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 Class Index

2.1 Class List

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

3 File Index

3.1 File List

Here is a list of all files with brief descriptions:

4 Namespace Documentation

4.1 iif_sadaf Namespace Reference

Namespaces

- namespace talk

4.2 iif_sadaf::talk Namespace Reference

Namespaces

- namespace GSV

4.3 iif_sadaf::talk::GSV Namespace Reference

Classes

- struct Evaluator

  Represents an evaluator for logical expressions.
- struct Formatter

  A visitor for formatting Expression objects.
- struct IModel

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

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

  Represents a referent system for variable assignments.

Typedefs

- using InformationState = std::set<Possibility>

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

Functions

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

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

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

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

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

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

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

Determines whether a set of premises entails a conclusion within a given model.

- std::expected< bool, std::string > equivalent (const Expression &expr1, const Expression &expr2, const IModel &model)

  Determines whether two expressions are logically equivalent within a given model.
- InformationState create (const IModel &model)

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

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

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

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

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

  Checks if an information state subsists within another.
- std::string str (const InformationState &state)
- 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 variable names 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)

4.3.1 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.

4.3.2 Function Documentation

allows()

```
std::expected< bool, std::string > iif_sadaf::talk::GSV::allows (
            const InformationState & state,
            const 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 67 of file semantic_relations.cpp.

coherent()

```
std::expected< bool, std::string > iif_sadaf::talk::GSV::coherent (
            const 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 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 218 of file semantic_relations.cpp.

consistent() [1/2]

```
std::expected< bool, std::string > iif_sadaf::talk::GSV::consistent (
            const 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 within the 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 181 of file semantic_relations.cpp.

consistent() [2/2]

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

Determines whether an expression is consistent with a given information state and 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 37 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 variable names in the referent system.

This function extracts all variable names present in the given ReferentSystem instance and returns them as a set of string views.

Parameters

r	The ReferentSystem instance whose variable names are being queried.

Returns

std::set<std::string_view> A set containing all variable names 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< Expression > & premises,
            const Expression & conclusion,
            const IModel & model)
```

Determines whether a set of premises entails a conclusion within 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 256 of file semantic_relations.cpp.

equivalent()

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

Determines whether two expressions are logically equivalent within 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 384 of file semantic_relations.cpp.

evaluate()

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

Evaluates a logical expression within a given information state and 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 476 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 79 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 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 115 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 124 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 107 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 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 85 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.

5 Class Documentation

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

Represents an evaluator for logical expressions.

`#include <evaluator.hpp>`

Public Member Functions

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

  Evaluates a quantified expression within a given information state and model.
- std::expected< InformationState, std::string > operator() (const std::shared_ptr< 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< PredicationNode > &expr, std::variant< std::pair< InformationState, const IModel * > > params) const

  Evaluates a predication expression and filters the information state accordingly.

5.1.1 Detailed Description

Represents an evaluator for logical expressions.

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

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

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

Definition at line 25 of file evaluator.hpp.

5.1.2 Member Function Documentation

operator()() [1/5]

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

Evaluates a binary logical expression 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 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:

- **CON (Conjunction)**: The lhs is evaluated first, and the resulting state is then used to evaluate the rhs.
- **DIS (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.
- **IMP (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 117 of file evaluator.cpp.

operator()() [2/5]

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

Evaluates an identity expression 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 360 of file evaluator.cpp.

operator()() [3/5]

```
std::expected< InformationState, std::string > iif_sadaf::talk::GSV::Evaluator::operator() (
            const std::shared_ptr< 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 417 of file evaluator.cpp.

operator()() [4/5]

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

Evaluates a quantified expression within a given information state and model.

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 (Ex F(x))**: Evaluates the scope F(x) for all values in the domain, then merges all resulting information states.
- **Universal Quantification (Vx F(x))**: Evaluates F(x) 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 254 of file evaluator.cpp.

operator()() [5/5]

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

Evaluates a unary logical expression 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 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:

- **E_POS (Epistemic Possibility)**: If the prejacent state is empty, the input state is cleared.
- **E_NEC (Epistemic Necessity)**: If the prejacent state is not contained in the input state, the input state is cleared.
- **NEG (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 50 of file evaluator.cpp.

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

- evaluator.hpp
- evaluator.cpp

5.2 iif_sadaf::talk::GSV::Formatter Struct Reference

A visitor for formatting Expression objects.

`#include <formatter.hpp>`

Public Member Functions

- std::string operator() (std::shared_ptr< UnaryNode > expr) const
- std::string operator() (std::shared_ptr< BinaryNode > expr) const
- std::string operator() (std::shared_ptr< QuantificationNode > expr) const
- std::string operator() (std::shared_ptr< PredicationNode > expr) const
- std::string operator() (std::shared_ptr< IdentityNode > expr) const

5.2.1 Detailed Description

A visitor for formatting Expression objects.

The `Formatter` struct provides a std::string representation of `Expression` objects.

Definition at line 13 of file formatter.hpp.

5.2.2 Member Function Documentation

operator()() [1/5]

```
std::string iif_sadaf::talk::GSV::Formatter::operator() (
            std::shared_ptr< BinaryNode > expr) const
```

Definition at line 21 of file formatter.cpp.

operator()() [2/5]

```
std::string iif_sadaf::talk::GSV::Formatter::operator() (
            std::shared_ptr< IdentityNode > expr) const
```

Definition at line 50 of file formatter.cpp.

operator()() [3/5]

```
std::string iif_sadaf::talk::GSV::Formatter::operator() (
            std::shared_ptr< PredicationNode > expr) const
```

Definition at line 37 of file formatter.cpp.

operator()() [4/5]

```
std::string iif_sadaf::talk::GSV::Formatter::operator() (
            std::shared_ptr< QuantificationNode > expr) const
```

Definition at line 30 of file formatter.cpp.

operator()() [5/5]

```
std::string iif_sadaf::talk::GSV::Formatter::operator() (
            std::shared_ptr< UnaryNode > expr) const
```

Definition at line 5 of file formatter.cpp.

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

- formatter.hpp
- formatter.cpp

5.3 iif_sadaf::talk::GSV::IModel Struct Reference

Interface for class representing a model for QML without accessiblity.

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

Public Member Functions

- virtual int world_cardinality () const =0
- virtual int domain_cardinality () const =0
- virtual 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 ()

5.3.1 Detailed Description

Interface for class representing a model for QML without accessiblity.

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

Any such implementation should contain four functions:

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

Definition at line 23 of file imodel.hpp.

5.3.2 Constructor & Destructor Documentation

~IModel()

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

Definition at line 29 of file imodel.hpp.

5.3.3 Member Function Documentation

domain_cardinality()

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

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]
```

termInterpretation()

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

world_cardinality()

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

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

- imodel.hpp

5.4 iif_sadaf::talk::GSV::Possibility Struct Reference

Represents a possibility as understood in the underlying semantics.

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

Public Member Functions

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

  Updates the assignment of a variable to an individual.

Public Attributes

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

5.4.1 Detailed Description

Represents a possibility as understood in the underlying semantics.

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

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

Definition at line 22 of file possibility.hpp.

5.4.2 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.

5.4.3 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.

5.4.4 Member Data Documentation

assignment

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

Definition at line 33 of file possibility.hpp.

referentSystem

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

Definition at line 32 of file possibility.hpp.

world

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

Definition at line 34 of file possibility.hpp.

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

- possibility.hpp
- possibility.cpp

5.5 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 = {}

5.5.1 Detailed Description

Represents a referent system for variable assignments.

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

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

Definition at line 21 of file referent_system.hpp.

5.5.2 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.

5.5.3 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.

5.5.4 Member Data Documentation

pegs

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

Definition at line 31 of file referent_system.hpp.

variablePegAssociation

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

Definition at line 32 of file referent_system.hpp.

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

- referent_system.hpp
- referent_system.cpp

6 File Documentation

6.1 evaluator.hpp File Reference

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

Classes

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

  Represents an evaluator for logical expressions.

Namespaces

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

Functions

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

  Evaluates a logical expression within a given information state and model.

6.2 evaluator.hpp

Go to the documentation of this file.

```
00001 #pragma once
00002
00003 #include <expected>
00004
00005 #include "expression.hpp"
00006 #include "information_state.hpp"
00007
00008 namespace iif_sadaf::talk::GSV {
00009
00025 struct Evaluator {
00026     std::expected<InformationState, std::string> operator()(const std::shared_ptr<UnaryNode>& expr,
      std::variant<std::pair<InformationState, const IModel*>> params) const;
00027     std::expected<InformationState, std::string> operator()(const std::shared_ptr<BinaryNode>& expr,
      std::variant<std::pair<InformationState, const IModel*>> params) const;
00028     std::expected<InformationState, std::string> operator()(const std::shared_ptr<QuantificationNode>&
      expr, std::variant<std::pair<InformationState, const IModel*>> params) const;
00029     std::expected<InformationState, std::string> operator()(const std::shared_ptr<IdentityNode>& expr,
      std::variant<std::pair<InformationState, const IModel*>> params) const;
00030     std::expected<InformationState, std::string> operator()(const std::shared_ptr<PredicationNode>&
      expr, std::variant<std::pair<InformationState, const IModel*>> params) const;
00031 };
00032
00033 std::expected<InformationState, std::string> evaluate(const Expression& expr, const InformationState&
      input_state, const IModel& model);
00034
00035 }
```

6.3 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 QML without accessiblity.

Namespaces

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

6.4 imodel.hpp

Go to the documentation of this file.

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

6.5 semantic_relations.hpp File Reference

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

Namespaces

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

Functions

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

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

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

  Determines whether two expressions are logically equivalent within a given model.

6.6 semantic_relations.hpp

Go to the documentation of this file.

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

6.7 information_state.hpp File Reference

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

Namespaces

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

Typedefs

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

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

Functions

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

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

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

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

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

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

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

6.8 information_state.hpp

Go to the documentation of this file.

```
00001 #pragma once
00002
00003 #include <set>
00004 #include <string>
00005 #include <string_view>
00006
00007 #include "model.hpp"
00008 #include "possibility.hpp"
00009
00010 namespace iif_sadaf::talk::GSV {
00011
00015 using InformationState = std::set<Possibility>;
00016
00017 InformationState create(const IModel& model);
00018 InformationState update(const InformationState& input_state, std::string_view variable, int
      individual);
00019 bool extends(const InformationState& s2, const InformationState& s1);
00020
00021 bool isDescendantOf(const Possibility& p2, const Possibility& p1, const InformationState& s);
00022 bool subsistsIn(const Possibility& p, const InformationState& s);
00023 bool subsistsIn(const InformationState& s1, const InformationState& s2);
00024
00025 std::string str(const InformationState& state);
00026 std::string repr(const InformationState& state);
00027
00028 }
```

6.9 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)

6.10 possibility.hpp

Go to the documentation of this file.

```
00001 #pragma once
00002
00003 #include <expected>
00004 #include <memory>
00005 #include <string>
00006 #include <unordered_map>
00007
00008 #include "referent_system.hpp"
00009
00010 namespace iif_sadaf::talk::GSV {
00011
00022 struct Possibility {
00023 public:
00024     Possibility(std::shared_ptr<ReferentSystem> r_system, int world);
00025     Possibility(const Possibility& other) = default;
00026     Possibility& operator=(const Possibility& other) = default;
00027     Possibility(Possibility&& other) noexcept;
00028     Possibility& operator=(Possibility&& other) noexcept;
00029
00030     void update(std::string_view variable, int individual);
00031
00032     std::shared_ptr<ReferentSystem> referentSystem;
00033     std::unordered_map<int, int> assignment;
00034     int world;
00035 };
00036
00037 bool extends(const Possibility& p2, const Possibility& p1);
00038 bool operator<(const Possibility& p1, const Possibility& p2);
00039 std::expected<int, std::string> variableDenotation(std::string_view variable, const Possibility& p);
00040
00041 std::string str(const Possibility& p);
00042 std::string repr(const Possibility& p);
00043
00044 }
```

6.11 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 variable names 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)

6.12 referent_system.hpp

Go to the documentation of this file.

```
00001 #pragma once
00002
00003 #include <expected>
00004 #include <set>
00005 #include <string>
00006 #include <string_view>
00007 #include <unordered_map>
00008
00009 namespace iif_sadaf::talk::GSV {
00010
00021 struct ReferentSystem {
00022 public:
00023     ReferentSystem() = default;
00024     ReferentSystem(const ReferentSystem& other) = default;
00025     ReferentSystem& operator=(const ReferentSystem& other) = default;
00026     ReferentSystem(ReferentSystem&& other) noexcept;
00027     ReferentSystem& operator=(ReferentSystem&& other) noexcept;
00028
00029     std::expected<int, std::string> value(std::string_view variable) const;
00030
00031     int pegs = 0;
00032     std::unordered_map<std::string_view, int> variablePegAssociation = {};
00033 };
00034
00035 std::set<std::string_view> domain(const ReferentSystem& r);
00036 bool extends(const ReferentSystem& r2, const ReferentSystem& r1);
00037 std::string str(const ReferentSystem& r);
00038 std::string repr(const ReferentSystem& r);
00039
00040 }
```

6.13 formatter.hpp File Reference

```
#include "expression.hpp"
#include <string>
```

Classes

- struct iif_sadaf::talk::GSV::Formatter

  A visitor for formatting Expression objects.

Namespaces

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

6.14 formatter.hpp

Go to the documentation of this file.

```
00001 #include "expression.hpp"
00002
00003 #include <string>
00004
00005 namespace iif_sadaf::talk::GSV {
00006
00013 struct Formatter {
00014     std::string operator()(std::shared_ptr<UnaryNode> expr) const;
00015     std::string operator()(std::shared_ptr<BinaryNode> expr) const;
00016     std::string operator()(std::shared_ptr<QuantificationNode> expr) const;
00017     std::string operator()(std::shared_ptr<PredicationNode> expr) const;
00018     std::string operator()(std::shared_ptr<IdentityNode> expr) const;
00019 };
00020
00021 }
```

6.15 evaluator.cpp File Reference

```
#include "evaluator.hpp"
#include <algorithm>
#include <expected>
#include <format>
#include <functional>
#include <ranges>
#include <stdexcept>
#include "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 Expression &expr, const InformationState &input_state, const IModel &model)

  Evaluates a logical expression within a given information state and model.

6.16 evaluator.cpp

Go to the documentation of this file.

```
00001 #include "evaluator.hpp"
00002
00003 #include <algorithm>
00004 #include <expected>
00005 #include <format>
00006 #include <functional>
00007 #include <ranges>
00008 #include <stdexcept>
00009
00010 #include "formatter.hpp"
00011 #include "possibility.hpp"
00012
00013 namespace iif_sadaf::talk::GSV {
00014
00015 namespace {
00016
00017 void filter(InformationState& state, const std::function<bool(const Possibility&)>& predicate) {
00018     for (auto it = state.begin(); it != state.end(); ) {
00019         if (!predicate(*it)) {
00020             it = state.erase(it);
00021         }
00022         else {
00023             ++it;
00024         }
00025     }
00026 }
00027
00028 } // ANONYMOUS NAMESPACE
00029
00050 std::expected<InformationState, std::string> Evaluator::operator()(const std::shared_ptr<UnaryNode>&
      expr, std::variant<std::pair<InformationState, const IModel*>> params) const
00051 {
00052     const auto prejacent_update = std::visit(Evaluator(), expr->scope, params);
00053
00054     if (!prejacent_update.has_value()) {
00055         return std::unexpected(
00056             std::format(
00057                 "In evaluating formula {}:\n{}",
00058                 std::visit(Formatter(), Expression(expr)),
00059                 prejacent_update.error()
00060             )
00061         );
00062     }
00063
00064     InformationState& input_state = std::get<std::pair<InformationState, const IModel*>>(params).first;
00065
00066     if (expr->op == Operator::E_POS) {
00067         if (prejacent_update.value().empty()) {
00068             input_state.clear();
00069         }
00070     }
00071     else if (expr->op == Operator::E_NEC) {
00072         if (!subsistsIn(input_state, prejacent_update.value())) {
00073             input_state.clear();
00074         }
00075     }
00076     else if (expr->op == Operator::NEG) {
00077         filter(input_state, [&](const Possibility& p) -> bool { return !subsistsIn(p,
      prejacent_update.value()); });
00078     }
00079     else {
00080         return std::unexpected(
00081             std::format(
00082                 "In evaluating formula {}:\n{}",
00083                 std::visit(Formatter(), Expression(expr)),
00084                 "Invalid unary operator"
00085             )
00086         );
00087     }
00088
00089     return std::move(input_state);
00090 }
00091
00117 std::expected<InformationState, std::string> Evaluator::operator()(const std::shared_ptr<BinaryNode>&
      expr, std::variant<std::pair<InformationState, const IModel*>> params) const
00118 {
00119     const IModel* model = (std::get<std::pair<InformationState, const IModel*>>(params)).second;
00120
00121     // Conjunction is sequential update, treated separately
00122     if (expr->op == Operator::CON) {
00123         const auto lhs_update = std::visit(Evaluator(), expr->lhs, params);
00124
```

```
00125         if (!lhs_update.has_value()) {
00126             return std::unexpected(
00127                 std::format(
00128                     "In evaluating formula {}:\n{}",
00129                     std::visit(Formatter(), Expression(expr)),
00130                     lhs_update.error()
00131                 )
00132             );
00133         }
00134 
00135         return std::visit(
00136                 Evaluator(),
00137                 expr->rhs,
00138                 std::variant<std::pair<InformationState, const
      IModel*>>(std::make_pair(lhs_update.value(), model)
00139             )
00140         );
00141     }
00142 
00143     // All other updates are filtering updates
00144     InformationState& input_state = (std::get<std::pair<InformationState, const
      IModel*>>(params)).first;
00145     const auto hypothetical_lhs_update = std::visit(Evaluator(), expr->lhs, params);
00146 
00147     if (!hypothetical_lhs_update.has_value()) {
00148         return std::unexpected(
00149             std::format(
00150                 "In evaluating formula {}:\n{}",
00151                 std::visit(Formatter(), Expression(expr)),
00152                 hypothetical_lhs_update.error()
00153             )
00154         );
00155     }
00156 
00157     if (expr->op == Operator::DIS) {
00158         const auto negated_lhs_update = std::visit(Evaluator(), negate(expr->lhs), params);
00159 
00160         if (!negated_lhs_update.has_value()) {
00161             return std::unexpected(
00162                 std::format(
00163                     "In evaluating formula {}:\n{}",
00164                     std::visit(Formatter(), Expression(expr)),
00165                     negated_lhs_update.error()
00166                 )
00167             );
00168         }
00169 
00170         const auto hypothetical_rhs_update = std::visit(
00171             Evaluator(),
00172             expr->rhs,
00173             std::variant<std::pair<InformationState, const
      IModel*>>(std::make_pair(negated_lhs_update.value(), model))
00174         );
00175 
00176         if (!hypothetical_rhs_update.has_value()) {
00177             return std::unexpected(
00178                 std::format(
00179                     "In evaluating formula {}:\n{}",
00180                     std::visit(Formatter(), Expression(expr)),
00181                     hypothetical_rhs_update.error()
00182                 )
00183             );
00184         }
00185 
00186         const auto in_lhs_or_in_rhs = [&](const Possibility& p) -> bool {
00187             return hypothetical_lhs_update.value().contains(p) ||
      hypothetical_rhs_update.value().contains(p);
00188         };
00189 
00190         filter(input_state, in_lhs_or_in_rhs);
00191     }
00192     else if (expr->op == Operator::IMP) {
00193         const auto hypothetical_consequent_update = std::visit(
00194             Evaluator(),
00195             expr->rhs,
00196             std::variant<std::pair<InformationState, const
      IModel*>>(std::make_pair(hypothetical_lhs_update.value(), model))
00197         );
00198 
00199         if (!hypothetical_consequent_update.has_value()) {
00200             return std::unexpected(
00201                 std::format(
00202                     "In evaluating formula {}:\n{}",
00203                     std::visit(Formatter(), Expression(expr)),
00204                     hypothetical_consequent_update.error()
00205                 )
00206             );
```

```
00207         }
00208
00209         const auto all_descendants_subsist = [&](const Possibility& p) -> bool {
00210             const auto not_descendant_or_subsists = [&](const Possibility& p_star) -> bool {
00211                 return !isDescendantOf(p_star, p, hypothetical_lhs_update.value()) ||
      subsistsIn(p_star, hypothetical_consequent_update.value());
00212             };
00213             return std::ranges::all_of(hypothetical_lhs_update.value(), not_descendant_or_subsists);
00214         };
00215
00216         const auto if_subsists_all_descendants_do = [&](const Possibility& p) -> bool {
00217             return !subsistsIn(p, hypothetical_lhs_update.value()) || all_descendants_subsist(p);
00218         };
00219
00220         filter(input_state, if_subsists_all_descendants_do);
00221     }
00222     else {
00223         return std::unexpected(
00224             std::format(
00225                 "In evaluating formula {}:\n{}",
00226                 std::visit(Formatter(), Expression(expr)),
00227                 "Invalid operator for binary formula"
00228             )
00229         );
00230     }
00231
00232     return std::move(input_state);
00233 }
00234
00254 std::expected<InformationState, std::string> Evaluator::operator()(const
      std::shared_ptr<QuantificationNode>& expr, std::variant<std::pair<InformationState, const IModel*>»
      params) const
00255 {
00256     InformationState& input_state = (std::get<std::pair<InformationState, const
      IModel*>»(params)).first;
00257     const IModel* model = (std::get<std::pair<InformationState, const IModel*>»(params)).second;
00258
00259     if (expr->quantifier == Quantifier::EXISTENTIAL) {
00260         std::vector<InformationState> all_state_variants;
00261
00262         for (const int i : std::views::iota(0, model->domain_cardinality())) {
00263             const InformationState s_variant = update(input_state, expr->variable.literal, i);
00264             const auto hypothetical_s_variant_update = std::visit(
00265                 Evaluator(),
00266                 expr->scope,
00267                 std::variant<std::pair<InformationState, const IModel*>(std::make_pair(s_variant,
      model))
00268             );
00269
00270             if (!hypothetical_s_variant_update.has_value()) {
00271                 return std::unexpected(
00272                     std::format(
00273                         "In evaluating formula {}:\n{}",
00274                         std::visit(Formatter(), Expression(expr)),
00275                         hypothetical_s_variant_update.error()
00276                     )
00277                 );
00278             }
00279
00280             all_state_variants.push_back(hypothetical_s_variant_update.value());
00281         }
00282
00283         InformationState output;
00284         for (const auto& state_variant : all_state_variants) {
00285             for (const auto& p : state_variant) {
00286                 output.insert(p);
00287             }
00288         }
00289
00290         return output;
00291     }
00292     if (expr->quantifier == Quantifier::UNIVERSAL) {
00293         std::vector<InformationState> all_hypothetical_updates;
00294
00295         for (const int d : std::views::iota(0, model->domain_cardinality())) {
00296             const auto hypothetical_update = std::visit(
00297                 Evaluator(),
00298                 expr->scope,
00299                 std::variant<std::pair<InformationState, const
      IModel*>(std::make_pair(update(input_state, expr->variable.literal, d), model))
00300             );
00301
00302             if (!hypothetical_update.has_value()) {
00303                 return std::unexpected(
00304                     std::format(
00305                         "In evaluating formula {}:\n{}",
00306                         std::visit(Formatter(), Expression(expr)),
```

```
00307                         hypothetical_update.error()
00308                     )
00309                 );
00310             }
00311
00312             all_hypothetical_updates.push_back(hypothetical_update.value());
00313         }
00314
00315         const auto subsists_in_all_hyp_updates = [&](const Possibility& p) -> bool {
00316             const auto p_subsists_in_hyp_update = [&](const InformationState& hypothetical_update) ->
      bool {
00317                 return subsistsIn(p, hypothetical_update);
00318             };
00319             return std::ranges::all_of(all_hypothetical_updates, p_subsists_in_hyp_update);
00320         };
00321
00322         filter(input_state, subsists_in_all_hyp_updates);
00323     }
00324     else {
00325         return std::unexpected(
00326             std::format(
00327                 "In evaluating formula {}:\n{}",
00328                 std::visit(Formatter(), Expression(expr)),
00329                 "Invalid quantifier"
00330             )
00331         );
00332     }
00333
00334     return std::move(input_state);
00335 }
00336
00360 std::expected<InformationState, std::string> Evaluator::operator()(const
      std::shared_ptr<IdentityNode>& expr, std::variant<std::pair<InformationState, const IModel*>> params)
      const
00361 {
00362     InformationState& input_state = (std::get<std::pair<InformationState, const
      IModel*>>(params)).first;
00363     const IModel& model = *(std::get<std::pair<InformationState, const IModel*>>(params)).second;
00364
00365     auto assigns_same_denotation = [&](const Possibility& p) -> bool {
00366         const auto lhs_denotation = expr->lhs.type == Term::Type::VARIABLE ?
      variableDenotation(expr->lhs.literal, p) : model.termInterpretation(expr->lhs.literal, p.world);
00367         const auto rhs_denotation = expr->rhs.type == Term::Type::VARIABLE ?
      variableDenotation(expr->rhs.literal, p) : model.termInterpretation(expr->rhs.literal, p.world);
00368
00369         if (!lhs_denotation.has_value()) {
00370             throw std::out_of_range(lhs_denotation.error());
00371         }
00372         if (!rhs_denotation.has_value()) {
00373             throw std::out_of_range(rhs_denotation.error());
00374         }
00375
00376         return lhs_denotation.value() == rhs_denotation.value();
00377     };
00378
00379     try {
00380         filter(input_state, assigns_same_denotation);
00381         return std::move(input_state);
00382     }
00383     catch (const std::out_of_range& e) {
00384         return std::unexpected(
00385             std::format(
00386                 "In evaluating formula {}:\n{}",
00387                 std::visit(Formatter(), Expression(expr)),
00388                 e.what()
00389             )
00390         );
00391     }
00392 }
00393
00417 std::expected<InformationState, std::string> Evaluator::operator()(const
      std::shared_ptr<PredicationNode>& expr, std::variant<std::pair<InformationState, const IModel*>>
      params) const
00418 {
00419     InformationState& input_state = (std::get<std::pair<InformationState, const
      IModel*>>(params)).first;
00420     const IModel& model = *(std::get<std::pair<InformationState, const IModel*>>(params)).second;
00421
00422     const auto tuple_in_extension = [&](const Possibility& p) -> bool {
00423         std::vector<int> tuple;
00424
00425         for (const Term& argument : expr->arguments) {
00426             const auto denotation = argument.type == Term::Type::VARIABLE ?
      variableDenotation(argument.literal, p) : model.termInterpretation(argument.literal, p.world);
00427             if (denotation.has_value()) {
00428                 tuple.push_back(denotation.value());
00429             }
```

```
00430             else {
00431                 throw std::out_of_range(denotation.error());
00432             }
00433         }
00434
00435         const auto predint = model.predicateInterpretation(expr->predicate, p.world);
00436         if (predint.has_value()) {
00437             return predint.value()->contains(tuple);
00438         }
00439         else {
00440             throw std::out_of_range(predint.error());
00441         }
00442     };
00443
00444     try {
00445         filter(input_state, tuple_in_extension);
00446         return std::move(input_state);
00447     }
00448     catch (const std::invalid_argument& e) {
00449         return std::unexpected(
00450             std::format(
00451                 "In evaluating formula {}:\n{}",
00452                 std::visit(Formatter(), Expression(expr)),
00453                 e.what()
00454             )
00455         );
00456     }
00457 }
00458
00476 std::expected<InformationState, std::string> evaluate(const Expression& expr, const InformationState&
    input_state, const IModel& model)
00477 {
00478     return std::visit(
00479         Evaluator(),
00480         expr,
00481         std::variant<std::pair<InformationState, const IModel*>>(std::make_pair(input_state, &model))
00482     );
00483 }
00484
00485 }
```

6.17 semantic_relations.cpp File Reference

```
#include "semantic_relations.hpp"
#include <algorithm>
#include <format>
#include <functional>
#include <ranges>
#include <stdexcept>
#include <variant>
#include <vector>
#include "evaluator.hpp"
#include "formatter.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 Expression &expr, const InformationState &state, const IModel &model)

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

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

  Determines whether two expressions are logically equivalent within a given model.

6.18 semantic_relations.cpp

Go to the documentation of this file.

```
00001 #include "semantic_relations.hpp"
00002
00003 #include <algorithm>
00004 #include <format>
00005 #include <functional>
00006 #include <ranges>
00007 #include <stdexcept>
00008 #include <variant>
00009 #include <vector>
00010
00011 #include "evaluator.hpp"
00012 #include "formatter.hpp"
00013 #include "imodel.hpp"
00014 #include "information_state.hpp"
00015 #include "possibility.hpp"
00016
00017 namespace iif_sadaf::talk::GSV {
00018
00037 std::expected<bool, std::string> consistent(const Expression& expr, const InformationState& state,
      const IModel& model)
00038 {
00039     const auto hypothetical_update = evaluate(expr, state, model);
00040
00041     if (!hypothetical_update.has_value()) {
00042         return std::unexpected(
00043             std::format(
00044                 "In evaluating formula {}:\n{}",
00045                 std::visit(Formatter(), Expression(expr)),
00046                 hypothetical_update.error()
00047             )
00048         );
00049     }
00050
00051     return !hypothetical_update.value().empty();
00052 }
00053
```

```
00067 std::expected<bool, std::string> allows(const InformationState& state, const Expression& expr, const
      IModel& model)
00068 {
00069     return consistent(expr, state, model);
00070 }
00071
00085 std::expected<bool, std::string> supports(const InformationState& state, const Expression& expr, const
      IModel& model)
00086 {
00087     const auto hypothetical_update = evaluate(expr, state, model);
00088
00089     if (!hypothetical_update.has_value()) {
00090         return std::unexpected(
00091             std::format(
00092                 "In evaluating formula {}:\n{}",
00093                 std::visit(Formatter(), Expression(expr)),
00094                 hypothetical_update.error()
00095             )
00096         );
00097     }
00098
00099     return subsistsIn(state, hypothetical_update.value());
00100 }
00101
00115 std::expected<bool, std::string> isSupportedBy(const Expression& expr, const InformationState& state,
      const IModel& model)
00116 {
00117     return supports(state, expr, model);
00118 }
00119
00120 namespace {
00121
00122 std::vector<InformationState> generateSubStates(int n, int k) {
00123     std::vector<InformationState> result;
00124
00125     if (k == 0) {
00126         result.push_back(InformationState());
00127         return result;
00128     }
00129
00130     if (k > n + 1) {
00131         return result;
00132     }
00133
00134     int estimate = 1;
00135     for (int i = 1; i <= k; i++) {
00136         estimate = estimate * (n + 2 - i) / i;
00137     }
00138     result.reserve(estimate);
00139
00140     std::function<void(int, InformationState&)> backtrack =
00141         [&](int start, InformationState& current) {
00142             if (current.size() == k) {
00143                 result.push_back(current);
00144                 return;
00145             }
00146
00147             ReferentSystem r;
00148
00149             for (int i = start; i <= n; ++i) {
00150                 Possibility p(std::make_shared<ReferentSystem>(r), i);
00151                 current.insert(p);
00152
00153                 backtrack(i + 1, current);
00154
00155                 current.erase(p);
00156             }
00157         };
00158
00159     InformationState current;
00160     backtrack(0, current);
00161
00162     return result;
00163 }
00164
00165 } // ANONYMOUS NAMESPACE
00166
00181 std::expected<bool, std::string> consistent(const Expression& expr, const IModel& model)
00182 {
00183     for (const int i : std::views::iota(0, model.world_cardinality())) {
00184         std::vector<InformationState> states = generateSubStates(model.world_cardinality() - 1, i);
00185         const auto is_consistent = [&](const InformationState& state) -> bool {
00186             const auto result = consistent(expr, state, model);
00187             if (!result.has_value()) {
00188                 throw std::runtime_error(result.error());
00189             }
00190             return result.value();
```

```
00191         };
00192         try {
00193             if (!std::ranges::any_of(states, is_consistent)) {
00194                 return false;
00195             }
00196         }
00197         catch (const std::runtime_error& e) {
00198             return std::unexpected(e.what());
00199         }
00200     }
00201     return true;
00202 }
00203
00218 std::expected<bool, std::string> coherent(const Expression& expr, const IModel& model)
00219 {
00220     for (const int i : std::views::iota(0, model.world_cardinality())) {
00221         std::vector<InformationState> states = generateSubStates(model.world_cardinality() - 1, i);
00222         const auto is_not_empty_or_supports_expression = [&](const InformationState& state) -> bool {
00223             const auto result = supports(state, expr, model);
00224             if (!result.has_value()) {
00225                 throw std::runtime_error(result.error());
00226             }
00227             return !state.empty() && result.value();
00228         };
00229         try {
00230             if (!std::ranges::any_of(states, is_not_empty_or_supports_expression)) {
00231                 return false;
00232             }
00233         }
00234         catch (const std::runtime_error& e) {
00235             return std::unexpected(e.what());
00236         }
00237     }
00238     return true;
00239 }
00240
00256 std::expected<bool, std::string> entails(const std::vector<Expression>& premises, const Expression&
      conclusion, const IModel& model)
00257 {
00258     for (const int i : std::views::iota(0, model.world_cardinality())) {
00259         std::vector<InformationState> states = generateSubStates(model.world_cardinality() - 1, i);
00260         for (InformationState& input_state : states) {
00261             // Update input state with premises
00262             for (const Expression& expr : premises) {
00263                 const auto update = evaluate(expr, input_state, model);
00264                 if (!update.has_value()) {
00265                     return std::unexpected(
00266                         std::format(
00267                             "In evaluating formula {}:\n{}",
00268                             std::visit(Formatter(), Expression(expr)),
00269                             update.error()
00270                         )
00271                     );
00272                 }
00273                 input_state = update.value();
00274             }
00275
00276             // check if update with conclusion exists
00277             const auto update = evaluate(conclusion, input_state, model);
00278
00279             // update does not exist
00280             if (!update.has_value()) {
00281                 return std::unexpected(
00282                     std::format(
00283                         "In evaluating formula {}:\n{}",
00284                         std::visit(Formatter(), Expression(conclusion)),
00285                         update.error()
00286                     )
00287                 );
00288             }
00289
00290             // update exists, check for support
00291             const auto does_support = supports(input_state, conclusion, model);
00292             if (!does_support.has_value()) {
00293                 return std::unexpected(
00294                     std::format(
00295                         "In evaluating formula {}:\n{}",
00296                         std::visit(Formatter(), Expression(conclusion)),
00297                         does_support.error()
00298                     )
00299                 );
00300             }
00301             if (!does_support.value()) {
00302                 return false;
00303             }
00304         }
00305     }
```

```
00306     return true;
00307 }
00308
00309 namespace {
00310
00311 std::expected<bool, std::string> similar(const Possibility& p1, const Possibility& p2)
00312 {
00313     const auto have_same_denotation = [&](std::string_view variable) -> bool {
00314         const auto denotation_at_p1 = variableDenotation(variable, p1);
00315         const auto denotation_at_p2 = variableDenotation(variable, p2);
00316         if (!denotation_at_p1.has_value()) {
00317             throw std::invalid_argument(denotation_at_p1.error());
00318         }
00319         if (!denotation_at_p2.has_value()) {
00320             throw std::invalid_argument(denotation_at_p2.error());
00321         }
00322         return denotation_at_p1.value() == denotation_at_p2.value();
00323     };
00324
00325     try {
00326         return p1.world == p2.world
00327             && domain(*p1.referentSystem) == domain(*p2.referentSystem)
00328             && std::ranges::all_of(domain(*p1.referentSystem), have_same_denotation);
00329     }
00330     catch (const std::invalid_argument& e) {
00331         return std::unexpected(e.what());
00332     }
00333 }
00334
00335 std::expected<bool, std::string> similar(const InformationState& s1, const InformationState& s2)
00336 {
00337     const auto has_similar_possibility_in_s2 = [&](const Possibility p) -> bool {
00338         const auto is_similar_to_p = [&](const Possibility p_dash) -> bool {
00339             const auto comparison_result = similar(p, p_dash);
00340             if (!comparison_result.has_value()) {
00341                 throw std::invalid_argument(comparison_result.error());
00342             }
00343             return comparison_result.value();
00344         };
00345         return std::ranges::any_of(s2, is_similar_to_p);
00346     };
00347
00348     const auto has_similar_possibility_in_s1 = [&](const Possibility p) -> bool {
00349         const auto is_similar_to_p = [&](const Possibility p_dash) -> bool {
00350             const auto comparison_result = similar(p, p_dash);
00351             if (!comparison_result.has_value()) {
00352                 throw std::invalid_argument(comparison_result.error());
00353             }
00354             return comparison_result.value();
00355         };
00356         return std::ranges::any_of(s1, is_similar_to_p);
00357     };
00358
00359     try {
00360         return std::ranges::all_of(s1, has_similar_possibility_in_s2)
00361             && std::ranges::all_of(s2, has_similar_possibility_in_s1);
00362     }
00363     catch (const std::invalid_argument& e) {
00364         return std::unexpected(e.what());
00365     }
00366 }
00367
00368 } // ANONYMOUS NAMESPACE
00369
00384 std::expected<bool, std::string> equivalent(const Expression& expr1, const Expression& expr2, const
      IModel& model)
00385 {
00386     for (const int i : std::views::iota(0, model.world_cardinality())) {
00387         std::vector<InformationState> states = generateSubStates(model.world_cardinality() - 1, i);
00388
00389         const auto dissimilar_updates = [&](const InformationState& state) ->bool {
00390             const auto expr1_update = evaluate(expr1, state, model);
00391             if (!expr1_update.has_value()) {
00392                 throw std::invalid_argument(expr1_update.error());
00393             }
00394             const auto expr2_update = evaluate(expr2, state, model);
00395             if (!expr2_update.has_value()) {
00396                 throw std::invalid_argument(expr2_update.error());
00397             }
00398
00399             return !similar(expr1_update.value(), expr2_update.value());
00400         };
00401
00402         try {
00403             if (std::ranges::any_of(states, dissimilar_updates)) {
00404                 return false;
00405             }
```

```
00406         }
00407         catch (const std::invalid_argument& e) {
00408             return std::unexpected(e.what());
00409         }
00410     }
00411
00412     return true;
00413 }
00414
00415 }
```

6.19 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)

6.20 information_state.cpp

Go to the documentation of this file.

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

```
00138     for (const Possibility& p : state) {
00139         desc += str(p);
00140         desc += "--------------------\n";
00141     }
00142 
00143     desc.pop_back();
00144 
00145     return desc;
00146 }
00147 
00148 std::string repr(const InformationState& state)
00149 {
00150     std::string desc = "Information State : [\n";
00151 
00152     for (const Possibility& p : state) {
00153         desc += "  " + repr(p) + "\n";
00154     }
00155 
00156     return desc + "]";
00157 }
00158 
00159 }
```

6.21 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)

6.22 possibility.cpp

Go to the documentation of this file.

```
00001 #include "possibility.hpp"
00002 
00003 #include <algorithm>
00004 
00005 namespace iif_sadaf::talk::GSV {
00006 
00007 Possibility::Possibility(std::shared_ptr<ReferentSystem> r_system, int world)
00008     : referentSystem(r_system)
00009     , assignment({})
00010     , world(world)
00011 { }
00012 
00013 Possibility::Possibility(Possibility&& other) noexcept
00014     : referentSystem(std::move(other.referentSystem))
```

```
00015     , assignment(std::move(other.assignment))
00016     , world(other.world)
00017 { }
00018
00019 Possibility& Possibility::operator=(Possibility&& other) noexcept
00020 {
00021     if (this != &other) {
00022         this->referentSystem = std::move(other.referentSystem);
00023         this->assignment.clear();
00024         this->assignment.swap(other.assignment);
00025         this->world = other.world;
00026     }
00027     return *this;
00028 }
00029
00039 void Possibility::update(std::string_view variable, int individual)
00040 {
00041     referentSystem->variablePegAssociation[variable] = ++(referentSystem->pegs);
00042     assignment[referentSystem->pegs] = individual;
00043 }
00044
00045 /*
00046 * NON-MEMBER FUNCTIONS
00047 */
00048
00060 bool extends(const Possibility& p2, const Possibility& p1)
00061 {
00062     const auto peg_is_new_or_maintains_assignment = [&](const std::pair<int, int>& map) -> bool {
00063         const int peg = map.first;
00064
00065         return !p1.assignment.contains(peg) || (p1.assignment.at(peg) == p2.assignment.at(peg));
00066     };
00067
00068     return (p1.world == p2.world) && std::ranges::all_of(p2.assignment,
      peg_is_new_or_maintains_assignment);
00069 }
00070
00071 bool operator<(const Possibility& p1, const Possibility& p2)
00072 {
00073     return p1.world < p2.world;
00074 }
00075
00090 std::expected<int, std::string> variableDenotation(std::string_view variable, const Possibility& p)
00091 {
00092     const auto peg = p.referentSystem->value(variable);
00093
00094     if (!peg.has_value()) {
00095         return std::unexpected(peg.error());
00096     }
00097
00098     // Whenever variable exists in referent system, assignment is guaranteed to
00099     // contain the corresponding peg, so there is no need to check for existence
00100     // before returning
00101     return p.assignment.at(peg.value());
00102 }
00103
00104 std::string str(const Possibility& p)
00105 {
00106     std::string desc = "[ ] Referent System:\n" + str(*p.referentSystem);
00107     desc += "[ ] Assignment function: \n";
00108
00109     if (p.assignment.empty()) {
00110         desc += "   [ empty ]\n";
00111
00112     }
00113     else {
00114         for (const auto& [peg, individual] : p.assignment) {
00115             desc += "   - peg_" + std::to_string(peg) + " -> e_" + std::to_string(individual) + "\n";
00116         }
00117     }
00118
00119     desc += "[ ] Possible world: w_" + std::to_string(p.world) + "\n";
00120
00121     return desc;
00122 }
00123
00124 std::string repr(const Possibility& p)
00125 {
00126     std::string desc = "Possibility : [ " + repr(*p.referentSystem) + ", Assignment : [ ";
00127
00128     if (p.assignment.empty()) {
00129         desc += "]";
00130     }
00131     else {
00132         for (const auto& [peg, individual] : p.assignment) {
00133             desc += "{ " + std::to_string(peg) + " : " + std::to_string(individual) + " }, ";
00134         }
```

```
00135         desc.resize(desc.size() - 2);
00136         desc += " ]";
00137     }
00138
00139     desc += ", World : " + std::to_string(p.world) + " ]";
00140
00141     return desc;
00142 }
00143
00144 }
```

6.23 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 variable names 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)

6.24 referent_system.cpp

Go to the documentation of this file.

```
00001 #include "referent_system.hpp"
00002
00003 #include <algorithm>
00004 #include <format>
00005 #include <stdexcept>
00006
00007 namespace iif_sadaf::talk::GSV {
00008
00018 std::set<std::string_view> domain(const ReferentSystem& r)
00019 {
00020     std::set<std::string_view> domain;
00021     for (const auto& [variable, peg] : r.variablePegAssociation) {
00022         domain.insert(variable);
00023     }
00024
00025     return domain;
00026 }
00027
00028 ReferentSystem::ReferentSystem(ReferentSystem&& other) noexcept
00029     : pegs(other.pegs)
00030     , variablePegAssociation(std::move(other.variablePegAssociation))
00031 { }
00032
00033 ReferentSystem& ReferentSystem::operator=(ReferentSystem&& other) noexcept
00034 {
00035     if (this != &other) {
```

```
00036         this->pegs = other.pegs;
00037         this->variablePegAssociation = std::move(other.variablePegAssociation);
00038         other.pegs = 0;
00039     }
00040     return *this;
00041 }
00042
00054 std::expected<int, std::string> ReferentSystem::value(std::string_view variable) const
00055 {
00056     if (!variablePegAssociation.contains(variable)) {
00057         return std::unexpected(std::format("Referent system does not contain variable {}",
      std::string(variable)));
00058     }
00059
00060     return variablePegAssociation.at(variable);
00061 }
00062
00077
00078  // TODO check that these calls to value() are safe
00079 bool extends(const ReferentSystem& r2, const ReferentSystem& r1)
00080 {
00081     if (r1.pegs > r2.pegs) {
00082         return false;
00083     }
00084
00085     std::set<std::string_view> domain_r1 = domain(r1);
00086     std::set<std::string_view> domain_r2 = domain(r2);
00087
00088     if (!std::ranges::includes(domain_r2, domain_r1)) {
00089         return false;
00090     }
00091
00092     const auto old_var_same_or_new_peg = [&](std::string_view variable) -> bool {
00093         return r1.value(variable).value() == r2.value(variable).value() || r1.pegs <=
      r2.value(variable).value();
00094     };
00095
00096     if (!std::ranges::all_of(domain_r1, old_var_same_or_new_peg)) {
00097         return false;
00098     }
00099
00100     const auto new_var_new_peg = [&](std::string_view variable) -> bool {
00101         return domain_r1.contains(variable) || r1.pegs <= r2.value(variable).value();
00102     };
00103
00104     return std::ranges::all_of(domain_r2, new_var_new_peg);
00105 }
00106
00107 std::string str(const ReferentSystem& r)
00108 {
00109     std::string desc = "Number of pegs: " + std::to_string(r.pegs) + "\n";
00110     desc += "Variable to peg association:\n";
00111
00112     if (r.variablePegAssociation.empty()) {
00113         desc += "  [ empty ]\n";
00114         return desc;
00115     }
00116
00117     for (const auto& [variable, peg] : r.variablePegAssociation) {
00118         desc += "  - " + std::string(variable) + " -> peg_" + std::to_string(peg) + "\n";
00119     }
00120
00121     return desc;
00122 }
00123
00124 std::string repr(const ReferentSystem& r)
00125 {
00126     std::string desc = "R-System : [ ";
00127
00128     if (r.variablePegAssociation.empty()) {
00129         return desc + "]";
00130     }
00131
00132     for (const auto& [variable, peg] : r.variablePegAssociation) {
00133         desc += "{ " + std::string(variable) + " : " + std::to_string(peg) + " }, ";
00134     }
00135
00136     desc.resize(desc.size() - 2);
00137
00138     return desc + " ]";
00139 }
00140
00141 }
```

6.25 formatter.cpp File Reference

```
#include "formatter.hpp"
```

Namespaces

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

6.26 formatter.cpp

Go to the documentation of this file.

```
00001 #include "formatter.hpp"
00002
00003 namespace iif_sadaf::talk::GSV {
00004
00005 std::string Formatter::operator()(std::shared_ptr<UnaryNode> expr) const
00006 {
00007     std::string op = expr->op == Operator::NEG ? "¬" :
00008         expr->op == Operator::E_NEC ? "□" :
00009         "◇";
00010
00011     if (typeid(expr->scope) == typeid(std::shared_ptr<IdentityNode>)) {
00012         std::string prejacent = std::visit(Formatter(), expr->scope);
00013         const std::string::size_type pos = prejacent.find("=");
00014         prejacent.replace(pos, 1, "≠");
00015         return prejacent;
00016     }
00017
00018     return op + std::visit(Formatter(), expr->scope);
00019 }
00020
00021 std::string Formatter::operator()(std::shared_ptr<BinaryNode> expr) const
00022 {
00023     std::string op = expr->op == Operator::CON ? " ∧ " :
00024         expr->op == Operator::DIS ? " ∨ " :
00025         " → ";
00026
00027     return "(" + std::visit(Formatter(), expr->lhs) + op + std::visit(Formatter(), expr->rhs) + ")";
00028 }
00029
00030 std::string Formatter::operator()(std::shared_ptr<QuantificationNode> expr) const
00031 {
00032     std::string quantifier = expr->quantifier == Quantifier::EXISTENTIAL ? "∃" : "∀";
00033
00034     return quantifier + expr->variable.literal + " " + std::visit(Formatter(), expr->scope);
00035 }
00036
00037 std::string Formatter::operator()(std::shared_ptr<PredicationNode> expr) const
00038 {
00039     std::string formula = expr->predicate + "(";
00040
00041     for (const Term& arg : expr->arguments) {
00042         formula += arg.literal + ", ";
00043     }
00044
00045     formula.resize(formula.size() - 2);
00046
00047     return formula + ")";
00048 }
00049
00050 std::string Formatter::operator()(std::shared_ptr<IdentityNode> expr) const
00051 {
00052     return expr->lhs.literal + " = " + expr->rhs.literal;
00053 }
00054
00055 }
```

Index