NTNU
Innovation and Creativity

Representing and reasoning with constraints in Creek

Martin Stige

Master of Science in Informatics
Submission date: June 2006
Supervisor: Agnar Aamodt, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

This work studies constraint mechanisms in frame-based knowledge representation systems with the aim of improving the knowledge modelling abilities of the TrollCreek system. TrollCreek is an implementation of Creek, an architecture for case based reasoning (CBR) that uses an explicit frame-based knowledge model to guide the CBR process. The objective of this project is to develop a constraint mechanism for TrollCreek. In doing this the earlier Lisp implementation of Creek and four other frame-based systems are examined with emphasize on their constraint mechanisms. Based on these systems a constraint mechanism for TrollCreek is discussed and specified. The part of the mechanism considered most central is implemented and evaluated.

Problem description

Reasoning in the system Creek is a combination of case-based and modell based reasoning. The mechanisms in the model-based part are based on inheriting properties along certain relations (plausible inheritance). In an earlier version of the system, implemented in Lisp, there were some mechanisms for defining constraints on entities and values. The current implementation, TrollCreek, implemented in Java, has no general mechanisms for handling constraints.

The task to be solved is to describe constraint mechanisms in frame-based systems, define a constraint mechanism in the Creek architecture generally and specify and implement the most essential constraints in TrollCreek. This should be done as a constraint language integrated in the existing representation of Creek. More concrete this project involves first, defining a coarse specification of a constraint language for Creek, and second define some important mechanisms more concrete. The second mechanisms should be implemented, tested and evaluated.

Acknowledgements

I would like to thank Professor Agnar Aamodt for supervising me in this project and for valuable input when writing this thesis.
I also would like to thank Frode Sørmo for helping me to understand the implementation of TrollCreek.

Table of contents

Appendix

Figures

Tables

1 Introduction

1.1 Background and motivation

The project is done in the context of the Creek framework for case based reasoning [2]. Creek is an abbreviation for *Case based Reasoning through Extensive Explicit Knowledge*. Creek and its knowledge representation language CreekL were originally implemented in Lisp. This implementation will from now on be referred to as LispCreek. Later an experimental java version of Creek was developed [17]. The implementational parts of this project are done on the java version TrollCreek [24]. The extensive and explicit knowledge is represented in a frame-based knowledge representation system that is used both for representing the cases and for storing an underlying knowledge model used to support the inference process.

Most frame-based systems have some kind of constraint mechanism. Constraints are useful in that they help the knowledge engineer to keep the model consistent. They also ad some semantics to a model when defining what values are legal for different slots. LispCreek has a constraint mechanism, but this is not yet implemented in the java version.

1.2 Goal

The goal of this project is to improve the knowledge modelling abilities in TrollCreek's frame system by introducing the ability to define constraints on the knowledge model.

To be more concrete this means:

- Discuss how constraint can be included in the TrollCreek system.
- Specify of a full constraint system for TrollCreek.
- Implement a fully working version of the most important part of this system.
- Evaluate the methods implemented.

1.3 Methodology

The project is based on analytical and experimental methods. First the concept of constraints in frame-based knowledge representations will be analysed through a study of similar systems in the literature. Based on what these systems can teach us, a constraint system for the Creek framework will be specified. The most important parts of the constraint system will then be designed and implemented into TrollCreek. Finally the system will be tested and evaluated.

1.4 Structure of the text

The text starts by discussing frame-based systems in general and five concrete systems with emphasise on knowledge representation and constraint mechanisms. A general framework for describing constraints is defined and the constraint mechanisms of the five systems are described with respect to this. Chapter 3 gives a brief overview of the TrollCreek system, before we in chapter 4 discuss the functional specifications of the constraint mechanism to develop. Chapter 5 discuss how the constraints can be designed and implemented and chapter 6 describes the result as the user of the system will see it. Chapter 7 tests and evaluates the implemented system. Functionality not implemented is discussed and possible solutions are sketched. The final chapter summarises the work.

2 Related systems

2.1 Introduction

2.1.1 Frame-based knowledge representation systems

Like many other knowledge representation systems and languages, frames are an attempt to resemble the way human beings are storing knowledge. It seems like we are storing our knowledge in rather large chunks and that the different chunks are highly interconnected [16].

In frame-based knowledge representations knowledge describing a particular concept or object is organized as a frame [15]. Some systems have one kind of frame whereas other have two or more, such as class frames and instance frames. The frame usually contains a name and a set of slots. Examples of frames are shown in Table 2.1.

The slots represent properties of the frames with attribute-value pairs `<slotname value>` or alternatively a triple containing framename, slotname and value in some order. Frame systems inspired from logics may view slots as binary predicates. In many frame systems the slots are complex structures that have facets describing the properties of the slot. The value of a slot may be a primitive such as a text string or an integer, or it may be another frame. Most systems allow multiple values for slots and some systems support procedural attachments. These attachments can be used to compute the slot value, or they can be triggers used to make consistency checking or updates of other slots. The triggers can be trigged by updates on slots.

As mentioned a slot can be viewed as a binary predicate stating a relation between two concepts (frame and value). When discussing the concept of a slot we need a terminology to use when referring to the two arguments off the slot. As we will see later, the TrollCreek system views the slots as relations between frames. A relation goes *from* a frame *to* the value of the slot. We will therefore use *from* and *to* or *from-frame* and *to-frame* to designate these two concepts (Figure 2.1). The *to-frame* will also be referred to as the *value* of the slot.

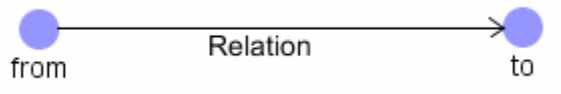


Figure 2.1 - The arguments of a slot

In most frame-based knowledge representations inheritance is the central inference mechanism [15]. The frames are arranged in a taxonomic hierarchy of generalization-specialization relations[1] where the most general frame is the top. Properties of a frame are propagated down the hierarchy to its specialization frames. Many systems support multiple-inheritance. This means that a frame can inherit from many other frames. In these systems the hierarchy can look more like a directed graph (with possible cycles) than a tree structure.

[1] This relation is also commonly called parent-child, is-a, superclass-subclass, AKO (a kind of).

```
(<Frame#1>
      (<Slot#1>   <slot value>)
      (<Slot#2>
            (<facet#1>   <facet value>)
            (<facet#2>   <facet value>))
)

(car
      (subclass-of
            (value                  vehicle))
      (has-subclass
            (value                  sports-car))
      (has-number-of-wheels
            (default                4)
            (cardinality            0..1))
      (has-part
            (value                  bodywork)
            (value                  engine)
            (value                  wheel)
            (cardinality            1..n))
      (has-number-of-seats
            (default                5)
            (value-constraint       (and (> value 0) (< value 9))))
)
```

Table 2.1 - Frame

At the top we see a template frame. Slot#1 has only a value wile Slot#2 has two facets describing it. In some frame systems the value of a slot is stored as a facet. This is exemplified in the frame *Car*. This frame also demonstrates two constraint facets, cardinality and value-constraint, and the default facet.

2.1.2 Constraints in frame-based systems

Sometimes simple slot values do not offer the expressive power we want. Logic is maybe the most expressive knowledge representation used today, when it comes to syntactical constructs. It provides mechanisms for representing incomplete knowledge, disjunctions and conjunctions. Many frame-based languages have constraint- or restriction mechanisms that extend the frame system with some of the abilities from logic. These mechanisms can be used to constrain possible values and combinations of values, perform consistency checking and express incomplete knowledge.

2.1.3 Types of constraints

There are many ways to describe and categorize constraints and different authors use different names for them. To make it easier to compare different constraint mechanisms, we will define some names to use for the different constraint types. The names are the most common names used in the literature. We will also give a formal description of the constraint types based on first order predicate logic. In the description we use the following notation:

- Dot-notation to access slot values. Ex `John.Father` means the slot *Father* on the frame *John.*
- Predicates are capitalized.
- Functions start with lower case letters.
- Variables are written in lower case.
- Constants/Frame instances are capitalized

Cardinality constraint constrains how many values are allowed for a slot on a frame. This may be a lower bound, an upper bound or an interval.

```
Cardinality(frame, slot, min, max) →
GreterThanOrEqual(numberOfValues(frame.slot), min) ∧
LessThanOrEqual(numberOfValues(frame.slot), max)

Cardinality(frame, slot, min) →
GreterThanOrEqual(numberOfValues(frame.slot), min)

Cardinality(frame, slot, max) →
LessThanOrEqual(numberOfValues(frame.slot), max)
```

Ex:
The frame *person* may have one or no value for the slot *spouse* – at least in our culture.

```
Cardinality(Person, Spouse, 0, 1) →
GreterThanOrEqual(numberOfValues (Person.Spouse), 0) ∧
LessThanOrEqual(numberOfValues (Person.Spouse), 1)
```

Concrete value constraint constrain the value of a slot, either by defining a list of all the allowed values (implicitly disallowing all other values), or by listing all disallowed values (implicitly allowing all other values).

ConcreteValueConstraint(frame, slot, {x_0, x_1… x_n}) →
frame.slot ∈ {x_0, x_1 ... x_n}

Ex:
The value of the slot *gender* of the frame *person* must be in the list {*male, female}.*

```
ConcreteValueConstraint(Person, Gender, {Male, Female}) →
Person.Gender ∈ {Male, Female}
```

Value range constraint constrains the allowed range of values for a numeric slot. This may be a lower bound, an upper bound or an interval.

```
ValueRangeConstraint(frame, slot, min, max) →
GreaterThanOrEqual(frame.slot, min) ∧ LessThanOrEqual(frame.slot, max)

ValueRangeConstraint(frame, slot, min) →
GreaterThanOrEqual(frame.slot, min)

ValueRangeConstraint(frame, slot, max) →
LessThanOrEqual(frame.slot, max)
```

Ex:
The slot *salary* of a frame *person* must have a value greater than 0.

```
ValueRangeConstraint(person, salary, 0) →
GreaterThanOrEqual(frame.slot, min)
```

The *age* of a *person* must be between 0 and 130.

```
ValueRangeConstraint(Person, Age, 0, 130) →
GreaterThanOrEqual(Person.Age, 0) ∧ LessThanOrEqual(Person.Age, 130)
```

Value class constraint constrains the value of a slot in a wider way than the two above. This constraint states that the value of a slot must be a subclass or an instance of another frame.

`ValueClassConstraint(frame, slot, super)` → `Isa(frame.slot, super)`

Ex: the value of a slot *father* must be a *male* (which again must be a person).

`ValueClassConstraint(Person, Father, Male)` → `Isa(Peron.Father, Male)`

General constraint. The final constraint type mentioned here is perhaps more correctly described as a class of constraints. This label is used to describe the ability to define complex constraints that combines or exceeds the simplest constraints.

The mechanisms described above allow us to constrain single values in the context of a frame and a slot. Sometimes it is useful to define constraints that include more than the to- and from-frames of a slot. In [9] an example knowledge base of famous persons and their doings is described. In this domain it can be reasonable to state that an event performed by a person must happen during the person's lifetime. Such constraints need more flexible mechanisms than those described above. Many systems offer this functionality through a logic-like language where the knowledge engineer can write logical expressions, being able to express arbitrarily complex constraints. Having such a constraint mechanism allows you to express all the other constraints to, seemingly taking the jobs from them. The above mentioned constraints are however useful short forms or syntactic sugar for the most used constraints.

Variations over the constraint types

The four first mechanisms constraining an individual slot value have (at least) two different subclasses. They can either be defined on a slot given a from-frame or on the slot without relating it to a frame. In the example above the value class constraint on *father* makes sense independent on which frame the constraint is defined on. It may be in an elephant family or among human beings. In both cases the father must be a male. In the following example, however it is reasonable to define the constraint on the combination of frame and slot:
Given the frame *engine* and the slot *part* it may be relevant to constrain the value to be an *engine part*. This constraint should be defined on the combination of slot and frame, because other things than engines can have parts too, and those parts are not necessarily engine parts.

So far constraints on the to-frame or value of a slot have been discusses. It may however be useful to constrain the from-frame of a relation as well. In Cyc [12] for example, you can use the constraint *makesSenseFor* on a slot to define which frames the slot may be used on.

Ex: the slot has*Colour* only makes sense for physical object-frames.

As we will see shortly, most frame systems defines the inverse of the slots. If we constrain the value class of some slot and this constraint is attached to the slot independent on what frame it constrains, we implicitly constrain the from-frame of the inverse slot. These two constraint types are thus related.

Summary

What is described here is by no means a complete list of constraint mechanisms suitable for frame-based systems. It is certainly possible to define other constraints than those mentioned here. It is also possible to combine some of these to make a shorter list. What we have called concrete value constraint can for example be

expressed using the value class constraint. You just make an intermediary frame acting as a super class of all the allowed values and saying that the value of the slot must be a member of this frame. The types of constraints given here do however cover most of the constraint mechanisms described in the literature. Through the text we will use these notions when comparing and describing constraints in different frame-based knowledge representations.

In the next sections we will describe five different frame-based systems with focus on their constraint mechanisms. The purpose of the discussion will be to see what we can learn from them in the process of developing constraint mechanisms in TrollCreek. When discussing the systems, the constraint mechanisms will be described with the names and terms used in the actual system. The discussion of each system is concluded with a summary section that relates the constraints in the mechanism to the framework described above.

2.2 LispCreek

As mentioned, Creek is a framework for case based reasoning. The architecture is an attempt to make a knowledge intensive CBR system, combining model based reasoning and case based reasoning. This section discusses the knowledge model and constraint aspects of LispCreek.

2.2.1 The language

As defined in the Creek framework [2] LispCreek [3] uses a frame-based approach for its knowledge model. A frame in Creek consists of a name and a set of zero or more slots. The slots consist of a name and a set of facets. The value of a slot is stored as a facet along with facets describing default values, constraints, inheritance type, value dimension etc. The value of a slot can be of four different types: concept (frame), number, string and lisp-expression.

Table 2.2 below shows a frame representing the concept *car.* The table gives examples of how inheritance, default values and constraints are represented in LispCreek.

```
car
      subclass-of             value       vehicle
      has-subclass            value       sports-car
      has-number-of-wheels    default     4
      has-part                value       bodywork
      has-function            value       transportation-of-people

      has-number-of-seats     value-constraint  (and
                                                  (> value 0)
                                                  (< value 9))
      has-owner               value-class       person
      has-age                 if-needed   (-
                                          *current-year*
                                          self.has-production-year)
```

Table 2.2 - Example of frame in LispCreek

The frame represents the general concept *car.* The first two slots *subclass-of* and *has-subclass* tells that *car* is subclass *vehicle* and superclass of *sports-car*. The next slot defines that cars on default have four wheels. This means that if an instance or subclass of car does not override this value, they have four wheels. Next we have two slots telling that cars has the part bodywork and has the function of transporting people. The next two slots are constraints. First *has-number-of-seats* has a value constraint. This is a lisp function defining that the value must be greater that 0 and less than 9. The *has-owner* slot has a value class constraint saying that the value of the slot must be a person. The last slot *has-age* has the facet if-needed. If for some instance of car there is no local, inherited or default value for *has-age* the lisp expression is evaluated and the result returned.

In Creek the whole representational structure is represented in the frame structure. Frames, slots, facets are defined as concepts/frames. This meta frame structure provides valuable semantics to the models and are essential in the frame matching mechanisms. Table 2.3 shows the meta frames for the relation *instance-of* and the concept *slot*.

```
instance-of
      used-to-describe  value  caused-by causes default (...)
      instance-of       value  transitive-relation
      has-inverse       value  has-instance

slot
      part-of            value  frame
      subclass-of        value  iso-thing
      has-part           value  facet
      has-subclass       value  transitive-relation
```

Table 2.3 - Examples of meta frames in LispCreek

2.2.2 Constraints

As mentioned, the slots have facets for describing constraints. There are two constraint facets: *value-constraint* and *value-class*. The *value-constraint* facet can contain a general lisp function describing an arbitrary constraint on the value. The other can hold a list of other frames that the value must be a specialization of. In 2.1.3 we saw that the constraint like LispCreek's *value-constraint* is flexible enough to offer all other constraint types. Consequently the value-class constraint does not introduce any new functionality. It can however be useful to offer a short form of the constraint syntax.

2.2.3 Summary

Types of constraints

Cardinality constraint
LispCreek do not have an explicit cardinality constraint. Cardinality constraints can however be expressed using the value-constraint mechanism.

Concrete value constraint
LispCreek do not have an explicit concrete value constraint. The constraint can however be expressed using the value-constraint mechanism.

Value range constraint
LispCreek do not have an explicit value range constraint. Value range constraints can however be expressed using the value-constraint mechanism.

Value class constraint
As described, slots in LispCreek have a facet called value-class. This mechanism corresponds to the value class constraint described in 2.1.3.

General constraint
The facet value-constraint allows the user to express arbitrary constraints. This equates the general constraint mechanism.

What can we learn from LispCreek?

TrollCreek do not have facets, but as in the lisp version slots and slot types are represented as frames in the knowledge base. This allows us to describe the slots with slots. It may be possible to use this to introduce facet-like behaviour to TrollCreek. Doing this we can introduce the constraint facets used in the previous Creek version. Using this approach the constraint mechanism will be tightly coupled to the rest of the knowledge representation.

2.3 The Knowledge Machine

The Knowledge Machine (KM) [7] is a knowledge representation language and reasoning engine. The knowledge is represented as frames, but KM is also influenced by logic. This combination makes KM very expressive and provides it with a clear, formal semantics. KM is an interpreter sitting on top of Lisp, and the user interface is through a read-eval-print cycle where the user can write assertions or queries. KM is a relatively mature language that has evolved over a long period of time. It is well documented.

2.3.1 The language

As in the object-oriented world, the frame concept contains both classes and instances. *Frames* contain *slots* that hold *values.* Because KM is closely connected to logic, the slots are binary relations that hold between an instance of a class and other instances. Slots are instances of the build in class Slot either directly or through one or more intermediate slots. Because the slots are frames themselves they can exist without being attached to a frame. The build in Slot class has a set of slots, for example domain, range and cardinality. The slot's slots are comparable to slot-facets in Creek [3] and Protégé [14]. The value of a slot may perfectly well be another frame as well as an atom or rules that are evaluated at query time. This allows the making of relationships between frames. The frames can form a taxonomy with single and multiple inheritance. Both classes and instances can have slots. The slots of a class can hold information about its sub- and super-classes and it can hold default values that are inherited to all descendants.

KM uses the simple data types number, integer, string and boolean, and the complex set (unordered, no duplicates), sequence (ordered, duplicates allowed) and bag (unordered, duplicates allowed). The system also offers the data type pair, but that is only a sequence of length two. KM offers a variety of set operations, both arithmetic and set expressions.

Table 2.4 and Table 2.5 show an example of a frame in KM. In the first table the class car is described and the second table shows an instance of car.

```
(Car has (superclasses (Vehicle)))
(every Car has
     (wheel-count (4))
     (uses-fuel-type (*Gas))
     (parts ((a Engine) (a Chassis))))
```

Table 2.4 - Class frame in KM

This frame represents the concept of a car. Car has vehicle as superclass. In KM names prefixed by an asterix (*) represents instances and names beginning with an upper case letter represents classes. The frame says that every car has four wheels, uses gas (an instance) as fuel and have the parts engine and chassis (classes).

```
(*myCar has
     (instance-of (Car))
     (parts (*myEngine *myChassis)))
```

Table 2.5 - Instance frame in KM

The table shows an instance of the class frame from Table 2.4. We see that the parts slot now has instances for engine and chassis as values.

KM supports multiple inheritance. When computing the values of an instance's slots, information from all super classes are merged. This differs from the object-oriented programming languages where one will take precedence over the others. The merging is in the KM nomenclature referred to as unification.

2.3.2 Constraints

KM offers two kinds of constraints: Value constraints and set constraints. The first type applies to the individual values of the slots. The second applies to slots with sets, sequences and bags as their value. Constraints are checked during inference. This means that if frame `a` is constrained by constraint `c`, `c` is not evaluated before some query accesses values from `a`.

Constraints play three roles in KM:

1) Debugging tool for knowledge engineer
2) Controlling unification (the merging mechanism described above)
3) In three special cases, KM will make assertions based on constraints. The three cases are listed in Table 2.6.

Constraint	**Assertion**
`(must-be-a` *`class`*`)`	If the value instance is not in *class*, it is "forced" to be one
`(exactly 1` *`class`*`)`	If there are multiple instances of *class* they are unified
`(at-most 1` *`class`*`)`	

Table 2.6 - Assertions by constraints in KM

Value constraints

KM has six value constraints that apply to individual values of slots. Table 2.7 lists and gives a brief description of the constraints. Two example constraints are given in Table 2.8.

`(must-be-a` *`class`* `[with slotsvals])`	The value must be a subclass of *class*
`(mustnt-be-a` *`class`* `[with slotsvals])`	The value must not be a subclass of *class*
`(possible-values` *`val1...valN`*`)`	List of possible values
`(excluded-values` *`val1...valN`*`)`	List of excluded values
`(<>` *`expr`*`)`	The value must not equal the the evaluation of *expr*
`(constraint` *`expr`*`)`	*expr* must evaluate to true

Table 2.7 - Value constraints in KM

```
(every Person has
     (friend ((must-be-a Person))))

(*Fred has
     (favorite-colors ((possible-values *Red *Blue *Green))))
```

Table 2.8 - Value constraint in KM

Example of value constraint. First we se a `must-be-a constraint`, saying that the friend of a person must be a person. The second example shows a possible-values constraint that states red, blue and green as possible favourite colours of Fred.

The first five constraints are in fact just short hands for the last one. Table 2.9 shows a rewriting of the second constraint in Table 2.8.

```
(every Person has
      (friend (constraint (TheValue &? (a Person)))))
```

Table 2.9 - Value constraint in KM

Example of the most general value constraint in KM. This constraint consist of an arbitrary expression. In this example we reproduce the `must-be-a` constraint from Table 2.8.

Set constraints

Set constraints in KM apply to the set of values of a slot. There are four such constraint types

`(at-least n` *`class`*`)`	At least n instances in the set must be in *class*
`(at-most n` *`class`*`)`	At most n instances in the set must be in *class*
`(exactly n` *`class`*`)`	Exactly n instances in the set must be in *class*
`(set-constraint` *`expr`*`)`	Arbitrary constraint. *expr* must evaluate to true

Table 2.10 - Set constraints in KM

If upper bound is 1 KM will unify together values if there is more than one. If the upper bound is higher than one, KM will instead make an error notification to inform the user instead of making hazardous unifications.

Only upper bound constraints are checked. Therefore KM treats `(at-least …)` as a dummy constraint. However, if lower bound is violated, KM can be set to create "missing" instances, but on default it will do nothing. Table 2.11 shows an example of a set constraint.

```
(every Airplane has
    (parts ((a Fuselage)
         (a Wing with (side (*Left)))
         (a Wing with (side (*Right)))
         (a Engine)
         (at-least 1 Engine)
         (exactly 2 Wing))))
```

Table 2.11 - Set constraint in KM

Constraint checking is time consuming as it can involve traversing large parts of the knowledge model. In KM a constraint can be tagged as sanity-checks. These constraints can be turned on and off depending on the phase of knowledge modelling.

Constraints through slot slots

In addition to what is called constraints in KM, there are three slots on the slots (the same as facets) that offer constraint functionality. First, `domain` defines the most general class or classes allowed for the first argument of the slot. That is the *from-frame* of the slot. Second, `range` defines the most general class(es) for the slots second argument – the value of the slot. Finally, `cardinality` constrains how many classes or instances can fill the two arguments of a slot. Allowed cardinality values are `1-to-1`, `1-to-N`, `N-to-1` and `N-to-N`.

2.3.3 Summary

Types of constraints

Cardinality constraint
As just mentioned cardinality constraints in KM are supported through a facet-like mechanism on the slots. This mechanism allows us to set the cardinality of either side of the slot to 1 or N.

Concrete value constraint
KM has the constraints `possible-values` and `excluded-values`, for defining allowed and disallowed values of a slot. The constraint `(<> expr)` is a special case of the excluded-values constraint.

Value range constraint
KM does not have an explicit value range constraint. Such constraints can however be defined through the general constraint `(constraint expression)` where `expression` is an arbitrary piece of KM code that must evaluate to true.

Value class constraint
The value class of a slot can be constrained using the `must-be-a` and `mustnt-be-a` constraints. Using these you can define a class that the slot value must be or must not be an instance or subclass of. Another way to define such constraints is through the `range` slot of the slot mentioned recently. Here you define the most general class that the slot value can be a member of.

General constraint
As mentioned several places already, arbitrary constraints can be defined using the `(constraint expression)` and `(set-constraint expression)` mechanism. These mechanisms take an arbitrary expression in the KM language that

must return a boolean value when evaluated.

Other constraint mechanisms
In 2.1.3 we mentioned that sometimes it makes sense to define constraints on a slot independent on what frame it is attached to. The constraints defined using the three constraint slots are of this category.
The set-constraints in KM do not fit into the framework defined in 2.1.3. Because TrollCreek does not have sets or list of values, we will not discuss such constraint further.

What can we learn from KM?

All the constraints that can be defined using the constraint mechanism in KM can be defined using the two constraints `(set-constraint expr)` and `(constraint expr)`. The other constraint keywords are only syntactic sugar for these two. This sugar may however be very useful because they save the knowledge engineer from a lot of typing. The readability of the constraints is also improved. If it later is decided to implement TrollCreek's constraints as a typed language, such syntactic sugar will be considered.

2.4 CYC

Existing expert systems may perform well in their narrow domains, but when asked a question at the edge of their knowledge, they can make big mistakes. A famous example in the AI world is Mycin diagnosing a male as pregnant. The idea of the Cyc project was to overcome this brittleness in AI- and knowledge systems by building a huge knowledge base of common sense [11]. Early in the project they estimated the size of this KB to five million frames, each containing several slots.

The project started out in 1984 and during the next ten years the Cyc project spent a person-century of effort in building the knowledge base [13]. In 1994 Cycorp, Inc was founded to continue the development of the Cyc knowledge base. The knowledge base is at the moment (May 2006) available in two versions. OpenCyc [20] is a free open source version of the Cyc knowledge base that in the current release contains 47000 concepts and 306000 facts. ResearchCyc [19] is designed for use by researchers and comes with a larger knowledge base. This version is proprietary but licences are granted to researchers free of charge.

2.4.1 The language

The representational language CycL (Cyc Language) started out as a frame system but has over the years evolved toward first order predicate logic [13]. Because the purpose of this review is to learn about constraints in frame systems, we will only focus on the early frame-days of CycL.

To represent this extremely large knowledge base, the developers decided to make their own knowledge representation language. The wish-list of the language was something like this:

- Clear and simple semantics
- Effective inferential capabilities
- Representing default knowledge
- Include all the expressiveness of first order predicate calculus (FOPC)
- A means of handling propositional attitudes (beliefs, goals, dreads)
- Facilities for operations such as reification, reflection, etc.

One of the intentions of the Cyc developers was that their system should be built upon and used by other researchers. This requires the system to have a clear and stable

semantics. To get the expressiveness of FOPC and at the same time have effective inference mechanisms may however force the developers to make special-purpose representations and inference routines. This disturbs the clear semantics. To solve this problem the Cyc KB was developed at two levels, the epistemological level and the heuristic level. Communication with users and external programs is supposed to be at the epistemological level. The language used at this level is the Cyc Constraint Language (Cyc CL).

The CycL was developed incrementally during the development of the Cyc knowledge base for natural knowledge understanding and common sense reasoning [11]. When the developers encountered something they wanted to express that was difficult to express in CycL, they augmented the language to handle it.

> *"it is a frame-based language embedded in a more expressive predicate calculus framework along with features for representing defaults, for reification (allowing one to talk about propositions in the KB) and for reflection (allowing one to talk about the act of working on some problem.)" [11]*

CycL is frame-based with frame-slot relationships defined as a triple <attribute/slot> <frame><value>. The value of a slot in CycL is always a list representing the possibly empty set of values for the slot. Triples describing the same concept are stored together to form a frame. Cyc has four basic kinds of frames: "normal frames", SlotUnits, SeeUnits and SlotEntryDetails [12].

Normal frames are the most common type. They represent most real world concepts such as the concept Person, the name Martin and the process of writing a master thesis.

Like in many other frame systems the slots are themselves frames. In Cyc slot types are represented by SlotUnits. These slot-frames can have slots constraining the from-frame of the slot, the value of the slot, describe what slots have related meaning, etc.

SeeUnits contain metalevel information about a specific slot for a specific frame. It can for example be used to express incomplete knowledge. If we have a frame describing the country Norway with the slot Residents we maybe lack some information to give a complete list of residents. Instead we can say that the number of residents are about 4,5 million, that the number is quite stable and that the residents must be members of the class Person.

SeeUnits describe a whole slot on a frame. SlotEntryDetails are very similar but differs in that it describes a single entry for a slot on a frame. For example the author of this text is an inhabitant in Norway. This fact became true in 1981 when he was born and were temporary false in the period 1984 to 1988 when his family lived abroad. Such information can be represented by SlotEntryDetails.

Table 2.12 shows how one of the authors of [12] describes himself in the Cyc language.

```
#%Guha
  #%instanceOf                (#%MechanicalEngineer #%LispHacker
                               #%HumanCyclist #%GraduateStudent)
  #%computersFamiliaryWith    (#%SymbolicsMachine)
  #%age                       (23)
  #%languageSpoken            (#%EnlishLanguage #%TamilLanguage
                               #%HindiLanguage #%GermanLanguage)
  #%programsIn                (#&Lisp)
  #%screenForUnitEditor       (:monochrome)
```

Table 2.12 - Frame in Cyc

2.4.2 Constraints

Cyc Constraint Language

CycL Constraint Language sits on top of the frame system and is essentially predicate calculus. The syntax of Cyc CL is similar to prefix predicate calculus and every slot in the frame-based representation may be used as predicates. Actually, binary predicates are represented as slots. Predicates with more than two arguments can be defined using Lisp expressions. In addition you can define functions that return other values than true/false. This makes the constraint language very expressive.

The constraint language enables:

- Disjunctions
- Quantifications (some, every)
- Negations
- Relationships (every person is *younger* than his mother)

In Cyc CL both relationships and quantifications are represented as predicates. This is illustrated in Table 2.13.

```
(#%ForAll X (#%InstanceOf X #%HumanBeing) (#%Mortal X)

(#%ThereExists X Y (#%GreaterThan X Y))
```

Table 2.13 - Constraints in Cyc

The first constraint states that all human beings are mortal. The second statement is not a traditional constraint, but rather an expression saying that some concept can be greater that another concept. Notice how quantifications (ForAll, ThereExists) and relations (GreaterThan) are represented as predicates.

The Cyc CL can naturally be used to state constraints on the knowledge in the knowledge base. Using the language we can state constraints on slot values and define which units can legally possess a certain type of slots. In Cyc, constraints are checked when knowledge are put into the system. This prevents the user from entering incorrect knowledge.

As shown in Table 2.13 the constraint language can also be used to express other properties of the model. It can for instance be used to state the definition of a collection or stating the premises and conclusions of different kinds for rules. The frame-based CycL and the logic based Cyc CL are used together to get the expressive benefit of logics and the frame's suppleness and effectiveness of inference.

Constraints defined by slots on slots

Slots in Cyc are represented as frames that themselves can have slots. This functionality is similar to facets in LispCreek. One slot used to describe other slots is *entryFormat* which can take the values *SingleEntry*, *SetTheFormat* and *SubAbs*. *Single* means that the slot can have only one value, while the two other are used for cardinality greater than one. *SubAbs* have the additional function of stating that the values must all be specializations of the same concept.

There are also a great number of other slots that can be used to describe slots. In the summary subsection the following are mentioned: *slotCardinality, noOfEntitiesLessThan, entryIsLessThan, entryIsGreaterThan, entryIsA* and *makesSenseFor*.

2.4.3 Summary

Types of constraints

Cardinality constraint
CycL offers several ways to define cardinality constraints. The slot *entryFormat* described above defines whether a slot can have one or many values. This is a simple version of cardinality constraint. Two other slots can also be used. *slotCardinality* defines the exact number of entries that a slot shall have, and *noOfEntitiesLessThan* defines an upper bound of the cardinality. More complex cardinality constraints defining different numbers for the cardinality than *one* and *many* can be expressed by Cyc CL.

Concrete value constraint
Concrete value constraint can be defined using Cyc CL.

Value range constraint
The two slots *entryIsLessThan* and *entryIsGreaterThan* are used to define the range of allowed values for a slot.

Value class constraint
The above mentioned *entryFormat* slot can act partly as a value class constraint in addition to cardinality. If *entryFormat* has the value *SubAbs* this means all the values of the slot must be of the same class. This is however not the whole functionality we expect from a value class constraint. The slot *entryIsA* fulfils the value class constraint as defined in 2.1.3. This slot defines a super slot that all values must inherit from.

General constraint
As mentioned above, CycL Constraint Language is heavily based on predicate calculus. Together with the ability to define new predicates and functions using Lisp, this makes it possible to express arbitrary constraints.

What can we learn from Cyc?

The CycL and its constraint language are built for representing common sense about nearly all possible concepts in the world. This is quite different from TrollCreek that is mainly used for making expert systems in closed domains. The constraint language from Cyc is thus maybe too comprehensive to be copied to TrollCreek. The possibility to express arbitrary constraints on the content of the knowledge base is however a useful function that will be considered in the further work. Cyc's use of the constraint language to express incomplete knowledge is also worth a study.

2.5 KL-ONE

2.5.1 The language

KL-ONE is a well known knowledge representation system in the tradition of semantic networks and frames. The system is an attempt to overcome semantic indistinctness in semantic network representations and builds upon the idea of *Structured inheritance networks* [6].

Frames in KL-ONE are called *concepts*. These form hierarchies using subsume-relations; in the KL-ONE terminology a super class is said to *subsume* its subclasses. Multiple inheritance is allowed. Actually a concept is said to be well-formed only if it inherits from more than one other concept. All concepts, except the top concept *Thing,* must have at least one super class. Figure 2.2 shows a sample ontology in KL-ONE.

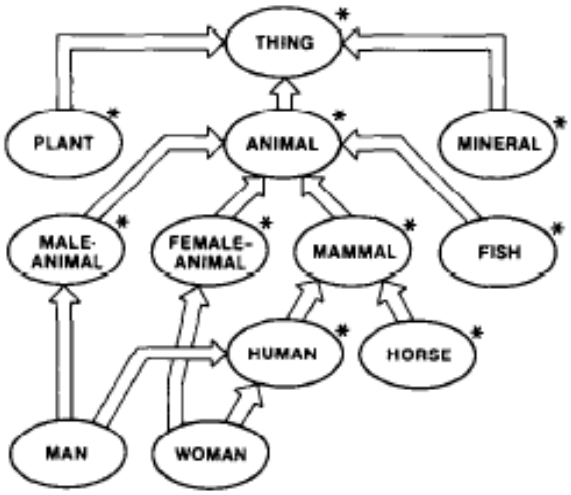


Figure 2.2 - A simple KL-ONE network of generic concepts

Concepts are represented as ellipses and the double arrows are subsume relations. This figure and all the following KL-ONE figures are taken from [6].

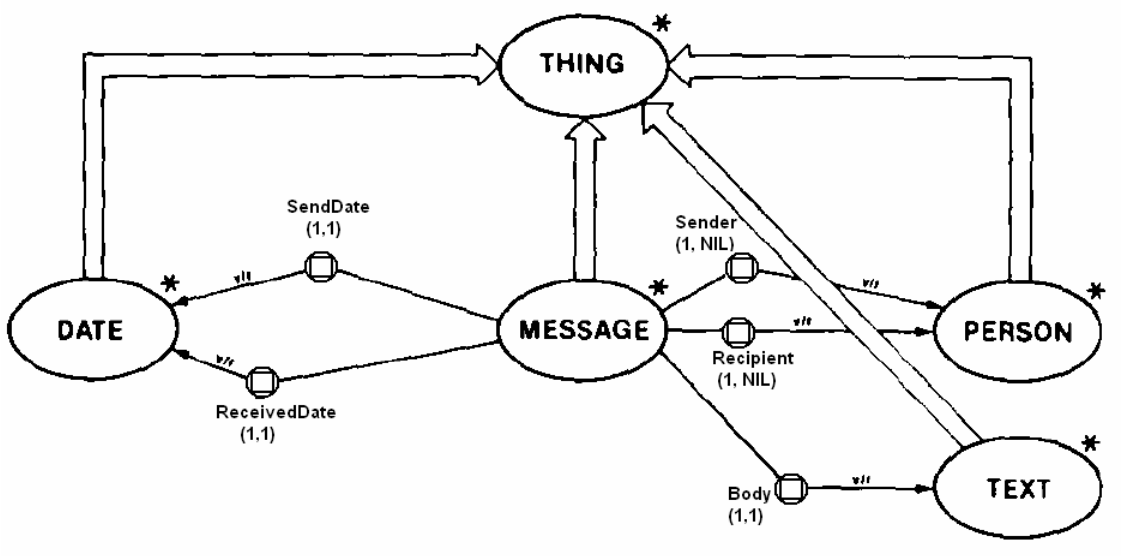


Figure 2.3 - KL-ONE representation of a message

> "A *Message* is something having exactly one *SendDate* and one *ReceivedDate* that are of type *Date*, one or more *Sender* and one or more *Recipient* both being *Persons* and exactly one *Body* of type *Text."*
>
> The encircled squares represent RoleSets. The numbers below the name of the RoleSets are cardinality constraints.

In KL-ONE descriptions is separated into two basic classes of concepts: *primitive* and *defined*. *Primitives* are domain concepts that are not fully defined. This means that given all the properties of a concept, this is not sufficient to classify it. They may also be viewed as incomplete definitions. Using the same view, *defined* concepts are complete definitions. Given the properties of a concept, these are necessary and sufficient conditions to classify the concept.

The slot-concept is called *roles* and the values of the roles are *role-fillers*. There are several different types of roles to be used in different situations. The most common and important role type is the *generic RoleSet* that captions the fact that the role may be filled with more than one filler.

2.5.2 Constraints

RoleSet restrictions

The constraints in KL-ONE are integrated parts of the language, not a plug-in-like constraint-engine. RoleSet restrictions are restrictions on the fillers of the RoleSets. There are two kinds of restrictions, number restrictions and value restrictions.

Number restrictions restrict the number of fillers allowed for a RoleSet. This restriction is used when you want to locally restrict the cardinality of a role inherited from another concept. Figure 2.4 shows an example of a *private-message* that inherits from the *message* in *Figure 2.3*. *Message* can have one or more recipients. *Private-message* uses a role set restriction to override this cardinality.

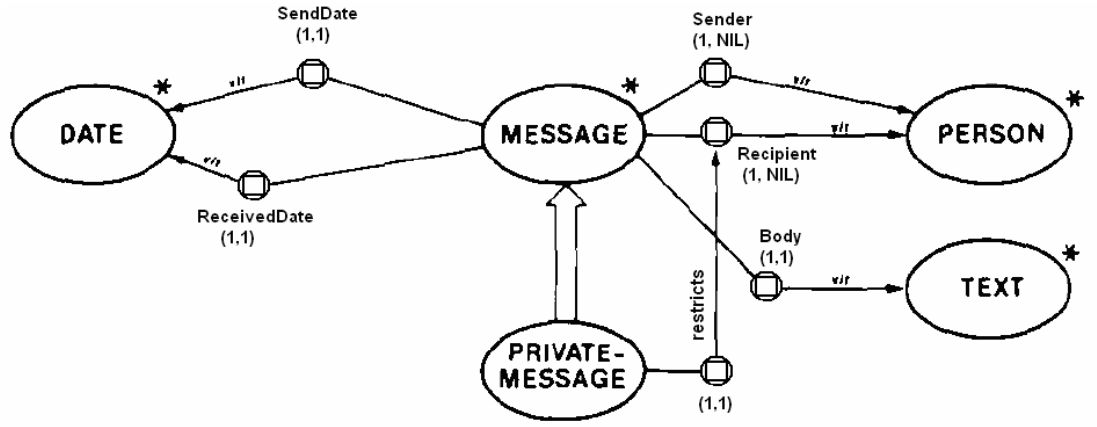


Figure 2.4 - Cardinality constraint in KL-ONE

"A *private-message* is a *message* with exactly one recipient."

Restrictions are denoted similar to RoleSets. The link pointing from *private-message* to the circle representing the RoleSet *Recipient* has the cardinality (1,1) saying that the allowed number is exactly one.

ValueRestrictions restricts which fillers can fill a role by defining the most general class allowed. In Figure 2.5 we see an example of a specialization of *message* that have restrictions on who can be the *sender.*

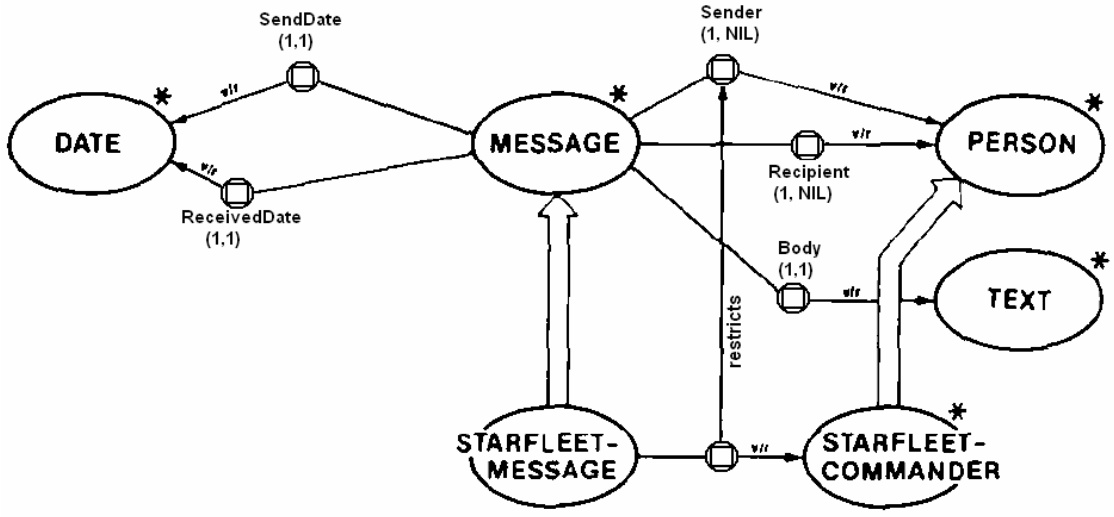


Figure 2.5 - Value class constraint in KL-ONE

"A *starfleet-message* is a *message* where only *starfleet-commanders* are among the senders."

The restriction link pointing from *starfleet-message* to the RoleSet *Sender* has an arrow pointing at the *starfleet-commander* concept saying that the filler of the role *Sender* on the concept *starfleet-message* must be a of this type.

Structural Descriptions

The RoleSet restrictions can constraint the number and class of fillers to fill a RoleSet. Sometimes it is however useful to define constraints that includes relationships among more than one of a Concepts roles. KL-ONE offers this through Structural Descriptions (SD).

There are two kinds of SD, the simplest is Role Value Maps (RVM). RVM is designed to let the knowledge engineer express equality between two sets of role fillers – for example that the grandmother of a person is the same person as the mother of one of the parents of the person. An example is shown in Figure 2.6.

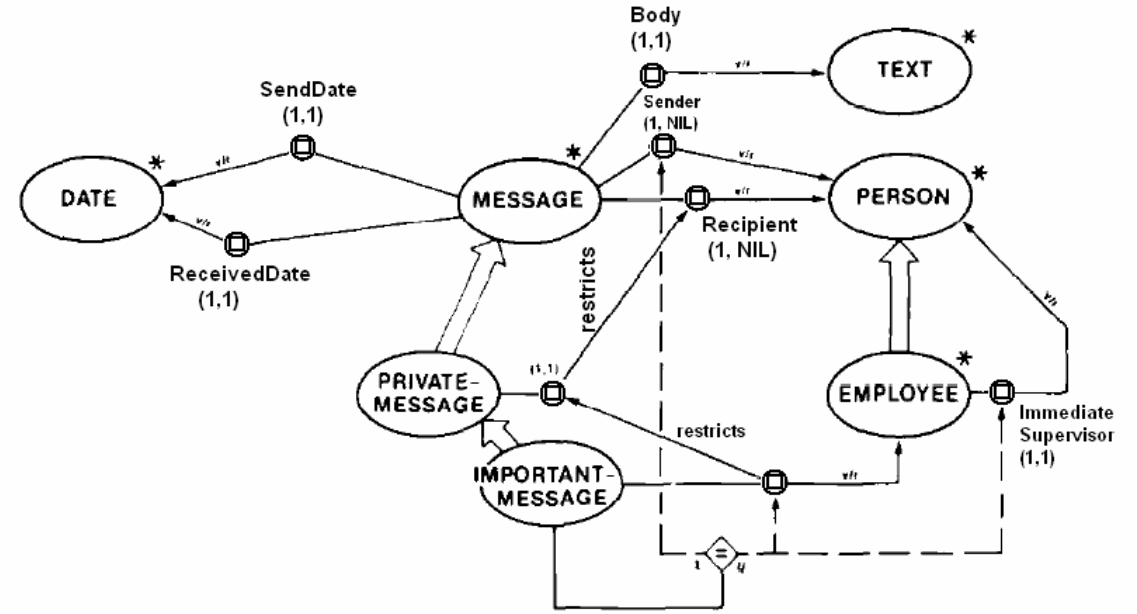


Figure 2.6 - Role value map in KL-ONE

> "An important-message is a private-message with an employee as Recipient and whose sender is the same as the ImmediateSupervisor of its Recipient."
>
> The constraint is represented with a ternary relation pointing at the two roles that must have equal fillers. At the bottom of the figure we see a link going from *important-message* to a rhomb containing an equality sign. From this rhomb two dotted arrows are pointing to the two RoleSets that must have the same fillers.

The more general form of SD has not got a name. It describes how Roles of a Concept relate to each other in terms of other concepts. To illustrate the mechanism we extend the message example with two new message types: Reply-requested-message and Urgent-message. The first type is a message with a reply deadline. The second is a specialisation of the former with short deadline.

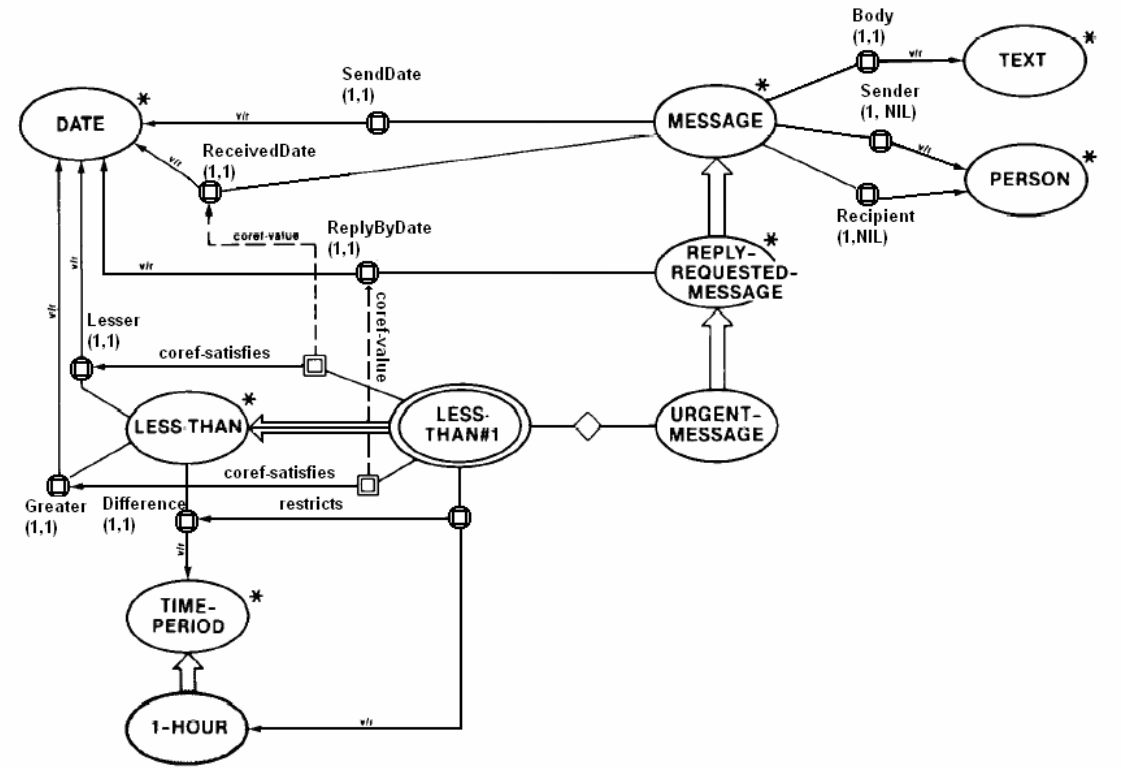


Figure 2.7 - Structural description in KL-ONE

"A reply-requested-message is a message with a ReplyByDate which is a date"

"An *urgent-message* is a *reply-requested-message* whose *ReceivedDate* and *ReplyByDate* satisfy a *less-than* whose *Lesser* is the *ReceivedDate,* whose *Greater* is the *ReplyByDate,* and whose *Greater* is the *ReplyByDate,* and whose *Difference* is *1-hour.*"

2.5.3 Summary

Types of constraints

Cardinality constraint
Cardinality constraints are supported through the cardinality property of the RoleSets. With this property we can set both upper and lower bound of the cardinality. Number restrictions can be used when a subclass inheriting a slot shall have another cardinality than the original slot, like in Figure 2.4.

Concrete value constraint
[6] does not explicitly mention any concrete value constraint. Some of the functionality of this constraint can however be achieved by the value restriction.

Value range constraint
The range of values can be constrained with *structural descriptions* like in Figure 2.7. Using this mechanism the value can be constrained to be less than some value and greater than some other value.

Value class constraint
Figure 2.5 illustrates a value restriction that restrict who can be the sender of a special kind of message. This corresponds to what we call value class constraint.

General constraint
Structural descriptions provide a means of defining some dependencies between values that are more complex than the four simplest constraint types. In the examples

above we saw how the structural descriptions could define how the filler of two different roles must be related to each other. This mechanism does however not offer the expressive power we want from the general constraint mechanism.

What can we learn from KL-ONE?

KL-ONE does not introduce any new constraint types or mechanisms to our review. The interesting part is the graphical representation used in [6]. As we will see, in TrollCreek the knowledge engineering can be done through the graphical user interface, drawing nodes and relations. Graphical representation of constraints should therefore be considered in the process of designing a constraint mechanism for TrollCreek.

2.6 Protégé

The first version of Protégé was made in 1988 and was a child of Mycin and ONCOCIN [8]. It was originally designed to ease the process of knowledge acquisition by allowing the domain expert to write the domain knowledge more or less without the help of a knowledge engineer. Protégé 2000 is the fourth version.

Protégé 2000 is a frame-based knowledge representation language, modelling tool and knowledge acquisition tool. It is developed under the Mozilla Public License [21] and is therefore free to download, redistribute and reengineer.

2.6.1 The language

The ontology in Protégé's knowledge model is built up by classes, slots, facets and axioms [14]. The classes are concepts in the domain of discourse. The properties of the classes are described by slots, and the properties of the slots are described by facets. Axioms are used to represent constraints additional to the constraints that can be described using facets. The axioms are written in the Protégé Axiom Language (PAL).

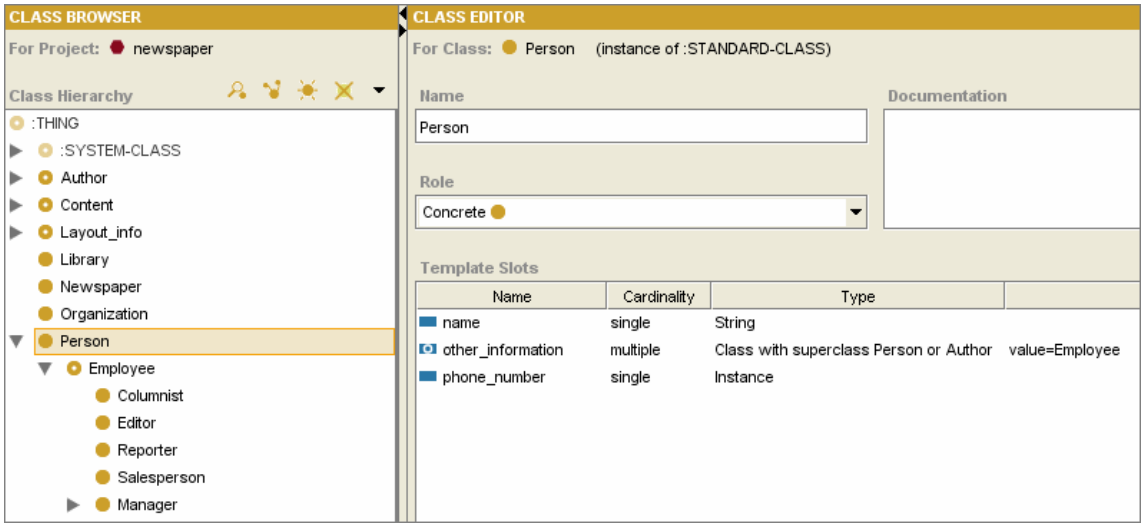


Figure 2.8 Protégé class hierarchy and class editor

In the left column we see the class tree. To the right we see the slots of the frame Person and some of the slots facets.

A knowledge base in Protégé consists of the ontology and a set of instances of the classes in the ontology. The core of the knowledge model is a hierarchy of classes. A class must have at least one super class and zero or many subclasses. Figure 2.8

shows how the class hierarchy is presented to the user. The meta class *:Thing* is the top of the hierarchy, so every class is a subclass of it. Classes can have instances and both individuals and classes can be instances.

The slots describe the properties of classes. Slots are classes too and are defined independently of the classes they describe. One single slot can be attached to many different classes. When a slot is attached to a frame it can have a value that is either a class or an individual.

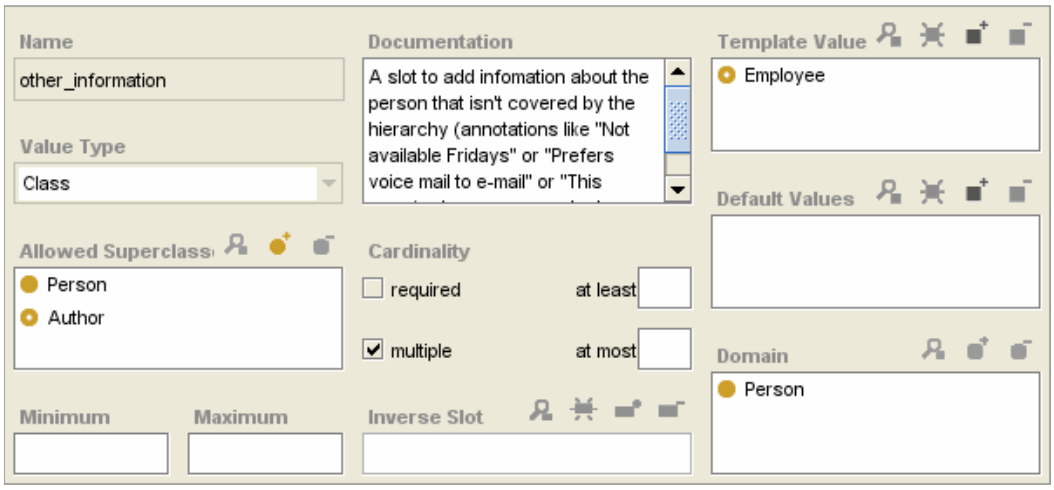


Figure 2.9 - Protégé's facet-dialog

The facet dialog is used to show and edit facets on slots.

Facets describe the properties of slots. Default values, allowed values, max/min values, cardinality and data type. This is in fact a constraint mechanism, but it offers only local constraints - constraints local to that particular slot. You cannot define dependencies between different slots of the same frame (eg. date of death can not be more than 120 years after date of birth). Figure 2.9 shows the facet dialog in Protégé.

2.6.2 Constraints

As mentioned above, facets allow only local constraints. This is the background of the Protégé Axiom Language (PAL). PAL allows arbitrary constraints that include other classes, dependencies among different slots and so on [9].

PAL is a complement to Protégé 2000 that enables logical constraints and query processing. It consists of a language (PAL) and a constraint-checking engine. PAL constraints are stored as instances of the *:PAL-CONSTRAINT* meta class – a normal frame so the knowledge base doesn't have to know that it is a constraint. The language is a variant of the Knowledge Interchange format (KIF), made to be easily parsed by computers, but its syntax is not easy read by human beings [9]. To assist the knowledge engineer in writing the constraints, Protégé offers a structured editor with guiding mechanisms. The language is independent of the constraint-checking engine, except for the syntax. Table 2.14 shows an example PAL constraint.

```
(defrange ?editor :FRAME Editor)
(defrange ?employee :FRAME Employee responsible_for)

(forall ?editor (forall ?employee
                  (=> (and
                         (responsible_for ?editor ?employee)
                         (own-slot-not-null salary ?editor)
                         (own-slot-not-null salary ?employee))
                      (> (salary ?editor) (salary ?employee)))))
```

Table 2.14 - PAL constraint

> "The salary of an editor should be greater than the salary of any employee which the editor is responsible for."
>
> The expression has two parts. First the range two variables `?editor` and `?employe` is defined. Then the constraint body is defined.

The Constraint checking engine is a plug-in and can be easily changed. The default engine is based on Model checking – to detect incomplete or inconsistent entries. When an inconsistency is detected in the knowledge base, the user is informed, but the engine does not change anything in the KB. The engine is invoked by the user.

2.6.3 Summary

Types of constraints

Cardinality constraint
Cardinality is supported through the facet mechanism. You can define two values, *at least* and *at most*. Alternatively you can say that the slot is *required* or *multiple*.

Concrete value constraint
It is possible to define a list of allowed values for a slot. To define disallowed values you have to write a PAL constraint.

Value range constraint
Protégé slots have two facets named *minimum* and *maximum* that provides the ability to define the range of numerical values.

Value class constraint
Every slot has a mandatory facet called *value type* that defines the data type of the slot [22]. The available data types are listed in Table 2.15.

Type	Description	Example
Any	Any of the types below	
Boolean	Logical value	True, False
Class	Class in the kb	Person, Car
Float	Decimal number	1.0, 3.4e10
Instance	Instance of a class in the kb	Instance_001
Integer	Integer number	1, 2, 12354987
String	String of alphanumeric characters including spaces	"Hello world, calling 1 2 3"
Symbol	List of values, which may not include spaces	Re, blue and green

Table 2.15 - Data types in Protégé

For the types *class* and *instance* you can define which class or classes the value has to be a subclass/instance of. This corresponds to the value class constraint.

General constraint
Constraints not expressible through the facet mechanism can be expressed by the

Protégé axiom language. PAL constraints have almost the expressibility of first order predicate calculus. This provides the ability to state arbitrary and complex constraints.

What can we learn from Protégé?

In Protégé there are mainly two ways of defining constraints: through facets and through PAL. The facet constraints are those constraints easily defined through a graphical user interface. The more complex constraints are offered through the constraint language. This two fold way of defining constraints allows novice users to define the simplest constraint types, lowering the threshold to use the system. The idea of using facets is also interesting in the TrollCreek setting.

3 TrollCreek

Before we start discussing how to design and implement constraints in TrollCreek it can be helpful to give a brief overview of the system, both at a conceptual and an implementational level. The overview only describes the aspects necessary to understand the remaining chapter of this thesis. For a comprehensive description of the system we refer to [17] and [5].

3.1 The conceptual model

As mentioned, Creek is an abbreviation for “Case-based Reasoning through Extensive Explicit Knowledge” [1]. The scope of this project is the “extensive explicit knowledge” part of the system that is realised through a frame-based knowledge representation. Consequently the case-based issues will not be discussed here.

The frame-based knowledge model in TrollCreek is represented by a semantic net structure. A node in the semantic net does together with its relations form a frame. In Figure 3.1 we see how these two representations works together.

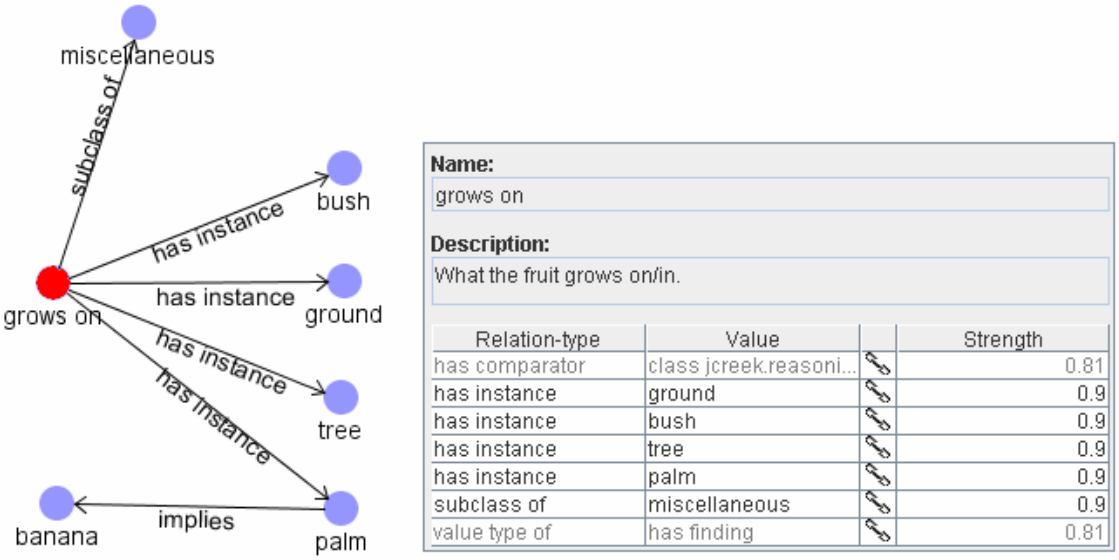
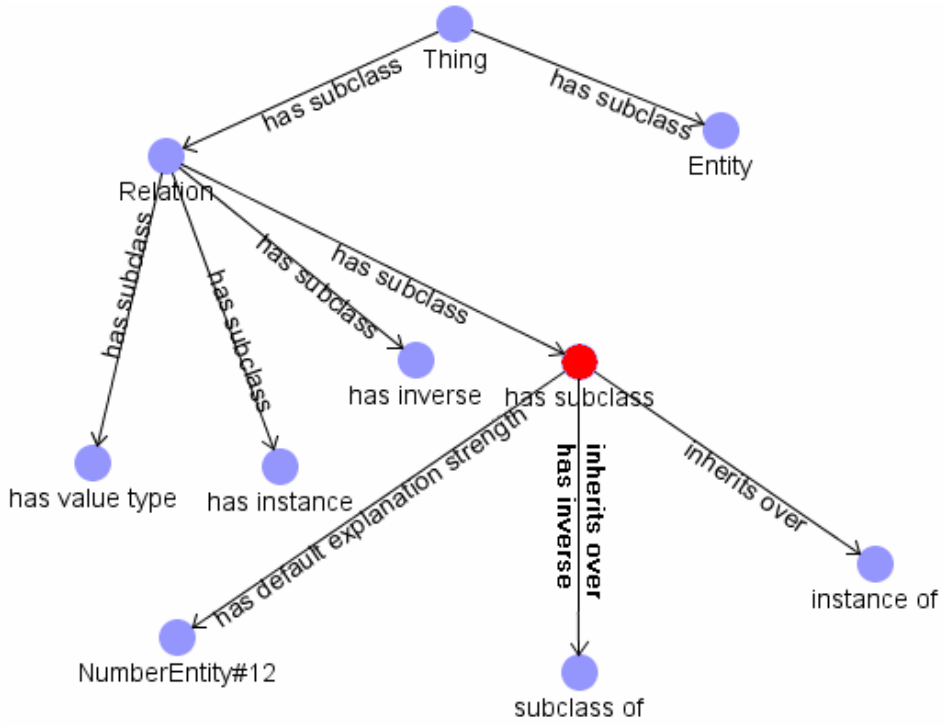


Figure 3.1 - Creek semantic net and frame

The frame view shows the *grows on* frame that we also find as a node in the semantic net. The slots of the frame are the relations of the semantic node.
The content of the figure is taken from a test knowledge model contained in TrollCreek.

As with the Lisp version of Creek the relations are also frames. More precisely, the relation types of TrollCreek are represented as frames in the knowledge model. This added semantic is useful as a clarification of what the different relations means, making it easier to make good models of complex domains. The taxonomy of relations can also be used in the inference process. Figure 3.2 shows an example top ontology.



Name:
has subclass

Description:

All Relations Local Only

Relation-type	Value		Strength
has default explanat...	0.9 (NumberEntity#1...		1.0
has inverse	subclass of		1.0
inherits over	subclass of		1.0
inherits over	instance of		1.0
subclass of	Relation		1.0

Figure 3.2 - Relation frame

The figure shows a subset of the semantic net representation of the top ontology in TrollCreek. The frame at the bottom shows the frame of the frame of the *has subclass* relation. It inherits over *instance of* and *subclass of.* Its inverse relation is *subclass of. NumberEntity#12* represent the default explanation strength of *has subclass.* Every relation in Creek has an explanation strength used to compute the total strength of a path between two frames.
The taxonomy is taken from an earlier version of TrollCreek. The current version is more comprehensive.

Two of the slots in Figure 3.1 are shown with grey font in the frame view. These slots are inherited. Inheritance is central in semantic nets and frame-based knowledge representations. The most common mechanism is that properties are inherited through relations corresponding to the Creek relations *has subclass* and *has instance.* In TrollCreek inheritance is augmented through a mechanism called plausible inheritance that allows properties to be transferred over other relation types as well. What relations that transfer other relations are easily defined by making a *transfers* relation between the entities representing the two relation types.

Figure 3.3 gives an example where the relation *requires* is transferred by *has part*.

From the fact that *car* has the part *engine*, which has the part *piston* that requires *oil*, we can infer that *engine* requires *oil*. Further we can infer that *car* requires *oil.* The plausible inheritance mechanism in TrollCreek is thoroughly described in [17].

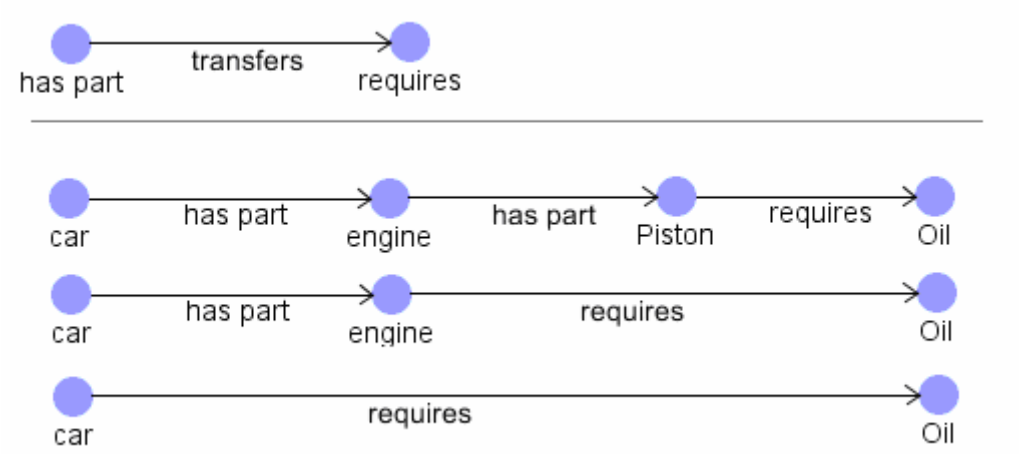


Figure 3.3 - Example of plausible inheritance.

The figure is adapted from [17].

Relations in TrollCreek are unidirectional. However, every relation type has an inverse relation. For instance *has part* has the inverse *part of*. When a relation is defined between two entities, the inverse relation is automatically entered by the system.

3.2 The implementation of TrollCreek

At the conceptual level the model of TrollCreek is built of entities and relations forming a semantic network. An entity and its relations can be seen as a frame where the entity is the frame and the relations with values are the slots.

3.2.1 KnowledgeModel

In the implementation the knowledge model is represented by an object of the class `KnowledgeModel` that contains information about the model and collections of entities, relations and other objects in the model. It also has methods for managing the model, such as creating entities and relations, mapping entity and relation names to the objects representing them, merging models etc.

Entities and relations are represented by the classes `Entity` and `Relation` that act as a programmatic interface to the corresponding concepts. This means that these classes have methods to access and manipulate the `Entity` and `Relation` objects structures. They do however not directly contain any of the data associated with entities and relations. To allow different physical storage methods for the model (SQL database, XML, binary file, etc), all data that are supposed to be persistent are put into objects of the classes `EntityData` and `RelationData`. These are defined as interfaces[2]. The same is true for `KnowledgeModel` mentioned above. Figure 3.4 illustrates the relationship between `KnowledgeModel`, `Entity` and `Relation` from the perspective of `KnowledgeModel`.

[2] Java interfaces are classes containing no variables and just the declarations of the methods. It is not possible to instantiate these classes, but they can be implemented by other classes. When a class implements an interface it commits to a contract that it must implement the methods in the interface.

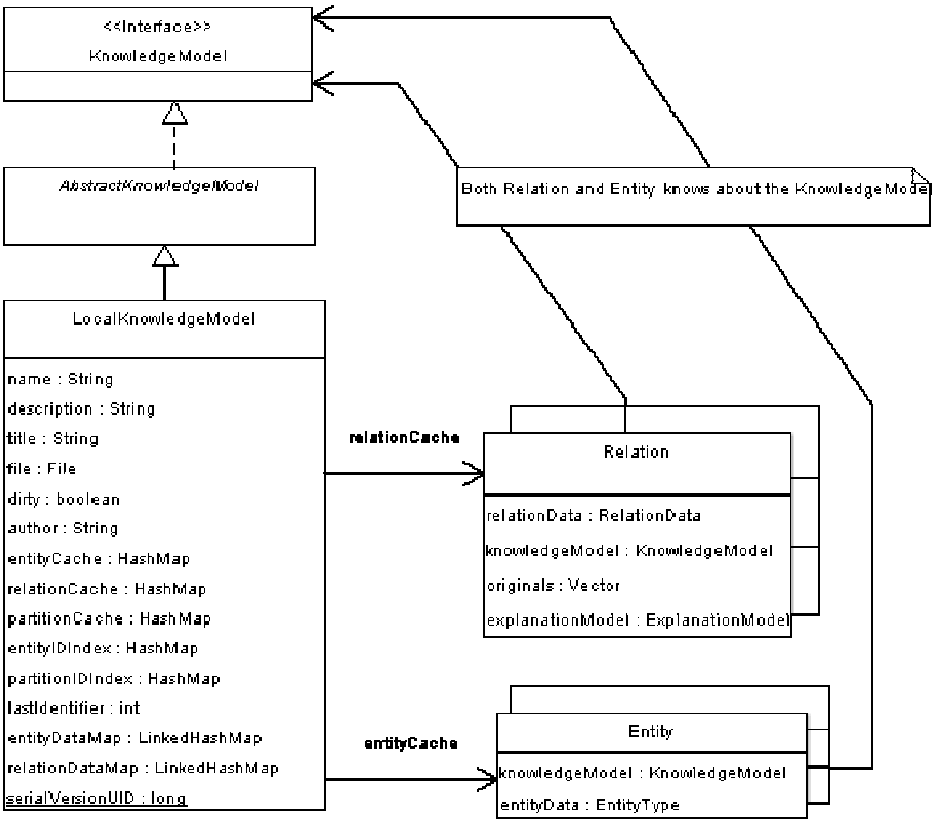


Figure 3.4 - Top level classes of the knowledge model

In the current TrollCreek release these interfaces are implemented in `LocalEntityData`, `LocalRelationData` and `LocalKnowledgeModel`. As the names indicate, they are designed for storing the knowledge model in a local file, but this is hidden from the `Entity` and `Relation` objects. They only know that they have a variable pointing to some object implementing the general interfaces. In the remaining of this section only the "local" version is considered. When written that a variable or piece of data is contained in an interface this means that the interface declares a method for accessing the data.

3.2.2 Entity

As described the `Entity` object has its data stored in an `EntityData` object. The data stored are among other the name and description of the entity, an identification number and a list of relations. There is also an attribute called `entityObject` that can store any java object, used to store type specific data. Numbers are in TrollCreek represented through a specialisation of the Entity-object named `NumberEntity`. This object uses the `entityObject` to store the number and has methods for setting and accessing the number. Number enities and other special entity types subclasses the abstract class `EntityType`. The class structure representing an entity is illustrated in Figure 3.5.

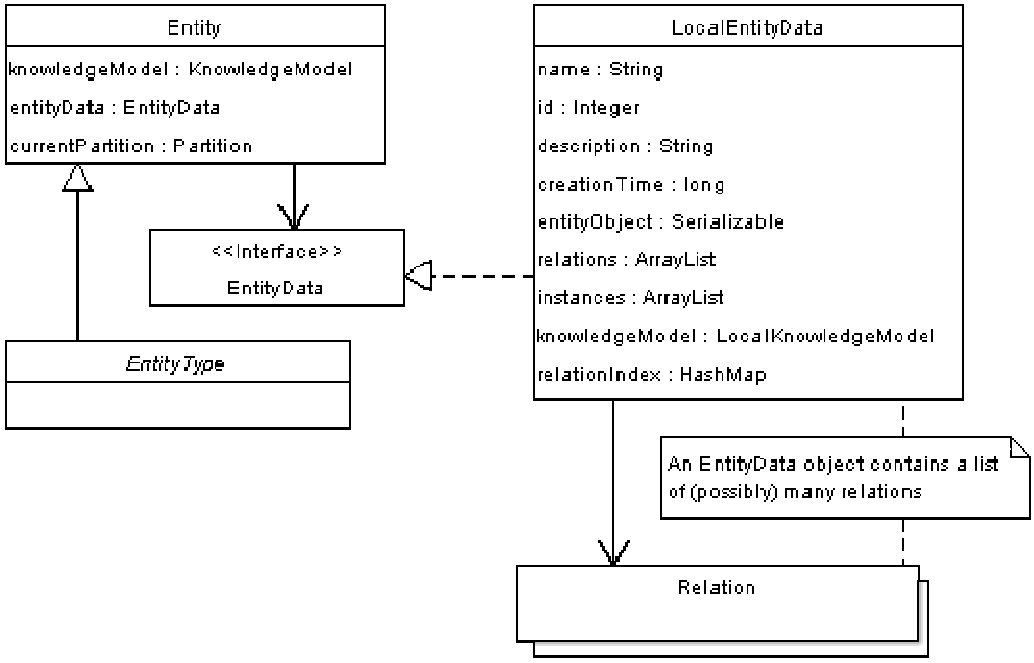


Figure 3.5 - The main classes in the representation of Entity

3.2.3 Relation

The object structure representing a relation is illustrated in Figure 3.6. The internal data of `Relation` are stored in the `RelationData` object. A relation connects two entities, represented by the `from` and `value` variables in `LocalRelationData`. From a `RelationData` object, and so also a `Relation` object, we can then access both entities involved in the relation. In the discussion on how to implement constraints, this may be an interesting fact.

A relation also has a type, for example *subclass of*. This is represented by a variable `type` of the class `RelationType` that is a specialisation of `Entity`. Recall from Figure 3.2 that *Relation* together with *Entity* are direct subclasses of the most general concept *Thing*. This relation frame is actually not a relation, but a relation type, and the frames inheriting from it are different kinds of relation types. For details see the lower right corner of Figure 3.6. As mentioned, relations in Creek always have an inverse relation. The `inverse` variable in `LocalRelationData` points to the `Relation` object of this inverse.

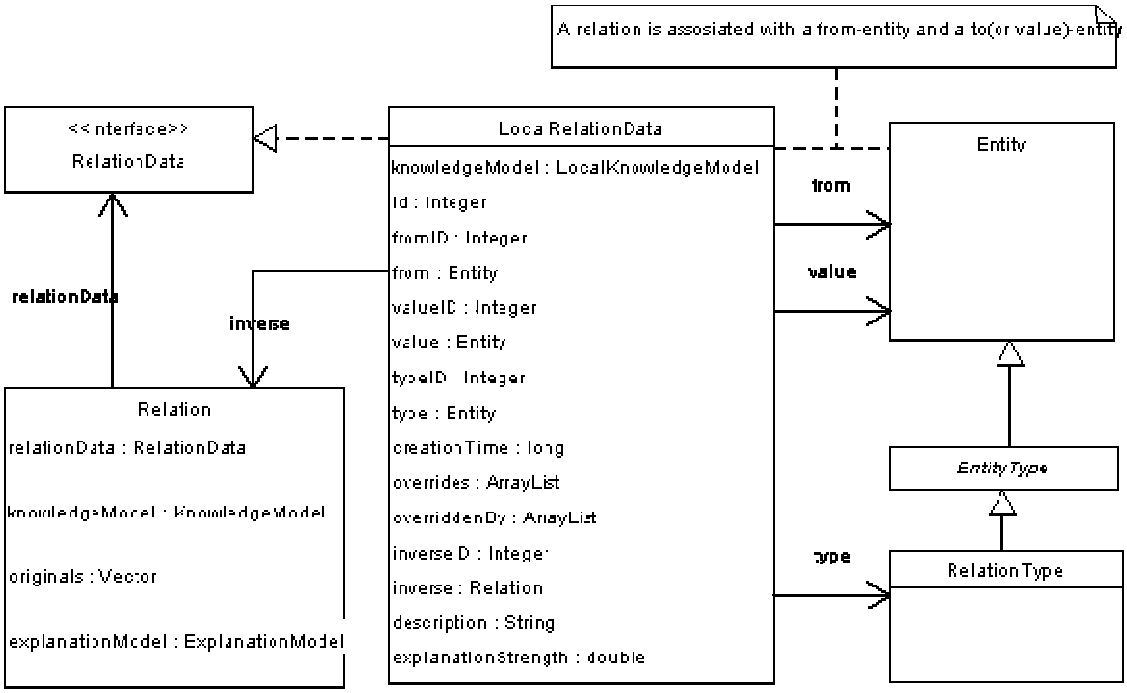


Figure 3.6 - The main classes in the representations of Relation

4 Functional requirements

In section 2.2 through 2.6 we discussed different frame-based knowledge representation systems with emphasis on their constraint mechanisms. We also gave an enumeration of constraint types found in these systems. Here we describe what we want from a constraint mechanism in TrollCreek.

4.1 Constraint types

4.1.1 Cardinality constraint

In LispCreek a slot can have a single value or a list of values [3]. If you want to have multiple values for a single slot in TrollCreek you have to add a new instance of the slot for every value. This means that one frame can have many slots of the same type. Representing a person Kari with three children will then be as in Figure 4.1.

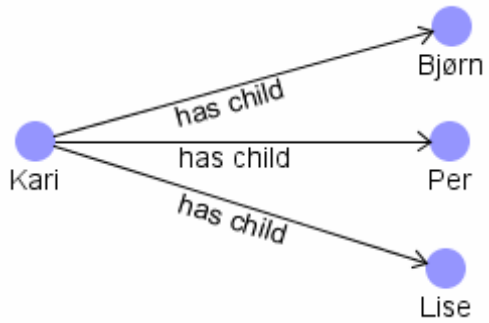


Figure 4.1 - Cardinality in TrollCreek

This is the natural way to do it when using a semantic net to represent the frames. Accordingly the cardinality constraint in TrollCreek will not be a constraint on how many values a slot can have, but rather how many slots of a given type an entity can have. In TrollCreek it is not possible to define that a frame shall have a given slot without giving it a value. The presence of a slot therefore implies that there is a value. Constraining the number of slots is thus semantically equivalent to constraining the number of values.

The cardinality constraint is a very common constraint type in frame-based knowledge modelling systems. Besides, it is also essential in database modelling. Cardinality constraints should therefore be included in the design of a constraint system in TrollCreek. The constraint should be as flexible as possible, allowing to define an upper bound (at most X), a lower bound (at least Y) and a range of allowed cardinalities (from Y to X).

4.1.2 Concrete value constraints

In addition to defining how many values there can be of a kind, it is useful to constrain what values a slot can have. One way to do this is through concrete value constraints. This should be done by defining the closed set of allowed values for a slot. In some cases it can also be useful to define some values as not allowed, indirectly allowing all others.

4.1.3 Value range constraint

Numerical values are common in most domains. For instance to avoid people getting unreasonable old in our knowledge bases it can be helpful to add range constraints to numerical values. Age may for example be constrained to be in the range 0 to 130. The range should be defined as an interval or by only stating a maximum or minimum value. This constraint type should be included in TrollCreek's constraint mechanism. In the above definition of value range constraints only one interval of allowed values can be stated. In some cases it may be interesting to define more than one interval, or maybe define an interval as disallowed. This should also be implemented.

4.1.4 Value class constraints

In the concrete value constraint we define a list of allowed values for a slot. Sometimes the set of allowed values has a size that makes it bothersome to enumerate. If then the allowed values are all subclasses of some common super class, we can use a value class constraint. Constraining the class of allowed values for a slot is maybe the most common constraint type in frame-based knowledge representations.

4.1.5 General constraints

As mentioned in section 2.1 there are many interesting constraints that we cannot express using the simple constraint mechanisms above. This is constraints dealing with relationships between more than two frames. The most obvious way to express such constraints is through a written language, either the native language of the representation system (as Lisp in LispCreek) or a special constraint language (like PAL in Protégé).

It should be possible to write expressions describing complex relationships between frames in the knowledge base. Every frame and slot should be accessible from the expressions. These constraints can either be attached to the frames they describe or they can have a global belonging.

4.2 Inheritance

In semantic nets and frame-based knowledge representations inheritance is central. Properties of a super class are inherited to its sub classes. The most common mechanism is that properties are inherited through sub-super class relations and instance relations. In TrollCreek inheritance is augmented through a mechanism called plausible inheritance that is described in section 3.1 .

Inheritance is also interesting when talking about constraints. Constraints defined in the top of a hierarchy are likely to be inherited to sub classes. In Figure 4.2 we see an example of a small knowledge model where the Norwegian folktale character Espen Askeladd is defined as instance of Boy that is a subclass of Child, subclass of Person. There are two value range constraints - One saying that the age of a person must be between 0 and 130, the other saying that the age of a child must be between 0 and 18. Without inheritance these constraints must have been defined on all instances of man, woman, boy and girl. With inheritance, the constraint defined on Person is inherited to all its subclasses and the Child-constraint is inherited to Girl, Boy and Espen Askeladd.

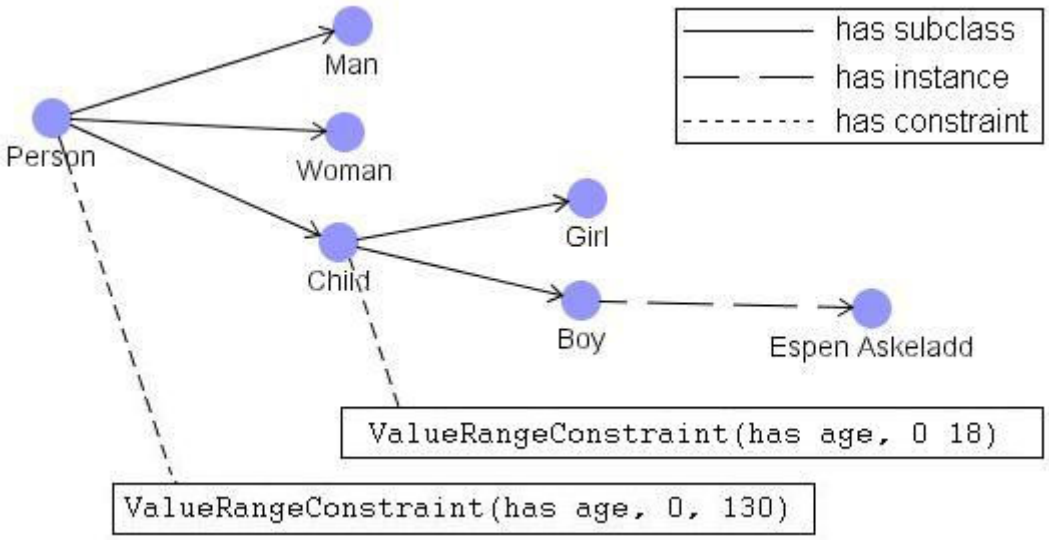


Figure 4.2 - Inherited constraints

In the fairy tale example the constraints is inherited all the way down the hierarchy. Person could have any number of subclasses in an unrestricted number of levels. All would have inherited the constraint on the age. In this case the unlimited inheritance is no problem. If some entity is as subclass of Person, it will probably be relevant to inherit the constraint.

The constraint mechanism in TrollCreek should allow constraints to be inherited through *has subclass* and *has instance of* relations. This solution is inherited from LispCreek. Transferring constraints over other relations do not necessarily make sense. We therefore omit the plausible inheritance mechanism on constraints in this project.

4.3 Constraints on relations or frames

In 2.1 it was mentioned that there are two ways to define constraints on simple <*entity slot value*> relationships. First, we can define the constraint on the pair of entity and slot. This is most easily conceptualised as constraints defined on the entity. Second, the constraint can be defined on the relation, independent on what entity the relation goes from. In TrollCreek the notion of a slot corresponds to the two concepts *relation type* and *relation*. A relation type is represented by an entity that inherits from the entity Relation. A relation from an entity to another can be seen as an instance of the relation type. The first constraint type above corresponds to putting a constraint on the relation instance or the entity and the second to constrain the relation type.

Protégé does not have relation types unless you make new meta classes for slots. By default all slots are instances of :STANDARD-SLOT. A slot is an independent object. This means that a slot once made can be attached to any number of frames. Facets and simple constraints are defined on the slots. This makes both facets and the simple constraints independent on what frames the slots goes from.

Slots in KM are as in TrollCreek defined as frames in the knowledge base. The slot frames have six build in slots. Three of them can be seen as constraints. *Domain* is a make-sense-for constraint – ergo a constraint on the from-frame of the relation. *Range* corresponds to the value class constraint. Finally the slots have a *Cardinality* slot that can take the values *1-to-1, 1-to-N, N-to-1* and *N-to-N.* These slot constraints are accordingly defined on the relation independent on the frame. In addition KM does have a constraint mechanism where the constraints are defined on the frames.

In KL-ONE constraints seem to be defined on the constraint instances. CreekL stores constraints as facets on the slots. Slots are independent frames that live perfectly well without being attached to any frame.

We see that different systems use different approaches in this question. Protégé uses one approach, KL-ONE another, and KM uses both of these. How shall we handle this in TrollCreek? In the Protégé method, adding a constraint to a slot affects all frames that have this slot. Given the example from Figure 4.2 it would have been impossible to define two different value range constraints on the *has age* relation from *person* and *child.* To do this you would have to define two different *has age* slots. Choosing between these two methods, binding the constraints to both the frame and the relation is most flexible. Some constraints are however practical to define on the relation type. If the constraint by nature should have validity on all relations of a given type, it is time-saving to be able to define it once on the relation type and not on all entities having this relation. An obvious example is the makes-sense-for constraint that does not make sense if it is defined on an entity.

In the implementation it will be prioritized to realize the constraint bound to an entity as these seem to give most utility. Constraints on relation types are however too important to be left out of the further discussion. They will therefore be discussed in section 7.3 .

5 Design and implementation

5.1 Representation

After listing the different constraints mechanisms we want to implement in TrollCreek, it is now time to discuss how they can be implemented. In this section we discuss how the constraints can be represented in the TrollCreek architecture, first at the model level and then at the implementational level. Topics concerning how constraints are evaluated are discussed in 5.2 .

5.1.1 Model level

In this section we will discuss how constraints can be included into the model level or conceptual level of the TrollCreek system. We will also examine whether it is possible to represent the constraints solely on this level using existing implementation level structures. If this is possible we will save the time and effort required to implement a low level representational structure for the constraints. We then only have to make some high level adaptations to allow editing and browsing constraints. Evidently we also must develop a constraint checking mechanism. This approach is used in Protégé (described in section 2.6).

In Protégé both classes, slots and constraints have system- or meta-classes describing them. All normal classes are instances of *:STANDARD-CLASS* or some user-defined meta-class. From Figure 5.1 we can see that slots, facets and constraints are represented correspondingly. This is meant as a way of creating templates for different kinds of classes, slots and facets. As described above, Protégé represents the simplest constraints as facets, and the more complex as PAL-constraints. Both are represented in the model as instances of some meta-class. This means that the constraints are stored as an integrated part of the knowledge model. The knowledge representation system does not have to know that these concepts are constraints – they are just some kind of class [23].

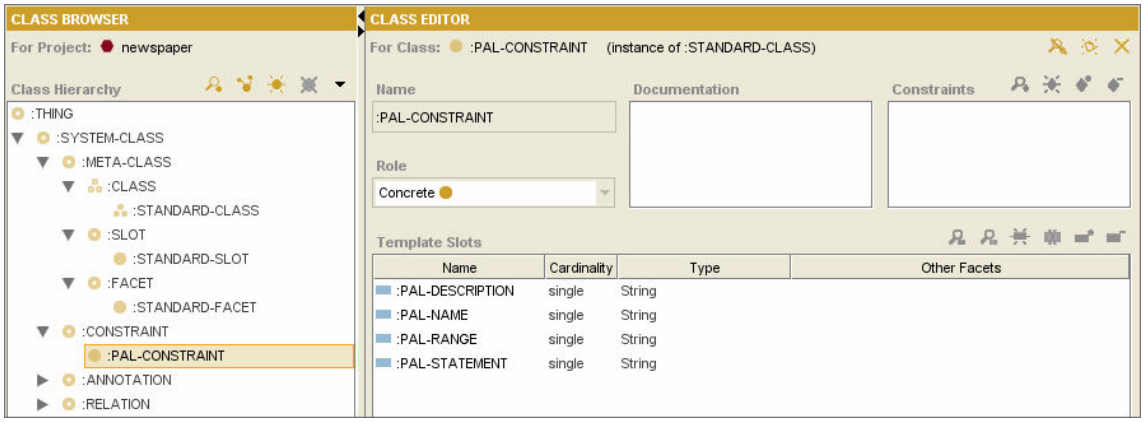


Figure 5.1 - Meta-classes in Protégé

To the left we see the tree of system- and meta-classes in Protégé. The system class representing PAL-constraints is selected and its facets are shown in the class editor to the right.

When creating a new knowledge model in TrollCreek we do not get an empty model. At a minimum a basic epistemological model is included. This model contains some entities and relations playing a role related to Protégé's meta-classes. The most general concept *Thing*, the concepts *Relation* and *Entity* and among other the basic relations *has subclass, has instance* and *has inverse* are part of this ontology. For case based reasoning either *Simple CBR model* or the more extensive *Creek IsoPod model* (see appendix A) are used. These models contain entities and relations used to model and represent cases and relations among cases. By connecting our Creek knowledge models a top ontology, we add semantics to our model. Creek's reasoning mechanisms distinguishes relations and entities, knowing their different roles in a model. Recall from 3.1 that the special relation *Transfers* are used to tell the reasoning mechanism that one relation transfers another.

Following this design we can add the concept of constraints to the top ontology. Figure 5.2 suggests making *Constraint* a subclass of *Descriptive Thing*. With this, the constraints will be integrated into the ontology of the knowledge model, also making a contribution to the overall semantics of the model.

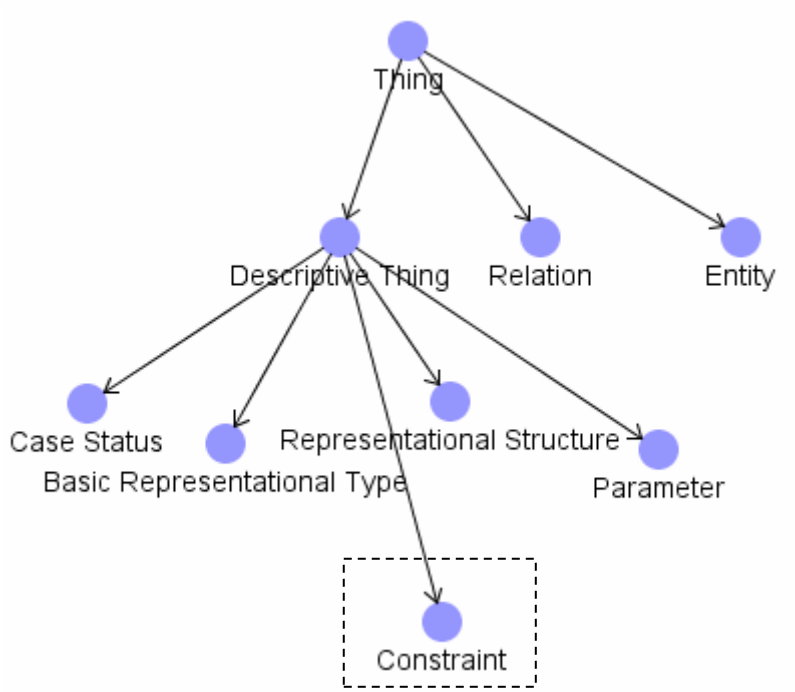


Figure 5.2 - Top ontology of TrollCreek

As described recently Protégé represents constraints as instances of different constraint meta-classes. In TrollCreek this corresponds to representing the constraints as frames inheriting from the constraint frame shown in Figure 5.2.
Copying the Protégé solution is however not straight forward. The representation in Protégé is at a higher abstraction level than in Creek. For example the slots in Protégé can have values of different datatypes (Table 2.15). PAL constraint frames have the four slots *:PAL-NAME, :PAL-DESCRIPTION, :PAL-RANGE* and *:PAL-STATEMENT,* all with data type *string*. All but the first of these typically contain multiple lines of text. In TrollCreek the slots can only have entities as their value. By default entities only have a name describing them while the properties are expressed through the relations. Representing constraints using the name field of entities will result in a messy and unreadable model. TrollCreek do however have a possibility to associate more information to an entity. Every entity has an (possibly empty) attribute called *entityObject* that can hold any java object as its value. The current user interface

enables inserting strings, URLs and numbers into this place. Using this attribute, constraints can be represented as a single entity/frame with a describing name in the entities name field, a description in the entities description field and a constraint object in the *entityObject* field. The constraint object can be a string with a constraint expressed in some constraint language or it can be a special constraint object.

As described above this approach has the advantage of not requiring any changes to the knowledge representation system in TrollCreek. We use the already existing structure to represent our new constraints. The user interface however, must be augmented to allow the definition of constraints. We also have to define how the constraints are to be represented inside the *entityObject.*

At the conceptual level this may seem as a good solution. The constraints are represented in the model as a special kind of frame tidy placed in the ontology alongside entities and relations. The solution may be good at the conceptual level, but using the entities entity object to store the constraint is not preferable. As is mentioned in section 3.2.2 this attribute are primarily used for storing type specific information of an entity. We are then using the underlying entity-structure to store something that is not entities. In TrollCreek's implementation there is a `Relation` class to represent relations and an `Entity` class to represent entities. To be consistent with the existing implementation we therefore have to introduce some class structure at the implementational level to represent the constraints. The solution sketched above can however be used at the model level. Entities and relations are represented both at the implementational level and at the model level. Constraints can be treated the same way adding some semantic to the model.

5.1.2 Implementational level

In section 3.2 we described the structure of TrollCreek's implementation. Here we will discuss possible ways to implement constraint representation into the existing class structure. The internal representation of the constraints should be tidy and ideally not involve changes in too many classes.

Above we suggested representing constraints as a special kind of entity with some constraint object in the entityObject attribute. This was rejected because the entity structure should not be used to represent something not being an entity. Another possibility is to store the constraints of an entity as a list in the entity's entityObject. This allows us to store the constraints without changing anything of the internal knowledge structure of TrollCreek. We just have to make a constraint structure to put as an object in the `entityObject` variable. The only part of the system that has to know this is the constraint checking engine, and of course the user interface component that allows the user to write the constraint. The thought may seem tempting, however, doing this we occupy the entityObject, blocking other data to be stored there. The `entityObject` can only store one object. As mentioned in 3.2.2, storing type-specific data is the intended purpose of the object. Using the `entityObject` is then no good solution either.

From the above we can conclude that some more extensive additions have to be made to the representation. This can be done by borrowing some design decisions from the way relations are implemented. In `EntityData` there is a list of relations that "goes from" the entity. We introduce a corresponding list of constraints in `EntityData`. The entries in the list point to `Constraint` objects that represent the constraints, possibly containing `ConstraintData` objects if necessary. The solution is sketched in Figure 5.3.

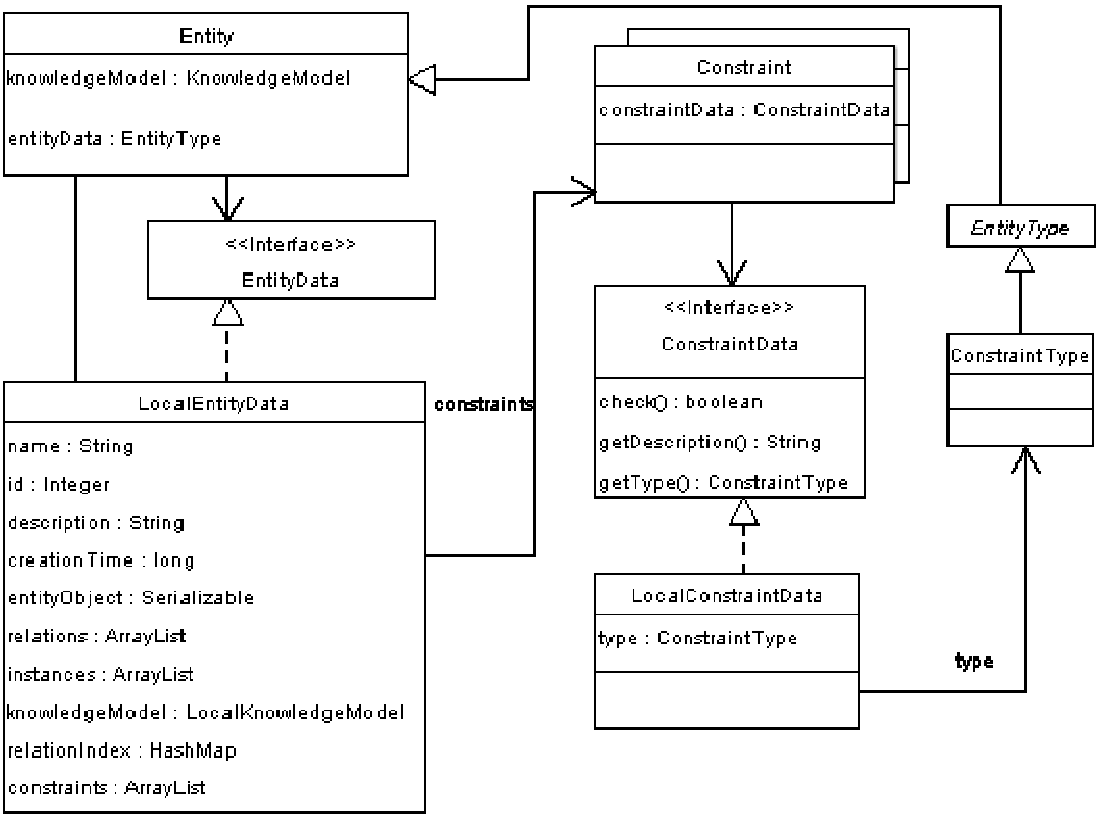


Figure 5.3 - Sketch of a constraint representation in TrollCreek

As described in 3.2.1 one important idea behind the design of TrollCreek is to allow the knowledge model to be implemented and stored different ways. This is the reason why the `KnowledgeModel, EntityData` and `RelationData` are represented as interfaces. When making extensions to TrollCreek it will be favourable to design the new components in the same "spirit" as the existing system. Constraints are therefore represented by a `Constraint` class being the programmatic interface to the constraint class structure. The internal data – or instance variables, to follow the object oriented jargon - of the constraint is stored in some implementation of the `ConstraintData` interface. To make a solution to use with the current version of TrollCreek this interface is implemented by a class called `LocalConstraintData`. To allow the constraint types to be represented in the semantic network, the ConstraintData class points to a class representing its type. This class inherits from `EntityType` just like `RelationType` does.

Constraint objects

In Figure 5.3 we saw a sketch of how constraints could be implemented into and attached to the existing java classes of TrollCreek. This sketch is however somewhat superficial to be used as a guide in the programming process.

So far we have used a top-down approach in discussing how to include constraints into TrollCreek. Now we will go bottom up to se how we can make the design more concrete. Figure 5.3 says that a constraint should be implemented using a constraint object and a constraint data object, just like the way entities and relations are implemented. Constraints are however more complex structures than entities and relations because they have a partly procedural nature.

For a moment we move away from the constraint design discussed so far. Assume we shall make a simpler constraint mechanism than the one discussed. The constraints are not integrated into the knowledge model. Instead we have an external mechanism with constraint objects that state restrictions on the knowledge model. There are one constraint class for each type of constraint. In these classes the constraints are expressed through a check-method that both define and checks the constraint.

The constraints are customized by attaching them to different entities and relations. In this simple mechanism constraints are stated by creating constraint objects with different entities and relations as arguments. Constraint checking is encapsulated into the objects and is as easy as evoking the check-method on all constraint objects. Figure 5.4 illustrates this using a class-diagram. Here all constraint classes implement an interface called `Constraint` that contains a single method `check`. This means that the constraint checking loop can treat all constraints equally despite their different types. The only thing it has to know is that a constraint has a method called check that returns something saying whether the constraint is met or not. In Figure 5.4 the method returns a boolean value – true if everything is ok and false if the constraint is violated. In a real system it should probably return some more information on how or why the constraint is violated.

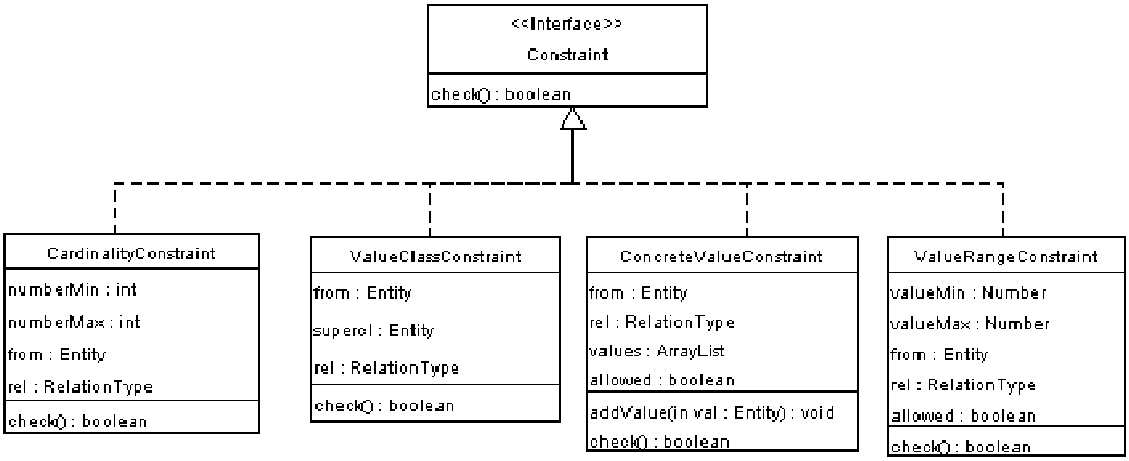


Figure 5.4 - Constraint objects

The general constraint can also get its own constraint object. This object may instead of taking entities and relations as arguments, take a string defining the arbitrarily complex constraint. Such an object must have methods for parsing the string and interpreting the constraint. This could however be hidden inside the object making it look like any of the other constraints. The implementation of the general constraint is discussed later.

The approach described here is certainly too simple to meet the needs for a constraint mechanism in TrollCreek. However, it may be used as an inspiration or a starting point. Especially the simplicity of constraint checking is tempting to exploit. Encapsulating complex details inside objects is one of the central things in object oriented programming. It is also quite easy to express the constraints as java code in a check method. The `representation` package in the TrollCreek implementation has a rich API that enables us to easily get information from the knowledge model. There is for instance no need to write intricate algorithms for traversing the graph representing the semantic network.

Connecting constraint objects to the model

We will now seek to find a way to sew together this simple mechanism with the overall design from Figure 5.3. Above we decided to follow the design principles from Entity and Relation having a `Constraint` class which wraps a `ConstraintData` class containing all the data of the constraint. The constraint classes from Figure 5.4 represent the core of the constraint. In addition we may need some way to store a description of the constraint. These data fits naturally into the `ConstraintData` object. Next we have to figure out whether the different constraint objects shall be attributes in the `ConstraintData` objects or if they themselves should play the role as `ConstraintData` objects. This question leads us to another important decision: At which "level" in the code do we distinguish between the different constraint types? Should we replace the one top level `Constraint` with four or more specified constraint objects, or should there probably be one `Constraint` class that can hold different types of `ConstraintData` objects? Again we borrow design decisions from the rest of the TrollCreek system. An analogous phenomenon to the different types of constraints is the different types of entities. In the current implementation we can mention `Case` and `NumberEntity` as special forms of the entity object. These two classes extend[3] the `Entity` class through the `EntityType` class (Figure 5.5).

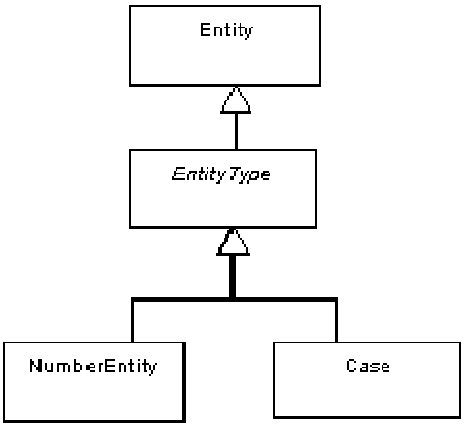


Figure 5.5 - Hierarchy of entity types

What distinguishes NumberEntity from Entity is the ability to store a number in the `entityObject` attribute. This involves a new constructor method taking a number as argument, and some methods to access and manipulate the number. By using the specialized entity class that knows how to handle numbers, the standard `EntityData` class can be used. EntityData does not know, and do not have to know, what kind of data it stores in its `entityObject` attribute. Similarly, by having different constraint objects at the top level that handles the differences, we can make a general ConstraintData class that handles different types of constraint data.

In Figure 5.6 we suggest a class design for the constraint mechanism. At the top of the diagram we find the `Entity` class. Because all constraints listed in the requirement section are centred around an entity, it is practical to tie the constraint structure to the entities. As we saw above the `Entity` object itself do not hold any data. All its data are hold by an entity data object. At the top of the constraint structure we have the abstract class[4] `Constraint` that act as a super class for the classes representing the different

[3] In the Java terminology the word *extend* is used to say that a class inherits from another class. It is also used as a key word in the programming language

[4] A class that is declared `abstract` cannot be instantiated, but must be subclassed by a child class. It can, in contrast to an interface, contain instance variables and complete methods.

constraint types. Methods and instance variables that are common for the different constraint types are placed in this class. As with `Relation` and `Entity` the `Constraint` class just act as a wrapper or programmatic interface for the data objects. Its only instance variables are pointers to the `KnowledgeModel` and the `ConstraintData`. Detailed class diagrams showing all attributes and methods are found in appendix B .

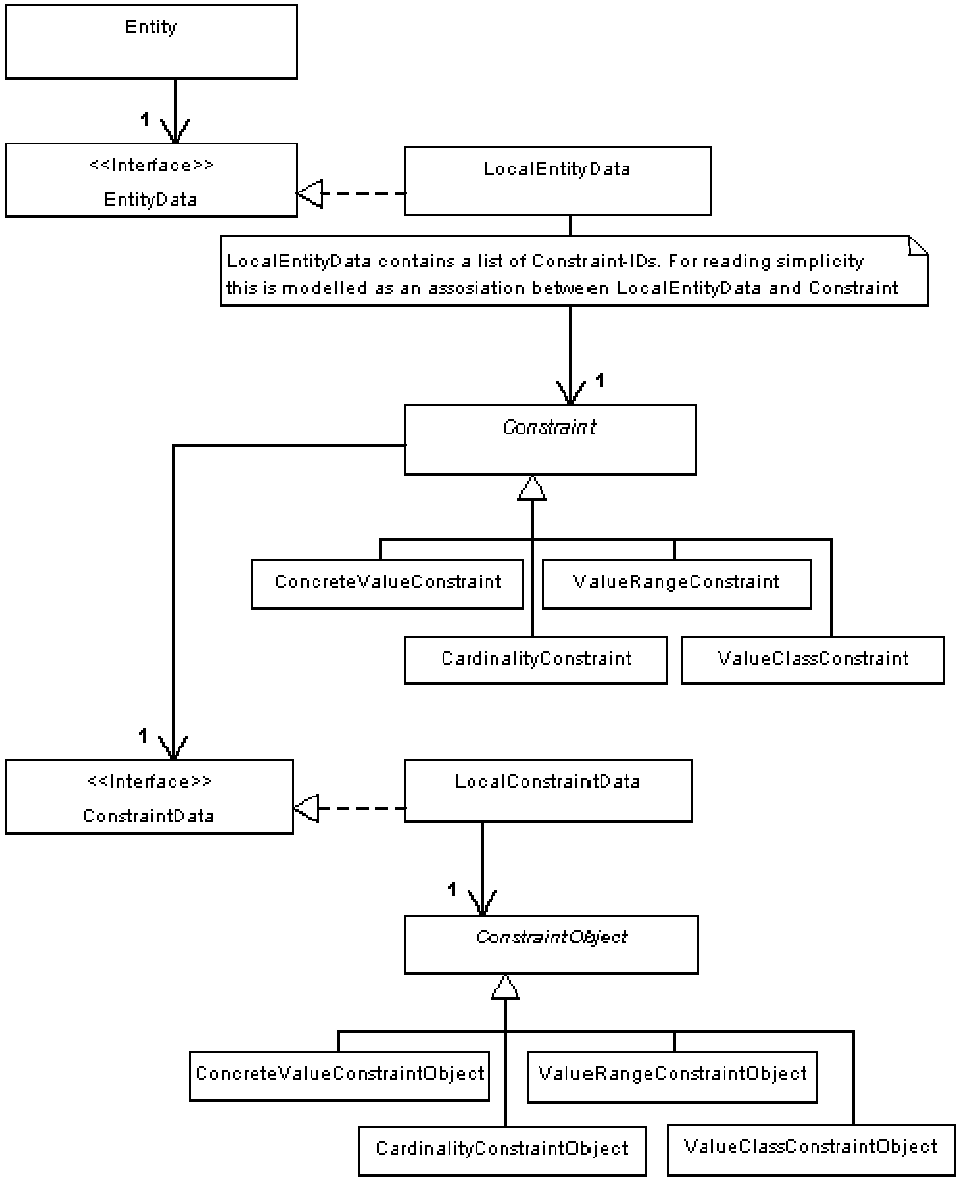


Figure 5.6 - Class diagram of constraint types

The figure shows the structure of the classes. Extensive diagrams with attributes and methods are listed in Appendix B

The interface `ConstraintData` defines all the public methods that the constraint data classes shall have. In this project this interface is implemented by `LocalConstraintData` that are written to work together with `LocalEntityData` and `LocalRelationData`. The `LocalConstraintData` contains the constraints ID, its description and a constraint-object that is an adaptation of the constraint classes from Figure 5.4.

`ConstraintObject` is an abstract class that is sub-classed by classes for the different constraint types. These classes are responsible for handling the behaviour of the constraints. Appendix B shows the details of `ConstraintObject` and its subclasses. The core of these classes is the check method that encapsulates the mechanisms for evaluating the constraint. The constraint is defined in this method that returns an object of type `ConstraintCheckResult` that encapsulates the answer from the check. Checking the constraint is done by evoking this method and the result from the check can be requested from the result object. This object contains a boolean value that says whether the constraint is violated or not, and a string with an explanation. If we in the future want to give the results another form, it is easy to put more data into this object. As an example we can imagine a scenario where we want to give a graphical view of a constraint violation. Then we probably will return the entities and relations involved in the violation.

5.1.3 General constraints

We have now made some decisions about how to implement the four simpler constraint types. The implementation of the general constraint mechanism is however not discussed yet. With the constraint design we have build so far, it is straight forward to add a new subclass of `Constraint` and `ConstraintObject`.
We now have to decide how the general constraint shall be expressed. As written earlier the most obvious way to express such constraints is through a written language, either the native language of the representation system or a special constraint language. The first approach is used in the Lisp version of Creek were all slot and facet values can be lisp expressions. In Protégé the most complex constraints are expressed through the Protégé axiom language.

Constraint language

A possible way to solve this in TrollCreek is to define a constraint language (possibly stolen from KM or Protégé) and make an interpreter for it. The interpreter translates the statements into some underlying constraints that must be defined in the system or directly to the native language of TrollCreek. The advantage of this approach is that you can accommodate the language to perfectly fit the conceptual model of the TrollCreek knowledge representation and such making it easier to use. To write an interpreter for such a language that translates to Java, the native language of TrollCreek, is however a quite demanding task involving problems attached to parsing etc. Making a new language also forces the user to learn a new language.

Write constraints in java

A possibly better and certainly easier solution is to allow the user to write the complex constraints in Java. Loading and unloading classes in Java is not straight forward, however there are code libraries available that emulates a Java virtual machine and can interpret Java source code dynamically. In this project DynamicJava [10] is used to prove the concept. This package contains a method that can be called with a piece of Java source code as argument and returns a Java object. The source code can be a complete class or just some lines of code.

```
private boolean constraint(Entity e){
      /* write your constraint here. return false if the
       * constraint is violated, else true
       */
}

private boolean constraint(Entity e){
      /* Cardinality constraint for the relation has owner.
      get all relations of type "has owner" */

      Relation[] rels = e.getRelations("has owner");

      /* If there are more than 1 owner, return false */
      if(rels.length > 1)
            return new ConstraintCheckResult(
                        false,
                        "more than one owner");
      else
            return new ConstraintCheckResult(true, "No problem");
}
```

Table 5.1 - General constraints in Java

In Table 5.1 it is shown how this can be done. The user gets an empty method with complete declaration where she can write her constraint. It is also shown how a simple cardinality constraint can be expressed, by counting the length of an array containing the relations of a specific type.

The disadvantage of this solution is that the user must have thorough knowledge about the internal representation of TrollCreek. This is not necessarily designed to fit the surface representation or be easily understood. However, the gap between the representation and the user interface in TrollCreek is not very large.

A way to make it easier to write such constraints is to make a set of wrapper-methods that supports the user with information relevant to constraint writing. This can simply be to supply already defined methods with new constraint-like names, or it can be a wrapping of more complex operations into one method. Such methods will not do anything with the functionality of the mechanism, but rather act as syntactic sugar, easing the job of the user.

5.1.4 Inheritance

In section 4.2 we stated that constraints in TrollCreek should be inherited over subclass and instance relations. The design currently defined do not support inheritance. In Figure 5.4 the constraint objects contains instance variables representing the from-entity and the relation that is involved in the constraint. The two instance variables were intended to be set during the initiation of the objects of this class. Consequently the constraint is bound to a given entity.

To allow constraints to be inherited it is not practical to store the from-entity in the constraint checking object, as was done in Figure 5.4. We must be able to check all constraints with many different entities. In addition to checking them with the entity that owns them, they must be checkable with the entities that inherit from the owner. One way to enable this is to send the entity to be checked as an argument to the constraint checking method.

When checking an entity's constraints we can iteratively check the constraints first with the entity itself and then with all its subclasses. Alternatively, when requesting all constraints from an entity, the entity can return its own constraints and all the

constraints of its super classes. Then we can check all these constraints with the entity. The latter is chosen in the implementation. Both approaches give the same result.

5.2 Constraint checking

Having established a representation of the constraint mechanism, it is time to decide how the constraints should be checked. It is already settled that the individual constraints are checked by calling the constraint object's check method. There is however many questions still to be answered.

5.2.1 Time of constraint checking

First we discuss when the constraints shall be checked. Cyc checks constraints when a new piece of information is introduced to the knowledge model. Doing this they assure they never get erroneous knowledge in the model. The same approach is followed by the lisp version of Creek. In LispCreek the user gets a message that the value entered is illegal and has been rejected. KM has a lazy approach compared to these two. In KM the constraints are checked during inference. When information from a frame is requested the frames constraints are checked and the user notified if the result is negative. In Protégé no constraints are checked unless the user asks for it. PAL constraints are run on demand. Violations are presented to the user, but the system does nothing to force a correction. The simpler constraints defined in the facet however, are not really checked at all. They are enforced by the graphical user interface that does not allow the user to insert erroneous knowledge.

In the frame systems examined above we see that there are mainly three different approaches to the question of when the constraints shall be checked.

1) During insertion of knowledge (as in Cyc and CreekL)
2) During inference (as in KM)
3) On demand (as in Protégé)

Checking constraints during insertion of knowledge ensures that all knowledge in the model is consistent. Erroneous knowledge is not allowed to be inserted. This is of course favourable, but the solution is not problem-free. In the process of building or editing a knowledge model there may be times when the model is temporary inconsistent. The most obvious example is perhaps cardinality. If we have a cardinality constraint stating a minimum number of values for a slot, it will consequently not be met when the first value or values are entered. The other constraints are probably possible to obey during the building of a model, but creation of entities and relating them must be done in the right order to avoid violations. This problem can be solved by allowing the constraint checking to be temporarily turned of or relaxed. However, if we want a 100% consistent model, it must be possible to check the whole model after activating the constraints again.

KM's solution with constraint checking during inference is not subject to the problems just mentioned. Because TrollCreek is both a frame-based system and a CBR system TrollCreek have no exact parallel to KM's inference. In CBR systems the reasoning process has four main steps: retrieve, reuse, revise and retain [4]. When comparing to KM we should however focus on the frame part of the system. Every time a node in the TrollCreek semantic network is viewed in the frame view, all its relations (slots) are shown – both local and inherited. The underlying task done by the system to do this must be seen as the inference process of the frame-based representation. If we should adopt KM's check method, this would be the place to do it. In TrollCreek the construction of a model involves using the frame view. If the constraints should be checked during inference, this will trigger constraint checks. This is however not a

problem because the frame view can be set to only show local relations. Constraint checking in a system based on inheritance can be computationally expensive if the knowledge model is large. This is also true for the inference process. To check constraints every time a frame is viewed can then make the knowledge system needlessly slow.

Finally we have the method used in Protégé Axiom Language where the constraints are evaluated on demand. The user can choose from a list which constraints to check and then run an evaluation on the selected ones. This solution avoids the problems stated above. During the creation of the model we are neither obstructed nor disturbed by the constraint mechanism. When we want it we can run an evaluation to get the constraint mechanism's opinion of our work.

The constraint mechanism designed above is flexible enough to deal with different checking procedures. TrollCreek fires a model change event every time a change is made to the knowledge model. This makes it easy to implement a mechanism that checks whether the constraints hold after every change to the model. The other checking mechanisms are also easily implemented. The most flexible solution is to let the user decide how and when the evaluation of the constraints shall be done. During different phases in the development of a knowledge model, or perhaps due to different styles of working, we can use different checking methods. We can then exploit the strengths of different methods. In the implementation of this project we will for simplicity chose one of the methods. Because the Protégé approach is most flexible we will chose something close to that. We will give the user the ability to either check all constraints in the knowledge model or check constraints, local and inherited, on a specific entity. The constraint mechanism will only be a guide to the user. It will not do anything to reject erroneous values.

5.2.2 Summary

In this section we have discussed how the constraint mechanism can be implemented into the TrollCreek system. The focus has been on the discussion and not on describing the final result. Javadoc and source code for the implementation can be found in the file archive attached to this thesis.

6 User interface

After discussing how to implement constraints in TrollCreek, it is time to study the result. This section describes the user interface created to manipulate the constraint mechanism. An illustrative demonstration of the functionality of the constraint system is given in section 7.1

6.1 User interface

In TrollCreek knowledge editor there are two main components in the UI. Map view shows the semantic network and frame view shows the frame of the node selected in the map view. The UI is shown in Figure 6.1.

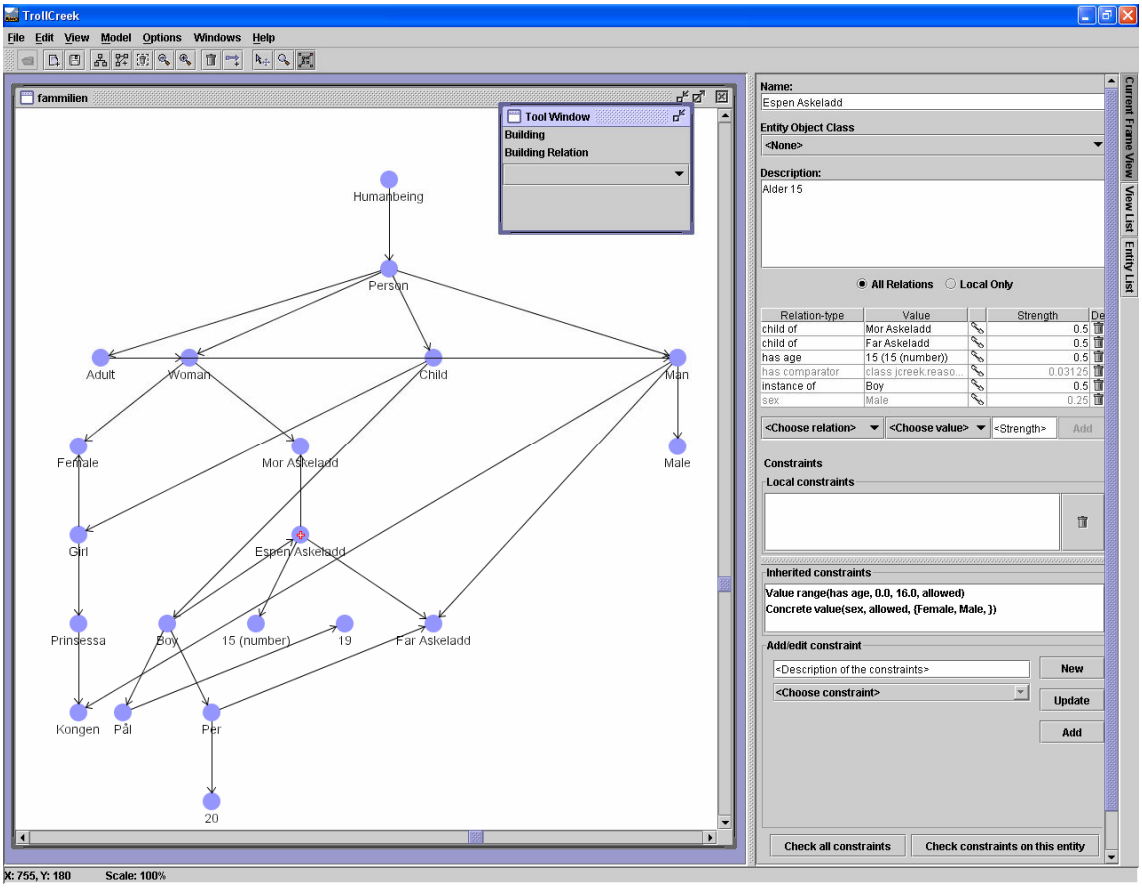


Figure 6.1 - TrollCreek knowledge editor

Editing the knowledge model is mainly done in the frame view part of the window. As described in 3.1 the frame view contains name, description and slots of the frame. To allow viewing and editing constraints a pane for handling constraints is added to the bottom of the frame view. To put it in the frame view was a natural choice as long as the constraints implemented are bound to the frames. The frame view with constraint pane added is shown in Figure 6.2.

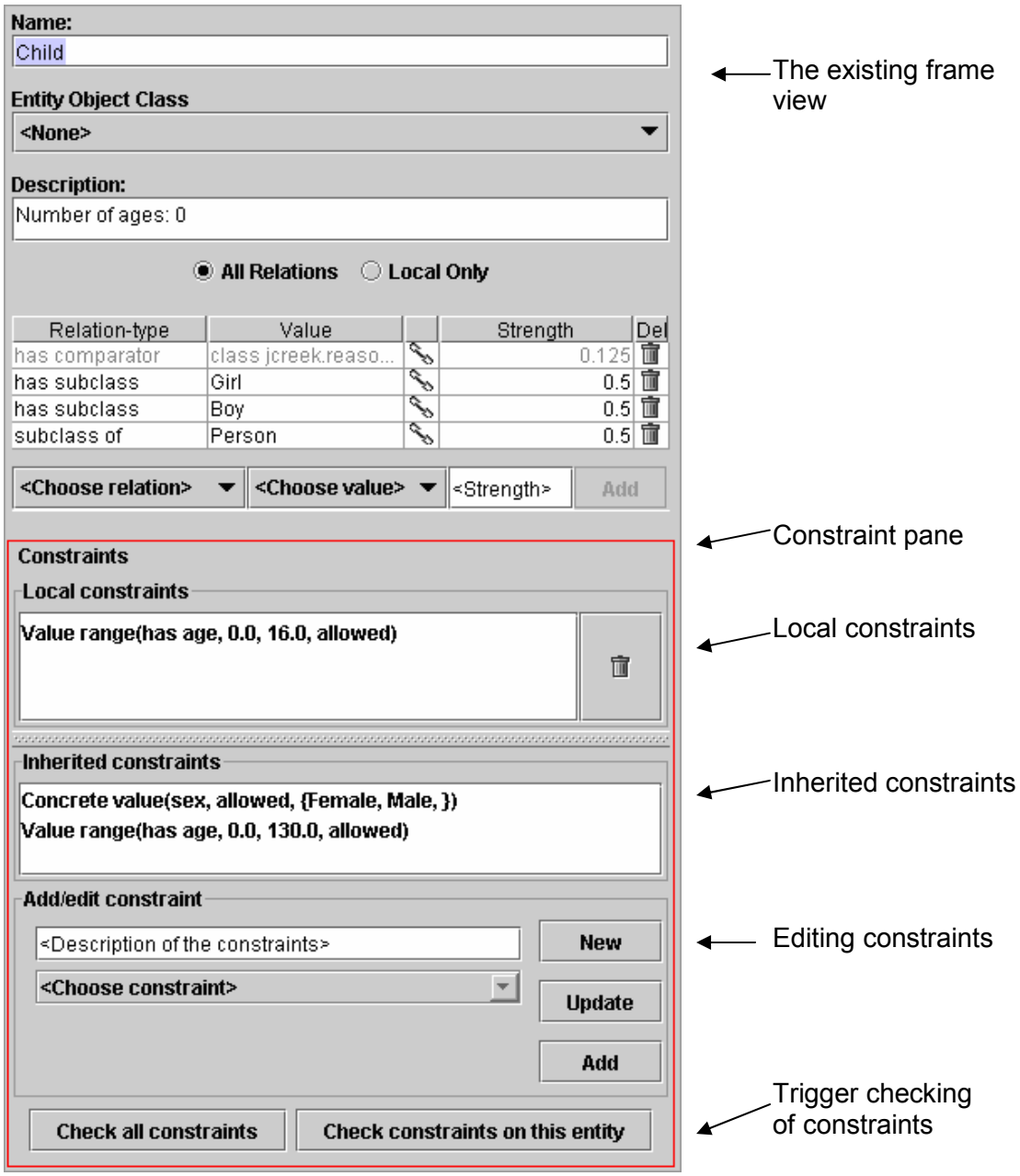


Figure 6.2 - Frame view with constranints

A constraint pane (in the red rectangle) is added in the bottom of the existing frame view pane.

The constraint pane contains two lists of constraints, one for local constraints and one for inherited constraints. The local constraint list has a button for deleting constrains next to it. Below the lists of constrains there is a panel for creating and editing constraints. After selecting a constraint type from the dropdown list more UI-components appears. The different versions of the *Add/edit constriaint* panel are shown in Figure 6.4 through Figure 6.8.

Creating a new constraint is done by defining the constraints properties in the form and then hitting the *Add* button. The constraint will then appear in the list of local constraints, and correspondingly in the inherited constrains list of the frames subclasses.

To edit a constraint, select the constraint from the local constraint list. The constraint will then appear in the *Add/Edit constraint* panel. Hitting the *Update* button after changing the constraints properties, will update the constraint. The *New* button does no more magic than clearing the *Add/Edit* panel.

Constraints are evaluated by pressing one of the two buttons in the bottom of the frame view. There are one button for checking all constraints for the given entity, and one for checking the whole model. From a human-computer interaction (HCI) point of view, the *Check all* button is misplaced. It should be placed somewhere naturally mapped to the whole model, and not in a window showing one single constraint. This project is however not about HCI, and placing the button there was time-saving.

When checking the constraints the constraint check result window (Figure 6.3) shows a list of the violated constraints with some information on how/why the constraint is not met. The checking mechanism does not do anything to enforce the constraint. The decisions on how to solve the problems is left to the user.

Figure 6.3 - Result of constraint check

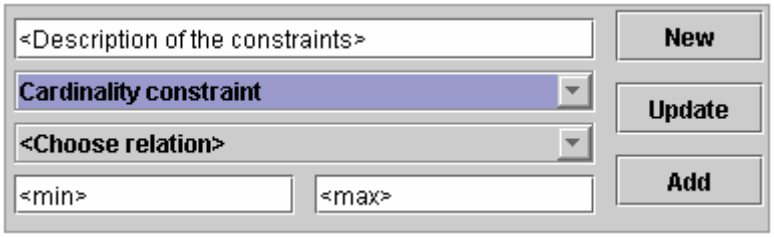


Figure 6.4 - Cardinality constraint form

A cardinality constraint is defined by the relation type it constraints and the minimum and maximum number of allowed values.

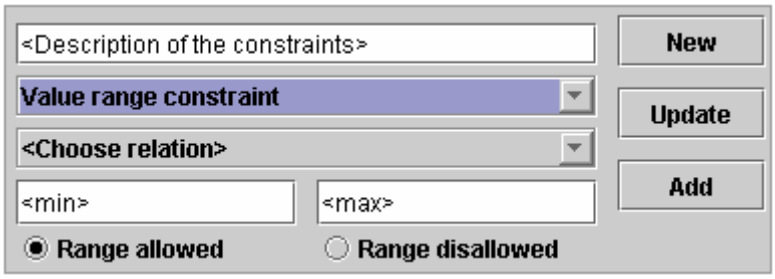


Figure 6.5 - Value range constraint form

A value range constraint is defined by the relation type it constraints and a minimum and maximum number making a range of values. The range can be allowed or disallowed as values of the relation. This is toggled by the radio buttons.

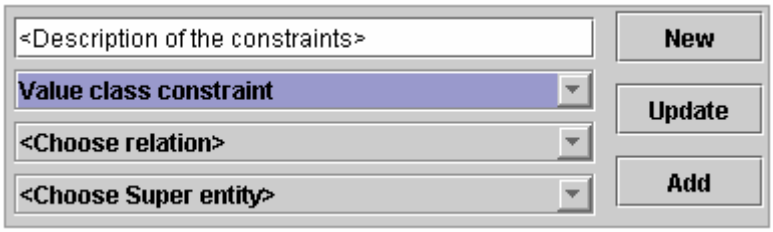


Figure 6.6 - Value class constraint form

A value class constraint defined by the relation type it constraints and the allowed super frame.

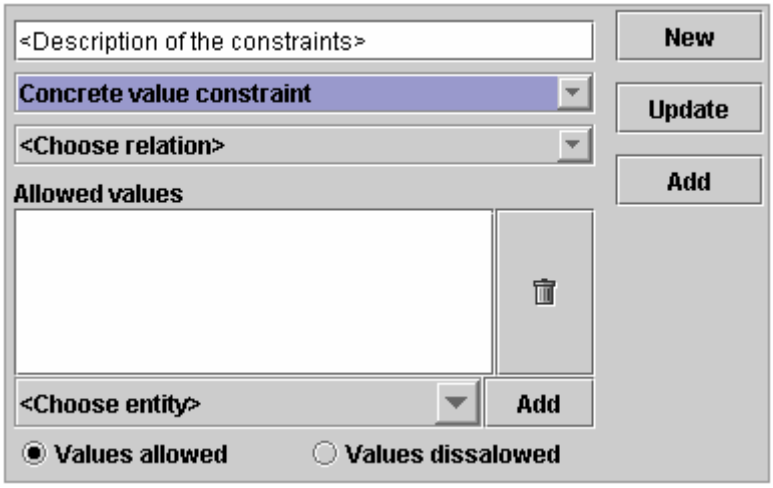


Figure 6.7 - Concrete value constraint form

A concrete value constraint is defined by the relation type it defines and a list of values/entities. Values are picked from the dropdown-list below the list of values, and added to the list by hitting the *Add* button. The list of values can either be allowed or disallowed as values for the relation. This is toggled by the radio buttons.

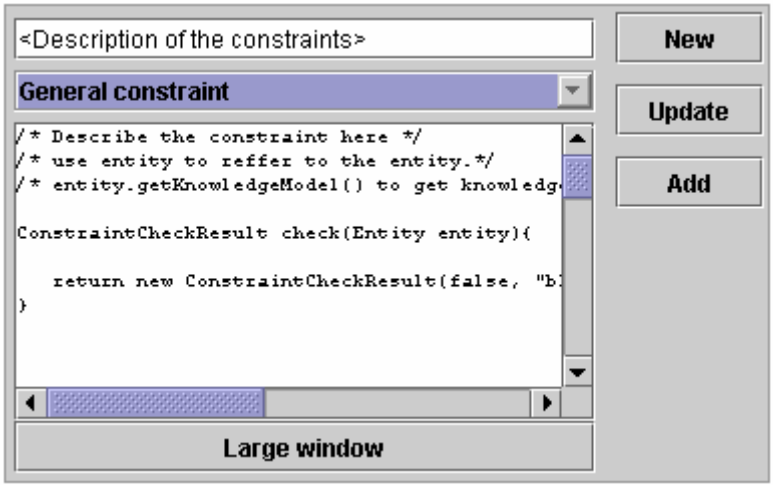


Figure 6.8 - General constraint form

A general constraint is defined by a script written in Java. The script must contain a method called check, but can be more extensive. By hitting the button labelled *Large window* a larger editor window pops up.

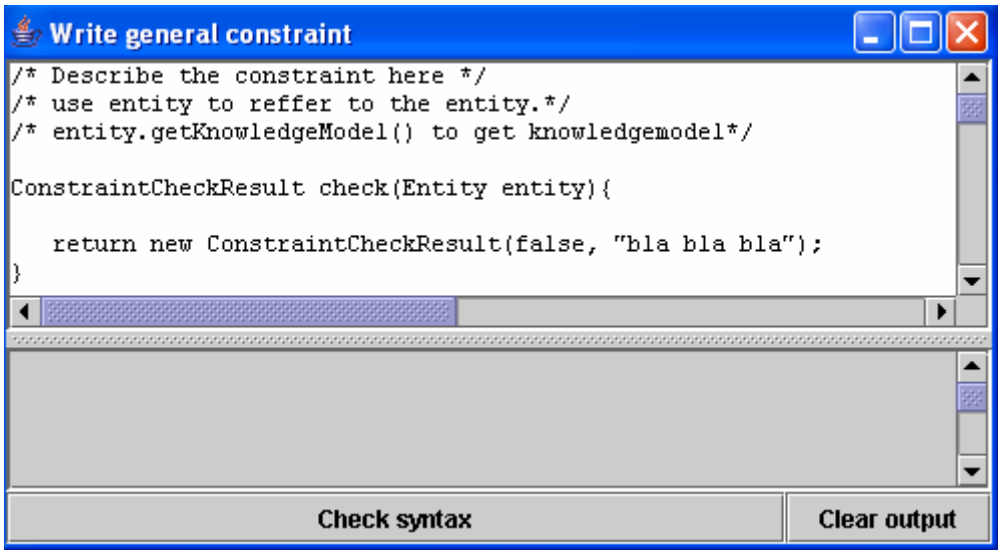


Figure 6.9 - General constraint form (advanced)

This window offers a bigger editor window, making it easier to write complex constraints. Hitting the *Check syntax* button runs the constraint. If there are syntax errors a descriptive message is given in the grey bottom panel.

7 Evaluation and discussion

7.1 Proof of concept

This project does not contain any mathematical description of the constraints mechanism. We can then not give any proofs that the mechanism works. In this section we will test out the constraints and se whether they give reasonable results.

As a test model we use an ontology from the Norwegian folk tale character Espen Askeladd and his family and friends. Espen has a mother and a father, although they are not mentioned in the tales. He also has the older brothers Per and Pål. In most of the stories Espen Askeladd attends to some kind competition where he, by impressing the King, wins the Princess and half of the kingdom. The ontology is shown in Figure 7.1.

Using this knowledge model, we will test the system by creating constraints and trying to violate them. The cardinality and value range constraints are defined by a maximum and a minimum value. Constraints are accepted by the system even if one of the values is undefined. This makes sense, because sometimes we want to constrain only the upper or lower bound of an interval. In the representation of the constraints below the undefined values are denoted by *–inf* for lower bound and *+inf* for upper bound.

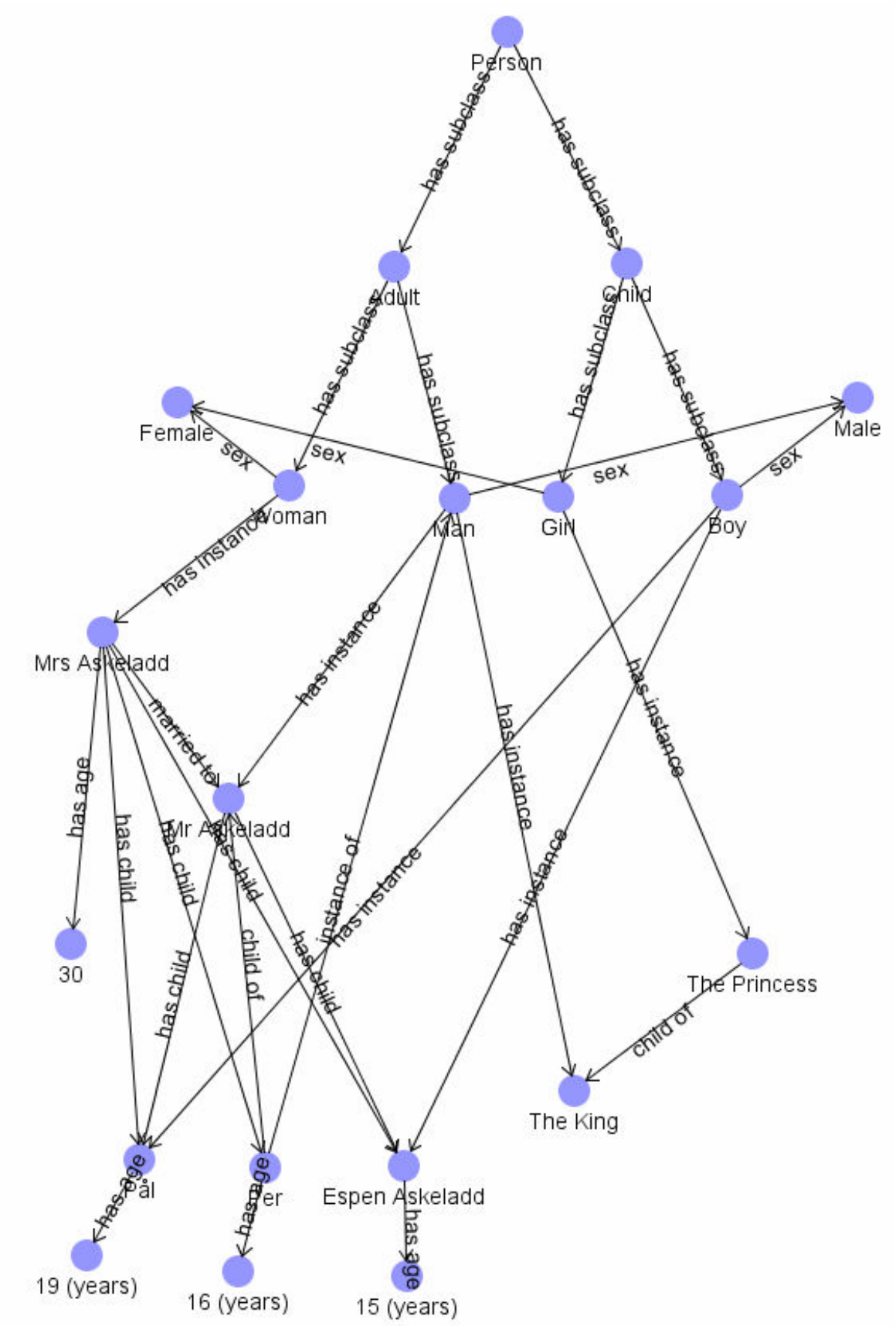


Figure 7.1 - Test ontology, Espen Askeladd

7.1.1 Cardinality constraint

To evaluate how the cardinality constraint mechanism works, we define some cardinality constraints on the above model.

1) A person can have only one age.

```
Person      cardinality(has age, 0, 1)
```

2) An adult person can be married to one or zero persons.

```
Adult       cardinality(married to, 0, 1)
```

3) A child can not be married.

```
Child       cardinality(married to, 0, 0)
```

Then we introduce some relations violating the constraints.

```
Mrs Askeladd -> married to -> The King
Espen Askeladd -> married to -> The Princess
Per -> has age -> 15 (years)
```

First, Mrs Askeladd is already married to Mr Askeladd. Marrying the King she violates constraint 2. Second, both Espen Askeladd and The Princess are children. According to constraint 3 they can not get married. Finally, Per already has 16 (years) as his age. Another age violates constraint 1.
When running the check we get the following result:

```
4 constraints violated

1 ---
Cardinality(married to, 0, 0)
Cardinality for Espen Askeladd -> married to should be 0 - 0
Espen Askeladd -> married to -> The Princess

2 ---
Cardinality(married to, 0, 1)
Cardinality for Mrs Askeladd -> married to should be 0 - 1
Mrs Askeladd -> married to -> The King
Mrs Askeladd -> married to -> Mr Askeladd

3 ---
Cardinality(has age, 0, 1)
Cardinality for Per -> has age should be 0 - 1
Per -> has age -> 16 (years)
Per -> has age -> 15 (years)
4 ---
Cardinality(married to, 0, 0)
Cardinality for The Princess -> married to should be 0 - 0
The Princess -> married to -> Espen Askeladd
```

Violation 1 and 4 in the result detects the fact that Espen Askeladd and the Princess can not get married. The second violation shows that the polygamy of Mrs Askeladd is detected. Per's confusion according to age is showed in the third violation.

The experiment shows that the cardinality constraint mechanism works for this test model. We see that the constraints are correctly inherited down *has subclass* and *has instance* relations.

7.1.2 Concrete value constraint

We define some concrete value constraints.

1) A person's sex must be male or female

```
Person Concrete value(sex, allowed, {Male, Female})
```

2) A man's sex must be male

```
Man Concrete value(sex, allowed, {Male})
```

3) A woman's sex must be female

```
Woman Concrete value(sex, allowed, {Female})
```

Then we add some knowledge to the model that will violate the constraints.

```
The King -> sex -> Female
Espen Askeladd -> sex -> Man
```

The King is already defined as an instance of Man. His sex is then constrained to be Male. Espen Askeladd is instance of Child that is a subclass of Person. His sex should then be Male or Female. These two errors should be detected by the mechanism.

2 constraints violated

1 ---
Concrete value(sex, allowed, {Male, Female, })
Concrete values allowed: Male, Female,
Espen Askeladd -> sex -> Man

2 ---
Concrete value(sex, allowed, {Male, })
Concrete values allowed: Male,
The King -> sex -> Female

We see that the mechanism detects both the predicted violations.

7.1.3 Value range constraint

To evaluate the value rage constraint, we define three constraints in the knowledge model.

1) A person's age must be between 0 and 150

```
Person      Value Range(has age, 0, 150, allowed)
```

2) An adult's age must be 16 or higher

```
Adult       Value Range(has age, 16, +inf, allowed)
```

3) A child's age must be no more than 16

```
Child       Value Range(has age, -inf, 16, allowed)
```

We add the relations

```
The King -> has age -> -3
Pål -> has age -> 19
```

Recall that the king is an instance of Man that is a subclass of Adult that subclasses Person. Pål is defined as a child. The age of the king is outside the allowed range for Person (0-150) and the allowed range of Adult (16- inf). Pål should as a child not be

more than 16 years old.

```
3 constraints violated

1 ---
Value range(has age, -inf, 16.0, allowed)
Pål -> has age -> 19.0  |  Allowed (n < 16.0)

2 ---
Value range(has age, 16.0, +inf, allowed)
The King -> has age -> -3.0  |  Allowed (16.0 < n)

3 ---
Value range(has age, 0.0, 150.0, allowed)
The King -> has age -> -3.0  |  Allowed (0.0 - 150.0)
```

The constraint mechanism finds all these violations.

7.1.4 Value class constraint

We evaluate the value class constraint by defining two constraints.

1) A child of a person must be a person

```
Person      Value class(has child, person)
```

2) An adult must be married to an adult(if he/she is married at all)

```
Person      Value class(married to, adult)

Mr Askeladd -> married to -> Number
Mr Askeladd -> has child -> Number
Number -> child of -> Per
```

We then add three relations that violate the constraints. Mr Askeladd is married to something that is not n adult person, and he has a child that is not a person. To demonstrate how the inverse relations affect constraints, we add a relation saying that Number is child of Per. There are no constraint defined on *Number*'s *child of* relation. However, defining this relation the inverse *Per –> has child -> Number* is automatically added.

```
3 constraints violated

1 ---
Value class(married to, Adult)
Value class allowed Adult
Mr Askeladd -> married to -> Number

2 ---
Value class(has child, Person)
Value class allowed Person
Mr Askeladd -> has child -> Number

3 ---
Value class(has child, Person)
Value class allowed Person
Per -> has child -> Number
```

We see that the two relationships between Mr Askeladd and Number is detected as violations. The third violation comes from the inverse relation of *Number -> child of -> Per.*

7.1.5 General constraint

To demonstrate the general constraint mechanism we write a constraint that is supposed to validate that the value of an *has age* relation is a number. In TrollCreek a number is represented by an entity holding a number as its entity object. The constraint then says that the value of an *has age* relation must be an entity with a number (subclass of `java.lang.Number`) in the `entityObject` attribute. The constraint is defined on the Person frame. Java code for the constraint is presented in Table 7.1.

```
ConstraintCheckResult check(Entity entity){
      Entity age;
      boolean violated = false;

      try{
            age = entity.getRelation("has age").getValue();
      }catch(NoSuchRelationException e){
            /* If the person does not have an age,
                  the constraint is not violated */
            return new ConstraintCheckResult(true, "No age");

      }

      /* If the value of "has age" is an entity with no entity object,
the constraint is violated */

      if(age.getEntityObject() == null){
            violated = true;
      }else{
            /* If the entity object is not a number,
                  the constraint is violated */
            if(!(age.getEntityObject() instanceof Number))
                  violated = true;
      }

      /* Return the result of the constraint check */
      if(violated){
            String res = entity.getName() + " -> has age -> " +
age.getName() + "\n" + age.getName() + " is not a number";

            return new ConstraintCheckResult(false, res);
      }else{
            return new ConstraintCheckResult(true, "ok");
      }
}
```

Table 7.1 - General constraint

We add the relation
`The Princess -> has age -> Child`

When evaluating the constraint we get the following result:

```
1 constraints violated

1 ---
The Princess -> has age -> Child
Child is not a number
```

7.2 Weaknesses with the solution

7.2.1 General constraints

Writing constraints in java using the inner representation of TrollCreek directly is not trivial. Java is a general programming language, not a constraint language. As an example we will try to write a constraint saying that a person's child must be at least 12 years younger than the person.

```
1   public ConstraintCheckResult check(Entity from) {
2       Number parentAge = (Number) (from.getRelation("has
    age").getValue().getEntityObject());
3       Relation[] children = from.getRelations("has child");

4       //For all children of the person
5       for(int i = 0; i < children.length; i++){
6           // Get the age of the child
7           Number childAge = (Number)
    (children[i].getValue().getRelation("has
    age").getValue().getEntityObject());
8
9           // If the age difference between parent and child
10          // is less than 12 years the constrait is violated
11          if((parentAge.floatValue() - childAge.floatValue()) < 12){
12              String res = children[i].getValue().getName() + " is
    to old to be child of " + from.getName();
13              return new ConstraintCheckResult(false, res);
14          }
15      }
16
17      return new ConstraintCheckResult(true, "All is ok");
18  }
```

Table 7.2 - General age constraint

Intuitively we would write the constraint like in Table 7.2. First we get the persons age by getting the *has age* relation of the person entity, and requesting the entity object of the entity representing the age. Second we get the array of *has child* relations of the person. Iteratively we get the age of each child and comparing it with the age of the parent. If the child is to old, we return a negative answer. If no violation is detected, we return a message that all is ok.
This solution will work in some cases, but not all. If for example one of the children has no age slot, a `jcreek.representation.NoSuchRelationException` is thrown. Consequently we will have to either catch the exception or make sure the failure does not happen. An experienced java programmer will have no problems handling exceptions. We do however assume that the systems will be used by persons not

familiar with java. The constraint above will have to be rewritten to work generally. Lines 2 and 3 must be replaced by the code in Table 7.3 With these changes the constraint works and is stable.

```
Relation[] children = null;
Number parentAge = null;

try{
      children = from.getRelations("has child");
      parentAge = (Number) (from.getRelation("has
      age").getValue().getEntityObject());

}catch(NoSuchRelationException e){
      return new ConstraintCheckResult(
               true,
               "Parent got no children or no age");
}
```

Table 7.3 - Exception thandling in general constraints

Writing general constraints in java is not as flexible as it should be. As we have seen, the user must know the implementational details of the knowledge representation including error handling and exceptions. Another weak point is that the constraints are defined as methods or functions. In the early years of work on knowledge representations there was a debate regarding procedural versus declarative representation of knowledge; knowing *how* versus knowing *that* [16]. In the implemented solution the user has to define how the constraint is to be checked instead of stating how the ideal model should be. Most frame-based knowledge representation systems use a logic-like constraint language that allows the constraints to be defined declaratively.

7.3 Missing functionality

During this text a lot of useful functionality is suggested for the TrollCreek system. Because of time restrictions not all is implemented. In this section we will discuss this functionality and sketch how such mechanisms can be implemented.

7.3.1 Constraints on relation type

In 4.3 it was discussed whether the constraints should be stored on the entity or on the relation type. In the implementation the first of the two was chosen. However, defining constraints on the relation type can also be useful.

Because relation types are represented as entities in the knowledge model there is already a possibility to place constraints on them. We can constrain the ontology of relation types. This can be constraints like:

- Relation types inheriting from *Temporal Relation* must have *default explanation strength* lower than 0.5 (value range).
  `Isa(x, Relation Type) ∧ Isa(x, Temporal Relation) → DefaultExplanationStrength(x, y) ∧ LessThan(y, 0.5)`

- A relation type can have only one *has inverse* relation (cardinality).
  `Isa(x, Relation Type) → Equals(NumberOfValues(Has Inverse), 1)`

- The value of *has inverse* must be a relation type (value class).
  `HasInverse(x, y) → Isa(y, Relation Type)`

This is however not what we seek when talking about constraints on relations. The constraints on the relation types are not transferred to the relations of the type. What we seek is constraints on the relation types that constrain how the relations of that type can be used.

Because relation types in TrollCreek are represented as entities it should be possible to reuse some of the constraint mechanism from the constraints on the entities. A sketch of how this constraint can be included in the system must describe the mechanism both at a conceptual level, and at the implementational level.

Conceptual level

As described above, there already are constraints on the relation type frames. To make the constraints we want, we have to introduce another set of constraints on these frames. Frames inheriting from *Relation* will then have two sets of constraints.

1) Constraints on the relation frame, the concept of the relation
2) Constraints on instances of the relation

As discussed these two sets are different in nature and should therefore be kept separate.

As mentioned several times, inheritance is central in frame-based systems. The relation types are represented as frames to allow reasoning about relation types, and to better define the meaning of a relation. Relation types can then inherit properties from other relation types. Like constraint on entities, constraint on relation types should then be inheritable over *has subclass* and *has instance* relations.

Constraining from-frame and to-frame

A relation is a concept that holds pointers to two frames telling that they are related some way. When designing a constraint mechanism to put on relations it is natural to consider if both to- and from-frames should be constrainable. The constraints implemented so far constrains the to-frame of the relations on a given frame. As discussed in 4.3 being able to constrain the to-frame of a relation is desirable. In 2.1.3 it was also shown that at least one constraint type, value class, makes sense when constraining the from-frame of a relation. Cyc has a constraint called *makesSenseFor* that restricts what values that can fill the *from* role of a constraint [11]. This is a useful constraint that should be available in TrollCreek as well. The concept of *makesSenseFor* is in fact already introduced in Creek as a relation in the extensive top ontology mentioned in 5.1.1. When used, relations of this type tell the reader of the model that the use of the given relation is restricted. It is however left to the modeller to make sure that the restriction is satisfied. No evaluation mechanism does this.

Constraint types

What constraints should be included in this mechanism? In the implemented constraint system we have five constraint types: Cardinality, concrete value, value class, value range and general. Do all these make sense in this setting? At the first glance the answer is no.

Using cardinality to constraint the from-frame of a relation will for instance have the following meaning: *How many times the relation can be used in the entire model*. Constraining the cardinality of the to-frame of a relation is also troublesome in the Creek framework. The meaning of the constraint will be *how many frames in the entire model can be pointed to by the relation.* This is presumably not what users of the system want to model. This is however a rather strict interpretation of the constraint. A

more pragmatic solution will be to give the cardinality constraint the following meanings:

- On to-frame: For any from-frame there can be x to y instances of the relation pointing away from it.
- On from-frame: For any to-frame there can be x to y instances of the relation pointing to it.

Interpreted this way the cardinality constraint makes good sense defined on a relation type.

Above we mentioned the *makesSenseFor* constraint from Cyc. This is in fact a value class constraint for the from-frame of a relation. In TrollCreek a relation has a *from* and a *value* attribute. The latter is the same as what is called to-frame earlier in this text. As long as the to-frame of the relation is called its value, it is confusing to talk about *value* class constraints on the from-frame of a relation. However, the mechanism of this constraint is useful. It is easy to see that value class constraints on the to-frame or value of a relation is possible and valuable.

Concrete value constraints are very similar to value class constraints. They both constrain what frame that can fill a role. From what is said about value class constraints in the previous paragraph we can conclude that concrete value constraints should also be available on relations – on both sides.

Value range constraints are also related to the two just mentioned in that they all constrain what frames can fill a role. A range constraint on the to-frame of a constraint is clearly sensible. For example it is reasonable to constrain the value of the relation *age* to be greater than zero. But is it reasonable to use this constraint on the from-frame? This will be to constrain what numbers can be the from-frame of a relation. In fact, this will make sense. For instance if we are modelling a mathematical domain and want to show what numbers are dividend and divisor in an equation. To avoid dividing by zero we can constrain the from-frame of the relation *divisor of* to be greater than zero.

Finally we have the general constraint mechanism. This constraint shall be the means to realize all types of constraints that can not be expressed using the four just mentioned. Allowing to define such constraints on relation types do not introduce any new functionality to the system. Because the existing general constraints on entities gives an access point to the knowledge model and allows us to write anything, general constraints on relation types can already be written. The already existing mechanism is attached to entities. Although it is possible, it is no good solution to write relation constraints on an entity. To make the concept more understandable there should be a general constraint mechanism tied to relation types. Here the access point to the knowledge model can be the relation.

As mentioned a few times earlier in this text, all relations in Creek have an inverse relation. A constraint defined on the to-frame of a relation will then implicitly be defined on the from-frame of the inverse relation. By using a constraint saying that multiple inheritance is not allowed Figure 7.2 illustrates how constraints on relations also pass for the inverse relation.

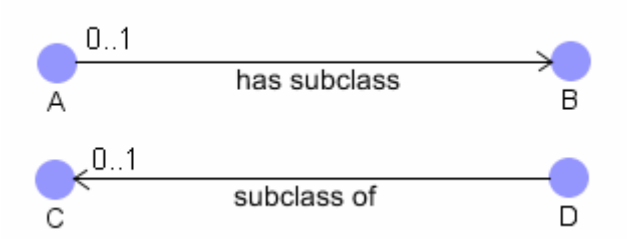


Figure 7.2 - Constraint: no multipple inheritance

Reviewing the five constraint types from entity constraints we have seen that all of them are sensible and useful when considering constraint on relations.

User interface
Although it is an implementational thing, we will now see how we can extend the user interface of the existing constraint mechanism to include constraints on relations. Because the user interface hides the internal implementation from the user, it may be said to be a part of the conceptual leve.

Recall the constraint panel in the frame view shown in Figure 6.2. In Figure 7.3 we suggest extending this panel with a tabbed pane with two tabs. The existing panel for constraints on entities is put as one tab and a new panel for constraints on relations is put on the other tab. When a frame not representing a relation type is selected the relation tab will be invisible or disabled. The new panel is similar to the entity constraint panel by having two lists of constraints and a section for editing and adding constraints. There are mainly two differences. First, we do not need the drop down list for choosing the relation to be constrained. Second, two radio buttons are added that allows the user to state whether the constraint shall constrain the from- or to-frame of the relation.

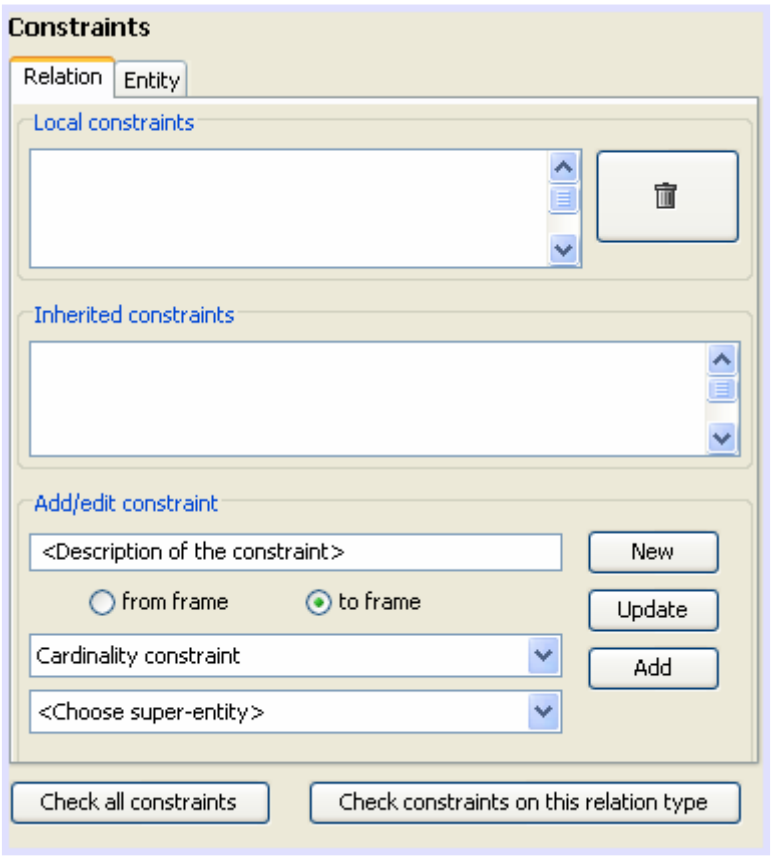


Figure 7.3 - GUI for relation constraints

Relation constraints should probably be visible in the frame view of the entities they are affecting. For instance, if there is a cardinality constraint on the relation *has parent*, all entities having this slot is affected by the constraint. When browsing an entity such constraints should be visible.

Implementational level

There are mainly two problems that have to be solved to implement these constrains into the system. First we have to find some way to represent and store the constraint attached to the relation type objects. Next the constraint checking mechanism must be updated to check constraints on relations. To make the constraint mechanism a whole, we will try to reuse the design of the entity constraint mechanism as long as possible.

Representation

We start with the representation. In the already implemented constraint mechanism we used a class structure similar to what is used for entities and relations to represent the constraints. Constraint objects for the different constraint types have methods for accessing the data and functionality of the constraints. They are interfaces to the constraint structure. The data and properties for the actual constraints are found in constraint data objects. The most central of constraint data's attributes are the constraint objects, objects inheriting from `ConstraintObject`. Recall from chapter 5 that these objects have a check method for evaluating the constraints. Figure 7.4 that are adapted from chapter 5 shows the structure of the constraint mechanism. As we see, the structure is identical to the one in Figure 5.6.

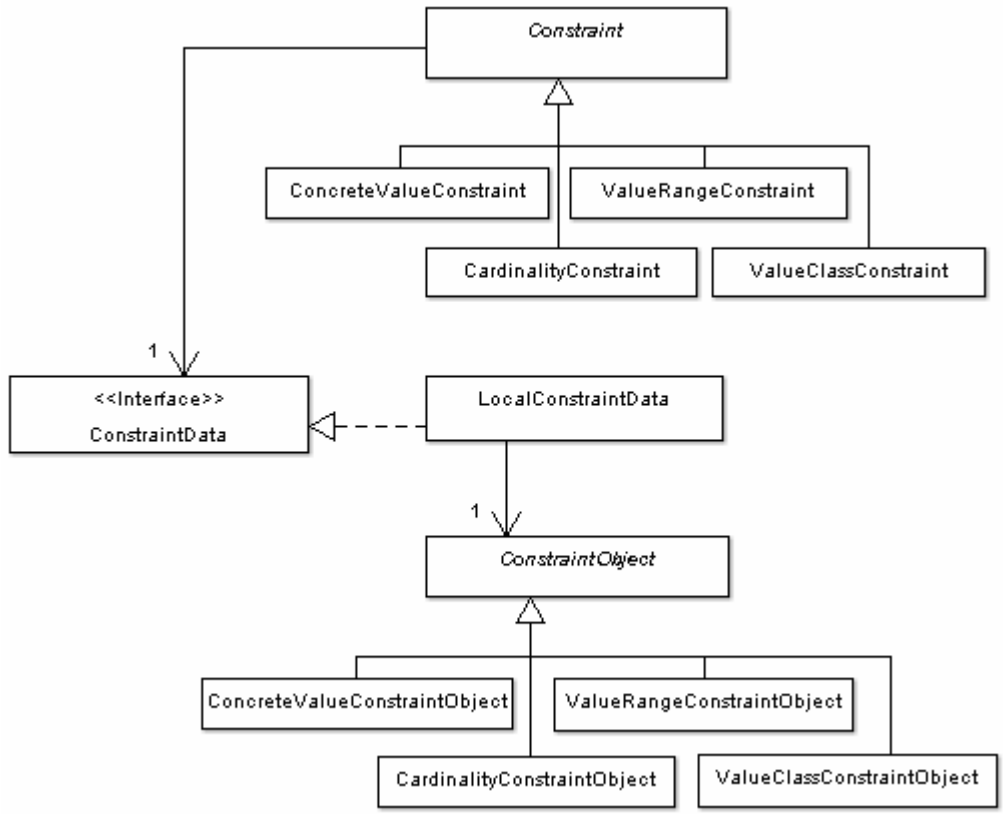


Figure 7.4 - Constraint structure

Make a new version with names distinguishing it from the one in chapter 5.

Relation constraints are however slightly different than entity constraints. We must therefore make new `Constraint` and `ConstraintData` classes for the new constraints. In entity constraints the constraint objects holds a reference to the relation they are constraining, and data about the particular constraint. As an example `CardinalityConstraintObject` have the following instance variables:

```
String rel;       // The name of the relation
int numberMin;    // Minimum bound
int numberMax;    // Maximum bound
```

When evaluating the constraint the entity to be checked is passed as an argument to the check method. Passing the entity as argument to the constraint makes it possible to check it with different entities. This is used to check the constraint with entities that inherits the constraint.

When making relation constraints the constraint is bound to the relation type and the constraint should be inheritable to other relation types. Consequently it should be possible to check the constraint with different relation types. We can then not have the relation type permanently stored in the constraint object. The relation type must be passed as argument to the check method. The instance variables of the constraint objects will then be the same as in the constraint objects of the entity constraint mechanism except for the relation reference that is removed.
In the entity constraints the check method has access to the knowledge model through the entity object. From the entity object the method could get relation objects and the to-frames of the relations. In the relation constraint mechanism the access to the knowledge model is through the relation type object. The algorithms of the check method will then be different than in the entity constraints. Table 7.4 shows how the

`check` method of a simplified value range constraint can be implemented.

```
int min;    // Minimum value
int max;    // Maximum value

public ConstraintCheckResult check(RelationType rel){
    boolean ok = true;

    //Get all instances of the relation type
    Relation[] relations = rel.getInstances();

    //For all relations
    for(int i = 0; i < relations.length; i++){
        Entity to = relations[i].getValue();
        int value = ((Number)(to.getEntityObject())).intValue();

        //Check if the value is ok
        if(value < min || value > max)
            ok = false;
    }

    if(ok)
        return new ConstraintCheckResult(true, "");
    else
        return new ConstraintCheckResult(false, "Value not ok");
}
```

Table 7.4 - Example of relation constraint

Having established the class hierarchy to represent the constraints we need to decide how these classes can be attached to the rest of the knowledge model. From 3.2.3 we remember that relation types are represented by objects of the class `RelationType` that inherits from `Entity` through `EntityType`. Objects inheriting from `Entity` have their properties and data stored in an `EntityData` object. The constraints on entities are stored in a list named `constraints` in this object. To be consistent with the existing implementation the relation constraints must be placed in the entity data object. The most obvious solution is to make a new list for the relation constraints. These constraints are in many ways different from the entity constraints: They have a slightly different function, they are checked slightly differently and handled by other GUI components. Putting the two constraint types in the same list will then be untidy.

When the constraint is represented in a similar fashion as the constraint on entities, the constraint checking mechanism can be similar too. The existing checking module has two check routines, one for checking a whole knowledge model, and one checking the constraints of a single entity. The first takes a knowledge model object as argument and the second takes an entity. Checking relation constraints should certainly be included in the two checking routines. In addition, primarily for testing purposes, it may be useful to be able to check all relation constraints for a given relation type.

7.3.2 Constraints as part of the ontology

In 5.1.1 we suggested that the constraints or at least the constraint types should be included in the top ontology of TrollCreek. This has not been implemented into the finished system. First, the time did not allow the development of a complete constraint system for TrollCreek. When deciding what to include in the implementation, making a runable version of parts of the system was prioritised. Second, it was not clear how this part should be integrated into the system. Making an ontology with a Constraint node, and nodes for the constraint types is trivial. Such an ontology is shown in Figure 7.5.

The difficult part is to tie the ontology together with the constraint mechanism, and making it mean something.

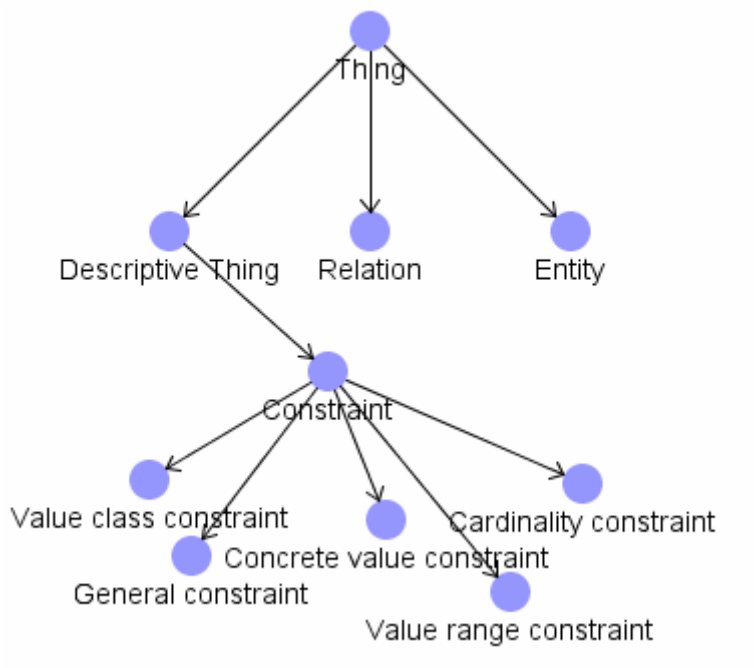


Figure 7.5 - Top ontology

7.3.3 Different approaches to constraint checking

In 5.2.1 we discussed at what time the constraint should be checked. In the implementation lazy evaluation is used. The constraints is checked when the user asks for it. This was seen as the most flexible solution, especially during the building phase of the knowledge model. It was however established that at being able to chose different checking approaches could be useful. Then the user can chose an on demand approach during the building of the model. After finishing the model a continuous checking can assure a consistent model during updates and further development. Such an option can easily be implemented due to a build in mechanism in TrollCreek that enables listening for model change events. A simple listener can then check the constraints of the entities involved in updates in the model.

8 Summary and further work

8.1 Summary

The goal of this work was to improve the knowledge modelling abilities of TrollCreek by offering the ability to state constraints on the knowledge model. In doing this we first discussed frame-based systems and constraint mechanisms in general and developed a simple framework for classifying constraints. The framework consists of cardinality constraint, concrete value constraint, value class constraint, value range constraint and general constraint. The four first constraint types cover most of the useful constraint mechanisms used in general purposed frame-based knowledge representations. These constraints restrict single values in the context of a frame and a slot. The last constraint type covers the need to define more complex constraints where the allowance of a value is dependent on the values of other slots.

With this framework in mind, the five frame systems, LispCreek, KM, Cyc, KL-ONE and Protégé was reviewed and discussed. Based on the learning from these systems we specified a constraint mechanism for TrollCreek. The specification is tightly tied to the five constraint types defined in the framework, but some details needed adaptations to work with the semantic net representation of the knowledge in TrollCreek.

A discussion on whether the constraints should be defined on the frames or on the slots concluded that the first of these was most flexible. The second is also useful, and the constraint mechanism in TrollCreek should have both options.

Because inheritance is central in frame systems, it was decided that constraints should be inheritable. This is also the norm in frame systems. TrollCreek have an augmented inheritance mechanism allowing properties to be inherited over all kinds of relations. It was however decided that constraints only should be inherited down *has subclass* and *has instance* relations.

Based on the specification, a constraint mechanism for TrollCreek was designed and implemented. The implementation was designed to fit in with the existing components of the system. One of the important aspects was the way TrollCreek allows its knowledge model to be stored different ways using an adapter design pattern.

The implemented constraint mechanism offers the knowledge modeller the ability to define cardinality constraints, concrete value constraints, value class constraints, value range constraints and general constraints on the knowledge model. General constraints are written in java as a `check` method that the system can evaluate to true or false. The constraints are defined on the entities, making it possible to state different constraint sets for the same slot type on different entities. The possibility do define constraints on the relations was not implemented due to time restrictions, but a sketch for how this could be solved was given. A simple user interface for defining and editing constraints was implemented.

Finally the constraint mechanism was evaluated. Using a small and uncomplicated test model the different constraint types was shown to give the expected result. Although this is no solid proof for the reliability of the mechanism, it demonstrates its functionality.

8.2 Further work

In section 7.3 we discussed parts of the constraint mechanism that is not realised in this project. The ability to define constraints on relation types constraining all constraints of the type is very useful when modelling large domains. This functionality can save the modeller from defining the same constraint on the same relation type for different entities.

Another important weakness of the implementation discussed in 7.3.2 7.3 is the failure to integrate the constraint mechanism in the frame ontology. For the current mechanism this is not a serious problem – the constraints works well. However, including the constraints or constraint types in the ontology can help the modeller to understand the meaning of each constraint type.

Partly related to this is the possibility to use the constraint mechanism in the inference process. Constraints can be used for example to represent incomplete knowledge. When used this way, constraints are not only constraints but also domain knowledge.

The KL-ONE system described in 2.5 has a graphical notation for constraints. As mentioned (3.1) the TrollCreek system has a graphical map view and a form based frame view as two different views of the knowledge model. The map view gives a good overview of the model while the form view shows the details of each frame. The user interface components developed for the constraints in this project is solely tied to the frame view. Making a graphical representation of the constraints in the map view should be considered as this will make it easier to “read” the model.

Although it is outside the scope of this project we will suggest using ideas from the general constraint mechanism to introduce procedural attachment functionality in TrollCreek. The underlying mechanisms can be used as they are, but a user interface that is not constraint-like must be developed. There are several situations where such a mechanism can be useful:

- Procedure as a value for slots
- Batch jobs
- Trigger routines

These aspects should be considered in the further development of the TrollCreek system.

8.3 End note

A digital version of this thesis and the attached file archive is accessible through the DAIM[5] system at http://daim.idi.ntnu.no. The file archive contains a standard TrollCreek distribution with source code, runnable jar-files and documentation in the Javadoc format. The documentation is customized with a description of what parts of the code that is developed in this project. See the `readme.htm` file for details.

The TrollCreek system without the additions from this project can be downloaded from the Creek home page at http://creek.idi.ntnu.no. The page also contains documentation on the system.

[5] System for digital archiving of Master’s theses at Department of Computer and Information Science, Norwegian University of Science and Technology.

References

[1] Agnar Aamodt: Knowledge-intensive Case-Based Reasoning and Sustained Learning. ECAI-90, Proceedings of the 9th European Conference on Artificial Intelligence, edited by Luigia Aiello, Stockholm, August, 6-10,1990.

[2] Agnar Aamodt: A Knowledge-Intensive, Integrated Approach to Problem Solving and Sustained Learning. Doctoral dissertation, University of Trondheim & Norwegian Institute of Technology (NTH), Division of Computer Science and Telematics, 1991.

[3] Agnar Aamodt: A Summary of the CreekL Knowledge Representation Language. Internal report, Norwegian University of Science and Technology, Department of Computer and Information Science. 1991.

[4] Agnar Aamodt, Enric Plaza: Case-based reasoning; Foundational issues, methodological variations, and system approaches. AI Communications, Vol.7, No.1, March 1994, pp. 39-59

[5] Agnar Aamodt: Knowledge-intensive case-based reasoning in Creek. Peter Funk, Pedro A. Gonzalez Calero (eds.), Advances in case-based reasoning, 7th European Conference, ECCBR 2004, Proceedings. Madrid, Spain, August/September 2004.

[6] R. Brachman, J. Schmolze. An overview of the KL-ONE Knowledge Representation System. Cognitive science 9, pp. 171-216, 1985

[7] Peter Clark & Bruce Porter: KM - The Knowledge Machine 2.0: Users Manual, http://www.cs.utexas.edu/users/mfkb/RKF/km.html. Last visited 1. June 2006.

[8] W. E. Grosso, H. Eriksson, R. W. Fergerson, J. H. Gennari, S. W. Tu, & M. A. Musen. Knowledge Modeling at the Millennium (The Design and Evolution of Protege-2000). In Proceedings ofthe Twelfth Workshop on Knowledge Acquisition, Modeling and Management (KAW99), Banff, Alberta, Canada, October 16-21, 1999.

[9] Bill Grosso: The Protégé Axiom Language: Overall Design Considerations, http://protege.stanford.edu/plugins/paltabs/OverallDesignConsiderations.zip link avaliable at: http://protege.stanford.edu/plugins/paltabs/pal-documentation/index.htm. Last visited 1. June 2006.

[10] Stéphane Hillion: DynamicJava - a Java source interpreter. http://koala.ilog.fr/djava/index.html. Last visited 1. June 2006.

[11] Doug Lenat, R. Guha, K. Pittman, D. Pratt, M Shepherd. CYC: Towards programs with common sense. Communications of the ACM, vol. 33, no. 8, 1990.

[12] Doug Lenat, R. Guha, Building large knowledge-based systems: representation and Inference in the Cyc Project, Addison-Wesley, 1990.

[13] Doug Lenat, CYC: A Large-Scale Investment in Knowledge Infrastructure. Communications of the ACM, vol. 38, no. 11, 1995.

[14] N. F. Noy, R. W. Fergerson, & M. A. Musen. The knowledge model of Protege-2000: Combining interoperability and flexibility. 2th International Conference on Knowledge Engineering and Knowledge Management (EKAW'2000), Juan-les-Pins, France. 2000.

[15] Peter D. Karp: The Design Space of Frame Knowledge Representation Systems, SRI AI Center Technical Note #520, 1993.

[16] Han Reichgelt: Knowledge Representation: An AI Perspective, Ablex Publishing Corporation, 1991.

[17] Frode Sørmo: Plausible Inheritance - Semantic Network Inference for Case-Based Reasoning. Master thesis at Norwegian University of Science and Technology, Department of Computer and Information Science, 2000.

[18] Cycorp, Inc. home page
http://www.cyc.com. Last visited 1. June 2006.

[19] ResearchCyc home page
http://research.cyc.com. Last visited 1. June 2006.

[20] OpenCyc home page
http://www.opencyc.org. Last visited 1. June 2006.

[21] The Mozilla Public License http://www.mozilla.org/MPL/.
Last visited 1. June 2006.

[22] Protégé-2000 users guide
http://protege.stanford.edu/publications/UserGuideA4.pdf.
Last visited 1. June 2006.

[23] PAL Documentation
http://protege.stanford.edu/plugins/paltabs/pal-documentation/index.htm
Last visited 1. June 2006.

[24] Creek home page
http://creek.idi.ntnu.no. Last visited 1. June 2006.

A Ontologies

Creek IsoPod model

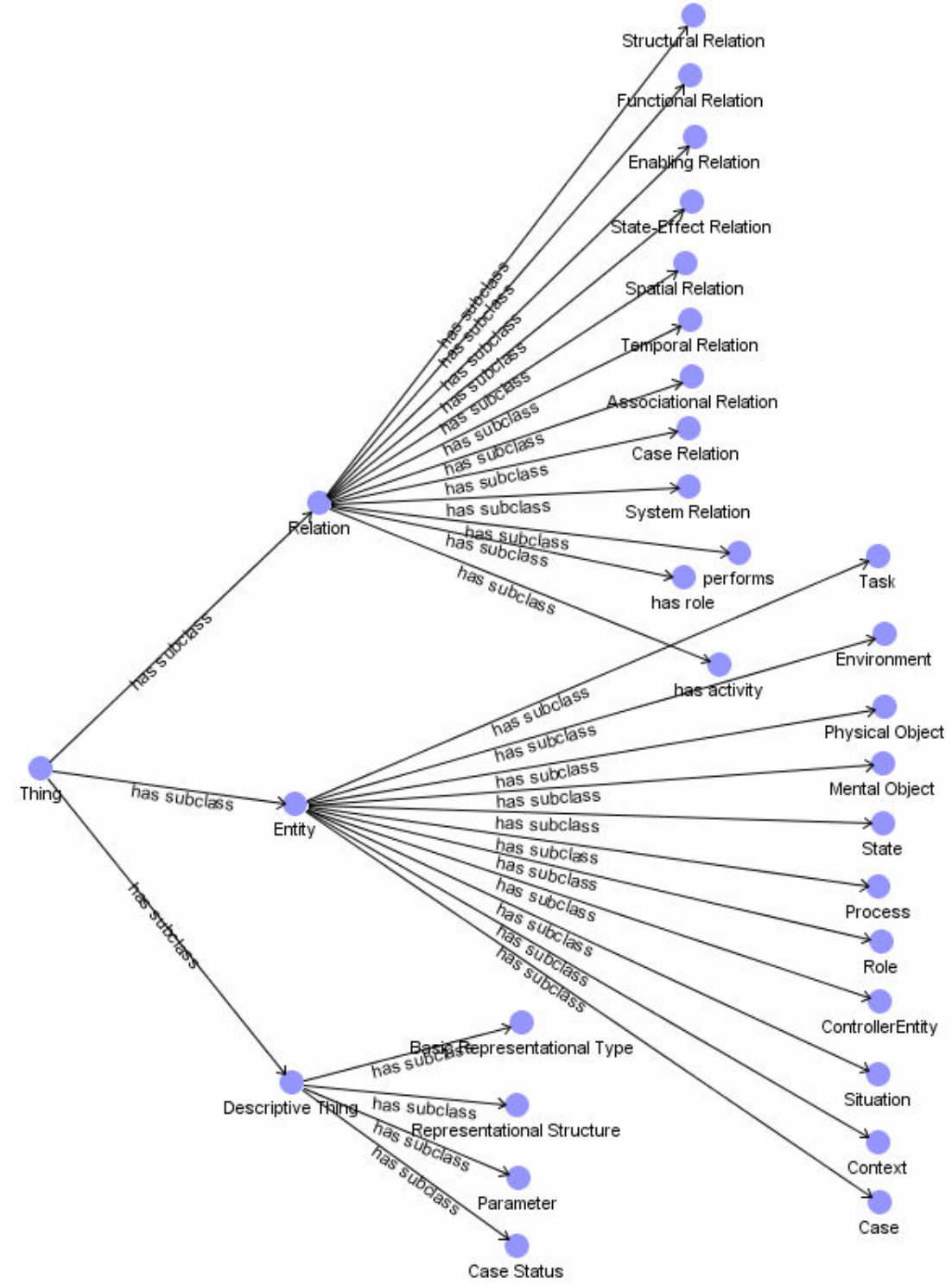


Figure 8.1 - Creek IsoPod model

The three highest levels of the IsoPod onltology

B Class diagrams

Constraint and its subclasses

ConstraintObjects