMLModel> QMLModelModel)
    : pImpl(std::make_unique<Impl>(std::move(QMLModelModel))) {}

int QMLModelAdapter::world_cardinality() const
{
    return pImpl->getModel().world_cardinality();
}

int QMLModelAdapter::domain_cardinality() const
{
    return pImpl->getModel().domain_cardinality();
}

std::expected<int, std::string> QMLModelAdapter::termInterpretation(std::string_view term, int world) const
{
    return pImpl->getModel().termInterpretation(term, world);
}

std::expected<const std::set<std::vector<int»*, std::string> QMLModelAdapter::predicateInterpretation(std::string_view predicate, int world) const
{
    return pImpl->getModel().predicateInterpretation(predicate, world);
}

}
```

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.

```
#pragma once

#include "information_state.hpp"
#include "possibility.hpp"
#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.

```
#pragma once

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

#include "imodel.hpp"
#include "possibility.hpp"

namespace iif_sadaf::talk::GSV {

using InformationState = std::set<Possibility>;

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

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

std::string str(const InformationState& state);
std::string repr(const InformationState& state);

}
```

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.

```
#pragma once

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

#include "referent_system.hpp"

namespace iif_sadaf::talk::GSV {

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

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

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

bool extends(const Possibility& p2, const Possibility& p1);
bool operator<(const Possibility& p1, const Possibility& p2);
std::expected<int, std::string> variableDenotation(std::string_view variable, const Possibility& p);

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

}
```

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.

```
#pragma once

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

namespace iif_sadaf::talk::GSV {

struct ReferentSystem {
public:
    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;

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

std::set<std::string_view> domain(const ReferentSystem& r);
bool extends(const ReferentSystem& r2, const ReferentSystem& r1);
std::string str(const ReferentSystem& r);
std::string repr(const ReferentSystem& r);

}
```

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.

```
#include "information_state.hpp"

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

namespace iif_sadaf::talk::GSV {

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

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

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

    return possibilities;
}

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

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

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

        p_star.update(variable, individual);

        output_state.insert(p_star);
    }

    return output_state;
}

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

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

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

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

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

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

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

    desc.pop_back();

    return desc;
}

std::string repr(const InformationState& state)
{
    std::string desc = "Information State : [\n";

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

    return desc + "]";
}

}
```

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.

```
#include "possibility.hpp"

#include <algorithm>

namespace iif_sadaf::talk::GSV {

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

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

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

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

/*
* NON-MEMBER FUNCTIONS
*/

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

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

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

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

std::expected<int, std::string> variableDenotation(std::string_view variable, const Possibility& p)
{
    const auto peg = p.referentSystem->value(variable);

    if (!peg.has_value()) {
        return std::unexpected(peg.error());
    }

    // Whenever variable exists in referent system, assignment is guaranteed to
    // contain the corresponding peg, so there is no need to check for existence
    // before returning
    return p.assignment.at(peg.value());
}

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

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

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

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

    return desc;
}

std::string repr(const Possibility& p)
{
    std::string desc = "Possibility : [ " + repr(*p.referentSystem) + ", Assignment : [ ";

    if (p.assignment.empty()) {
        desc += "]";
    }
    else {
        for (const auto& [peg, individual] : p.assignment) {
            desc += "{ " + std::to_string(peg) + " : " + std::to_string(individual) + " }, ";
        }
        desc.resize(desc.size() - 2);
        desc += " ]";
    }

    desc += ", World : " + std::to_string(p.world) + " ]";

    return desc;
}

}
```

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.

```
#include "referent_system.hpp"

#include <algorithm>
#include <format>
#include <stdexcept>

namespace iif_sadaf::talk::GSV {

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

    return domain;
}

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

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

std::expected<int, std::string> ReferentSystem::value(std::string_view variable) const
{
    if (!variablePegAssociation.contains(variable)) {
        return std::unexpected(std::format("Referent system does not contain variable {}", std::string(variable)));
    }

    return variablePegAssociation.at(variable);
}

bool extends(const ReferentSystem& r2, const ReferentSystem& r1)
{
    // TODO check that these calls to value() are safe
    if (r1.pegs > r2.pegs) {
        return false;
    }

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

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

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

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

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

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

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

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

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

    return desc;
}

std::string repr(const ReferentSystem& r)
{
    std::string desc = "R-System : [ ";

    if (r.variablePegAssociation.empty()) {
        return desc + "]";
    }

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

    desc.resize(desc.size() - 2);

    return desc + " ]";
}

}
```

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.

```
#pragma once

#include <expected>

#include <QMLExpression/expression.hpp>

#include "information_state.hpp"

namespace iif_sadaf::talk::GSV {

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

std::expected<InformationState, std::string> evaluate(const QMLExpression::Expression& expr, const InformationState& input_state, const IModel& model);

}
```

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.

```cpp
#include "evaluator.hpp"

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

#include <QMLExpression/formatter.hpp>

#include "possibility.hpp"

namespace iif_sadaf::talk::GSV {

namespace {

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

QMLExpression::Expression negate(const QMLExpression::Expression& expr)
{
    return std::make_shared<QMLExpression::UnaryNode>(QMLExpression::Operator::NEGATION, expr);
}

} // ANONYMOUS NAMESPACE

std::expected<InformationState, std::string> Evaluator::operator()(const std::shared_ptr<QMLExpression::UnaryNode>& expr, std::variant<std::pair<InformationState, const IModel*» params) const
{
    const auto prejacent_update = std::visit(Evaluator(), expr->scope, params);

    if (!prejacent_update.has_value()) {
        return std::unexpected(
            std::format(
                "In evaluating formula {}:\n{}",
                std::visit(QMLExpression::Formatter(), QMLExpression::Expression(expr)),
                prejacent_update.error()
            )
        );
    }

    InformationState& input_state = std::get<std::pair<InformationState, const IModel*»(params).first;

    if (expr->op == QMLExpression::Operator::EPISTEMIC_POSSIBILITY) {
        if (prejacent_update.value().empty()) {
            input_state.clear();
        }
    }
    else if (expr->op == QMLExpression::Operator::EPISTEMIC_NECESSITY) {
        if (!subsistsIn(input_state, prejacent_update.value())) {
            input_state.clear();
        }
    }
    else if (expr->op == QMLExpression::Operator::NEGATION) {
        filter(input_state, [&](const Possibility& p) -> bool { return !subsistsIn(p, prejacent_update.value()); });
    }
    else {
        return std::unexpected(
            std::format(
                "In evaluating formula {}:\n{}",
                std::visit(QMLExpression::Formatter(), QMLExpression::Expression(expr)),
                "Invalid unary operator"
            )
        );
    }

    return std::move(input_state);
}

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

    // Conjunction is sequential update, treated separately
    if (expr->op == QMLExpression::Operator::CONJUNCTION) {
        const auto lhs_update = std::visit(Evaluator(), expr->lhs, params);

        if (!lhs_update.has_value()) {
            return std::unexpected(
                std::format(
                    "In evaluating formula {}:\n{}",
                    std::visit(QMLExpression::Formatter(), QMLExpression::Expression(expr)),
                    lhs_update.error()
                )
            );
        }

        return std::visit(
                Evaluator(),
                expr->rhs,
                std::variant<std::pair<InformationState, const
IModel*»(std::make_pair(lhs_update.value(), model)
            )
        );
    }

    // All other updates are filtering updates
    InformationState& input_state = (std::get<std::pair<InformationState, const
IModel*»(params)).first;
    const auto hypothetical_lhs_update = std::visit(Evaluator(), expr->lhs, params);

    if (!hypothetical_lhs_update.has_value()) {
        return std::unexpected(
            std::format(
                "In evaluating formula {}:\n{}",
                std::visit(QMLExpression::Formatter(), QMLExpression::Expression(expr)),
                hypothetical_lhs_update.error()
            )
        );
    }

    if (expr->op == QMLExpression::Operator::DISJUNCTION) {
        const auto negated_lhs_update = std::visit(Evaluator(), negate(expr->lhs), params);

        if (!negated_lhs_update.has_value()) {
            return std::unexpected(
                std::format(
                    "In evaluating formula {}:\n{}",
                    std::visit(QMLExpression::Formatter(), QMLExpression::Expression(expr)),
                    negated_lhs_update.error()
                )
            );
        }

        const auto hypothetical_rhs_update = std::visit(
            Evaluator(),
            expr->rhs,
            std::variant<std::pair<InformationState, const
IModel*»(std::make_pair(negated_lhs_update.value(), model))
        );

        if (!hypothetical_rhs_update.has_value()) {
            return std::unexpected(
                std::format(
                    "In evaluating formula {}:\n{}",
                    std::visit(QMLExpression::Formatter(), QMLExpression::Expression(expr)),
                    hypothetical_rhs_update.error()
                )
            );
        }

        const auto in_lhs_or_in_rhs = [&](const Possibility& p) -> bool {
            return hypothetical_lhs_update.value().contains(p) ||
hypothetical_rhs_update.value().contains(p);
```

```
        };

        filter(input_state, in_lhs_or_in_rhs);
    }
    else if (expr->op == QMLExpression::Operator::CONDITIONAL) {
        const auto hypothetical_consequent_update = std::visit(
            Evaluator(),
            expr->rhs,
            std::variant<std::pair<InformationState, const IModel*»(std::make_pair(hypothetical_lhs_update.value(), model))
        );

        if (!hypothetical_consequent_update.has_value()) {
            return std::unexpected(
                std::format(
                    "In evaluating formula {}:\n{}",
                    std::visit(QMLExpression::Formatter(), QMLExpression::Expression(expr)),
                    hypothetical_consequent_update.error()
                )
            );
        }

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

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

        filter(input_state, if_subsists_all_descendants_do);
    }
    else {
        return std::unexpected(
            std::format(
                "In evaluating formula {}:\n{}",
                std::visit(QMLExpression::Formatter(), QMLExpression::Expression(expr)),
                "Invalid operator for binary formula"
            )
        );
    }

    return std::move(input_state);
}

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

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

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

            if (!hypothetical_s_variant_update.has_value()) {
                return std::unexpected(
                    std::format(
                        "In evaluating formula {}:\n{}",
                        std::visit(QMLExpression::Formatter(), QMLExpression::Expression(expr)),
                        hypothetical_s_variant_update.error()
                    )
                );
            }

            all_state_variants.push_back(hypothetical_s_variant_update.value());
        }

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

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

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

            if (!hypothetical_update.has_value()) {
                return std::unexpected(
                    std::format(
                        "In evaluating formula {}:\n{}",
                        std::visit(QMLExpression::Formatter(), QMLExpression::Expression(expr)),
                        hypothetical_update.error()
                    )
                );
            }

            all_hypothetical_updates.push_back(hypothetical_update.value());
        }

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

        filter(input_state, subsists_in_all_hyp_updates);
    }
    else {
        return std::unexpected(
            std::format(
                "In evaluating formula {}:\n{}",
                std::visit(QMLExpression::Formatter(), QMLExpression::Expression(expr)),
                "Invalid quantifier"
            )
        );
    }

    return std::move(input_state);
}

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

    auto assigns_same_denotation = [&](const Possibility& p) -> bool {
        const auto lhs_denotation = expr->lhs.type == QMLExpression::Term::Type::VARIABLE ? variableDenotation(expr->lhs.literal, p) : model.termInterpretation(expr->lhs.literal, p.world);
        const auto rhs_denotation = expr->rhs.type == QMLExpression::Term::Type::VARIABLE ? variableDenotation(expr->rhs.literal, p) : model.termInterpretation(expr->rhs.literal, p.world);

        if (!lhs_denotation.has_value()) {
            throw std::out_of_range(lhs_denotation.error());
        }
        if (!rhs_denotation.has_value()) {
            throw std::out_of_range(rhs_denotation.error());
        }

        return lhs_denotation.value() == rhs_denotation.value();
    };

    try {
        filter(input_state, assigns_same_denotation);
        return std::move(input_state);
    }
    catch (const std::out_of_range& e) {
        return std::unexpected(
            std::format(
                "In evaluating formula {}:\n{}",
                std::visit(QMLExpression::Formatter(), QMLExpression::Expression(expr)),
                e.what()
            )
        );
    }
}

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

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

        for (const QMLExpression::Term& argument : expr->arguments) {
            const auto denotation = argument.type == QMLExpression::Term::Type::VARIABLE ? variableDenotation(argument.literal, p) : model.termInterpretation(argument.literal, p.world);
            if (denotation.has_value()) {
                tuple.push_back(denotation.value());
            }
            else {
                throw std::out_of_range(denotation.error());
            }
        }

        const auto predint = model.predicateInterpretation(expr->predicate, p.world);
        if (predint.has_value()) {
            return predint.value()->contains(tuple);
        }
        else {
            throw std::out_of_range(predint.error());
        }
    };

    try {
        filter(input_state, tuple_in_extension);
        return std::move(input_state);
    }
    catch (const std::out_of_range& e) {
        return std::unexpected(
            std::format(
                "In evaluating formula {}:\n{}",
                std::visit(QMLExpression::Formatter(), QMLExpression::Expression(expr)),
                e.what()
            )
        );
    }
}

std::expected<InformationState, std::string> evaluate(const QMLExpression::Expression& expr, const InformationState& input_state, const IModel& model)
{
    return std::visit(
        Evaluator(),
        expr,
        std::variant<std::pair<InformationState, const IModel*»(std::make_pair(input_state, &model))
    );
}

}
```

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.

```
#pragma once

#include <expected>
#include <string>
#include <vector>

#include <QMLExpression/expression.hpp>

#include "information_state.hpp"
#include "imodel.hpp"

namespace iif_sadaf::talk::GSV {

std::expected<bool, std::string> consistent(const QMLExpression::Expression& expr, const InformationState& state, const IModel& model);
std::expected<bool, std::string> allows(const InformationState& state, const QMLExpression::Expression& expr, const IModel& model);
std::expected<bool, std::string> supports(const InformationState& state, const QMLExpression::Expression& expr, const IModel& model);
std::expected<bool, std::string> isSupportedBy(const QMLExpression::Expression& expr, const InformationState& state, const IModel& model);

std::expected<bool, std::string> consistent(const QMLExpression::Expression& expr, const IModel& model);
std::expected<bool, std::string> coherent(const QMLExpression::Expression& expr, const IModel& model);
std::expected<bool, std::string> entails(const std::vector<QMLExpression::Expression>& premises, const QMLExpression::Expression& conclusion, const IModel& model);
std::expected<bool, std::string> equivalent(const QMLExpression::Expression& expr1, const QMLExpression::Expression& expr2, const IModel& model);

}
```

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.

```cpp
#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"

namespace iif_sadaf::talk::GSV {

std::expected<bool, std::string> consistent(const QMLExpression::Expression& expr, const InformationState& state, const IModel& model)
{
    const auto hypothetical_update = evaluate(expr, state, model);

    if (!hypothetical_update.has_value()) {
        return std::unexpected(
            std::format(
                "In evaluating formula {}:\n{}",
                std::visit(QMLExpression::Formatter(), QMLExpression::Expression(expr)),
                hypothetical_update.error()
            )
        );
    }

    return !hypothetical_update.value().empty();
}

std::expected<bool, std::string> allows(const InformationState& state, const QMLExpression::Expression& expr, const IModel& model)
{
    return consistent(expr, state, model);
}

std::expected<bool, std::string> supports(const InformationState& state, const QMLExpression::Expression& expr, const IModel& model)
{
    const auto hypothetical_update = evaluate(expr, state, model);

    if (!hypothetical_update.has_value()) {
        return std::unexpected(
            std::format(
                "In evaluating formula {}:\n{}",
                std::visit(QMLExpression::Formatter(), QMLExpression::Expression(expr)),
                hypothetical_update.error()
            )
        );
    }

    return subsistsIn(state, hypothetical_update.value());
}

std::expected<bool, std::string> isSupportedBy(const QMLExpression::Expression& expr, const InformationState& state, const IModel& model)
{
    return supports(state, expr, model);
}

namespace {

std::vector<InformationState> generateSubStates(int n, int k) {
    std::vector<InformationState> result;

    if (k == 0) {
        result.push_back(InformationState());
        return result;
    }

    if (k > n + 1) {
        return result;
    }

    int estimate = 1;
    for (int i = 1; i <= k; i++) {
        estimate = estimate * (n + 2 - i) / i;
    }
    result.reserve(estimate);

    std::function<void(int, InformationState&)> backtrack =
        [&](int start, InformationState& current) {
            if (current.size() == k) {
                result.push_back(current);
                return;
            }

            ReferentSystem r;

            for (int i = start; i <= n; ++i) {
                Possibility p(std::make_shared<ReferentSystem>(r), i);
                current.insert(p);

                backtrack(i + 1, current);

                current.erase(p);
            }
        };

    InformationState current;
    backtrack(0, current);

    return result;
}

} // ANONYMOUS NAMESPACE

std::expected<bool, std::string> consistent(const QMLExpression::Expression& expr, const IModel& model)
{
    for (const int i : std::views::iota(0, model.world_cardinality())) {
        std::vector<InformationState> states = generateSubStates(model.world_cardinality() - 1, i);
        const auto is_consistent = [&](const InformationState& state) -> bool {
            const auto result = consistent(expr, state, model);
            if (!result.has_value()) {
                throw std::runtime_error(result.error());
            }
            return result.value();
        };
        try {
            if (!std::ranges::any_of(states, is_consistent)) {
                return false;
            }
        }
        catch (const std::runtime_error& e) {
            return std::unexpected(e.what());
        }
    }
    return true;
}

std::expected<bool, std::string> coherent(const QMLExpression::Expression& expr, const IModel& model)
{
    for (const int i : std::views::iota(0, model.world_cardinality())) {
        std::vector<InformationState> states = generateSubStates(model.world_cardinality() - 1, i);
        const auto is_not_empty_or_supports_expression = [&](const InformationState& state) -> bool {
            const auto result = supports(state, expr, model);
            if (!result.has_value()) {
                throw std::runtime_error(result.error());
            }
            return !state.empty() && result.value();
        };
        try {
            if (!std::ranges::any_of(states, is_not_empty_or_supports_expression)) {
                return false;
            }
        }
        catch (const std::runtime_error& e) {
            return std::unexpected(e.what());
        }
    }
    return true;
}

std::expected<bool, std::string> entails(const std::vector<QMLExpression::Expression>& premises, const QMLExpression::Expression& conclusion, const IModel& model)
{
    for (const int i : std::views::iota(0, model.world_cardinality())) {
        std::vector<InformationState> states = generateSubStates(model.world_cardinality() - 1, i);
        for (InformationState& input_state : states) {
            // Update input state with premises
            for (const QMLExpression::Expression& expr : premises) {
                const auto update = evaluate(expr, input_state, model);
                if (!update.has_value()) {
                    return std::unexpected(
                        std::format(
                            "In evaluating formula {}:\n{}",
                            std::visit(QMLExpression::Formatter(), QMLExpression::Expression(expr)),
                            update.error()
                        )
                    );
                }
                input_state = update.value();
            }

            // check if update with conclusion exists
            const auto update = evaluate(conclusion, input_state, model);

            // update does not exist
            if (!update.has_value()) {
                return std::unexpected(
                    std::format(
                        "In evaluating formula {}:\n{}",
                        std::visit(QMLExpression::Formatter(), QMLExpression::Expression(conclusion)),
                        update.error()
                    )
                );
            }

            // update exists, check for support
            const auto does_support = supports(input_state, conclusion, model);
            if (!does_support.has_value()) {
                return std::unexpected(
                    std::format(
                        "In evaluating formula {}:\n{}",
                        std::visit(QMLExpression::Formatter(), QMLExpression::Expression(conclusion)),
                        does_support.error()
                    )
                );
            }
            if (!does_support.value()) {
                return false;
            }
        }
    }
    return true;
}

namespace {

std::expected<bool, std::string> similar(const Possibility& p1, const Possibility& p2)
{
    const auto have_same_denotation = [&](std::string_view variable) -> bool {
        const auto denotation_at_p1 = variableDenotation(variable, p1);
        const auto denotation_at_p2 = variableDenotation(variable, p2);
        if (!denotation_at_p1.has_value()) {
            throw std::out_of_range(denotation_at_p1.error());
        }
        if (!denotation_at_p2.has_value()) {
            throw std::out_of_range(denotation_at_p2.error());
        }
        return denotation_at_p1.value() == denotation_at_p2.value();
    };

    try {
        return p1.world == p2.world
            && domain(*p1.referentSystem) == domain(*p2.referentSystem)
            && std::ranges::all_of(domain(*p1.referentSystem), have_same_denotation);
    }
    catch (const std::out_of_range& e) {
        return std::unexpected(e.what());
    }
}

std::expected<bool, std::string> similar(const InformationState& s1, const InformationState& s2)
{
    const auto has_similar_possibility_in_s2 = [&](const Possibility p) -> bool {
        const auto is_similar_to_p = [&](const Possibility p_dash) -> bool {
            const auto comparison_result = similar(p, p_dash);
            if (!comparison_result.has_value()) {
                throw std::out_of_range(comparison_result.error());
            }
            return comparison_result.value();
        };
        return std::ranges::any_of(s2, is_similar_to_p);
    };

    const auto has_similar_possibility_in_s1 = [&](const Possibility p) -> bool {
        const auto is_similar_to_p = [&](const Possibility p_dash) -> bool {
            const auto comparison_result = similar(p, p_dash);
            if (!comparison_result.has_value()) {
                throw std::out_of_range(comparison_result.error());
            }
            return comparison_result.value();
        };
        return std::ranges::any_of(s1, is_similar_to_p);
    };

    try {
        return std::ranges::all_of(s1, has_similar_possibility_in_s2)
            && std::ranges::all_of(s2, has_similar_possibility_in_s1);
    }
    catch (const std::out_of_range& e) {
        return std::unexpected(e.what());
    }
}

} // ANONYMOUS NAMESPACE

std::expected<bool, std::string> equivalent(const QMLExpression::Expression& expr1, const QMLExpression::Expression& expr2, const IModel& model)
{
    for (const int i : std::views::iota(0, model.world_cardinality())) {
        std::vector<InformationState> states = generateSubStates(model.world_cardinality() - 1, i);

        const auto dissimilar_updates = [&](const InformationState& state) ->bool {
            const auto expr1_update = evaluate(expr1, state, model);
            if (!expr1_update.has_value()) {
                throw std::out_of_range(expr1_update.error());
            }
            const auto expr2_update = evaluate(expr2, state, model);
            if (!expr2_update.has_value()) {
                throw std::out_of_range(expr2_update.error());
            }

            return !similar(expr1_update.value(), expr2_update.value());
        };

        try {
            if (std::ranges::any_of(states, dissimilar_updates)) {
                return false;
            }
        }
        catch (const std::out_of_range& e) {
            return std::unexpected(e.what());
        }
    }

    return true;
}

}
```

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.

```
#pragma once

#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"
```

7.31 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.

```
#pragma once

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

namespace iif_sadaf::talk::GSV {

struct IModel {
public:
    virtual int world_cardinality() const = 0;
    virtual int domain_cardinality() const = 0;
    virtual 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() {};
};

}
```

Index