SMO

System Management Ontology

Description Logic Formalization of CIM

Andrea Calì, Diego Calvanese, Giuseppe De Giacomo,
Maurizio Lenzerini, Daniele Nardi

Research Report N. 1 **March 8, 2001**

Abstract

Common Information Model (CIM) has the goal of providing a suitable approach for modeling systems and networks using the object-oriented paradigm. In this document we illustrate how to map CIM onto an expressive Description Logics, called $\mathcal{DLR}_{ifd}$, so as to obtain a rigorous logical framework for representing and reasoning on managed systems. The document is organized as follows. We first give an overview of both CIM and the Description Logic $\mathcal{DLR}_{ifd}$. We then illustrate the formalization of CIM in terms of $\mathcal{DLR}_{ifd}$. Finally, we present an example of how such a formalization works, by showing how the CIM Core model and the CIM Common model is expressed in $\mathcal{DLR}_{ifd}$.

Product	Research Report N. 1
Date	March 8, 2001
Number of pages	18
Authors	Andrea Calì, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, Daniele Nardi
Affiliation	Dipartimento di Informatica e Sistemistica Università di Roma "La Sapienza" Via Salaria 113, I-00198 Roma, Italy *lastname*@dis.uniroma1.it

1 Introduction

Common Information Model (henceforth denoted by CIM) is a model based on UML (Unified Modeling Language). CIM was defined by DMTF (Distributed Management Task Force). The purpose of CIM is to provide a rigorous approach for modelling systems and networks using the object-oriented paradigm. More precisely, CIM is *based* on the specification of class diagrams in UML. CIM has got a *Meta Schema* made up of constructs borrowed from UML. Following the specifications of the Meta Schema other schemas are constructed, which form the basis for a sort of vocabulary for analyzing and describing managed systems. It should be pointed out that CIM is not bound to any particular implementation.

CIM offers three conceptual layers: the *Core Model*, the *Common Model* and the *Extension Schemas*. The Core Model and the Common Model together form the *CIM Schema*.

(i) The Core Model is an information model that captures basic notions which are applicable to all areas of management (e.g., logical device or system component).

(ii) The Common Model is an information model that expresses concepts related to specific management areas, but still independent of a particular technology or implementation. The common areas defined in the Common Model are: *Systems*, *Devices*, *Applications*, *Networks*, *Physical*.

(iii) Extension Schemas are made up of classes that represent managed objects that are *technology specific additions to the Common Model*.

In this overview we will focus our attention on CIM Meta Schema, with the goal of mapping the constructs of the Meta Schema onto those of Description Logics, in order to have a rigorous logical framework for representing and reasoning on managed systems.

This document is organized as follows: in Section 2 we give an overview of CIM, and in Section 3 an overview of $\mathcal{DLR}_{ifd}$. In Section 4 we illustrate the formalization of CIM in terms of $\mathcal{DLR}_{ifd}$. Finally, in Section 5 we present an example of how such a formalization works, by showing how the CIM Core model and the CIM Common model is expressed in $\mathcal{DLR}_{ifd}$.

2 CIM Meta Schema

In this section we examine CIM in detail, describing each basic construct and its meaning.

2.1 Classes, properties and methods

A *class* in CIM is a construct which represents *sets of objects* with common structural features and semantics. As in UML, in CIM a diagram is denoted by a rectangle divided into compartments, as shown in Figure 1.

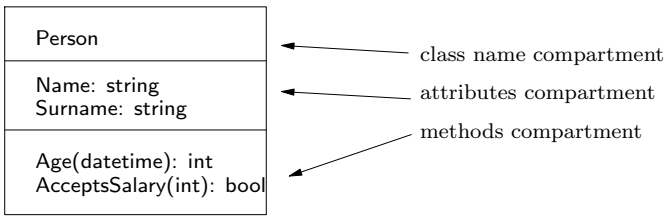

Figure 1: Representation of a class in CIM

In the *class name compartment* we indicate the name of the class, which has to be unique in the whole schema.

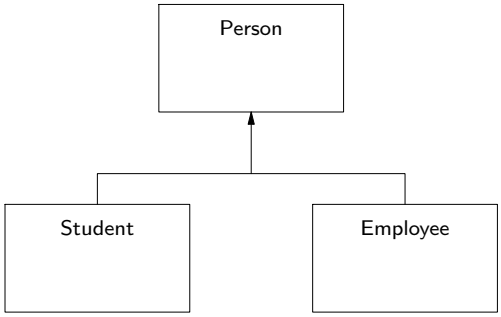

Figure 2: A class hierarchy in CIM

In the *attributes compartment* we declare the attributes of the class. An attribute is denoted by a name (possibly followed by the *multiplicity*, between square brackets) and a type, which indicates the domain to which the values of the attribute are to belong. For example Date[3]:datetime means that the class has got three values of type datetime (and labeled with Date). In CIM attributes are *single-valued*, i.e. to each field we associate only one value of the correspondent domain, if not otherwise specified.

In the *methods compartment* we declare the methods of the class, which describe the operations that can be performed on the objects of the class, i.e. the behavioural features of the class itself. The syntax of a method definition is of the usual form *method-name*(*parameter-list*):(*return-type-list*). A method may return a *tuple* of values, i.e., an element of the cartesian product of a collection of sets $T_1, \ldots, T_k$.

As in UML, in CIM we can define class hierarchies, represented in the usual way, as shown in Figure 2. A subclass inherits all properties and methods of the father class (superclass); with respect to UML, in CIM we have a further restriction stating that *multiple inheritance is not allowed*, i.e. a subclass can have at most one superclass.

2.2 Associations

An *association* in CIM is a relation between two or more classes; an association is always represented by a class. A relation R between two classes C_1 and C_2 is represented as in Figure 3; r_1 and r_2 are the *role names* of C_1 and C_2 respectively; the role names specify the role that each class plays within the relation R. The *cardinality constraint* $a..b$ specifies that each instance of class C_1 can participate at least a times and at most b times to relation R; $c..d$ has an analogous meaning for class C_2.

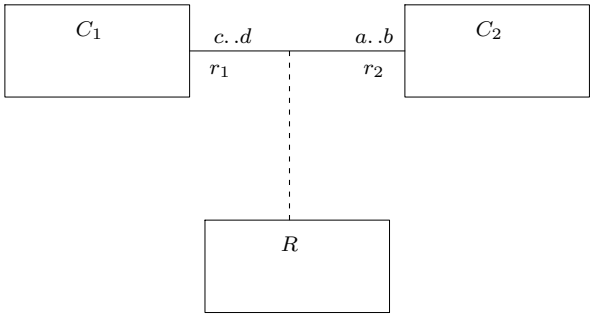

Figure 3: Association in CIM

Example 1 In Figure 4 we model ownership of houses; a person can own 0 or more houses (the

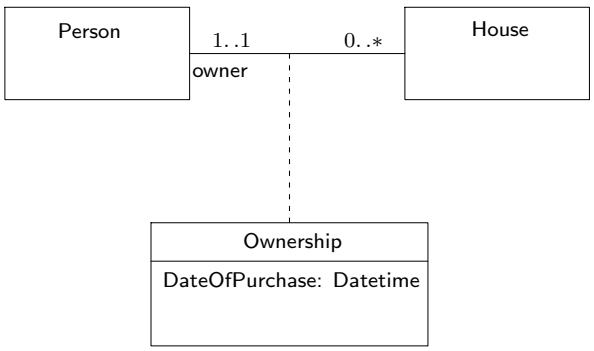

Figure 4: Example of association in CIM

symbol $*$ means ∞), while a house has to be owned by exactly one person, which is denoted by the role owner.

2.3 Aggregations

An aggregation in CIM is a binary relationship between two classes. An aggregation is a part-whole relationship, specifying that each instance of a class is made up of a set of instances of another class. An aggregation is denoted as shown in Figure 5, where the diamond indicates the *containing* class, opposed to the *contained* class. The cardinality constraints have the same meaning as in associations. Also in aggregation it is possible to define role names which denote

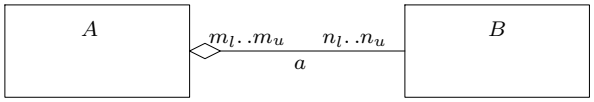

Figure 5: Aggregation in CIM

Figure 6: Example of aggregation in CIM

the role each class plays in the aggregation.

Example 2 In figure 6 we have persons belonging to families; a family can have 1 or more persons, while a person belongs to at most one family.

2.4 Overriding

In CIM it is possible to explicitly override with a method of a subclass a method of the corresponding superclass. This *override relation* in CIM has to be declared *explicitly*.

2.5 Schemas

A CIM schema is a set of classes, associations, and aggregations representing information about a fragment of real world. Within a single schema, class names have to be unique. A class have to belong to exactly one schema.

2.6 Qualifiers

In CIM it is possible to add information to a schema by using *qualifiers*. For example, a qualifier can specify required properties of a class. In general, qualifiers are used in CIM to express in an informal way information which cannot be expressed by other constructs of CIM Meta Schema. More investigation is needed on qualifiers in order to come up with a proposal on how to express them in Description Logics.

3 Description Logic $\mathcal{DLR}_{ifd}$

The goal of this section is to give an overview on the Description Logic $\mathcal{DLR}_{ifd}$ introduced in [4], which is able to capture a great variety of data models with many forms of constraints [3, 6].

3.1 Syntax

The basic elements of $\mathcal{DLR}_{ifd}$ are *concepts* (unary relations), and *n-ary relations*. We assume to deal with a finite set of atomic relations and atomic concepts, denoted by P and A, respectively. We use R to denote arbitrary relations (of given arity between 2 and n_{max}), and C to denote arbitrary concepts, respectively built according to the following syntax:

$$\begin{array}{rcl} R & ::= & \top_n \mid P \mid (i/n : C) \mid \neg R \mid R_1 \sqcap R_2 \\ C & ::= & \top_1 \mid A \mid \neg C \mid C_1 \sqcap C_2 \mid (\leq k\,[i]R) \end{array}$$

where i denotes a component of a relation, i.e., an integer between 1 and n_{max}, n denotes the *arity* of a relation, i.e., an integer between 2 and n_{max}, and k denotes a non-negative integer. We consider only concepts and relations that are *well-typed*, which means that (i) only relations of the same arity n are combined to form expressions of type $R_1 \sqcap R_2$ (which inherit the arity n), and (ii) $i \leq n$ whenever i denotes a component of a relation of arity n.

We introduce the following abbreviations:

- $C_1 \sqcup C_2$ for $\neg(\neg C_1 \sqcap \neg C_2)$;
- $C_1 \Rightarrow C_2$ for $\neg C_1 \sqcup C_2$;
- $(\geq k\,[i]R)$ for $\neg(\leq k{-}1\,[i]R)$;
- $\exists[i]R$ for $(\geq 1\,[i]R)$;
- $\forall[i]R$ for $\neg\exists[i]\neg R$.

Moreover, we abbreviate $(i/n : C)$ with $(i : C)$ when n is clear from the context.

A $\mathcal{DLR}_{ifd}$ *TBox* (or *schema*) is constituted by a finite set of *inclusion assertions*, where each assertion has one of the forms:

$$R_1 \sqsubseteq R_2 \qquad\qquad C_1 \sqsubseteq C_2$$

with R_1 and R_2 of the same arity.

Besides inclusion assertions, $\mathcal{DLR}_{ifd}$ TBoxes allow for assertions expressing identification constraints and functional dependencies.

An *identification assertion* on a concept has the form:

$$(\textbf{id}\ C\ [i_1]R_1, \ldots, [i_h]R_h)$$

where C is a concept, each R_j is a relation, and each i_j denotes one component of R_j. Intuitively, such an assertion states that no two different instances of C agree on the participation to

$$
\begin{array}{rcl}
\top_n^{\mathcal{I}} & \subseteq & (\Delta^{\mathcal{I}})^n \\
P^{\mathcal{I}} & \subseteq & \top_n^{\mathcal{I}} \\
(i/n : C)^{\mathcal{I}} & = & \{t \in \top_n^{\mathcal{I}} \mid t[i] \in C^{\mathcal{I}}\} \\
(\neg R)^{\mathcal{I}} & = & \top_n^{\mathcal{I}} \setminus R^{\mathcal{I}} \\
(R_1 \sqcap R_2)^{\mathcal{I}} & = & R_1^{\mathcal{I}} \cap R_2^{\mathcal{I}} \\
\top_1^{\mathcal{I}} & = & \Delta^{\mathcal{I}} \\
A^{\mathcal{I}} & \subseteq & \Delta^{\mathcal{I}} \\
(\neg C)^{\mathcal{I}} & = & \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}} \\
(C_1 \sqcap C_2)^{\mathcal{I}} & = & C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}} \\
(\leq k\, [i]R)^{\mathcal{I}} & = & \{a \in \Delta^{\mathcal{I}} \mid \sharp\{t \in R_1^{\mathcal{I}} \mid t[i] = a\} \leq k\}
\end{array}
$$

Figure 7: Semantic rules for $\mathcal{DLR}_{ifd}$ (P, R, R_1, and R_2 have arity n)

$R_1, \ldots, R_h$. In other words, if a is an instance of C that is the i_j-th component of a tuple t_j of R_j, for $j \in \{1, \ldots, h\}$, and b is an instance of C that is the i_j-th component of a tuple s_j of R_j, for $j \in \{1, \ldots, h\}$, and for each j, t_j agrees with s_j in all components different from i_j, then a and b coincide.

A *functional dependency assertion* on a relation has the form:

$$(\textbf{fd}\ R\ i_1, \ldots, i_h \rightarrow j)$$

where R is a relation, $h \geq 2$, and $i_1, \ldots, i_h, j$ denote components of R. The assertion imposes that two tuples of R that agree on the components $i_1, \ldots, i_h$, agree also on the component j.

Note that unary functional dependencies (i.e., functional dependencies with $h = 1$) are ruled out in $\mathcal{DLR}_{ifd}$, since these lead to undecidability [4]. Note also that the right hand side of a functional dependency contains a single element. However, this is not a limitation, because any functional dependency with more than one element in the right hand side can always be split into several dependencies of the above form.

3.2 Semantics

The semantics of $\mathcal{DLR}_{ifd}$ is specified through the notion of interpretation. An *interpretation* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ of a $\mathcal{DLR}_{ifd}$ TBox $\mathcal{T}$ and a set of constants $\mathcal{C}$ (to be used in queries) is constituted by an *interpretation domain* $\Delta^{\mathcal{I}}$ and an *interpretation function* $\cdot^{\mathcal{I}}$ that assigns

- to each concept C a subset $C^{\mathcal{I}}$ of $\Delta^{\mathcal{I}}$
- to each relation R of arity n a subset $R^{\mathcal{I}}$ of $(\Delta^{\mathcal{I}})^n$

such that the conditions in Figure 7 are satisfied. In the figure, $t[i]$ denotes the i-th component of tuple t. We observe that $\top_1$ denotes the interpretation domain, while $\top_n$, for $n > 1$, does *not* denote the n-Cartesian product of the domain, but only a subset of it, that covers all relations of arity n. It follows, from this property, that the "$\neg$" constructor on relations is used to express difference of relations, rather than complement.

- An interpretation $\mathcal{I}$ *satisfies* an inclusion assertion $R_1 \sqsubseteq R_2$ (resp. $C_1 \sqsubseteq C_2$) if $R_1^{\mathcal{I}} \subseteq R_2^{\mathcal{I}}$ (resp. $C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$).
- An interpretation $\mathcal{I}$ *satisfies* the assertion $(\textbf{id}\ C\ [i_1]R_1, \ldots, [i_h]R_h)$ if for all $a, b \in C^{\mathcal{I}}$ and for all $t_1, s_1 \in R_1^{\mathcal{I}}, \ldots, t_h, s_h \in R_h^{\mathcal{I}}$ we have that:

$$\left.\begin{array}{l} a = t_1[i_1] = \cdots = t_h[i_h], \\ b = s_1[i_1] = \cdots = s_h[i_h], \\ t_j[i] = s_j[i], \text{ for } j \in \{1, \ldots, h\}, \\ \qquad\qquad \text{and for } i \neq i_j \end{array}\right\} \text{ implies } a = b$$

- An interpretation $\mathcal{I}$ *satisfies* the assertion (**fd** $R\, i_1, \ldots, i_h \rightarrow j$) if for all $t, s \in R^{\mathcal{I}}$, we have that:

$$t[i_1] = s[i_1], \ \ldots, \ t[i_h] = s[i_h] \quad \text{implies} \quad t[j] = s[j]$$

3.3 Reasoning tasks

Several reasoning services are applicable to $\mathcal{DLR}_{ifd}$ TBoxes. The most important ones are TBox satisfiability (unsatisfiability) and logical implication. An interpretation that satisfies all assertions in a TBox $\mathcal{T}$ is called a *model* of $\mathcal{T}$. A TBox $\mathcal{T}$ is *satisfiable* (*unsatisfiable*) if there exists a model of $\mathcal{T}$ (there are no model of $\mathcal{T}$). An assertion α is *logically implied* by $\mathcal{T}$ if all models of $\mathcal{T}$ satisfy α. We recall that logical implication and TBox unsatisfiability are mutually reducible.

One of the distinguishing features of $\mathcal{DLR}_{ifd}$ is that it is equipped with algorithms for checking satisfiability and logical implication. Such reasoning tasks are in fact EXPTIME-decidable [4].

3.4 Discussion

It can be shown that $\mathcal{DLR}_{ifd}$ is able to capture a great variety of data models with many forms of constraints. For example, we obtain the entity-relationship model (including *is-a* relations on both entities and relations) in a straightforward way [2], and an object-oriented data model (extended with several types of constraints), by restricting the use of existential and universal quantifications in concept expressions, by restricting the attention to binary relations, and by eliminating negation, and disjunction. The following observations point out the kinds of constraints typically used in databases and software engineering that can be expressed using $\mathcal{DLR}_{ifd}$.

- Assertions directly express a special case of typed inclusion dependencies, namely the one where no projection of relations is used.

- Unary inclusion dependencies are easily expressible by means of the $\exists[2]P$ construct. For example, $\exists[2]P_1 \sqsubseteq \exists[3]P_2$ is a unary inclusion dependency between attribute 2 of P_1 and attribute 3 of P_2.

- Existence and exclusion dependencies are expressible by means of $\exists$ and $\neg$, respectively.

- Functional dependencies are directly expressible by means of functional dependencies assertions. The only functional dependencies that are not admitted in $\mathcal{DLR}_{ifd}$ are unary functional dependencies in the context of non-binary relations. This is because they lead to undecidability of reasoning. Note also, that the presence of such functional dependencies is considered as an indication of bad design in the framework of the relational data model (see [1], Chapter 11). In fact, a unary functional dependency in the context of an n-ary relation (with $n > 2$) represents a hidden relationship between the arguments of the relation, which may cause several modeling problems.

- The possibility of defining identification constraints substantially enriches the modeling power of DLs. In particular, it is possible to show that, even if only binary relations are allowed in a DL, then the use of identification constraints permits simulating the presence of n-ary relations in such a logic. For example, a relation with arity 3 can be modeled by means of a concept and 3 binary relations. Number restrictions are used to state that every instance of the concept participates in exactly one instance of the binary relation, and a suitable identification assertion states that the combination of the three binary relations form a key for the concept. Obviously, $\mathcal{DLR}_{ifd}$ further increases the modeling power

by allowing the explicit use of n-ary relations, and the possibility of imposing functional dependencies in the context of relations.

- The possibility of constructing complex expressions provides a special form of view definition. Indeed, the two assertions $P \sqsubseteq R$, $R \sqsubseteq P$ (where R is a complex expression) is a view definition for P. Notably, views can be freely used in assertions (even with cyclic references), and, therefore, all the above discussed constraints can be imposed not only on atomic relations, but also on views. These features make our logic particularly suited for expressing inter-schema relationships in the context of information integration [5], where it is crucial to be able to state that a certain concept of a schema corresponds (by means of inclusion or equivalence) to a view in another schema.

4 Formalization in Description Logics

To formalize any CIM schema in Description Logics we take advantage of the expressive power of $\mathcal{DLR}_{ifd}$. We point out that in this document we don't consider the *meta-level constructs* of CIM (e.g. qualifiers); our aim is to give a formal semantics to CIM by formalizing each basic construct of CIM Meta-Schema into Description Logics.

A CIM schema is formalized in a $\mathcal{DLR}_{ifd}$ TBox which expresses a set of constraints over a suitable set of $\mathcal{DLR}_{ifd}$ concepts and roles. The concepts and roles with which we formalize a CIM schema are derived in the following way. We partition $\mathcal{DLR}_{ifd}$ class and relation names into more specific sets: given a CIM schema, the sets of symbols that we use in our framework are the following:

(a) Set of *base types* $\mathcal{T}$: this set contains the concepts representing the base types (e.g. integer or boolean). For each base type in a CIM schema we introduce a concept in $\mathcal{T}$ with the same name.

(b) Set of *classes* $\mathcal{C}_{cl}$: this set contains the concepts representing CIM classes (but it does not contain $\mathcal{DLR}_{ifd}$ concepts representing associations). For each CIM class representing a set of objects (and not an association) we have a class in $\mathcal{C}_{cl}$ with the same name.

(c) Set of *classes representing associations* $\mathcal{C}_{as}$: this set contains the concepts representing CIM associations. We introduce a concept in $\mathcal{C}_{as}$ for each CIM association; the concept and the association have the same name.

(d) Set of *association role names* $\mathcal{R}_{as}$: this set contains the relations which we use to model CIM associations; as we will see later, the relations in $\mathcal{R}_{as}$ take their name from the role names in the corresponding CIM association. In particular, for each role name appearing in a CIM association we introduce a $\mathcal{DLR}_{ifd}$ binary relation in our framework; each association is thus modelled by a class and a set of binary relations in $\mathcal{R}_{as}$.

(e) Set of *aggregations* $\mathcal{R}_{ag}$: this set contains the relations used to model CIM aggregations. For each CIM aggregation we introduce a binary relation in $\mathcal{R}_{ag}$ with the same name.

(f) Set of *attributes* $\mathcal{A}$: this set contains the relations used to model attributes of CIM classes. We have a binary relation in $\mathcal{A}$ for each attribute appearing in a class of the CIM schema; the name of the relation is the same as that of the attribute.

(g) Set of *methods* $\mathcal{M}$: this set contains the relations which model the methods in CIM classes. For each method with n parameters appearing in a CIM class of the schema we introduce a relation in $\mathcal{M}$. Given a methof M, the name of the corresponding relation in our framework is the name M concatenated by the pair $[x, y]$ as a subscript, where y is the number of parameters of M, k is the number of elements forming the tuples returned by M, and

$x = y + k + 1$. The number of arguments of the relation corresponding to M is $y + k + 1$, since we want to have:

- an argument for the class where the method is defined,
- one argument for each type of the result, and
- one argument for each parameter.

(h) Set of *role names* $\mathcal{N}$: this set contains the symbols used to denote the roles of classes participating in aggregations, and it is identical to the set of role names appearing in the CIM schema. The symbols in $\mathcal{N}$ are not actually part of the $\mathcal{DLR}_{ifd}$ syntax; they are a syntactic addition which adds a convenient way to denote roles.

So we have that the set of concepts in our framework is given by

$$\mathcal{C} = \mathcal{T} \cup \mathcal{C}_{cl} \cup \mathcal{C}_{as},$$

while the set of relations is

$$\mathcal{R} = \mathcal{M} \cup \mathcal{A} \cup \mathcal{R}_{ag} \cup \mathcal{R}_{as}.$$

Now we expose how we can formalize each basic construct of the CIM Meta Schema in $\mathcal{DLR}_{ifd}$.

4.1 Classes, properties and methods

A CIM class is mapped onto a $\mathcal{DLR}_{ifd}$ concept; this is obvious, if we think that both CIM classes and $\mathcal{DLR}_{ifd}$ roles represent *sets of individuals*.

To formalize *attributes* we have to think of an attribute a of a class C, whose values belong to the data type T, as a function

$$a : \ C \ \longrightarrow \ T.$$

What we want to do is to assign a *type* T to the attribute a. We want to specify that, given an object $c \in C^{\mathcal{I}}$, for each t such that $(c, t) \in a^{\mathcal{I}}$ we have $t \in T^{\mathcal{I}}$, or equivalently: there is no element $(c, t) \in a^{\mathcal{I}}$ of relation a such that $t \notin T^{\mathcal{I}}$. We can express this by using universal quantification in the following way:

$$C \ \sqsubseteq \ \forall[1](a \Rightarrow (2 : T)) \tag{1}$$

with $C \in \mathcal{C}_{cl} \cup \mathcal{C}_{as}$, $T \in \mathcal{T}$, $a \in \mathcal{A}$. We remind that the above assertion is equivalent to $C \sqsubseteq \neg\exists[1](a \sqcap \neg(2 : T))$.

If the attribute is *single-valued* (i.e. a is a function), we have to add to the right side of the assertion 1, the conjunct

$$(\leq 1\,[1]a).$$

If we want a to be a non optional attribute we have to add to the right side of the assertion 1, the conjunct

$$\exists[1]a.$$

It may happen that a is a *key* for C, i.e. there is no pair of elements in $C^{\mathcal{I}}$ that share the same attribute a with the same value in $T^{\mathcal{I}}$. In this case we add the following assertion:

$$(\textbf{id}\ C\ [1]a) \tag{2}$$

In a similar way, we are able to specify that a *set* of attributes $\{a_1, \ldots, a_n\}$, with $n \geq 2$, is a key for C; in this case we have the assertion

$$(\textbf{id}\ C\ [1]a_1,\ [1]a_2,\ \cdots [1]a_n) \tag{3}$$

Note that we have disregarded the option of declaring the attribute a with a $\mathcal{DLR}_{ifd}$ assertion of the form $a \sqsubseteq (1:C) \sqcap (2:T)$. Indeed, this could cause problems because two different classes could have the same attribute a.[1]

In a similar way, we formalize methods of a class C. Suppose that C has a method

$$m(T_1, T_2) : (T_3, T_4).$$

This notation indicates that the method has two parameters of type T_1 and T_2 respectively and a return type $T_3 \times T_4$. We formalize m as a relation of arity 5 between the class C and the types $T_1, T_2, T_3, T_4 \in \mathcal{C}$. We say that m has *arity* 5, i.e., represents a relation among 5 types (the first of which is C). Obviously, another class C' could have a method named m. In our framework we do want to make a distinction between methods which have different arity and number of arguments. So the name for m in our framework is $m_{[5,2]} \in \mathcal{M}$, i.e. the name of the method followed by its arity and the number of its arguments. Therefore we have the following assertion:

$$C \sqsubseteq \forall[1](m_{[5,2]} \Rightarrow ((2:T_1) \sqcap (3:T_2) \sqcap (4:T_3) \sqcap (5:T_4))) \tag{4}$$

This assertion specifies that method m of class C is a relation between C, T_1, T_2, T_3, T_4. We also need to specify that all the methods named $m_{[5,2]}$ in our framework specify *functions* and not relations in general. This is done by adding to our TBox the following functional dependency assertions:

$$\begin{array}{l}(\textbf{fd}\ m_{[5,2]}\ 1,2,3 \rightarrow 4)\\ (\textbf{fd}\ m_{[5,2]}\ 1,2,3 \rightarrow 5)\end{array} \tag{5}$$

4.2 Subclasses and superclasses

When we have a class C that is a superclass of class D, this is immediately formalized in $\mathcal{DLR}_{ifd}$ in terms of an assertion of the form

$$D \quad \sqsubseteq \quad C \tag{6}$$

with $C, D \in \mathcal{C}_{cl} \cup \mathcal{C}_{as}$.

We remark that in CIM multiple inheritance is not supported, which means that a class can be a subclass of at most one superclass.

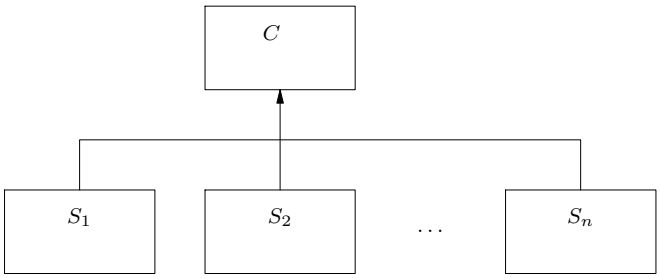

Figure 8: A typical class hierarchy in CIM

A particular case of subclassing is the one of generalization hierarchy. An example of such hierarchy is illustrated in Figure 8, in which C is a superset of each S_i, and $S_1, \ldots, S_n$ are mutually disjoint. The relationship among such classes can be expressed by the following assertions:

$$\begin{array}{lcl} S_1 & \sqsubseteq & C \\ \ldots & & \\ S_n & \sqsubseteq & C \\ S_k & \sqsubseteq & \neg S_{k+1} \sqcap \ldots \sqcap \neg S_n \ \text{ for each } k \in \{1, \ldots, n-1\} \end{array} \tag{7}$$

[1] Obviously, we want an attribute name not to be necessarily unique in the whole schema

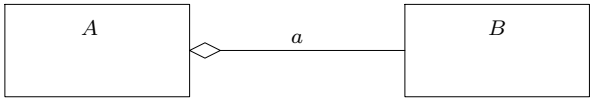

Figure 9: Aggregation in CIM

4.3 Associations and aggregations

Associations and aggregations in CIM both represent relations. The difference between the two is that in CIM, an association is *always* modeled as a class. Moreover, in general associations are n-ary relations, while aggregations are binary relations. In general, an aggregation like that in Figure 9 is formalized into the following assertion:

$$a \quad \sqsubseteq \quad (1:A) \sqcap (2:B) \tag{8}$$

with $a \in \mathcal{R}_{ag}$ e $A, B \in \mathcal{C}_{cl} \cup \mathcal{C}_{as}$.

Note that the distinction between the contained class and the containing class is not lost. Indeed, we introduce the following convention: *the first argument of the relation is the containing class.*

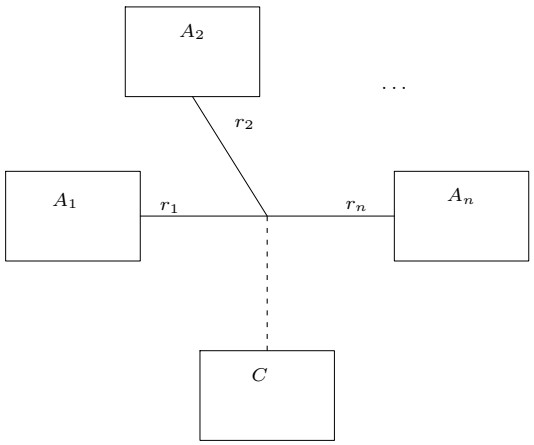

Figure 10: Association in CIM

An association like that in Figure 10 can be modelled by as a set of n *binary* relations which we will denote with the name of the roles of the association C, which are $r_1, \ldots, r_n \in \mathcal{R}_{as}$. We want to ensure the possibility of using the same role name in different association. In order to obtain this possibility, we avoid declaring $r_1, \ldots, r_n$ globally; on the contrary, we think of C as a class having n attributes of types $A_1, \ldots, A_n$ respectively. Thus we have the assertion

$$\begin{array}{rcl} C & \sqsubseteq & \forall[1](r_1 \Rightarrow (2:A_1)) \sqcap (\leq 1\,[1]r_1) \sqcap \exists[1]r_1 \sqcap \\ & & \forall[1](r_2 \Rightarrow (2:A_2)) \sqcap (\leq 1\,[1]r_2) \sqcap \exists[1]r_2 \sqcap \\ & & \vdots \\ & & \forall[1](r_n \Rightarrow (2:A_n)) \sqcap (\leq 1\,[1]r_1) \sqcap \exists[1]r_n \end{array} \tag{9}$$

where $(\leq 1\,[1]r_i)$ (with $i \in \{1, \ldots, n\}$) specifies that attributes are to be single-valued, and $\exists[1]r_i$ (with $i \in \{1, \ldots, n\}$) specifies that the class C has got to have all $r_1, \ldots, r_n$ (this is obvious because the class models the relation and not a set of instances). Moreover, we use the assertion

$$(\mathbf{id}\ C\ [1]r_1,\ [1]r_2,\ \ldots,\ [1]r_n) \tag{10}$$

to specify that each instance of C represents a *distinct* tuple in $(A_1, \ldots, A_n)$.

4.4 Cardinality constraints in associations and aggregations

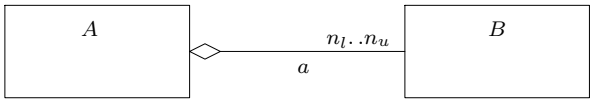

Figure 11: Cardinality constraints in aggregation

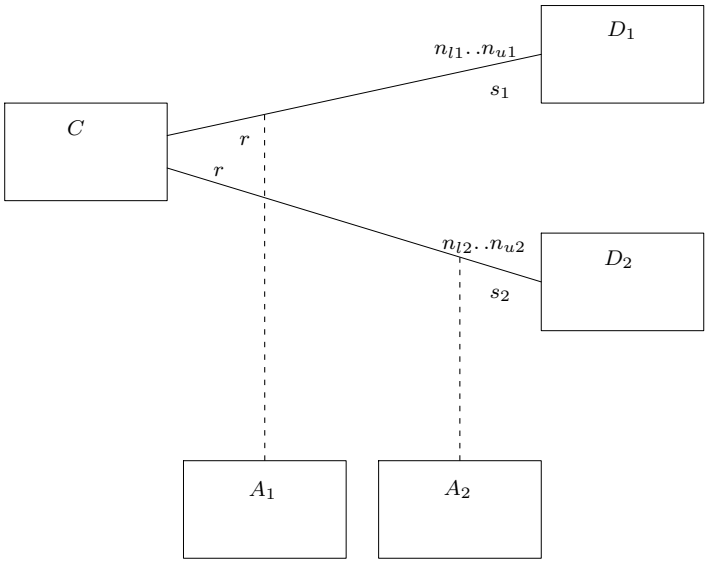

Figure 12: Cardinality constraints in aggregation

As we did in CIM, we may have restrictions over the number of instances of a class that are allowed to participate to a given association or aggregation. With the expressive power of $\mathcal{DLR}_{ifd}$ we are able to enforce these cardinality constraints in a straightforward way. Let us examine the cases of aggregation and association separately.

Aggregation Suppose we have a cardinality constraint like that in Figure 11; we simply add to the TBox the assertions

$$A \sqsubseteq (\geq n_l\,[1]a) \sqcap (\leq n_u\,[1]a) \tag{11}$$

This assertion does not generate conflicts with other aggregations, as we suppose that each aggregation name is *unique* in the schema.

Association According to the CIM specification, we impose cardinality constraints on binary associations only. The way we enforce cardinality constraints in associations is similar to the way we do with aggregations. The difference is that a role name in an association is not unique in the schema. Suppose we have a situation like that in Figure 12. In this case we have that $r \in \mathcal{R}_{as}$ expresses both the binary relation between C and A_1 and the binary relation between C and A_2. At the same time, the cardinality constraints over the participation of C to the two relations are different in general. This is not a problem anyway, as we are able to enforce cardinality constraints over the participation to $r \sqcap (1:A_1)$ and $r \sqcap (1:A_2)$, thus distinguishing between the two different associations. In conclusion, we have the following assertions:

$$C \sqsubseteq (\geq n_{l1}\,[2](r \sqcap (1:A_1))) \sqcap (\leq n_{u1}\,[2](r \sqcap (1:A_1))) \tag{12}$$

$$C \sqsubseteq (\geq n_{l2}\,[2](r \sqcap (1:A_2))) \sqcap (\leq n_{u2}\,[2](r \sqcap (1:A_2))) \tag{13}$$

4.5 Role names

The decision of representing an aggregation by a binary $\mathcal{DLR}_{ifd}$ relation leads some implications; first of all, we note that role names are lost. In our framework role names are replaced by an integer i (whose value can be only 1 or 2), which specifies whether the corresponding argument is the first or the second in the aggregation.

Now, we want to preserve the role names in our framework. Therefore we take advantage of the *set of role names* $\mathcal{N}$ introducing, for each *atomic* relation R of arity k, a *role name function*

$$f_R : \{1, \ldots, k\} \longrightarrow \mathcal{N} \cup \{\varepsilon\}.$$

The function f_R returns, given an integer i between 1 and k (the arity of the relation), the role name associated to the i-th role, if it has one, ε if the role has no name.

When we compose two relations with the the operator $\sqcap$ (we recall that the relations have got to have the same arity), role name are preserved if both overlapping roles have the same name in $\mathcal{N}$; otherwise, the role names are lost. Formally:

$$f_{R_1 \sqcap R_2}(i) \begin{cases} f_{R_1}(i) & \text{if } f_{R_1}(i) = f_{R_2}(i) \\ \varepsilon & \text{otherwise} \end{cases} \tag{14}$$

The negation of a relation R of arity k retains all the role names of the original relation. This choice could seem insensible at a first glance, but it ensures for example that matching role names are preserved when we do the union of relations (like $R_1 \sqcup R_2 = \neg(\neg R_1 \sqcap \neg R_2)$). Formally we have:

$$f_{\neg R}(i) = f_R(i) \text{ for each } i \in \{1, \ldots, k\} \tag{15}$$

We impose f_R to be *injective* for every relation R; instead, we would have the same name for more than one role, within the same relation.

4.6 Overriding

In CIM it is possible to override properties or methods of a superclass. In our framework we do not want to hide in a subclass properties and methods of the corresponding superclass. On the contrary, we want the method/property of the subclass to be a *restriction* of the corresponding method belonging to the superclass.

Example 3 For example, if $D \sqsubseteq C$ and C has a method $m(T_1, T_2) : T_3$, D automatically inherits the method $m_{[4,2]}$. Thus we have the following assertions in our TBox:

$$\begin{array}{lcl} D & \sqsubseteq & C \\ C & \sqsubseteq & \forall[1](m_{[4,2]} \Rightarrow (2 : T_1) \sqcap (3 : T_2) \sqcap (4 : T_3)) \end{array} \tag{16}$$

We also have in our TBox the following assertion related to the method name $m_{[5,2]}$:

$$(\mathbf{fd}\ m_{[4,2]}\ 1, 2, 3 \rightarrow 4)$$

Moreover, we are able to specify that the restriction of $m_{[5,2]}$ to D is *injective* with the following assertions:

$$\begin{array}{l} (\mathbf{fd}\ m_{[4,2]} \sqcap (1 : D)\ 1, 4 \rightarrow 2) \\ (\mathbf{fd}\ m_{[4,2]} \sqcap (1 : D)\ 1, 4 \rightarrow 3) \end{array} \tag{17}$$

Note that the last two assertions specify properties of $m_{[4,2]} \sqcap (1 : D)$.

4.7 Inheritance

It is easy to observe that in our framework inheritance is naturally supported among classes in $\mathcal{C}_{cl} \cup \mathcal{C}_{as}$. In fact when we have an assertion $C_1 \sqsubseteq C_2$, where $C_1, C_2 \in \mathcal{C}$, every relation having C_2 as i-th argument is automatically allowed to accept C_1 instead of C_2. As a consequence, each attribute or method of C_2, and each aggregation and association involving C_2 is inherited by C_1.

Similarly, inheritance among classes in $\mathcal{C}_{as}$ is correctly modelled in $\mathcal{DLR}_{ifd}$.

4.8 Specification of additional knowledge

We discuss here a number of aspects that can be taken into account when formalizing CIM in Description Logics.

Disjointness Both UML and CIM are not completely clear about disjointness of classes. In other words, it is left unspecified whether two classes that are not related by the is-a relation are disjoint or not. In our formalization, we follow this approach:

1. If not specified otherwise, two classes may have common instances, i.e., they are *not disjoint*.
2. If we want to impose the disjointness of two classes, say C and D, this should be done explicitly by means of the assertion:

$$C \sqsubseteq \neg D$$

Negative information Disjointness of classes is just one example of negative information. Neither UML nor CIM gives special attention to modeling other forms of negative information. Again, by exploiting the expressive power of $\mathcal{DLR}_{ifd}$, we can add negative information to the CIM schema by introducing suitable assertions. For example, we can enforce that each instance of a class C has no value for attribute a by means of the assertion:

$$C \sqsubseteq \forall[1](a \Rightarrow (2 : \bot))$$

Analogously, one can assert that no instance of C is involved in a given aggregation or association.

Complete information By default, a generalization hierarchy is open, in the sense that there may be instances of the superclass that are not instances of any of the subclasses. This allows for extending the schema more easily, in the sense that the introduction of a new subclass does not change the semantics of the superclass. However, in specific situations, it may happen that in a generalization hierarchy, the superclass C is a covering of the subclasses $C_1, \ldots, C_n$. When this is the case, we can simply include the assertion:

$$C \sqsubseteq C_1 \sqcup \cdots \sqcup C_n$$

The above assertion is a form of modeling complete information: we have complete knowledge about the class C in terms of its subclasses. Other forms of complete information can be modeled by exploiting the expressive power of $\mathcal{DLR}_{ifd}$. For example that a certain attribute is valid only for a specified set of classes can be modeled by suitably using union of classes ($\sqcup$).

Keys Although there is no explicit notion of key in UML (nor CIM), we can exploit the expressive power of $\mathcal{DLR}_{ifd}$ in order to associate keys to classes.

5 Example: CIM Core Model

We illustrate the formalization of CIM by means of Description Logics with an example; we express CIM Core Model, as represented in Figure 13, by means of a siutable $\mathcal{DLR}_{ifd}$ TBox.

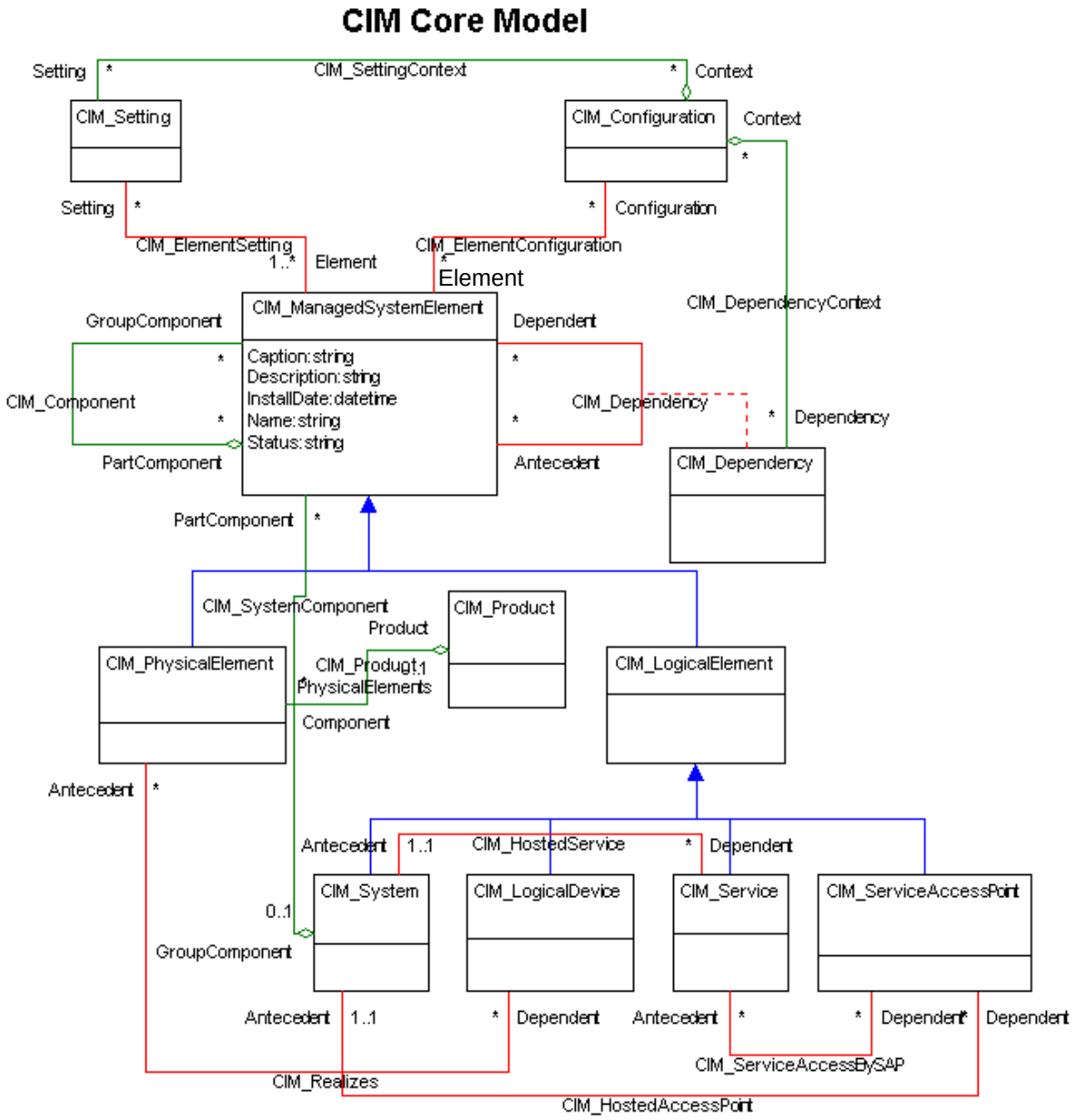

Figure 13: Representation of CIM Core Model

5.1 Alphabet

The $\mathcal{DLR}_{ifd}$ TBox with which we formalize CIM Core Model is built using the following alphabet (see Section 4):

$$
\begin{aligned}
\mathcal{C}_{cl} = \; & \{\mathsf{CIM\_Setting}, \mathsf{CIM\_Configuration}, \mathsf{CIM\_ManagedSystemElement}, \\
& \mathsf{CIM\_PhysicalElement}, \mathsf{CIM\_Product}, \mathsf{CIM\_LogicalElement}, \\
& \mathsf{CIM\_System}, \mathsf{CIM\_LogicalDevice}, \mathsf{CIM\_Service}, \mathsf{CIM\_ServiceAccessPort}\} \\
\mathcal{C}_{as} = \; & \{\mathsf{CIM\_Dependency}, \mathsf{CIM\_ElementSetting}, \mathsf{CIM\_ElementConfiguration}, \\
& \mathsf{CIM\_Realizes}, \mathsf{CIM\_HostedAccessPoint}, \mathsf{CIM\_ServicaAccessBySAP}, \\
& \mathsf{CIM\_HostedService}\} \\
\mathcal{T} = \; & \{\mathsf{string}, \mathsf{datetime}\} \\
\mathcal{A} = \; & \{\mathsf{Caption}, \mathsf{Description}, \mathsf{InstallDate}, \mathsf{Name}, \mathsf{Status}\} \\
\mathcal{R}_{as} = \; & \{\mathsf{Dependent}, \mathsf{Antecedent}, \mathsf{Setting}, \\
& \mathsf{Element}, \mathsf{Configuration}\} \\
\mathcal{R}_{ag} = \; & \{\mathsf{CIM\_SettingContext}, \mathsf{CIM\_Component}, \mathsf{CIM\_DependencyContext}, \} \\
& \mathsf{CIM\_SystemComponent}, \mathsf{CIM\_ProductPhysicalElement} \\
\mathcal{M} = \; & \{\mathsf{SetPowerState}_{[4,2]}\} \\
\mathcal{N} = \; & \{\mathsf{Setting}, \mathsf{Context}, \mathsf{GroupComponent}, \mathsf{PartComponent}\}
\end{aligned}
$$

5.2 Classes

The only class in Figure 13 that has got attributes is CIM_ManagedSystemElement; the attributes are modelled by the following assertions:

$$
\begin{aligned}
\mathsf{CIM\_ManagedSystemElement} &\sqsubseteq \forall[1](\mathsf{Caption} \Rightarrow (2 : \mathsf{string})) \\
\mathsf{CIM\_ManagedSystemElement} &\sqsubseteq \forall[1](\mathsf{Description} \Rightarrow (2 : \mathsf{string})) \\
\mathsf{CIM\_ManagedSystemElement} &\sqsubseteq \forall[1](\mathsf{InstallDate} \Rightarrow (2 : \mathsf{datetime})) \\
\mathsf{CIM\_ManagedSystemElement} &\sqsubseteq \forall[1](\mathsf{Name} \Rightarrow (2 : \mathsf{string})) \\
\mathsf{CIM\_ManagedSystemElement} &\sqsubseteq \forall[1](\mathsf{Status} \Rightarrow (2 : \mathsf{string}))
\end{aligned}
$$

As specified in the *CIM Core Model White Paper*, Version 2.4, the class CIM_ManagedSystemElement has got a method (not shown in Figure 13 whose signature is as follows:

SetPowerState([IN] PowerState: uint16, [IN]Time: datetime):uint32

To model this method we use the name $\mathsf{SetPowerState}_{[4,2]}$ and the assertion

$$
\begin{aligned}
&\mathsf{CIM\_ManagedSystemElement} \sqsubseteq \\
&\qquad \forall[1](\mathsf{SetPowerState}_{[4,2]} \Rightarrow \\
&\qquad\qquad ((2 : \mathsf{PowerState}) \sqcap (3 : \mathsf{datetime}) \sqcap (4 : \mathsf{uint32})))
\end{aligned}
$$

Additionally, we need in our TBox the assertion specifying that $\mathsf{SetPowerState}_{[4,2]}$ is a function, i.e.

$$(\mathbf{fd}\ \mathsf{SetPowerState}_{[4,2]}\ 1, 2, 3 \rightarrow 4)$$

The cardinality restriction over the participation of classes to associations and aggregations

is expressed by the assertions

$$
\begin{aligned}
\mathsf{CIM\_Setting} &\sqsubseteq (\geq 1\,[2]\mathsf{CIM\_ElementSetting}) \\
\mathsf{CIM\_ManagedSystemElement} &\sqsubseteq (\leq 1\,[2]\mathsf{CIM\_SystemComponent}) \\
\mathsf{CIM\_ServiceAccessPoint} &\sqsubseteq (\geq 1\,[2](\mathsf{Antecedent} \sqcap (1 : \mathsf{CIM\_HostedAccessPoint}))) \sqcap \\
&\quad\ (\leq 1\,[2](\mathsf{Antecedent} \sqcap (1 : \mathsf{CIM\_HostedAccessPoint}))) \\
\mathsf{CIM\_Service} &\sqsubseteq (\geq 1\,[2](\mathsf{Antecedent} \sqcap (1 : \mathsf{CIM\_HostedService}))) \sqcap \\
&\quad\ (\leq 1\,[2](\mathsf{Antecedent} \sqcap (1 : \mathsf{CIM\_HostedService})))
\end{aligned}
$$

5.3 Associations

$$
\begin{aligned}
\mathsf{CIM\_ElementSetting} &\sqsubseteq \forall[1](\mathsf{Element} \Rightarrow (2 : \mathsf{CIM\_ManagedSystemElement})) \sqcap \\
&\quad\ \forall[1](\mathsf{Setting} \Rightarrow (2 : \mathsf{CIM\_Setting})) \sqcap \\
&\quad\ (\leq 1\,[1]\mathsf{Element}) \sqcap \exists[1]\mathsf{Element} \sqcap \\
&\quad\ (\leq 1\,[1]\mathsf{Setting}) \sqcap \exists[1]\mathsf{Setting} \\
\mathsf{CIM\_ElementConfiguration} &\sqsubseteq \forall[1](\mathsf{Element} \Rightarrow (2 : \mathsf{CIM\_ManagedSystemElement})) \sqcap \\
&\quad\ \forall[1](\mathsf{Configuration} \Rightarrow (2 : \mathsf{CIM\_Configuration})) \sqcap \\
&\quad\ (\leq 1\,[1]\mathsf{Element}) \sqcap \exists[1]\mathsf{Element} \sqcap \\
&\quad\ (\leq 1\,[1]\mathsf{Configuration}) \sqcap \exists[1]\mathsf{Configuration} \\
\mathsf{CIM\_Dependency} &\sqsubseteq \forall[1](\mathsf{Antecedent} \Rightarrow (2 : \mathsf{CIM\_ManagedSystemElement})) \sqcap \\
&\quad\ \forall[1](\mathsf{Dependent} \Rightarrow (2 : \mathsf{CIM\_ManagedSystemElement})) \sqcap \\
&\quad\ (\leq 1\,[1]\mathsf{Antecedent}) \sqcap \exists[1]\mathsf{Antecedent} \sqcap \\
&\quad\ (\leq 1\,[1]\mathsf{Dependent}) \sqcap \exists[1]\mathsf{Dependent} \\
\mathsf{CIM\_Realizes} &\sqsubseteq \forall[1](\mathsf{Antecedent} \Rightarrow (2 : \mathsf{CIM\_PhysicalElement})) \sqcap \\
&\quad\ \forall[1](\mathsf{Dependent} \Rightarrow (2 : \mathsf{CIM\_LogicalDevice})) \sqcap \\
&\quad\ (\leq 1\,[1]\mathsf{Antecedent}) \sqcap \exists[1]\mathsf{Antecedent} \sqcap \\
&\quad\ (\leq 1\,[1]\mathsf{Dependent}) \sqcap \exists[1]\mathsf{Dependent} \\
\mathsf{CIM\_HostedService} &\sqsubseteq \forall[1](\mathsf{Antecedent} \Rightarrow (2 : \mathsf{CIM\_System})) \sqcap \\
&\quad\ \forall[1](\mathsf{Dependent} \Rightarrow (2 : \mathsf{CIM\_Service})) \sqcap \\
&\quad\ (\leq 1\,[1]\mathsf{Antecedent}) \sqcap \exists[1]\mathsf{Antecedent} \sqcap \\
&\quad\ (\leq 1\,[1]\mathsf{Dependent}) \sqcap \exists[1]\mathsf{Dependent} \\
\mathsf{CIM\_HostedAccessPoint} &\sqsubseteq \forall[1](\mathsf{Antecedent} \Rightarrow (2 : \mathsf{CIM\_System})) \sqcap \\
&\quad\ \forall[1](\mathsf{Dependent} \Rightarrow (2 : \mathsf{CIM\_ServiceAccessPoint})) \sqcap \\
&\quad\ (\leq 1\,[1]\mathsf{Antecedent}) \sqcap \exists[1]\mathsf{Antecedent} \sqcap \\
&\quad\ (\leq 1\,[1]\mathsf{Dependent}) \sqcap \exists[1]\mathsf{Dependent} \\
\mathsf{CIM\_ServiceAccessBySAP} &\sqsubseteq \forall[1](\mathsf{Antecedent} \Rightarrow (2 : \mathsf{CIM\_Service})) \sqcap \\
&\quad\ \forall[1](\mathsf{Dependent} \Rightarrow (2 : \mathsf{CIM\_ServiceAccessPoint})) \sqcap \\
&\quad\ (\leq 1\,[1]\mathsf{Antecedent}) \sqcap \exists[1]\mathsf{Antecedent} \sqcap \\
&\quad\ (\leq 1\,[1]\mathsf{Dependent}) \sqcap \exists[1]\mathsf{Dependent}
\end{aligned}
$$

(**id** CIM_ElementSetting [1]Element, [1]Setting)
(**id** CIM_ElementConfiguration [1]Element, [1]Configuration)
(**id** CIM_Dependency [1]Antecedent, [1]Dependent)
(**id** CIM_Realizes [1]Antecedent, [1]Dependent)
(**id** CIM_HostedService [1]Antecedent, [1]Dependent)
(**id** CIM_HostedAccessPoint [1]Antecedent, [1]Dependent)
(**id** CIM_ServiceAccessBySAP [1]Antecedent, [1]Dependent)

5.4 Aggregations

$$\begin{aligned}
\mathsf{CIM_SettingContext} &\sqsubseteq (1:\mathsf{CIM_Configuration}) \sqcap (2:\mathsf{CIM_Setting}) \\
\mathsf{CIM_DependencyContext} &\sqsubseteq (1:\mathsf{CIM_Configuration}) \sqcap (2:\mathsf{CIM_Dependency}) \\
\mathsf{CIM_Component} &\sqsubseteq (1:\mathsf{CIM_ManagedSystemElement}) \sqcap \\
&\quad (2:\mathsf{CIM_ManagedSystemElement}) \\
\mathsf{CIM_SystemComponent} &\sqsubseteq (1:\mathsf{CIM_System}) \sqcap (2:\mathsf{CIM_ManagedSystemElement}) \\
\mathsf{CIM_ProductPhysicalElement} &\sqsubseteq (1:\mathsf{CIM_Product}) \sqcap (2:\mathsf{CIM_PhysicalElement})
\end{aligned}$$

5.5 Inclusion Relationships

In this example we have inclusion assertions only between classes; these relations are expressed by the followind $\mathcal{DLR}_{ifd}$ assertions:

$$\begin{aligned}
\mathsf{CIM_PhysicalElement} &\sqsubseteq \mathsf{CIM_ManagedSystemElement} \\
\mathsf{CIM_LogicalElement} &\sqsubseteq \mathsf{CIM_ManagedSystemElement} \\
\mathsf{CIM_System} &\sqsubseteq \mathsf{CIM_LogicalElement} \\
\mathsf{CIM_LogicalDevice} &\sqsubseteq \mathsf{CIM_LogicalElement} \\
\mathsf{CIM_Service} &\sqsubseteq \mathsf{CIM_LogicalElement} \\
\mathsf{CIM_ServiceAccessPoint} &\sqsubseteq \mathsf{CIM_LogicalElement}
\end{aligned}$$

6 Example: CIM Common Model

Extension of CIM Core Model are modelled *adding* assertions to the TBox which models the Core Model. As an example, we examine a fragment of the *Devices* common area of CIM Common Model, shown in Figure 14 (note that CIM_LogicalDevice is already present in CIM Common Model).

We therefore add a suitable set of assertions to the CIM Core Model TBox. We choose the arbitrary role names $\mathsf{r}_1, \mathsf{r}_2, \mathsf{r}_C, \mathsf{r}_D$ for the missing roles in associations CIM_DeviceConnection and CIM_DeviceConnection.

$$\begin{aligned}
\mathsf{CIM_DeviceConnection} &\sqsubseteq \forall[1](\mathsf{r}_1 \Rightarrow (2:\mathsf{CIM_LogicalDevice})) \sqcap \\
&\quad \forall[1](\mathsf{r}_2 \Rightarrow (2:\mathsf{CIM_LogicalDevice}))
\end{aligned}$$

$$\begin{aligned}
\mathsf{CIM_ControlledBy} &\sqsubseteq \forall[1](\mathsf{r}_C \Rightarrow (2:\mathsf{CIM_Controller})) \sqcap \\
&\quad \forall[1](\mathsf{r}_2 \Rightarrow (2:\mathsf{CIM_LogicalDevice}))
\end{aligned}$$

Finally, we add the inclusion assertion

$$\mathsf{CIM\_Controller} \sqsubseteq \mathsf{CIM\_CIMLogicalDevice}.$$

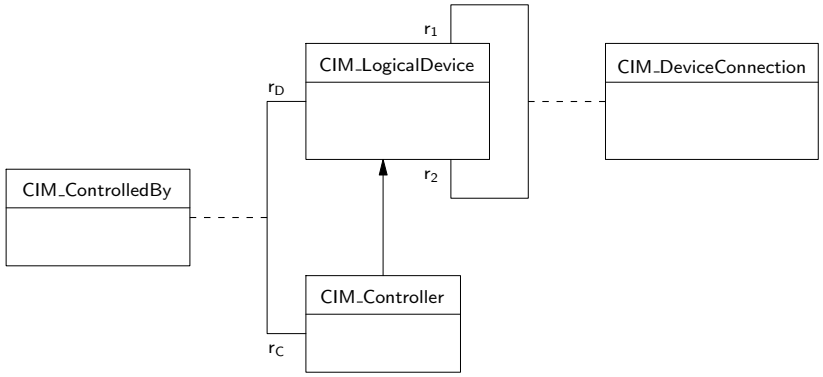

Figure 14: Fragment of *Devices* common area of CIM Common Model

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison Wesley Publ. Co., Reading, Massachussetts, 1995.

[2] Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. Structured objects: Modeling and reasoning. In *Proc. of the 4th Int. Conf. on Deductive and Object-Oriented Databases (DOOD'95)*, volume 1013 of *Lecture Notes in Computer Science*, pages 229–246. Springer-Verlag, 1995.

[3] Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. On the decidability of query containment under constraints. In *Proc. of the 17th ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS'98)*, pages 149–158, 1998.

[4] Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. Identification constraints and functional dependencies in description logics. Submitted for publication, 2001.

[5] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, Daniele Nardi, and Riccardo Rosati. Description logic framework for information integration. In *Proc. of the 6th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 2–13, 1998.

[6] Diego Calvanese, Maurizio Lenzerini, and Daniele Nardi. Unifying class-based representation formalisms. *J. of Artificial Intelligence Research*, 11:199–240, 1999.