Coherent Description Framework Programmer's Manual Version 1(beta)

Terrance Swift
David S. Warren

Acknowledgements

The Coherent Description Framework, of which Cold Dead Fish is the open-source version, was developed with funding by Medicine Rules Inc, Medical Decision Logix Inc, and with major funding by XSB, Inc. Of the functionality included in Cold Dead Fish, most of the Type-0 interface, along with I/O routines, etc. was developed jointly by the authors; the Type-1 interface and reasoning routines were developed by Swift. Tanya Vidrevich, Bob Pokorny, and Harpreet Singh discovered many bugs during development, and made many valuable suggestions.

Contents

Chapter 1

Introduction

"If logic programmers developed sushi, they'd market it as cold dead fish".

Logic programming in its various guises: using Prolog with logical constraints, or using Answer Set Programming, can provide a useful mechanism for representing knowledge, particularly when a program requires default knowledge. However formal ontologies based on description logics have also received a great deal of attention as formalisms for knowledge representation. Description logics have a clear semantics as a subset of first-order logic in which determining consistency (and implication) of a set of sentences is decidable. Furthermore, the worst-case complexity of these problems is well-understood for various description logics. From a practical point of view, a user's intuitions about object-oriented programming are helpful when ontologies are first encountered, since information in ontologies consists of descriptions of classes, objects and relations. In addition, ontologies can be readily visualized and, in certain cases, manipulated by non-programmers using grapical interfaces (e.g. [Pro01], and many others). This has led to a profusion of systems based around description logics (see [MH03] for a review of some of these systems), and to standard representations that allow such systems to exchange knowledge, such as the recent OWL standard [SMVW02].

The *CDF* system allows various sorts of support for management of formal ontologies from within XSB. The full version of CDF is called the *Coherent Description System* which has been developed largely by XSB, Inc and has been heavily used in commercial software systems that extract information from free text, gather information from the world-wide web; and classify input strings according to given ontologies. Many of these applications generate code based on information in an ontology, with the result that CDF has formed the basis for model-driven commercial architectures. An open-source version of CDF is called *Cold Dead Fish* and contains many, though not all, of the features of the Coherent Description System. In this manual we describe all of CDF and note in passing which parts of it are open-source and which proprietary.

A high-level architecture of CDF is shown in Figure 1.1. CDF stores knowledge in a *CDF Instance* consisting of information in the form of Prolog facts (called extensional facts), Prolog rules (called intensional rules), or in various database-resident formats. Throughout the CDF architecture, information produced by evaluating intensional CDF rules is handled in the same manner as that produced by asserting extensional CDF facts. This includes query evaluation, consistency checking, update, and other routines. CDF intensional rules themselves are executed upon being invoked by a goal. Thus when intensional rules are written in Prolog they avoid the view-maintenance problems that arise when forward-chaining rules are updated; when tabling is used CDF provides predicates that allow tables to be abolished whenever a CDF instance changes. In addition, the goal orientation of the rules allows CDF to lazily obtain information from databases; and the various CDF database interfaces allow maintenance of ontologies that are too large for the virtual memory of a given machine.

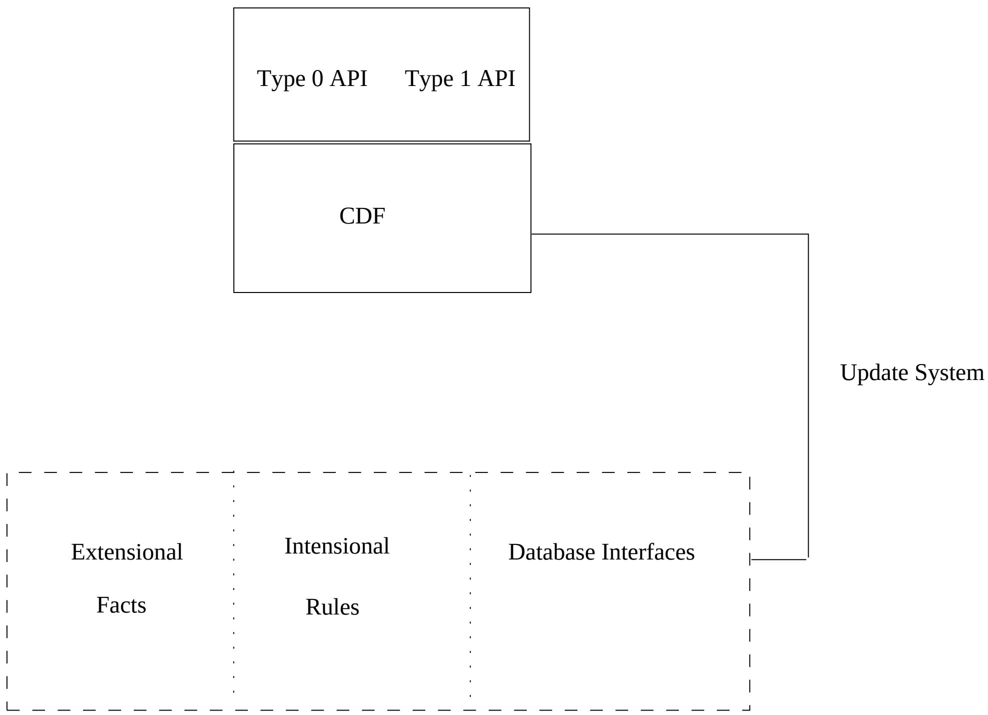

Figure 1.1: A High-Level Architecture of CDF and XJ

CDF instances can be classified either as Type-0 or a Type-1, each of which has its own interface. Type-0 instances are useful for storing large amounts of information; and consistency and implication in Type-0 instances is computable in polynomial time. Type-0 instances describe classes by existential and universal relations, qualified number restrictions, and relational hierarchies, but descriptions omit negation and disjunction. Type-0 instances also support a direct product construction for objects and classes. Information in Type-0 CDF instances is tightly coupled to XSB's query mechanism, and CDF ensures that only the most specific answers (according to a given inheritance hierarchy) are returned for any Type-0 query.

Type-1 instances extend Type-0 instances to describe classes using negation and disjunction, and thus permit descriptions that are equivalent to an expressive description logic. In fact, a Type-1 CDF instance can be seen as a knowledge base in which various classes are described via *class expressions*, which correspond to formulas in description logics. Reasoning in Type-1 instances is done via the CDF theorem-prover. Using the Type-1 API, users may ask whether a given class or object is consistent, whether a given class expression is consistent with a class or object; or whether a given class expression is entailed by a given class or object. The problems of determining consistency or entailment of a Type-1 class expression have a high degree of complexity. To solve these problems the CDF theorem prover uses several heuristics, but a determined (or unlucky) user can always find class expressions that require a large amount of time to check.

Of course, ontology management systems require many features in addition to reasoning and representation features [MPS03]. We mention some of these features.

- *A Semantic Checking System.* CDF has various mechanisms for ensuring consistency of objects and classes both at the Type-0 or Type-1 level. Various levels of consistency can be checked during various operations on the CDF instance.
- *A Component System.* Reusability of ontologies is supported by the *component* structure of CDF. An ontology component may be maintained by separate users or organizations in different locations and assembled in various ways by applications.
- *A Concurrency System.* (Non open-source) Based on the component structure, the *concurrency* mechanism for CDF allows users to update their own CDF instances and to periodically update a common store. Naturally, the various mechanisms in CDF for ensuring consistency that are vital to ensuring coherency when users update their systems concurrently.
- *Database Interfaces.* (Non open-souce) CDF supports various interfaces to databases so that CDF facts can be stored in a database or mapped to database tables.

Based on these features, CDF can support user interfaces in a number of ways. One of the most convenient is to use a XSB/Java interface such as InterProlog [Cal01] or JAXSB (see `http://xsb.sourceforge.net`) and then write a user interface in Swing or some other Java Graphics library. One of the easiest ways to do this is to make use of the *XJ system* which allows Swing Gui objects to be represented as Prolog terms (the XJ system is non open-source) From a systems perspective, a graphical interface is then written XJ library Swing widgets or specialized XJ-CDF Swing widgets. CDF per-se has the following graphical packages and applications.

- *An XJ Caching System.* Adds and deletes to CDF are extended with a notification mechanism so that Java Swing objects (created with XJ, XSB's graphics system) reflect the state of CDF even when it dynamically changes.
- *A Visual Editor.* Finally, CDF supports a graphical editor that allows users both to visualize an ontology and to perform the functions mentioned so far.

Extensional facts, intensional rules, updates, the Type-0 and Type-1 interfaces, consistency checking predicates and the full component system are available as an open-source package for XSB. Other features, concurrency mechanisms, specialized database interfaces, XJ support and the editor are not open-source. The open-source code can be obtained via `xsb.sourceforge.net`. Inquiries about the full Coherent Description System should be made to `ode@xsb.com`.

Part I

A (somewhat) Formal Introduction to CDF

Chapter 2

The Meaning of Type-0 CDF Instances

Facts in both Type-0 and Type-1 CDF instances are closely related to class expressions in description logics. However, because CDF often stores class expressions as Prolog facts in an unconventional way, and because description logics may not be familiar to a logic programming audience, we present here a somewhat formal introduction to how CDF represents knowledge. Users without a mathematical background can ignore the various axioms and formal definitions that are presented in this chapter. Our approach is to introduce a semantics of CDF based on a translation of a *CDF instance* into a set of first-order logic sentences that constitute an *Ontology Theory* whose models are the models of a CDF instance. For simplicity of presentation the description of CDF instances in this section omits certain details about components, extensional facts and intensional rules, and other topics that will be introduced in later chapters.

We illustrate aspects of CDF by means of an example drawn from electronic commerce. Health care organizations, such as hospitals, clinics, etc., have difficulties in buying disposable medical devices such as sutures, bandages, gloves, and so on. The difficulty arises from the fact that these devices may be quite specialized, for intance when they are used in surgery. At the same time, since these devices are disposable, they may need to be purchased frequently. We consider concretly the class of *absorbable sutures*, which are used for stitching and securing tissues, and which can be absorbed by the human body. Information below is adapted from he U.S. Defence Logistics Information Service `http://www.dlis.mil`, from the Universal Standard Products and Services Classification [UNS02], as well as from websites of various commercial medical supply companies.

2.1 Type-0 CDF Instances

We begin with the syntax of Type-0 instances [1]:

Definition 2.1.1 [Type-0 Instances: Semantic Level] A *Type-0 CDF instance* is a finite set of

[1] The syntax for identfiers differs in the actual CDF implementation. See Section 4.1.

ground facts for the predicates `isa/2`, `hasAttr/3`, `allAttr/3`, `classHasAttr/3`, `minAttr/4`, and `maxAttr/4`. An *identifier* is either a constant or a term. The arguments of these predicates are *concrete identifiers*, where a term T is an identifier iff T has the functor symbol `cid/1`, `oid/1`, `rid/1`, or `crid/1` whose argument is either

1. a constant; or
2. a term $f(I_1, \ldots, I_n)$ where $I_1, \ldots, I_n$ are identifiers.

In the first case, an identifier is called *atomic*; in the second it is called a *product identifier*. □

Despite the simple syntax of Type-0 CDF instances, their semantics differs from the usual semantics assigned to facts in Prolog. Identifiers identify sets of objects, or binary relations between objects. Furthermore, the facts of a Type-0 CDF instance can implicitly denote inheritance of various relationships among classes and objects, as well as inheritance constraints about what relationships are allowed.

2.1.1 Simple Taxonomies in CDF

Example 2.1.1 The following CDF instance illustrates a fragment of a taxonomy for medical equipment.

```
isa(cid(medicalEquipment),cid('CDF Classes'))
   isa(cid(woundCareProducts),cid(medicalEquipment))
         isa(cid(suturesAndRelatedProducts),cid(woundCareProducts))
         isa(cid(sutures),cid(suturesAndRelatedProducts))
            isa(cid(absorbableSutures),cid(sutures))
            isa(cid(nonAbsorbableSutures),cid(sutures))

         isa(oid(sutureU245H),cid(absorbableSutures))
               isa(oid(suture547466),cid(sutures))
```

In CDF, sets of objects are termed *classes* to stress the informality of its sets from the perspective of set theory, and class identifiers have the functor `cid/1`. One can read the fact

```
isa(cid(nonAbsorbableSutures),cid(sutures))
```

as "all elements in the class `cid(nonAbsorbableSutures)` are also in the class `cid(sutures)`" — in other words, that `cid(nonAbsorbableSutures)` is a subclass of `cid(sutures)`. Object identifiers have the functor `oid/1`, and denote classes with cardinality 1, or *singleton classes*. The fact

```
isa(oid(sutureU245H),cid(absorbableSutures))
```

can be read as "the element of the singleton class `oid(sutureU245H)` is in the class `cid(absorbableSutures)`". Note that `cid(absorbableSutures)` is (potentially) more specific than the class `cid(sutures)`, to which `oid(suture547466)` belongs. The class `cid('CDF Classes')` is taken to contain all objects in the domain of discourse. All identifiers in this simple taxonomy are atomic.

The decision of whether to denote an entity as an object or as a class depends on the use of a given CDF instance. Here, a given part number can specify a number of physical parts, but the physical parts are taken to be identical for the purposes of this instance. However, if we were constructing a CDF instance for warehouse management, the above objects might be better represented as classes, and the physical objects represented as CDF objects.

Implicit in the above example is the fact that we use the term *object identifiers* or *objects* to denote singleton classes, class identifiers to denote all classes (including singleton classes). Elements cannot be denoted directly by CDF facts, but only through singleton classes that contain them (and are isomorphic to them). At this point, we can begin defining the semantics of Type-0 CDF instances.

Definition 2.1.2 An *ontology language* is a first-order language with equality containing the predicates: *isClass/1, isElt/1, isRel/1, isCrel/1, elt/2, rel/3, and crel/3*, and a countable set of constants. An *ontology structure* is a structure defined over an ontology language. An *ontology theory* is a set of first-order sentences formed over an ontology language that includes a set of *core axioms*, defined below, along with the defined predicate:

$$isObj(X) =_{def} isClass(X) \wedge ((elt(E_1, X) \wedge elt(E_2, X)) \Rightarrow E_1 = E_2)$$

If $\mathcal{T}$ is an ontology theory formed over an ontology language $\mathcal{L}$, an ontology structure S over $\mathcal{L}$ is a model of $\mathcal{T}$ if every sentence of $\mathcal{T}$ is satisfied in S. □

By convention we assume that the variables in an ontology language are indexed by the set of natural numbers. In this paper we will restrict our attention to ontology languages whose constant and function symbols are identifiers as described in Definition 2.1.1. Informally $isClass/1$ indicates that an identifier I is a class name or *class identifier*; $isElt/1$ indicates that an identifier I is an element of a class; $isRel/1$ that I is a *relation identifier*; and $isCrel/1$ that I is a *class-relation identifier*; and $isObj/1$ that I is an *object identifier*. We sometimes call these 5 predicates *sorting predicates*. $elt(E, C)$ indicates that an element O is a member of class identifier C; $rel(O_1, R, O_2)$ indicates that an element E_1 has a R relation to an element E_2; and $crel(C_1, R, E)$ indicates that the class identifier C_1 has a R relation to an object identifier E.

The first core axiom ensures that objects, classes, relations and class-relations all have distinct identifiers within an ontology language.

Axiom 1 (Distinct Identifiers)

$$\begin{array}{l}
\neg\exists Id[isClass(Id) \wedge (isElt(Id) \vee isRel(Id) \vee isCrel(Id))]\wedge \\
\neg\exists Id[isObj(Id) \wedge (isElt(Id) \vee isCrel(Id))]\wedge \\
\neg\exists Id[isElt(Id) \wedge isCrel(Id)]
\end{array}$$

$isClass/1$, $isElt/1$, $isRel/1$, and $isCrel/1$ provide an effective sorting that extends to all predicates, as the next axiom indicates.

Axiom 2 (Predicate Sorts)

$$\begin{array}{l}(\forall X,Y)[elt(X,Y) \Rightarrow (isElt(X) \wedge isClass(Y))] \wedge \\ (\forall X,Y,Z)[rel(X,Y,Z) \Rightarrow (isElt(X) \wedge isRel(Y) \wedge isElt(Z))] \wedge \\ (\forall X,Y,Z)[crel(X,Y,Z) \Rightarrow (isClass(X) \wedge isRel(Y) \wedge isElt(Z))]\end{array}$$

The following definition of $IdSort$ relates the functors of identifiers in a Type-0 CDF instance to their sort in an ontology theory. It will be used in the various instance axioms to enforce proper sorting of product identifiers.

Definition 2.1.3 **[IdSort]** Let I is be an identifier. Then *IdSort(I)* is equal to *isClass(I)* if the main functor symbol of I is `cid/1`; *isObj(I)* if the main functor symbol of I is `oid/1`; *isRel(I)* if the main functor symbol of I is `rid/1`; and *isCrel(I)* if the main functor symbol of I is `crid/1`. □

From Definitions 2.1.3 and 2.1.2, it is easy to see that for any object identifier O, $IdSort(O) = isObj(O)$, and $isObj(O) \Rightarrow isClass(O)$.

Translation Rule 1 (Translation of `isa/2`) For each fact of the form $\texttt{isa(Id}_1\texttt{,Id}_2\texttt{)}$ add the axiom

$$\begin{array}{l}IdSort(Id_1) \wedge IdSort(Id_2) \wedge \\ \quad (((isClass(Id_1) \wedge isClass(Id_2)) \wedge (\forall X)[elt(X,Id_1) \Rightarrow elt(X,Id_2)]) \vee \\ \quad ((isRel(Id_1) \wedge isRel(Id_2)) \wedge (\forall X,Y)[rel(X,Id_1,Y) \Rightarrow rel(X,Id_2,Y)]) \vee \\ \quad ((isCrel(Id_1) \wedge isCrel(Id_2)) \wedge (\forall X,Y)[crel(X,Id_1,Y) \Rightarrow crel(X,Id_2,Y)]))\end{array}$$

denoted as $\texttt{isa(Id}_1\texttt{,Id}_2\texttt{)}^{\mathcal{I}}$.

Note that the reflexive and transitive closure of `isa/2` is an immediate consequence of its translation rule. The next axiom is technical. It is important for the semantics of relations that each class have at least one member.

Axiom 3 (Non-Empty Classes)

$$(\forall X)[isClass(X) \Rightarrow (\exists Y)[elt(Y,X)]]$$

The last core axiom for these predicates ensures is that each class is a subclass of `cid('CDF Classes')`.

Axiom 4 (Domain Containment)

$$(\forall X)[isElt(X) \Rightarrow elt(X, cid('CDF\ Classes'))]$$

2.1.2 General Relations between Objects in Classes

Example 2.1.2 The class `cid(sutures)` can be further defined by its relations to other classes. For instance, an object in `cid(sutures)` may have a designation of its needle design indicating whether it is to be used for abdominal surgeries, thoracic surgeries, or other purposes. This information is indicated by the following CDF facts:

```
isa(rid(hasNeedleDesign),rid('CDF Relations'))

isa(cid(domainTypes),cid('CDF Classes'))
  isa(cid(needleDesignTypes),cid(domainTypes))
    isa(cid(Abdominal),cid(needleDesign))
    isa(cid(Abscisson),cid(needleDesign))
    isa(cid('Adson Dura'),cid(needleDesign))
% 126 other values..

allAttr(cid(sutures),rid(hasNeedleDesign),cid(needleDesign))
```

The `allAttr/3` fact above can be read as "if any object in `cid(absorbableSutures)` has a `rid(hasNeedleDesign)` relation, it must be to an object in the class `cid(needleDesign)`". That `hasNeedleDesign` is a relation is indicated by clothing it in the functor `rid/1`. This relation is an immediate subclass of all `CDF Relations` which in turn is taken to represent the universal binary relation over the domain of discourse. Thus the `allAttr/3` fact effectively types the range of `rid(hasNeedleDesign)` relations, stemming from objects in the class `cid(absorbableSutures)`, but it does not indicate the existence of such a relationship. From a user's point of view, the `rid(hasNeedleDesign)` relation can be thought of as an optional attribute for a given `cid(absorbableSutures)` object. Sample values for `cid(hasNeedleDesignTypes)` are also given.

`allAttr/3` provides a simple but powerful mechanism for inheritance of typing among CDF classes:

Translation Rule 2 (Translation of `allAttr/3`) For each fact of the form `allAttr(`Id_1`,Rid,`Id_2`)` add the instance axiom:

$$
\begin{array}{l}
IdSort(Id_1) \wedge IdSort(Rid) \wedge IdSort(Id_2) \wedge \\
\quad IsClass(Id_1) \wedge IsRel(Rid) \wedge IsClass(Id_2) \wedge \\
\quad (\forall X, Y)[(elt(X, Id_1) \wedge rel(X, Rid, Y)) \Rightarrow elt(Y, Id_2)]
\end{array}
$$

denoted as `allAttr(`Id_1`,Rid,`Id_2`)`$^{\mathcal{I}}$.

Example 2.1.3 While `allAttr/3` indicates a typing for relations, it does not indicate that a relation must exist for elements of a class. This statement is made by `hasAttr/3`. The relation `rid(hasPointStyle)` for the class `cid(absorbableSutures)` is taken to be required in this schema. The facts

```
allAttr(cid(absorbableSutures),rid(hasPointStyle),cid(pointStyle))
hasAttr(cid(absorbableSutures),rid(hasPointStyle),cid(pointStyle))
```

indicate not only the range of such relationships, but that such a relationship must exist. Indeed, the `hasAttr/3` fact can be read as "all objects in the class `cid(absorbableSutures)` have a relation `rid(hasPointStyle)` to an object in the class `cid(pointStyle)`". The facts below also give information about the `rid(hasPointStyle)` relation.

```
isa(cid(pointStyle),cid(domainTypes))
isa(cid(regularCuttingEdge),cid(pointStyle))
isa(cid(reverseCuttingEdge),cid(pointStyle))
isa(cid(scalpelPoint),cid(pointStyle))
% 10 other values.

hasAttr(oid(sutureU245H),rid(hasPointStyle),cid(regularCuttingEdge))
```

The last of the above facts can be read as "the object `oid(sutureU245H)` has a `rid(hasPointStyle)` relation to an object in the class `cid(pointStyle)`".

Not surprisingly, the definition of `hasAttr/3` bears some similarity to that of `allAttr/3`.

Translation Rule 3 (Translation of `hasAttr/3`) For each fact of the form $\texttt{hasAttr(Id}_1\texttt{,Rid,Id}_2\texttt{)}$ add the instance axiom:

$$\begin{array}{l} IdSort(Id_1) \wedge IdSort(Id_2) \wedge IdSort(Id_2) \wedge \\ \quad IsClass(Id_1) \wedge IsRel(Rid) \wedge IsClass(Id_2) \wedge \\ \qquad (\forall X)[elt(X, Id_1) \Rightarrow \exists Y[rel(X, Rid, Y) \wedge elt(Y, Id_2)]] \end{array}$$

denoted as $\texttt{hasAttr(Id}_1\texttt{,Rid,Id}_2\texttt{)}^{\mathcal{I}}$.

We next turn to relational axioms that indicate the cardinality of various relations.

Example 2.1.4 For our purposes, an object in `cid(absorbableSutures)` can be thought of as consisting of a needle and a thread [2]. This is represented by the facts:

```
allAttr(cid(absorbableSutures),rid(hasImmedPart),cid(absSutPart))
hasAttr(cid(absorbableSutures),rid(hasImmedPart),cid(absSutNeedle))
hasAttr(cid(absorbableSutures),rid(hasImmedPart),cid(absSutThread))

isa(cid(absSutPart),cid(suturesAndRelatedProducts))
  isa(cid(absSutNeedle),cid(absSutPart))
  isa(cid(absSutSuture),cid(absSutPart))
```

A needle for an absorbable suture is typically made of a different material than the thread to which the needle is attached. Each of these materials may be important in choosing an absorbable suture, and each of these materials are unique. The facts

[2] The thread is often called a suture. We are assuming for purposes of illustration that all sutures are — in suture-speak — "armed".

```
hasAttr(cid(absSutPart),rid(hasMaterial),cid(absSutMaterial))
maxAttr(cid(absSutPart),rid(hasMaterial),cid(absSutMaterial),1)

isa(cid(material),cid(domainTypes))
isa(cid(absSutMaterial),cid(material)
isa(cid(gut),cid(absSutMaterial))
isa(cid(polyglyconate),cid(absSutMaterial))
isa(cid(polyglyconicAcid),cid(absSutMaterial))
```

indicate that each `cid(absSutPart)` has a unique material. The `maxAttr/4` fact can be read as "Each object in the class `cid(absSutPart)` has at most 1 `rid(hasMaterial)` relation to an object in the class `cid(absSutMaterial)`".

In order to define the semantics of `maxAttr/4`, let $\exists^{\leq n} X_m[\phi(X,Z)]$ be defined as an abbreviation for the formula

$$\exists x_m, ..., \exists x_{m+n}[\bigwedge_{m \leq i \leq m+n} \phi(x_i, \overline{z}) \Rightarrow \bigvee_{m \leq i < j \leq m+n} x_i = x_j]$$

i.e., for the formula indicating that there are at most N non-equal elements satisfying $\phi(x,z)$. The abbreviation $\exists^{\geq N}$ is defined similarly to indicate that a formula is satisfied by at least N non-equal elements.

Translation Rule 4 (Translation of `maxAttr/4`) For each fact of the form `maxAttr(Id`$_1$`,Rid,Id`$_2$`,N)` add the instance axiom:

$$\begin{array}{l} IdSort(Id_1) \wedge IdSort(Id_2) \wedge IdSort(Id_2) \wedge \\ \quad IsClass(Id_1) \wedge IsRel(Rid) \wedge (IsClass(Id_2) \wedge \\ \qquad (\forall X)[elt(X, Id_1) \Rightarrow \exists^{\leq N} Y[rel(X, Rid, Y) \wedge elt(Y, Id_2)]] \end{array}$$

denoted as `maxAttr(Id`$_1$`,Rid,Id`$_2$`,N)`$^{\mathcal{I}}$.

A corresponding predicate, `minAttr/4` is used to indicate a minimality restriction on a relation. `minAttr/4` is defined similarly to `maxAttr/4`, but using $\exists^{\geq N}$ rather than $\exists^{\leq N}$. Indeed, the predicate

```
hasAttr(cid(absSutPart),rid(hasMaterial),cid(absSutMaterial)).
```

could be replaced by the predicate

```
minAttr(cid(absSutPart),rid(hasMaterial),cid(absSutMaterial),1).
```

2.1.3 Class Relations

Each of the predicates discussed so far are inheritable in their first argument. For instance, the fact

```
hasAttr(cid(absSutPart),rid(hasMaterial),cid(absSutMaterial)).
```

implies that every subclass of `cid(absSutures)` will have a material in the class `cid(absSutMaterial)`. However, classes may have relations that do *not* hold for their subclasses or members. For instance, a finite set may have a given cardinality, but its proper subsets will have a different cardinality. Such relations are termed *class relations*.

Example 2.1.5 A practical example of a class relation comes from an application that may be called part equivalency matching. In this application, the possible attributes for a class of parts are given various weights. Two parts match if the sum of the weights of their attributes that match are above a given threshold. The weighting for the `cid(pointStyle)` of sutures might be given as:

```
isa(cid(pointStyleWeight),cid('CDF Classes'))
   isa(cid(highWeight),cid(pointStyleWeight))
   isa(cid(lowWeight),cid(pointStyleWeight))

classHasAttr(cid(sutures),crid(pointStyleWeight),cid(highWeight))
```

The `classHasAttr/3` fact can be read as “the class `cid(sutures)` has a `crid(pointStyleWeight)` relation to an object in `cid(highWeight)`. Matching weights are made non-inheritable via `classHasAttr/3` because a weight may depend on a given classification of a part. For instance if a part were classified as a `cid(nonAbsorbableSuture)`, its `cid(pointStyle)` might weigh less (or more) for determining whether two sutures are equivalent.

Translation Rule 5 (Translation of `classHasAttr/3`) For each fact of the form `classHasAttr(`$\mathtt{Id_1}$`,CRid,`$\mathtt{Id_2}$ add the instance axiom:

$$
\begin{aligned}
&IdSort(Id_1) \wedge IdSort(CRid) \wedge IdSort(Id_2) \wedge \\
&\quad IsClass(Id_1) \wedge IsCrel(CRid) \wedge isClass(Id_2) \wedge \\
&\qquad (\exists X)[elt(X, Id_2) \wedge crel(Id_1, CRid, X)]
\end{aligned}
$$

denoted as `classHasAttr(`$\mathtt{Id_1}$`,Rid,`$\mathtt{Id_2}$`)`$^{\mathcal{I}}$.

2.2 Product Classes

The above predicates allow the definition of various named binary relations between classes. However, binary definitions can sometimes be inconvenient to use. For instance, in the part equivalency matching example, (Example 2.1.5), it may be desirable to make explicit the weight of the match as an indication of the strength of the match. The weight could be made explicit by a series of definitions

```
allAttr(cid(dlaPart),rid(suturesRusMatch_low),cid(suturesRusPart))
:
allAttr(cid(dlaPart),rid(suturesRusMatch_high),cid(suturesRusPart))
```

indicting that a given part has a match of weight *low* through *high*. However, for a scale with a large number of values, defining matches in this way is time-consuming and prone to errors. To address this, we first define a new class `cid(matchScale)` containing as subclasses the various match levels. We then combine `cid(matchScale)` with the class `cid(suturesRusPart)` in a product with a *product identifier*, as in the following fact

```
allAttr(cid(dlaPart),rid(suturesRusMatch),
          cid(partMach(cid(suturesRusPart),cid(matchScale))))
```

which indicates that a `cid(dlaPart)` can have a `cid(suturesRusMatch)` to an object in the `partMatch/2` class, which has both a `cid(suturesRusPart)` component and a `cid(matchScale)` component.

We capture the intuition behind product classes through the following axiom schemas. The first indicates that product identifiers are constructed from *constituent identifiers* of the same sort.

Axiom 5 (Downward Closure)
For each product identifier $cid(f(x_1,\ldots,x_n))$, $oid(f((x_1,\ldots,x_n))$, $rid(f(x_1,\ldots,x_n))$, *and* $crid(f((x_1,\ldots,x_n))$ *the following axiom is added,*

$$
\begin{aligned}
&isClass(cid(f(x_1,\ldots,x_n))) \Rightarrow isClass(x_1) \wedge \ldots \wedge isClass(x_n)\\
&isObj(oid(f(x_1,\ldots,x_n))) \Rightarrow isObj(x_1) \wedge \ldots \wedge isObj(x_n)\\
&isRel(rid(f(x_1,\ldots,x_n))) \Rightarrow isRel(x_1) \wedge \ldots \wedge isRel(x_n)\\
&isCrel(crid(f(x_1,\ldots,x_n))) \Rightarrow isCrel(x_1) \wedge \ldots \wedge isCrel(x_n)
\end{aligned}
$$

With this axiom, along with the use of $IdType/1$ in the various instance axioms, we will sometimes refer to a CDF identifier I as a class identifier if its outer functor is `cid/1`, an object identifier if its outer functor is `oid/1` etc. The next axiom associates product classes with the objects they contain.

Axiom 6 (Implicit Subclassing)

1. *For each product identifier* $cid(f(x_1,\ldots,x_n))$ *or* $oid(f(x_1,...,x_n))$, *the following axioms are added for* $x_i, 1 \leq i \leq n$:

$$(\forall E)[(elt(E,y_i) \Rightarrow elt(E,x_i)) \Rightarrow (\forall E')[elt(E',f(x_1,...x_n))[x_i/y_i] \Rightarrow elt(E',f(x_1,...x_n))]]$$

2. *For each product identifier* $rid(f(x_1,\ldots,x_n))$ *the following axioms are added for* $x_i, 1 \leq i \leq n$:

$$
\begin{aligned}
&(\forall E_1,E_2)[(rel(E_1,y_i,E_2) \Rightarrow rel(E_1,x_i,E_2)) \Rightarrow\\
&\quad (\forall E_1',E_2')[rel(E_1',f(x_1,...x_n),E_2')[x_i/y_i] \Rightarrow rel(E_1',f(x_1,...x_n),E_2')]]
\end{aligned}
$$

Example 2.2.1 Axiom 5 simply states that identifier types cannot be mixed within a product identifier. For instance, if `oid(matchLevelN)` is an object in the `cid(matchScale)`, then the identifier

```
cid(partMatch(cid(suturesRusMatch),oid(matchLevelN)))
```

is improperly formed. On the other hand, if `oid(sutureU245H)` is in the class `cid(suturesRusPart)`, then the identifier `oid(partMatch(oid(sutureU245H),oid(matchLevelN)))` is a product identifier that is also an object identifier.

Axiom 6 also means that the inheritance relation of a product class is partly determined by the inheritance relation of its constituent elements.

2.3 Models of Type-0 Instances

Type-0 instances have been designed so that checking their consistency — whether they have a model — is easily done. Models of Type-0 instances are discussed in [SW03], here we review some of the main points without proofs. We begin by defining a model of a Type-0 instance.

Definition 2.3.1 Let $\mathcal{O}$ be a CDF instance. We define as $\mathcal{L}_{\mathcal{O}}$, the ontology language whose functions and constant symbols are restricted to the identifiers in $\mathcal{O}$. $TH(\mathcal{O})$ is the ontology theory over $\mathcal{L}_{\mathcal{O}}$ whose non-logical axioms consist of the core axioms and the instance axiom for each fact in $\mathcal{O}$. We say that an ontology structure $\mathcal{M}$ is a model of $\mathcal{O}$ if it is a model of $TH(\mathcal{O})$.

If $\mathcal{O}$ is Type-0 it is *well-sorted* if each instance axiom cojoined with Core Axioms 1 (Distinct Identifiers) and Axiom Schema 5 (Downward Closure) is consistent. □

Intuitively, a CDF fact is well-sorted if its arguments have class, relation, class relation identifiers, etc. in the right place. There is no room for contradiction in a well-sorted Type-0 instance $\mathcal{O}$ unless it contains `minAttr/4` and `maxAttr/4` facts. If it does, inconsistency can arise if $\texttt{minAttr(C}_1\texttt{,R,C}_2\texttt{,M)}^{\mathcal{I}}$ holds, $\texttt{maxAttr(C}_1\texttt{,R,C}_2\texttt{,N)}^{\mathcal{I}}$ holds, and $M > N$. Thus, consistency of Type-0 instances is easy to check, and can in fact be done in polynomial time.

However, CDF is a logic programming system, so it is important to know whether an instance $\mathcal{O}$ over a language $\mathcal{L}_{\mathcal{O}}$ has a *Herbrand* model [Llo84] — essentially one in which the universe $\mathcal{U}$ of the model are the terms of $\mathcal{L}_{\mathcal{O}}$, and where the mapping of terms from $\mathcal{L}_{\mathcal{O}}$ to $\mathcal{U}$ is the identity mapping. Herbrand models are the basis of Prolog semantics, so that if CDF instances have Herbrand models, they can be easily implemented in Prolog without resorting to a constraint system or to meta-interpretation. Care has been taken to ensure that any Type-0 CDF instance does in fact have a Herbrand model, so that the Type-0 interface in Chapter 4 can be supported [3].

2.4 Inheritance

Example 2.1.3 indicates that each object in the class of `cid(absorbableSutures)` has a `rid(hasPointStyle)` relation to an object in the domain `cid(pointStyle)`. Thus, if no stronger information were provided for, say, part `oid(sutureU245H)`, then

$$\texttt{hasAttr(oid(sutureU245H),rid(hasPointStyle),cid(pointStyle))}^{\mathcal{I}}$$

would be true in any model of the CDF instance. This consequence can be seen as a primitive form of "default", or more precisely indefinite, reasoning that is provided by CDF. Indeed,

$$\texttt{hasAttr(cid(suture547466),rid(hasPointStyle),cid(pointStyle))}^{\mathcal{I}}$$

would also be true, but would be of less interest, since Example 2.1.3 specifically indicates that `oid(suture547466)` is related to a subclass of `cid(pointStyle)`, namely `cid(regularCuttingEdge)`.

[3] This is one of the reasons why object identifiers in CDF denote singleton classes. If two classes c_1 and c_2 are "equal", this simply means that a model must satisfy the sentence $(\forall X) elt(X, c_1) \leftrightarrow elt(X, c_2)$ which has a Herbrand model.

In contrast to `rid(endType)` the relation `rid(suturesRusMatch)` was defined via the `allAttr/3` predicate. However the constraint that any `rid(suturesRusMatch)` must be to a `cid(suturesRusPart)` holds for members of the class `cid(sutures)`, just as it holds for subclasses of `cid(sutures)`. The following formulas summarize inheritance in the first argument of Type-0 relations.

Proposition 2.4.1 (First Argument Inheritance Propagation) Let $\mathcal{M}$ be an ontology model.

1. If $\mathcal{M} \models \mathtt{hasAttr}(\mathtt{Id}_1, \mathtt{Id}_2, \mathtt{Id}_3)^{\mathcal{I}} \wedge \mathtt{isa}(\mathtt{Id}_0, \mathtt{Id}_1)^{\mathcal{I}}$ then $\mathcal{M} \models \mathtt{hasAttr}(\mathtt{Id}_0, \mathtt{Id}_2, \mathtt{Id}_3)^{\mathcal{I}}$
2. If $\mathcal{M} \models \mathtt{allAttr}(\mathtt{Id}_1, \mathtt{Id}_2, \mathtt{Id}_3)^{\mathcal{I}} \wedge \mathtt{isa}(\mathtt{Id}_0, \mathtt{Id}_1)^{\mathcal{I}}$ then $\mathcal{M} \models \mathtt{allAttr}(\mathtt{Id}_0, \mathtt{Id}_2, \mathtt{Id}_3)^{\mathcal{I}}$
3. If $\mathcal{M} \models \mathtt{minAttr}(\mathtt{Id}_1, \mathtt{Id}_2, \mathtt{Id}_3, \mathtt{N})^{\mathcal{I}} \wedge \mathtt{isa}(\mathtt{Id}_0, \mathtt{Id}_1)^{\mathcal{I}}$ then $\mathcal{M} \models \mathtt{minAttr}(\mathtt{Id}_0, \mathtt{Id}_2, \mathtt{Id}_3, \mathtt{N})^{\mathcal{I}}$
4. If $\mathcal{M} \models \mathtt{maxAttr}(\mathtt{Id}_1, \mathtt{Id}_2, \mathtt{Id}_3, \mathtt{N})^{\mathcal{I}} \wedge \mathtt{isa}(\mathtt{Id}_0, \mathtt{Id}_1)^{\mathcal{I}}$ then $\mathcal{M} \models \mathtt{maxAttr}(\mathtt{Id}_0, \mathtt{Id}_2, \mathtt{Id}_3, \mathtt{N})^{\mathcal{I}}$

There is inheritance also in the third argument of relations. Consider again the `hasAttr/3` facts of Example 2.1.3 that states that any element of `cid(absorbableSutures)` is related via `rid(hasPointStyle)` to an element or subclass of `cid(pointStyle)`. By this definition, it also holds that `rid(hasPointStyle)` constrains any element of `rid(absorbaleSutures)` to an element or subclass of any *superclass* of `cid(pointStyle)` so that

$\mathtt{hasAttr(cid(absorbableSutures),rid(hasPointStyle),cid('CDF\ Classes'))}^{\mathcal{I}}$

should also hold in any model of the CDF instance. Similiarly,

$\mathtt{allAttr(cid(dlaPart),rid(suturesRusMatch),id('CDF\ Classes'))}^{\mathcal{I}}$

should also hold. In English, every member or subclass of `cid(sutures)` that has a relation `rid(suturesRusMatch)` has the same relation to a member or subclass of `cid('CDF Classes')`. `minAttr/4` and `classHasAttr/4` behave similarly with regards to third argument inheritance.

Third argument inheritance is different for `maxAttr/4`, however. The fact

```
maxAttr(cid(person),rid(hasGeneticRelation),cid(mother))
```

states that each person has at most one genetic mother. If `cid(mother)` is a subclass of `cid(parent)`, then it is not true that each person has at most one genetic parent. On the other hand, if `cid(elderlyMother)` is a subclass of `cid(mother)` then it is true that each person has at most one genetic elderly mother. The behavior of `maxAttr/4` in models of CDF instances accords with this intuition.

Proposition 2.4.2 (Third-Argument Inheritance Propagation)

1. If $\mathcal{M} \models \mathtt{hasAttr}(\mathtt{Id}_1, \mathtt{Id}_2, \mathtt{Id}_3)^{\mathcal{I}} \wedge \mathtt{isa}(\mathtt{Id}_3, \mathtt{Id}_4)^{\mathcal{I}}$ then $\mathcal{M} \models \mathtt{hasAttr}(\mathtt{Id}_0, \mathtt{Id}_2, \mathtt{Id}_4)^{\mathcal{I}}$
2. If $\mathcal{M} \models \mathtt{allAttr}(\mathtt{Id}_1, \mathtt{Id}_2, \mathtt{Id}_3)^{\mathcal{I}} \wedge \mathtt{isa}(\mathtt{Id}_3, \mathtt{Id}_4)^{\mathcal{I}}$ then $\mathcal{M} \models \mathtt{allAttr}(\mathtt{Id}_0, \mathtt{Id}_2, \mathtt{Id}_4)^{\mathcal{I}}$
3. If $\mathcal{M} \models \mathtt{minAttr}(\mathtt{Id}_1, \mathtt{Id}_2, \mathtt{Id}_3, \mathtt{N})^{\mathcal{I}} \wedge \mathtt{isa}(\mathtt{Id}_3, \mathtt{Id}_4)^{\mathcal{I}}$ then $\mathcal{M} \models \mathtt{minAttr}(\mathtt{Id}_0, \mathtt{Id}_2, \mathtt{Id}_4, \mathtt{N})^{\mathcal{I}}$

4. If $\mathcal{M} \models \mathtt{classHasAttr}(\mathtt{Id}_1, \mathtt{Id}_2, \mathtt{Id}_3)^{\mathcal{I}} \wedge \mathtt{isa}(\mathtt{Id}_3, \mathtt{Id}_4)^{\mathcal{I}}$ then $\mathcal{M} \models \mathtt{classHasAttr}(\mathtt{Id}_0, \mathtt{Id}_2, \mathtt{Id}_4)^{\mathcal{I}}$
5. If $\mathcal{M} \models \mathtt{maxAttr}(\mathtt{Id}_1, \mathtt{Id}_2, \mathtt{Id}_3, \mathtt{N})^{\mathcal{I}} \wedge \mathtt{isa}(\mathtt{Id}_4, \mathtt{Id}_3)^{\mathcal{I}}$ then $\mathcal{M} \models \mathtt{maxAttr}(\mathtt{Id}_0, \mathtt{Id}_2, \mathtt{Id}_4, \mathtt{N})^{\mathcal{I}}$

In CDF, the inheritance in the second argument of relations generalizes or specializes relations. For instance, the relation *parent* can be generalized to *ancestor* or specialized to *mother*. Thus, if *Abraham* is the *parent* of *Isaac*, it is true that he is the *ancestor* of *Isaac* but not necessarily the *mother* of *Isaac*. It follows from the semantics of CDF that `hasAttr/3`, `classHasAttr/3`, and `minAttr/4` all propigate inheritance in their second argument from relations to the super-relations that contain them. `allAttr/3` works differently, however. Any *mother* of *Isaac* is female, but any *parent* is not. Second argument inhertiance propagates from relations to subrelations both in `allAttr/3` and `maxAttr/3`.

Proposition 2.4.3 (Second-Argument Inheritance Propagation)

1. If $\mathcal{M} \models \mathtt{hasAttr}(\mathtt{Id}_1, \mathtt{Id}_2, \mathtt{Id}_3)^{\mathcal{I}} \wedge \mathtt{isa}(\mathtt{Id}_2, \mathtt{Id}_4)^{\mathcal{I}}$ then $\mathcal{M} \models \mathtt{hasAttr}(\mathtt{Id}_0, \mathtt{Id}_4, \mathtt{Id}_3)^{\mathcal{I}}$
2. If $\mathcal{M} \models \mathtt{classHasAttr}(\mathtt{Id}_1, \mathtt{Id}_2, \mathtt{Id}_3)^{\mathcal{I}} \wedge \mathtt{isa}(\mathtt{Id}_2, \mathtt{Id}_4)^{\mathcal{I}}$ then $\mathcal{M} \models \mathtt{classHasAttr}(\mathtt{Id}_0, \mathtt{Id}_4, \mathtt{Id}_3)^{\mathcal{I}}$
3. If $\mathcal{M} \models \mathtt{minAttr}(\mathtt{Id}_1, \mathtt{Id}_2, \mathtt{Id}_3, \mathtt{N})^{\mathcal{I}} \wedge \mathtt{isa}(\mathtt{Id}_2, \mathtt{Id}_4)^{\mathcal{I}}$ then $\mathcal{M} \models \mathtt{minAttr}(\mathtt{Id}_0, \mathtt{Id}_4, \mathtt{Id}_3, \mathtt{N})^{\mathcal{I}}$
4. If $\mathcal{M} \models \mathtt{allAttr}(\mathtt{Id}_1, \mathtt{Id}_2, \mathtt{Id}_3)^{\mathcal{I}} \wedge \mathtt{isa}(\mathtt{Id}_4, \mathtt{Id}_2)^{\mathcal{I}}$ then $\mathcal{M} \models \mathtt{allAttr}(\mathtt{Id}_0, \mathtt{Id}_4, \mathtt{Id}_3)^{\mathcal{I}}$
5. If $\mathcal{M} \models \mathtt{maxAttr}(\mathtt{Id}_1, \mathtt{Id}_2, \mathtt{Id}_3, \mathtt{N})^{\mathcal{I}} \wedge \mathtt{isa}(\mathtt{Id}_4, \mathtt{Id}_2)^{\mathcal{I}}$ then $\mathcal{M} \models \mathtt{maxAttr}(\mathtt{Id}_0, \mathtt{Id}_4, \mathtt{Id}_3, \mathtt{N})^{\mathcal{I}}$

We conclude this chapter with a discussion of irredundant sets and principle classes, both of which are used in the operational semantics of the Type-0 interface in Chapter 4.

The above inheritance propositions can be made into a simple proof system, INH, by considering the facts themselves rather than their interpretations into an ontology theory. For instance, give a CDF instance $\mathcal{O}$ containing `hasAttr(Id1,Id2,Id3)` and `isa(Id3,Id4)` then `hasAttr(Id1,Id2,Id4)` can be deduced. In other words, INH contains an inference rule,

$$\frac{\mathtt{hasAttr}(\mathtt{Id1}, \mathtt{Id2}, \mathtt{Id3}), \mathtt{isa}(\mathtt{Id3}, \mathtt{Id4})}{\mathtt{hasAttr}(\mathtt{Id1}, \mathtt{Id2}, \mathtt{Id4})}$$

based on Proposition 2.4.2.1. The INH proof system contains similar inference rules obtained from Proposition Proposition 2.4.1.(1-2), Proposition 2.4.2.(2-3), Proposition 2.4.3.(1-3), and the transitive reflexive closure of the `isa/2` facts, and forms the main reasoning mechanism for Type-0 instances.

Definition 2.4.1 Let $\mathcal{O}$ be a Type-0 CDF instance, $\mathcal{S} \subseteq \mathcal{O}$, and $\mathcal{O}_{isa}$ be the set of `isa/2` facts in $\mathcal{O}$. Then a fact $f \in \mathcal{S}$ is *irredundant in* $\mathcal{S}$ if there is no other $f' \in \mathcal{S}, f' \neq f$ such that

$$\mathcal{O}_{isa}, f' \vdash_{\mathrm{INH}} f$$

$\mathcal{S}$ is irredundant if each fact in it is irredundant. An irredundant basis for $\mathcal{S}$ is a irredundant $\mathcal{S}' \subseteq \mathcal{S}$ such that for all $f \in \mathcal{S}$,

$$(\mathcal{S}' \cup \mathcal{O}_{isa}) \vdash_{\mathrm{INH}} f$$

Given an identifier I, a class C is a *principle class for I* if $\mathcal{O}_{isa} \vdash_{\mathrm{INH}} I \neq C \wedge isa(I,C)$, but $\mathcal{O}_{isa} \not\vdash_{\mathrm{INH}} isa(I,C') \wedge (C' \neq C) \wedge isa(C',C)$. Similarly a relation identifier R is a *principle relation for object identifiers O_1 and O_2* if $\mathcal{O}_{isa} \vdash_{\mathrm{INH}} rel(O_1,R,O_1)$ but $\mathcal{O}_{isa} \not\vdash_{\mathrm{INH}} rel(O_1,R',O_2) \wedge (R \neq R') \wedge isa(R',R)$

Finally, if $\mathcal{S},\mathcal{S}' \subseteq \mathcal{O}$, and $\mathcal{S}$ and $\mathcal{S}'$ are both irredundant sets, then $\mathcal{S}$ *is more specific than $\mathcal{S}'$ in $\mathcal{O}$*, $\mathcal{S} \succ_{spec} \mathcal{S}'$ if there is a $f \in \mathcal{S}$ and $f' \in \mathcal{S}'$, $f \neq f'$ such that $\mathcal{O}_{isa}, f \vdash_{\mathrm{INH}} f'$, but there is no $g' \in \mathcal{S}'$ $g \in \mathcal{S}$, $g \neq g$ such that $\mathcal{O}_{isa}, g \vdash_{\mathrm{INH}} g'$. □

It is straightforward that any $\mathcal{S} \subseteq \mathcal{O}$ contains an irredundant basis (recall that $\mathcal{O}$ is finite). Furthermore, $\mathcal{S}$ contains a unique irredundant basis if the `isa/2` facts in $\mathcal{O}$ are "acyclic", i.e. if there are no two non-identical identifiers $I, I' \in \mathcal{O}$ such that $\vdash_{\mathrm{INH}} isa(I,I')$ and $\vdash_{\mathrm{INH}} isa(I',I)$

Chapter 3

The Meaning of Type-1 CDF Instances

Type-1 CDF instances differ from Type-0 CDF instances simply in that they allow a new kind of fact: `necessCond/2`. To explain the power and usefulness of this kind of fact, we first describe class expressions, which arise from the field of description logic.

3.1 Class Expressions and CDF Facts

While Type-0 CDF Instances are useful in practice, they lack expressiveness in certain situations, as the following example shows.

Example 3.1.1 Consider the following CDF fragment which defines conflicting materials for the thread of a suture object, named `oid(inconSuture)`, using the schema of Example 2.1.4.

```
isa(oid(inconSuture),cid(absorbableSuture))
isa(oid(inconNeedle),cid(absSutNeedle))
isa(oid(inconThread),cid(absSutThread))

hasAttr(oid(inconSuture),rid(hasImmedPart),oid(inconThread))
hasAttr(oid(inconThread),rid(hasMaterial),cid(gut))
hasAttr(oid(inconThread),rid(hasMaterial),cid(polyglyconate))
```

It is easy to show that when the schema of Example 2.1.4 is extended with this fragment, it has a model. Because the schema contains the fact:

```
maxAttr(cid(absSutPart),rid(hasMaterial),cid(absSutMaterial),1)
```

the model contains an element that is in the class `cid(polyglyconate)` as well as in the class `cid(gut)`. However, such a model is unintended, as gut and polyglyconate are separate materials.

In order to address such situations, classes and objects may be defined in terms of *class expressions*.

Definition 3.1.1 Let $\mathcal{L}$ be an ontology language. A *CDF class expression* C over $\mathcal{L}$ is formed by one of the following constructions in which A is a class or object identifier C_1 and C_2 class expressions, R a relation identifier, and n a natural number.

$$C \leftarrow A | not\ C_1 | C_1, C_2 | C_1; C_2 || all(R, C_1) | exists(R, C_1) | atLeast(n, R, C_1) | atMost(n, R, C_1)$$

If C_1 and C_2 are class expressions, then $C_1 \subseteq C_2$ is an *inclusion statement*, □

Note that in the definition above, A can be either an atomic or product identifier. Also note that Prolog-style syntax is used: *and* is represented as *","* and *or* as *";"*.

The meaning of class expressions in terms of ontology theories will be defined in Section 3.1.1. For now, we informally illustrate their meaning through a series of examples.

As their name implies, class expressions provide a means for describing classes. Intuitively, the *all* constructor may seem to have some correspondance to `allAttr/3`, *exists* to `hasAttr/3`, *atLeast* to `minAttr/4` and *atMost* to `maxAttr/4`. More precisely if the sorting predicates are ignored, then

```
allAttr(cid(source),rid(r),cid(target))
```

translates to an *inclusion statement* between class expressions.

$$cid(source) \subseteq all(rid(r), cid(target))$$

where the inclusion statement $C_1 \subseteq C_2$ is true in an ontology structure if the set of elements denoted by C_1 is a subset of the set of elements denoted by C_2. The other Type-0 predicates translate to inclusion statements in a similar manner.

Example 3.1.2 Let C_1 be the class expression:

cid(absorbableSutures),
all(rid(hasNeedleDesign),cid(needleDesign)),
all(rid(hasPointStyle),cid(pointStyle)),
  exists(rid(hasPointStyle),cid(pointStyle),
all(rid(hasImmedPart),cid(absSutPart)),
  exists(rid(hasImmedPart),cid(absSutNeedle)),
  exists(rid(hasImmedPart),cid(absSutThread))

This class expression can be read in English as

"The class of all elements that are in `cid(absorbableSutures)` and
all of whose `rid(hasNeedleDesign)` relations are to elements in the class `cid(needleDesign)` and
all of whose `rid(hasPointStyle)` relations are to elements in the class `cid(pointStyle)` and
that has a `rid(hasPointStyle)` relation to an element in the class `cid(pointStyle)` and
all of whose `rid(hasImmedPart)` relations are to elements in the class `cid(absSutPart)` and
that has a `rid(hasPointStyle)` relation to an element in the class `cid(absSutNeedle)` and
that has a `rid(hasPointStyle)` relation to an element in the class `cid(absSutThread)`".

Note that from our running example, the CDF facts that pertain to `absorbableSutures)` are:

```
isa(cid(absorbableSutures),cid(sutures))
allAttr(cid(absorbableSutures),rid(hasNeedleDesign),cid(needleDesign))
allAttr(cid(absorbableSutures),rid(hasPointStyle),cid(pointStyle))
     hasAttr(cid(absorbableSutures),rid(hasPointStyle),cid(pointStyle))
allAttr(cid(absorbableSutures),rid(hasImmedPart),cid(absSutPart))
     hasAttr(cid(absorbableSutures),rid(hasImmedPart),cid(absSutNeedle))
     hasAttr(cid(absorbableSutures),rid(hasImmedPart),cid(absSutThread))
```

Other facts hold, but they are redundant (Definition 2.4.1). From these facts along with the semantics given in Section 2, it is not hard to see that if an element O is in the class `cid(absorbableSutures)` then it must also be in C.

From Definitions 2.1.2 and 2.1.3, a CDF object identifier denotes a class with a uniqueness constraint. Because of this, object identifiers and class identifiers can be intermingled withing class expressions.

Example 3.1.3 Consider the object identifier `oid(inconSuture)`, introduced in Example 3.1.1. As `oid(inconSuture)` is a subclass of `cid(absorbableSuture)`, the facts

```
(1)  isa(oid(inconSuture),cid(absorbableSuture)
(2)  allAttr(oid(inconSuture),rid(needleDesign),cid(needleDesignType))
```

hold (cf. Example 2.1.2). as does

```
(3) allAttr(oid(inconSuture),rid(hasPointStyle),cid(pointStyle))
(4) hasAttr(oid(inconSuture),rid(hasPointStyle),cid(pointStyle))
```

(cf. Example 2.1.3). Furthermore, from Example 2.1.4

```
(5) allAttr(oid(inconSuture),rid(hasImmedPart),cid(absSutPart))
(6) hasAttr(oid(inconSuture),rid(hasImmedPart),cid(absSutNeedle))
```

all hold, along with

```
(7) hasAttr(oid(inconSuture),rid(hasImmedPart),oid(inconThread))
```

It is not hard to convince yourself that these facts imply that the `oid(inconSuture)` belongs to the class C_2 described by

oid(inconSuture), cid(absorbableSutures),
all(rid(hasneedleDesign),cid(needleDesign)),
all(rid(hasPointStyle),cid(pointStyle)),
exists(rid(hasPointStyle),cid(pointStyle)),
all(rid(hasImmedPart),cid(absSutPart)),
exists(rid(hasImmedPart),cid(absSutNeedle)),
exists(rid(hasImmedPart),oid(inconThread))

Note that C_2 is similar to C_1, except that the object identifiers imply that `cid(inconNeedle)` and `cid(inconThread)` contain one and only one element.

3.1.1 Class Expressions and Ontology Theories

Class expressions can be formally translated into the ontology languages of Definition 2.1.2 using the following definition from [Swi04] (see, e.g. [CGLN02] for a general exposition of class expressions).

Definition 3.1.2 Let C a CDF class expression, over an ontology language $\mathcal{L}$ in which the variables are denoted as X_i, for positive integers i. Then the translation of C, $C[X_0]^{\mathcal{I}}$, is a first-order formula defined by the following rules, which use a global variable number, $vnum$ that is initialized to 1 at the start of the transformation.

- if $C = A$ and A is an object identifier, then $C[X_N]^{\mathcal{I}} = elt(X_N, A) \wedge \forall X_{vnum}[elt(X_{vnum}, A) \Rightarrow X_N = X_{vnum}]$, where $vnum$ is incremented before the next step of the translation.
- if $C = A$ and A is a class identifier, then $C[X_N]^{\mathcal{I}} = elt(X_N, A)$.
- if $C = (C_1, C_2)$, then $C[X_N]^{\mathcal{I}} = C_1[X_N]^{\mathcal{I}} \wedge C_2[X_N]^{\mathcal{I}}$.
- if $C = (C_1; C_2)$, then $C[X_N]^{\mathcal{I}} = C_1[X_N]^{\mathcal{I}} \vee C_2[X_N]^{\mathcal{I}}$.
- if $C = not\ C_1$, then $C[X_N]^{\mathcal{I}} = \neg C_1[X_N]^{\mathcal{I}}$.
- if $C = exists(R, C_1)$, then $C[X_N]^{\mathcal{I}} = \exists X_{vnum}.rel(X_N, R, X_{vnum}) \wedge C_1[X_{vnum}]^{\mathcal{I}}$, where $vnum$ is incremented before the next step of the translation.
- if $C = all(R, C_1)$, then $C[X_N]^{\mathcal{I}} = \forall X_{vnum}.rel(X_N, R, X_{vnum}) \Rightarrow C[X_{vnum}]^{\mathcal{I}}$, where $vnum$ is incremented before the next step of the translation.
- if $C = atLeast(m, R, C_1)$, then $C[X_N]^{\mathcal{I}} = \exists^{\geq m} X_{vnum}[rel(X_N, R, X_{vnum}) \wedge C[X_{vnum}]^{\mathcal{I}}]$ where $vnum$ is incremented m times before the next steps of the translation.
- if $C = atMost(m, R, C_1)$, then $C[X_N]^{\mathcal{I}} = \neg atLeast(m + 1, R, C_1)[X_N]^{\mathcal{I}}$

If $C_1 \subseteq C_2$ is an inclusion statement, its tranlation is $(\forall X_0)[C_1[X_0]^{\mathcal{I}} \Rightarrow C_2[X_0]^{\mathcal{I}}]$ □

Example 3.1.4 If C_1 is the class expression from Example 3.1.3, then $C_1[X_0] =$

$$
\begin{array}{l}
elt(X_0, cid(absorbableSutures)) \wedge (\forall X_1)[elt(X_1, oid(absorbableSutures)) \Rightarrow X_0 = X_1] \\
(\forall X_2)[rel(X_0, rid(hasNeedleDesign), X_2) \Rightarrow elt(X_2, cid(needleDesign))] \wedge \\
(\forall X_3)[rel(X_0, rid(hasPointStyle), X_3) \Rightarrow elt(X_3, cid(pointStyle))] \wedge \\
\qquad (\exists X_4)[rel(X_0, rid(hasPointStyle), X_4) \wedge elt(X_4, cid(pointStyle))] \wedge \\
(\forall X_5)[rel(X_0, rid(hasImmedPart), X_5) \Rightarrow elt(X_5, cid(absSutPart))] \wedge \\
\qquad (\exists X_6)[rel(X_0, rid(hasImmedPart), X_6) \wedge elt(X_6, cid(absSutNeedle))] \wedge \\
\qquad (\exists X_7)[rel(X_0, rid(hasImmedPart), X_7) \wedge elt(X_7, oid(inconThread)) \wedge \\
\qquad\qquad (\forall X_8)[elt(X_8, oid(inconThread)) \Rightarrow X_7 = X_8]]
\end{array}
$$

To summarize this section, CDF instances have direct translations into ontology theories, as presented in Chapter 2. Class expressions can also be translated into ontology theories via Definition 3.1.2, and these two translations induce a unique translation from CDF instances to inclusion statements between class expressions. Thus a CDF instance can be seen as a set of sentences in an ontology theory, or as a set of inclusion statements between class expressions. At the same time, there are class expresion constructors that are not used when Type-0 instances are translated into class expressions, namely *";"* and *not*. Arbitrary class expressions are denoted by stating necessary conditions, to which we now turn.

3.2 Necessary Conditions

The unintended model of Example 3.1.1 can be prevented by stating that a necessary condition for an object to be in `cid(gut)` is that the object not be in `cid(polyglyconate)`. This is handled by a new type of fact `necessCond/2` that is used to state necessary conditions.

Example 3.2.1 Adding the fact

`necessCond(cid(gut),vid(not(cid(polyglyconate)))).`

denotes that no element of `cid(gut)` can be in `cid(polyglyconate)`.

Type-1 CDF instances thus have a new kind of identifier: a *virtual identifier*, denoted by the functor `vid/1`. A virtual identifier indicates that rather than denoting a class by its name, it is denoted by a class expression whose syntax is given in Definition 3.1.1, and whose meaning is given by the translation of Definition 3.1.2.

Translation Rule 6 (Translation of `necessCond/2`) For each fact of the form `necessCond(Cid,Vid)` add the axiom

$$isClass(Cid) \wedge (\forall X)[elt(X, Cid) \Rightarrow Vid[X]^{\mathcal{I}}]$$

denoted as $\texttt{necessCond(Cid,Vid)}^{\mathcal{I}}$.

Example 3.2.2 The translation of the fact from Example 3.2.1 is the axiom

$$isClass(cid(gut)) \wedge (\forall X_0)[elt(X_0, cid(gut)) \Rightarrow \neg elt(X_0, cid(polyglyconate))]$$

It is easy to see that an instance containing this fact and those of Example 3.1.1, combined with the Schema of Example 2.1.4 will not have a model. Note that Axiom 3 (Non-Empty Classes) is essential to deriving this inconsistency.

The following proposition is straightforward.

Proposition 3.2.1 (First Argument Inheritance Propagation for `necessCond/2`) Let $\mathcal{M}$ be an ontology model.

1. If $\mathcal{M} \models \texttt{necessCond}(\texttt{Id}_1, \texttt{Id}_2)^{\mathcal{I}} \wedge \texttt{isa}(\texttt{Id}_0, \texttt{Id}_1)^{\mathcal{I}}$ then $\mathcal{M} \models \texttt{necessCond}(\texttt{Id}_0, \texttt{Id}_2)^{\mathcal{I}}$

The INH proof system can be extended to use a deduction rule based on this proposition, in a straightforward way.

3.3 Local Class Expressions

When translating information in a CDF instance directly to a class expression, a difficulty can arise. In the examples 3.1.2 and 3.1.3 class identifiers within the *all*, *exists*, *atLeast* and *atMost* constructors can be considered "primitive" in the sense that these class identifiers do not occur as the first argument of any `allAttr/3`, `hasAttr/3`, `minAttr/4` or `maxAttr/4` predicates. However, it is not always the case that only primitive classes occur in these positions.

Example 3.3.1 Consider the following schema for family relations, adapted from [BN03].

```
isa_ext(cid(gender),cid('CDF Classes')).
isa_ext(cid(female),cid(gender)).
isa_ext(cid(male),cid(gender)).

isa_ext(cid(person),cid('CDF Classes')).
     allAttr_ext(cid(person),rid(hasBrother),cid(man)).
     allAttr_ext(cid(person),rid(hasSister),cid(woman)).

isa_ext(cid(woman),cid(person)).
     isa_ext(cid(woman),cid(female)).
isa_ext(cid(man),cid(person)).
```

The class expression for `cid(person)` is

cid('CDF Classes'),
all(rid(hasBrother),cid(man)),
all(rid(hasSister),cid(woman))

while the class expression for `cid(man)` is

cid(person),
all(rid(hasBrother),cid(man)),
all(rid(hasSister),cid(woman))

In the above example, a class expression for `cid(person)` contains subclasses of `cid(person)`, while a class expression for `cid(man)` contains the class `cid(man)` itself. A CDF instance can be seen as a type of *termonology system*, (sometimes called a TBox in the literature). CDF classes and objects are defined in terms of other classes and objects that may themselves be defined in terms of classes and objects. If a terminology system has a class identifier that is defined in terms of itself, it is sometimes called *cyclic*, otherwise the terminology system is *acyclic*. As the previous example shows, many of the most natural terminology systems are cyclic.

At this point, it is useful to introduce some "terminology" of our own, at a somewhat informal level. Given a class or object identifier C in a CDF instance $\mathcal{O}$, a *level 1 local class expression* for C is formed by cojoining all principle classes for C along with translations of non-isa facts for C (i.e. `hasAttr/3` into an *exists* constructor, `allAttr/3` into an *all* constructor, etc.) A *level n local class* *expression* for C is then defined as follows. We start with an empty *ancestorList*, and construct a level 1 local class expression, CE for C. Next we add C to the *ancestor list*, and construct a level $n-1$ class expression for each identifier in CE that is not in the ancestor list. Intuitively, constructing a level n local class expression "unfolds" or "expands" class expressions and identifiers n times, ignoring cycles.

The concept of local class expressions is used to explain features of the Type-1 CDF interface, as descrined in Section 4.3.

Part II

Using CDF

Chapter 4

Implementation and System Features

In this section we describe how the CDF system implements the semantincs of Part I, as well as many other features for ontology management. We begin by describing CDF identifiers, facts, and rules in Section 4.1. The Type-0 query interface, based on tabled resolution is described in Section 4.2. Section 4.2 describes the Type-1 API, which is based on the CDF therorem prover. Update and consistency predicates for CDF are described in Section 4.4. Section 4.6 describes how to configure CDF, along with predicates that allow the user to examine aspects of the CDF state. Section 4.7 describes basic I/O for CDF, along with a more sophisticated *component* system that is built on top of basic I/O. Section 4.8 describes database interfaces for CDF, and Section 4.9 describes concurrency support.

Terminology Throughout this chapter, we assume a general familiarity with the terminology and conventions of Prolog in general and XSB in particular. We discuss here our specialized conventions. By *sorts* we mean the various sorts for identifiers discussed in Chapter 2 and in Definition 4.1.1. We reserve the term *types* for types as defined in the ISO-Prolog standard [**?**]. We use *domains* to refer to the "types" that are useful in CDF. Each CDF domain is defined by a 1-ary predicate that begins with `is`, such as `isContextType/1`, `isType0Term/1` etc., and we refer to domains via their predicate names and arities (e.g. `isContextType/1`, etc).

4.1 CDF Instances

Part I simplified the syntax of CDF instances in certain ways. In this section we describe those aspects of the actual CDF implementation that differ from the abstract presentation Part I.

The first major difference concerns CDF identifiers.

Definition 4.1.1 [CDF Identifiers] A *CDF identifier* has the functor symbol `cid/2`, `oid/2`, `rid/2`, or `crid/2`. The second argument of a CDF identifier is a Prolog atom and is called its *component tag*, while the first argument is either

1. a Prolog atom or

2. a term $f(I_1, \ldots, I_n)$ where $I_1, \ldots, I_n$ are CDF identifiers.

In the first case, an identifier is called *atomic*; in the second it is called a *product identifier*. □

Thus in an implementation of, say, Example 2.1.1 all identifiers would have component tags. For instance the identifier `cid(absorbableSutures)` might actually have the form `cid(absorbableSutures,unspsc)` and `oid(sutureU245H)` would have the form `oid(sutureU245H,sutureRus)`. These component tags have two main uses. First, they allow ontolgies from separate soruces to be combined, and thus function in a manner somewhat analogous to XML namespaces. Second, the component tags are critical to the CDF component system, described in Section 4.7.

4.1.1 Extensional Facts and Intensional Rules

An actual CDF instance is built up of *extensional facts* and *intensional rules* defined for the CDF predicates `isa/2` `allAttr/3`, `hasAttr/3`, `classHasAttr/3`, `coversAttr/3`, `minAttr/4` and `maxAttr/4`. Extensional facts for these predicates add the suffix `_ext` to the suffix name leading to `isa_ext/2`, `allAttr_ext/2` and so on. Intensional rules add the suffix `_int` leading to `isa_int/2`, `allAttr_int/2` etc.

Extensional facts make use of XSB's sophisticated indexing of dynamic predicates. Since CDF Extensional Facts use functors such as `cid/1` or `oid/1` to type their arguments, traditional Prolog indexing, which makes use only of the predicate name and outer functor of the first argument, is unsuitable for large CDF instances. CDF extensional facts use XSB's star-indexing [XSB03]. For ground terms, star-indexing can index on up to the first five positions of a specified argument. In addition, various arguments and combinations of arguments can be used with star-indexing of dynamic predicates. The ability to index within a term is critical for the performance of CDF; also since star-indexing bears similarities to XSB's trie-indexing [RRS+99], it is spatially efficient enough for large-scale use. Section 4.6 provides information on default indexing in CDF and how it may be reconfigured.

Intensional rules may be defined as XSB rules, and may use any of XSB's language or library features, including tabling, database, and internet access. Intensional rules are called on demand, making them suitable for implementing functionality from lazy database access routines to definitions of primitive types.

Example 4.1.1 In many ontology management systems, integers, floats, strings and so on are not stored explicitly as classes, but are maintained as a sort of *primitive type*. In CDF, primitive types are easily implemented via intensional rules like the following.

```
isa_int(oid(Float),cid(allFloats)):-
   (var(Float) -> cdf_mode_error ; float(Float).
```

CDF provides intensional rules defining all Prolog primitive types as CDF primitive types in the component `cdfpt` (see below). Other, more specialized types can be defined by users by defining intensional rules along the same lines FILL IN 'BELOW'; TABLING AND INTENSIONAL RULES

As mentioned above, the predicate `immed_hasAttr/3`, (and `immed_allAttr/3`, etc) is used to store basic CDF information that is used by predicates implementing `hasAttr/3` and other relations. `immed_hasAttr/3` itself is implemented as:

```
immed_hasAttr(X,Y,Z):- hasAttr_ext(X,Y,Z).
immed_hasAttr(X,Y,Z):- hasAttr_int(X,Y,Z).
immed_hasAttr(X,Y,Z):- immed_minAttr(X,Y,Z,_).
```

The above code fragment illustrates two points. First, `immed_hasAttr/3` is defined in terms of `immed_minAttr/3`, fulfilling the semantic requirements of Section 2.1. It also illustrates that `immed_hasAttr/3` is implemented in terms both of extensional facts `hasAttr_ext/3` and intensional rules `hasAttr_int(X,Y,Z)`.

4.1.2 The Top-level Hierarchy and Primitive Types

All CDF instances share the same top-level hierarchy, as depicted in Figure 4.1. All classes and objects are subclasses (through the `isa` relation) to `cid('CDF Classes',cdf)`, all relations are subrelations of `rid('CDF Object-Object Relations',cdf)` and all class relations are subrelations of `crid('CDF Class-Object Relations',cdf)`.

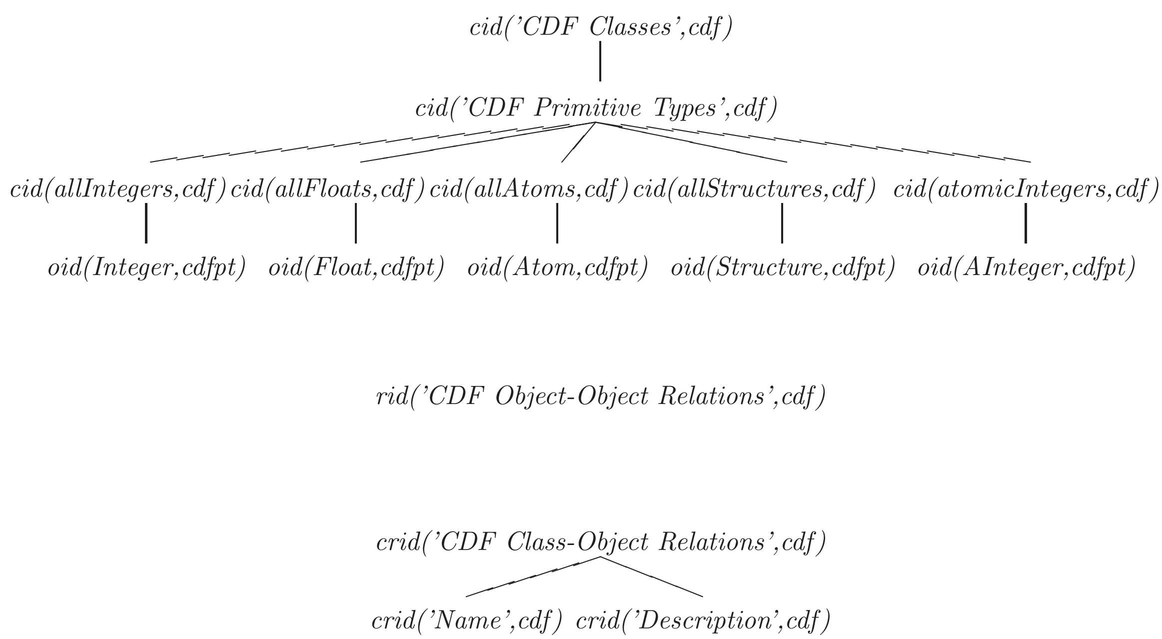

Figure 4.1: Built-in Inheritance Structure of CDF

An immediate subclass of `cid('CDF Classes',cdf)` is `cid('CDF Primitive Types',cdfpt)`. This class allows users to maintain in CDF any legally syntactic Prolog element, and forms an exception to Definition 4.1.1. Specifically `cid('CDF Primitive Types',cdf)` contains Prolog atoms, integers, floats, structures and what are termed "atomic integers" — integers that are represented in atomic format, e.g. '01234'. Primitive types are divided into five subclasses, `cid(allIntegers,cdf)`, `cid(allFloats,cdf)`, `cid(allAtoms,cdf)`, `cid(allStructures,cdf)`, and `cid(atomicInteger,cdf)`. Each of these in turn have various objects as their immediate subclasses [1], whose inheritance relation is defined by an intensional rule like the one presented in Example 4.1.1. Thus, if the number 3.2 needs to be added to an ontology, perhaps as the value of an attribute, it is represented as `oid(3.2,cdfpt)`, and it will fit into the inheritance hierarchy as a subclass of `cid(allFloats,cdf)`. The intensional rules are structured so that for any Prolog syntactic element `X`, when `X` is combined with the component `cdfpt`, then `cid(X,cdfpt)` will be a subclass of `cid('CDF Primitive Types',cdfpt)`, as will be `oid(X,cdfpt)`.

4.1.3 Basic CDF Predicates

`isa_ext/2` usermod
`allAttr_ext/3` usermod
`hasAttr_ext/3` usermod
`classHasAttr_ext/3` usermod
`minAttr_ext/4` usermod
`maxAttr_ext/4)` usermod
`necessCond_ext/2)` usermod

These dynamic predicates are used to store extensional facts in CDF. They can be called directly from the interpreter or from files that are not modules, but must be imported from `usermod` by those files that are modules. Extensional facts may be added to a CDF system via `newExtTerm/1` (Section 4.4), or imported from a `cdf_extensional.P` file (Section 4.7).

`isa_int/2` usermod
`allAttr_int/3` usermod
`hasAttr_int/3` usermod
`classHasAttr_int/3` usermod
`minAttr_int/4` usermod
`maxAttr_int/4)` usermod
`necessCond_int/2)` usermod

These dynamic predicates are used to store intensional rules in CDF. They can be called directly from the interpreter or from files that are not modules, but must be imported from `usermod` by those files that are modules. Intensional rules may be added to a CDF system via `???newIntRule/1` (Section 4.4), or imported from a `cdf_intensional.P` file (Section 4.7).

`immed_isa/2` cdf_init_cdf

`immed_isa(SubCid,SupCid)` is true if there is a corresponding fact in `isa_ext/2` or in the intensional rules. It does not use the Implicit Subclassing Axiom 6, the Domain Containment Axiom 4, or reflexive or transitive closure.

`immed_allAttr/3` cdf_init_cdf
`immed_hasAttr/3` cdf_init_cdf
`immed_classHasAttr/3` cdf_init_cdf
`immed_minAttr/4` cdf_init_cdf

[1] Recall that objects in CDF are singleton classes.

`immed_maxAttr/4)` cdf_init_cdf
`immed_necessCond/2)` cdf_init_cdf

Each of these predicates calls the corresponding extensional facts for the named predicate as well as the intensional rules. No inheritance mechanisms are used, and any intensional rules unifying with the call must support the call's instantiation pattern.

`cdf_id_fields/4` cdf_init_cdf

`cdf_id_fields(ID,Functor,NatId,Component)` is true if `ID` is a legal CDF identifier, `Functor` is its main functor symbol, `NatId` is its first field and `Component` is its second field. This convenience predicate provides a faster way to examine CDF identifiers than using `functor/3` and `arg/3`.

4.2 The Type-0 Query Interface

There are two main design goals behind the Type-0 query interface.

- to provide a Prolog interface to CDF based on the axioms in Chapter 2.1, and the INH proof system derived from Section 2.4 along with Proposition 3.2.1.
- to provide a highly efficient and scalable interface to CDF.

Indeed, the Type-0 interface has been used to support CDF instances containing nearly a million extensional facts that require heavy manipulation and access, and are used as back-ends to interactive graphical systems. As discussed below, this need for efficiency affects certain aspects of the interface.

4.2.1 Virtual Identifiers

As discussed, Type-0 instances do not make contain facts of the form `necessCond/2`. In the implementation of CDF, `necessCond/2` goals can be called, and their implementation obeys the first-argument inheritance for `necessCond`. However, it is important to note that **the Type-0 interface does not use information in virtual identifiers** as the following example shows.

Example 4.2.1 Consider the CDF instance containing only the fact

```
necessCond_ext(cid(a,test),vid(exists(rid(r,test),cid(b,test)))).
```

by the semantics of Chapter 3, this CDF instance logically entails

`hasAttr(oid(a,test),rid(r.text),cid(b,text))`$^{\mathcal{I}}$

However the Type-0 interface will answer "no" to the query

```
?- hasAttr(oid(a,test),rid(r.text),cid(b,text)).
```

4.2.2 Computing Irredundant Answers

Consider the running sutures example of Chapter 2.1 to which is added a fact

```
hasAttr(oid(sutureU245H),rid(needleDesign),cid('Adson Dura')).
```

Suppose the query `?- hasAttr(oid(sutureU245H),rid(needleDesign),Y)` were asked. Via an INH proof, rhe CDF instance would imply the answers

```
hasAttr(oid(sutureU245H),rid(needleDesign),cid('Adson Dura'))
hasAttr(oid(sutureU245H),rid(needleDesign),cid(needleDesignTypes))
hasAttr(oid(sutureU245H),rid(needleDesign),cid('CDF Classes))
```

The last two answers are, of course, redundant according to Definition 2.4.1. Omitting redundant answers is important both for human comprehension of information in a CDF instance, and to reduce excessive backtracking in applications.

Computation of irredundant answers is done in CDF by creating a *preference relation* on the relations `hasAttr/3`, `classHasAttr/3`, `allAttr/3`, `minAttr/3`, `maxAttr/3` and `necessCond` using the techniques of [CS02]. The schematic code for a query to `hasAttr/3` in which the first argument is known to be bound, and the second two free, is shown in Figure 4.2. Basic information concerning `hasAttr/3` within a CDF instance is kept via the predicate `immed_hasAttr/3` (and similarly for other CDF relations), and `hasAttr/3` uses `immed_hasAttr/3` to compute implications via inheritance upon demand. In the compilation of the code in Figure 4.2, well-founded negation is used to ensure that only preferred answers are returned. It is easy to see by comparing the preference rules of Figure 4.2 with Propositions 2.4.1-2.4.3, that the preference rule ensures that answers are returned only if they are not implied by other answers. Similar approaches are used for other query modes and CDF relations.

```
hasAttr(X,Y,Z):-
    nonvar(X),
    (var(Y) -> hasAttr_bff(X,Y,Z) ; hasAttr_bbf(X,Y,Z)).

:- table hasAttr_bbf/3.
hasAttr_bbf(X,Y,Z):-
    isa(X,XSup),
    isa(Y,YSup),
    immed_hasAttr(XSup,YSup,Z).

prefer(hasAttr_bbf(X,y,Z1),hasAttr_bbf(X,Y,Z2)):-
    isa(Z1,Z2),\+(Z1 = Z2).

:- table hasAttr_bff/3.
hasAttr_bff(X,Y,Z):-
    isa(X,XSup),
    immed_hasAttr(XSup,Y,Z).

prefer(hasAttr_bff(X,Y1,Z1),hasAttr_bff(X,Y2,Z2)):-
    isa(Y1,Y2),\+(Y1 = Y2),
    isa(Z1,Z2),\+(Z1 = Z2).
```

Figure 4.2: Schematic Representation for Selected Modes of `hasAttr/3` Implementation

4.2.3 Implementations of `isa/2`

In implementing `isa/2` there are a number of tradeoffs to be made between semantic power and efficiency. We discuss some of them here in order to motivate the design of the Type-0 API.

To Table or Not to Table

Tabling `isa/2` (or the predicates that underly it) may have several advantages. First, consider the goal `?- isa(cid('CDF Root',cdf),X)` that traverses through the entire `isa/2` hierarchy. Is `isa/2` is tabled, `X` will be instantiated once for each class in the hierarchy. If `isa/2` is not tabled, `X` will be instantiated for every path in the hierarchy whose initial class is `cid('CDF Root',cdf)`. Since the number of paths in a directed graph can be exponential in the number of nodes in the graph, a failure to table `isa/2` can potentially be disasterous. Whether it is or not depends on the structure of the inheritance hierarchy. To the extent the hierarchy is tree-like, tabling `isa/2` will not be of benefit, as the number of paths from any node in a tree is equal to the number of nodes in the tree. Indeed, in such a case, tabling `isa/2` could be a disadvantage, as large parts of the hierarchy may need to be materialized in a table. On the other hand, if there is much multiple inheritance in the hierarchy, tabling `isa/2` can vastly improve performance. A second consideration is whether intensional rules are used in a CDF instance, and if so, the form of the rules. If intensional rules themselves call predicates in the Type-0 interface, there is a risk of infinite loops if `isa/2` isn't tabled.

As a result of these considerations, certain predicates underlying `isa/2` are tabled. However, this tabling can be removed by reconfiguring and recompiling CDF. To do this, the file `cdf_definitions.h` in `$XSBHOME/packages/altCDF` must be edited, changing the line

```
DEFINE TABLED_ISA 1
```

to

```
DEFINE TABLED_ISA 0
```

and recompiling `cdf_init_cdf.P`.

Product Classes

From an operational perspective however, a query @tt?- isa(X,Y) can easily be intractable if product classes are used.

Example 4.2.2 Let `cid(boolean,s)` be a class with two subclasses: `oid(true,s)` and `oid(false,s)`. Then the product class `cid(f(cid(boolean,s),...,cid(boolean,s),s)` will contain a number of subclasses exponential in the arity of `f`.

In order to use product classes in practical applications the implementation of `isa/2` distinguishes a general isa relation in which a given fact may be proved using Instance Axioms, the Domain Containment Axiom (Axiom 4) and the Implicit Isa Axiom (Axiom 6) from *explicit* isa proved without the Implicit Subclassing Axiom. Based on this distinction, two restrictions are made:

1. *Restriction 1*: Axioms used to prove answers to the query `?- isa(X,Y)` depend on the instantiation of `X` and `Y`.
2. *Restriction 2*: If `immediate_isa(Id1,Id2)` is true then `Id2` is an atomic identifier.

We discuss each restriction in turn. The behavior of `isa/2` for various instantiation patterns is as follows.

1. If `Id1` and `Id2` are both ground, the Implicit Subclassing Axiom is used, if necessary.
2. If `Id1` is not ground, the Implicit Subclassing Axiom is *not* used, in order to avoid returning a number of answers that is exponential in the size of a product identifier.
3. If `Id1` is ground but not `Id2` then the Implicit Subclassing Axiom may used in the first step of a derivation. In other words, in any isa derivation for this instantiation pattern, the first step may use the Implicit Subclassing Axiom to "match" a term in the `immediate_isa/2` relation, but subsequent steps must use explicit isa. Upon backtracking the Implicit Subclassing Axiom may be used again to begin a new derivation, but subsequent steps in this derivation must cannot use this axiom.

The second assumption helps to reinforce the assumption of case 3 above. Without it, users might expect that the Implicit Subclassing Axiom could be used in each step of a derivation of an `isa/2` fact. Such an implementation would slow down the execution of `isa/2` so that it would be unusable for many applications [2].

Example 4.2.3 Suppose we have the following CDF instance.

```
isa_ext(cid(bot,s),cid(mid,s)).
isa_ext(cid(mid,s),cid(top,s)).

isa_ext(cid(prod(cid(mid,s),cid(top,s)),s),cid(myClass,s)).
```

- The query

  ```
  ?- isa(cid(prod(cid(bot,s),cid(mid,s)),s),cid(prod(cid(mid,s),cid(mid,s)),s)
  ```

  would succeed.
- ```
  ?- isa(cid(prod(cid(bot,s),cid(mid,s)),s),X)
  ```
  would successively unify `X` with

[2] Given Restriction 2, atomic identifiers usually occur within the inner loops of `isa/2`. Atomic identifiers have the advantage that these inner loops can use unification to traverse the hierarchy. If product identifers are used, they must be abstracted using `functor/3` and the hierarchies of their inner arguments traversed, a much slower method.
  ```

```
(1) cid(prod(cid(bot,s),cid(mid,s)),s),
(2) cid(prod(cid(mid,s),cid(top,s)),s),
(3) cid(myClass,s), and
(4) cid('CDF Root',cdf)
```

- The query

  ```
  ?- isa(X,cid(prod(cid(mid,s),cid(mid,s)),s))
  ```

  would unify `X` only with `cid(prod(cid(mid,s),cid(mid,s)),s)`

4.2.4 The Type-0 API

Exceptions to all predicates in this API are based on the context `query` (See Section 4.5).

`isa/2` cdf_init_cdf

The operational semantics of `isa/2` is defined in Section 4.2.3.

`explosive_isa/2` cdf_init_cdf

`explosive_isa(Sub,Sup)` follows the isa axioms for product identifiers rather than the algorithm of `isa/2`. Thus if neither `Id1` nor `Id2` are product identifiers, or if `Id1` and `Id2` are fully ground product identifiers, `explosive_isa/2` behaves as `isa/2`. Otherwise, suppose `Id1` is a (perhaps partially ground) product identifier whose Nid has the outer functor `F/A`. If the Nid of `Id2` is a variable, it is instantiated to a skeleton of `F/N`; otherwise its outer functor must be `F/A`. In either case, both Nids are broken into their constituent identifiers and `explosive_isa/2` is recursively called on each of these. `explosive_isa/2` thus removes Restriction 1 above, but not Restriction 2.

`allAttr/3` cdf_init_cdf
`hasAttr/3` cdf_init_cdf
`maxAttr/4` cdf_init_cdf
`minAttr/4` cdf_init_cdf
`classHasAttr/3` cdf_init_cdf

These predicates assume they are operating on a CDF instance, $\mathcal{O}$ in which the `isa/2` relation is acyclic. For efficiency reasons, given a goal, G, the behavior of these predicates further depends on whether various arguments of G are ground atomic identifiers.

- If either the first argument of G is a ground atomic identifier, or the second and third arguments of G are ground atomic identifiers, each answer $G\theta$ will be a member of a set S which is the most specific irredundant set containing only elements that unify with G.
- Otherwise, each answer $G\theta$ will be a member of a set S that is the most specific irredundant set containing only elements that unify with G.

`nessesCond/2` cdf_init_cdf

Given a goal `nessesCond(?Id,-Vid)`, each answer $nessesCond(Id, Vid)\theta$ will be a member of a set S which is the most specific irredundant set containing only elements that unify with `nessesCond(?Id,-Vid)`.

`isType0Term/1` cdf_checks

`isType0Term(?Term)` succeeds if `Term` unifies with an extensional fact (e.g. a term of the form `isa_ext(A,B)`, `hasAttr_ext(A,B)`, etc.), an intensional rule head (e.g. a term of the form `isa_int(A,B)`, `hasAttr_int(A,B)`, etc.), or a semantic type-0 predicate (e.g. a term of the form `isa(A,B)`, `hasAttr(A,B)`, etc.).

4.3 The Type-1 API

The Type-1 interface is radically different than the Type-0 interface. While the Type-0 interface uses tabling and logical preferences to return correct answers according to the INH proof system, it is still resolution-based. However, when the disjunction and negation of virdual identifiers is added such an approach is no longer possible, so that the Type-1 query interface is based on computing logican consistency and entailment. Since logical entailment of class expressions can be reduced to consistency, the Type-1 interface is based on consistency checking of CDF instances that have been transformed into class expressions.

Consistency checking of class expressions such as those of Definition 3.1.1 is decidable, but *P-space* complete [3], so that determening whether a Type-1 instance has a model requires radically different checking techniques than Type-0 instances. Query answering for Type-1 instances is performed by using theorem proving techniques.

4.3.1 The CDF Theorem Prover

Specialized theorem-provers are generally implemented to check consistency of class expressions. These provers may use based on structural subsumption techniques (e.g. as used in CLASSIC [PSMB+91], LOOM [MB92] and GRAIL [RBG+97]); tableaux construction [HPS99]; or stable model generation [Swi04] — in Version 1(beta) of CDF a tableaux-style prover is used.

At a high level, the CDF prover first translates a class expression CE to a formula ψ in an ontology language according to Definition 3.1.2. It then attempts to construct a model for ψ: if it succeeds CE is consistent, otherwise CE is inconsistent (since the prover can be shown to be complete). The CDF prover has access to the relational and class hierarchies of a CDF instance during its execution. As a result, only the principle classes and relations of an identifier (Definition 2.4.1) need be entered in class expressions. Finally, since objects in the semantics of CDF are indistinguishable from singleton sets, an object identifier O can be used in any context that a class identifier can be used. The prover takes accont of this by enforcing a cardinality constraint for the set O.

The theorem prover of Version 1(beta) uses exhaustive backtracking, rather than the dependency-directed backtracking that is typical of recent provers such as the DLP prover [], the FaCT prover [] or the Racer prover []. As a result, the CDF prover may be slow for certain types of queries relative to these other provers. Dependency-directed backtracking has not been added to the CDF prover largely to keep it simple enough to experiment with different extensions to the types of

[3] Assuming a linear encoding of the integers in the *atLeast* and *atMost* constructs. Formally, the CDF prover is complete for $\mathcal{ALCQ}$ description logics extended with relational hierarchies and product classes.

class expressions it supports [4]. On the other hand, the CDF prover is relatively efficient on how it traverses a CDF instance to check consistency.

When a CDF Type-1 instance is checked, the instance is translated into either a class expression before it can be sent to the CDF prover. Due to the high worst-case complexity of consistency checking, input strings to the prover should be kept as small as possible. The CDF system accomplishes this by translating information about a given CDF identifier into a series of local class expressions (Section 3.3), sending a local class expresion to the CDF prover, then producing and checking other local class expressions as needed. Since CDF instances differ in philosophy from terminological systems, they may be expected to be cyclic, so that a given class identifier may occur in a level n local class expression of itself.

4.3.2 The Type-1 API

`checkIdConsistency/1`

cdftp_chkCon In `checkIdConsistency(IdList)` `IdList` is a (list of) class or object identifier(s) which is taken as a conjunction. The predicate succeeds if `IdList` is consistent in the current CDF instance.

Exceptions `Domain Exception:` `IdList` is not a class identifier, an object identifier, or a list of class or object identifiers.

`consistentWith/2` cdftp_chkCon

In `consistentWith(Id,CE)`, `Id` can either be a class or an object identifier and `CE` is a class expression. This predicate checks whether `CE` is logically consistent with all that is known about `Id` in the current CDF instance. `consistentWith/2` determines whether there is a model of the current CDF instance that satisfies the expression `Id,CE`.

This predicate assumes that all class and object identifiers in a given CDF instance are consistent.

Exceptions

`Domain Exception:` `Id` is not a class or object identifier.

`Domain Exception:` `CE` is not a well-formed class expression.

`allModelsEntails/2` cdftp_chkCon

In `allModelsEntails(Id,CE)`, `Id` is a class or object identifier and `CE` is a class expression. `allModelsEntails/2` succeeds if `CE` is entailed by what is known about `Id` in the current CDF instance. In other words, `allModelsEntails/2` determines whether in all models of the current CDF instance, if an element is in `Id` then it is also in `CE`. It does this by checking the inconsistency of `Id,CE`.

This predicate assumes that all class and object identifiers in a given CDF instance are correct.

Exceptions

`Domain Exception:` `Id` is not a class or object identifier.

[4] In particular, work is underway on extending the CDF prover to handle functional attribute chains and concrete domains (see []).

`Domain Exception:` `CE` is not a well-formed class expression.

`localClassExpression/3` cdftp_chkCon

In `localClassExpression(+IdList,+N,-Expr)` `IdList` is a list of class identifiers, and `N` is a positive integer. In its semantics, `IdList` is interpreted as a conjunction of identifiers, and upon success, `Expr` is a class expression, unfolded to depth `N`, that describes `IdList` according to gthe current CDF instance.

Exceptions

`Domain Exception:` `IdList` is not a class identifier or object identifier, or a list of class or object identifiers.

`Type Exception:` `N` is not a positive integer.

`Instantiation Exception:` `Expr` is not a variable.

`check_lce/2` cdftp_chkCon

In the goal `check_lce(+IdList,+N)` `IdList` is a list of class identifiers, and `N` a positive integer. In its semantics, `IdList` is interpreted as a conjunction of identifiers, and `check_lce(+IdList,+N)` pretty-prints a class expression, unfolded to depth `N`, that describes `IdList` according to the current CDF instance.

Exceptions

`Domain Exception:` `IdList` is not a class identifier or object identifier, or a list of class or object identifiers.

`Type Exception:` `N` is not a positive integer.

4.4 Updating CDF Instances

Both extensional facts and intensional rules are dynamic, so that they can be asserted or retracted. Any attempt to directly assert or retract to a CDF instance, however, will almost certainly lead to disaster. When the state of a CDF instance changes, tables that support the Type-0 interface may need to be abolished; consistency checks may need to be rerun; various components may need to be marked as dirty (so that they will be written out the next time the component is updated); the update logged to support future merges; and the XJ cache may need to be updated, and so on. The Update API supports all the tasks that need to be done when a CDF instance changes.

Abolishing tables for the Type-0 interface, can be seen as an instance of the *table update problem*. This problem can occur when a table depends on dynamic information. When the dynamic information changes the table may become out-of-date. In principle, this problem might be addressed by changing the tables themselves – adding or deleting answers as needed. However, relating specific answers to dynamic information is not always easy to do, (as when recursive predicates are tabled). As an alternative various tables may be abolished when there is a danger that they are based on out-of-date information. Using this approach, one can abolish entire tables for certain queries, or all tables for certain predicates.

CDF takes the simple but sound solution of abolishing all calls to a tabled system predicate whenever that predicate might depend on updated dynamic information. Such dependencies are most common for the Type-0 API. Various tables are used to compute the `xxxAttr` relations and `isa/2` (when `isa/2` is tabled). If, say, a `hasAttr_ext/3` fact is added, then the `hasAttr/3` tables should be abolished: otherwise the tables will represent outdated information. Similarly, if a change is made to an `isa/2` extensional fact or intensional rule, all of the tables used to compute the `xxxAttr` predicates must be abolished, as well as any tables used to compute `isa/2` itself.

A more difficult problem can arise when intensional rules are added or deleted. If the intensional rule depends on tabled predicates, and the tabled predicates themselves depend on Type-0 predicates, the tabled predicates supporting the rule must be abolished. To address this when a CDF intensional rule R is updated, CDF must be informed of those Type-0 predicates upon which tables in R depend using the predicate `addTableDependency/2` defined below.

4.4.1 The Update API

The following predicates form the update API for CDF. No exceptions are listed for these predicates: rather a user can chose the semantic checks she wants by adding checks to various contexts, or removing them, as described in Section 4.5.

`newExtTerm/1` cdf_init_cdf

`newExtTerm(+Term)` is used to add a new extensional fact to the CDF instance. This predicate applies those consistency checks that have been specified for addition of a single extensional facts (see Section 4.6 for the default checks, and Section 4.5 for a description of the checks themselves). It then logs the fact that `Term` has been asserted, marks the component to which `Term` belongs as *dirty*, invalidates the XJ cache, and finally abolishes the appropriate tables.

Exceptions: based on the context `newExtTermSingle`

`retractallExtTerm/1` cdf_init_cdf

`retractallExtTerm(?Term)` retracts all extensional CDF facts that unify with `Term`. This predicate applies those consistency checks that have been specified for retraction of extensional facts unifying with a term (see Section 4.6 for the default checks, and Section 4.5 for a description of the checks themselves). It then logs the fact that the facts unifying with `Term` have been retracted, marks the components to which those facts belong as *dirty*, invalidates the XJ cache, and finally abolishes the appropriate tables. Note that this operation does not affect information derived via intensional rules, and may not affect information derived via inheritance.

Exceptions: based on the context `retractallExtTermSingle`

`newIntRule/2` cdf_init_cdf

`newIntRule(+Head,+Body)` is used to add a new intensional rule to the CDF instance. This predicate applies those consistency checks that have been specified for addition of a single intensional rule (see Section 4.6 for the default checks, and Section 4.5 for a description of the checks themselves. It then logs that the rule has been asserted, marks the component to which the rule belongs as *dirty*, invalidates the XJ cache, and finally abolishes the appropriate tables.

`retractallIntRule/2` cdf_init_cdf

`retractallExtTerm(?Head,?Body)` retracts all intensional rules for which `clause(Head,Body)` is true. This predicate applies those consistency checks that have been specified for retraction of rules (see Section 4.6 for the default checks, and Section 4.5 for a description of the checks themselves. Note that this operation does not affect information derived via extensional facts, and may not affect information derived via inheritance.

`addTableDependency/2` cdf_init_cdf

`addTableDependencies(+TableList,+DependencyList` informs CDF that each table in `TableList` depends on all dynamic predicates in `DependencyList`. Both lists use predicate indicators (i.e. terms of the form `Functor/Arity`, see the XSB manual) and the predicate specifiers in `DependencyList` must be of type `isDynSupportedPred/1`. These dependencies ensure that tables for predicates in `TableList` are abolished whenever extensional facts or intensional rules for tables in `DependencyList` are updated.

`isDynSupportedPred/1` cdf_init_cdf

The goal `isDynSupportedPred(?Pred)` succeeds if `Pred` unifies with a specifier for a predicate defined to be the Type-0 API, currently `isa/2`, `allAttr/3`, `hasAttr/3`, `maxAttr/4`, `minAttr/4`, `classHasAttr/3` or `necessCond/2`.

`abolish_cdf_tables/0` cdf_init_cdf

This predicate abolishes all tables used by CDF. It does not reinitialize any other aspects of the CDF system state as does `initialize_state/0` (see Section 4.6.1).

4.5 Semantic Checking

Enforcing the semantic axioms of Chapters 2 and 3 is critical to developing ontologies. In the first place, enforcing a correct semantics is, in the long run, critical for user confidence; in the second place, enforcing axioms at a system level eases the coding of higher layers of CDF functionality and of applications. At the same time, semantic checking can slow down performance if done redundantly or unnecessarily.

CDF provides several predicates for semantic checking, as well as different methods to adjust the amount of checking done during execution of system or user code. Semantic checking in system code can be adjusted by the predicicates `addCheckToContext/2` and `removeCheckToContext/2`, which determine which checks are to be performed at various *contexts* during the course of managing an ontology. The checking predicates themselves also can be called directly, and users can even create their own contexts. To explain the consistency checking system, we first discuss the type of checking predicates that are provided, then discuss system contexts, and finally how a user can create his or her own contexts for application code.

4.5.1 Classes of Semantic Checking Predicates

The semantic checking predicates may be categorized in different ways. An obvious distinction is between Type-1 consistency checking and the simpler semantic checks that make use of Type-0 axioms or of INH proofs. The main Type-1 consistency checking predicate is `checkIdConsistency/1` discussed in Section 4.3. We discuss here the simpler semantic checking for (portions of) Type-0 Axioms. Detailed information about these predicates are discussed in Section 4.5.4.

Since well-sorting is a necessary condition for the consistency of Type-0 (and Type-1) instances, certain of the checking predicatess concern themselves with ensuring that given facts are well-sorted. These include the predicates `cdf_check_ground/1`, and `cdf_check_sorts/1`. The first checks that a term is ground (modulo the exception for identifiers whose component tag is `cdfpt`); the second checks that an extensional fact is well-sorted.

Other predicates perform checks to ensure that a CDF instance does not have "redundant" extensional facts. As notation, let us write an extensional fact as $F\_ext\theta$ where F_ext is the predicate skeleton (i.e. `hasAttr_ext(A,B,C)`, `allAttr(A,B,C)` etc.) and θ is the binding to the variables of the skeleton. One predicate that checks for redundancy is `cdf_check_implication(`$F\_ext\theta$`)` which takes an extensional fact as input and checks whether $F\theta$ holds in the current CDF instance. For example if the extensional fact were `hasAttr_ext(A,B,C)`θ the predicate would check whether `hasAttr(A,B,C)`θ held. A weaker predicate is `cdf_check_identity/1` which checks for $F\_ext\theta$ whether `imnmed_`$F\theta$ holds in the current instance. In other words, if the extensional fact were `hasAttr_ext(A,B,C)`θ the predicate would check whether `immed_hasAttr(A,B,C)`θ held. Working at a somewhat broader level, the predicate `check_redundancies(Component)` performs the same check as `check_identity/1` (in a somewhat optimized manner) for each extensional fact in `Component`, removing any redundant facts (see Section 4.7 for assignment of facts to components).

If a check takes as input an extensional fact it is called a *fact-level check*; if it takes as input a component tag it is called a *component-level check*. The type of input affects the contexts where a check can be used, as we now describe.

4.5.2 System Contexts

The available system contexts for Version 1(beta) of CDF are listed.

- Contexts where fact-level checks can be applied and are of type `factLevel`.
  - `query`. This context occurs when a query is made using the Type-0 API.
  - `newExtTermSingle`. This context occurs when an attempt is made to add a new extensional fact to the CDF store by `newExtTerm/1`, as occurs for instance when facts are added using the CDF editor.
  - `newExtTermBatch` This context occurs when an attempt is made to add a given extensional term to the CDF store during component load by `load_component/3`, or file load by `load_extensional_facts/1`.
  - `retractallExtTermSingle`. This context occurs when an attempt is made to retract an extensional fact from the CDF store by `retractExtTerm/1`, as occurs for instance when facts are removed using the CDF editor.
- Contexts where component-level checks can be applied, and are of type `componentLevel`.

- `componentUpdate` This context occurs when the predicate `update_all_components/3` is called to save or move a component.
- `loadComponent` This context occurs when the predicate `load_component/3` is called to load a new component.

The checks are associated with a given context by default in `cdf_config.P` or in the user's `xsbrc.P` file. However at any point in a computation, a user may associate a given check to a context or dissasociate the check from the context using `addCheckToContext/2` and `removeCheckFromContext/2`.

In addition, certain of the semantics checking predicates may be useful outside of contexts, in order to check that a CDF fact is properly ground or well-sorted. To support this, there is a "dummy" context, `usercall` that is displayed in warnings and errors in these cases.

4.5.3 Adding User Contexts

At an operational level, a context is executed whenever the predicate `apply_checks(+Context,Argument)` is called, where `Argument` is either an extensional fact or a component tag, depending on the type of the context. This predicate determines which checks are associated with `Context` and ensures that all of the checks are called. Thus, the first step in adding a context to user code is to determine what point (or points) checks should be made, and figure out a name for the context.

In order to ensure that the checks are performed properly, the context must be made known to CDF. This is done by the predicate `addUserContext(+Context,+Type)` which informs CDF that a context of a given type is added. The context may be removed by `removeUserContext(+Context)`.

TLS ADD SOMETHING ABOUT PERSISTANCE FOR CDF FLAGS.

4.5.4 Semantic Checking Predicates

`cdf_check_ground/1` cdf_checks

`cdf_check_ground(+Term)` requires that `Term` be in the domain `isType0Term/1`, and checks that each argument `A` of `Term` is a CDF identifier and that each identifier occurring in `A` is either ground or has the component tag `cdfpt`. If not, an instantiation error is thrown.

Exceptions

`Domain Exception:` `Term` is not in the domain `isType0Term/1`.

`Instantiation Exception:` `Term` has a non-ground identifier, not in `cdfpt`.

`cdf_check_sorts/2` cdf_checks

`cdf_check_sorts/1` cdf_checks

`cdf_check_sorts(+Context,+Term)` requires that `Term` be in the domain `isType0Term/1`. The predicate checks that each argument `A` of `Term` is a CDF identifier that obeys the sorting constraints of the instance axiom for `Term`, as well as the Downward Closure Axiom for Product Identifiers (Axiom 5). If `Term` is not in the domain `isType0Term/1` or if it is not well-sorted, a warning is written out to the XSB messages stream indicating that the predicate failed during `Context` and the predicate fails. `cdf_check_sorts(+Term)` behaves similarly, with the context set to `usercall`.

`cdf_check_implication/1` cdf_checks

`cdf_check_implication(+Term)` requires that `Term` be in the domain `isType0Term/1`. The predicate silently fails if the corresponding Type-0 predicate is derivable. For instance, if `Term` were equal to `hasAttr_ext(C1,R1,C2)`, then the predicate fails if `hasAttr(C1,R1,C2)` holds in the current CDF instance.

`cdf_check_identity/1` cdf_checks

`cdf_check_identity(+Term)` silently fails if `ExtTerm` has been asserted, or if the arguments of `ExtTerm` can be derived by a corresponding intensional rule; otherwise it succeeds. For instance, if `ExtTerm` were equal to `hasAttr_ext(C1,R1,C2)`, then the predicate fails if `ExtTerm` were already in the CDF instance or if `hasAttr_int(C1,R1,C2)` were derivable. As it does not make use of inheritance rules, this predicate can much faster than `cdf_check_implication/1`

`cdf_check_redundancies/3` cdf_checks

If the mode in `cdf_check_redundancies(+Context,+Component,+Mode)` is `retract`, this predicate backtracks through each extensional fact $F\_ext\theta$ (see Section 4.5.1) that are associated with `Component` and removes $F\_ext\theta$ if $F\theta$ is INH-implied by other extensional facts or intensional rules. Otherwise, if `Mode` is set to `warn`, rather than removing redundant facts, a warning is issued for each fact found to be redundant.

`checkComponentConsistency/1` cdf_checks

`checkComponentConsistency(+Component)` ensures that each class or object identifier associated with `Component` has a consistent definition using the CDF theorem prover.

`addCheckToContext/2` cdf_checks

`addCheckToContext(+PredIndicator,+Context)` ensures that the check `PredIndicator` (i.e. a term of the form `F/N`) will be performed when executing any context that unifies with `Context`. If `PredIndicator` has already been added to `Context`, `addCheckToContext/2` succeeds.

Exceptions:

`instantiation_error` `PredIndiecator` or `Context` is not instantiated at time of call.

`misc_error` `Predindicator` is not a predicate indicator.

`domain_error` File `Context` is not currently a context.

`misc_error` `Context` and `Predindicator` use different context types.

`removeCheckFromContext/2` cdf_checks

`removeCheckToContext(?PredIndicator,?Context)` ensures that any checks that unify with `PredIndicator` (i.e. a term of the form `F/N`) will *not* be no longer performed when executing `Context`. If `PredIndicator` is not associated with `Context`, `removeCheckFromContext/2` succeeds.

`addUserContext/2` cdf_checks

`addUserContext(+Context,+Type)` adds a context named `Context` of type `Type`. This action must be performed before any checks can be added to `Context`. If `Context` has already been added with the same type the predicate succeeds, otherwise it throws a permission error.

Exceptions:

`instantiation_error` `Context` or `Type` is not instantiated at the time of call.

`domain_error` File `Type` is not in domain `isContextType`

`misc_error` `Context` is a system context.

`misc_error` `Context` has already been added, with a type different than `Type`.

`removeUserContext/1` cdf_checks

`removeUserContext(?Context)` removes any contexts unifying with `Context` from CDF. If `Context` is not currently a context, the predicate succeeds.

`apply_checks/2` cdf_checks

`apply_checks(+Context,+Argument)` applys any checks that have currently been added for `Context` to `Argument`. If `Argument` is not of the right type, handling this error is left to the checks for `Context`, which may fail, emit warnings, throw errors, etc.

`isContextType/1` cdf_checks

`isContextType(?Type)` defines the context types for Version 1(beta) of CDF.

`currentContext/3` cdf_checks

`currentContext(?Context,?Mode,?Type)` defines the contexts for the current CDF state, their "mode" (i.e. system or user) and their type.

4.6 Configuring and Examining the CDF System

The file `cdf_config.P` contains all facts, tables etc. that may need to be configured for a particular user or application. In addition to the predicates described below, `cdf_config.P` contains a fact for the dynamic predicate `default_user_error_handler/1` which is used to handle errors. See the XSB manual for documentation of this predicate.

4.6.1 The Configuration API

`cdf_configuration/2` usermod

`cdf_configuration(?Flag,?Value)` allows access to configuration parameters for CDF, and is analogous to `xsb_configuration/2` These parameters are set during the compilation of CDF by define statements in `cdf_definitions.h`. Currently, the only parameters that can be set are

- `using_xj`. This parameter is set `on` if XJ is to be supported by CDF, and set off otherwise. This parameter is set on by default in the proprietary version of CDF and off by default in the open-source version.

- `tabled_isa`. This parameter is set on if `isa/2` is tabled, and off otherwise. It is set on by default.

`cdf_index/3` usermod

`cdf_index(+Functor,+Arity,+Index)` is used to set initial indices for the various types of extensional facts. These indices can be changed, if necessary to give better performance, but note that the semantics of predicates in the Type-0 API may not use all indices. See Section 4.2.4 for details.

`initialize_state/0` usermod

Normally, initialization is done automatically upon loading CDF at the start of a session. The predicate `initialize_state/0` should be called only when a state is to be reinitialized during a session. This predicate removes all data in extensional and intensional forat, and reasserts the basic CDF classes and relations, and resets internal state variables to values in the CDF configuration file.

4.7 CDF Components and I/O

4.7.1 CDF Components

Typically, a CDF instance can be partitioned into several separate cells representing information that arises from different sources. For instance, in the example from the previous chapter, the taxonomy shown in Example 2.1.1 is largly drawn from the UNSPSC taxonomy, while the relations and domains of Example 2.1.2 and later are adapted from DLA's FIIG meta-data. Practical systems often need to update data from different sources separately, and may need to incorporate information from one source independantly from that of another. The CDF components system attempts to address this need by allowing ontologies to be built from discrete *components* that can be maintained separately.

First of all, the CDF component system provides an implicit partition of all facts and rules in a CDF instance. Recall from Defintion 4.1.1 all identifiers have an outer 2-ary functor whose component tag or *source* is in the second argument. Extensional facts in a CDF instance can be assigned a component by choosing a *component argument* for each predicate symbol, and assigning as the component of each fact the component tag of the identifier in the component argument. If the identifier is a product identifier, the source is the source of the outer function symbol). Intensional rules can be assigned a component in a similar way, by assigning a component argument to each intensional rule predicate symbol and requiring that the component tag of the component argument in each intensional rule be ground.

By default, CDF uses a *relation-based component system*, which chooses as argument identifiers

- the first argument of all `isa_ext/2` and `necessCond_ext/2` facts, along with `isa_int/2` and `necessCond_int/2` rules.
- the second argument of all `hasAttr_ext/3`, `allAttr_ext/3`, `minAttr_ext/4`, `maxAttr_ext/4` and `classHasAttr_ext/3` facts and their associated `_int` rules.

Given such a partition, a dependency relation between components arises in a natural way. Let $Id : components.tex, v1.12005/04/1223 : 48 : 59tswiftExp$ range over CDF identifier functors. A component C_1 *directly depends* on component C_2 if $Id(Nid_2,C_2)$ is an argument in a fact or rule head in component C_1. In addition, C_1 durectly depends on C_2 if $Id(Nid_p,C_1)$ is a product identifier in a component argument, and $Id(Nid_2,C_2)$ occurs as an argument in Nid_p. It is easy to see that component dependency need not be hierarchical so that two components may directly depend on each other; furthermore each component must directly depend on itself. Component dependency is defined as the transitive closure of direct dependency.

Dependency information is used to determine how to load a component and when to update it and is usually computed by the CDF system. Computing dependency information is easy for extensional facts, but computing dependency information for intensional rules is harder, as the component system would need to compute all answer substitutions to determine all dependencies, and this in impractical for some sets of intensional rules. Rather, dependencies are computed by checking the top-level arguments of intensional rules, which leads to an under-approximation of the dependencies.

Component Names, Locations, and Versions

A component is identified by a structured *component name*, which consists of a *location* and a *source*. For example, information in the directory `/home/tswift/unspsc` would have a location of `/home/tswift` and source `unspsc`. For efficiency, only the source is used as a source argument for identifiers within a CDF instance; the location is maintained separately. The structuring of component names has implications for the behavior of the component system. If two components with the same sources and different locations are loaded, facts and rules from the two different components cannot be distinguished, as only the source is maintained in identifiers. The attempt to load two such components with the same source and different locations can be treated as an error; or the load can be allowed to succeed unioning the information from both components, implicitly asserting an axiom of equality for the two (structured) component names.

The same component name can have multiple *versions*. An CDF state can contain only one version for each component name, and an attempt to load two different versions for the same name always gives rise to an error. On the other hand, if two component names C_1 and C_2 have the same source and different locations, they do not have the same name. If they are loaded so that their information is unioned together an error is not raised if C_1 and C_2 have different versions.

4.7.2 I/O for CDF

The component system allows users to create or update components from a CDF state, as well as load a component along with its dependencies. In the present version of CDF, it provides a basis for concurrent editing of ontologies by different users sharing the same file system (see Section 4.9). Users can either load the latest version of a component, or work with previous versions, or move their work to a separate directory where they can work until they feel it is time to merge with another branch. While the current version of CDF only supports loading and saving components to a file system, work is underway to load and save components as RDF resources, using translators between XSB and RuleML [Rul03] [5].

At a somewhat lower level, the CDF I/O system allows a user to load or dump a file of extensional facts or intensional rules The I/O system also forms the building blocks of the component system.

4.7.3 Component and I/O API

Component API

`load_component/1` cdf_comps_share

`load_component(Name,Loc,Parameter_list)` loads the component `Name`, from location `Loc` and recursively, all other components upon which the component depends. If a version conflict is detected between a component tag to be loaded and one already in the CDF state or about to be loaded, `load_component/3` aborts without changing CDF extensional facts or intensional rules. In Version 1(beta) of CDF, `Loc` must be a file system path, and the rules for filename expansion of relative and other paths is the same as in XSB.

The order of loading is as follows. First, all extensional facts are loaded for the component `Name` at location `Loc`, and then recursively for all components on which `Name` at `Loc` depends. Next, the dependency graph is re-traversed so that intensional rules are loaded and initialization files are consulted in a bottom-up manner (i.e. in a post-order traversal of the dependency graph).

`Parameter_list` may contain the following elements:

- `action(Action)` where `Action` is `check` or `union`. If the action is `check`, two components with the same component but different locations or versions cannot be loaded: an attempt to do so will cause an error. If the action is `union`, two components with the same name and different locations may be loaded, and the effect will look as if the two components had been unioned together. However an error will occur if two components with the same name and location but different versions are loaded.
- `force(Bool)` where `Bool` is `yes` or `no` (default `no`). If `Force` is `yes`, any components that have previously been loaded into the CDF are reloaded, and their initialization files reconsulted. If `Force` is `no`, no actions will be taken to load or initialize components already loaded into the CDF.
- `version(V)` where `V` is a version number. If the parameter list contains such a term, the loader attempts to load version `V` of component. The default action is to load the latest version of a component.

`update_all_components/2` cdf_comps_noshare

`update_all_components(Loc,Option_list)` analyzes components of a CDF state and their dependencies, determining whether they need to be updated or not, and creating components when necessary. When a component is created with location `Loc`, the files `cdf_extensional.P` and `cdf_intensional.P` are created. Initialization files must be added manually for new

[5] These translators were written by Carlos Damásio.

components. Currently, `Dir` must specify a file system directory. `Option_list` contains a list of parameters which currently specifies the effect on previously existing components:

- `action(Action)`.
  - If `Action` is `create`, then a new set of components is created in `Loc`. Information not previously componentized is added to new components whose location is `Loc` and whose version number is 0. Facts that are parts of previously created components are also written to `Loc`; if their previous location was `Loc`, their versions are updated if needed (i.e. if any facts or rules in the component have changed). Otherwise, if the location of a previously created component `C` was not `Loc`, `C` is written to `Loc` its location information updated, and its version is set to 0.
  - If `Action` is `in_place`, then components are created in `Loc` only for information that was not previously componentized. Newly created components are written to `Loc` which serves as their locations, and their version number is 0. Previously created components whose locations were not `Loc` are updated using their present location, if needed.

  In either case all dependency information reflects new component and location and version information.

I/O API

`load_extensional_facts/1` cdf_io

`load_extensional_facts(DirectoryList)`: loads the file `cdf_extensional.P` from directories in `DirectoryList`. The files loaded must contain extensional data. `load_extensional_facts/1` does not abolish any extsnsional information already in memory; rather, it merges the information from the various files with that already loaded. Intensional rules will not be affected by this predicate.

`load_intensional_rules/1` cdf_io

`load_intensional_rules(Dir)` ynamically loads intensional rules from `cdf_intensional.P` in `Directory`. This predicate is designed for the component system, but can be used outside of it. The leaf directory name in `Dir` is assumed to be the component name of the rules. As the intensional rules are loaded, their functors are rewritten from `XXX_int` to `XXX_int_Name`, to avoid any conflicts with intensional rules loaded from other components or directories.

`merge_intensional_rules/0` cdf_io

`merge_intensional_rules/0`: This utility predicate takes the current intensional rules for all sources and transforms them to extensional form by backtracking through them, and asserting them to the Prolog store. All intensional information is then retracted.

`dump_extensional_facts/1` cdf_io

`dump_extensional_facts(Dir)` writes extensional facts to the file `cdf_extensional.P` in `Directory`. No intensional rules are dumped by this predicate. `dump_extensional_facts/0` writes the `cdf_extensional.P` file to the current directory.

`dump_intensional_rules/1` cdf_io

`cdf_exists/1` cdf_io

`cdf_exists(Dir)` checks whether `cdf_extensional.P` file is present in directory `Dir`.

4.8 Database Access for CDF

4.8.1 Storing a CDF in an ODBC database

The 4 CDF relations, while usually stored in prolog .P files, may also be stored in 4 relations in a relational database. Each relation is stored in its external form. Each field (except an object ID) is stored as a Prolog Term. Object ID's are stored as strings.

These routines allow the loading of a CDF into memory from an ODBC database and the dumping of an CDF from memory to an ODBC database.

In the future they will allow all updates to such a CDF in memory to be immediately reflected back out to the stored DB.

4.8.2 Lazy Access to an CDF Stored in a RDB

A facility is also provided to lazily access an CDF stored externally as 4 relations in external form in a relational database.

`clear_db_cdf/1` `clear_db_cdf(+Connection)`

deletes all tuples in the CDF tables in the database accessible through the database connection named `Connection`, which must have been opened with odbc_open/1.

`clear_db_cdf_component/2` `clear_db_cdf(+Connection,+Component)`

deletes all tuples for component `Component` in the CDF tables in the database accessible through the database connection named `Connection`, which must have been opened with odbc_open/1.

`drop_db_cdf/1` `drop_db_cdf(+Connection)`

removes the tables used to dump a CDF to a database. Connection must be an opened ODBC connection (using odbc_open/4).

`drop_db_cdf_component/2` `drop_db_cdf(+Connection,+Component)`

removes the tables used to dump a CDF `Component` to a database. Connection must be an opened ODBC connection (using odbc_open/4).

`create_db_cdf/1` `create_db_cdf(+Connection)`

creates the 4 tables necessary to dump a CDF to a database. Connection must be an opened ODBC connection (using odbc_open/4).

`create_db_cdf_indices/1` `create_db_cdf_indices(+Connection)`
creates the appropriate indices for the tables necessary to dump a CDF to a database. `Connection` must be an opened ODBC connection (using odbc_open/4).

`create_db_cdf_component/2` `create_db_cdf_component(+Connection,+Component)`
creates the tables necessary for dumping a given CDF *Component* into a database. Connection must be an opened ODBC connection (using odbc_open/4).

`create_db_cdf_component_indices/2` `create_db_cdf_component_indices(+Connection,+Component)`
creates the appropriate indices for the tables necessary to dump a CDF `Component` to a database. `Connection` must be an opened ODBC connection (using odbc_open/4).

`load_db_cdf/1` `load_db_cdf(+Connection)`
loads an CDF from tables stored in a database in external form. Connection must be an opened ODBC connection created by odbc_open/4.

`dump_db_cdf/1` `dump_db_cdf(+Connection)`
dumps a CDF into a database. It assumes the necessary tables have been created by `create_db_cdf/1`. `Connection` must be an opened ODBC connection (using odbc_open/4).

`dump_db_component/3` `dump_db_component(+Connection,+Component,+Path)`
dumps a CDF `Component` into appropriate tables in a database. It assumes the tables have been created by a call to `create_db_cdf_component/2`. `Connection` must be an opened ODBC connection (using odbc_open/4). Component information is created in the `Path`. This includes necessary initialization files as well as static dependencies obtained from the current dependencies for the Component in CDF. The created files should be patched to contain appropriate path information for dependencies.

`cache_abstract/3` `cache_abstract(Connection, CallTemplate, AbstractCallTemplate)`
is a dynamic predicate which can be used to control call abstraction, thus implementing a form of cache prefetching.

4.8.3 Updatable External Object Data Sources

Updatable External Object Data Sources provide a way for CDF object data (objects and their attributes) to be stored and retrieved from external tables within a relational database that is accessible through the ODBC interface. Objects can be created and deleted and their attributes can be added, deleted and updated. The table(s) in the (external) relational database are updated to reflect such changes.

An Updatable External Object Data Source is stored as a ""component"" Section 4.7, which is identified by a particular, unique, *Source*.

The external database table(s) of an Updatable External Object Data Source must be of specific form(s), and represent objects and their attributes in particular ways. However, the ways supported are general enough to allow reasonably natural external data representations.

The main table in an Updatable External CDF Object Data Source is called the *Object Table*. An object table is a database table whose rows represent objects to be seen within a CDF. Such a set of objects will share the same ""source"" in the CDF, indicating their ""component"".

An Object Table contains a column which is the CDF object Native ID, and that column must be a key for the object table. The table may have other columns that can be reflected as CDF attributes for the objects. Each such attribute must (clearly) be functional. There may also be other tables, called Attribute Tables, which have a foreign key to the object table, and a column that can be reflected as a CDF attribute for the object. These attributes need not be functional.

An Object Table must be declared with the following fact.

```
ext_object_table(Source,TableName,NOidAttr,NameAttr,
 MemberofNCid,MemberofSou).
```

where:

- `Source` is the component identifier for this object table.
- `TableName` is the name of the object table in the database.
- `NOidAttr` is the column name of the key column of the object table.
- `NameAttr` is the column name of field that contains the name field for the object. (If there is no special one, it should be the same as `NOidAttr`.)
- `MemberofNCid` determines the Native ID of the classes of which the objects are members. If it is an atom, then it is the name of a column in the table whose values contain the Native ID for the class containing the corresponding object. If it is of the form `con(Atom)`, then `Atom`, itself, is the Native ID of the single class that contains *all* objects in the table.
- `MemberofSou` determines the Sources of the classes containing the objects in the table. If it is an atom, then it is the name of a column in the object table whose values are the sources of the classes containing the corresponding objects. If it is of the form `con(Atom)`, then `Atom`, itself, is the Source for all classes containing objects in the table. (Note if `memberofNCid` is of the form con(_), then `MemberofSou` should also be of that form.)

The caller must have previously opened an ODBC connection, named Source, by using `cdf_db_open/3`, before these routines will work.

For each functional attribute represented in an Object Table, there must be a fact of the following form:

```
ext_functional_attribute_table(Source,RelNatCid,RelSou,
                                        TarAttr,Trans).
```

where:

- `Source` is the component identifier for this object table.
- `RelNatCid` is the Native ID of the CDF relationship for this attribute.
- `RelSou` is the Source of the CDF relationship for this attribute.
- `TarAttr` is the name of the column(s) in the table containing the value of this attribute. If the internal target type is a product type, then this is a list of the column names of the columns that contain the product values. The predicate `coerce_db_type/5` converts from internal Native Ids to (and from) (lists of) data field values. Rules for `coerce_db_type/5` are provided to do standard (Trans = std) conversion between primitive CDF and database types. Product types are unfolded to be a list of primitive CDF and database types and are converted as such. If desired, `coerce_db_type/5` could be extended to include special-purpose conversion methods (probably only of interest for special product types.)
- `Trans` is an atom that indicates the type of translation from internal to external format. Normally it is '`std`', unless `coerce_db_type/5` has been extended to include special translation capabilities.

There must be a `schrel` in the CDF for each of these CDF relationships indicating the CDF type of the attribute value.

For each attribute table, there must be a fact of the following form:

```
ext_attribute_table(Source,TableName,NOidAttr,
                    RelationNatCid,RelationSou,TarAttr,Trans)
```

where:

- `Source` is the component identifier for this object table.
- `TableName` is the name of the attribute table in the database.
- `NOidAttr` is the column name of the column of the attribute table which is a foreign key to the object table.
- `RelationsNatCid` is the Native ID of the CDF relationship for this attribute.
- `RelationSou` is the Source of the CDF relationship for this attribute.
- `TarAttr` is the name(s) of the column(s) in the table containing the value(s) of this attribute. It is an atomic name if the value is of a primitive CDF type; it is a list of names if the value of this attribute is of a product type.
- `Trans` is an atom that indicates the type of translation from internal to external format. Normally it is `std`.

For each functional `attribute_object`, there must be a fact of the following form:

```
ext_functional_attribute_object_table(Source,RelationNatCid,RelationSou,
                                             TarAttr,TarSource)
```

where, as above

- `Source` is the component identifier for this object table.
- `RelationsNatCid` is the Native ID of the CDF relationship for this attribute.
- `RelationSou` is the Source of the CDF relationship for this attribute.
- `TarAttr` is the name of the column in the table containing the value of a native object ID.
- `TarSource` is the Source of the native Oids in the TarAttr field.

For each attribute_object table, there must be a fact of the following form:

```
ext_attribute_object_table(Source,TableName,NOidAttr,
                    RelationNatCid,RelationSou,TarAttr,TarSource)
```

where, once again:

- `Source` is the component identifier for this object table.
- `TableName` is the name of the attribute table in the database.
- `NOidAttr` is the column name of the column of the attribute table which is a foreign key to the object table.
- `RelationsNatCid` is the Native ID of the CDF relationship for this attribute.
- `RelationSou` is the Source of the CDF relationship for this attribute.
- `TarAttr` is the name of the column in the table containing the value of a native object ID.
- `TarSource` is the Source of the native Oids in the TarAttr field.

`cdf_db_open/3` `cdf_db_open(Component,CallToGetPar,Parameter)`
opens an odbc connection to a database for use by cdf_db_updatable, or cdf_db_storage, predicates. `Component` is an atom representing the component; `CallToGetPar` is a callable term, which will be called to instantiate variables in `Parameters`. If `Parameters` is given as a ground term, then `CallToGetPar` should be `true`. It can be used, for example, to ask the user for a database and/a password. `Parameter` specifies the necessary information for odbc_open to open a connection. It is one of the following forms: `odbc(Server,Name,Passwd)` or `odbc(ConnectionString)`. See the `odbc_open/1/3` documentation [6] for details on what these parameters must be.

[6] See Volume 2 of the XSB manual.

`isa_int_udb/2` `isa_int_udb(?Arg1,?Arg2)`
is used to provide access to database-resident `isa_ext` facts. It is typically used in the definition of `isa_int/2` in `cdf_intensional.P` files for db_updatable components.

`assert_cdf_int_udb/1` `assert_cdf_int_udb(+Term)`
is used to assert a `Term` in a db_updatable component. It is typically used in the definition of `assert_cdf_int` for db_updatable components.

`retractall_cdf_int_udb/1` `retractall_cdf_int_udb(+Term)`
is used to retract a `Term` from a db_updatable component. It is typically used in the definition of `retractall_cdf_int` for db_updatable components.

`hasAttr_int_udb/3` `hasAttr_int_udb(?Source,?Relation,?Target)`
is used to provide access to database-resident relations in CDF components. It is typically used in the definition of `hasAttr_int/3` for db_updatable components.

4.9 Concurrency Control in CDF

Logging has multiple purposes in CDF use. It is used to allow incremental checkpointing, which is faster and takes less space than continual saving of an entire CDF. Logging is also used to support concurrent use of a CDF.

Logging can be turned on and/or off. When it is on, every update to the CDF is ""logged"" in an in-memory predicate. This log can then be saved to disk in a ""checkpoint"" file, and later used to recreate the CDF state as it was at the time of the file-saving. The checkpoint file contains the locations of the versions of the components needed to reconstruct the state.

When used to support multiple concurrent use of a CDF, first logging is turned on, and CDF components are loaded from a stored (shared) CDF, and their versions are noted. Subsequent updates to the in-memory CDF are logged as they are done. Then when the in-memory CDF is to be written back to disk to create new versions of the updated components, using update_all_components(in_shared_place), the following is done for each component. If the current most-recent-version on disk is the same as the one orginally loaded to memory, then update_all_components works normally (in_place), incrementing the version number and writing out the current in-memory component as that new version. Otherwise, there is a more recent version of the CDF on disk (written by a ""concurrent user"".) The most recent version is loaded into memory, and the log is used to apply all the updates to that new version. (If conflicts are detected, they must be resolved. At the moment, no conflict detection is done.) Then update_all_components(in_place) is used to store that updated CDF. After update_all_components is run, the log is emptied, and the process can start again.

`cdf_log_component_dirty/1` T
his is a dynamic predicate. After restoring a checkpoint file and applying the updates, `cdf_log_component_dirty/1` is true of all components that differ from their stored versions. It is a ""local version"" of cdf_flags(dirty,_), and should be ""OR-ed"" with it to find the components that have been updated from last stored version.

`cdf_set_log_on/2` `cdf_set_log_on(+LogFile,+Freq)`

This predicate creates a new log and ensures that logging will be performed for further updates until logging is turned off.

`cdf_set_log_suspend/0` `cdf_set_log_suspend/0`

temporarily turns logging off, if it is on. It is restarted by `cdf_set_log_unsuspend/0`.

`cdf_reset_log/0` |

f logging is on, this predicate deletes the current log, and creates a new empty one. If logging is off, no action is taken

`cdf_log/1` `cdf_log(ExtTermUpdate)`

takes a term of the form assert(ExtTerm) or retractall(ExtTerm) and adds it to the log, if logging is on. ExtTerm must be a legal extensional fact in the CDF.

`cdf_apply_log/0` `cdf_apply_log`

applies the log to the current in-memory CDF. For example, if the in-memory CDF has been loaded from a saved CDF version, and the log represents the updates made to that CDF saved in a checkpoint file, then this will restore the CDF to state at the time the checkpoint file was written.

When applying the updates, it does NOT update the CDF dirty flags. However, it does add the name of any modified component to the predicate `cdf_log_component_dirty/1`. This allows a user to determine both when a change has been made since the last checkpoint has been saved and since the last saved component version. (See also `cdf_log_OR_dirty_flags/0`.)

`cdf_apply_merge_log/0` `cdf_apply_merge_log`

applies the current log to the current CDF. The CDF may not be the one that formed the basis for the current log. I.e., it may have been updated by some other process. This function depends on the user-defined predicate, `check_log_merge_assert(+Term,-Action)` to provide information on whether the assert actions should be taken or not, and which provide a conflict. (Retract actions are always assumed to be acceptable.)

`cdf_log_OR_dirty_flags/0` `cdf_log_OR_dirty_flags`

makes every component in `cdf_log_component_dirty/1` to be dirty, i.e., `cdf_flags(dirty,CompName)` to be made true.

`cdf_save_log/1` `cdf_save_log(LogFile)`

writes a checkpoint file into the file named `LogFile`. The file contains the current in-memory log and the components and their versions from which these updates created the current state.

`cdf_remove_log_file/1` `cdf_remove_log_file(LogFile)`

renames the indicated file to the name obtained by appending a ~ to the file (deleting any previous file with this name.) This effectively removes the indicated file, but allows for external recovery, if necessary.

`cdf_restore_from_log/1` `cdf_restore_from_log(LogFile)`

recreates the CDF state represented by the chekpoint file named `LogFile`. The current CDF is assumed initialized. It loads the versions of the components indicated in the checkpoint file, and then applies the logged updates to that state.

Chapter 5

Programming with CDF

5.1 CDF as a Constraint Language

As do most description logics, Type-1 CDF Ontologies have the ability to express a large class of constraint problems in a succinct manner.

Example 5.1.1 As a simple example of the relationship between ontologies and constraints, we consider a simplified ontology of manufactured metal alloys. Metal alloys are characterized by a number of characteristics such as chemical composition and the form in which the alloy is primarily manufactured. For example, the Americal Society for Technical Manufacturers alloy ASTM B 107 is manufactured as a form TUBE, WIRE, or ROD, while the Sikorsky Corporation specification SS 9705 indicates that the metal has the form BAR or TUBE. This information is obtained by the fragment.

```
necessCond_ext(cid('SS 9705',specs),
      vid(exists(rid(hasForm,specs),(cid(bar,specs) ; cid(tube,specs)) ),
      atMost(1,rid(hasForm,specs)) )

necessCond_ext(cid('ASTM B 107',specs),
      vid(exists(rid(hasForm,specs), (cid(wire,specs) ; cid(tube,specs) ; cid(rod,specs)) ),

necessCond_ext(cid(bar,specs),vid(not(wire,specs)))
necessCond_ext(cid(bar,specs),vid(not(tube,specs)))
necessCond_ext(cid(bar,specs),vid(not(rod,specs)))
necessCond_ext(cid(rod,specs),vid(not(tube,specs)))
necessCond_ext(cid(rod,specs),vid(not(wire,specs)))
necessCond_ext(cid(tube,specs),vid(not(wire,specs)))
```

CDF has been used in systems that read technical drawings of airplane parts and infer properties about the parts. If a given part `oid(p1,specs)` were described by both ASTM B 107 and SS 9705 the goal `allModelsEntails(oid(p1,specs),exists(rid(hasForm,specs),cid(tube,specs)))` would succeed, indicating that the part must have the form of a tube. Such information could be useful in automatically deciding that the part must be manufactured by a supplier with specialized tube-bending machinery, which is often not available for suppliers who perform general machining.

The next example is more abstract, and indicates how CDF ontologies can represent propositional clauses. As discussed in Section 4.3, in Version 1(beta), the CDF theorem prover is not especially efficient, and should not be used for

Example 5.1.2 Consider the propositional clauses, $p \vee q \vee \neg r$ and $\neg p \vee \neg q \vee r$. These clauses can be represented by the ontology

```
necessCond_ext(oid(o1,prop), (cid(p,prop) ; cid(q,prop) ; not(cid(r,prop))) ),
necessCond_ext(oid(o1,prop), (not(cid(p,prop)) ; not(cid(q,prop)) ; cid(r,prop)) ),
```

The consistency of the two clauses can be determined by checking the consistency of `oid(o1,prop)`.

It is no surprise to a reader familiar with description logics that a CDF ontology can represent even more expressive theories. A modal formula, such as $\Box p \vee \Diamond(q \wedge r)$ can easily be expressed as a class expression

```
all(rid(rel,modal),(cid(p,modal) ; exists(rid(rel,modal),(cid(q,modal) , cid(r,modal)))))
```

and incorporated into an ontology. It is easy to see from Chapter 3 that CDF class expressions express *multi-modal* formulas, in which more than one relation can exist within a relational quantifier, and allows numeric constraints on relational quantifiers as well. On the other hand Version 1(beta) of CDF does not allow specification of transitive relations footnoteSoon!, and so does not allow the representation of temporal logics such as CTL* or the modal-μ calculus (see e.g. [**?**]).

5.2 Non-Monotonic Reasoning and CDF

XSB supports non-monotonic reasoning through its implementation of the well-founded semantics, its support of ASP through the XASP package. As a result, a number of non-monotonic formalisms have been implemented in XSB and used for a variety of applications (e.g. [Swi99, CS02, APS04] However the use of negation within CDF class expressions differs significantly from the use of negation in XSB. Consider an example taken from medical reasoning in which a patient in the US with fever and headache is assumed to have influenza unless other knowledge about the cause of the symptoms, say menengitis, is present. Such reasoning is easy to represent in Prolog.

```
has_influenza:- has_fever, has_headache, tnot(menengitis).
```

If `has_fever`, and `has_headache` are both true, but nothing is known about `menengitis`, the atom `has_influenza` will be inferred. Consider a translation of the above rule to a class expression:

```
has_influenza or not(has_fever and has_headache and notmenengitis)
```

Now if `has_fever`, and `has_headache` are both true the above formula reduces to

`has_influenza or menengitis`.

So that either influenza or menengitis is consistent with the class expression In other words, the preference for non-monotonically inferring `has_infuenza` is not expressable with a simple class expression. Only a limited amount of work has been done to relate reasoning in description logics to non-monotonic reasoning (e.g. [BH95]), and non-monotonic reasoning is not supported within Version 1(beta) of CDF. However, there is no restriction on using non-monotonic negation within intentional rules. Accordingly, the above example could be written as

```
hasAttr_int(oid(Patient,ex),rid(hasDiagnosis,ex),cid(influenza,ex)):-
     has_fever,has_headache,tnot(menengitis).

hasAttr_int(oid(Patient,ex),rid(hasDiagnosis,ex),cid(menengitis,ex)):-
     menengitis.
```

This example indicates a simple programming architecture for CDF. Below and above CDF are any XSB rules that are used by intensional rules, and these rules may use non-monotonic negation, numeric constraints, or other features. As long as, say, CDF intensional rules do not depend non-monotonically on other CDF relations the semantic properties of CDF will be preserved. Of course, more sophisticated notions of stratification of non-monotonic negation can be explored, such as the ability of two disjoint CDF components to be non-monotonically related. In any case, Version 1(beta) of CDF does not prohibit non-monotonic dependencies among CDF components, but a user who programs such dependencies should fully understand the difference between classical and non-monotonic negation and ensure that non-monotonic dependencies do not compromise consistency and entailment checking.

5.3 Using CDF Relations in Rules

5.4 CDF and FLORA-2

The Type-0 ontologies of CDF have an "object-oriented" or frame-based flavor. However, CDF is not the only object-oriented packaged for XSB: the *FLORA-2* package [YKZ05] is also a sophisticated package that allows object-oriented logic programming based on the semantics of F-Logic [KLW95] and that has a growing user community. It is natural to ask about the relation between CDF and F-Logic. Figure 5.1 (from [YKZ05]) is an introductory example of an FLORA-2 object base concerning publications. We explain its semantics in terms of a translation into CDF (Figure 5.2). From comparing the two figures, (or the formal semantics of FLORA-2 and CDF) it can be seen that the FLORA-2 schema operator `::/2` corresponds to an `isa/2` relation between two classes, while the object operator `:/2` corresponds to an `isa/2` relation between an object and a class. Other relations for classes and objects have a rather different syntax in FLORA-2 than in CDF. First note that in FLORA-2, the molecule `institution[name => string, address => string]` corresponds to two simpler molecules `institution[name => string]` and `institution[address => string]`. Each of these latter molecules can be translated into 2 CDF facts, an `allAttr_ext/3` fact to denote the typing and a `maxAttr/4` fact to denote the cardinality. The FLORA-2 operator `=>>` indicates a non-functional schema dependency and can be translated using an `allAttr_ext/3` fact alone. At the object level, the operators `->` and `->>` are each handled by a `hasAttr_ext` fact for the appropriate object: information that the dependency is functional in a `->` relation is handled

Schema:
conf_p :: paper.
journal_p :: paper.
paper[authors⇛person, title⇒string].
journal_p[in_vol⇒journal_vol].
journal_vol[of ⇒journal, volume⇒integer, year⇒integer].
journal[name⇒string, publisher⇒string, editors⇛person].
person[name⇒string, affil(integer)⇒institution].
institution[name⇒string, address⇒string].

Objects:
o_{j1} : journal_p[title→'Records, Relations, Sets, and Entities', authors↠{o_{mes}}, in_vol→o_{i11}].
o_{i11} : journal_vol[of→o_{is}, volume→1, year→1975].
o_{is} : journal[name→'Information Systems', editors↠{o_{mj}}].
o_{mes} : person[name→'Michael E. Senko',affil(1976)→o_{rwt}].
o_{rwt} : institution[name→'RWTH_Aachen'].

Figure 5.1: Part of a Publications Object Base and its Schema in *FLORA*-2

by the CDF schema and does not need to be repeated. Also note that the FLORA-2 `@` operator, which is used to handle the non-binary (i.e. parameterized) attribute `affil` for a person is reflected by the CDF product class.

FLORA-2 syntax is more succinct for this example than CDF syntax for three reasons: the use of F-Logic molecules in FLORA-2, the use of components in CDF identifiers, and the need for both a `allAttr_ext/3` and a `maxAttr/4` fact in the translation of each `=>` operator. These features stem from the different design choices behind CDF and FLORA-2. FLORA-2 is intended for programmers who want to program in an object-oriented logic, while CDF is intended as a Prolog-based repository for logically structured knowledge. From this latter perspective the use of components in CDF means that objects can have more meaningful identifiers than shown in Figure 5.1 – `oid('Michael E. Senko',flora)` would have a different meaning and different properties than a 'Michael E. Senko' taken from a different ontology and having a different component. Furthermore, the necessity of using an `allAttr/3` along with a `maxAttr/3` relation to represent `=>` arises from the ability of CDF to represent fine shades of meaning within the schema of an ontology. For instance, it may be the case that a heterosexual male has many lovers, all of whom are women, but that only one of the women is a primary lover. It is difficult to express this meaning in FLORA-2, but it can be represented in CDF as

```
isa_ext(rid(hasPrimaryLover,ex),rid(hasLover,ex)).
allAttr_ext(cid(heteroMale,ex),rid(hasLover,ex),cid(woman,ex)).
maxAttr_ext(cid(heteroMale,ex),rid(hasPrimaryLover,Ex),cid(woman,ex)).
```

The above differences, however, are relatively minor compared to the main differences between the systems. CDF supports consistency and entailment checking for Type-1 ontologies which FLORA-2 does not support; FLORA-2 however has support for non-monotonic inheritance which is not supported in CDF (nor in description logics). By and large a determined programmer could obtain most of the benefits of FLORA-2 using a combination of CDF and XSB, while a determined FLORA-2 programmer could obtain most of the benefits of CDF within FLORA-2 (perhaps even by translating monotonic FLORA-2 knowledge bases to class expressions and calling the CDF theorem prover).

Schema:
isa_ext(cid(conf_p,flora),cid(paper,flora)).
isa_ext(cid(journal_p,flora),cid(paper,flora)).

allAttr_ext(cid(paper,flora),rid(authors,flora),cid(person,flora)).
allAttr_ext(cid(paper,flora),rid(title,flora),cid(allAtoms,cdfpt)).
maxAttr_ext(cid(paper,flora),rid(title,flora),cid(allAtoms,cdfpt)).

allAttr_ext(cid(journal_p,flora),rid(in_vol,flora),cid(journal_volume,flora)).
maxAttr_ext(cid(journal_p,flora),rid(in_vol,flora),cid(journal_volume,flora)).

allAttr_ext(cid(journal_vol,flora),rid(of,flora),cid(journal,flora)).
maxAttr_ext(1,cid(journal_vol,flora),rid(of,flora),cid(journal,flora)).
allAttr_ext(cid(journal_vol,flora),rid(volume,flora),cid(allIntegers,cdfpt)).
maxAttr_ext(1,cid(journal_vol,flora),rid(volume,flora),cid(allIntegers,cdfpt)).
allAttr_ext(cid(journal_vol,flora),rid(year,flora),cid(allIntegers,cdfpt)).
maxAttr_ext(1,cid(journal_vol,flora),rid(year,flora),cid(allIntegers,cdfpt)).

allAttr_ext(cid(journal,flora),rid(name,flora),cid(allAtoms,cdfpt)).
maxAttr_ext(1,cid(journal,flora),rid(name,flora),cid(allAtoms,cdfpt)).
allAttr_ext(cid(journal,flora),rid(publisher,flora),cid(allAtoms,cdfpt)).
maxAttr_ext(1,cid(journal,flora),rid(publisher,flora),cid(allAtoms,cdfpt)).
allAttr_ext(cid(journal,flora),rid(editors,flora),cid(person,flora)).

allAttr_ext(cid(person,flora),rid(name,flora),cid(allAtoms,cdfpt)).
maxAttr_ext(1,cid(person,flora),rid(name,flora),cid(allAtoms,cdfpt)).
allAttr_ext(cid(person,flora),rid(affil,flora),cid(pair(cid(institution,flora),cid(allIntegers,cdfpt)))).
maxAttr_ext(1,cid(person,flora),rid(affil,flora),cid(pair(cid(institution,flora),cid(allIntegers,cdfpt)))).

allAttr_ext(cid(institution,flora),rid(name,flora),cid(allAtoms,cdfpt)).
maxAttr_ext(1,cid(institution,flora),rid(name,flora),cid(allAtoms,cdfpt)).
allAttr_ext(cid(institution,flora),rid(address,flora),cid(allAtoms,cdfpt)).
maxAttr_ext(1,cid(institution,flora),rid(address,flora),cid(allAtoms,cdfpt)).

Objects:
isa_ext(oid(o_{j1},flora),cid(journal,flora)).
hasAttr_ext(oid(o_{j1},flora),rid(title,flora),cid('Records, Relations, Sets, and Entities',cdfpt)).
hasAttr_ext(oid(o_{j1},flora),rid(authors,flora),oid(o_{mes},flora)).
hasAttr_ext(oid(o_{j1},flora),rid(in_vol,flora),oid(o_{i11},flora)).

hasAttr_ext(oid(o_{i11},flora),rid(of,flora),oid(o_{is},flora))
hasAttr_ext(oid(o_{i11},flora),rid(volume,flora),oid(1,cdfpt))
hasAttr_ext(oid(o_{i11},flora),rid(year,flora),oid(1976,cdfpt))

hasAttr_ext(oid(o_{mes},flora),rid(name,flora), oid('Michael E. Senko',cdfpt)).
hasAttr_ext(oid(o_{mes},flora),rid(affil,flora), oid(pair(oid(o_{rwt},flora),oid(1976,cdfpt)).

hasAttr_ext(oid(o_{rwt},flora),rid(name,flora), oid('RWTH_Aachen',cdfpt)).

Figure 5.2: CDF Encoding for FLORA-2

Bibliography

[APS04] J. Alferes, L. M. Pereira, and T. Swift, *Well-founded abduction and generalized stable models via tabled dual programs*, Theory and Practice of Logic Programming **4** (2004), no. 4, 383–428.

[BH95] F. Baader and B. Hollunder, *Embedding defaults into terminological representation systems*, J. Automated Reasoning **14** (1995), 149–180.

[BN03] F. Baader and W. Nutt, *Basic description logics*, pp. 43–95, Cambridge University Press, 2003.

[Cal01] M. Calejo, *Interprolog: A declarative java-Prolog interface*, EPIA, Springer-Verlag, 2001.

[CGLN02] D. Calvenese, G. De Giacomo, M. Lenzerini, and D. Nardi, *Reasoning in expressive description logics*, Handbook of Automated Theorem Proving, vol. 2, 2002, pp. 1582–1634.

[CS02] B. Cui and T. Swift, *Preference logic grammars: Fixed-point semantics and application to data standardization*, Artificial Intelligence **138** (2002), 117–147.

[HPS99] I. Horrocks and P. Patel-Schneider, *Optimizing description logic subsumption*, Journal of Logic and Computation **9** (1999), no. 3.

[KLW95] M. Kifer, G. Lausen, and J. Wu, *Logical foundations of object-oriented and frame-based languages*, Journal of the ACM **42** (1995), 741–843.

[Llo84] J. W. Lloyd, *Foundations of logic programming*, Springer-Verlag, Berlin Germany, 1984.

[MB92] R. MacGregor and D. Brill, *Recognition algorithms for the LOOM classifier*, Proceedings of the Tenth National Conference on Artificial Intelligence, 1992, pp. 774–779.

[MH03] R. Moller and V. Haarslev, *Description logic systems*, The Description Logi Handbook, Cambridge University Press, 2003, pp. 282–305.

[MPS03] D. McGuiness and P. Patel-Schneider, *From description logic provers to knowledge representation systems*, The Description Logi Handbook, Cambridge University Press, 2003, pp. 265–281.

[Pro01] *Protege documentation set: release 1.7*, 2001, Available via `http://protege.stanford.edu`.

[PSMB+91] P. Patel-Schneider, D.L. McGuiness, R. Brachman, L. Resnick, and A. Borgida, *The CLASSIC knowledge representation system: Guiding principles and implementation rationale*, SIGARGT Bull. **2** (1991), no. 3, 108–113.

[RBG+97] A. Rector, S. Bechhofer, C. Goble, I. Horrocks, W. Nowlan, and W.D. Soloman, *The GRAIL concept modelling language for medical terminology*, Artificial Intelligence in Medicine **9** (1997), 131–171.

[RRS+99] I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren, *Efficient access mechanisms for tabled logic programs*, Journal of Logic Programming **38** (1999), no. 1, 31–55.

[Rul03] *The Rule Markup Initiative*, Available at `http://www.dfki.uni-kl.de/ruleml`, 2003.

[SMVW02] M. Smith, D. McGuiness, R Volz, and C Welty, *Web ontology language (owl) guide version 1.0*, 2002.

[SW03] T. Swift and D. S. Warren, *The meaning of cold dead fish*, Available via `http://www.cs.sunysb.edu/~tswift`, 2003.

[Swi99] T. Swift, *Tabling for non-monotonic programming*, Annals of Mathematics and Artificial Intelligence **25** (1999), no. 3-4, 201–240.

[Swi04] ______, *Deduction in ontologies via answer set programming*, International Conference on Logic Programming and Non-Monotonic Reasoning, LNAI, no. 2923, 2004, pp. 275–289.

[UNS02] *Universal standard products and services classification*, `http://www.unspsc.org`, 2002.

[XSB03] XSB, *The XSB Logic Programming System version 2.6*, 2003, `http://xsb.sourceforge.net`.

[YKZ05] G. Yang, M. Kifer, and C. Zhao, *Flora-2: User's manual version 0.94*, 2005, Avaliable via flora.sourceforge.net.